Implementing Reverse Engineering

The Real Practice of X86 Internals, Code Calling Conventions, Ransomware Decryption, Application Cracking, Assembly Language, and Proven Cybersecurity Open Source Tools

JITENDER NARULA

Implementing
Reverse
Engineering

The Real Practice of x86 Internals, Code Calling Conventions, Ransomware Decryption, Application Cracking, Assembly Language, and Proven Cybersecurity Open Source Tools

---

Jitender Narula

www.bpbonline.com

FIRST EDITION 2022

ISBN: 978-93-91030-377

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

Dedicated to

My parents

Always seen God in them

About the Author

Jitender Narula is an experienced Cyber Security Specialist currently associated with International Institute of Cyber Security having over 18 years of industry experience. He has many years of cyber security experience with governments and corporate world. In India, he has worked with government entities like Delhi Police, ICAI (Institute of Chartered Accountants of India), Delhi University and other private organizations.

He has worked on the projects of AT&T, Citrix, Google, Conexant, IPolicy Networks (Tech Mahindra now), Narus (a wholly owned subsidiary of The Boeing Company) and HFCL. Apart from this, he has conducted training programs for various corporate and government officials in India and Mexico.

He has published articles, research information and interviews in the area of cyber security for Information Security Newspaper Noticias de seguridad informática Cibertip - Noticias de Hacking Exploit One iicybersecurity blog and also contributed to the Vishvas News, which is part of Dainik Jagran (Indian language daily newspaper).

About the Reviewer

Sanil Nadkarni is CISO /CRO/DPO in an MNC. He has worked for organizations and clients such as Microsoft, Symantec, MphasiS, Capita, SLK and has supported International Banks and Financial Organizations.

He is an international speaker, author and trainer. Sanil has published over 100 articles online and also for various magazines.

His certifications include CFE, CISSP, CISA, CISM, Security+, ISO 27001 LA, ISO 22301 LA, A+, N+, MCP, MCSA, CCNA, SCSA, ISO 10012 (GDPR), RHCE, CEH,CBCP, ISO 31000 Risk and Microsoft Azure Architect.

Sanil has received honors and awards such as CISO TOP 100 Award, Dynamic CISO Award, Top 100 CIO award and many more.

He has published a book on Fundamentals on Information Security which is available on Amazon.

Acknowledgement

First and foremost, praises and thanks to my Dad, Mom and God for showering blessings throughout my work to complete the book successfully.

I would like to express my deep and sincere gratitude to Atul Narula, my colleague at International Institute of Cyber Security, Mexico for helping with the review process of this book. I am extremely grateful for what he has offered me. I would also like to thank him for his friendship, empathy and great sense of humor.

I am extremely grateful to my parents (Ramesh Narula and Mohini Narula) for their love, prayers, care and sacrifices for educating and preparing me for my future. I am very much thankful to my wife and son for their love, understanding, prayers and continuing support to complete this book. Also, I would like to express my thanks to Dr. Shilpi Sahi and Om Narula for their support and motivation throughout this process of writing. Once again, I would like to thank my family for putting up with me while I was spending many nights writing—I could have never completed this book without their support.

Finally, I would like to thanks BPB Publications for giving me the opportunity to write my first book for them.

Preface

Reverse Engineering (RE) is an art of understanding any program code when no source code is available. This book provides step-by-step explanation of the essential concepts and practical examples to understand and implement Reverse Engineering. It will enable the readers to understand the application code flow to identify vulnerabilities and bugs in the application.

This book is for the readers who want to start learning Reverse Engineering from basics in a step-by-step manner. The book is divided into three parts:

Exploring Reverse Engineering

Reverse Engineering Applications

Real World Examples with Solutions

The first part Exploring Reverse Engineering starts with the basic concepts of Computing System and Data Building Blocks of the Computing System. This part also enlists open-source tools required for RE applications and the programming instructions of RE. The second part Reverse Engineering Applications walks us through the different applications/programs to understand the implementation of RE. This part covers various practicals, which give the users a hands-on experience. All the applications/programs mentioned in this part are aligned in a systematic manner; from reverse engineering of basic C/C++ programs to complex C/C++ programs. In the third part Real World Examples and Solutions of this book, RE of well-known Windows application along with different exercises are demonstrated in a step-by-step manner. Over the 18 chapters in this book, you will learn the following:

PART 1: Exploring Reverse Engineering

In this part, the readers will understand the impact of RE on industry, building blocks of x86 computing system and the role of each in the overall functioning of x86 system.

Chapter 1 talks about the impact of RE on IT industry and how it originated as an area.

Chapter 2 talks about the building blocks of a computing system and the role of each building block in the overall functioning of the system. This chapter is important in order to understand the core concept behind the working of x86 computing systems.

Chapter 3 focuses on the open-source tools used in RE and how these tools are used for debugging and analysis. These tools will be used in all illustrations shown in this book.

Chapter 4 explains about the major assembly instructions used and also about how different instructions are segmented in various sections for easy understanding along with examples.

Chapter 5 helps us understand the different calling conventions along with practical illustrations.

PART 2: Reverse Engineering Applications

This is where the strategic way of learning RE applications/programs is explained with different illustrations. Every case is the outcome of research explained in a very simplified and step-by-step manner.

Chapter 6 gives a step-by-step understanding of the assembly code generated from basic C/C++ program.

Chapter 7 provides a step-by-step understanding of the assembly code generated from printf() function in C/C++ program.

Chapter 8 gives a step-by-step understanding of the assembly code generated from pointers in C/C++ program.

Chapter 9 provides a step-by-step understanding of the assembly code generated from decision control structure in C/C++ program.

Chapter 10 gives a step-by-step understanding of the assembly code generated from loop control structure in C/C++ program.

PART 3: Real World Examples and Solutions

In this part, understanding of whatever learned in the previous chapters is explained with real world exercises with solutions and also reversing of Windows well-known application is demonstrated.

Chapter 11 covers RE exercise of an array code along with the solution used in the RE process.

Chapter 12 covers RE exercise of a structure code along with the solution used in the RE process.

Chapter 13 explains RE exercise of a Scanf program along with the solution used in the RE process.

Chapter 14 explains RE exercise of a strcpy program along with the solution used in the RE process.

Chapter 15 covers RE exercise of a simple interest code along with the solution used in the RE process.

Chapter 16 explains RE exercise of breaking Wannacry ransomware with Ghidra.

Chapter 17 covers RE exercise using Cutter tool.

Chapter 18 demonstrates the process of RE of Windows Calculator in a step-by-step manner.

This book is to educate the learners on the topic of Reverse Engineering on x86 platform. This will be a good book for beginners and computer graduates in the area of RE. Professionals who want to switch their career to RE can also use this book. Other readers can be from schools, universities or those who are passionate to get into the area of cyber security.

Downloading the code bundle and coloured images:

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/b2fe9c

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

BPB is searching for authors like you

If you're interested in becoming an author for BPB, please visit www.bpbonline.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at Check them out!

PIRACY

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit

REVIEWS

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Table of Contents

1. Impact of Reverse Engineering

Structure

Objective

Introduction to Reverse Engineering

Importance of Reverse Engineering

Studying an existing design

Redeveloping an outdated or lost product

Security auditing

Finding sensitive data

Military espionage

Finding product vulnerabilities

Bounty for cyber enthusiasts

The Role of Reverse Engineering

Conclusion

2. Understanding Architecture of x86 Machines

Structure

Objective

Architecture of a Computing System

CPU

Memory

Input/output Devices

System Bus

Building blocks of a Computing System

Microprocessor

Memory

Registers

General Purpose Register
Segment Registers
Status Registers
Instruction Pointer Register
Concept of Stack
Caller Before Callee Call
Callee After Function Call
Callee Before Returning
Caller After Returning
Conclusion

3. Up and Running with Reverse Engineering Tools
Structure
Objective
Importance of tools in reverse engineering
Reverse engineering tools
Portable Executable Editors
CFF Explorer
Disassembler
Ghidra
Cutter
Debuggers
x32dbg
Conclusion

4. Walk Through on Assembly Instructions
Structure
Objective
Different assembly language instructions
Stack Instructions

PUSH
PUSHAD
PUSHFD
POP
POPAD
POPFD
RET
Data Transfer Instructions
MOV
LEA
XCHG
CMPXCHG
LAHF
SAHF
LAR
MOVSX
MOVZX
XLAT
MOVS
Arithmetic Instructions
AAA
AAS
AAD
AAM
ADC
ADD
CMP
DAA
DAS

DEC

DIV
IDIV
MUL
IMUL
INC
NEG
SBB
SUB
XADD
Program_Execution_Instructions
CALL
ENTER
LEAVE
INT
INTO
IRET
LOOP
LOOPE
LOOPNE
TEST
Branching_Instructions
JMP
JZ
JNZ
JE
JNE
JG
JGE

JA
JAE
JL

JLE
JB
JBE
JO
JS
JECXZ
Bit Manipulation Instructions
BSWAP
AND
NOT
OR
XOR
RCL
RCR
ROL
ROR
SHR
SHL
SAR
SAL
SHLD
SHRD
Processor Control Instructions
CLC
CLD
CLI

CMC
ESC
LOCK
NOP
STC

STD
STI
String Instructions
CMPS/CMPSB/CMPSW
IN/INSB/INSW/INSD
OUT/OUTSB/OUTSW/OUTSD
LODS/LODSB/LODSW/LODSD
STOS/STOSB/STOSW
SCAS/SCASB/SCASW
MOVS/MOVSB/MOVSW
REP
REPE/REPZ
REPNE/REPNZ
Conclusion

5. Types of Code Calling Conventions
Structure
Objective
Understand types of calling conventions
CDECL
STDCALL
FASTCALL
Concept behind different calling conventions
CDECL
STDCALL

FASTCALL
Conclusion

6. Reverse Engineering Pattern of Basic Code
Structure

Objective
What is Code Optimization?
Empty function
Empty Function without Optimization
Empty Function with Optimization
Returning Value
Returning Value without Optimization
Returning Value with Optimization
Basic "Hello, World" Program
Basic "Hello, World" Program without Optimization
Basic "Hello, World" Program with Optimization
Conclusion

7. Reverse Engineering Pattern of Printf Program
Structure
Objective
Function printf with Integers
Function printf Printing Integers without Optimization
Function printf Printing Integers with Optimization
Function printf with Float
Function printf Printing Float without Optimization
Function printf Printing Float with Optimization
Function printf with char
Function printf Printing Char without Optimization
Function printf printing Char with Optimization

Conclusion

8. Reverse Engineering Pattern of Pointer Program
Structure
Objective

Pointers
Pointer without Optimization
Pointer with Optimization
Conclusion

9. Reverse Engineering Pattern of Decision Control Structure
Structure
Objective
If-else statement
If-else statement without Optimization
If-else statement with Optimization
Conclusion

10. Reverse Engineering Pattern of Loop Control Structure
Structure
Objective
While Condition
While condition without Optimization
While condition with Optimization
For Loop
For Loop without Optimization
For Loop with Optimization
Conclusion

11. Array Code Pattern in Reverse Engineering
Structure

Objective
Understanding an array
Array Loop without Optimization
Array Loop with Optimization

Conclusion

12. Structure Code Pattern in Reverse Engineering
Structure
Objective
Understanding of structures
Structure without Optimization
Structure with Optimization
Conclusion

13. Scanf Program Pattern in Reverse Engineering
Structure
Objective
Function scanf with Integers
Function scanf without Optimization
Function scanf with Optimization
Conclusion

14. Strcpy Program Pattern in Reverse Engineering
Structure
Objective
Strcpy
Strcpy without Optimization
Strcpy with Optimization
Conclusion

15. Simple Interest Code Pattern in Reverse Engineering

Structure
Objective
Program to Calculate Simple Interest

Calculate Simple Interest Without Optimization
Conclusion

16. Breaking Wannacry Ransomware With Reverse Engineering
Structure
Objective
Installation
Analyzing and Breaking Wannacry
Conclusion

17. Generate Pseudo Code From Binary File
Structure
Objective
Cutter Installation
Binary Analysis Using Cutter
Dashboard
Strings
Imports
Disassembly
Graph
Hexdump
Decompiler
Decrypting the Hidden URL
Conclusion

18. Fun With Windows Calculator Using Reverse Engineering
Structure

Objective
Reverse Engineering Calculator
Understanding the code flow with breakpoints

Finding a placeholder to call our code
Writing our code in the Code Cave
Patching the binary
Conclusion

Appendix

Macro
Procedure
npad
LSB and MSB
Signed and Unsigned
Unsigned
Signed
Bit Shifting
Logical bit shifting
Arithmetic bit shifting
ASCII
Unicode
Disable Address Space Layout Randomization

Index

CHAPTER 1

Impact of Reverse Engineering

Before we start on the implementation of reverse engineering, it will be interesting to understand what reverse engineering really is, how it came into existence, and how it is beneficial in the modern era. Reverse engineering, as the name suggests, is a combination of two words: Reverse and Engineering. Engineering is the science of designing and building something beneficial for the human race. Engineering has provided us with both advantages and disadvantages. Engineering equipped us with the knowledge and means to build essential things for the human race, including roads, buildings, bridges, cars, airplanes, software, and more. However, gradually, we also started using engineering to produce weapons of mass destruction like missiles, malware, and other deadly products harmful for humans and nature itself. When anything is engineered, it goes through many phases of design, development, and testing. With reverse engineering, things have really changed.

The concept behind reverse engineering is to break something to understand its internal architecture to build a copy or for the purpose of improvement or modification. In this chapter, we will talk about some real-life examples to understand the importance of reverse engineering and how it is changing the way the software industry works.

Structure

In this chapter, we will cover the following topics:

Introduction to Reverse Engineering

Importance of Reverse Engineering

The Role of Reverse Engineering

Objective

After studying this chapter, you should be able to understand the importance of reverse engineering and its impact on the software industry. We will also talk about the opportunities associated with reverse engineering and how malware writers are using it to exploit the software systems of big companies.

Introduction to Reverse Engineering

In software terms, **Reverse Engineering** is the art of understanding any program code when no source code is available. All of this started in the late 1980s when the **Disk Operating System** was in use. Most of us were not born at that time or might be in our childhood time. During that time, people used to play DOS-based video games. Most of the games were player-based video games, where the game player had a lifeline and is equipped with the weapons. This is where some group of computer geeks followed reverse engineering techniques to increase the lifeline of the game player and change the number of weapons a player could use. This was done by simply modifying the values at the memory location where the lifeline and the number of weapons of a player were stored. This might sound like cheating, but in reality, it was a way to breach the security of the video game.

To understand the importance of reverse engineering in the present times, we will take an example. Imagine that three people named Jitender, Shilpi, and Atul are working for a research and development organization, the International Institute of Cyber Security, having offices in India, Mexico, and the US. These three employees are working from three different geographical locations.

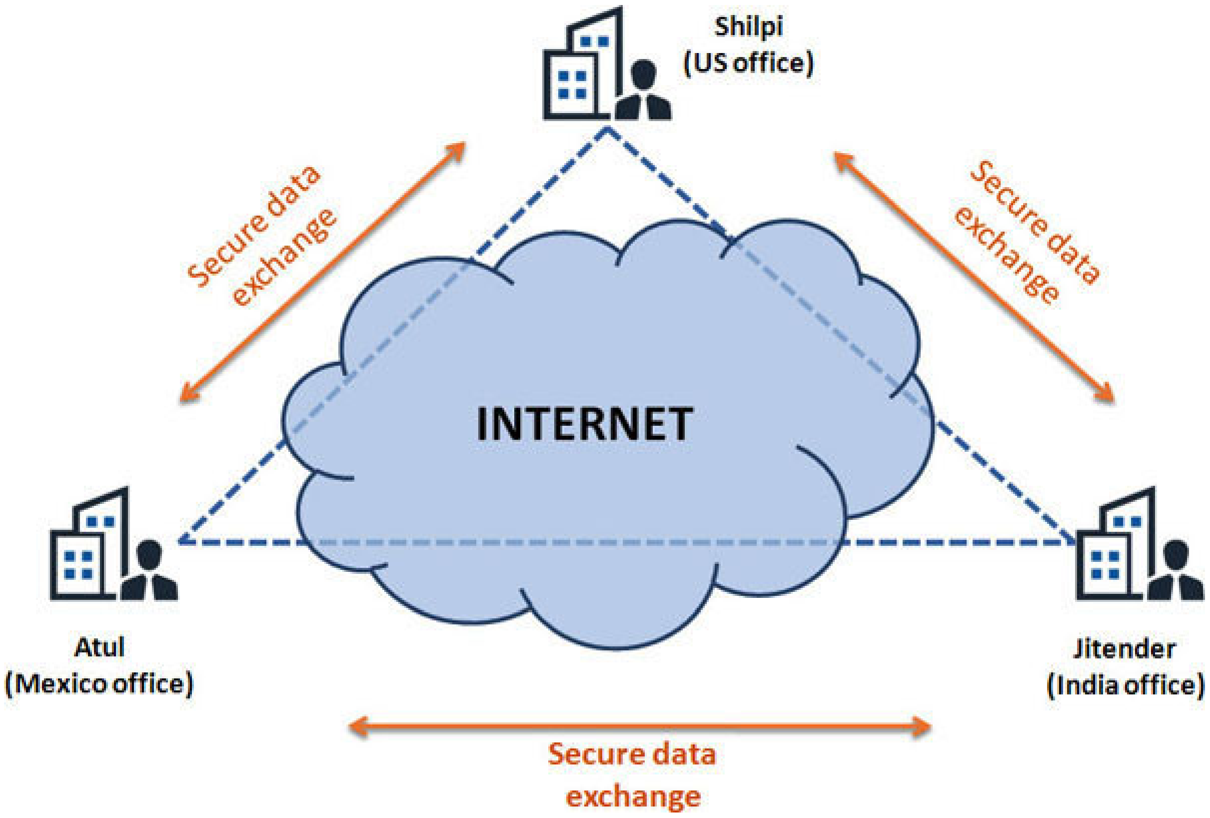


***Figure 1.1:** An example of reverse engineering*

They are all working on some research and development project and so they share their research findings over the internet. They use some secure software to share the data among themselves. As the data is very critical for the organization, the security of the software used to share this data should also be very secure. Now, this software can be open-source software or closed-source software. If the software they are using is open-source, then they can check the security of the software using code review. But what if the software is closed-source? They will not have an access to the source code of the software.

In this case, reverse engineering plays a big role in checking the security of closed-source software. With the help of reverse engineering, software security can be evaluated even if you do not have the source code available. It will also help in finding vulnerabilities in the software or application if any.

The process of reverse engineering was initially applied to computer applications and hardware but now, reverse engineering is applied everywhere, from software and machinery to even human DNA. Reverse engineering is important especially when you have closed-source software or software with malicious content.

Let us study another famous example of reverse engineering. A company named Phoenix Technologies, based out in San Jose, wanted to develop a BIOS compatible with IBM PCs. Rather than developing a self-designed BIOS, they took the IBM proprietary BIOS, reverse engineered it using the "clean room" or "Chinese wall" approach. Under this approach, they took two teams of engineers. The first team reverse engineered the IBM proprietary BIOS to recreate the design of the IBM proprietary BIOS. Everything was documented by the first team of engineers for the second team to work on. Once this design was recreated, the second team followed the documentation of the design specifications along with the functional requirements created by the first team to code the BIOS compatible with IBM PCs. The second team was totally ignorant about the reverse engineering work of the first team. The final product developed by Phoenix Technologies was sold to other PC manufacturers. The product developed by Phoenix Technologies was operationally identical but with no copyright infringement.

Moreover, other companies like Advanced Micro Devices also reverse engineered Intel corporation microprocessors to make less expensive chips. Reverse engineering is not only used for unethical purposes but also ethical purposes. One among them is malware analysis. As malware's are closed-source binaries, reverse engineering helps malware researchers decode malware functionality to break them.

To understand the real importance of reverse engineering, let's talk about a famous ransomware known as Wannacry ransomware. Ransomwares are the kind of malwares that, when installed in a victim's computer, encrypts the victim's files and demands a ransom to decrypt those files. If the victim does not pay the ransom within time, the victim's computer data may be deleted or the data may be left encrypted forever or there are chances that this data might be sold in the black market. Wannacry targeted Windows users by encrypting their data and then demanded a ransom to decrypt the data. To escape the law enforcement agencies, the ransom demanded in Bitcoin cryptocurrency. Bitcoin is a digital currency that is also known as cryptocurrency. It allows people to send and receive money on the internet without having to disclose the real identity of the sender or the receiver. With the efforts of a reverse engineer, Wannacry ransomware was made ineffective. We will study this in detail in *Chapter 16, Breaking Wannacry Ransomware With Reverse*

Importance of Reverse Engineering

Studying an existing design

Before designing anything, it is always a good approach to study the existing products available in the market. A good understanding of what a product does and how it works is important for new insights, but identifying where it can be improved can lead to several advantages.

Redeveloping an outdated or lost product

Every product in the market today is the outcome of hard work in terms of time and money. Imagine a situation where a company's product is in great demand in the market, but due to some unforeseen situation, the product is not getting any upgrades with time. This can be due to some internal reasons or the company that developed the product is no more in the market. With reverse engineering, such outdated products can be studied to recreate updated products.

Security auditing

Reverse engineering sometimes is a part of the security audit done for organizations. This is to check the security of software and the applications used within these organizations. It helps in finding unknown vulnerabilities running inside the organizations.

Finding sensitive data

Sensitive data encoded or encrypted in the software code can be extracted with the help of reverse engineering. This is done to validate the security posture of the software.

Military espionage

This is done to learn the strength of the opponent or enemy by capturing the high-level prototype of devices obtained by troops in the field and dismantling it to develop something new.

Finding product vulnerabilities

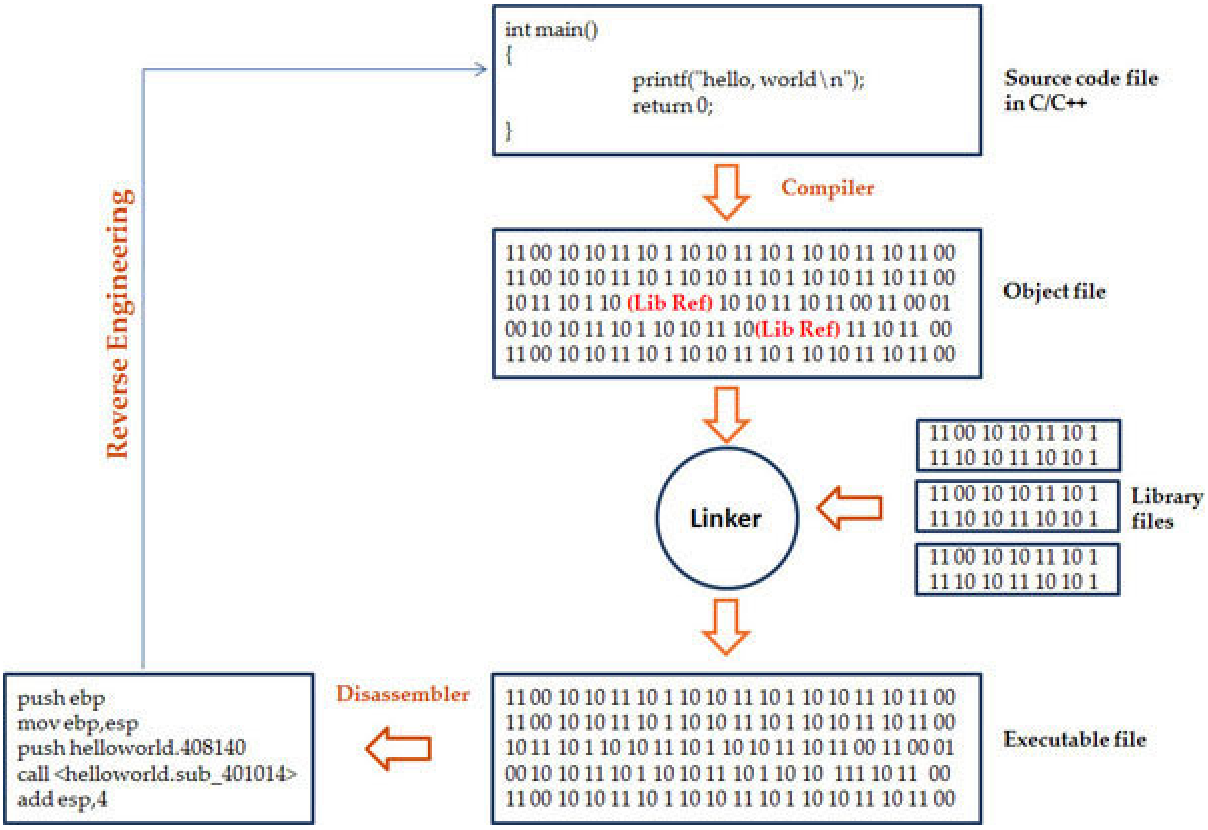
For the well-being and safety of the customers using a given product, reverse engineering is used to find defects or vulnerabilities in such a product. Every organization spends a substantial amount of time and money on efforts to find bugs or vulnerabilities in their products. But as it is well known, "nothing is secure". During the design, development, and testing, some bugs don’t get caught. This is where reverse engineering plays a vital role in aiding security researchers to uncover the issues that couldn’t be detected earlier.

Bounty for cyber enthusiasts

Earlier, product-based companies had an internal quality assurance team for security testing as well as functional testing for their products. But with time, everything changes. Cybersecurity requirements in the market changed drastically with an increase in cybersecurity attacks. Companies started offering security researchers a bounty to find vulnerabilities in their products. This helped both the security researchers in terms of money and the product companies in fixing uncaught bugs.

The Role of Reverse Engineering

Computer programs written in C/C++ are human-readable. When these programs are compiled using a compiler, an object file is created which is further passed through a linker to get a binary file or an executable file or, we can say, the ones and zeros of the machine language.



```
int main()
{
                printf("hello, world\n");
                return 0;
}
```

```
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
10 11 10 1 10 (Lib Ref) 10 10 11 10 11 00 11 00 01
00 10 10 11 10 1 10 10 11 10(Lib Ref) 11 10 11  00
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
```

```
11 00 10 10 11 10 1
11 10 10 11 10 10 1
```

```
11 00 10 10 11 10 1
11 10 10 11 10 10 1
```

```
11 00 10 10 11 10 1
11 10 10 11 10 10 1
```

```
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
10 11 10 1 10 10 11 10 1 10 10 11 10 11 00 11 00 01
00 10 10 11 10 1 10 10 11 10 1 10 10  111 10 11  00
11 00 10 10 11 10 1 10 10 11 10 1 10 10 11 10 11 00
```

```
push ebp
mov ebp,esp
push helloworld.408140
call <helloworld.sub_401014>
add esp,4
```

Figure 1.2: *The role of reverse engineering*

The ones and zeros are not human-readable. To convert the machine code back to a human-readable format, a tool called the decompiler is used. The role of a decompiler is to convert binary code into a human readable format and regenerate the code out of it. We will talk about such tools in *Chapter 3, Up and Running with Reverse Engineering Tools*

Conclusion

In this chapter, we learned how reverse engineering all began and how it is playing a big role in today’s era. We also studied the importance of reverse engineering and its impact on the software industry. We discussed opportunities associated with reverse engineering and how malware writers are using it to exploit the software of big companies. In the next chapter, we will study the internals of a computing system in terms of reverse engineering.

CHAPTER 2

Understanding Architecture of x86 Machines

In the future, every device or machine will become ‘smart’. The big difference between a normal device (or ‘the legacy device’, as we call it) and a smart device is the presence of the internet feature in a smart device. By smart, it means that the device is programmed to function in a smart fashion and it can be operated remotely using the internet feature. Today, most of the devices we use in our households are internet enabled or we can say, smart devices. Televisions are now smart televisions, washing machines are now smart washing machines, refrigerators we use are also now smart refrigerators, and many more. All this became possible with the introduction of a small computer in the legacy devices like televisions, washing machines, refrigerators, and others. Now a big question is, what’s inside these small computers and how do they work? These small computers are made up of small components, where every component plays an important role in the functioning of the overall system. Imagine that these small computers are a smaller version of your personal computer.

All these devices are addressed as modern computing devices. These computing devices are made up of several components for processing, data storage, data transfer, and more. Modern computing devices coupled with software are programmed to do many tasks. To understand **Reverse Engineering** on modern computing devices, we need to first understand what goes inside these computing devices and how they work.

Structure

In this chapter, we will cover the following topics:

Architecture of a Computing System

Building Blocks of a Computing System

History of the Different Types of Processors

Registers, Types of Registers and their Roles

Concept of Stack

Objective

In this chapter, we will talk about computing systems and their types. We will also talk about the components of modern computing systems. Then we will cover the topics of processors and the difference between processor variants along with their numbering scheme. We will also take a look at the role of stack in reverse engineering to understand the difference between caller and callee.

Architecture of a Computing System

Any computing system we see around is made up of some basic building blocks. When we say computing system, it can be your computer, laptop, mobile, IoT devices, and other devices which are capable of performing tasks. Basically, there are two types of computing systems:

Fixed Program Computing These systems are architected to perform a specific task. For example, a calculator.

Stored Program Computing On the other hand, these systems are architected in such a way that they can be programmed as per the requirements. They can run many tasks simultaneously and we can store and run applications on them. For example, a computer. The architecture of these systems was introduced by John von Neumann in 1945.

The von Neumann architecture is based on the stored program concept, where program data and instruction data are stored in the same memory. This design is used by modern computing systems, which are made up of the following building blocks:

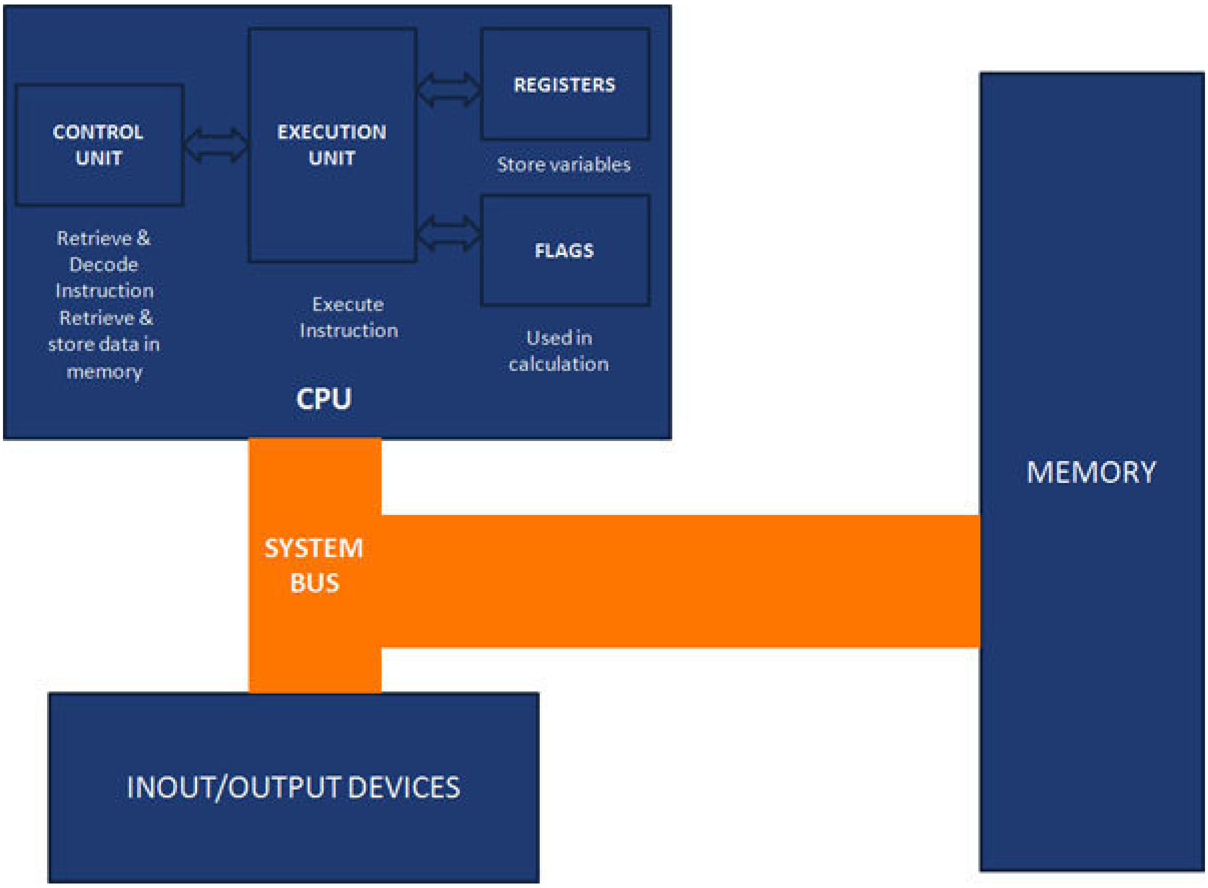


Figure 2.1: *Architecture of a Computing System*

CPU

The Central Processing Unit controls the operations of our computing device or system. In our computing system, the CPU is also referred to as processor, which is the brain of our computing system. The job of the CPU is to fetch instructions from the memory, decode the instructions into a series of actions, and carry out these steps in a sequence. Inside the CPU, we have several components. Some of them are:

Control This is responsible for retrieving and decoding instructions from the memory or RAM.

Execution This unit is responsible for the execution of instructions with the help of registers.

To save time, the CPU does not access RAM every single time to fetch instructions. So CPU has in itself basic storage units called registers. There are many types of registers, which we will study in the following sections. One among them is Instruction Pointer register, which stores the memory address of the next instruction to be executed.

These are registers only, but they record the state of CPU after arithmetic calculations.

Memory

This can be **Random Access Memory** or **Read Only Memory** It can also be an external storage device such as **Hard Disk** optical disk, and others. The primary purpose of memory is to store the sequence of instructions that our computer or computing system executes. This is also called program code. The second purpose of memory is to store data, on which our computer works.

Input/output Devices

All the devices which are interfaced with our computing system are called I/O (Input/Output) devices. This can be our keyboard, mouse, monitor, and others. These devices are interfaced using ports and there are two types of ports, Input & Output ports. Input ports are used for reading data from these peripheral devices into the computing system. Output ports are used to send data from the computing system to the peripheral devices such as video display, printer, and others.

System Bus

The System Bus can be imagined as a group of wires that carry information or data between different components in our computing system. Depending on the type of information carried between the components, buses are classified as Address Bus, Data Bus, and Control Bus.

Address These are parallel signal lines which are used to send out the address of the memory location that is to be read from or written to. The number of memory locations that a CPU can address is calculated by the number of signal lines or address lines. Suppose a CPU has N address lines, so the total number of memory locations the CPU can address is For example, a CPU has 8 address lines. This CPU can address 256 memory locations. If a CPU has 16 address lines, then the CPU can address 65,536 memory locations.

Data These are also parallel signal lines which are used to transfer data between the CPU and memory.

Control A Control Bus contains parallel signal lines carrying synchronizing signals to control various peripheral devices connected to the CPU. These are used to transfer information required to coordinate multiple tasks. This consists of 4-10 parallel signal lines to send out signals on the control bus. Typical control bus signals are I/O Read, I/O Write, Memory Read, and Memory Write. Suppose a CPU needs to read a byte of data from the memory location. In this process, the following activities will happen:

The CPU will send the memory address of the desired byte on the Address Bus.

The CPU will then send the Memory Read signal on the Control Bus.

The Memory Read signal will enable the addressed memory device to output data (or byte) on to the Data Bus.

The Data (or byte) travels from the desired memory address to the CPU using the Data Bus.

Building blocks of a Computing System

To understand reverse engineering, knowledge of the basic data building blocks is a must. These data building blocks include the meaning of Bit, Nibble, Byte, Word, and DWORD. All of these can be explained from the following figure:

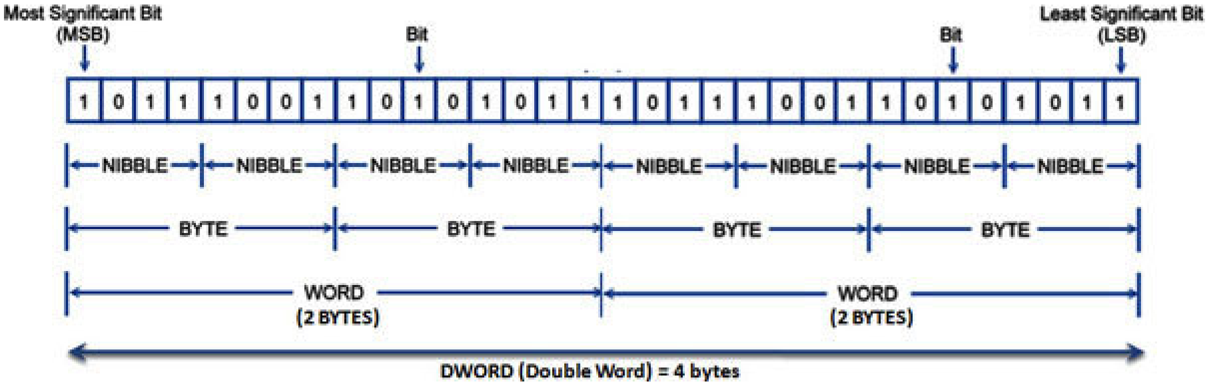


***Figure* 2.2:** *Understanding Bit, Nibble, Byte*

Humans can communicate with each other in different languages based on the countries they reside in. But when we talk about a computing system like computers, they can only understand binary, which is 0 or 1. Computers communicate with each other by sending or exchanging data. The smallest unit of data is called bit, which can be 0 or 1.

1 Nibble means 4 bits. Similarly, we can refer to BYTE, WORD, and DWORD as:

1 BYTE = 2 NIBBLES = 8 bits

1 WORD = 2 BYTES = 16 bits

1 DWORD = 4 BYTES = 32 bits

Microprocessor

As we know, the CPU is the brain of a computing system. The CPU is surrounded by circuitry which in its whole is referred to as the microprocessor. A microprocessor can have more than one CPU, like graphics processor. So, the CPU is actually a part of the microprocessor, but microprocessors can have more than one CPU. There are many types of microprocessors. You must have heard of companies like Intel, AMD, and many more. They are the top manufacturers of microprocessors. Some of the most popular models of the first generation microprocessors are:

are:
are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:

are:
are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:

are:

are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:

are:

are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:

are:

are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are: are:

So collectively, all the processors are referred to as the x86 Intel family.

8086, 80186, 80286, 80386, 80486 -> x86

Generally, we refer to the Intel processor as follows:

It means a 16-bit processor.

x86-32 (aka It means a 32-bit processor (IA means: Intel Architecture), also referred to as x86 only.

It means a 64-bit processor, also referred to as x64.

Note: Throughout this book, we will focus on the Intel x86-32 processor.

Memory

The memory, which we call RAM, for a single process running on the x86-32 architecture is divided into the following sections:

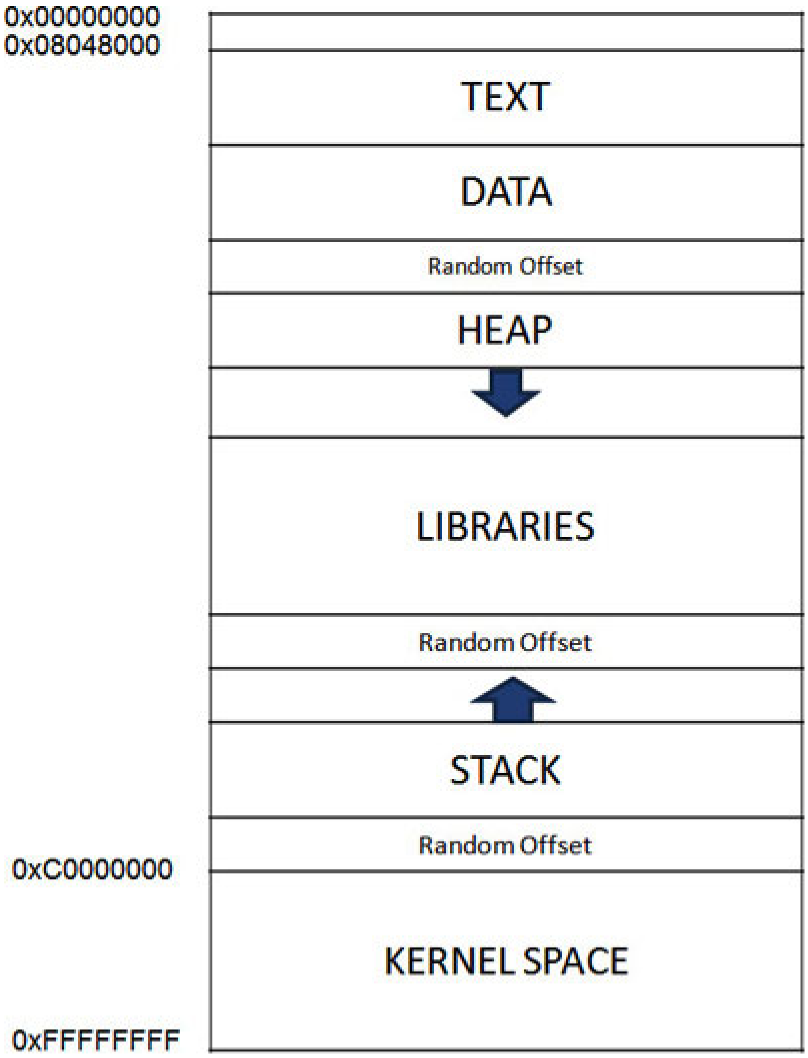


Figure 2.3: *Process Address Space*

The memory address is ranged from **0x00000000** – The prefix 0x refers to hexadecimal numbers. Every hexadecimal number is 4 bit in size, so any memory address of x86-32 architecture is referred by a combination of 8 hexadecimal numbers, which make 4 x 8 = 32 bits in size. This is why, the memory address of x86-32 computer is 32 bits in size.

Kernel 1GB is reserved for the Operating System kernel.

This is the space reserved for the function local variables and parameters. A stack grows up to a fixed memory size. It grows from a higher memory address to a lower memory address.

This is where our Shared Libraries are loaded. The common dialog box like *save dialog box* is stored in library which is shared among many programs.

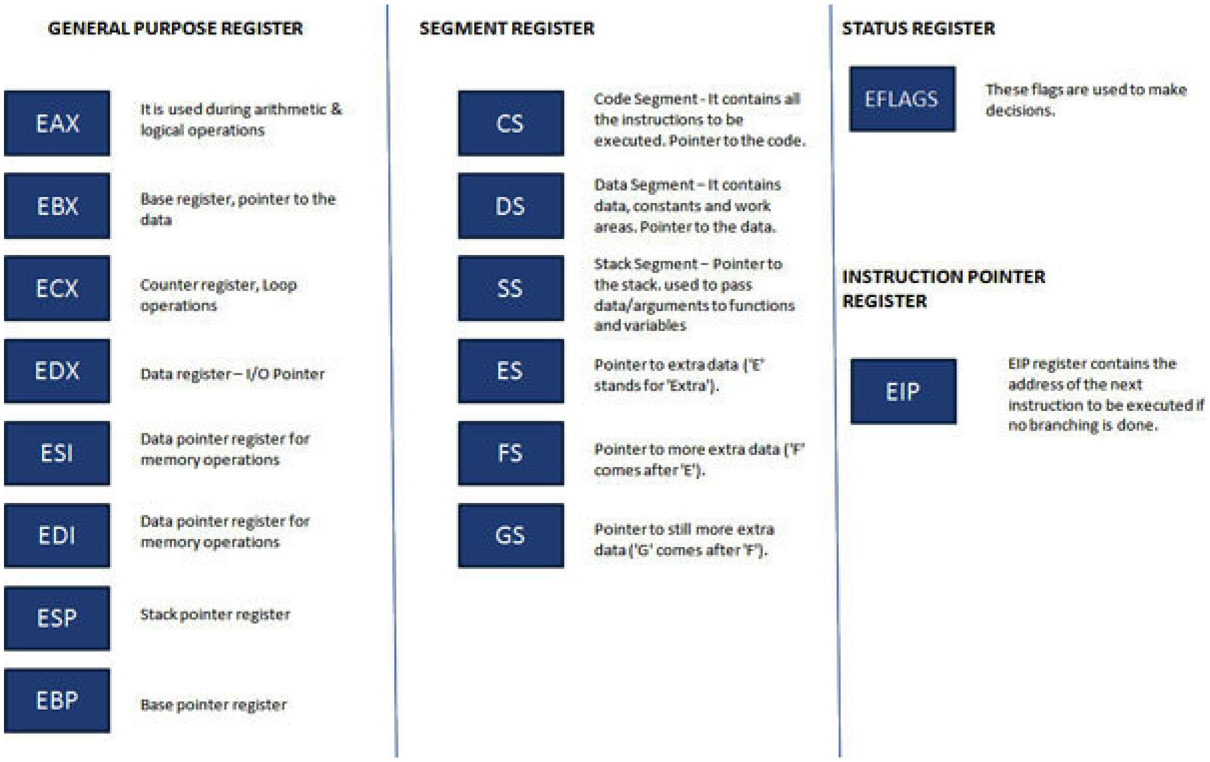
Heap grows down. When an image is loaded, depending on the size of the image, dynamic memory is required to load an image during the program execution. This memory is freed when the program finishes. This heap memory dynamically changes during program execution. It grows from a lower memory address to a higher memory address.

This is the section of memory used to store static variables and global variables in the code.

This section of memory holds the instruction or code to be executed to perform some action.

Registers

To save time, a small amount of temporary storage is available with the CPU called In the x86 processor, registers are divided into the following categories:



GENERAL PURPOSE REGISTER

EAX	It is used during arithmetic & logical operations
EBX	Base register, pointer to the data
ECX	Counter register, Loop operations
EDX	Data register – I/O Pointer
ESI	Data pointer register for memory operations
EDI	Data pointer register for memory operations
ESP	Stack pointer register
EBP	Base pointer register

SEGMENT REGISTER

CS	Code Segment - It contains all the instructions to be executed. Pointer to the code.
DS	Data Segment – It contains data, constants and work areas. Pointer to the data.
SS	Stack Segment – Pointer to the stack, used to pass data/arguments to functions and variables
ES	Pointer to extra data ('E' stands for 'Extra').
FS	Pointer to more extra data ('F' comes after 'E').
GS	Pointer to still more extra data ('G' comes after 'F').

STATUS REGISTER

EFLAGS	These flags are used to make decisions.

INSTRUCTION POINTER REGISTER

EIP	EIP register contains the address of the next instruction to be executed if no branching is done.

Figure 2.4: *Registers*

General Purpose Register

X86 architecture has 8 general purpose registers:

EAX is used for arithmetic and logical operations. It is also used to store the function return value.

EBX is used as a pointer to data.

ECX is used for loop operations.

EDX is used for I/O and arithmetic operations.

ESI is used as a pointer to the source in string operations.

EDI is used as a pointer to the destination in string operations.

ESP is a pointer to the top of the stack.

EBP is a pointer to the base of the stack frame.

All general purpose registers are 32 bits in size and they can also be referred in the sizes of 16 bits and 8 bits. The smaller unit of any register can be referred as shown below.

The size of the EAX is 4 byte (32 bits). The “E” in **EAX** stands for Extended.

Figure 2.5: *Smaller unit of register*

AX is the lower half of the EAX register, which is of size 16 bits. AX is further divided into AH (A-High) and AL (A-Low), each of 8 bits in size. The same goes for other general purpose registers.

32-bit version of general purpose registers: EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP

16-bit version of general purpose registers: AX, BX, CX, DX, SI, DI, BP, SP

8-bit version of general purpose registers: AH, AL, BH, BL, CH, CL, DH, DL

Segment Registers

The segment registers CS, DS, SS, ES, FS and GS are 16 bits in size. A segment register points at the start of a segment in memory. Segments can be categorized based on the three types of storage: Code, Data, and Stack.

Code This segment contains all the instructions to be executed and the **Code Segment** register contains the pointer to the code.

Stack This segment holds the data, variables, and arguments of the functions. **Stack Segment** register holds the pointer to the stack.

Data For code efficiency and security, four separate Data Segments are created. These are:

One for data structure of current loaded module

Data exported from other third-party modules

Dynamically created data structures

Data which is shared among different programs

To access the different types of data structures and additional data segments, DS, ES, FS, GS registers are used.

Status Registers

The status register is EEFLAGS register. This register is also 4 byte in size. Each bit of this register represents some flag, which can be either zero (0) or one (1). The status of each bit represents some result of the CPU operation. Some of the common flags used while performing reverse engineering are:

Zero Flag This flag is set to 1 when the result of the operation is zero. Otherwise, it is cleared.

Carry Flag This flag is set to 1 when the result of the operation is either too large or too small for the destination. Otherwise, it is cleared.

Sign Flag This flag is set to 1 when the result of the operation is negative. The flag is cleared, or we can say 0, when the result of the operation is a positive value.

Trap Flag This flag is set to 1 when the processor executes one instruction at a time and it is used for debugging.

Parity Flag It is set to 1 when **Least Significant Bit** of the result contains an even number. Otherwise, it is cleared.

Overflow Flag This flag is set to 1 when the result of the operation is too large to fit.

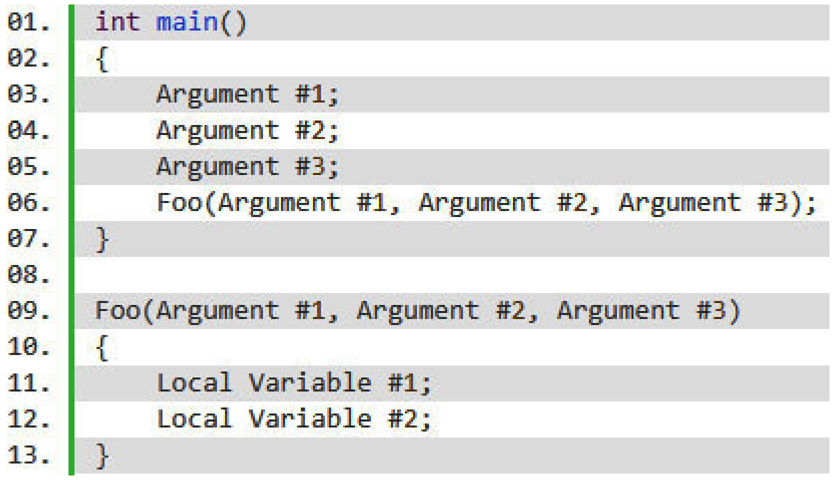
Instruction Pointer Register

Instruction Pointer is also known as EIP. This register contains the memory address of the next instruction to be executed. EIP tells the processor what to do next. From the security point of view, EIP is very important. An attacker compromises any system by taking control over the instruction pointer. Once the EIP is in control of the attacker, a malware code can be executed to perform any task.

Concept of Stack

This is the most important concept in Reverse Engineering. Imagine a stack as a pile of lunch plates lying one above the other at your local restaurant or cafeteria. To pick up a plate, we take out the plate from the top and the same process is followed to add more plates to the pile. The plate added to the pile will be added on the top. If we have to take out the plate at the bottom of the pile, we will have to take out every plate above it, one by one. For the time being, assume that adding a plate to the pile is called **PUSH** and taking out a plate from the pile is called

A stack works the same way, **Last In First Out** This means that the last thing added (pushed) will be the first thing to get pulled off (popped). To understand the working of a stack, we will take a pseudo code:



```
int main()
{
    Argument #1;
    Argument #2;
    Argument #3;
    Foo(Argument #1, Argument #2, Argument #3);
}

Foo(Argument #1, Argument #2, Argument #3)
{
    Local Variable #1;
    Local Variable #2;
}
```

Figure 2.6: *Pseudo code*

In this pseudo code, there are two functions: one is the **main** function from where our code execution starts and another is the **Foo** function. **Foo** function has 3 arguments which are local to **main** function and 2 local variables. During the code execution, the **Foo** function is called from the **main** function. In this code, the **main** function is the Caller and the **Foo** function is the Callee.

We are assuming that the 3 arguments and 2 local variables mentioned in the code are of type The data type of **int** occupies 4 bytes in the memory, as **sizeof(int)** is 4 bytes.

When our execution reaches inside the **Foo** function, the typical stack state will look like something as follows:

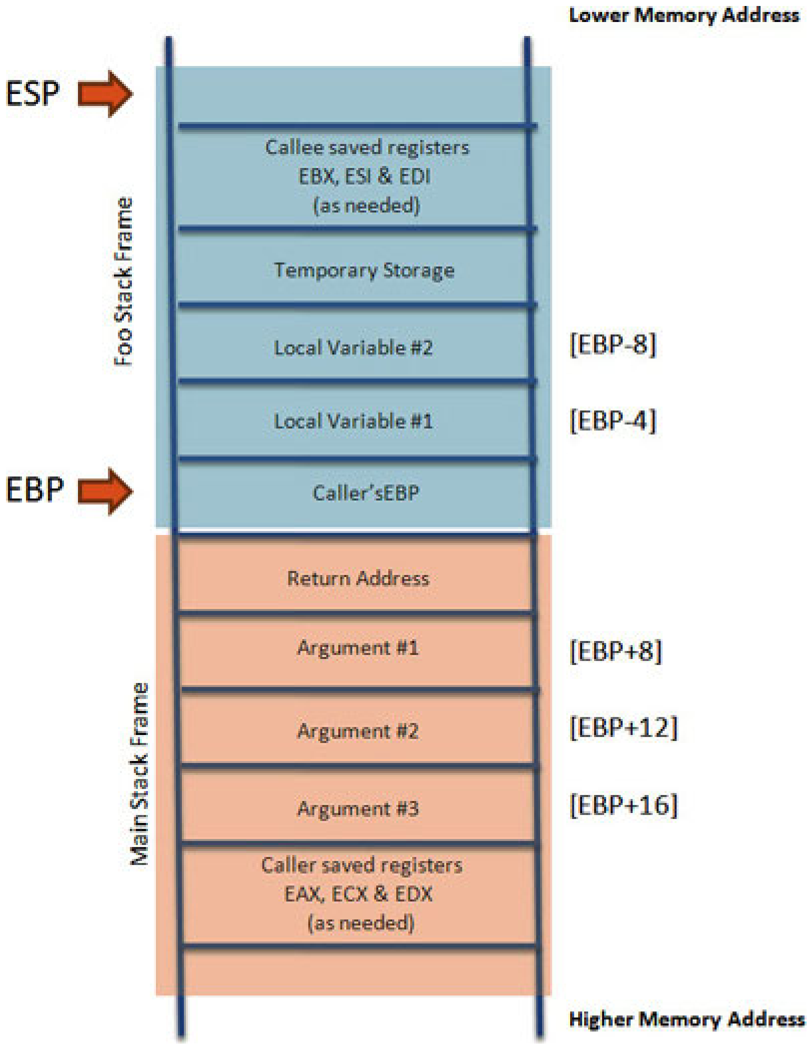


***Figure* 2.7:** *Stack*

Every function has a stack frame. As we have two functions **main** and we will have two stack frames. The highest address of the stack frame is the EBP of that stack frame.

Let's go step by step to understand how a stack frame is set up and cleaned upon function return.

Caller Before Callee Call

In this section, we will look at the stack state when the **main** function is just about to call the **Foo** function. Consider the **Foo** function call as follows:

```
int main()
{
FooReturnValue = Foo(10, 20, 30);
}
```

Caller is our **main** function and is about to call the **Foo** function which is Callee. The **main** function has its own stack frame, where the ESP is pointing to the top of the stack and EBP is a base pointer of the **main** function's stack frame.

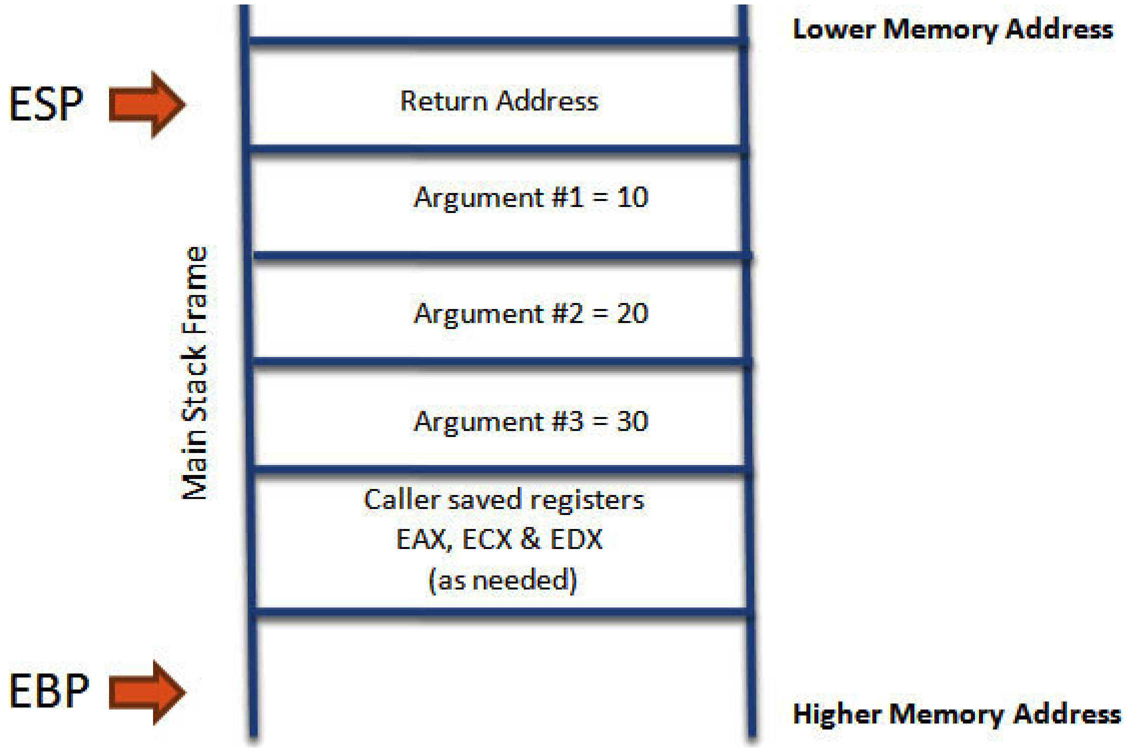


Figure 2.8: *Caller Before Callee Call*

Before the **Foo** function is called, the **main** function pushes the EAX, ECX and EDX registers onto the stack, only if the content of these registers needs to be preserved. Next, the **main** function pushes the **Foo** function arguments one by one onto the stack.

FooReturnValue = Foo(10, 20, 30); //Foo function arguments are 10, 20, 30

Arguments are pushed from the right to left order, so first *30* then *20* and then *10* is pushed onto the stack. The assembly instructions for the same are:

```
PUSH 30
PUSH 20
```

```
PUSH 10
```

After pushing the **Foo** arguments on the stack, a call to **Foo** function is made using the **CALL** instruction in the assembly language:

```
CALL Foo
```

When the **CALL** instruction is executed, the content of the EIP register is pushed onto the stack as EIP points to the next instruction in the **main** after the **CALL** instruction. After pushing the EIP onto the stack, we will have the return address on the top of the stack. This return address will help the instruction pointer to resume execution in the **main** once the execution of the **Foo** function is over.

Callee After Function Call

When the **Foo** function gets the control, it will perform the following three tasks:

Set up the **Foo** function stack frame.

Allocate space for the **Foo** function local variables.

If required, preserve the contents of EBX, ESI and EDI.

The **Foo** function stack frame is set up using the following assembly instructions:

```
PUSH EBP
MOV EBP, ESP
```

To set up the stack frame for the **Foo** function, we will first preserve the **main's** EBP (which is the base pointer of the **main** function) by pushing it onto the stack.

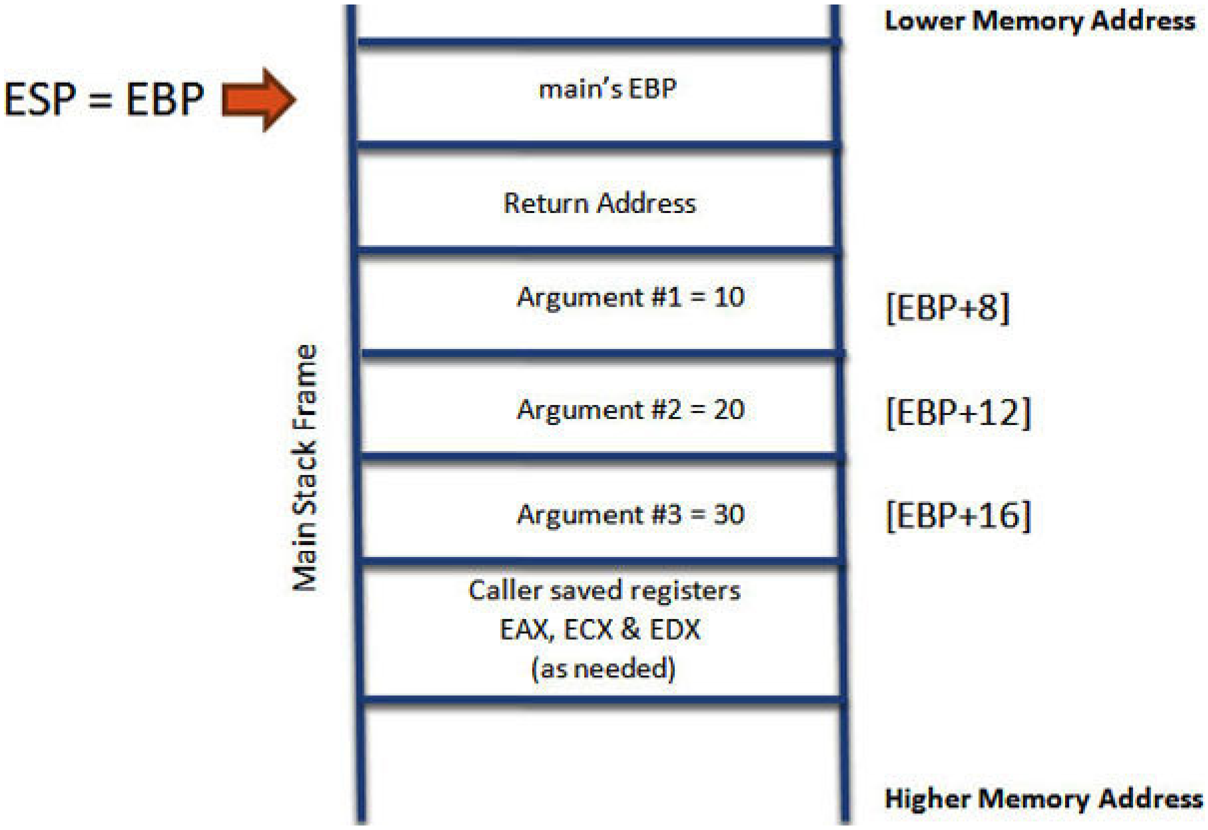


Figure 2.9: *Callee after function call*

After pushing the **main's** EBP on the stack, ESP (which is pointing to the top of the stack) will become the new EBP (base pointer of the **Foo** function). This EBP, along with the offset, will be used to refer to the variables on the stack. As we can notice in the preceding figure, the first argument can be accessed using EBP plus 8 bytes (4 bytes for **main's** EBP + 4 bytes for the Return Address).

Now we need to allocate space for the **Foo** function local variables onto the stack. This is done by subtracting 20 bytes from the stack pointer, where 20 bytes include 8 bytes (4 bytes + 4 bytes space for 2 variables) and 12 bytes for temporary storage. This will be done using:

SUB ESP, 20

Local variables and temporary variables can be accessed using the offset from EBP. After allocating room for the local variables on the stack, EBX, ESI and EDI registers are preserved by pushing on the stack.

The stack state after preserving the contents of **ESI** and **EDI** will be as follows:

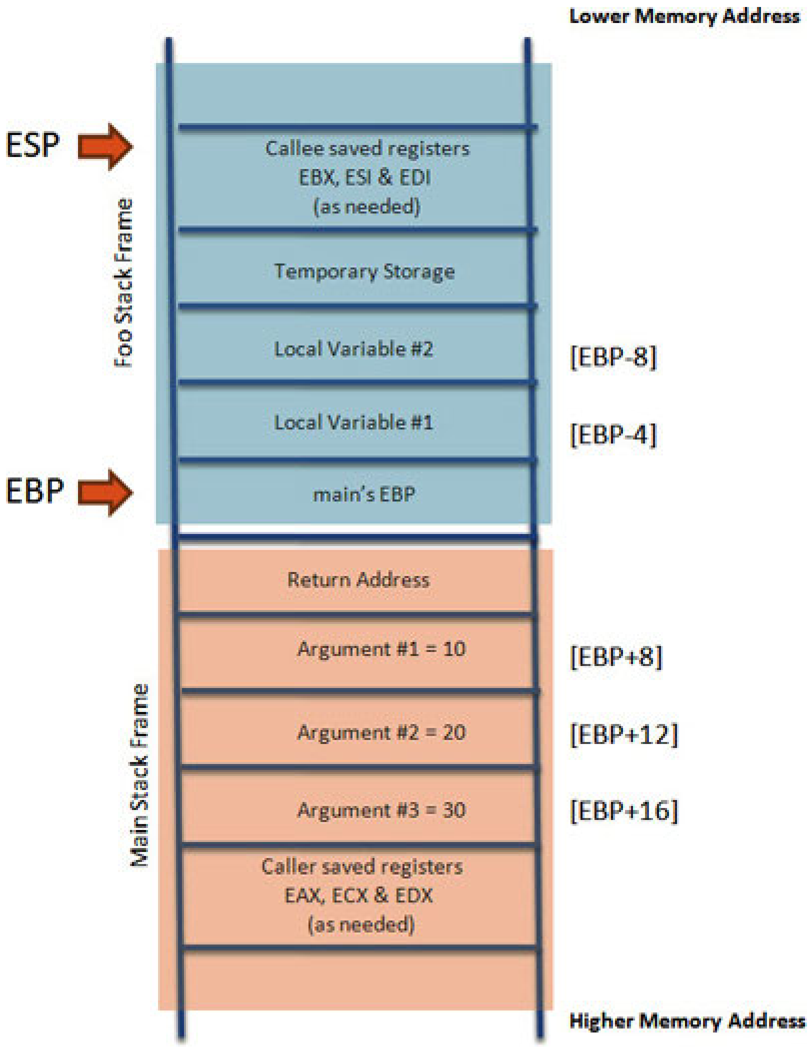


Figure 2.10: *Stack Frames of Foo and Main*

During the **Foo** function execution, there can be many pushing and popping on the stack. In this case, **ESP** will move up and down, but the **EBP** will remain unchanged. With the help of variables can be accessed using the offset from

Callee Before Returning

Now suppose we are done with the **Foo** function execution. Before returning, the callee will make arrangements to save the **Foo** function return value in the EAX register. Secondly, the values of EBX, EDI and ESI are restored back. The **Foo** function stack frame will be taken down by the following assembly instructions:

```
MOV ESP, EBP
POP EBP
RET
```

These instructions will bring the stack to the following state by moving the stack pointer ESP back to the EBP (base pointer of the **Foo** function) and the old EBP EBP) is restored by popping EBP from the stack.

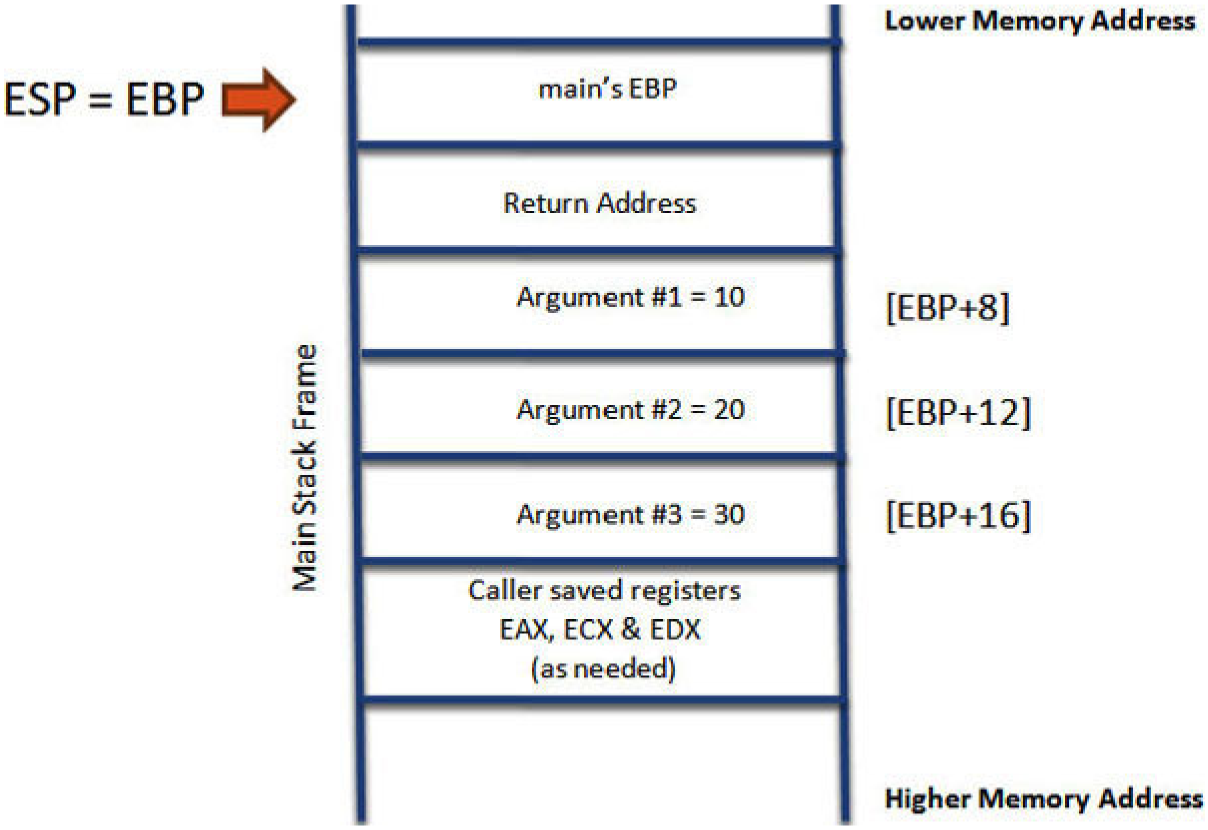


Figure 2.11: *Callee before returning*

RET, return instruction will pop return address from the stack and move it to EIP. This will point the **Instruction pointer** back to the **main** function, to resume execution of **main** function.

Caller After Returning

As the instruction pointer is back in the **main** function, now we do not require arguments on the stack anymore.

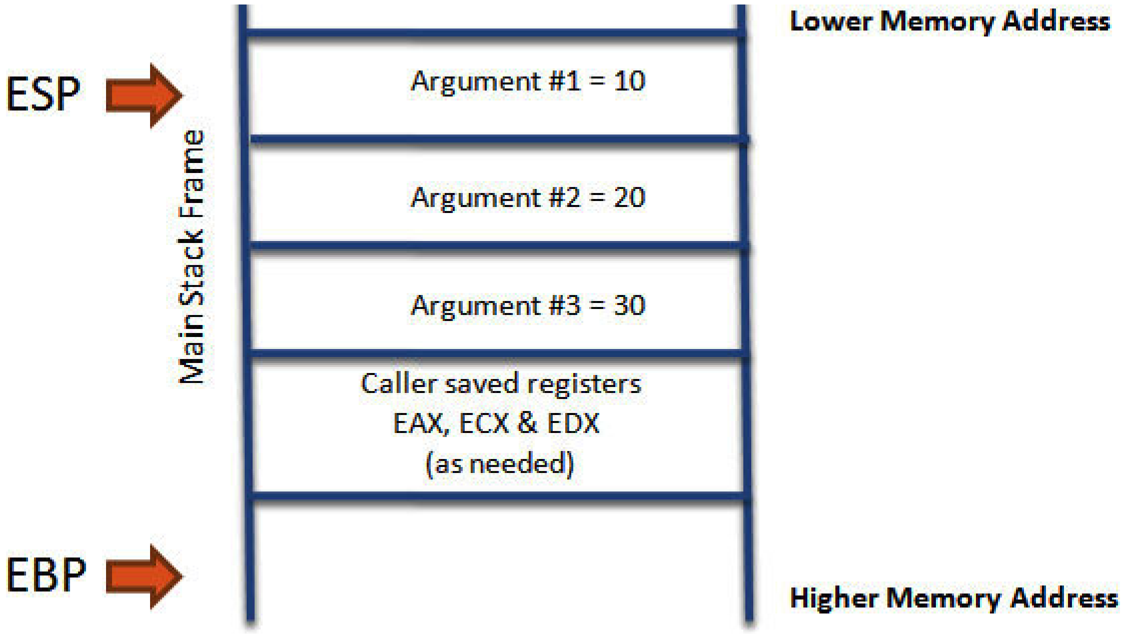


Figure 2.12: *Caller after returning*

All the arguments on the stack are cleaned by adding **ESP** by 12 bytes (4 bytes for each argument and multiplied by the number of arguments, which is 3).

```
ADD ESP, 12
```

As the EAX register is holding the return value of the **Foo** function, the content of the EAX register could be moved to some other register. Finally, the preserved registers and **EDX** are popped from the stack and ESP is pointed back to the same location on the stack where it started.

Conclusion

We covered the architecture of modern computing devices and the different components that make up a computing device. We also covered the functioning of different components used in a computing device. This covered the basics about microprocessors, memory, and the different types of registers. The different processor variants were also discussed along with the relevance of the x86 numbering convention.

We covered an important aspect of stack. With the help of pseudo code, we understood the difference between caller and callee. Understanding of stack plays an important role in the reverse engineering of any application. So don’t skip the stack part. In the next chapter, we will talk about the different reverse engineering tools used by professionals in the industry.

CHAPTER 3

Up and Running with Reverse Engineering Tools

Tools play a vital role in every aspect of life. We often use mobile calculators to perform basic math in our day-to-day life. Before mobile calculators, we used hardware calculator to perform math calculations. This calculator is a simple example of a tool we use to perform certain tasks. Simple calculations can be done verbally but when it comes to complex ones, it becomes necessary to use a tool. Similarly, for reverse engineering there are plenty of tools available in the market. Some are commercial and some are free to use or open source. For selection of the correct tools, conceptual knowledge of the topic becomes essential.

If you search the internet for reverse engineering tools, you will find several. It is always important to have the right selection of tools based on your requirements. In this chapter, we will first learn about the concept of tools in reverse engineering and then understand the importance of these tools in the process of reverse engineering. We will talk about the tools that are easily and freely accessible.

Structure

In this chapter, we will cover the following topics:

Importance of tools in reverse engineering

Reverse engineering tools

Portable executable editors

Disassemblers

Debuggers

Objective

The objective of this chapter is to understand the basic concept behind tools and why they are required in the process of reverse engineering. Then we will talk about some tools required to read the binary format. We will also cover some concepts related to **Portable Executable (PE)** editors, disassemblers, and debuggers.

Importance of tools in reverse engineering

When we compile a program using a high-level language like C/C++, it gets converted to a series of bytes that a CPU can understand. This is a machine code, which is difficult to understand by humans. To make this code easier to understand, we use a tool called disassembler. This disassembler translates machine code to human-readable format. This format is called assembly code which uses the syntax of assembly language. The following figure will help you understand the concept throughout your life:

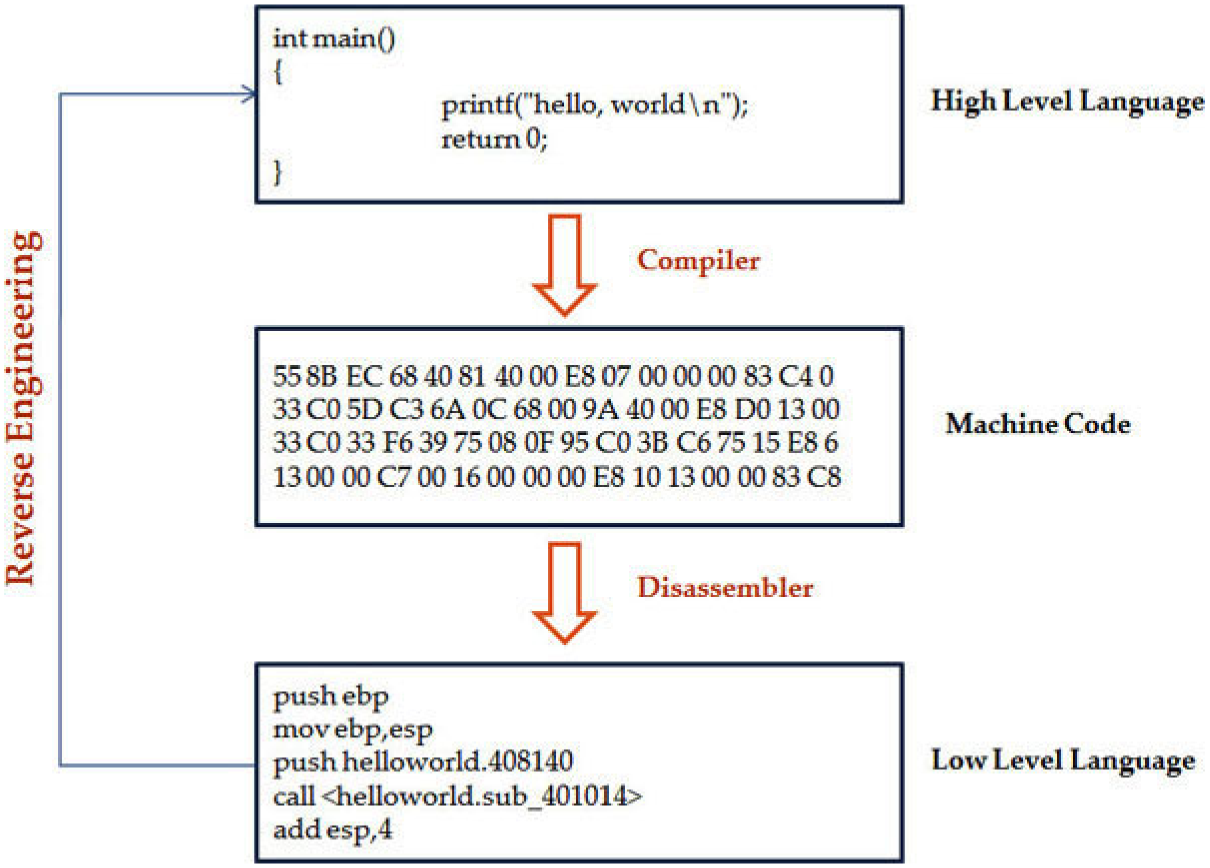


Figure 3.1: Importance of RE tools

Reverse engineering helps to regenerate the application logic without having the source code of an application or a program. Malware researchers follow this concept to perform reverse engineering tasks.

Now we will discuss a few tools required for reverse engineering. We will also use these tools throughout this book.

Reverse engineering tools

For easy understanding, reverse engineering tools are divided in different categories like PE editors, disassemblers, and debuggers. Within each category, we have free or open-source tools available, and all tools within a given category serve the same purpose with some slight feature differences. We will focus on free and open-source tools with graphical user interface. All these tools are used throughout this book.

Portable Executable Editors

PE stands for **Portable** Portable Executable is the standard Windows file format. Every Windows executable uses this file format. **Dynamic Link Library Component Object Model** files, and .NET executable use the PE file format. The following figure shows the basic structure of PE:

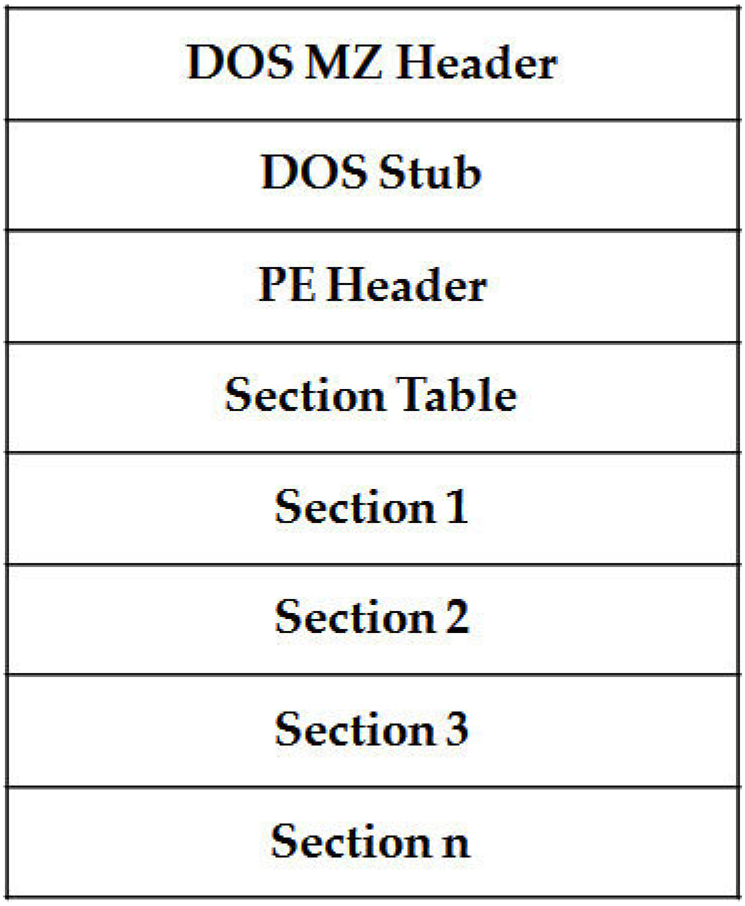


Figure 3.2: *PE structure*

All the PE files start with the DOS header then PE header also called NT header and sections that are common in executable. These common sections are as follows:

This section contains the actual binary executable code.

This section represents uninitialized data.

This section contains read only data, such as constants, strings.

This section is the resource section where resource information is stored.

This is the export data section containing information about exported functions.

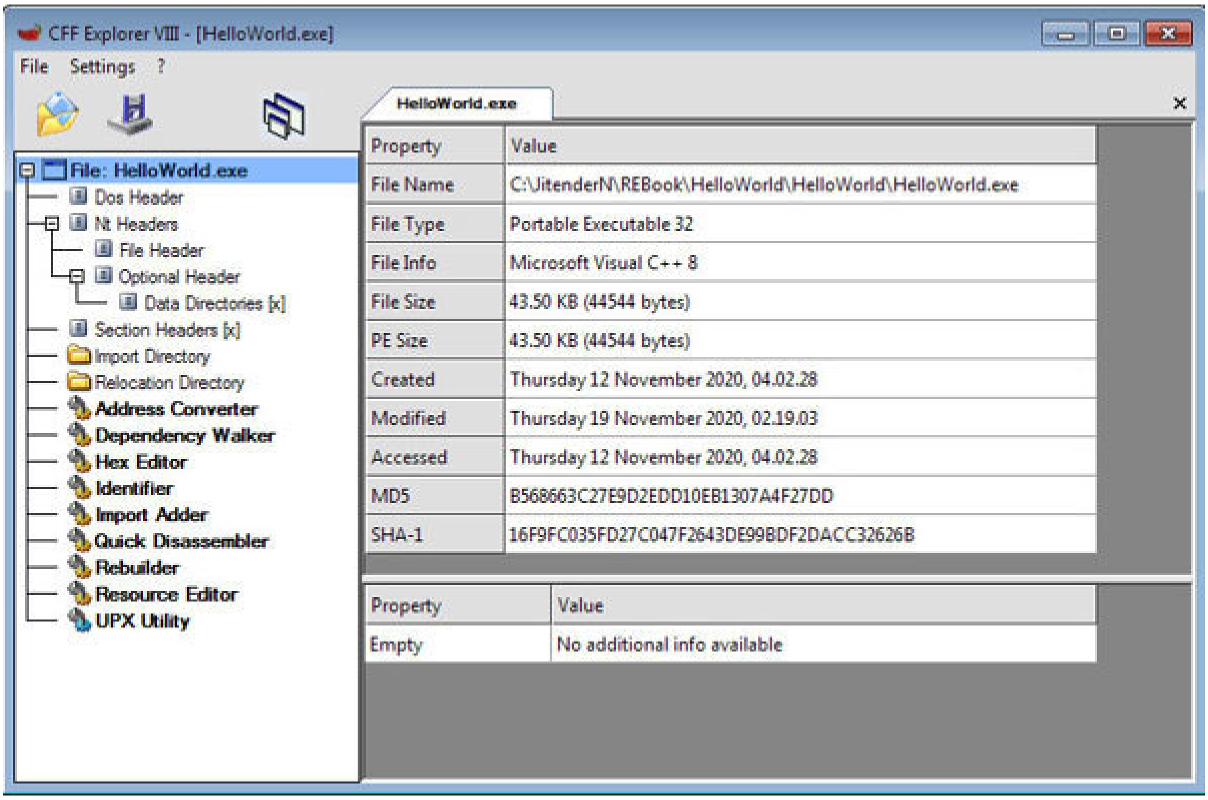
This is import data section which contains information about imported functions along with the import directory and import address table.

Initially, debug information was placed in this debug information section. PE files also support a separate debug file with **.dbg** extension.

To view and edit all these details, we can use the following tools.

CFF Explorer

This tool is designed to view and edit PE files without losing the internal structure of PE files. This tool is not only used by reverse engineers but also the developer of applications.



Property	Value
File Name	C:\JitenderN\REBook\HelloWorld\HelloWorld\HelloWorld.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ 8
File Size	43.50 KB (44544 bytes)
PE Size	43.50 KB (44544 bytes)
Created	Thursday 12 November 2020, 04.02.28
Modified	Thursday 19 November 2020, 02.19.03
Accessed	Thursday 12 November 2020, 04.02.28
MD5	B568663C27E9D2EDD10EB1307A4F27DD
SHA-1	16F9FC035FD27C047F2643DE99BDF2DACC32626B

Property	Value
Empty	No additional info available

Figure 3.3: *CFF Explorer*

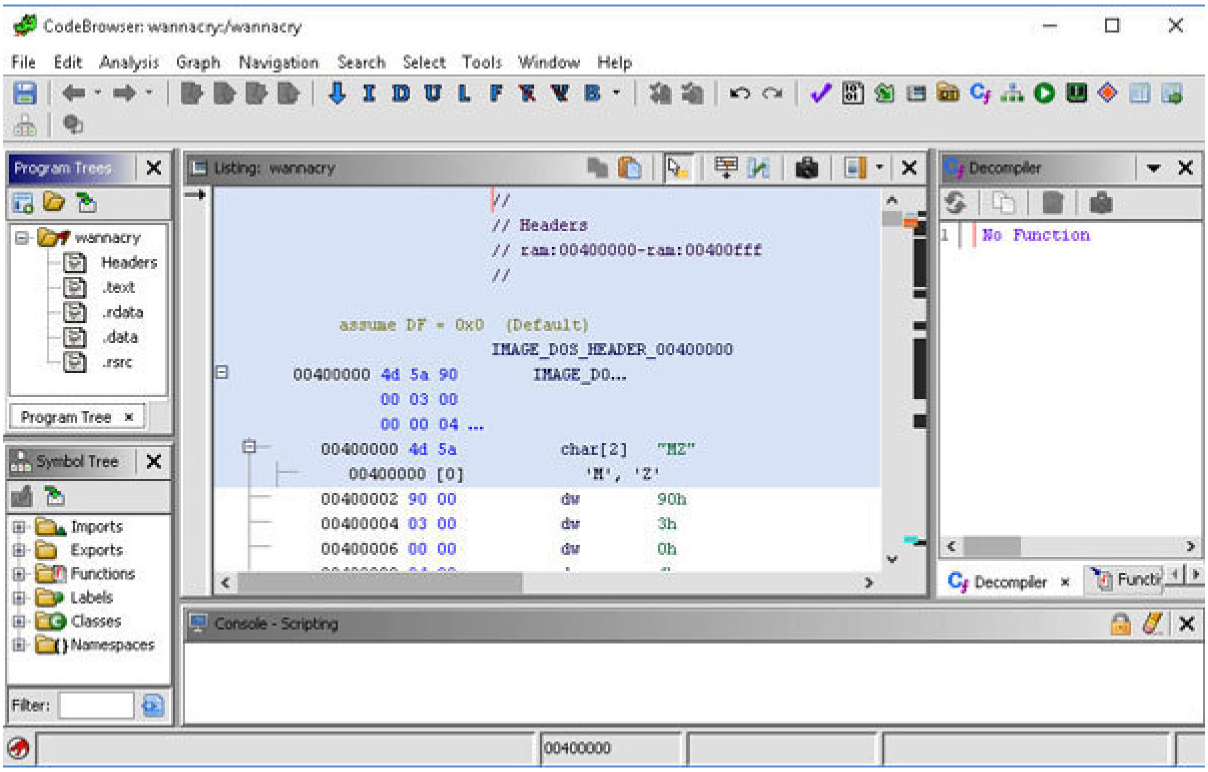
We will be using this CFF explorer to edit PE files in our subsequent chapters.

Disassembler

As we discussed in the earlier section that a machine code is not human readable, we need some tools to convert machine code into a human-readable format. This is where disassemblers come into picture. Disassemblers are used to convert a machine code into a human-readable assembly code. The following are some disassemblers used in this book.

Ghidra

This is an open-source tool developed by the **National Security Agency** It is free and used for reverse engineering. Its source code was released on April 4, 2019. This tool is used by malware researchers and reverse engineers to analyze malwares and find vulnerabilities in applications.



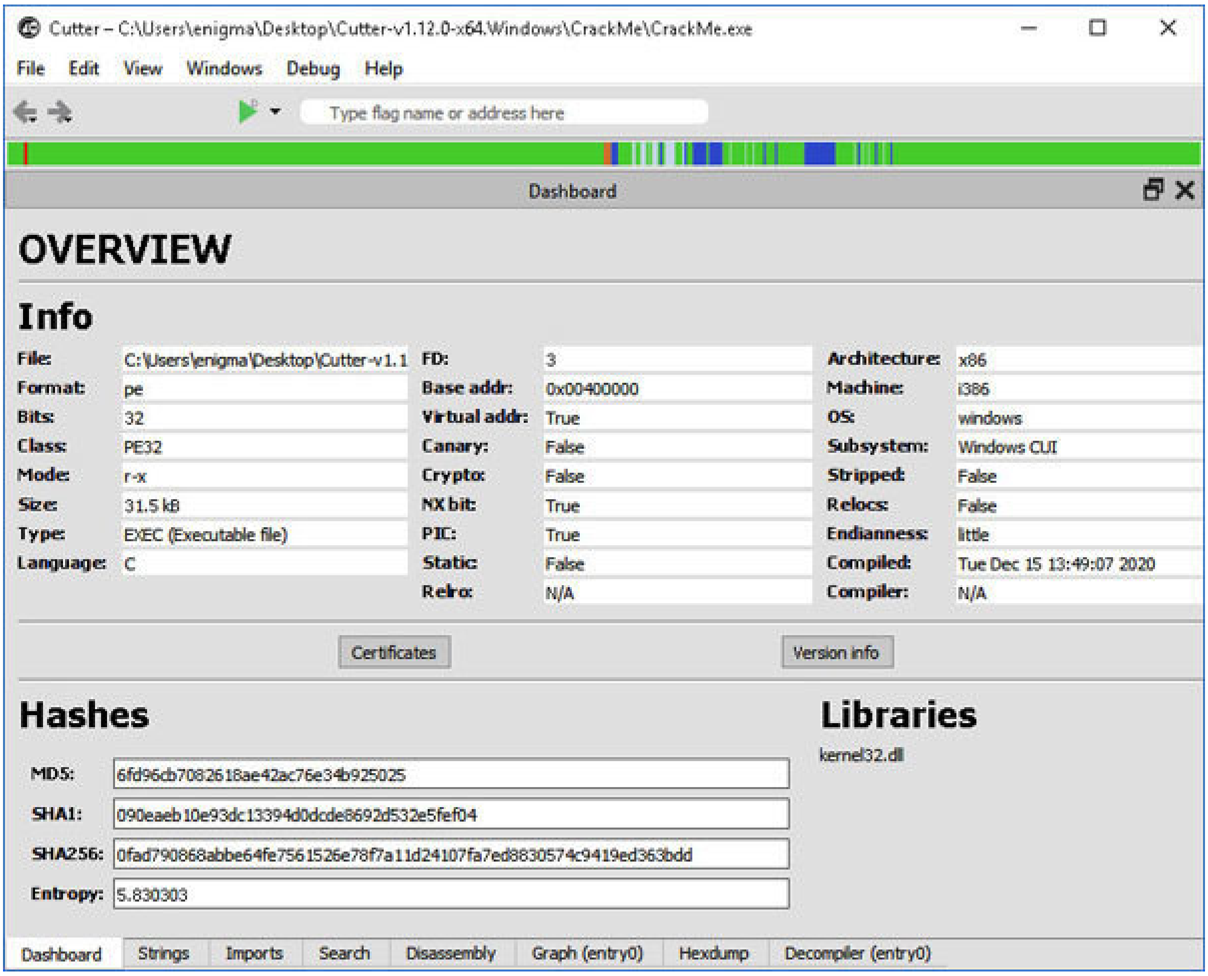
```
                        //
                        // Headers
                        // ram:00400000-ram:00400fff
                        //

           assume DF = 0x0  (Default)
                        IMAGE_DOS_HEADER_00400000
     00400000 4d 5a 90        IMAGE_DO...
              00 03 00
              00 00 04 ...
       00400000 4d 5a              char[2]   "MZ"
         00400000 [0]               'M', 'Z'
       00400002 90 00              dw        90h
       00400004 03 00              dw        3h
       00400006 00 00              dw        0h
```

```
1  No Function
```

Figure 3.4: Ghidra

Cutter

Cutter is an open-source interface to the Radare2 reverse engineering framework. Radare2 is the command line tool for reverse engineering and is used for static and dynamic analysis of binary formats on different platforms and architectures. Cutter is the graphical user interface of Radare2.



File: C:\Users\enigma\Desktop\Cutter-v1.1
Format: pe
Bits: 32
Class: PE32
Mode: r-x
Size: 31.5 kB
Type: EXEC (Executable file)
Language: C

FD: 3
Base addr: 0x00400000
Virtual addr: True
Canary: False
Crypto: False
NX bit: True
PIC: True
Static: False
Relro: N/A

Architecture: x86
Machine: i386
OS: windows
Subsystem: Windows CUI
Stripped: False
Relocs: False
Endianness: little
Compiled: Tue Dec 15 13:49:07 2020
Compiler: N/A

MD5: 6fd96cb7082618ae42ac76e34b925025
SHA1: 090eaeb10e93dc13394d0dcde8692d532e5fef04
SHA256: 0fad790868abbe64fe7561526e78f7a11d24107fa7ed8830574c9419ed363bdd
Entropy: 5.830303

kernel32.dll

Figure 3.5: Cutter

Debuggers

When we run an application or a binary, we check the state of a running program. A debugger gives the dynamic state of a binary when it is executed in the memory. Some vulnerabilities are not captured by the developer while writing or executing a code. This is where we use debuggers to run the code and monitor the registers, memory locations, and other parameters. We will be heavily using the following debugger in this book.

x32dbg

A debugger comes in two flavors: 32 bit and 64 bit. x32dbg is used for x86 (32 bit) binaries. Another version is x64dbg, which is used for x64 (64 bit) binaries. As shown in the following figure, x32dbg is divided into 4 sections:

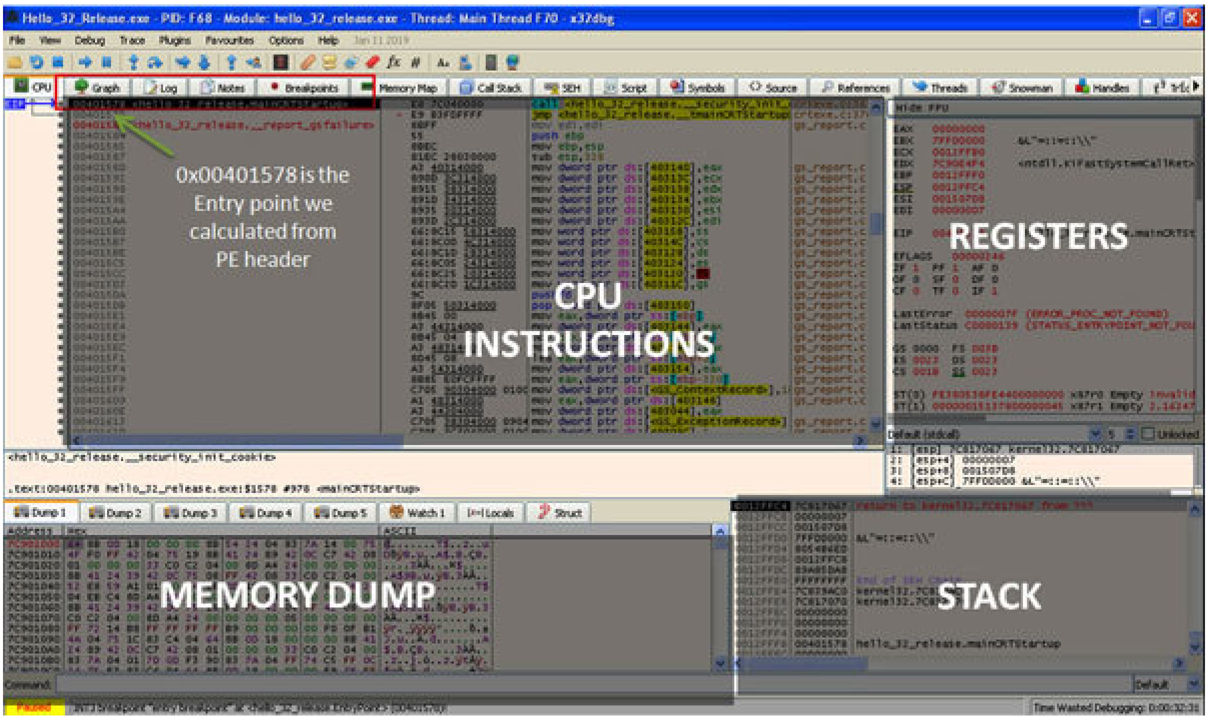


Figure 3.6: x32dbg

As shown in the preceding figure, these four sections are as follows:

Disassembly or CPU This is where a machine code converted into an assembly code is shown. The first column is the memory address of the instruction. The second column is the opcode, also called the **operation** The third column is the assembly instructions. The forth column shows the comment about instructions.

Registers and This section shows the registers and their values during a dynamic analysis of the binary. It also shows the value of a flag to display the current state of the processor.

As discussed in the earlier chapter, a stack is **Last In First Out** This means the last thing added (pushed) will be the first thing to get pulled off (popped). A stack grows from a higher memory address to a lower memory address.

Memory It is like hex editor which shows the hexadecimal code of a binary in the memory. It shows raw data in the hexadecimal and ASCII formats. The value on a particular memory address can be changed by double clicking on the respective memory location.

Conclusion

In this chapter, we covered different reverse engineering tools used throughout this book. We also discussed the difference between a disassembler and a debugger. The importance of tools in reverse engineering was also discussed and then we listed some PE editors, disassemblers, and debuggers. In the next chapter, we will walk through the assembly instructions that will help us read and understand a disassembled assembly code.

CHAPTER 4

Walk Through on Assembly Instructions

In the last chapter, we introduced some assembly language instructions. There are many types of instructions in the assembly language which can be grouped together to get a clear understanding and objective of a specific set of assembly instructions. To understand the relevance of walking over the basic assembly instructions in reverse engineering, we will take up a real-life example.

Have you ever opened a toy in your childhood? It was always fun to open a toy and check its internal components. Today, if you had to understand the internal working of a toy, you need to first understand the different components installed internally in the hardware design of the toy. Now, to uncover the internal working of the toy, we have to understand the working of each component installed in it. This whole process can be somewhat compared to the reverse engineering of the toy.

On similar lines, for reverse engineer of any software or application, we need to disassemble it into different instructions and the understanding of the disassembled assembly instructions becomes a must. So, to understand the working of any software or application, we need to have a clear understanding of the instructions and their execution path. This is where understanding of different instructions in assembly language is needed.

Structure

In this chapter, we will cover the following topics:

Different assembly language instructions

Syntax of the assembly instructions

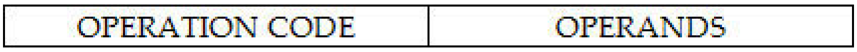
Grouping of assembly instructions

Objective

After going through this chapter, you will be able to understand the important assembly instructions used in reverse engineering and how these instructions are grouped in various sections for easy understanding along with some examples. We will also learn the syntax of the instructions, their internal working, and the basic concept behind the different types of instructions.

Different assembly language instructions

Instructions are the basic building blocks of Assembly Code. Instructions are a combination of the operation code and zero or more operands.



OPERATION CODE	OPERANDS

***Figure* 4.1:** *Assembly instruction syntax*

Operation code is often referred to as opcode. Operands can be of three types:

Intermediate These are fixed hexadecimal value, like 0x0A.

Register They can be any register, such as ECX, EAX, etc.

Memory address These are memory addresses and are represented between square brackets, like [EAX].

The program code is saved in the memory in a sequence of operation code (or opcode) and operands. They are saved in consecutive memory locations. As shown in the following image, instruction 1 occupies two memory locations, instruction 2 occupies the same two memory locations, and instruction 3 occupies only one memory location.

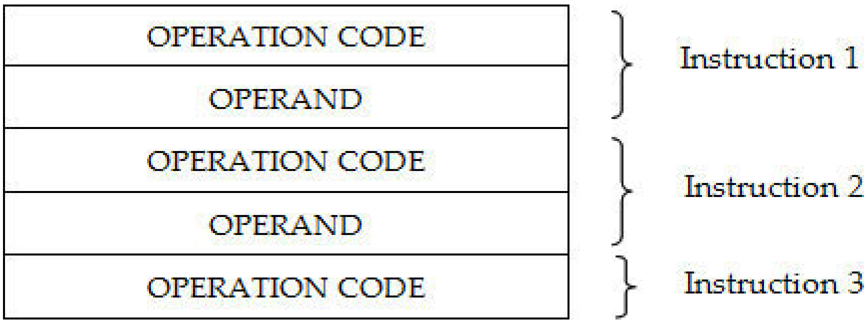


Figure 4.2: *Program code in memory*

Look at the following example of an assembly instruction with register operands:

Instruction Format:

OPERATION_CODE DESTINATION_OPERAND SOURCE_OPERAND

Example

MOV EAX, ECX
OPERATION_CODE = MOV, stands for move
DESTINATION_OPERAND = EAX, is a destination register
SOURCE_OPERAND = ECX, is a source register

This instruction moves data from the ECX register to the EAX register. Now each instruction represented by the opcodes is also called the operation code. These opcodes tell the CPU what operation the program wants to perform. So, the opcodes for the preceding instruction is 89 C8, which is the hexadecimal representation of the instruction. The disassembler converts these opcodes into a human readable format. So, 89 C8 will be converted to **MOV EAX,**

Now that we are clear with the concept of assembly instructions, we will move on to the explanation of major assembly instructions that come on the way to understand reverse engineering. These assembly instructions can be segmented or grouped in various sections for easy understanding. Grouping of assembly instructions can be broadly classified into the following categories:

Stack Instructions

Data Transfer Instructions

Arithmetic Instructions

Program Execution Instructions

Branching Instructions

Bit Manipulation Instructions

Processor Control Instructions

String Instructions

Let's walk through each section one by one and understand assembly instructions that fall under each category.

Stack Instructions

These are general purpose instructions used for transfer operations across the stack.

PUSH

Instruction Format:

PUSH SOURCE_OPERAND

Meaning:

Push copies the value from the source operand onto the top of the stack and decrements the ESP register.

Flags Affected: None

PUSHAD

Instruction Format:

PUSHAD

Meaning:

Pushes the values of all the registers onto the stack. Registers are pushed in the order of EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI.

Flags Affected: None

PUSHFD

Instruction Format:

PUSHFD

Meaning:

This instruction pushes the flags register onto the stack.

Flags Affected: None

POP

Instruction Format:

POP DESTINATION_OPERAND

Meaning:

POP retrieves the value from the top of the stack and copies it to the destination operand. It increments the ESP register.

Flags Affected: None

POPAD

Instruction Format:

POPAD

Meaning:

This instruction pops the values of the stack and copies them to all the registers. Registers are popped in the order of EDI, ESI, EBP, ESP, EDX, ECX, and EAX. The ESP value is ignored in POPAD.

Flags Affected: None

POPFD

Instruction Format:

POPFD

Meaning:

This instruction pops the DWORD from the stack into the flags register.

Flags Affected: None

RET

Instruction Format:

RET [nBytes]

Meaning:

When a function calls another function, the function that calls another function is called ‘caller’ and the function that is called is named ‘callee’. RET transfers control from the callee to the caller, to the return address saved on the stack. nBytes are optional; when nBytes are mentioned, it means that n bytes are released to clean up the stack.

Data Transfer Instructions

These are general purpose instructions used for data transfer operations.

MOV

Instruction Format:

MOV DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Moves data from the source operand to the destination operand and the result is stored in the destination operand.

Flags Affected: None

LEA

Instruction Format:

LEA REGISTER OPERAND

Meaning:

LEA stands for Load Effective Address and it loads the register with the memory address of the operand.

Flags Affected: None

XCHG

Instruction Format:

XCHG DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

XCHG instruction exchanges the values between the source operand and the destination operand.

Flags Affected: None

CMPXCHG

Instruction Format:

CMPXCHG DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

This instruction is used for comparing and exchanging. EAX is compared with the

If EAX is equal to **DESTINATION_OPERAND** then **DESTINATION_OPERAND** is loaded with

If EAX is not equal to **DESTINATION_OPERAND** then EAX is loaded with

Flags Affected: CF ZF SF AF PF OF

LAHF

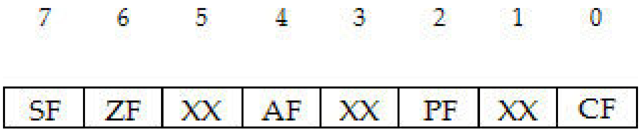
Instruction Format:

LAHF

Meaning:

This instruction loads AH from the flags. It copies the status to the AH register and only 5 flags are copied to bits 7, 6, 4, 2, and 0 of the AH register; bits 5, 3 and 1 of the AH register will be unaffected.

AH register after running this instruction:



7	6	5	4	3	2	1	0
SF	ZF	XX	AF	XX	PF	XX	CF

Figure 4.3: *Load AH from Flags*

Flags Affected: None

SAHF

Instruction Format:

SAHF

Meaning:

It stores the AH into the flags. This instruction copies the bits of the AH register (bits 7, 6, 4, 2, and 0) to SF, ZF, AF, PF, and CF flags.

Flags Affected: SF, ZF, AF, PF, and CF

LAR

Instruction Format:

LAR DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Loads the effective right from the segment descriptor specified by the SOURCE_OPERAND into the DESTINATION_OPERAND.

Flags Affected: ZF

MOVSX

Instruction Format:

MOVSX DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Move with Sign Extension. Suppose you have a smaller value and want to copy it to a big register. In that case, we will use it copies **SOURCE_OPERAND** to **DESTINATION_OPERAND** and fills the bits not provided by **SOURCE_OPERAND** with sign bits.

Flags Affected: None

MOVZX

Instruction Format:

MOVZX DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Move with Zero Extension. Suppose you have a smaller value and want to copy it to a big register. In that case, we will use it copies **SOURCE_OPERAND** to **DESTINATION_OPERAND** and fills the bits not provided by **SOURCE_OPERAND** with zero.

Flags Affected: None

XLAT

Instruction Format:

XLAT

Meaning:

The XLAT instruction replaces the AL register from the table index to the table entry. It sets the AL to DS:[EBX + unsigned AL]. The value of AL is treated as an unsigned index, which is added to the EBX register to get the table entry. EBX contains the base address of the table.

MOVS

Instruction Format:

MOVS DESTINATION_STRING SOURCE_STRING

Meaning:

This instruction copies data (Byte, WORD, DWORD) from **SOURCE_STRING** to where:

SOURCE_STRING is pointed by DS:SI (or ESI)

DESTINATION_STRING is pointed by ES:DI (or EDI)

SI is the offset address in **Data Segment**

DI is the offset address in **Extra Segment**

Flags Affected: None

Arithmetic Instructions

These instructions are used for arithmetic operations in the assembly language.

AAA

Instruction Format:

AAA

Meaning:

This instruction adjusts the ASCII after addition. The AAA instruction is used after the addition (ADD instruction) of two unpacked BCD numbers. Unpacked BCD numbers are ASCII single-digit numbers from 0 to 9 or 0x30 to 0x39. When the AAA instruction is used after the ADD instruction, it converts the result to a two-digit BCD number. For ASCII codes, refer to the

When two ASCII coded numbers are added, the result is not ASCII. To convert this to ASCII, AAA is used after ADD.

Flags Affected: AF CF (SF,ZF,OF,PF undefined)

Example

```
XOR AH, AH          ;clear AH register

MOV AL, 32H       ;move ASCII 2 in AL
```

```
ADD AL, 39H         ;add ASCII 9, the result should be ASCII 11,
but we get 6B in AL

AAA                             ;AH=0x01, AL=0x01 and AX=0x0101
```

AAS

Instruction Format:

AAS

Meaning:

The instruction adjusts the ASCII after subtraction. The AAS instruction is used after the subtraction (SUB instruction) of two unpacked BCD numbers. Unpacked BCD numbers are ASCII single-digit numbers from 0 to 9 or 0x30 to 0x39. When the AAS instruction is used after the SUB instruction, it converts the result to a two-digit BCD number.

Flags Affected: AF CF (SF,ZF,OF,PF undefined)

AAD

Instruction Format:

AAD

Meaning:

The ASCII is adjusted before division. This instruction is used before the division instruction to convert the unpacked BCD number in AH and AL to the binary equivalent. The quotient will be saved in AL and the remainder will be saved in AH; both are unpacked BCD.

Flags Affected: PF, SF, ZF

Example

Divide 68 by 8.

MOV AX, 0608H ;AH=0x06, AL=0x08 and AX=0x0608

MOV CH, 08H ;divide ASCII 8, the result should be 8 in quotient and 4 in remainder

AAD ;AX = 0044 = 44H = 68

```
DIV CH                    ;Divide AX by in CH, result: AL = 08, AH = 04 unpacked BCD
```

AAM

Instruction Format:

AAM

Meaning:

The ASCII is adjusted for multiplication. This instruction is used in the multiplication of two ASCII numbers. When the unpacked BCD digits are multiplied, the result stored in AX is converted to the unpacked BCD number in AH and AL.

Flags Affected: PF, SF, ZF

Example

```
MOV AL, 00000111        ;move 7 in AL, masking upper 4 bits

MOV BH, 00000101        ;move 5 in BH, masking upper 4 bits

MUL BH                               ;AX = 23H, 35 in decimal

AAM                                     ;result: AX = 0203H, AH = 02H, AL = 03H unpacked BCD
```

ADC

Instruction Format:

ADC DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Adds two operands, source operand and destination operand. The result is stored in the destination operand. If CF flag is set to 1, then 1 is added to the destination.

Flags Affected: AF CF OF PF SF ZF

ADD

Instruction Format:

ADD DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

The instruction adds two operands, source operand and destination operand. The result is stored in the destination operand.

Flags Affected: AF CF OF PF SF ZF

CMP

Instruction Format:

CMP DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

It subtracts two operands, source operand and destination operand. The result is not stored, but the flags are updated. The flags are subsequently checked in the instructions.

Flags Affected: AF CF OF PF SF ZF

DAA

Instruction Format:

DAA

Meaning:

The decimal is adjusted after addition. This instruction comes after the ADD or ADC instruction to adjust the final result in BCD. This only works with the AL registers.

Flags Affected: AF CF (OF, PF, SF, ZF undefined)

Example

```
MOV DX, 1122H      ;load 1122H BCD

MOV BX, 3088H      ;load 3088H BCD

MOV AL, BL             ;only AL register works for DAA

ADD AL, DL             ;
```

```
DAA                              ; BCD adjusted result, answer
saved in AL
```

DAS

Instruction Format:

DAS

Meaning:

The decimal is adjusted after subtraction. This instruction comes after the SUB or SBB instruction to adjust the final result in BCD. This only works with the AL registers.

Flags Affected: AF CF (OF, PF, SF, ZF undefined)

Example:

```
MOV DX, 1122H      ;load 1122H BCD

MOV BX, 3088H      ;load 3088H BCD

MOV AL, BL             ;only AL register works for DAA

SUB AL, DL              ;
```

```
DAS                                ; BCD adjusted result, answer
saved in AL
```

DEC

Instruction Format:

DEC DESTINATION_OPERAND

Meaning:

The instruction decrements one from the destination operand which can be register or a memory location. The result is saved back in the register or memory location.

Flags Affected: AF OF PF SF ZF

Example:

```
MOV EAX, 02H        ;load EAX with 02H

DEC EAX                   ;EAX will be 01H
```

DIV

Instruction Format:

DIV SOURCE_OPERAND

Meaning:

The DIV instruction is used to divide the unsigned QWORD/DWORD/WORD by DWORD/WORD/BYTE.

When a WORD is divided by byte, the number to be divided, that is WORD, must be in the AX register and the divisor, which is source operand, can be in register or memory location. After division, the quotient will be saved in AL and the remainder will be in AH.

When a DWORD is divided by WORD, the number to be divided, that is DWORD, must be in DX:AX (most significant WORD in DX and least significant WORD in AX). The divisor, which is the source operand, can be in register or memory location. After division, the quotient will be saved in AX and the remainder will be saved in DX.

When a Double DWORD (QWORD) is divided by DWORD, the number to be divided, that is Double DWORD (QWORD), must be in EDX:EAX (higher order DWORD in EDX and lower order

DWORD in EAX). The divisor, which is the source operand, can be in register or memory location. After division, the quotient will be saved in EAX and remainder in EDX.

Flags Affected: None

Example

```
MOX DX, 0                 ;load DX with 00H

MOV AX, 0x8003       ;load AX with 8003H

MOV CX, 0x100         ;load CX with 100H

DIV CX                      ;AX=80H, DX=03H
```

IDIV

Instruction Format:

IDIV SOURCE_OPERAND

Meaning:

The DIV instruction (Integer Divide) is used in division of signed data. The rest is the same with respect to the dividend, divisor, quotient, and remainder as in DIV instruction.

Flags Affected: None

MUL

Instruction Format:

MUL SOURCE_OPERAND

Meaning:

The MUL instruction is used to multiple the unsigned DWORD/WORD/BYTE by DWORD/WORD/BYTE.

When a BYTE is multiplied by BYTE, the multiplicand, that is BYTE, must be in the AL register and the multiplier, which is the source operand, can be in the register or memory location. After multiplication, the result will be saved in AX. Higher order 8 bits are stored in AH and lower order 8 bits are stored in AL.

When a WORD is multiplied by WORD, the multiplicand, that is WORD, must be in the AX register and the multiplier, which is the source operand, can be in the register or memory location. After multiplication, the result will be DWORD, which is saved in DX:AX. The higher order WORD is stored in DX and the lower order WORD in AX.

When a DWORD is multiplied by DWORD, the multiplicand, that is DWORD, must be in the EAX register and the multiplier, which is the source operand, can be in the register or memory location.

After multiplication, the result will be QWORD, which is saved in EDX:EAX. The higher order DWORD is stored in EDX and the lower order DWORD in EAX.

Flags Affected: OF, CF

Example

```
MOV AX, 0x8003      ;load AX with 8003H

MOV CX, 0x100        ;load AX with 100H

MUL CX                    ;DX=80H, AX=0300H
```

IMUL

Instruction Format:

IMUL SOURCE_OPERAND

Meaning:

The **Integer Multiple** instruction is used to multiply the signed data. The rest is the same with respect to the multiplicand and multiplier as in MUL instruction.

Flags Affected: OF, CF

INC

Instruction Format:

INC DESTINATION_OPERAND

Meaning:

It increments or adds one to the destination operand, which can be the register or a memory location. Result is saved back in register or a memory location.

Flags Affected: AF OF PF SF ZF

Example

```
MOV EAX, 02H        ;load EAX with 02H

INC EAX                     ;EAX will be 03H
```

NEG

Instruction Format:

NEG DESTINATION_OPERAND

Meaning:

This instruction changes the sign of the destination operand from a positive number to a negative number or from a negative number to a positive number. Basically, it subtracts 0 from the destination operand and performs the 2's complement to save the result back in the destination operand. 2's complement of an 8-bit number 00100101 -> 11011010 (Invert bits) -> then add 00000001 = 11011011

Flags Affected: AF OF PF SF ZF

Example

```
MOV EAX, 02H        ;load EAX with 02H, (2 in decimal)

NEG EAX                  ;EAX will be FFFFFFFEH, (-2 in decimal)
```

SBB

Instruction Format:

SBB DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

It subtracts source operand from the destination operand. The result is stored in the destination operand. If the CF flag is set to 1, then 1 is subtracted from the destination.

Flags Affected: AF CF OF PF SF ZF

SUB

Instruction Format:

SUB DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

Subtract the source operand from destination operand. The result is stored in the destination operand.

Flags Affected: AF CF OF PF SF ZF

XADD

Instruction Format:

XADD DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

The **Exchange and Add** instruction is the same as the ADD instruction where it adds the source operand and the destination operand, to store the result in the destination operand. The difference from ADD instruction is that after the XADD instruction, the original value of the destination operand is copied to the source operand.

Flags Affected: AF CF OF PF SF ZF

Example

```
MOV EAX, 0x00000001     ; load EAX with 01H, (1 in decimal)

MOV EBX, 0x00000002     ; load EBX with 02H, (2 in decimal)

XADD EAX,EBX            ; EAX= 03H (3 in decimal), EBX=01H
```

Program Execution Instructions

These instructions are used in controlling the program execution flow and flags.

CALL

Instruction Format:

CALL DESTINATION_OPERAND

Meaning:

During program execution, a procedure is called from another function; the CALL instruction is used. Using CALL instruction pointer jump to the code of a procedure called within a function. When the CALL instruction is executed, the memory address of the next instruction after the CALL instruction is pushed onto the stack. This memory address is popped back from the stack once the called procedure execution is over with the RET instruction.

Flags Affected: None

Example

CALL Program flow is deviated to the procedure label memory location

ENTER

Instruction Format:

ENTER Storage, Level

Meaning:

In Intel Architecture, another method of performing the procedure call is supported with the ENTER and LEAVE instructions. These instructions create and release the stack frame for the procedure to store variables and pointers to return the execution from the procedure. In block-structured language like C and Pascal, these instructions provide machine language support for procedure calls.

The ENTER instruction has 2 operands:

This tells the number of bytes to be reserved on the stack for the procedure call.

Also called lexical nesting level (from 0 to 31), it is the depth of procedure in a ladder of procedure calls.

Flags Affected: None

LEAVE

Instruction Format:

LEAVE

Meaning:

This instruction releases the stack frame created for the procedure to store variables and pointers to return the execution from the procedure by restoring (E)SP/(E)BP.

Flags Affected: None

INT

Instruction Format:

INT
type is between 0 and 255.

Meaning:

Interrupt is a condition which halts the processor to temporarily work on some other task and then return to the main task. Interrupt is an event or signal that asks for the CPU's attention. This is used by peripheral hardware devices to access the CPU. Whenever the interrupt occurs, the processor completes the current set of instructions and then starts the **interrupt service routine** or interrupt handler. ISR is a routine which contains a set of instructions to handle specific interrupts. This ISR tells the processor what to do when an interrupt occurs.

Now, the INT instruction allows a program to explicitly call the interrupt handler. The following tasks are done when an INT instruction is called:

The FLAGS register is pushed on the stack.

The **Code segment** is pushed on the stack.

The offset of the next instruction after the INT instruction is pushed onto the stack.

The IP (Instruction Pointer) is loaded from an absolute memory address, which is a multiple of the interrupt type by 4. If INT is 8, the IP (Instruction Pointer) will be read from 8 * 4 = 32 decimal = 0x00020 memory location.

The code segment will be the next WORD location. The CS will be 0x00022 (0x00020 + 2).

Reset TF Trap flag (TF) and IF Interrupt flag (IF).

Flags Affected: IF, TF

INTO

Instruction Format:

INTO

Meaning:

Interrupt Overflow If the **Overflow flag** is set, this instruction raises the overflow exception. If OF is not set, then the instruction execution continues without raising an exception. This helps to check the overflow condition. The following tasks are done when an INT instruction is called:

The FLAGS register is pushed on the stack.

The CS (code segment) is pushed on the stack.

The offset of the next instruction after the INT instruction is pushed onto the stack.

The IP (Instruction Pointer) is loaded from an absolute memory address, which is multiple of interrupt type by 4.

If the overflow condition is INT 4, the IP (Instruction Pointer) will be read from 4 * 4 = 16 decimal = 0x00010 memory location.

The code segment will be the next WORD location. The CS will be 0x00012 (0x00010 + 2).

Reset Interrupt flag and Trap flag to 0.

Flags Affected: IF, TF

IRET

Instruction Format:

IRET

Meaning:

Interrupt Return This instruction is used to end the **Interrupt Service Routine** or interrupt handler and return the execution to the interrupted program. When an **Interrupt Service Routine** is called, the instruction pointer, code segment, and flags registers are pushed onto the stack. On return from the ISR instruction pointer, the code segment and flags are restored back from the stack to continue the execution of the interrupted program.

Flags Affected: AF, CF, DF, IF, ZF, SF, TF, PF

LOOP

Instruction Format:
LOOP DESTINATION

Meaning:

In **LOOP** instruction, the (E)CX register is decremented by 1. If the new value in the (E)CX register is non-zero, then a jump is taken to the destination mentioned in the instruction. If the (E)CX register is decremented and the ECX is equal to 0, then no action will be taken and the instruction next to the LOOPE instruction is executed.

Flags Affected: None

Example

LOOPE MEM_LOC ; Program will jump to the memory location if ECX is non-zero

; after decrementing

LOOPE

Instruction Format:

LOOPE DESTINATION

Meaning:

In the LOOPE instruction, the (E)CX register is decremented by 1. If the new value in the (E)CX register is non-zero and the ZF flag is set to 1, then a jump is taken to the destination mentioned in the instruction. If the (E)CX register is decremented and ECX is equal to 0, then no action will be taken and the instruction next to the LOOPE instruction is executed.

Flags Affected: None

Example

LOOPE MEM_LOC ; Program will jump to the memory location if ECX is non-zero

; after decrementing and ZF=1 due to the previous instruction

LOOPNE

Instruction Format:

LOOPNE DESTINATION

Meaning:

In the LOOPNE instruction, the (E)CX register is decremented by 1. If the new value in the (E)CX register is non-zero and the ZF flag is set to 0, then a jump is taken to the destination mentioned in the instruction. If the (E)CX register is decremented and ECX is equal to 0, then no action will be taken and the instruction next to the LOOPNE instruction is executed.

Flags Affected: None

Example

LOOPNE MEM_LOC ; Program will jump to the memory location if ECX is non-zero

; after decrementing and ZF=0 due to the previous instruction

TEST

Instruction Format:

TEST DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

This instruction performs the logical AND between the source operand and the destination operand. Unlike the AND instruction, the TEST instruction does not update any of the operands. It updates the flags without saving the results. This instruction is used to check the registers for zero, without altering its value.

Flags Affected: SF, ZF, and PF

Example

```
TEST EAX, 01H      ; IF EAX=01H
                   ; 01H AND 01H = 01H (non –zero value)
                   ; Result is non-zero value, ZF is not set, ZF=0
```

```
TEST EAX, 02H      ; IF EAX=01H

                                        ; 01H AND 02H = 00H (zero
value)

                                        ; Result is zero value, ZF is
set, ZF=1

TEST EAX, EAX    ; if EAX is equal to 0, set ZF to 1
```

Branching Instructions

This includes instructions which help in controlling the code flow. There are two types of jumps in x86:

Unconditional The instruction pointer jumps to the code path mentioned.

Conditional The instruction pointer jumps after evaluating the condition. The condition is commonly evaluated using two instructions. Both the instructions do not store any result but change the flags in the EFLAGS register.

TEST It performs the logical AND.

CMP It performs subtraction.

Note: The DESTINATION_OPERAND used in branching instructions is also named as a LABEL or DESTINATION LABEL or DESTINATION ADDRESS. This is a displacement from the address of the unconditional/conditional jump instruction itself or the absolute address.

JMP

Instruction Format:

JMP DESTINATION_OPERAND

Meaning:

This is an unconditional jump instruction where the instruction pointer jumps to the destination operand during execution.

Flags Affected: None

Example

JMP PROC_LABEL ;Program flow jumps to the procedure label memory location

JZ

Instruction Format:

JZ DESTINATION_OPERAND

Meaning:

This is a conditional jump instruction which jumps to the destination operand if the zero flag (ZF) is set to 1. If ZF is set to 0, then no action will be taken and the next instruction following it will be executed.

Flags Affected: None

Example

JZ MEM_LOC ; Program will jump to the memory location if ZF=1

JNZ

Instruction Format:

JNZ DESTINATION_OPERAND

Meaning:

This is a conditional jump instruction which jumps to the destination operand if the zero flag (ZF) is set to 0. If ZF is set to 1, then no action will be taken and the next instruction following it will be executed.

Flags Affected: None

Example

JNZ MEM_LOC ; The program will jump to the memory location if ZF=0

JE

Instruction Format:

JE DESTINATION_OPERAND

Meaning:

This instruction is the same as JZ. As **Jump if Equal** is a conditional jump instruction, it is commonly used after the CMP instruction. The jump will happen if the destination operand is equal to the source operand in the conditional jump.

Flags Affected: None

Example

CMP EAX, 01H ; Compare the EAX value with 01H

JE MEM_LO ; The program will jump to the memory location if EAX = 01H and ZF = 1

JNE

Instruction Format:

JNE DESTINATION_OPERAND

Meaning:

This instruction is the same as JNZ. As **Jump if Not Equal** is a conditional jump instruction, it is commonly used after the CMP instruction. The jump will happen if the destination operand is not equal to the source operand.

Flags Affected: None

Example

CMP EAX, 01H ; Compare the EAX value with 01H

JNE MEM_LO ; The program will jump to the memory location if EAX != 01H and ZF = 0

JG

Instruction Format:

```
JG DESTINATION_OPERAND
```

Meaning:

As **Jump if Greater (JG)** is a conditional jump instruction, it is commonly used after the CMP instruction where it performs a signed comparison between the destination operand and the source operand. The jump will happen if the destination operand is greater than the source operand.

Flags Affected: None

Example

```
CMP EAX, 02H        ; Compare the EAX value with 02H

JG MEM_LO           ; The program will jump to the memory location if EAX > 02H
```

JGE

Instruction Format:

JGE DESTINATION_OPERAND

Meaning:

The **Jump if Greater or Equal** is a conditional jump instruction, so it is commonly used after the CMP instruction where it performs a signed comparison between the destination operand and the source operand. The jump will happen if the destination operand is greater than or equal to the source operand.

Flags Affected: None

Example

```
CMP EAX, 02H        ; Compare the EAX value with 02H by subtracting 02H from EAX

JGE MEM_LO          ; The program will jump to the memory location if EAX ≥ 02H
```

JA

Instruction Format:

JA DESTINATION_OPERAND

Meaning:

This instruction is the same as JG. As **Jump if above** is a conditional jump instruction, it is commonly used after the CMP instruction where it perform an unsigned comparison between the destination operand and the source operand. The jump will happen if the destination operand is above the source operand.

Flags Affected: None

Example

CMP EAX, 04H ; Compare the EAX value with 04H by subtracting 04H from EAX

JA MEM_LO ; The program will jump to the memory location if EAX is above 04H

JAE

Instruction Format:

JAE DESTINATION_OPERAND

Meaning:

This instruction is the same as JGE. As **Jump if above or equal** is a conditional jump instruction, it is commonly used after the CMP instruction where it performs an unsigned comparison between the destination operand and the source operand. The jump will happen if the destination operand is above or equal to the source operand.

Flags Affected: None

Example

CMP EAX, 04H ; Compare the EAX value with 04H, by subtracting 04H from EAX

JAE MEM_LO ; The program will jump to the memory location if EAX is above or equal to 04H

JL

Instruction Format:

JL DESTINATION_OPERAND

Meaning:

As **Jump if Less** is a conditional jump instruction, it is commonly used after the CMP instruction where it performs a signed comparison between the destination operand and the source operand. The jump will happen if the destination operand is less than the source operand.

Flags Affected: None

Example

CMP EAX, 02H ; Compare the EAX value with 02H by subtracting 02H from EAX

JL MEM_LO ; The program will jump to the memory location if EAX < 02H

JLE

Instruction Format:

```
JLE DESTINATION_OPERAND
```

Meaning:

The **Jump if Less or equal** is a conditional jump instruction. It is commonly used after the CMP instruction where it performs a signed comparison between the destination operand and the source operand. The jump will happen if the destination operand is less than or equal to the source operand.

Flags Affected: None

Example

```
CMP EAX, 02H      ; Compare the EAX value with 02H by subtracting 02H from EAX

JLE MEM_LO        ; The program will jump to the memory location if EAX ≤ 02H
```

JB

Instruction Format:

JB DESTINATION_OPERAND

Meaning:

This instruction is the same as JL. As **Jump if below** is a conditional jump instruction, it is commonly used after the CMP instruction where it performs an unsigned comparison between the destination operand and the source operand. The jump will happen if the destination operand is below the source operand.

Flags Affected: None

Example

CMP EAX, 04H ; Compare the EAX value with 04H by subtracting 04H from EAX

JB MEM_LO ; The program will jump to the memory location if EAX is below 04H

JBE

Instruction Format:

JBE DESTINATION_OPERAND

Meaning:

This instruction is the same as JLE. As **Jump if below or equal** is a conditional jump instruction, it is commonly used after the CMP instruction to perform an unsigned comparison between the destination operand and the source operand. The jump will happen if the destination operand is below or equal to the source operand.

Flags Affected: None

Example

CMP EAX, 04H ; Compare the EAX value with 04H by subtracting 04H from EAX

JBE MEM_LO ; The program will jump to the memory location if EAX is below or equal to ; 04H

JO

Instruction Format:

JO DESTINATION_OPERAND

Meaning:

This is a conditional jump instruction which jumps to the destination operand if the **overflow flag (OF)** is set to 1. If OF is set to 0, then no action will be taken and the next instruction following it will be executed.

Flags Affected: None

Example

ADD AL, BL ; Add the signed bytes in AL and BL

JO MEM_LOC ; The program will jump to the memory location if OF=1, due to addition ; above

JS

Instruction Format:

JS DESTINATION_OPERAND

Meaning:

This is a conditional jump instruction which jumps to the destination operand if the sign flag (SF) is set to 1. If SF is set to 0, then no action will be taken and the next instruction following it will be executed.

Flags Affected: None

Example

```
ADD AL, BL          ; Add signed bytes in AL and BL

JS MEM_LOC    ; The program will jump to the memory location
if SF=1, due to addition ; above
```

JECXZ

Instruction Format:

JECXZ DESTINATION_OPERAND

Meaning:

This is a conditional jump instruction which jumps to the destination operand if the ECX register is equal to 0.

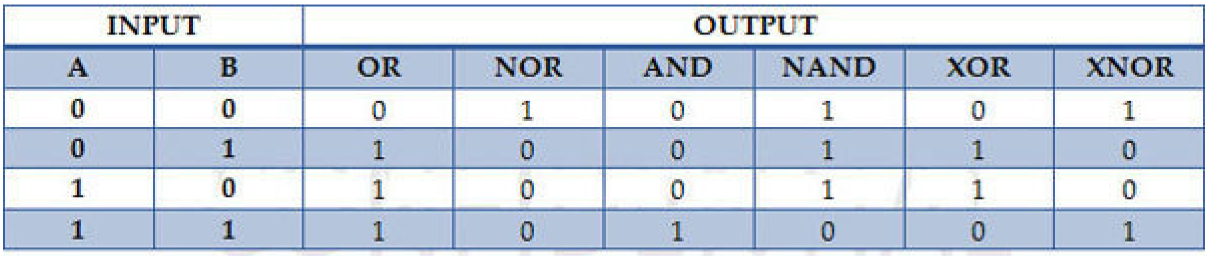
Flags Affected: None

Example

JECXZ MEM_LOC ; The program will jump to the memory location if ECX = 0

Bit Manipulation Instructions

This includes instructions used in bit manipulation operations. Before we get started with the instructions, let’s walk through the truth table:



INPUT		OUTPUT					
A	B	OR	NOR	AND	NAND	XOR	XNOR
0	0	0	1	0	1	0	1
0	1	1	0	0	1	1	0
1	0	1	0	0	1	1	0
1	1	1	0	1	0	0	1

Figure 4.4: *Truth table*

BSWAP

Instruction Format:

BSWAP REGISTER32

Meaning:

This instruction changes the byte order of the register from Big-endian to Little-endian or from Little-endian to Big-endian.

Flags Affected: None

Example

MOV EAX, 87654321H ; EAX is 87654321H

BSWAP EAX ; EAX will become 21436587H

AND

Instruction Format:

AND DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

This instruction performs the logical AND operation between the destination operand and the source operand. The result of the AND operation is saved in the destination operand.

Flags Affected: CF OF PF SF ZF

Example

```
MOV EAX, 87654321H      ; EAX is 87654321H

MOV EBX, 21436587H      ; EBX is 21436587H

AND EAX, EBX                    ; EAX will become 0x01414101
```

NOT

Instruction Format:

NOT DESTINATION_OPERAND

Meaning:

This instruction inverts all the bits of the destination operand. All the 1s will become 0 and all the 0s will become 1 by taking one's complement. One’s complement is obtained by toggling all the bits.

Flags Affected: None

Example

```
MOV EAX, 12121212H      ; EAX is 12121212H

NOT EAX                 ; EAX will become EDEDEDEDH
```

OR

Instruction Format:

OR DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

This instruction performs the logical OR operation between the destination operand and the source operand. The result of the logical OR operation is saved in the destination operand.

Flags Affected: CF OF PF SF ZF

Example

```
MOV EAX, 87654321H      ; EAX is 87654321H

MOV EBX, 21436587H      ; EBX is 21436587H

OR EAX, EBX                     ; EAX will become 0xA76767A7
```

XOR

Instruction Format:

XOR DESTINATION_OPERAND SOURCE_OPERAND

Meaning:

This instruction performs the exclusive-OR operation between the destination operand and the source operand. The result of the XOR operation is saved in the destination operand.

Flags Affected: CF OF PF SF ZF

Example

```
MOV EAX, 87654321H      ; EAX is 87654321H

XOR EAX, EAX                    ; EAX will become 0x00000000
```

RCL

Instruction Format:

RCL DESTINATION_OPERAND COUNT

Meaning:

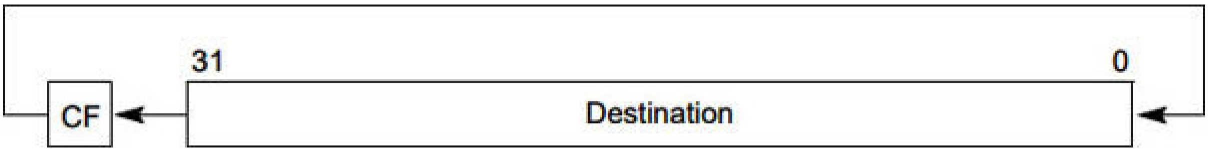


Figure 4.5: RCL

Rotate through Carry Left rotates the bits n (count) times in the destination operand from the right to left through the CF Flag. On every rotation, **Most Significant Bit** is moved to the CF flag and the CF flag enters into **Least significant bit**

Flags Affected: CF OF

Example

MOV EAX, 01H ; EAX is 00000001H and CF=0

```
RCL EAX, 2                ; RCL EAX bits 2 times, EAX will become 0x00000004
```

RCR

Instruction Format:

RCR DESTINATION_OPERAND COUNT

Meaning:

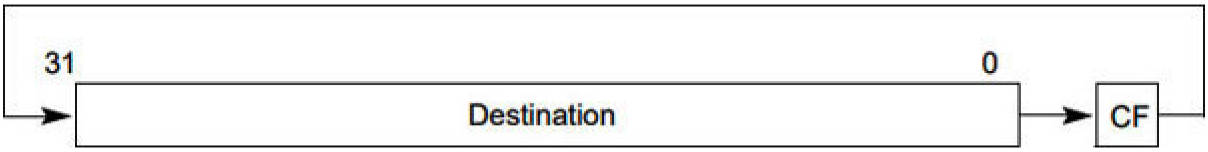


Figure 4.6: RCR

Rotate through carry right (RCR) rotates the bits n (count) times in the destination operand from the left to right through the CF flag. On every rotation, LSB is moved to the CF flag and the CF flag enters into MSB.

Flags Affected: CF OF

Example

MOV EAX, 01H ; EAX is 00000001H and CF=0

```
RCR EAX, 2                ; RCR EAX bits 2 times, EAX will become 0x80000000
```

ROL

Instruction Format:

ROL DESTINATION_OPERAND COUNT

Meaning:

Figure 4.7: ROL

Rotate Left rotates the bits n (count) times in the destination operand from the right to left and the CF flag will have the value of the last bit rotated.

Flags Affected: CF OF

Example

MOV EAX, 40000001H ; EAX is 40000001H and CF=0

```
ROL EAX, 2                            ; ROL EAX bits 2 times, EAX
will become 0x00000005 and ; CF will set to 1
```

ROR

Instruction Format:

ROR DESTINATION_OPERAND COUNT

Meaning:

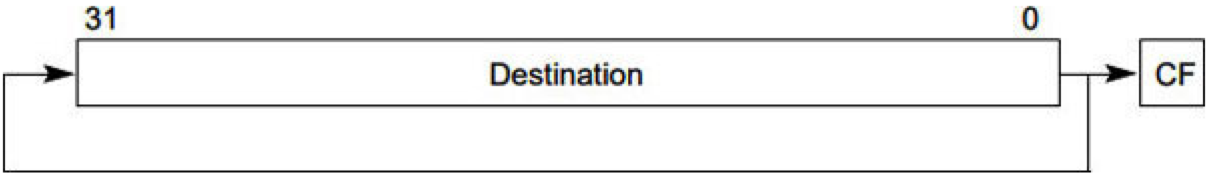


Figure 4.8: ROR

Rotate Right rotates the bits n (count) times in the destination operand from the left to right and the CF flag will have the value of the last bit rotated.

Flags Affected: CF OF

Example

MOV EAX, 40000001H ; EAX is 40000001H and CF=1

ROR EAX, 2 ; ROR EAX bits 2 times, EAX will become 0x50000000 and ; CF will set to 0

SHR

Instruction Format:

SHR DESTINATION_OPERAND COUNT

Meaning:

To understand the bit shifting concept, refer to it in the

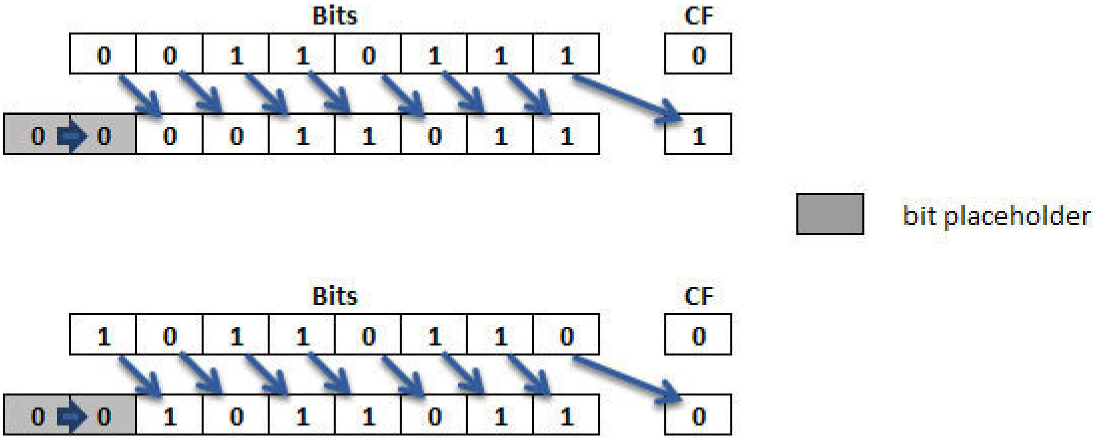


Figure 4.9: SHR

Shift Logical Right shifts the bits n (count) times in the destination operand from the left to right and the CF flag will have the value of the last bit shifted out. As it is logical shifting, the bit placeholder on every shift count is set to 0.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

```
MOV EAX, 40000001H        ; EAX is 40000001H and CF=0

SHR EAX, 2                            ; SHR EAX bits 2 times, EAX
will become 0x10000000 and ; CF will set to 0
```

SHL

Instruction Format:

SHL DESTINATION_OPERAND COUNT

Meaning:

As said above, to understand bit shifting concept, refer bit shifting section in

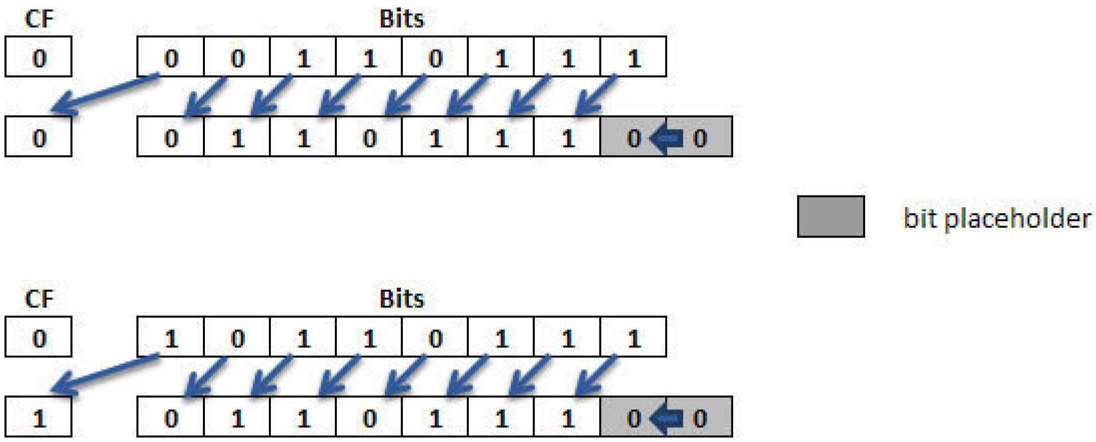


Figure 4.10: *SHL*

Shift Logical Left shifts the bits n (count) times in the destination operand from the right to left and the CF flag will have the value of the last bit shifted out. As it is logical shifting, the bit placeholder on every shift count is set to 0.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

```
MOV EAX, 40000001H          ; EAX is 40000001H and CF=0

SHL EAX, 2                          ; SHL EAX bits 2 times, EAX
will become 0x00000004 and ; CF will set to 1
```

SAR

Instruction Format:

SAR DESTINATION_OPERAND COUNT

Meaning:

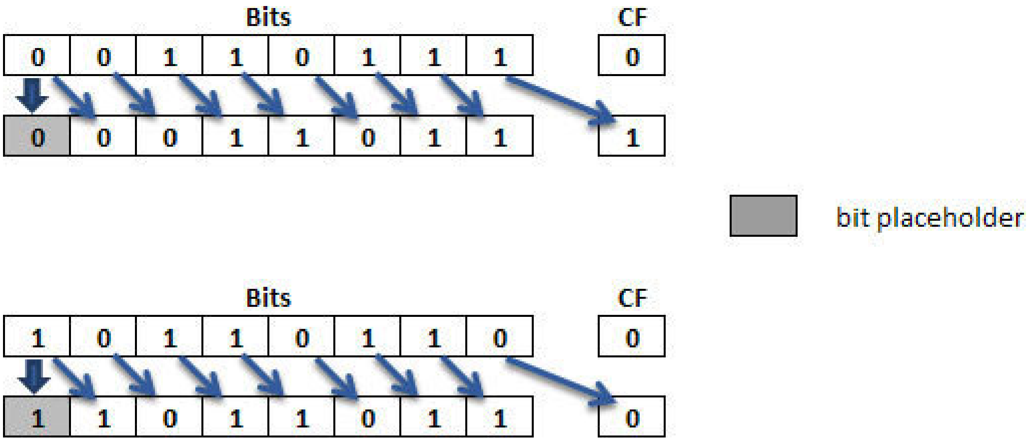


Figure 4.11: SAR

Shift Arithmetic Right shifts the bits n (count) times in the destination operand from the left to right and the CF flag will have the value of the last bit shifted out. As it is arithmetic shifting, the bit placeholder is determined by MSB.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

```
MOV EAX, 40000001H          ; EAX is 40000001H and CF=1

SAR EAX, 2                               ; SAR EAX bits 2 times, EAX
will become 0x10000000 and ; CF will set to 0
```

SAL

Instruction Format:

SAL DESTINATION_OPERAND COUNT

Meaning:

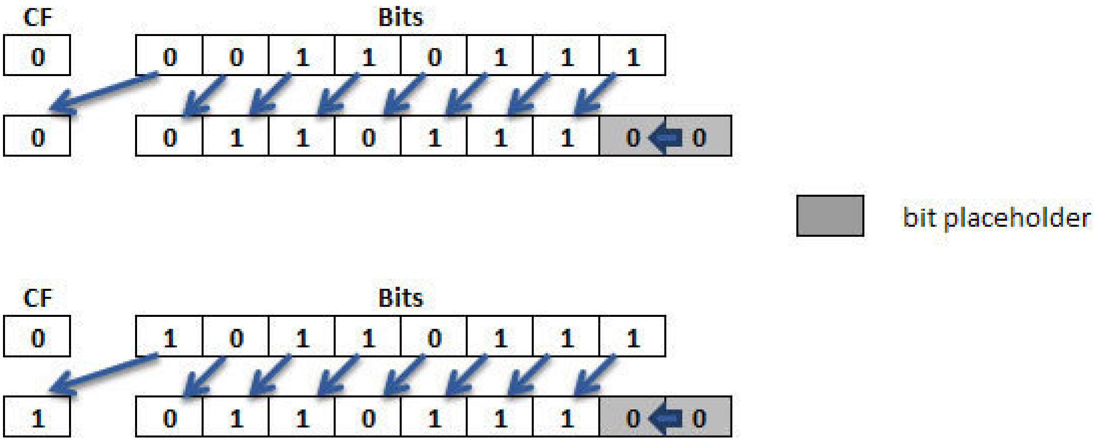


Figure 4.12: SAL

Shift Arithmetic Left shifts the bits n (count) times in the destination operand from the right to left and the CF flag will have the value of the last bit shifted out. As it is arithmetic shifting, the bit placeholder at LSB will always be 0.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

```
MOV EAX, 40000001H          ; EAX is 40000001H and CF=0

SAL EAX, 2                              ; SAL EAX bits 2 times, EAX
will become 0x00000004 and ; CF will set to 1
```

SHLD

Instruction Format:

SHLD DESTINATION_OPERAND, SOURCE_OPERAND, COUNT

Meaning:

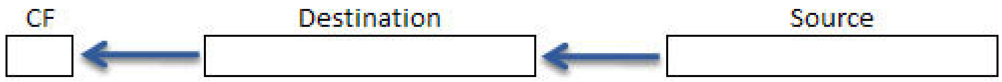


Figure 4.13: *SHLD*

Shift Left Double shifts the bits n (count) times in the destination operand from the right to left and the empty bit placeholders in the destination operand are filled by the bits shifted out of the source operand. The CF flag will have the value of the last bit shifted out of the destination operand. The source operand will not be modified.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

MOV EAX, 40000001H ; EAX is 40000001H and CF=0

```
MOV EBX, 50000001H        ; EBX is 50000001H and CF=0

SHLD EAX, EBX, 2            ; EAX = 0x00000005 and EAX =
0x50000001

                                            ; CF will set to 1
```

SHRD

Instruction Format:

SHRD DESTINATION_OPERAND, SOURCE_OPERAND, COUNT

Meaning:

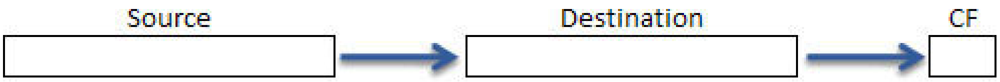


Figure 4.14: *SHRD*

Shift Right Double shifts the bits n (count) times in the destination operand from the left to right and the empty bit placeholders in the destination operand are filled by the bits shifted out of the source operand. The CF flag will have the value of the last bit shifted out of the destination operand. The source operand will not be modified.

Flags Affected: CF OF PF SF ZF (AF undefined)

Example

MOV EAX, 40000001H ; EAX is 40000001H and CF=1

```
MOV EBX, 50000001H        ; EBX is 50000001H and CF=1

SHRD EAX, EBX, 2               ; EAX = 0x50000000 and EAX =
0x50000001

                                                       ; CF will set to 0
```

Processor Control Instructions

This includes instructions used in controlling the processor operations.

CLC

Instruction Format:
CLC

Meaning:

Clear the Carry Flag This instruction clears the Carry Flag.

Flags Affected: CF

Example

MOV EAX, 40000001H ; EAX is 40000001H and CF=0

SAL EAX, 2 ; SAL EAX bits 2 times, EAX will become 0x00000004 and ; CF will set to 1

CLC ; Clear the CF flag, CF=0

CLD

Instruction Format:

CLD

Meaning:

Clear the Direction Flag This instruction clears the Direction Flag to 0.

Flags Affected: CF

CLI

Instruction Format:

CLI

Meaning:

Clear the Interrupt Flag This instruction clears the Interrupt Flag to 0. Once the IF flag is reset, the processor will not respond to the interrupt signal.

Flags Affected: CF

CMC

Instruction Format:

CMC

Meaning:

Complement Carry This instruction inverts the Carry Flag.

Flags Affected: CF

Example

```
MOV EAX, 40000001H          ; EAX is 40000001H and CF=0

SAL EAX, 2                  ; SAL EAX bits 2 times, EAX will become 0x00000004 and ; CF will set to 1

CMC                         ; Toggle CF flag from 1 to 0, CF=0
```

ESC

Instruction Format:

ESC OPCODE SOURCE_OPERAND

OPCODE = D8 to DF
SOURCE_OPERAND = REGISTER OR MEMORY

Meaning:

Escape to Floating Point Coprocessor This instruction passes the instructions to the coprocessor, also called floating point or math coprocessor. The microprocessor fetches the instruction bytes and the coprocessor also fetches these instruction bytes from the data bus and queues them. All the normal microprocessor instructions are treated as NOP by the coprocessor but when the ESC instruction is fetched by the microprocessor, the coprocessor decodes the instruction to carry out the action. When the ESC instruction is executed, the microprocessor provides the memory address otherwise perform NOP.

Flags Affected: None

LOCK

Instruction Format:

LOCK: [INSTRUCTION]

Meaning:

LOCK is the not an instruction but an instruction prefix. When LOCK is used as a prefix in front of any instruction, the processor LOCK pin is activated or also called asserted. When the LOCK pin is activated by the LOCK instruction, the external bus master and other peripherals are disabled until the instruction after LOCK is executed. So, LOCK is used in front of critical instructions that are to be executed without any disturbance to bus master (Bus Master is a program that controls the bus on which the address and control signals flow) and system bus (System Bus is a common term for address bus and data bus).

Flags Affected: None

Example

LOCK: MOV EAX, EBX ; Activate the LOCK pin of the processor for the MOV instruction ; execution

NOP

Instruction Format:

NOP

Meaning:

No Operation This instruction does nothing. It is used to eat up the processor time and memory.

Flags Affected: None

Example

```
NOP     ; Do nothing
```

STC

Instruction Format:

STC

Meaning:

Set the Carry Flag This instruction sets the Carry Flag to 1.

Flags Affected: CF

Example

```
CLC    ; Clear the CF flag, CF=0

STC    ; Set the CF flag, CF=1
```

STD

Instruction Format:

STD

Meaning:

Set the Direction Flag This instruction sets the Direction Flag to 1.

Flags Affected: DF

STI

Instruction Format:

STI

Meaning:

Set the Flag This instruction sets the Interrupt Flag to 1. As already explained in above instructions, when the IF flag is enabled, the interrupt event or signal will cause the processor to interrupt the program execution. Whenever the interrupt occurs, the processor completes the current set of instructions and then starts the **interrupt service routine** or interrupt handler. ISR is a routine which contains a set of instructions to handle specific interrupts. This ISR tells the processor what to do when an interrupt occurs.

The IRET instruction after ISR returns the execution back to the interrupted program.

Flags Affected: IF

String Instructions

This includes instructions used in handling string operations.

CMPS/CMPSB/CMPSW

Instruction Format:

This instruction can be visualized as a combination of CMP for compare + S for String + B for Byte or W for WORD or D for DWORD

CMPSB SOURCE, DESTINATION
CMPSW SOURCE, DESTINATION
CMPSD SOURCE, DESTINATION

Meaning:

This instruction is used to compare two strings where the source string is pointed by ESI and the destination string is pointed by EDI. When CMPSB is used, the comparison is done between every byte. When CMPSW is used, the comparison is done between every word. When CMPSD is used, the comparison is done between every DWORD. By comparison, it means subtracting a byte/word/dword pointed by the ESI and EDI registers, just as we studied in the CMP instruction.

The CMPS instruction is used with the prefixes REPE/REPZ, which means repeat the comparison until ECX=0 or ZF=0. We will cover REPE/REPZ later.

If Direction flag, DF=0, then ESI and EDI are incremented by 1 for byte, 2 for word, and 4 for dword after each move.

If Direction flag, DF=1 then ESI and EDI are decremented by 1 for byte, 2 for word, and 4 for dword after each move.

Flags Affected: AF,CF,OF,PF,SF,ZF

Example

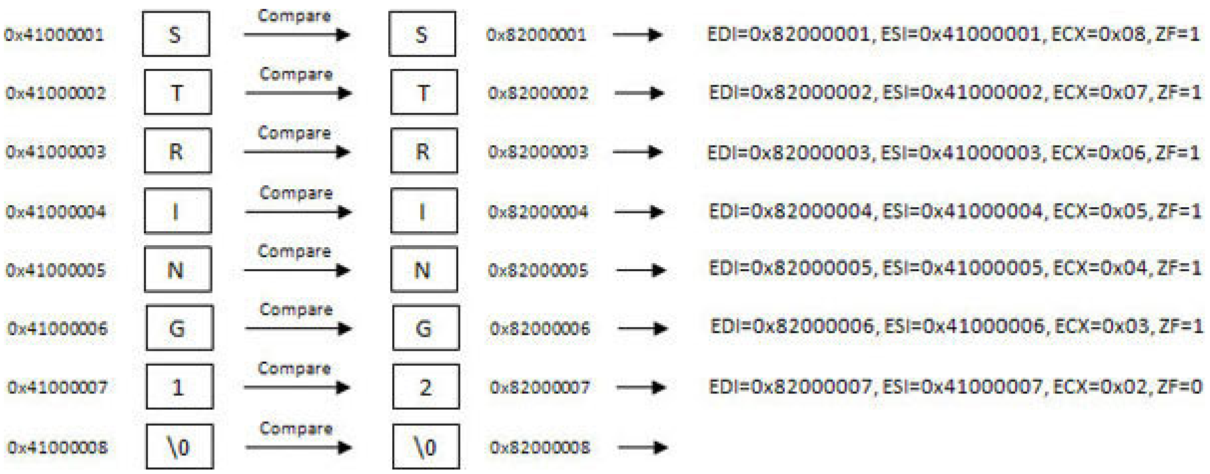


Figure 4.15: *CMPSB example*

MOV ESI, 0x41000001 ; ESI pointing to STRING1

MOV EDI, 0x82000001 ; ESI pointing to STRING2

MOV ECX, 0x08 ; ECX is initialized to 0x08

```
CLD                            ; Clear direction flag, DF=0

REPE CMPSB                     ; Compare two strings until ECX=0 or ZF=0
```

IN/INSB/INSW/INSD

This instruction can be visualized as a combination of IN for Input + S for String + B for Byte or W for WORD or D for DWORD.

Meaning:

There are two types of instructions used to transfer data between the processor input/output ports and the peripheral devices. These two types of instruction are as follows:

Register input/output These instructions move the data between the processor I/O port and the register.

Block (or string) input/output These instructions move blocks of data between the processor I/O port and the memory.

Register input instruction

IN DESTINATION SOURCE

Where, SOURCE is the port address

And DESTNATION is the register, EAX (32-bit) register or AX (16-bit) register or AL (8-bit) register. The destination register length determines the amount of data to be read from the input port, which can be byte, word, or dword. The destination can be some other general purpose register.

IN EAX, 80 ; It will read dword (32-bits) from port 80 and store it in EAX.

IN AX, 80 ; It will read word (16-bits) from port 80 and store it in AX.

Block (or string) input instruction

INSB/INSW/INSD

Where the processor input port address is specified in the DX register and the destination is the memory location pointed by the EDI register.

The INSB instruction moves a byte (8-bits) data from the input port address specified in the DX register to the memory location pointed by the EDI register. To transfer blocks of data between the processor input port and the memory, the INSB instruction is used with the repeat prefix REP. After every byte transfer between the input port and the memory pointed by EDI, EDI is incremented (when DF=0) or decremented (when DF=1) by 1.

The INSW instruction moves a word (16 bits) data from the input port address specified in the DX register to the memory location pointed by the EDI register. To transfer blocks of data between the processor input port and the memory, the INSW instruction is used with the repeat prefix REP. After every word transfer between the input port and the memory pointed by EDI, EDI is incremented (when DF=0) or decremented (when DF=1) by 2.

The INSD instruction moves a dword (32 bits) data from the input port address specified in the DX register to the memory location pointed by the EDI register. To transfer blocks of data between the processor input port and the memory, the INSD instruction is used with the repeat prefix REP. After every dword transfer between the input port and the memory pointed by EDI, EDI is incremented (when DF=0) or decremented (when DF=1) by 4.

Flags Affected: None

Example

```
MOV DX, PORT_ADDRESS     ; Move the input port address to
the DX register



MOV EDI, offset STR                ; Move the memory offset
for string to the EDI register


INSW                                                  ; Move a
word (16 bits) from the input port address


                                                              ;
specified in the DX register to the memory location pointed
```

; by EDI

register

OUT/OUTSB/OUTSW/OUTSD

This instruction can be visualized as a combination of OUT for Output + S for String + B for Byte or W for WORD or D for DWORD.

Meaning:

The same concept from input instruction is reiterated here. There are two types of instructions used to transfer data between the processor input/output ports and the peripheral devices. These two types of instruction are as follows:

Register input/output These instructions move the data between the processor I/O port and the register.

Block (or string) input/output These instruction move blocks of data between the processor I/O port and the memory.

Register output instruction

OUT DESTINATION SOURCE

SOURCE is the register, EAX (32 bit) register or AX (16 bit) register or AL (8 bit) register. The source register length determines the amount of data to be output from the source register to the port, which can be byte, word, or dword. The source can also be some other general purpose register.

And DESTNATION is the port address

OUT 80, EAX ; It will output dword (32-bits) from EAX to port 80.

OUT 80, AX ; It will output word (16-bits) from AX to port 80.

Block (or string) output instruction

OUTSB/OUTSW/OUTSD

Where the processor output port address is specified in the DX register and the source is the memory location pointed by the ESI register.

The OUTSB instruction outputs a byte (8-bits) data from the memory location pointed by the ESI register to the output port address specified in the DX register. To transfer blocks of data between the memory and the processor output port, the OUTSB instruction is used with the repeat prefix REP. After every byte (8-bits) output from the memory pointed by ESI to the output port address specified in the DX register, ESI is incremented (when DF=0) or decremented (when DF=1) by 1.

The OUTSW instruction outputs a word (16-bits) data from the memory location pointed by the ESI register to the output port address specified in the DX register. To transfer blocks of data between the memory and the processor output port, the OUTSW instruction is used with the repeat prefix REP. After every word (16-bits) output from the memory pointed by ESI to the output port address specified in the DX register, ESI is incremented (when DF=0) or decremented (when DF=1) by 2.

The OUTSD instruction outputs a dword (32-bits) data from the memory location pointed by the ESI register to the output port address specified in the DX register. To transfer blocks of data between the memory and the processor output port, the OUTSD instruction is used with the repeat prefix REP. After every dword (32-bits) output from the memory pointed by ESI to the output port address specified in the DX register, ESI is incremented (when DF=0) or decremented (when DF=1) by 4.

Flags Affected: None

Example

```
MOV ESI, offset STR                  ; Move memory offset
for string to the ESI register

MOV DX, PORT_ADDRESS          ; Move Output Port address to
the DX register
```

```
OUTSW                                        ; Output a
word (16 bits) from memory location pointed

                                                  ; by
ESI register to output Port address in DX register
```

LODS/LODSB/LODSW/LODSD

Instruction Format:

LODSB
LODSW
LODSD

Meaning:

Load String loads the string pointed by the ESI register to the EAX register.

Load String Byte loads a byte of string pointed by the ESI register to the AL register.

Load String Word loads a word of string pointed by the ESI register to the AX register.

Load String DWORD loads a dword of string pointed by the ESI register to the EAX register.

ESI is incremented (when DF=0) or decremented (when DF=1) by 1 for **Load String Byte** 2 for **Load String Word** and 4 for **Load String DWORD**

Flags Affected: None

Example

```
CLD                                  ; Clear Direction flag,
DF=0

MOV ESI, Offset STR     ; Move memory offset of string to ESI
register

LODSB                              ; Loads a byte of the string
pointed by ESI register to AL

                                          ; register
```

STOS/STOSB/STOSW

Instruction Format:

STOSB
STOSW
STOSD

Meaning:

Store String stores the string from the EAX register to the memory location pointed by the EDI register.

Store String Byte stores a byte of string from the AL register to the memory location pointed by the EDI register.

Store String Word stores a word of string from the AX register to the memory location pointed by the EDI register.

Store String DWORD stores a dword of string from the EAX register to the memory location pointed by the EDI register.

EDI is incremented (when DF=0) or decremented (when DF=1) by 1 for **Store String Byte** 2 for **Store String Word** and 4 for **Store String DWORD**

Flags Affected: None

Example

```
CLD                                   ; Clear Direction flag, DF=0

MOV ESI, Offset STR    ; Move memory offset of string to ESI
register

STOSB                                 ; Stores a byte of the string
from AL register to memory

                                             ; pointed by EDI
register
```

SCAS/SCASB/SCASW

Instruction Format:

SCASB
SCASW
SCASD

Meaning:

Scan String scans the string in the memory location pointed by the EDI register and compares (or subtracts) it with the contents of the EAX register. The result of the comparison or subtraction is discarded and the status flags are updated accordingly.

Scan String Byte scans a byte of string in the memory location pointed by the EDI register and compares it with the contents of the AL register.

Scan String Word scans a word of string in the memory location pointed by the EDI register and compares it with the contents of the AX register.

Scan String DWORD scans a dword of string in the memory location pointed by the EDI register and compares it with the contents of the EAX register.

EDI is incremented (when DF=0) or decremented (when DF=1) by 1 for **Scan String Byte** 2 for **Scan String Word** and 4 for **Scan String DWORD**

Flags Affected: None

Example

```
MOV ECX, 100                    ; Scan a string of 100 characters

MOV EDI, offset STR    ; Move memory offset of string to ESI register


MOV AL, 0x20                    ; Scanning string for space character, space=0x20

REPNE SCASB                     ; Repeat until ECX=0 or ZF=1
```

MOVS/MOVSB/MOVSW

Instruction Format:

MOVSB
MOVSW
MOVSD

Meaning:

Move String moves the string in the memory location pointed by the ESI register to the memory location pointed by the EDI register.

Move String Byte moves a byte (8-bits) of string in the memory location pointed by the ESI register to the memory location pointed by the EDI register.

Move String Word moves a word (16-bits) of string in the memory location pointed by the ESI register to the memory location pointed by the EDI register.

Move String DWORD moves a dword (32-bits) of string in the memory location pointed by the ESI register to the memory location pointed by the EDI register.

ESI and EDI are incremented (when DF=0) or decremented (when DF=1) by 1 for **Move String Byte** 2 for **Move String Word** and 4 for **Move String DWORD**

Flags Affected: None

Example

```
MOV ESI, SRC_STR    ; Move source string location in ESI

MOV EDI, DST_STR    ; Move destination string location in EDI

MOV ECX, 05H              ; Initialize ECX to 0x05, which is
string length


CLD                                    ; Clear the direction flag,
DF=0

REP MOVSB                 ; Moves the string of length 5 bytes
from src to dest
```

REP

Instruction Format:

REP

Meaning:

Repeat is not an instruction. It is a prefix which is used before string instructions. When REP is used before a string instruction, it will repeat the instruction until ECX counter becomes 0. On every execution of the instruction after REP, the ECX counter is decremented by 1 until ECX=0.

Flags Affected: Depends on the instruction used after REP.

Example:

```
MOV DWORD PTR DS:[0x011E8000], 0x41424344        ;write ABCD
to 0x011E8000


                                         ; memory location, ABCD will
be
```

```
                                        ; written in reverse order (little

                                        ; endian)

MOV ESI,
0X11E8000
   ; Move source string location in ESI

MOV EDI,
0X11E8010
  ; Move source string location in EDI

MOV ECX,
0X05
    ; Initialize ECX to 0x05, to run iteration 5 times


CLD
                                 ; Clear the direction flag, DF=0

REP
MOVSB
                     ; Moves the ABCD from src


                                      ; to dest, until ECX=0
```

REPE/REPZ

Instruction Format:

REPE
REPZ

Meaning:

Repeat if Equal and **Repeat if Zero** cause the preceding string instruction to repeat until ECX=0 or Zero flag (ZF) = 0. These prefixes are used with the CMPS and SCAS instructions.

Flags Affected: Depends on the instruction used after REPE/REPZ instruction.

REPNE/REPNZ

Instruction Format:

REPNE
REPNZ

Meaning:

Repeat if Not Equal and **Repeat if Not Zero** cause the preceding string instruction to repeat until ECX=0 or ZF=1. These prefixes are used with the CMPS and SCAS instructions.

Flags Affected: Depends on instruction used after REPNE/REPNZ instruction.

Conclusion

In this chapter, we studied the explanation of major assembly instructions used in reverse engineering. We also covered the different types of instructions for stack, data transfer, arithmetic, program execution, branching, bit manipulation, processor control, and string. Examples with some instructions were also covered to elaborate the working of the instructions.

In the next chapter, we will talk about some stack based instructions in detail and understand the concept of code calling conventions. This concept is important from the reverse engineering point of view and you will come across this quite often in implementing reverse engineering.

CHAPTER 5

Types of Code Calling Conventions

In *Chapter 2, Understanding Architecture of x86 Machines* , we understood the concept of stack. Several things happen in the background when we call a function. Control is transferred to the new function in a way that the stack frame is allocated for local variables and parameters are passed for a callee to understand. On return, the return address is placed on the stack so that the caller can find it and a stack clean-up is performed as well. Imagine if the callee and the caller both clean up the stack, then it will create a disastrous situation where the stack is cleaned twice both by callee and caller. So, understanding the difference in code calling conventions becomes important. As there are many variants of CPUs, some CPUs have a strict protocol on how this is performed. But when we talk about the x86 architecture, it is all flexible where a programmer decides on how to call the methods. This is the reason that led to different calling conventions.

When you write C/C++ code wherein you use shared libraries, code calling convention becomes important, as the code you are interacting is beyond your control. If you are a programmer who writes C/C++ code, then as a programmer, you don't have to worry as the compiler takes care of the calling convention for you. The compiler generally takes the default code convention automatically based on the language. In this chapter, we will understand the difference in code calling conventions.

Structure

In this chapter, we will cover the following topics:

Understand the types of calling conventions

Concept behind different calling conventions

Objective

After studying this chapter, you will be able to differentiate between different assembly codes with respect to the calling convention. If in case you receive some assembly code then, by going over the assembly listing, you will be able to evaluate the calling convention used. To understand the different code calling conventions, we will take up a pseudo code to run over the concept behind each calling convention in detail.

Understand types of calling conventions

To understand calling conventions, let’s take a basic pseudo code. In this code, function **funcA** calls another function In this case, **funcA** is called the **caller** and **funcB** is called the

```
FuncA()
{
Arg1;
Arg2;
FuncB(Arg1, Arg2);
}
```

Now when **FuncB** is called, it is called with 2 arguments When this code is compiled with different compilers, it will generate different assembly codes. Calling convention is this set of rules that specify how C or C++ functions are converted into an assembly code. So, calling convention basically defines:

How arguments are passed to the function

How functions return values

How the caller calls the callee

How stack is managed when one function calls another

How stack is cleared

All these are defined by the calling convention method the compiler uses. In C/C++ language, there are three types of calling conventions majorly used: and We will walk through all these calling conventions one by one.

CDECL

CDECL can also be read as C Declaration. When CDECL calling convention is used:

Arguments are passed from the right to left order. In the same pseudo code, right to left order means **Arg2** is first pushed on the stack and then **Arg1** is pushed on the stack.

```
FuncA()
{
Arg1;
Arg2;
FuncB(Arg1, Arg2);
}
```

The function return value is passed into the EAX register.

Calling function, the caller cleans the stack.

STDCALL

STDCALL stands for Standard Call. This calling convention is defined by Microsoft as a standard calling convention for Win32 API. When STDCALL is used:

The first point is the same as CDECL. Arguments are passed from the right to left order.

The function return value is passed into the EAX register. This point is also the same as CDECL.

This point differs from CDECL where called function, callee cleans the stack.

FASTCALL

The main difference between CDECL/STDCALL and FASTCALL is that the initial arguments are not pushed on stack but rather passed in the registers. Keeping data in registers is faster than in memory, hence it is named FASTCALL. In FASTCALL, when the calling convention is used:

The first two or three parameters are passed in the registers EDX, ECX, or EAX. Additional parameters are passed on to the stack. Arguments are passed from the right to left order.

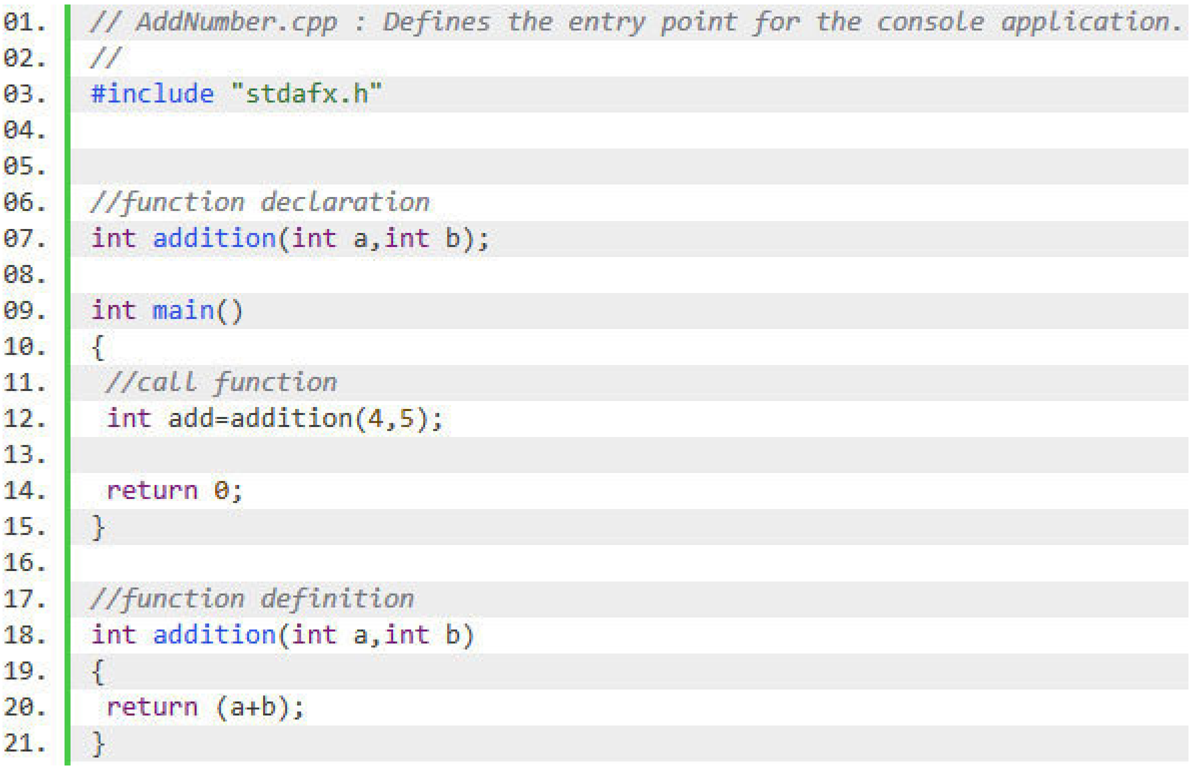
The function return value is passed into the EAX register.

Calling function, the caller cleans the stack if needed.

Concept behind different calling conventions

Now to understand the concept behind different calling conventions, we will write a simple code in C/C++. Then using **cl.exe** (VS compiler), we will compile the code with different calling conventions and understand the difference between the assembly codes generated.

The following image shows a simple C/C++ code that will add numbers:



```
// AddNumber.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"


//function declaration
int addition(int a,int b);

int main()
{
 //call function
 int add=addition(4,5);

 return 0;
}

//function definition
int addition(int a,int b)
{
 return (a+b);
}
```

Figure 5.1: AddNumber.cpp

The **AddNumber.cpp** code is a simple code of adding two numbers. A few points to notice in the preceding program are as follows:

The **main** function is the entry point of the program.

The **main** function is calling an **addition** function. So, the **main** function is the caller and the **addition** function is the callee.

The local variable defined in the **main** function and the **addition** function are of type integer.

Two parameters are passed to the callee, which are 4 and 5.

After writing the code, we will compile the code with **cl.exe** and use different switches that will force the compiler to change the calling convention.

Use this switch for the CDCEL calling convention.

Use this switch for the STDCALL calling convention.

Use this switch for the FASTCALL calling convention.

We will take up each calling convention one by one and understand how they differ from each other.

CDECL

We will compile the **AddNumber.cpp** code with no optimization and with CDECL calling convention. We will use the **/Gd** switch for this. Run the commands given below on the Windows command prompt to set the environment for **cl.exe** (VS compiler) and then compile the code with the following switches.

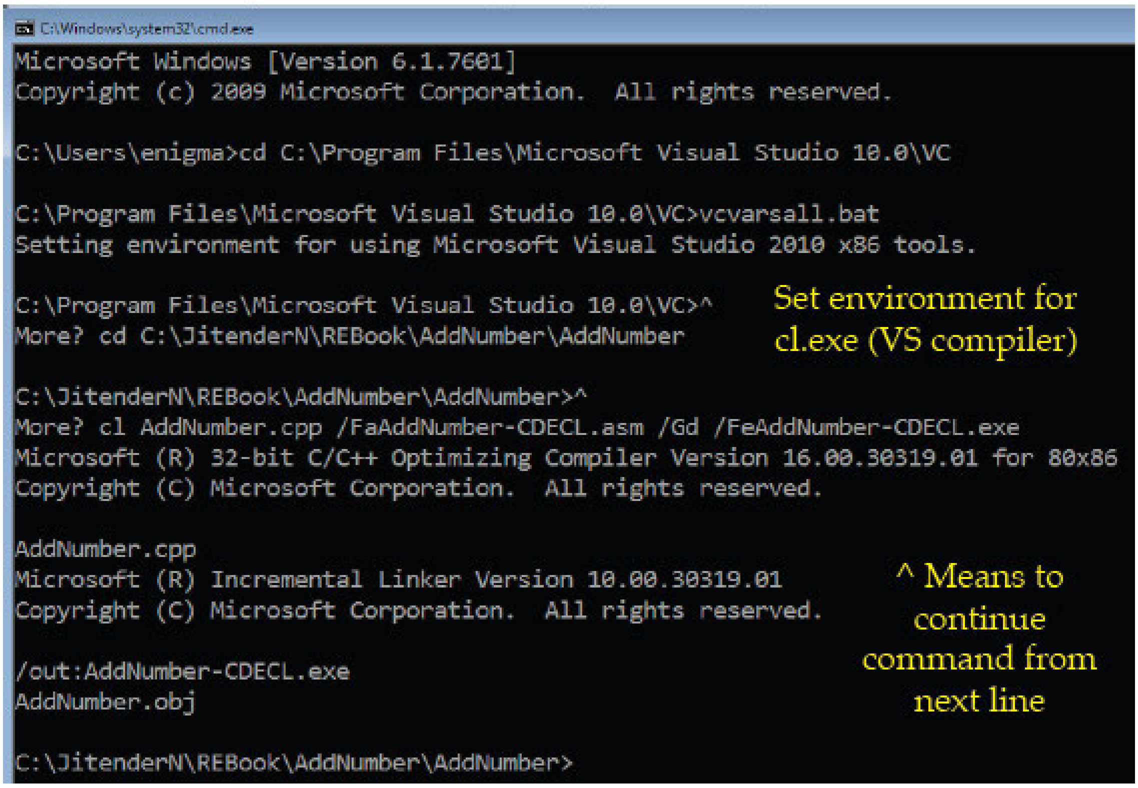
Use this switch for the CDCEL calling convention.

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



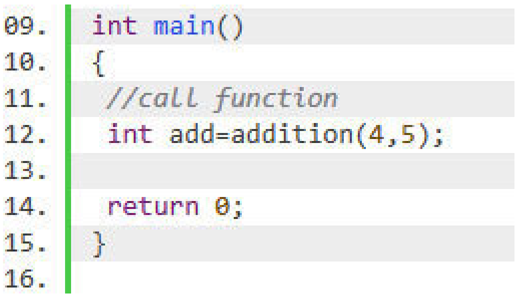
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\AddNumber\AddNumber

C:\JitenderN\REBook\AddNumber\AddNumber>^
More? cl AddNumber.cpp /FaAddNumber-CDECL.asm /Gd /FeAddNumber-CDECL.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

AddNumber.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:AddNumber-CDECL.exe
AddNumber.obj

C:\JitenderN\REBook\AddNumber\AddNumber>
```

Set environment for cl.exe (VS compiler)

^ Means to continue command from next line

Figure 5.2: CDECL

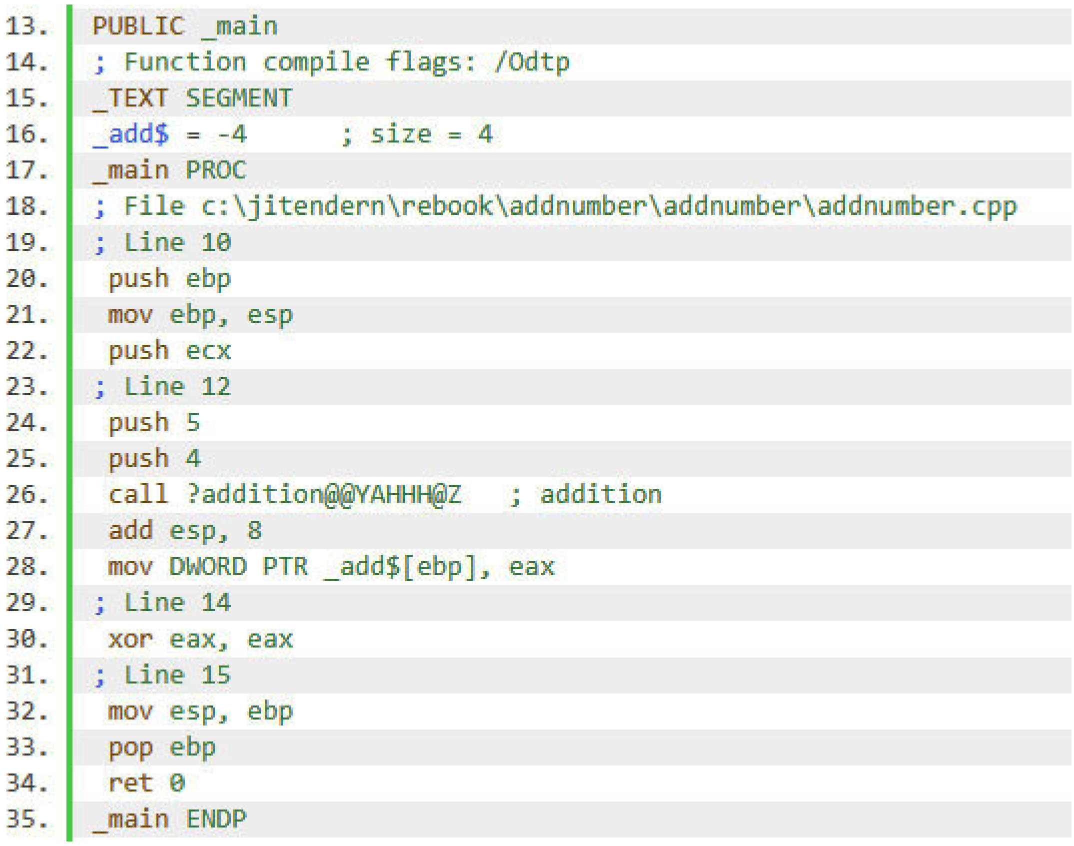
This will generate Now we will move on to analyze the assembly code generated in Our C++ code is divided into two functions, one is the **main** function (caller) and other is the **addition** function To understand CDECL calling convention, we will take up the **main** function code conversation from the C++ code to assembly.



```
int main()
{
 //call function
 int add=addition(4,5);

 return 0;
}

```

Figure 5.3: Main function code in AddNumber.cpp

The assembly code of the main function becomes:



```
13.  PUBLIC _main
14.  ; Function compile flags: /Odtp
15.  _TEXT SEGMENT
16.  _add$ = -4      ; size = 4
17.  _main PROC
18.  ; File c:\jitendern\rebook\addnumber\addnumber\addnumber.cpp
19.  ; Line 10
20.   push ebp
21.   mov ebp, esp
22.   push ecx
23.  ; Line 12
24.   push 5
25.   push 4
26.   call ?addition@@YAHHH@Z   ; addition
27.   add esp, 8
28.   mov DWORD PTR _add$[ebp], eax
29.  ; Line 14
30.   xor eax, eax
31.  ; Line 15
32.   mov esp, ebp
33.   pop ebp
34.   ret 0
35.  _main ENDP
```

Figure 5.4: *Main proc assembly code in AddNumber-CDECL.asm*

Let's analyze the assembly code of the main function in **asm** file:

Line 20-21 is function prolong. These are a sequence of instructions to start a function.

In line 22, the objective of **PUSH ECX** after the function prologue is not to save ECX on stack but to allocate 4 bytes on the stack for storing local variables, which is

Add variable can be accessed with the help of the **_add$** macro, which is equal to -4. So, **Add** can be accessed at **-**

From line 24, we will understand the concept of CDECL. Let’s recall the CDCEL calling convention points one by one to understand the concept practically:

Arguments are passed from the right to left order.

At line 12 of **AddNumber.cpp,** which is **int** we are passing 4,5 parameters to the **addition** function. Now, from right to left means 5 will be pushed on the stack first and then 4 will be pushed on the stack, as done in **AddNumber-CDECL.asm** code at line 24-25. After the parameters are pushed on the stack, the **addition** function is called at line 26 of

Calling caller cleans the stack.

After returning from the **addition** function, at line 27 of **AddNumber-CDECL.asm** instruction **add esp, 8** shrinks the stack by 8. This is because the caller (which is **main** in our case) cleans up the stack as per CDECL's calling convention.

The function return values are passed into the EAX register.

The call to **addition** function at line 26 of **AddNumber-CDECL.asm** returns the **addition** function return value to the EAX register, which is then copied to **add** the variable location, – This copy operation is done by the instruction at line 28 of

mov DWORD PTR _add$[ebp], eax

Similarly, the **main** function returns 0, which is achieved by **xor eax, eax** instruction.

If you are not able to understand any instruction, please refer to *Chapter 4, Walk Through On Assembly*

STDCALL

We will compile the code with no optimization and with STDCALL calling convention. We will use the **/Gz** switch for this. Run the commands given below on the Windows command prompt to set the environment for **cl.exe** (VS compiler) and then compile the code with the following switches:

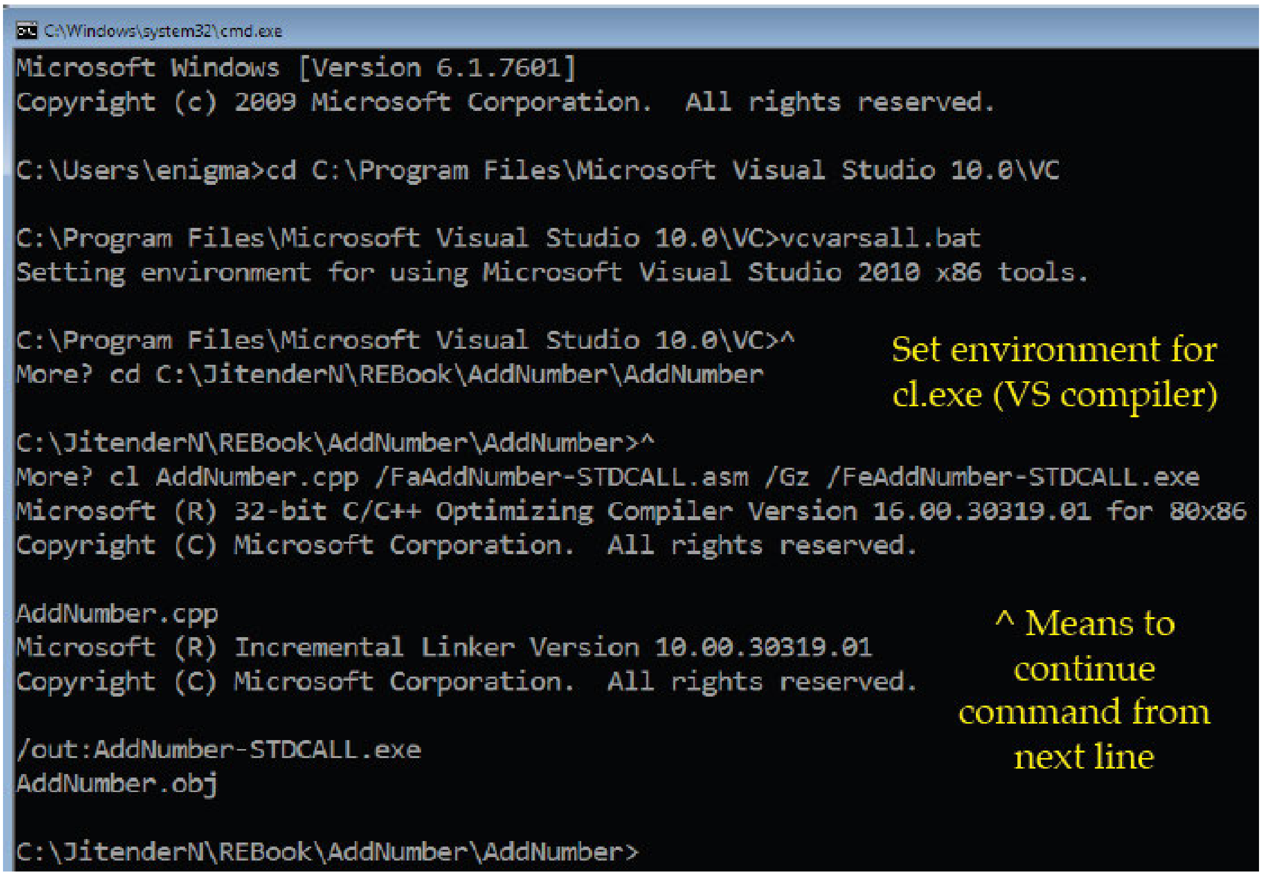
Use this switch for the STDCALL calling convention.

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



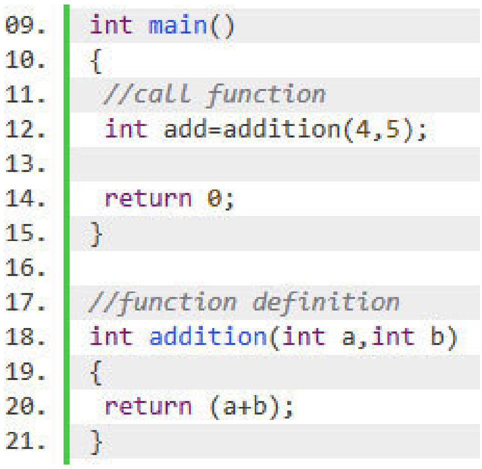
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\AddNumber\AddNumber

C:\JitenderN\REBook\AddNumber\AddNumber>^
More? cl AddNumber.cpp /FaAddNumber-STDCALL.asm /Gz /FeAddNumber-STDCALL.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

AddNumber.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:AddNumber-STDCALL.exe
AddNumber.obj

C:\JitenderN\REBook\AddNumber\AddNumber>
```

Set environment for cl.exe (VS compiler)

^ Means to continue command from next line

Figure 5.5: STDCALL

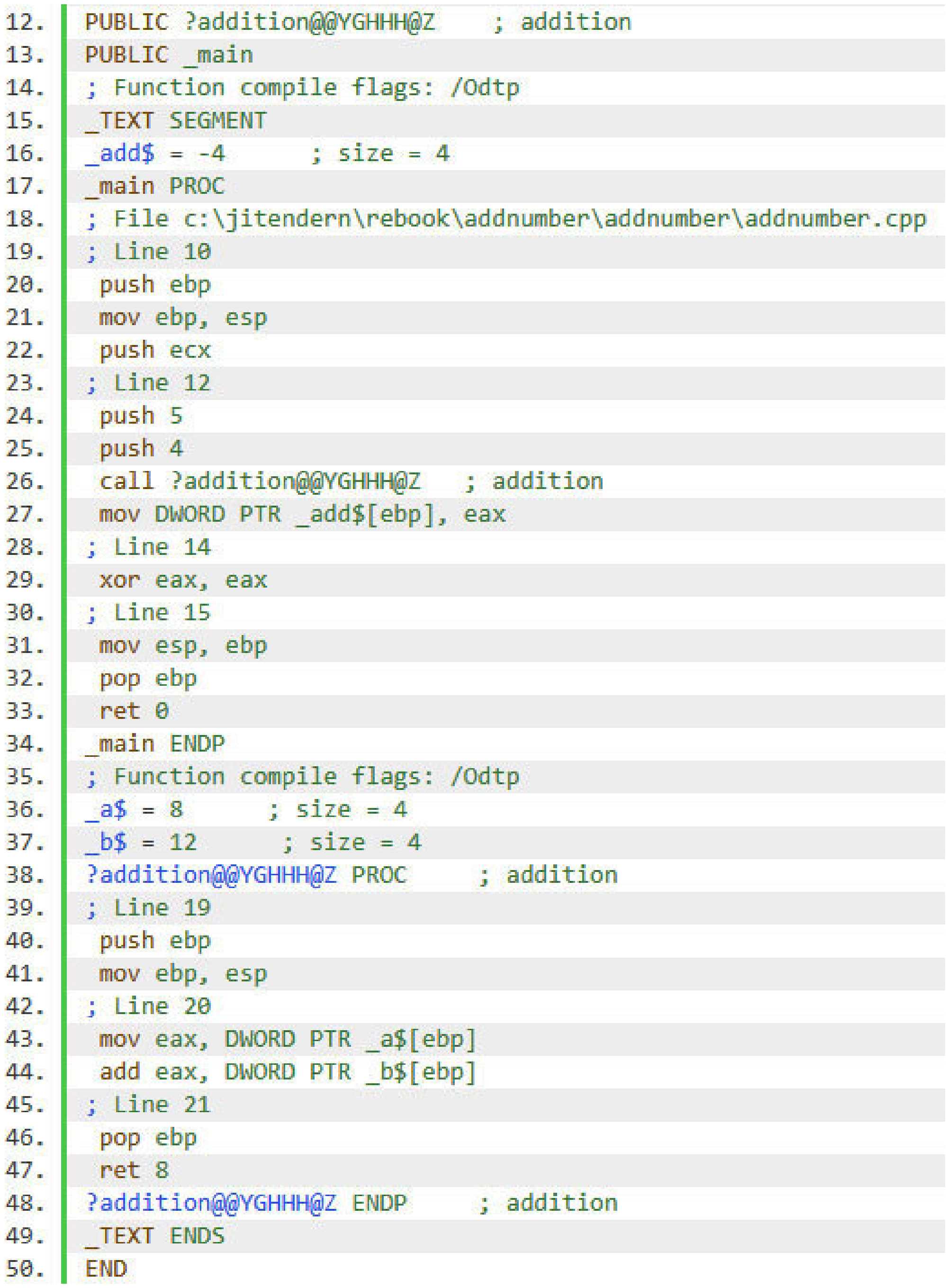
This will generate Now we will move on to analyze the assembly code in To understand the **STDCALL** calling convention, we will again take up the **main** function code along with the **addition** function code to the see conversation from the C++ code to assembly.



```
int main()
{
 //call function
 int add=addition(4,5);

 return 0;
}

//function definition
int addition(int a,int b)
{
 return (a+b);
}
```

Figure 5.6: *Main & addition function code in AddNumber.cpp*

In the **STDCALL** calling convention, the assembly code of the **main** function and the **addition** function becomes:



```
PUBLIC ?addition@@YGHHH@Z    ; addition
PUBLIC _main
; Function compile flags: /Odtp
_TEXT SEGMENT
_add$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\addnumber\addnumber\addnumber.cpp
; Line 10
 push ebp
 mov ebp, esp
 push ecx
; Line 12
 push 5
 push 4
 call ?addition@@YGHHH@Z   ; addition
 mov DWORD PTR _add$[ebp], eax
; Line 14
 xor eax, eax
; Line 15
 mov esp, ebp
 pop ebp
 ret 0
_main ENDP
; Function compile flags: /Odtp
_a$ = 8       ; size = 4
_b$ = 12       ; size = 4
?addition@@YGHHH@Z PROC      ; addition
; Line 19
 push ebp
 mov ebp, esp
; Line 20
 mov eax, DWORD PTR _a$[ebp]
 add eax, DWORD PTR _b$[ebp]
; Line 21
 pop ebp
 ret 8
?addition@@YGHHH@Z ENDP      ; addition
_TEXT ENDS
END
```

Figure 5.7: *Main & addition proc assembly code in AddNumber-STDCALL.asm*

Let's analyze the assembly code with respect to STDCALL. Most of the points will be the same as that of CDCEL calling convention. We will discuss the few differences here:

In ASM code line 20-21 is function prolong. These are a sequence of instructions to start a function.

In ASM code line 22, the objective of **PUSH ECX** after the function prologue, is not to save ECX on stack but to allocate 4 bytes on the stack for storing local variable, which is

Add variable can be accessed with the help of the **_add$** which is equal to -4. So **Add** variable can be accessed at –

From line 24 we will understand the concept of STDCALL. Let's recall STDCALL points one by one:

Arguments are passed in from the right to left order, which is same as CDCEL

At line 12 of **AddNumber.cpp,** which is **int** we are passing 4,5 parameters to the **addition** function. Now from right to feft means 5 will be pushed on the stack first and then 4 will be pushed on the stack. This can be seen at line 26 of **AddNumber-STDCALL.asm** by **PUSH 5 & PUSH 4** instructions before the **addition** function is called.

Called function, callee cleans the stack. This point differs from that in CDCEL.

This can be seen at line 47 of by the **RET 8** instruction. The callee cleans up the stack by using the **RET nBytes** instruction, where the RET instruction transfers control from the callee to the caller to the return address saved on the stack. which in our case, is 8. So, 8 bytes are released on the stack to clean up the stack.

The function return values are passed into the EAX register.

The call to **addition** function at line 26 of **AddNumber-STDCALL.asm** returns the **addition** function return value to the EAX register, which is then copied to **add** the variable location, – This copy operation is done by the following instruction:

mov DWORD PTR _add$[ebp], eax

at line 27 of **AddNumber-STDCALL.asm**

Similarly, the **main** function returns 0, which is achieved by **xor eax, eax** instruction.

FASTCALL

As we read previously, the FASTCALL calling convention differs majorly in passing arguments. To understand this, we will compile the code with the optimization off and with the **FASTCALL** switch. We will use the **/Gr** switch for this. Run the commands given below on the Windows command prompt to set the environment for **cl.exe** (VS compiler) and then compile the code with the following switches:

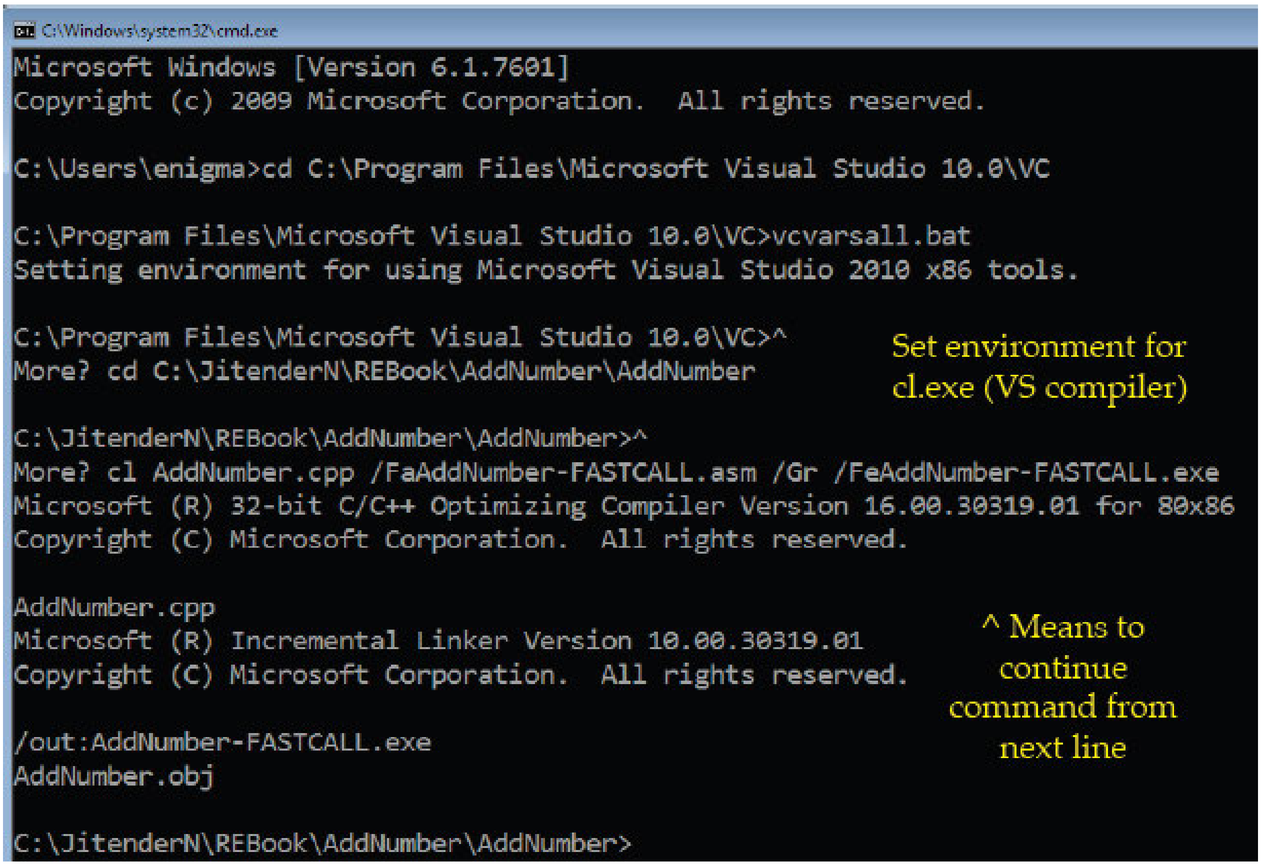
Use this switch for the FASTCALL calling convention.

Name of output assembly listing file

Name of output executable file

file file file file file file file file file file file file file file file file

The following is the output of running the preceding commands:



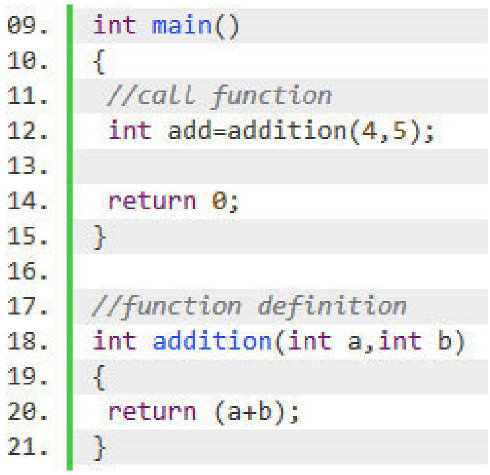
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\AddNumber\AddNumber

C:\JitenderN\REBook\AddNumber\AddNumber>^
More? cl AddNumber.cpp /FaAddNumber-FASTCALL.asm /Gr /FeAddNumber-FASTCALL.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

AddNumber.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:AddNumber-FASTCALL.exe
AddNumber.obj

C:\JitenderN\REBook\AddNumber\AddNumber>
```

Set environment for cl.exe (VS compiler)

^ Means to continue command from next line

Figure 5.8: FASTCALL

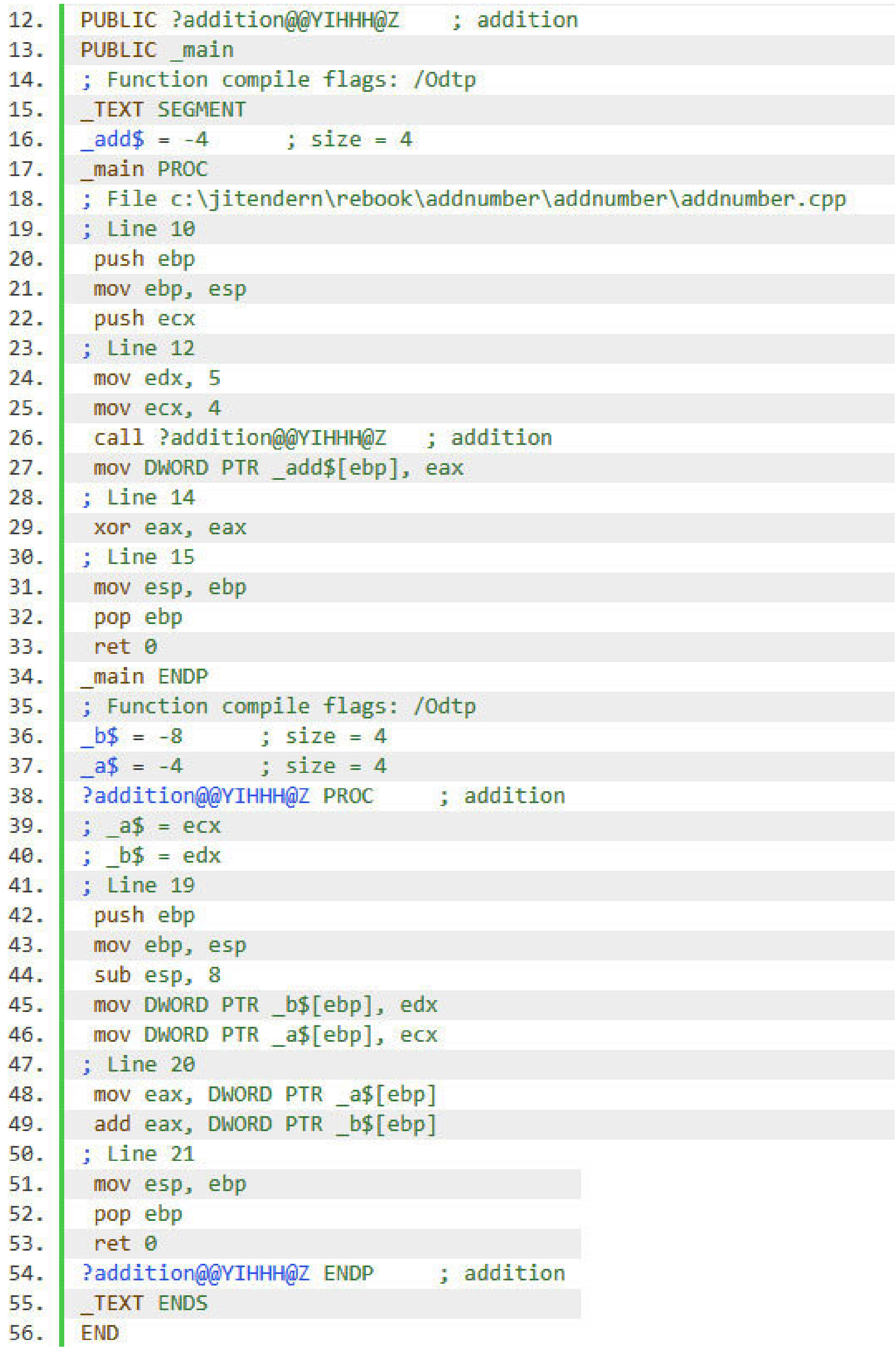
This will generate To analyze the assembly code in we will once again see our C/C++ code.



```
int main()
{
 //call function
 int add=addition(4,5);

 return 0;
}

//function definition
int addition(int a,int b)
{
 return (a+b);
}
```

Figure 5.9: *Main & addition function code in AddNumber.cpp*

Let’s see **AddNumber-FASTCALL.asm** generated as follows:



```
PUBLIC ?addition@@YIHHH@Z     ; addition
PUBLIC _main
; Function compile flags: /Odtp
_TEXT SEGMENT
_add$ = -4       ; size = 4
_main PROC
; File c:\jitendern\rebook\addnumber\addnumber\addnumber.cpp
; Line 10
 push ebp
 mov ebp, esp
 push ecx
; Line 12
 mov edx, 5
 mov ecx, 4
 call ?addition@@YIHHH@Z    ; addition
 mov DWORD PTR _add$[ebp], eax
; Line 14
 xor eax, eax
; Line 15
 mov esp, ebp
 pop ebp
 ret 0
_main ENDP
; Function compile flags: /Odtp
_b$ = -8       ; size = 4
_a$ = -4       ; size = 4
?addition@@YIHHH@Z PROC      ; addition
; _a$ = ecx
; _b$ = edx
; Line 19
 push ebp
 mov ebp, esp
 sub esp, 8
 mov DWORD PTR _b$[ebp], edx
 mov DWORD PTR _a$[ebp], ecx
; Line 20
 mov eax, DWORD PTR _a$[ebp]
 add eax, DWORD PTR _b$[ebp]
; Line 21
 mov esp, ebp
 pop ebp
 ret 0
?addition@@YIHHH@Z ENDP      ; addition
_TEXT ENDS
END
```

Figure 5.10: *Main & addition proc assembly code in AddNumber-FASTCALL.asm*

Let's analyze the assembly code in the same order we did for the other calling conventions.

The function prolong instruction on line 20-21 is a sequence of instructions to start a function.

In ASM code line 22, PUSH ECX is the same. It is not to save ECX on stack but to allocate 4 bytes on the stack for storing the local variable, which is

The **add** variable can be accessed with the help of the **_add$** which is equal to -4. So, **add** can be accessed at –

From line 24, we will see a different approach to pass arguments to function. To understand, let's recall the FASTCALL concept:

Initial arguments are not pushed on stack but rather passed in the registers. The first two or three arguments are passed in the registers EDX, ECX, or EAX. Additional arguments are passed on to the stack. Arguments are passed from the right to left order.

By now, we are clear with the right to left approach. The point to note here is that argument 5 is moved to EDX and argument 4 is moved to ECX, as done at line 24-25 of Once the arguments are moved to registers, the call to **addition** function is made at line 26 of

During the **addition** function execution, we can see that the arguments in EDX, ECX are passed to the stack for further processing between lines 45-49 of

Calling function, caller cleans the stack.

Arguments are passed to EDX, ECX so no arguments are passed to stack, no stack cleanup is needed.

Function return values are passed into EAX register this is same as other 2 calling convention we discussed. Restating same point about FASTCALL.

Call to **addition** function at line 26 of **AddNumber-FASTCALL.asm** returns the **addition** function return value to EAX register, which then is copied to **add** variable location, – This copy operation is done by this instruction:

mov DWORD PTR _add$[ebp], eax

at line 27 of **AddNumber-FASTCALL.asm**

Similarly, the **main** function returns 0, which is achieved by xor eax, eax instruction at line 29 of

Conclusion

In this chapter, we covered three types of code calling conventions majorly used: CDCEL, STDCALL, and FASTCALL. We learned that in CDCEL and STDCALL, arguments are passed from the right to left order and the function return value is passed into the EAX register. In the CDCEL calling function, the caller cleans the stack and in STDCALL, callee cleans the stack.

In FASTCALL, the initial arguments are not pushed on the stack but rather passed in the registers. The rest function return value is passed into the EAX register and in calling function, the caller cleans the stack if needed. In the next chapter, we will take C/C++ codes and compile them to understand assembly output.

CHAPTER 6

Reverse Engineering Pattern of Basic Code

In this chapter, we will write small pieces of code and compile them to understand assembly output. We will walk through step-by-step instructions in the assembly code and understand code flow from the assembly point of view.

Throughout this chapter of compiling small pieces of code, we will use Microsoft compiler on the 32-bit environment. All the programs are compiled on Microsoft Windows 32-bit environment. We will also use code optimization during our analysis.

Structure

In this chapter, we will cover the following topics:

What is code optimization

Understanding assembly pattern of the C/C++ program

Concept of code optimization

Tools used to generate the assembly pattern of C/C++ program

Objective

After studying this chapter, you should be able to:

Understand code optimization and its importance

Assembly code with and without optimization

What is Code Optimization?

Optimization means doing something at its best in order to effectively utilize resources. Code optimization means to transform the code to remove unnecessary lines, so as to consume fewer resources (Memory, CPU and others) during execution. When code is optimized by compilers, the following things are taken care of:

The meaning of the code should not be changed while optimizing the code.

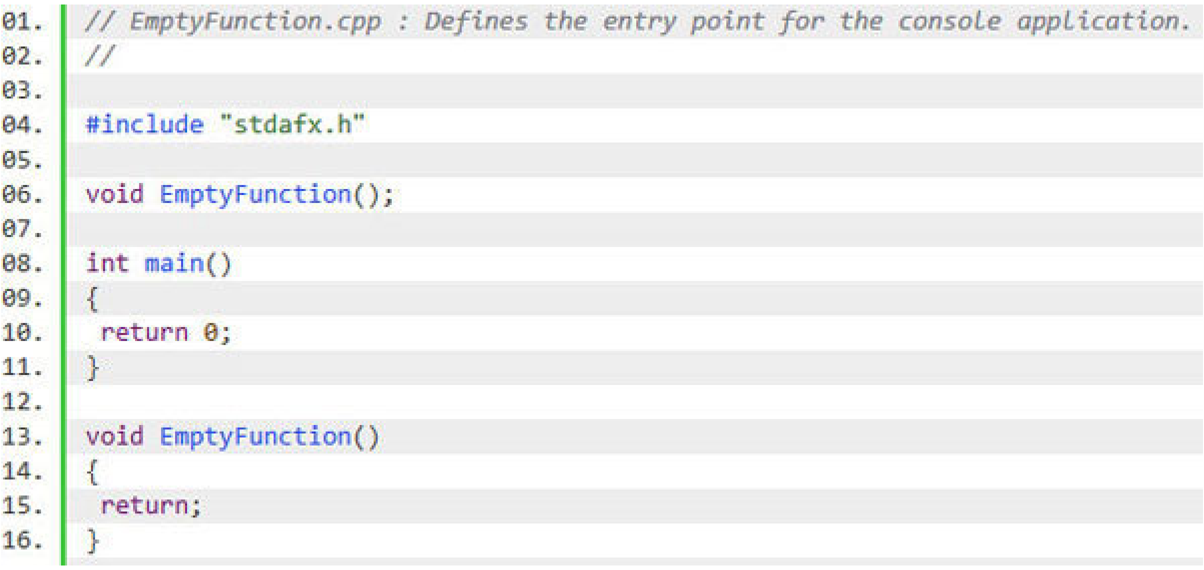
An optimized code should consume fewer resources.

Optimization should not impact the compiling time of the program.

Let's begin with a small C/C++ program and gradually move to complex programs. This process will help you understand the pattern of code in assembly language with respect to C/C++ applications.

Empty function

An empty function is something that does nothing. Let’s create an empty function in C/C++ code. Here, we are defining and declaring an empty function as



```
// EmptyFunction.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

void EmptyFunction();

int main()
{
 return 0;
}

void EmptyFunction()
{
 return;
}
```

Figure 6.1: *EmptyFunction.cpp*

Empty Function without Optimization

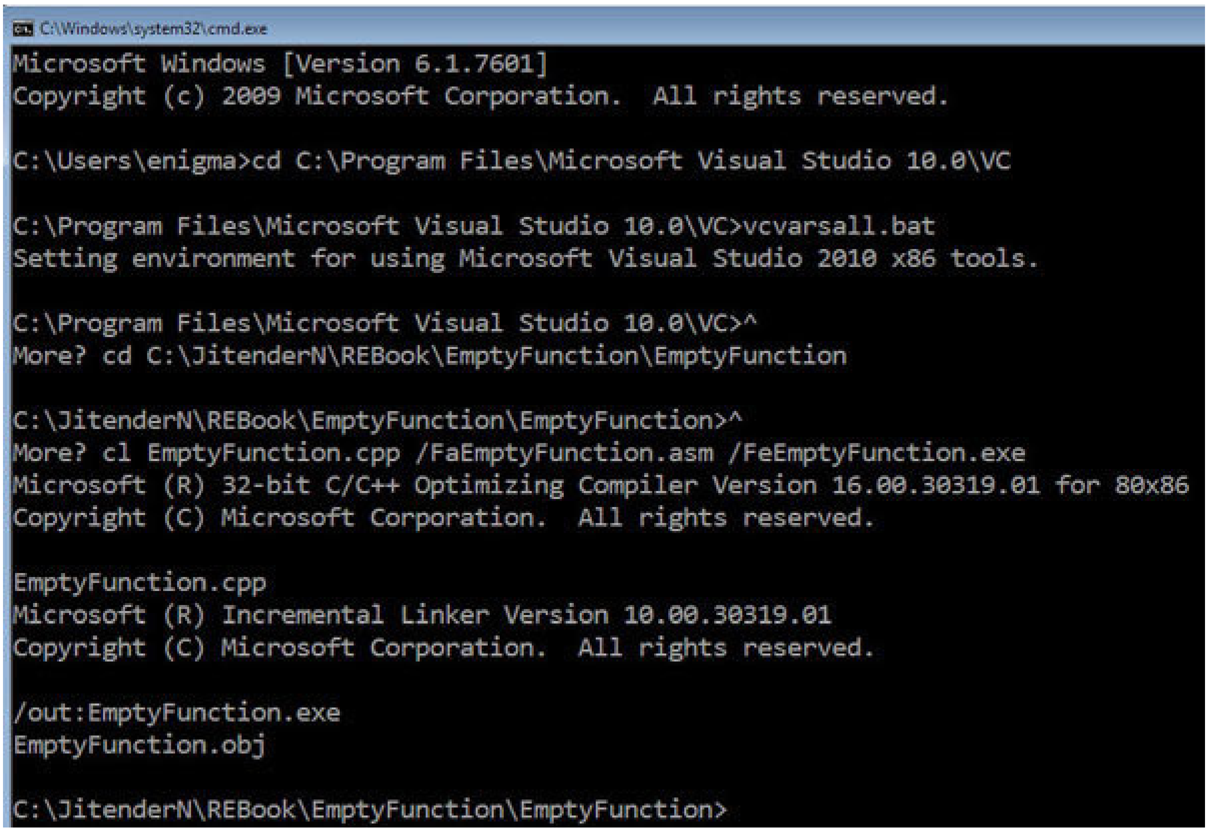
Now, let’s compile it without optimization using the MSVC compiler Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\EmptyFunction\EmptyFunction

C:\JitenderN\REBook\EmptyFunction\EmptyFunction>^
More? cl EmptyFunction.cpp /FaEmptyFunction.asm /FeEmptyFunction.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

EmptyFunction.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:EmptyFunction.exe
EmptyFunction.obj

C:\JitenderN\REBook\EmptyFunction\EmptyFunction>
```

Figure 6.2: *Empty Function without Optimization*

The assembly code generated without optimization is as follows:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\EmptyFunction\EmptyFunction\EmptyFunction.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. PUBLIC _main
13. ; Function compile flags: /Odtp
14. _TEXT SEGMENT
15. _main PROC
16. ; File c:\jitendern\rebook\emptyfunction\emptyfunction\emptyfunction.cpp
17. ; Line 9
18.  push ebp
19.  mov ebp, esp
20. ; Line 10
21.  xor eax, eax
22. ; Line 11
23.  pop ebp
24.  ret 0
25. _main ENDP
26. _TEXT ENDS
27. PUBLIC ?EmptyFunction@@YAXXZ    ; EmptyFunction
28. ; Function compile flags: /Odtp
29. _TEXT SEGMENT
30. ?EmptyFunction@@YAXXZ PROC    ; EmptyFunction
31. ; Line 14
32.  push ebp
33.  mov ebp, esp
34. ; Line 16
35.  pop ebp
36.  ret 0
37. ?EmptyFunction@@YAXXZ ENDP    ; EmptyFunction
38. _TEXT ENDS
39. END
```

Figure 6.3: *EmptyFunction.asm*

We will walk through the assembly code line by line to understand the meaning and working of the code pattern. Line 1 says:

▼Line 1

; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

This is a comment as it starts with a semicolon. This comment states the compiler that we are using to generate the assembly code, Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86. The compiler basically translates the program from one language to another. But when we talk about optimizing compiler, it improves the code to run faster.

▼Line 3

TITLE **C:\JitenderN\REBook\EmptyFunction\EmptyFunction\EmptyFunction.cpp**

Title defines the name of the absolute path of the C/C++ program.

▼Line 4

.686P

This enables all the instructions for the Pentium Pro processor (32-bit MASM only).

▼Line 5

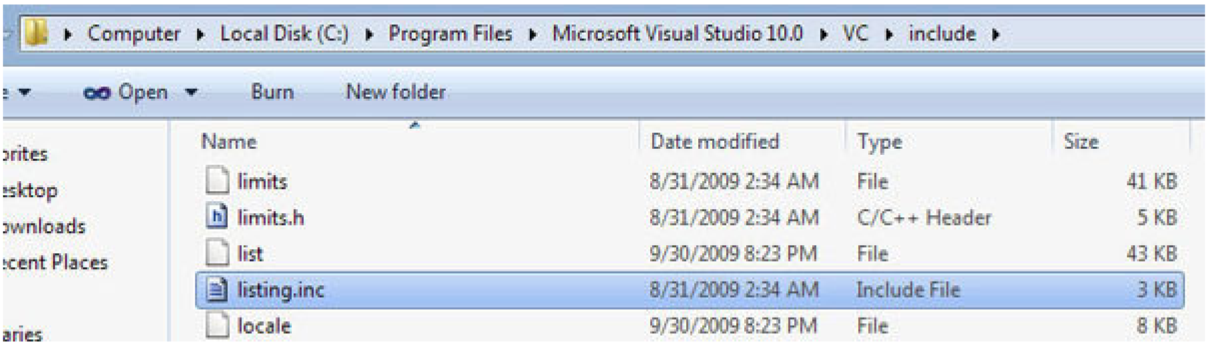
```
.XMM
```

This means that the program requires a CPU with the Streaming SIMD Extensions instruction set.

▼Line 6

```
include listing.inc
```

This **listing.inc** file contains the assembler macros. Macros are used in assembly language for modular programming. Visual C++ does not embed the macro code in the assembly code to improve performance and align the code. You can find this file in the Visual C++ **include** folder.



Name	Date modified	Type	Size
limits	8/31/2009 2:34 AM	File	41 KB
limits.h	8/31/2009 2:34 AM	C/C++ Header	5 KB
list	9/30/2009 8:23 PM	File	43 KB
listing.inc	8/31/2009 2:34 AM	Include File	3 KB
locale	9/30/2009 8:23 PM	File	8 KB

Figure 6.4: *listing.inc path*

▼Line 7

```
.model flat
```

This is directive for enabling the flat memory model. To understand memory models, we have to understand that memory is accessed using 3 memory models:

This is a non-segmented memory model where the whole memory appears to a program as one massive array of bytes. Code, data, and stack all reside in the same address space.

In this model, memory is divided into groups of spaces called segments.

The real address memory model is a very old memory model used in the Intel 8086 processor.

▼Line 9-10

```
INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES
```

With **INCLUDELIB** derivative, we link **LIBCMT.LIB** and **OLDNAMES.LIB** libraries located in the following path:

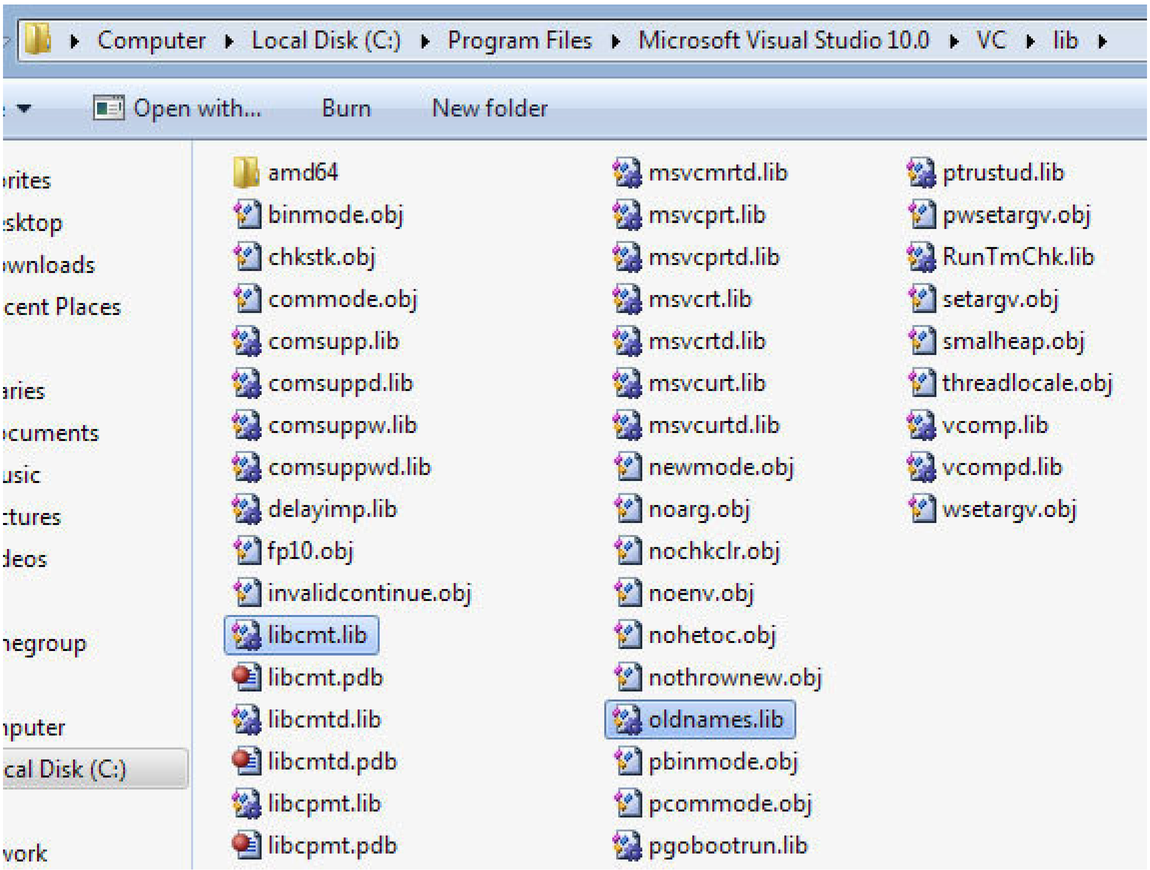


Figure 6.5: *Libraries path*

▼Line 12

PUBLIC _main

PUBLIC is the derivative which makes the procedure public. A derivative is an instruction which is used by the assembler to automate the assembly process; it also helps improve the code readability.

all functions begin with an underscore. **main** function is labeled as a public function, which means it can be accessed by other modules.

▼Line 13

```
; Function compile flags: /Odtp
```

All the comments begin with semicolons. This line states that the code is compiled with the **/Odtp** switch.

▼Line 14

```
_TEXT SEGMENT
```

This is the start of the text segment or, we can say, the code segment.

▼Line 15

```
_main PROC
```

Procedures are defined with the PROC statement and they must be closed by the ENDP statement. We can see **_main ENDP** on line 25.

▼ Line 16

```
; File c:\jitendern\rebook\emptyfunction\emptyfunction\emptyfunction.cpp
```

This is a comment stating the file path of the C/C++ source code.

▼Line 17

```
; Line 9
```

This is a comment which states that line number 9 of the C/C++ source code file is mapped with the instruction following this comment.

▼Line 18-19

```
push ebp
mov ebp, esp
```

This is function prologue, which is the sequence of instructions at the start of a function.

▼Line 20-21

```
; Line 10
xor eax, eax
```

XOR is an exclusive OR. The EAX register is used for storing the return value of the function. The **main** function is returning 0 in the C/C++ code, so XOR EAX with EAX will set the EAX register to zero. Also, compilers can use MOV EAX, 0 in place of XOR EAX, EAX. But XOR is preferred over MOV as XOR occupies 2-byte opcode and an MOV instruction occupies 5 bytes.

▼Line 22

```
; Line 11
```

This comment states that line number 12 of the C/C++ source code file is mapped with the instruction following this comment.

▼Line 23

```
pop ebp
```

POP EBP is a function epilogue, which is the sequence of instructions to end a function.

▼Line 24

```
ret 0
```

RET is the return instruction. It returns the instruction pointer to the caller procedure. The syntax of the RET instruction is:

```
RET nBytes
```

The return instruction has an optional **nBytes** operand that specifies the number of bytes to be added to the value of the

ESP register after the return.

▼Line 25-26

```
_main ENDP
_TEXT ENDS
```

This is the close of the **main** procedure and the end of the text segment or code segment.

▼Line 27

```
PUBLIC ?EmptyFunction@@YAXXZ      ; EmptyFunction
```

Internally, functions are represented by their decorated names, which are encoded string created during the compilation process. It appends the calling convention, function return type, function parameters, and other information with the function name. This process helps the linker find the correct function when linking an executable. This process of name decoration is also known as name mangling.

Like main, **EmptyFunction** is made public with the help of the **PUBLIC** derivative.

▼Line 28

```
; Function compile flags: /Odtp
```

It is the same as explained previously. It states that the code is compiled with the **/Odtp** switch.

▼Line 29

```
_TEXT SEGMENT
```

This is the start of the text segment or code segment.

▼Line 30

```
?EmptyFunction@@YAXXZ PROC      ; EmptyFunction
```

Starting of **EmptyFunction** procedure with **PROC** statement

▼Line 31

```
; Line 15
```

This comment states that line number 15 of the C/C++ source code file is mapped with the instruction following this comment.

▼Line 32-33

```
push ebp
mov ebp, esp
```

This is a function prologue for

▼Line 34

```
; Line 16
```

This comment states that line number 17 of the C/C++ source code file is mapped with the instruction following this comment.

▼Line 35

```
pop ebp
```

POP EBP is a function epilogue of

▼Line 36

```
ret 0
```

As **EmptyFunction** is doing nothing, it's just an empty/blank function. So, it's retuning the instruction pointer back to the caller. 0 means that the ESP will be unchanged.

▼Line 37-38

```
?EmptyFunction@@YAXXZ ENDP      ; EmptyFunction
_TEXT ENDS
```

This is the close of the **EmptyFunction** procedure and the end of the text segment or code segment.

▼Line 39

```
END
```

The **END** statement ends the source code.

Empty Function with Optimization

Now let's compile the code with optimization using the **/ox** switch on the x86 platform. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

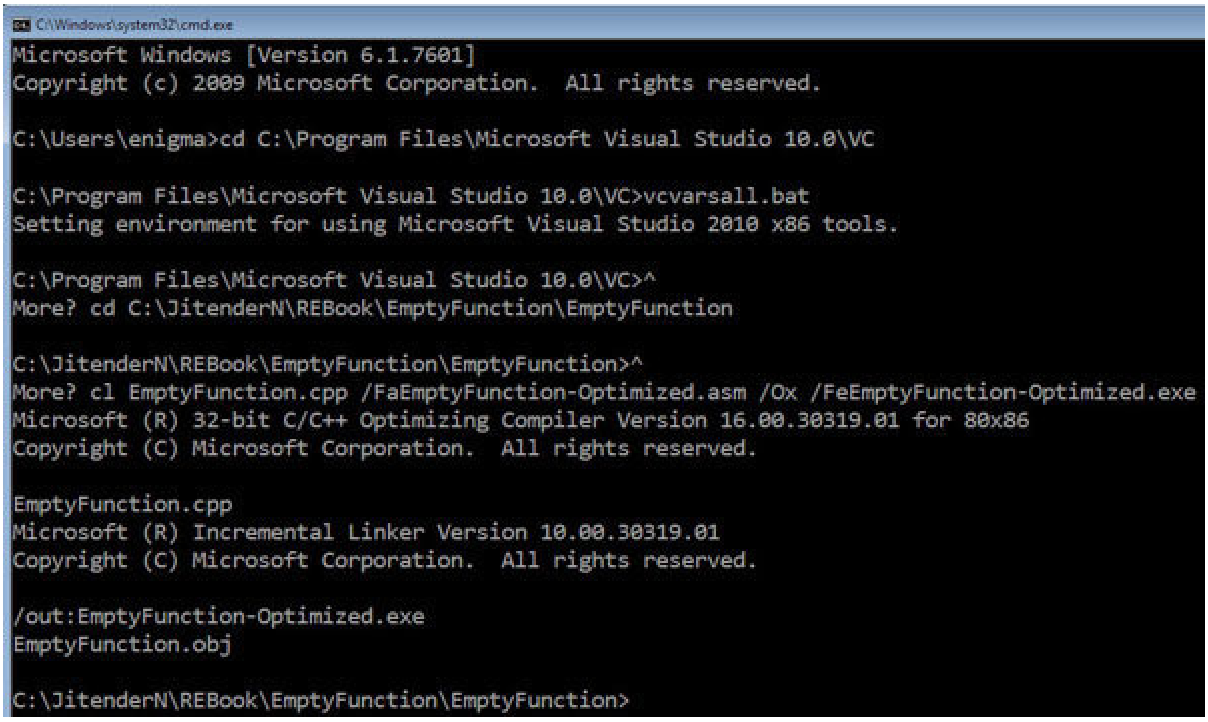
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

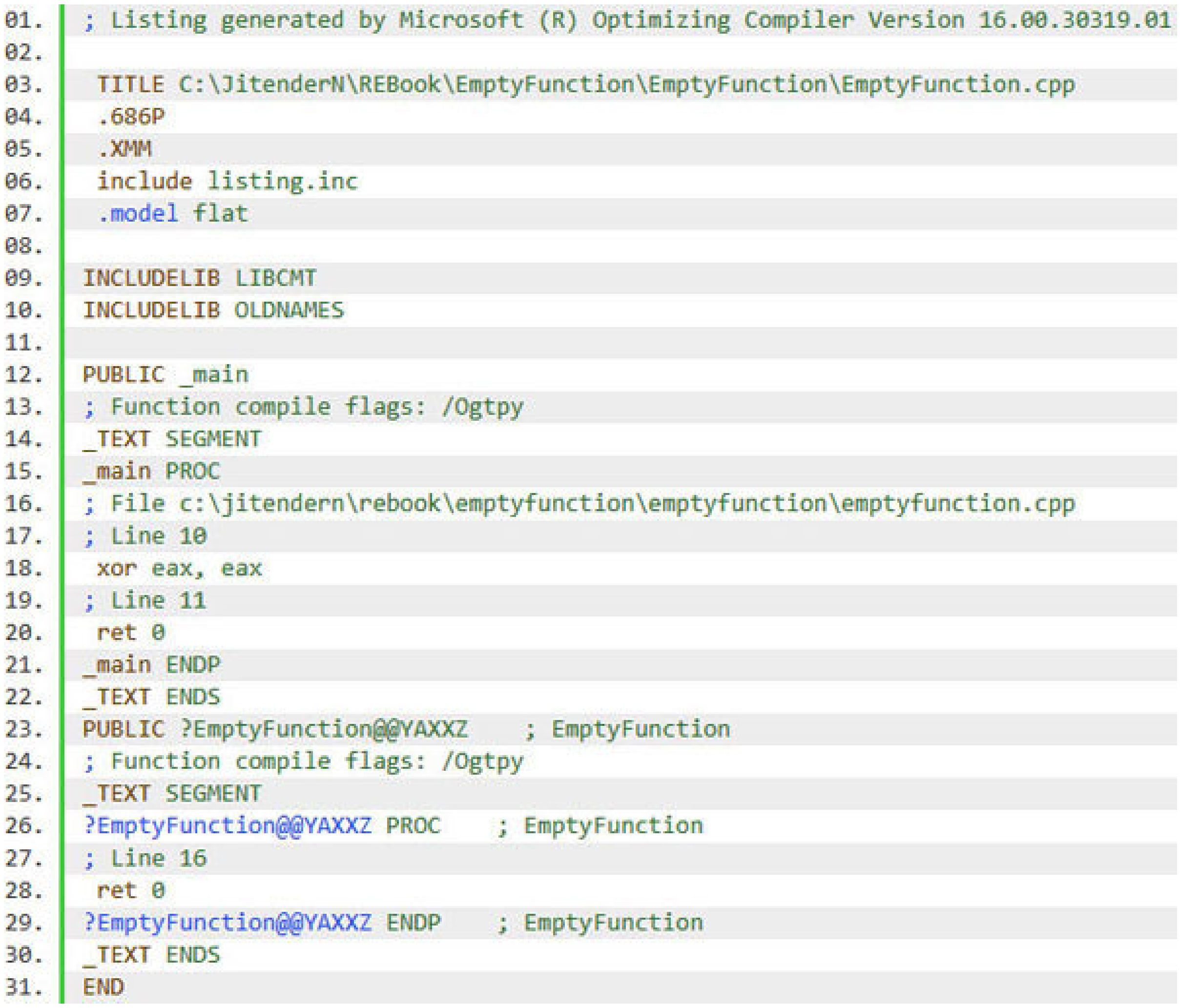
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\EmptyFunction\EmptyFunction

C:\JitenderN\REBook\EmptyFunction\EmptyFunction>^
More? cl EmptyFunction.cpp /FaEmptyFunction-Optimized.asm /Ox /FeEmptyFunction-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

EmptyFunction.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:EmptyFunction-Optimized.exe
EmptyFunction.obj

C:\JitenderN\REBook\EmptyFunction\EmptyFunction>
```

Figure 6.6: *Empty Function with Optimization*

The assembly code we get is as follows:



```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\EmptyFunction\EmptyFunction\EmptyFunction.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

PUBLIC _main
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_main PROC
; File c:\jitendern\rebook\emptyfunction\emptyfunction\emptyfunction.cpp
; Line 10
 xor eax, eax
; Line 11
 ret 0
_main ENDP
_TEXT ENDS
PUBLIC ?EmptyFunction@@YAXXZ    ; EmptyFunction
; Function compile flags: /Ogtpy
_TEXT SEGMENT
?EmptyFunction@@YAXXZ PROC    ; EmptyFunction
; Line 16
 ret 0
?EmptyFunction@@YAXXZ ENDP    ; EmptyFunction
_TEXT ENDS
END
```

Figure 6.7: *EmptyFunction-Optimized.asm*

As we can see, when the code is optimized, it transforms the code to remove the unnecessary lines. During this optimization, the meaning of the code remains the same. Compilers nowadays are good at optimization. So as a reverse engineer, it is always a good practice to understand the concept or the logic behind the code rather than the original code. If we can understand the logic then we can write our prototype.

Coming back to the code, we can see most of the lines of the code are the same as explained in the earlier section. So, we will not restate all of them. We will take up instructions that are specific to the code optimization.

▼Line 18

```
xor eax, eax
```

main procedure is XOR'ing EAX to return zero, as the EAX register is used for storing the return value of the function.

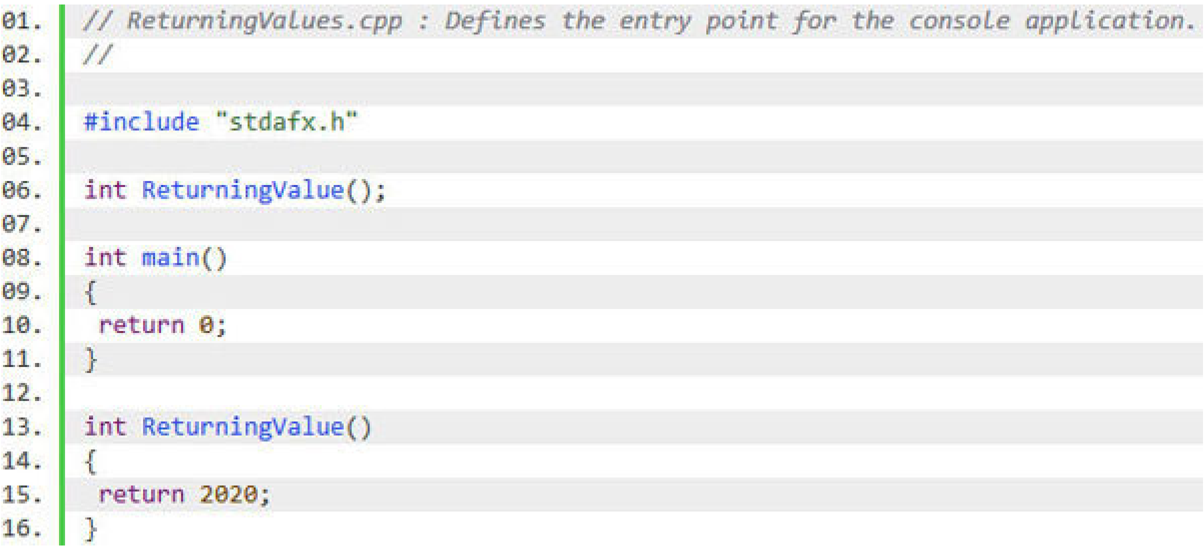
▼Line 28

```
ret 0
```

EmptyFunction is returning the instruction pointer to the caller with just the **RET** instruction.

Returning Value

In this section, we will create a function in the C/C++ code to return a constant value. We will define and declare a



```cpp
// ReturningValues.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int ReturningValue();

int main()
{
 return 0;
}

int ReturningValue()
{
 return 2020;
}
```

Figure 6.8: *ReturningValues.cpp*

Returning Value without Optimization

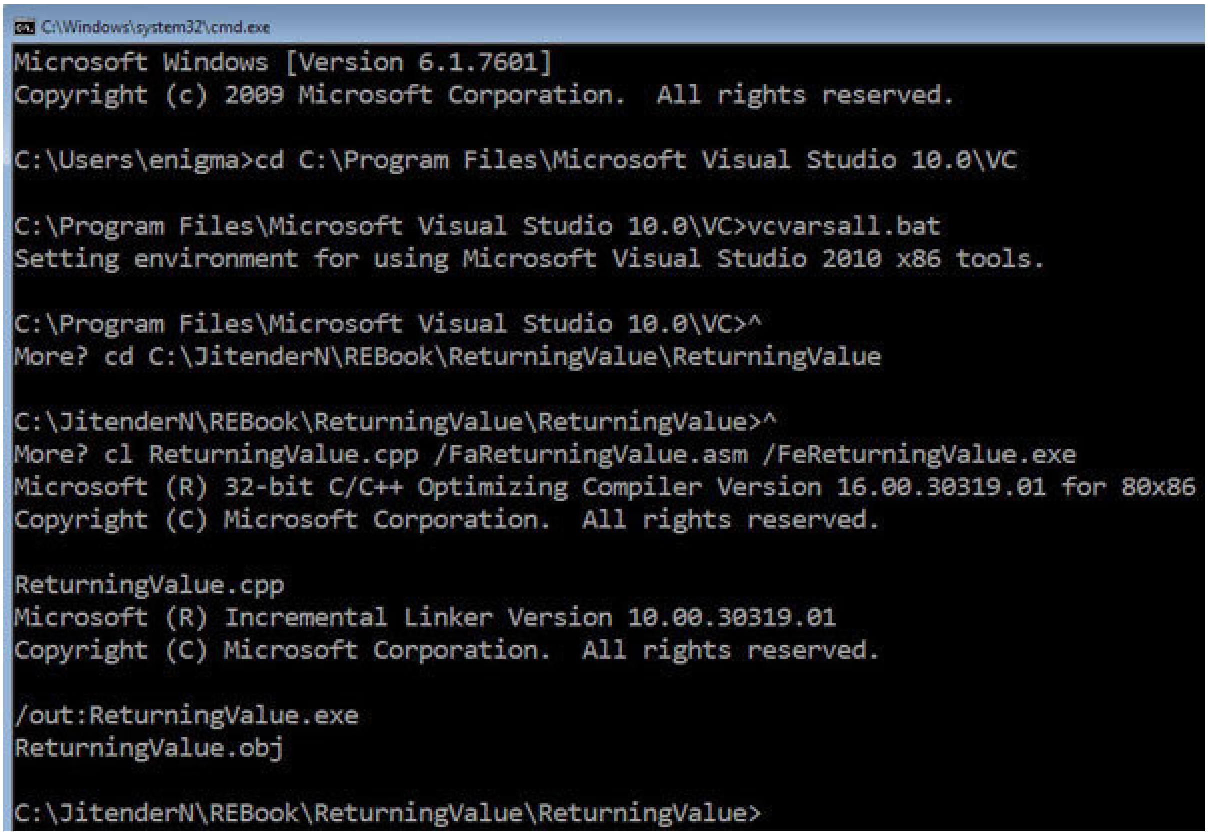
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\ReturningValue\ReturningValue

C:\JitenderN\REBook\ReturningValue\ReturningValue>^
More? cl ReturningValue.cpp /FaReturningValue.asm /FeReturningValue.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ReturningValue.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ReturningValue.exe
ReturningValue.obj

C:\JitenderN\REBook\ReturningValue\ReturningValue>
```

Figure 6.9: *Returning a Value without Optimization*

Here is what we get after compiling:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\ReturningValue\ReturningValue\ReturningValue.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. PUBLIC _main
13. ; Function compile flags: /Odtp
14. _TEXT SEGMENT
15. _main PROC
16. ; File c:\jitendern\rebook\returningvalue\returningvalue\returningvalue.cpp
17. ; Line 9
18.  push ebp
19.  mov ebp, esp
20. ; Line 10
21.  xor eax, eax
22. ; Line 11
23.  pop ebp
24.  ret 0
25. _main ENDP
26. _TEXT ENDS
27. PUBLIC ?ReturningValue@@YAHXZ    ; ReturningValue
28. ; Function compile flags: /Odtp
29. _TEXT SEGMENT
30. ?ReturningValue@@YAHXZ PROC    ; ReturningValue
31. ; Line 14
32.  push ebp
33.  mov ebp, esp
34. ; Line 15
35.  mov eax, 2020    ; 000007e4H
36. ; Line 16
37.  pop ebp
38.  ret 0
39. ?ReturningValue@@YAHXZ ENDP    ; ReturningValue
40. _TEXT ENDS
41. END
```

Figure 6.10: *ReturningValue.asm*

We have discussed most of the lines in the code in earlier section. So, we will focus on the main instructions.

▼Line 20-24

; Line 10

```
xor eax, eax
; Line 11
pop ebp

ret 0
```

This is the part of instructions from the **main** function, where we are XOR'ing EAX to make EAX equal to zero. Once the EAX is zero, we are calling function epilogue using the POP instruction. The **RET** instruction returns the execution back to the caller, where the caller can take the return value from the EAX register.

▼Line 34-38

```
; Line 15
mov eax, 2020       ; 000007e4H
; Line 16
pop ebp
ret 0
```

Now, we move to the **ReturningValue** function, where EAX is filled with 2020, which is the return value of the **ReturningValue** function. The POP instruction is function epilogue and finally, RET passes the instruction pointer back to the caller, where the caller will take the result from the EAX register.

Returning Value with Optimization

Compile the code with optimization (with **/ox** switch) in the MSVC compiler on the x86 platform. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

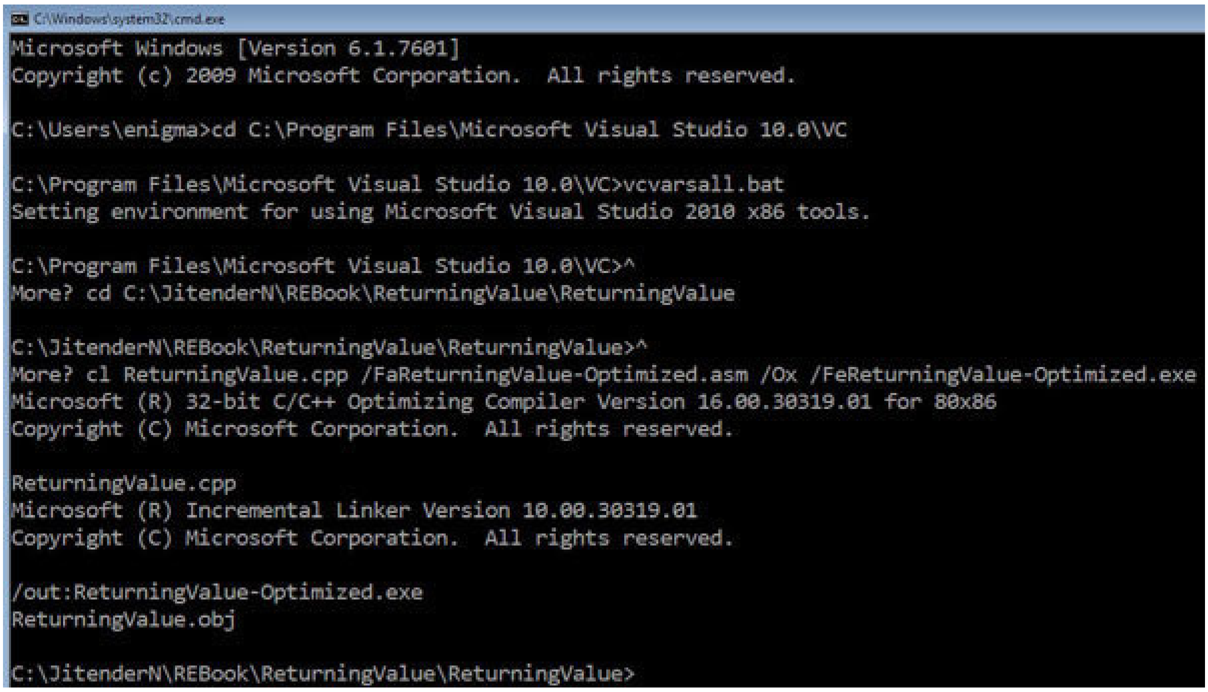
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

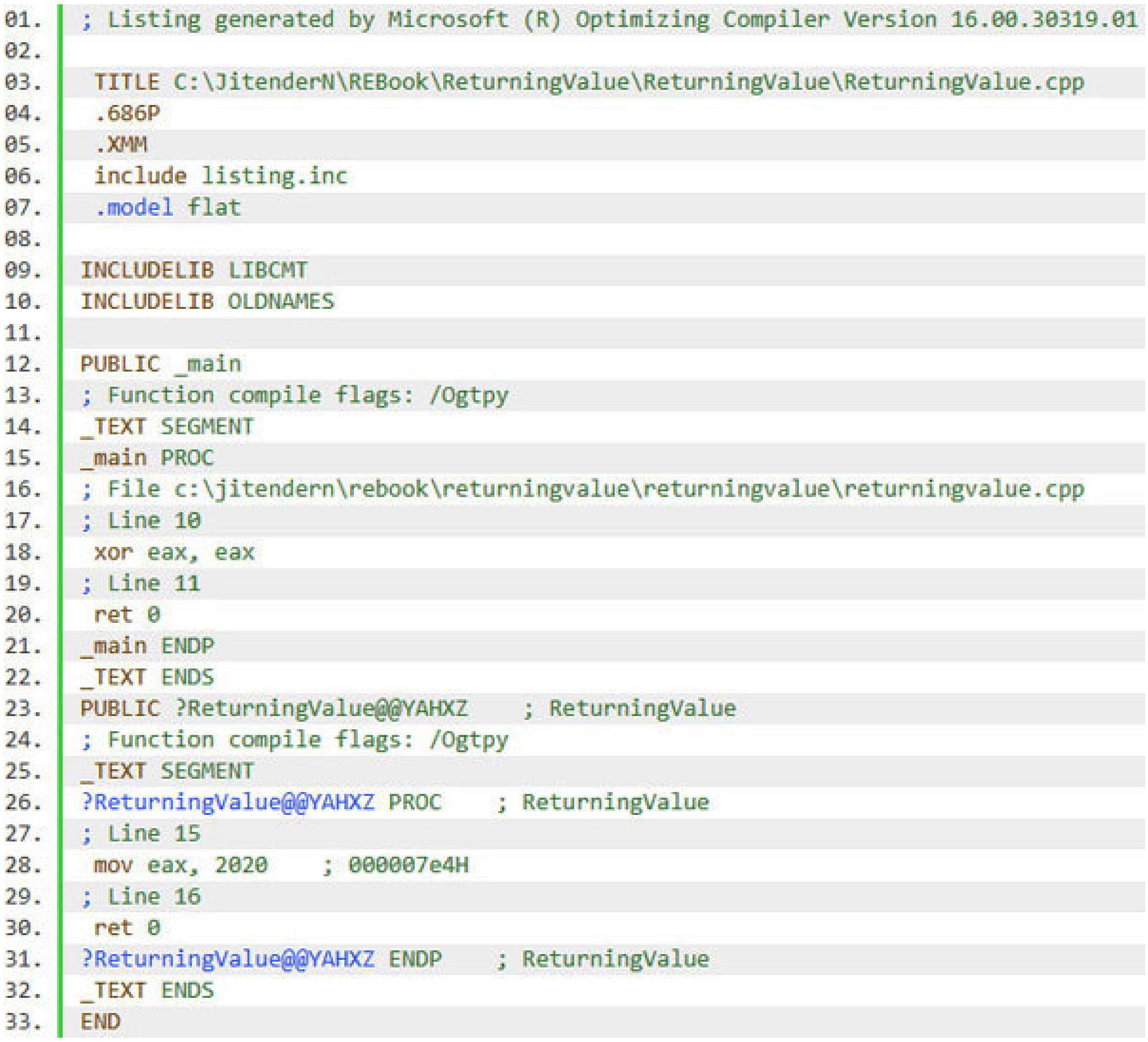
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\ReturningValue\ReturningValue

C:\JitenderN\REBook\ReturningValue\ReturningValue>^
More? cl ReturningValue.cpp /FaReturningValue-Optimized.asm /Ox /FeReturningValue-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ReturningValue.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ReturningValue-Optimized.exe
ReturningValue.obj

C:\JitenderN\REBook\ReturningValue\ReturningValue>
```

Figure 6.11: *Returning Value with Optimization*

Assembly code generated will be:



```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\ReturningValue\ReturningValue\ReturningValue.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. PUBLIC _main
13. ; Function compile flags: /Ogtpy
14. _TEXT SEGMENT
15. _main PROC
16. ; File c:\jitendern\rebook\returningvalue\returningvalue\returningvalue.cpp
17. ; Line 10
18.  xor eax, eax
19. ; Line 11
20.  ret 0
21. _main ENDP
22. _TEXT ENDS
23. PUBLIC ?ReturningValue@@YAHXZ     ; ReturningValue
24. ; Function compile flags: /Ogtpy
25. _TEXT SEGMENT
26. ?ReturningValue@@YAHXZ PROC     ; ReturningValue
27. ; Line 15
28.  mov eax, 2020     ; 000007e4H
29. ; Line 16
30.  ret 0
31. ?ReturningValue@@YAHXZ ENDP     ; ReturningValue
32. _TEXT ENDS
33. END
```

Figure 6.12: *ReturningValue-Optimized.asm*

We can observe in the optimized code that all the unnecessary code lines are removed.

The **main** function shows only 2 instructions:

▼Line 18,20

xor eax, eax
ret 0

EAX is XOR'ed to reset EAX to 0, and the return value is stored in EAX. The RET instruction passes the instruction pointer back to the caller.

The **ReturningValue** function also shows only 2 instructions:

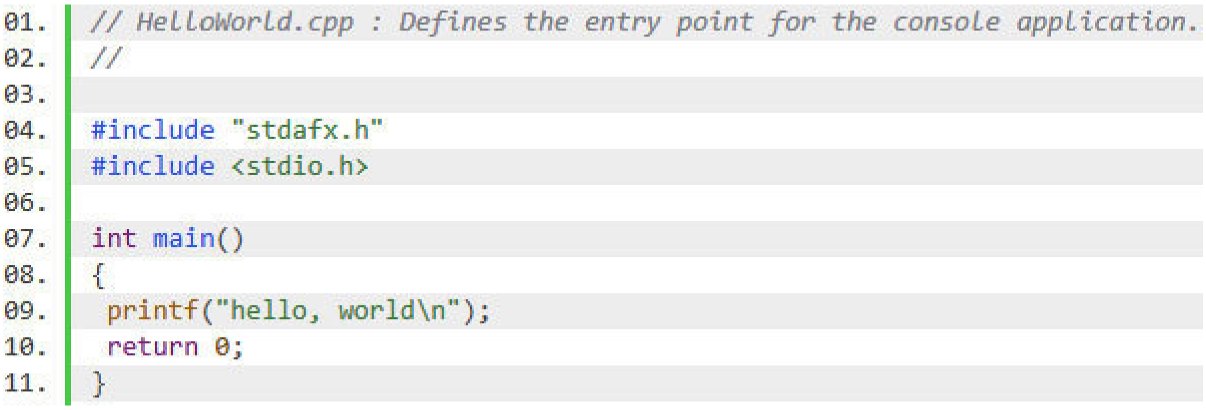
▼Line 28, 30

```
mov eax, 2020      ; 000007e4H
ret 0
```

MOV fills EAX with the return value of 2020 and the RET instruction passes the instruction pointer back to the caller.

Basic “Hello, World” Program

In this simple C/C++ code, we are just printing “hello, world” on the console. We are printing it using the **printf()** function.



```
// HelloWorld.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>

int main()
{
 printf("hello, world\n");
 return 0;
}
```

Figure 6.13: *HelloWorld.cpp*

Basic "Hello, World" Program without Optimization

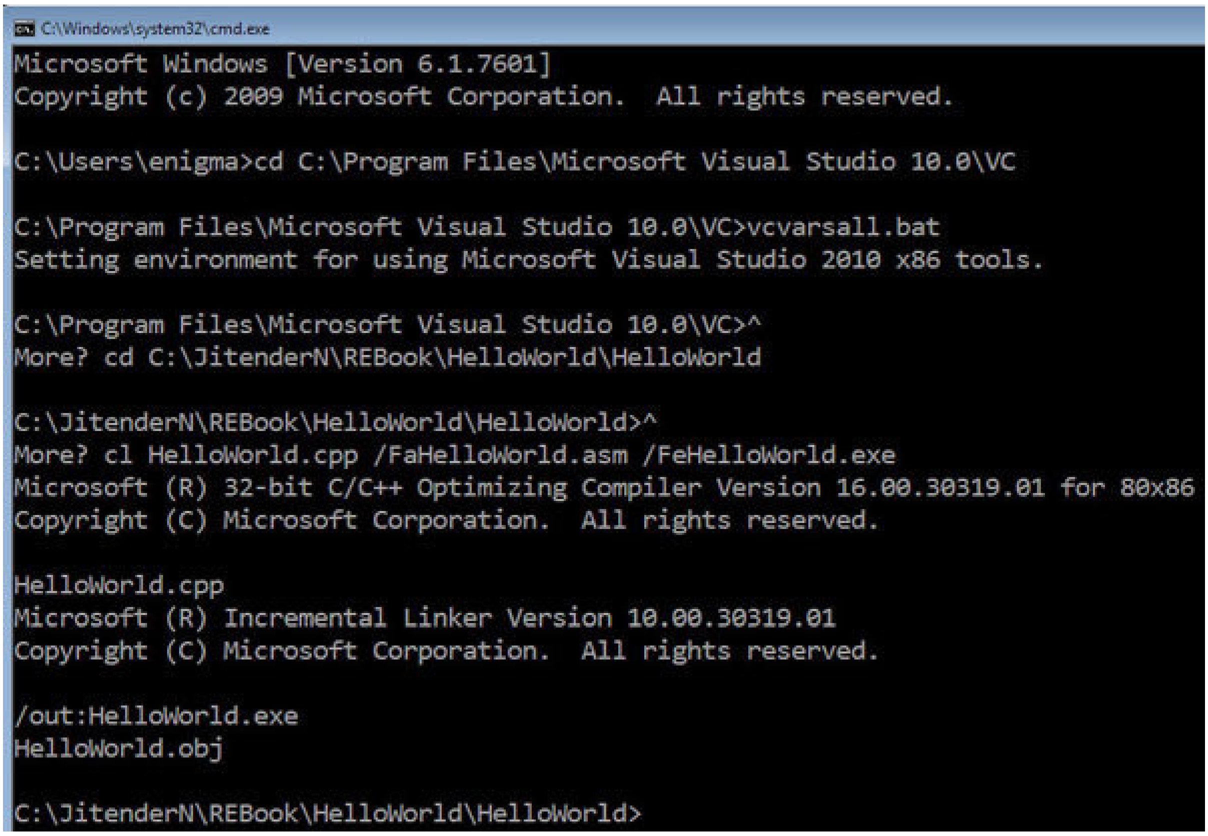
Compile the code without optimization with the MSVC compiler on the x86 platform. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

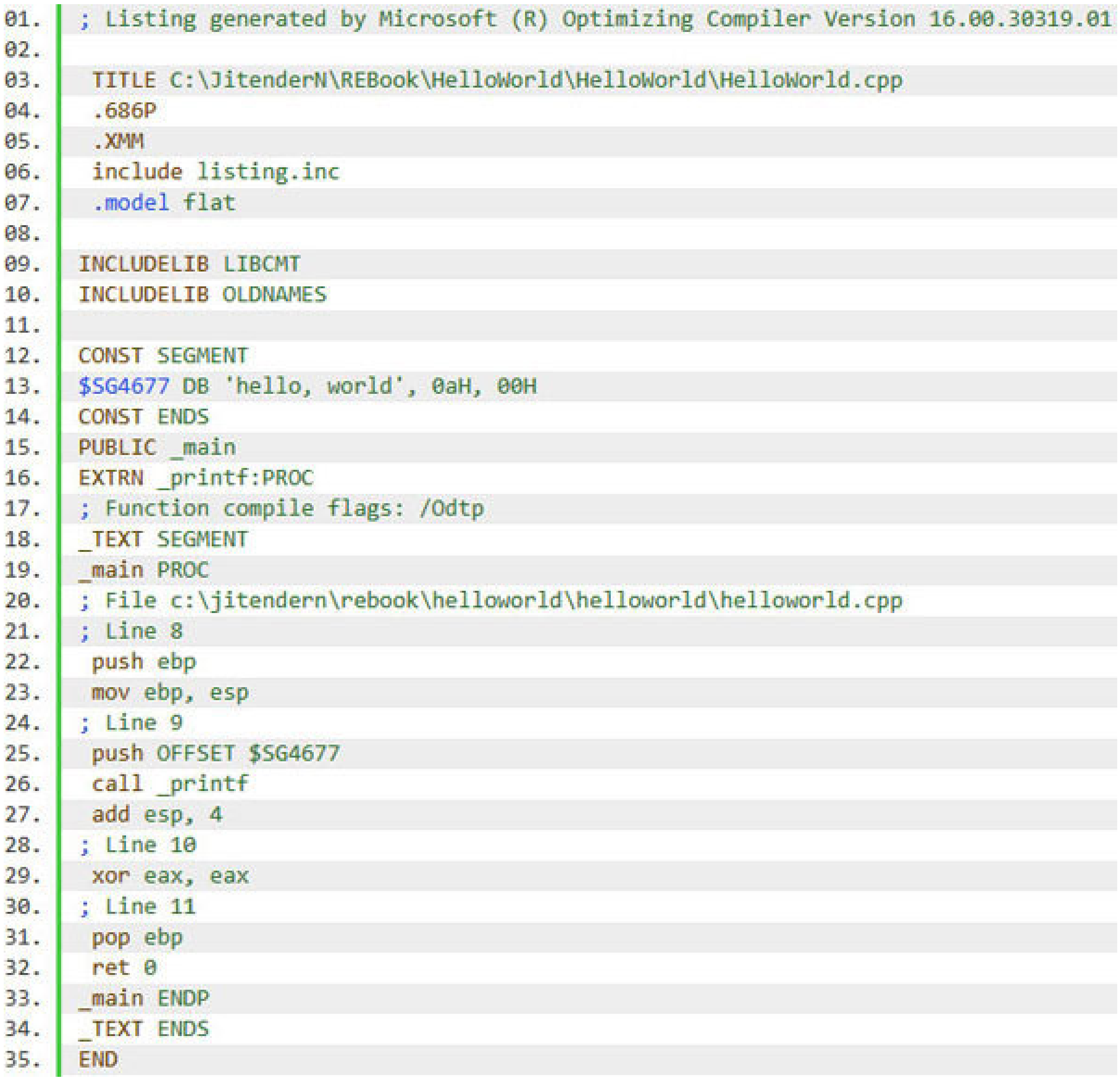
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\HelloWorld\HelloWorld

C:\JitenderN\REBook\HelloWorld\HelloWorld>^
More? cl HelloWorld.cpp /FaHelloWorld.asm /FeHelloWorld.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

HelloWorld.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:HelloWorld.exe
HelloWorld.obj

C:\JitenderN\REBook\HelloWorld\HelloWorld>
```

Figure 6.14: *Basic "Hello, World" program without Optimization*

Following is the generated assembly code which we shall now analyze:



```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\HelloWorld\HelloWorld\HelloWorld.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4677 DB 'hello, world', 0aH, 00H
14. CONST ENDS
15. PUBLIC _main
16. EXTRN _printf:PROC
17. ; Function compile flags: /Odtp
18. _TEXT SEGMENT
19. _main PROC
20. ; File c:\jitendern\rebook\helloworld\helloworld\helloworld.cpp
21. ; Line 8
22.  push ebp
23.  mov ebp, esp
24. ; Line 9
25.  push OFFSET $SG4677
26.  call _printf
27.  add esp, 4
28. ; Line 10
29.  xor eax, eax
30. ; Line 11
31.  pop ebp
32.  ret 0
33. _main ENDP
34. _TEXT ENDS
35. END
```

Figure 6.15: *HelloWorld.asm*

Let's walk through the assembly code line by line:

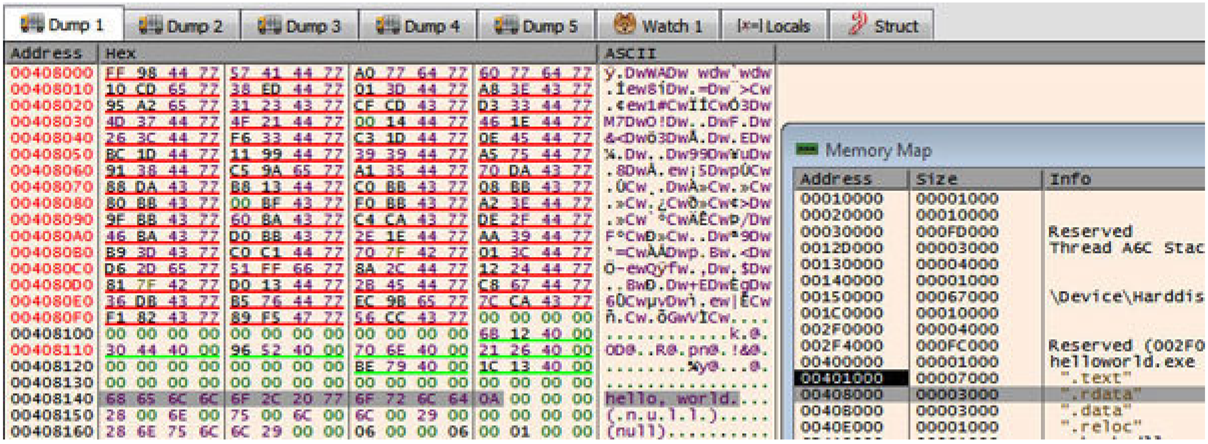
▼Line 1-10

We have already discussed this in the EmptyFunction section.

▼Line 12-14

```
CONST SEGMENT
$SG4677 DB 'hello, world', 0aH, 00H
CONST ENDS
```

The string constant, which in our case is “hello, world”, is allocated in the constant segment. The CONST SEGMENT derivative is used to define the start of the constant segment in the memory. In our case linker renamed **CONST SEGMENT** to which can be dumped using any debugger. In the following screenshot, you can see that we have opened the EXE file (generated after compilation) in x32dbg and in this EXE file, we have disabled the **Address Space Layout Randomization** using CFF Explorer. Follow the steps mentioned in the Appendix to disable ASLR on an EXE file. Disabling ASLR on an EXE file will help us load the EXE file on the same memory space every time the EXE file is opened in the debugger:



```
Address   Hex                                              ASCII
00408000  FF 98 44 77 57 41 44 77 A0 77 64 77 60 77 64 77  ÿ.DwWADw wdw`wdw
00408010  10 CD 65 77 38 ED 44 77 01 3D 44 77 A8 3E 43 77  .Ìew8íDw.=Dw¨>Cw
00408020  95 A2 65 77 31 23 43 77 CF CD 43 77 D3 33 44 77  .¢ew1#CwÏÍCwÓ3Dw
00408030  4D 37 44 77 4F 21 44 77 00 14 44 77 46 1E 44 77  M7DwO!Dw..DwF.Dw
00408040  26 3C 44 77 F6 33 44 77 C3 1D 44 77 0E 45 44 77  &<Dwö3DwÃ.Dw.EDw
00408050  BC 1D 44 77 11 99 44 77 39 39 44 77 A5 75 44 77  ¼.Dw..Dw99Dw¥uDw
00408060  91 38 44 77 C5 9A 65 77 A1 35 44 77 70 DA 43 77  .8DwÅ.ew¡5DwpÚCw
00408070  88 DA 43 77 B8 13 44 77 C0 BB 43 77 08 BB 43 77  .ÚCw¸.DwÀ»Cw.»Cw
00408080  80 BB 43 77 00 BF 43 77 F0 BB 43 77 A2 3E 44 77  .»Cw.¿Cwð»Cw¢>Dw
00408090  9F BB 43 77 60 BA 43 77 C4 CA 43 77 DE 2F 44 77  .»Cw`ºCwÄÊCwÞ/Dw
004080A0  46 BA 43 77 D0 BB 43 77 2E 1E 44 77 AA 39 44 77  FºCwÐ»Cw..Dwª9Dw
004080B0  B9 3D 43 77 C0 C1 44 77 70 7F 42 77 01 3C 44 77  ¹=CwÀÁDwp.Bw.<Dw
004080C0  D6 2D 65 77 51 FF 66 77 8A 2C 44 77 12 24 44 77  Ö-ewQÿfw.,Dw.$Dw
004080D0  81 7F 42 77 D0 13 44 77 28 45 44 77 C8 67 44 77  ..BwÐ.Dw(EDwÈgDw
004080E0  36 DB 43 77 B5 76 44 77 EC 9B 65 77 7C CA 43 77  6ÛCwµvDwì.ew|ÊCw
004080F0  F1 82 43 77 89 F5 47 77 56 CC 43 77 00 00 00 00  ñ.Cw.õGwVÌCw....
00408100  00 00 00 00 00 00 00 00 00 00 00 00 68 12 40 00  ............h.@.
00408110  30 44 40 00 96 52 40 00 70 6E 40 00 21 26 40 00  0D@..R@.pn@.!&@.
00408120  00 00 00 00 00 00 00 00 BE 79 40 00 1C 13 40 00  ........¾y@...@.
00408130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00408140  68 65 6C 6C 6F 2C 20 77 6F 72 6C 64 0A 00 00 00  hello, world....
00408150  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
00408160  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
```

```
Memory Map
Address   Size      Info
00010000  00001000
00020000  00010000
00030000  000FD000  Reserved
0012D000  00003000  Thread A6C Stac
00130000  00004000
00140000  00001000
00150000  00067000  \Device\Harddis
001C0000  00010000
002F0000  00004000
002F4000  000FC000  Reserved (002F0
00400000  00001000  helloworld.exe
00401000  00007000   ".text"
00408000  00003000   ".rdata"
00408000  00003000   ".data"
0040E000  00001000   ".reloc"
```

Figure 6.16: .rdata

$SG4677 is the internal name given by a compiler to handle the string constant. DB, Defines the byte, which is the data type.

'hello, world', 0aH, 00H is the string data, which is null terminated ASCII string.

By **CONST** the constant segment is ended.

▼Line 15

PUBLIC _main

PUBLIC is the derivative which makes the **_main** procedure public, to be accessed by other modules.

▼Line 16

EXTRN _printf:PROC

Note: A code is placed in the .code segment, a constant string is placed in the CONST (.rdata) segment and if it is not a constant, it is placed in the .data segment.

EXTRN derivative declares the **extern** function, which is **printf** in our case. All functions begin with an underscore.

▼Line 18

_TEXT SEGMENT

This starts the **_TEXT** segment or code segment, where our **main** function code resides.

▼Line 19

```
_main PROC
```

This is the start of the **main** procedure.

▼Line 20-23

```
; File c:\jitendern\rebook\helloworld\helloworld\helloworld.cpp
; Line 8
push ebp
mov ebp, esp
```

Here, it's the same as we discussed earlier that all comments begin with semicolons. One comment is stating the C/C++ source code file path and the other comment is defining the line number in C/C++ source code is mapped with the instruction following this comment. With the **main** function prologue code starts.

▼Line 25-27

```
push OFFSET $SG4677
call _printf
add esp, 4
```

Before calling the **printf** function, we push the pointer to our constant string onto stack with the help of the **PUSH** instruction. The **CALL** instruction is calling the **printf** function.

After the execution of the **printf** function, the control is transferred back to the caller, which is **main** function in our case. Throughout the execution of the **printf** function, the pointer to the string will be on the stack. So, when the execution is returned back to the **main** function, the stack needs to be cleaned as we don't need the string pointer anymore.

Since we are following the CDECL calling convention, it is the caller's responsibility to clean up stack, which in our case is done using the **add esp, 4** instruction.

A 32-bit program uses 4 bytes for addressing. So, when we add 4 bytes to ESP, we increment ESP by 4 bytes to clean up the stack and remove the constant string pointer on the stack. An equivalent of ADD instruction can also be **POP** which is often used by other compilers.

▼Line 28-29

```
; Line 10
xor eax, eax
```

As **main** is returning 0 in C/C++ code and we know that the return value of the function is stored in the EAX register, EAX is XOR'ed to return 0.

▼Line 30-32

```
; Line 11
pop ebp
ret 0
```

It is calling the **main** function epilogue code and the **RET** instruction returns execution back to the caller, where the caller can take the return value from the EAX register.

▼Line 33

```
_main ENDP
```

With this, the _main function is closed.

▼Line 34-35

```
_TEXT ENDS
END
```

This is ending the code segment and source code.

Basic “Hello, World” Program with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

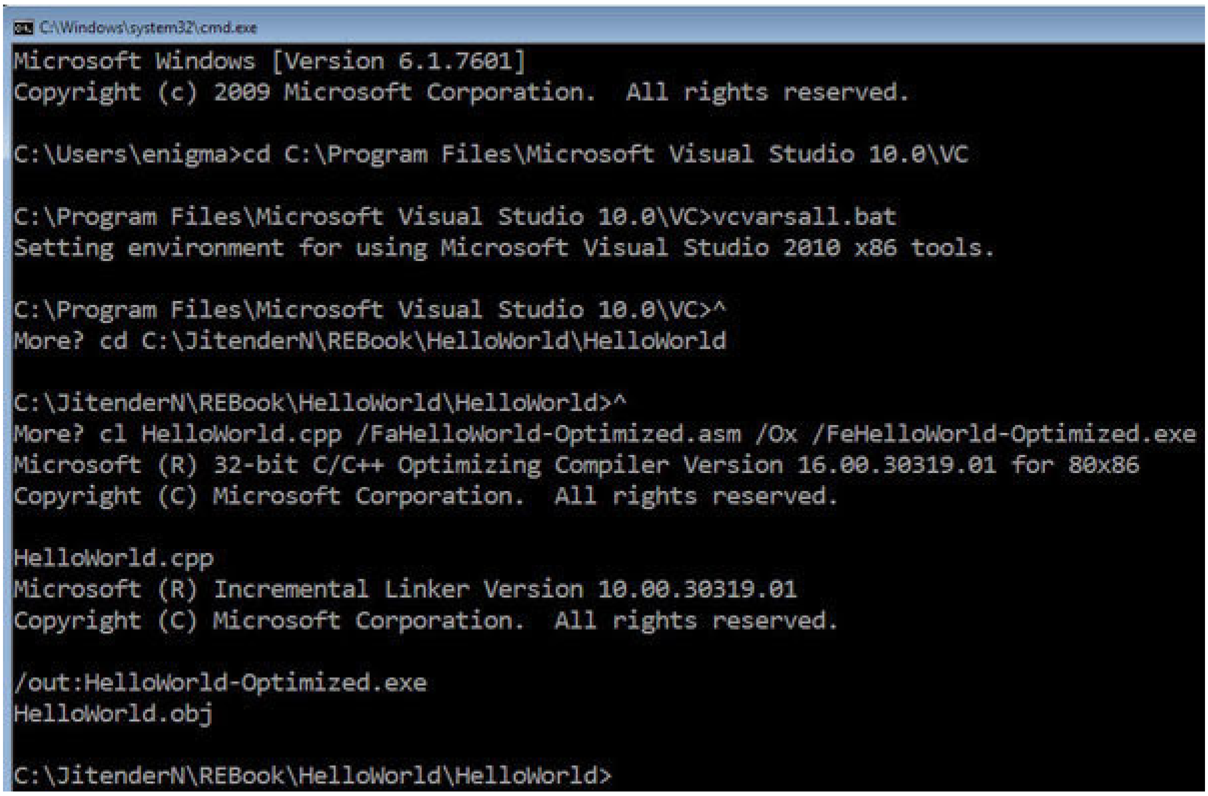
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

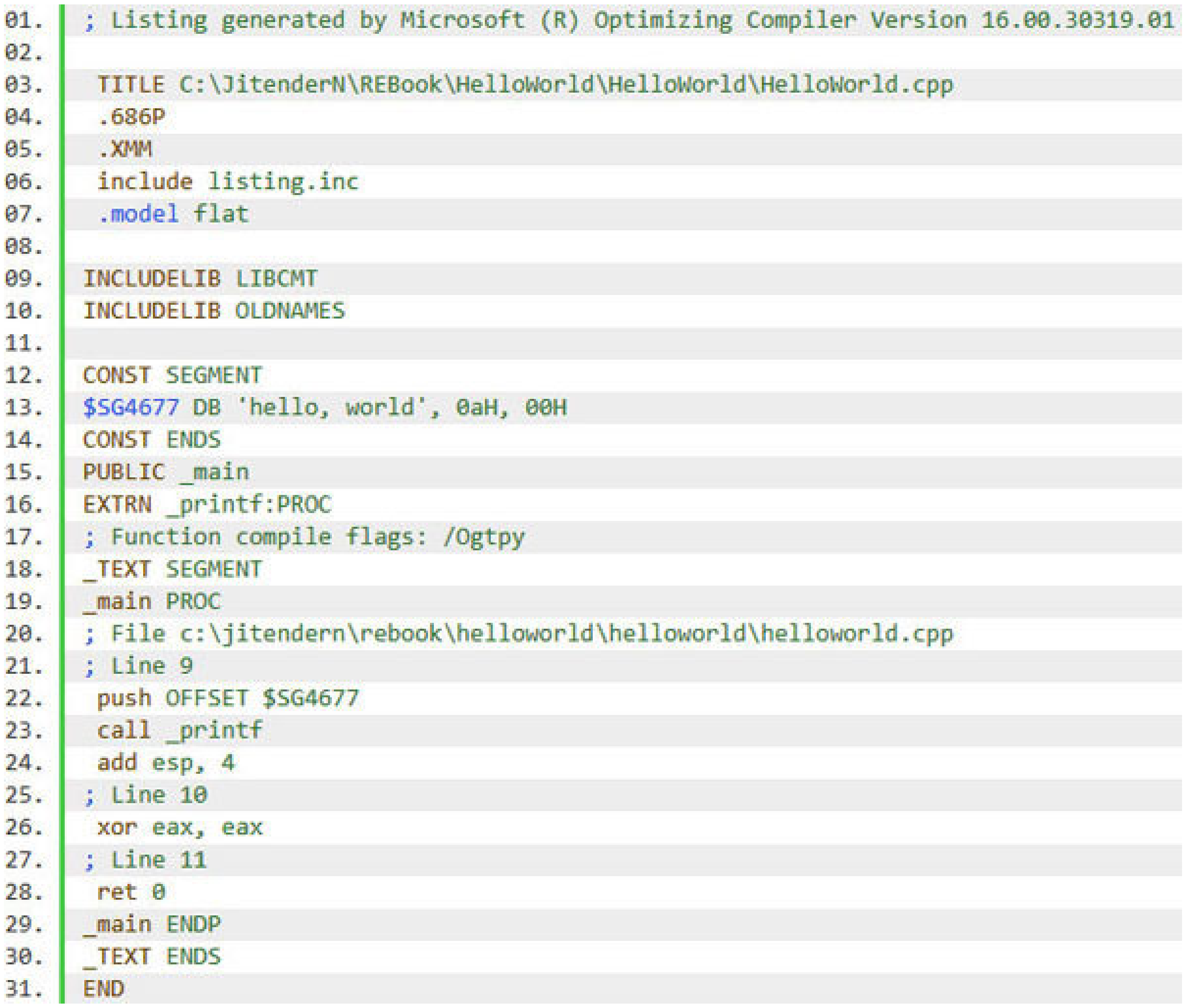
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\HelloWorld\HelloWorld

C:\JitenderN\REBook\HelloWorld\HelloWorld>^
More? cl HelloWorld.cpp /FaHelloWorld-Optimized.asm /Ox /FeHelloWorld-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

HelloWorld.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:HelloWorld-Optimized.exe
HelloWorld.obj

C:\JitenderN\REBook\HelloWorld\HelloWorld>
```

Figure 6.17: *Basic "Hello, World" program with Optimization*

The generate assembly code will be as follows:



```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\HelloWorld\HelloWorld\HelloWorld.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4677 DB 'hello, world', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_main PROC
; File c:\jitendern\rebook\helloworld\helloworld\helloworld.cpp
; Line 9
 push OFFSET $SG4677
 call _printf
 add esp, 4
; Line 10
 xor eax, eax
; Line 11
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 6.18: *HelloWorld-Optimized.asm*

All the code lines are the same as we discussed in preceding section. The only difference is that the function prologue and the epilogue code are removed by the compilers to consume fewer resources (Memory, CPU, and so on).

All the rest of the instructions are the same as we discussed in the without optimization section. Point to note here is that the meaning of the code is the same as that in the without optimization code. We are pushing the pointer to the constant string on the stack to call the **printf** function.

The **printf** function upon execution returns the execution back to **main** (caller) to clean up stack and fill EAX with 0 to return what is in the EAX.

Conclusion

In this chapter we understood the concept of code optimization. We took examples of empty function, function returning value and printing “hello, world” programs to understand the assembly listing of optimized and non-optimized code. In the next chapter we will talk about the code optimization concept on the programs with printf function and also we will discuss on how Integer, Float and char variables are stored in the memory.

CHAPTER 7

Reverse Engineering Pattern of Printf Program

Every time we write a program, we use the printf function to print something or the other on the output screen. It can be something for the end user of the program or it can be something for the debugging purpose or a simple welcome message. The same logic is followed by malware or virus writers. Programs are coded to print something or the other using the printf function. So as a reverse engineer, we should understand the printf function pattern while reversing any program coded to behave as a virus or a malware. Most of the virus or malware writers while coding print something or the other for their own purpose or for the target to act upon.

So, it is important to understand the programs that use the printf function to print messages on the console. Along with this, we will also discuss the usage of printf with different variables. Different types of variables allocate different amounts of memory, which is quite interesting to know. In this chapter, we take C/C++ program that uses the printf function to print integer, float, and char on the console. Each program will be taken separately to understand the pattern of the printf program when reverse engineered.

Structure

In this chapter, we will cover the following topics:

Understanding the assembly pattern of printf with Integer

Understanding the assembly pattern of printf with Float

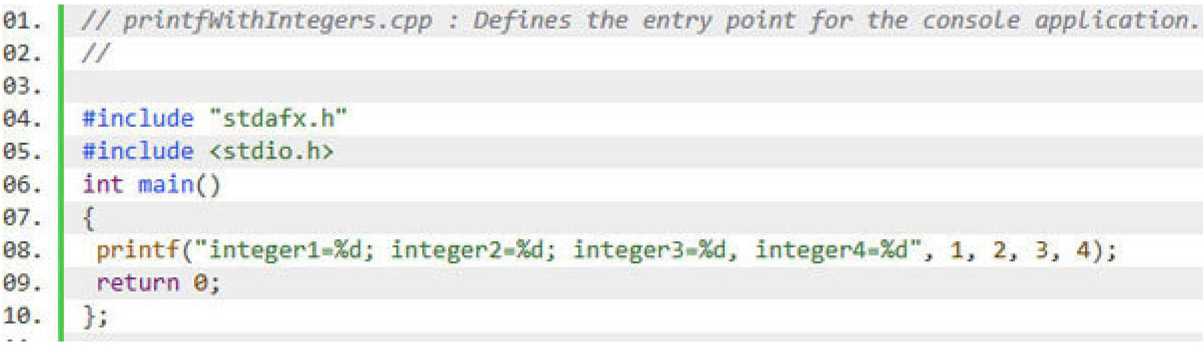
Understanding the assembly pattern of printf with Char

Objective

After studying this chapter, we will be able to understand the code optimization concept on programs with the printf function. We will understand how assembly code, with optimization, is different from without optimization. During this, we will also discuss how Integer, Float, and Char variables are stored in memory. A floating point variable takes a different approach in working as compared to integer or char. We will cover the approach with detailed examples.

Function printf with Integers

In this simple C/C++ code, we are printing 4 integers on the console. We are using the printf function to print integers.



```
// printfWithIntegers.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
int main()
{
 printf("integer1=%d; integer2=%d; integer3=%d, integer4=%d", 1, 2, 3, 4);
 return 0;
};
```

Figure 7.1: *printfWithIntegers.cpp*

Function printf Printing Integers without Optimization

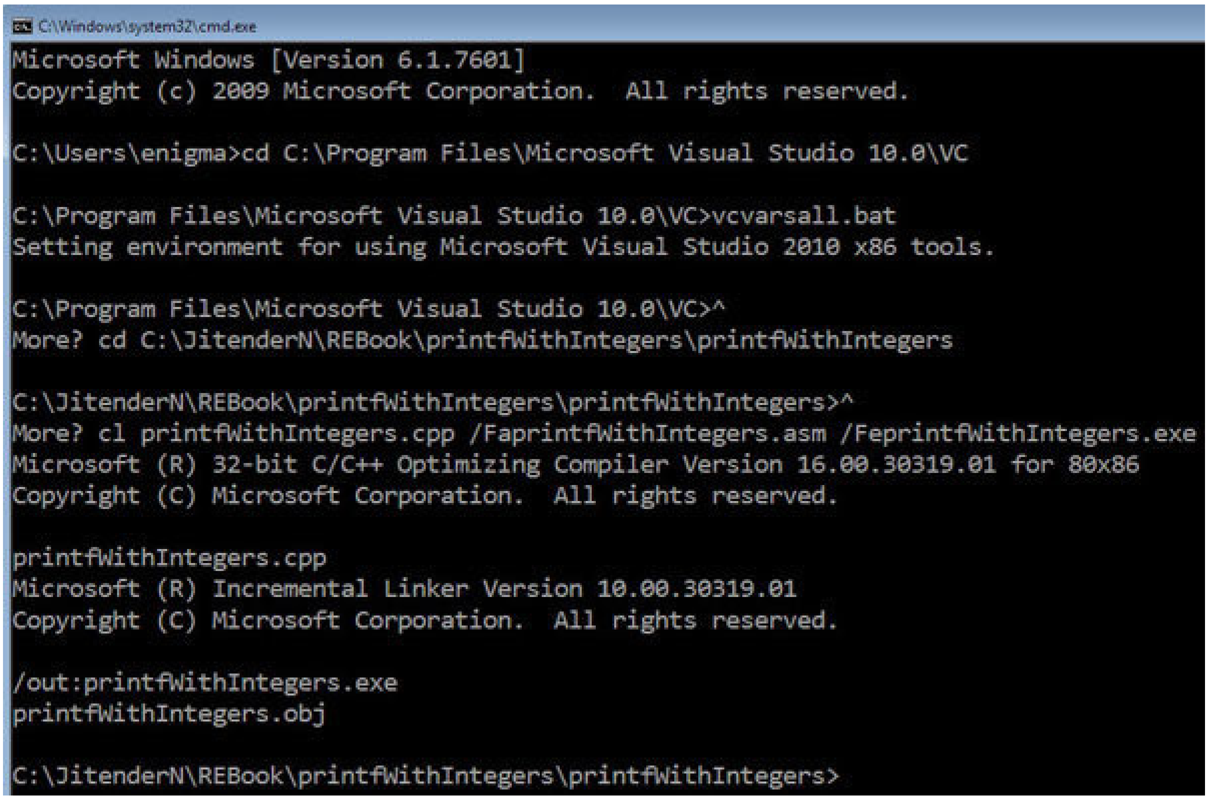
Compile the code without optimization with the MSVC compiler on the x86 platform. Run the following commands on the Windows Command Prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

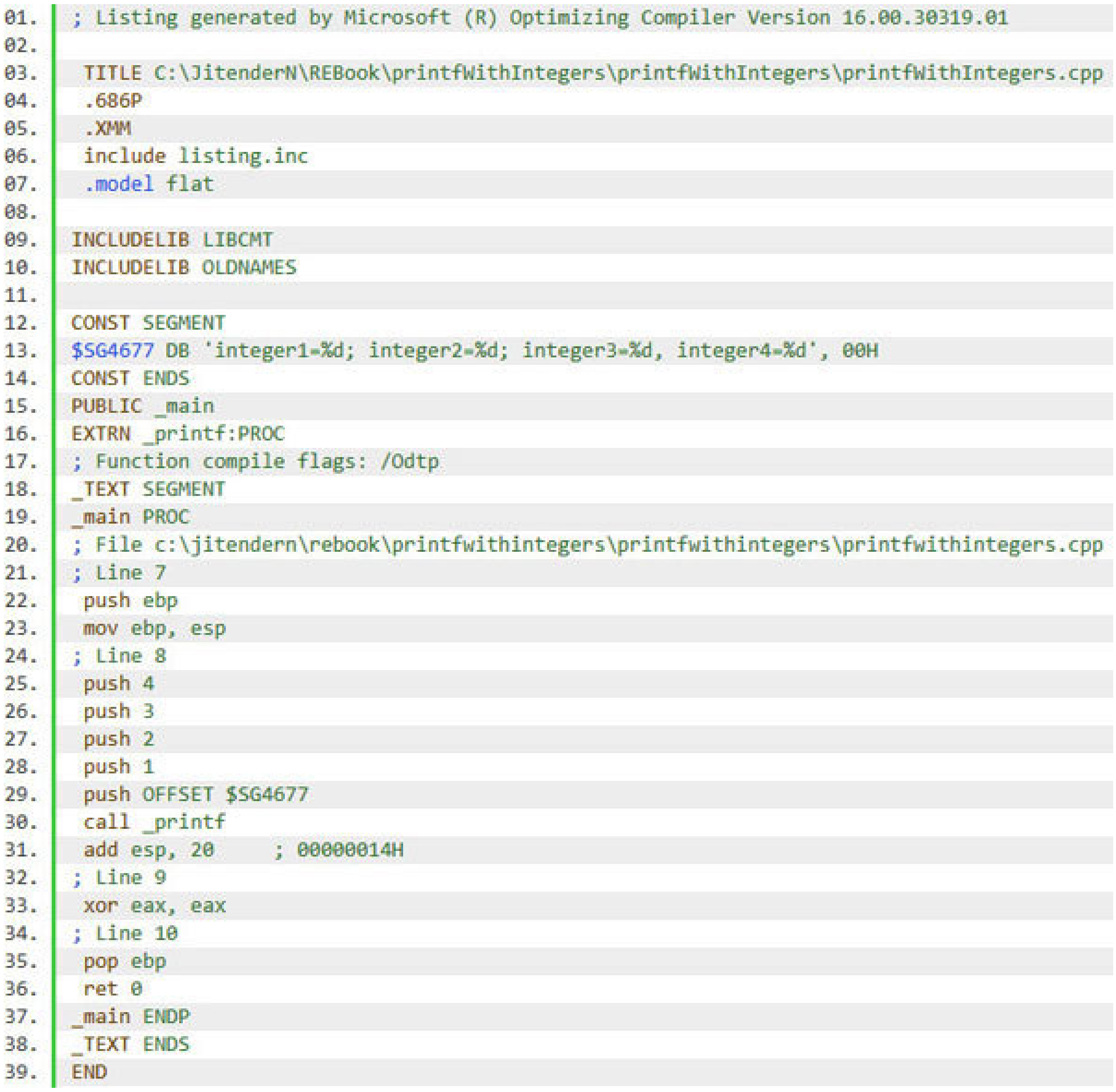
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers

C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers>^
More? cl printfWithIntegers.cpp /FaprintfWithIntegers.asm /FeprintfWithIntegers.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithIntegers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithIntegers.exe
printfWithIntegers.obj

C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers>
```

Figure 7.2: *Function printf printing Integers without optimization*

The generated assembly code will be as follows:



```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers\printfWithIntegers.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4677 DB 'integer1=%d; integer2=%d; integer3=%d, integer4=%d', 00H
14. CONST ENDS
15. PUBLIC _main
16. EXTRN _printf:PROC
17. ; Function compile flags: /Odtp
18. _TEXT SEGMENT
19. _main PROC
20. ; File c:\jitendern\rebook\printfwithintegers\printfwithintegers\printfwithintegers.cpp
21. ; Line 7
22.  push ebp
23.  mov ebp, esp
24. ; Line 8
25.  push 4
26.  push 3
27.  push 2
28.  push 1
29.  push OFFSET $SG4677
30.  call _printf
31.  add esp, 20     ; 00000014H
32. ; Line 9
33.  xor eax, eax
34. ; Line 10
35.  pop ebp
36.  ret 0
37. _main ENDP
38. _TEXT ENDS
39. END
```

Figure 7.3: *printfWithIntegers.asm*

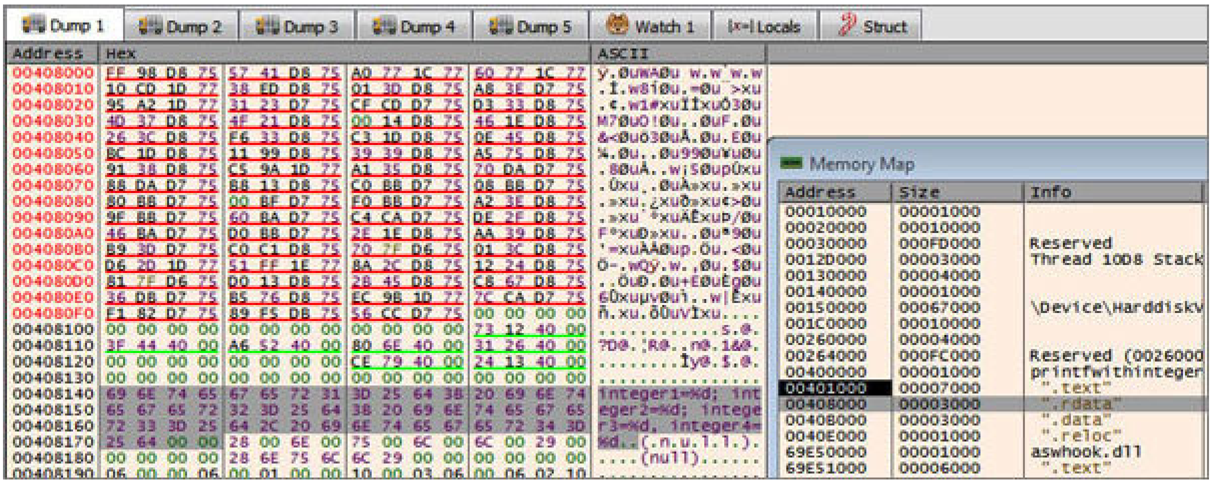
As we have already discussed the initial instructions of assembly code generated in [Chapter 6, Reverse Engineering Pattern of Basic](#) so we will start with line 12.

▼Line 12-14

CONST SEGMENT

$SG4677 DB 'integer1=%d; integer2=%d; integer3=%d, integer4=%d', 00H
CONST ENDS

It is the start and the end of **CONST SEGMENT** in memory, within which the string constant **$SG4677** is defined. This can be viewed in the memory by dumping **.rdata** (linker renamed **CONST SEGMENT** to using any debugger. You can view the string constant using the x32dbg debugger as follows:



```
Address   Hex                                              ASCII
00408000  FF 98 D8 75 57 41 D8 75 A0 77 1C 77 60 77 1C 77  ÿ.ØuWAØu w.w`w.w
00408010  10 CD 1D 77 38 ED D8 75 01 3D D8 75 A8 3E D7 75  .Í.w8íØu.=Øu¨>xu
00408020  95 A2 1D 77 31 23 D7 75 CF CD D7 75 D3 33 D8 75  .¢.w1#xuÏÍxuÓ3Øu
00408030  4D 37 D8 75 4F 21 D8 75 00 14 D8 75 46 1E D8 75  M7ØuO!Øu..ØuF.Øu
00408040  26 3C D8 75 F6 33 D8 75 C3 1D D8 75 DE 45 D8 75  &<Øuö3ØuÃ.ØuÞEØu
00408050  BC 1D D8 75 11 99 D8 75 39 39 D8 75 A5 75 D8 75  ¼.Øu..Øu99Øu¥uØu
00408060  91 38 D8 75 C5 9A 1D 77 A1 35 D8 75 70 DA D7 75  .8ØuÅ..w¡5ØupÚxu
00408070  88 DA D7 75 88 13 D8 75 C0 BB D7 75 08 BB D7 75  .Úxu..ØuÀ»xu.»xu
00408080  80 BB D7 75 00 BF D7 75 F0 BB D7 75 A2 3E D8 75  .»xu.¿xuð»xu¢>Øu
00408090  9F BB D7 75 60 BA D7 75 C4 CA D7 75 DE 2F D8 75  .»xu`ºxuÄÊxuÞ/Øu
004080A0  46 BA D7 75 D0 BB D7 75 2E 1E D8 75 AA 39 D8 75  FºxuÐ»xu..Øuª9Øu
004080B0  B9 3D D7 75 C0 C1 D8 75 70 7F D6 75 01 3C D8 75  ¹=xuÀÁØup.Öu.<Øu
004080C0  D6 2D 1D 77 51 FF 1E 77 8A 2C D8 75 12 24 D8 75  Ö-.wQÿ.w.,Øu.$Øu
004080D0  81 7F D6 75 D0 13 D8 75 28 45 D8 75 C8 67 D8 75  ..ÖuÐ.Øu(EØuÈgØu
004080E0  36 D8 D7 75 B5 76 D8 75 EC 98 1D 77 7C CA D7 75  6ØxuµvØuì..w|Êxu
004080F0  F1 82 D7 75 89 F5 D8 75 56 CC D7 75 00 00 00 00  ñ.xu.õØuVÌxu....
00408100  00 00 00 00 00 00 00 00 00 00 00 00 73 12 40 00  ............s.@.
00408110  3F 44 40 00 A6 52 40 00 80 6E 40 00 31 26 40 00  ?D@.¦R@..n@.1&@.
00408120  00 00 00 00 00 00 00 00 CE 79 40 00 24 13 40 00  ........Îy@.$.@.
00408130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00408140  69 6E 74 65 67 65 72 31 3D 25 64 3B 20 69 6E 74  integer1=%d; int
00408150  65 67 65 72 32 3D 25 64 3B 20 69 6E 74 65 67 65  eger2=%d; intege
00408160  72 33 3D 25 64 2C 20 69 6E 74 65 67 65 72 34 3D  r3=%d, integer4=
00408170  25 64 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  %d..(.n.u.l.l.).
00408180  00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00  ....(null)......
00408190  06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10
```

```
Address   Size      Info
00010000  00001000
00020000  00010000
00030000  000FD000  Reserved
00120000  00003000  Thread 1008 Stack
00130000  00004000
00140000  00001000
00150000  00067000  \Device\Harddiskv
001C0000  00010000
00260000  00004000
00264000  000FC000  Reserved (0026000
00400000  00001000  printfwithinteger
00401000  00007000   ".text"
00408000  00003000   ".rdata"
0040B000  00003000   ".data"
0040E000  00001000   ".reloc"
69E50000  00001000  aswhook.dll
69E51000  00006000   ".text"
```

Figure 7.4: *.rdata*

▼Line 15

PUBLIC _main

All functions begin with an underscore. The **main** function is labeled as public function, which means it can be accessed by other modules.

▼Line 16

```
EXTRN _printf:PROC
```

With the **EXTRN** derivative, it is defining the external symbol of the name _printf and the type procedure.

Tip: Syntax of EXTRN derivative is label:type, where label can be variable/function and Label Type can be as below:

Label Type	Meaning
`BYTE`	`Variable of 8 bits`

WORD	Variable of 16 bits
DWORD	Variable of 32 bits
QWORD	Variable of 64 bits
PROC	Procedure Name

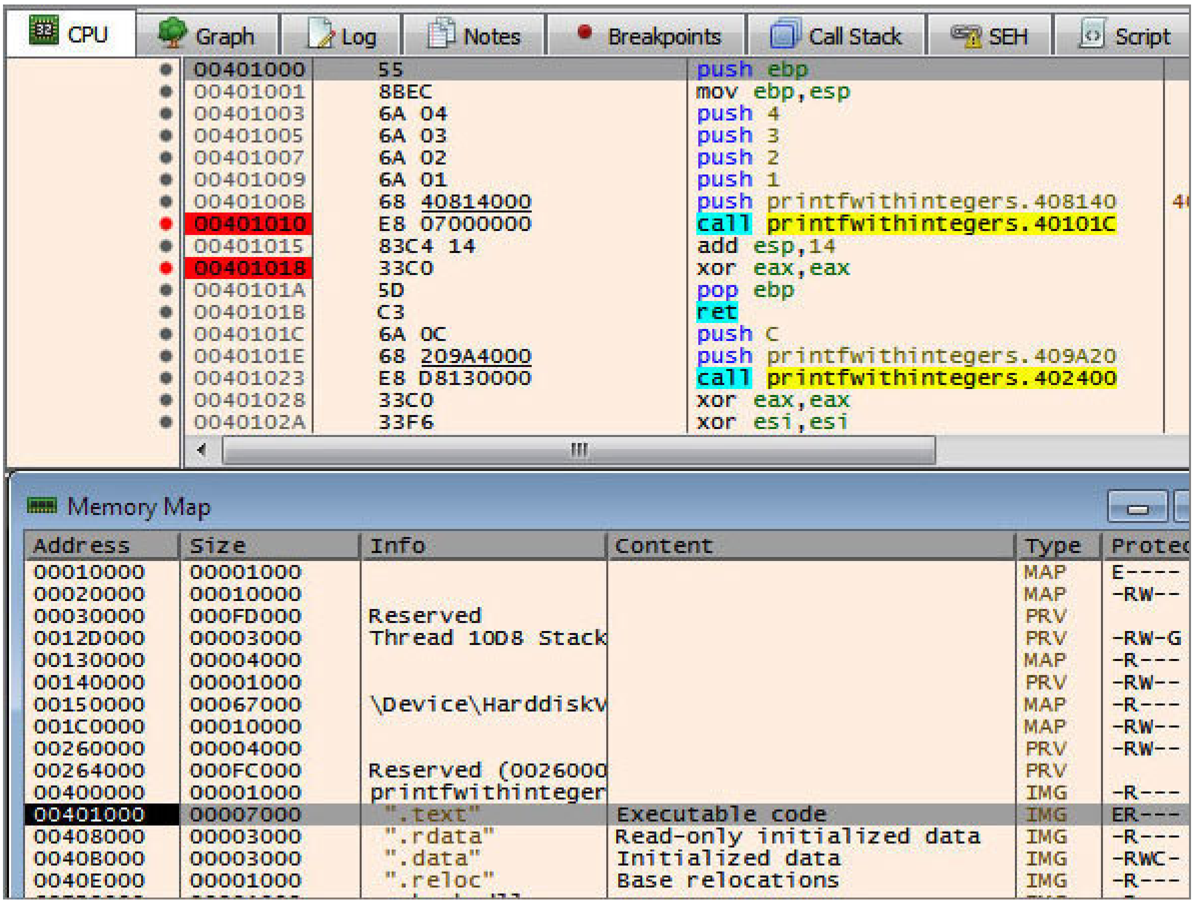
Table 7.1

▼Line 18, 19

```
_TEXT SEGMENT
_main PROC
```

This starts the _TEXT segment where our **main** function code resides. This can also be visualized in the x32dbg debugger:

.text segment of **printfWithIntegers.exe** starts from the **0x00401000** address and we can see that the **main** function/procedure code starts from the same address.



```
00401000   55          push ebp
00401001   8BEC        mov ebp,esp
00401003   6A 04       push 4
00401005   6A 03       push 3
00401007   6A 02       push 2
00401009   6A 01       push 1
0040100B   68 40814000 push printfwithintegers.408140
00401010   E8 07000000 call printfwithintegers.40101C
00401015   83C4 14     add esp,14
00401018   33C0        xor eax,eax
0040101A   5D          pop ebp
0040101B   C3          ret
0040101C   6A 0C       push C
0040101E   68 209A4000 push printfwithintegers.409A20
00401023   E8 D8130000 call printfwithintegers.402400
00401028   33C0        xor eax,eax
0040102A   33F6        xor esi,esi
```

```
Address   Size      Info                 Content                     Type  Protec
00010000  00001000                                                   MAP   E----
00020000  00010000                                                   MAP   -RW--
00030000  000FD000  Reserved                                         PRV
0012D000  00003000  Thread 10D8 Stack                                PRV   -RW-G
00130000  00004000                                                   MAP   -R---
00140000  00001000                                                   PRV   -RW--
00150000  00067000  \Device\HarddiskV                                MAP   -R---
001C0000  00010000                                                   MAP   -RW--
00260000  00004000                                                   PRV   -RW--
00264000  000FC000  Reserved (0026000                                PRV
00400000  00001000  printfwithinteger                                IMG   -R---
00401000  00007000   ".text"             Executable code             IMG   ER---
00408000  00003000   ".rdata"            Read-only initialized data  IMG   -R---
0040B000  00003000   ".data"             Initialized data            IMG   -RWC-
0040E000  00001000   ".reloc"            Base relocations            IMG   -R---
```

Figure 7.5: *.text*

▼Line 20-23

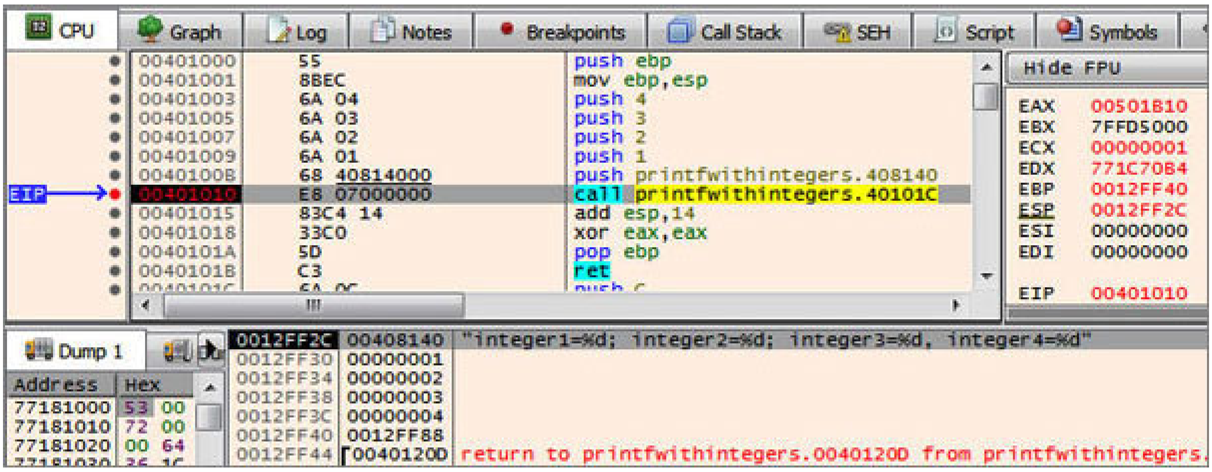
```
; File c:\jitendern\rebook\helloworld\helloworld\helloworld.cpp
; Line 8
push ebp
mov ebp, esp
```

This is the same as earlier. This comment states the C/C++ source code file path and other defining line number in C/C++ source code that is mapped with the instruction following this comment. With **PUSH** instruction, the **main** function prologue code starts.

▼Line 25-30

```
push4
push3
push2
push1
push OFFSET $SG4677
call_printf
```

From here, something interesting begins. All the arguments to printf function are pushed onto the stack in a reverse order. Each argument is of type integer. In a 32-bit environment, each integer occupies 4 bytes in size. To understand the stack state during execution, we can put breakpoint on the call of the printf function in the x32dbg debugger. Once the breakpoint is set, we will run the code to see the stack state when breakpoint is hit. This will help us visualize how the arguments to printf are pushed onto the stack.



```
00401000   55            push ebp
00401001   8BEC          mov ebp,esp
00401003   6A 04         push 4
00401005   6A 03         push 3
00401007   6A 02         push 2
00401009   6A 01         push 1
0040100B   68 40814000   push printfwithintegers.408140
00401010   E8 07000000   call printfwithintegers.40101C
00401015   83C4 14       add esp,14
00401018   33C0          xor eax,eax
0040101A   5D            pop ebp
0040101B   C3            ret
```

```
EAX   00501B10
EBX   7FFD5000
ECX   00000001
EDX   771C70B4
EBP   0012FF40
ESP   0012FF2C
ESI   00000000
EDI   00000000

EIP   00401010
```

```
77181000  53 00
77181010  72 00
77181020  00 64
```

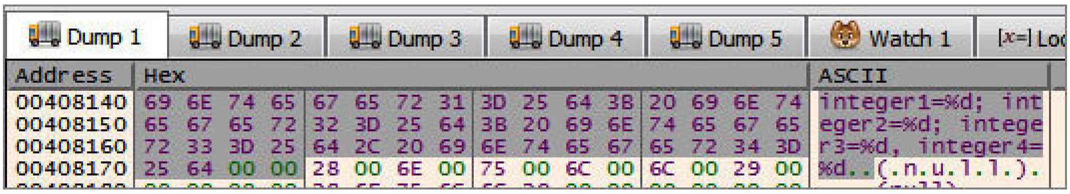
```
0012FF2C 00408140 "integer1=%d; integer2=%d; integer3=%d, integer4=%d"
0012FF30 00000001
0012FF34 00000002
0012FF38 00000003
0012FF3C 00000004
0012FF40 0012FF88
0012FF44 [00401200 return to printfwithintegers.0040120D from printfwithintegers.
```

Figure 7.6: Breakpoint on the call of printf

We can see that the argument integer 4 is pushed first on the stack and then 3, 2, and 1 are pushed. The following is the explanation for the stack state when breakpoint is hit:

0012FF2C 00408140 "integer1=%d; integer2=%d; integer3=%d, integer4=%d"
0012FF30 00000001 Argument 1 of type int is pushed on stack
0012FF34 00000002 Argument 2 of type int is pushed on stack
0012FF38 00000003 Argument 3 of type int is pushed on stack
0012FF3C 00000004 Argument 4 of type int is pushed on stack

At the **0x0012FF2C** location, the pointer to the constant string (which is is pushed on the stack. We can dump the location in x32dbg to view the constant string,



```
Address   Hex                                                ASCII
00408140  69 6E 74 65 67 65 72 31 3D 25 64 3B 20 69 6E 74  integer1=%d; int
00408150  65 67 65 72 32 3D 25 64 3B 20 69 6E 74 65 67 65  eger2=%d; intege
00408160  72 33 3D 25 64 2C 20 69 6E 74 65 67 65 72 34 3D  r3=%d, integer4=
00408170  25 64 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  %d..(.n.u.l.l.).
```

Figure 7.7: *String dumped*

Once all the arguments are pushed on the stack, the call to the printf function is made by:

▼Line 30

call _printf

This will execute the printf function and after the execution of the printf function, the instruction pointer will return the execution pointer back to the caller.

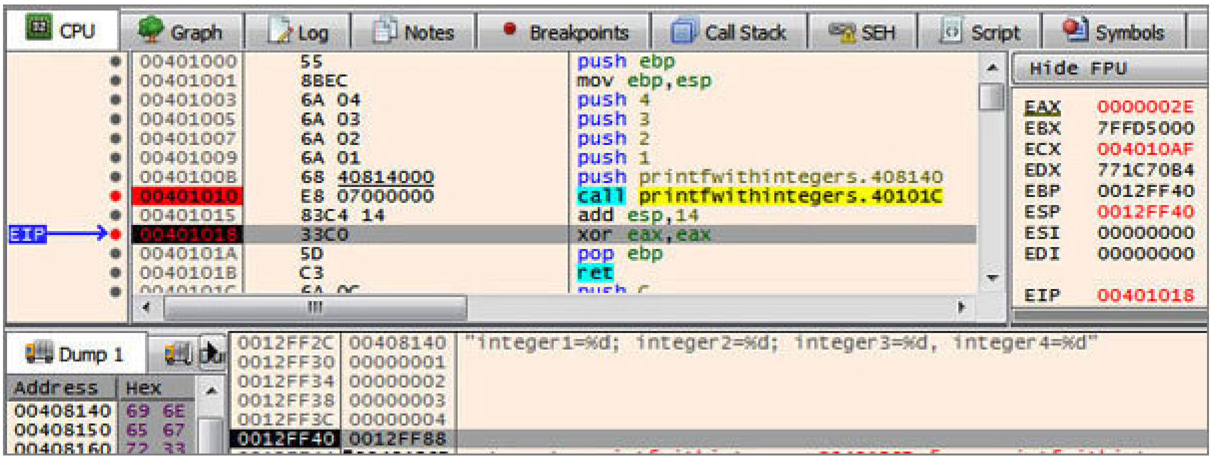
▼Line 31

add esp, 20   ; 00000014H

As we are using the CDECL calling convention, the caller cleans up the stack. So, after returning from the printf function, **add esp, 20** shrinks the stack by 0x20.

20 bytes is calculated by adding the size of 4 arguments, 4 bytes each plus one pointer argument to the constant string of 4 bytes size. This makes a total of 4x5 bytes, which is equal to 20 bytes in Hex).

Now, we will understand something more interesting related to garbage on the stack. To understand the concept of garbage, we put breakpoint on the instruction next to **add esp,** We see something like the following on the stack:



```
00401000   55          push ebp
00401001   8BEC        mov ebp,esp
00401003   6A 04       push 4
00401005   6A 03       push 3
00401007   6A 02       push 2
00401009   6A 01       push 1
0040100B   68 40814000 push printfwithintegers.408140
00401010   E8 07000000 call printfwithintegers.40101C
00401015   83C4 14     add esp,14
00401018   33C0        xor eax,eax
0040101A   5D          pop ebp
0040101B   C3          ret
```

```
EAX 0000002E
EBX 7FFD5000
ECX 004010AF
EDX 771C70B4
EBP 0012FF40
ESP 0012FF40
ESI 00000000
EDI 00000000

EIP 00401018
```

```
00408140 69 6E
00408150 65 67
```

```
0012FF2C 00408140 "integer1=%d; integer2=%d; integer3=%d, integer4=%d"
0012FF30 00000001
0012FF34 00000002
0012FF38 00000003
0012FF3C 00000004
0012FF40 0012FF88
```

***Figure 7.8:** Garbage on stack*

The caller, as per the CDECL calling convention, is responsible for cleaning the stack, which is done using the ADD instruction by moving ESP back by 20 bytes. As we can see, ESP is moved backed to but all the arguments and pointer to the constant string are still on the stack. These values are not cleared or set to zeros. Everything above the ESP value is garbage with no meaning.

▼Line 34-39

```
; Line 9
xor eax, eax
; Line 10
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

All the remaining instructions are the same as we discussed in the chapters before. The **main** function is returning a zero by making EAX to zero.

With END, everything is closed and the program is ended.

Function printf Printing Integers with Optimization

Compile the code with optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

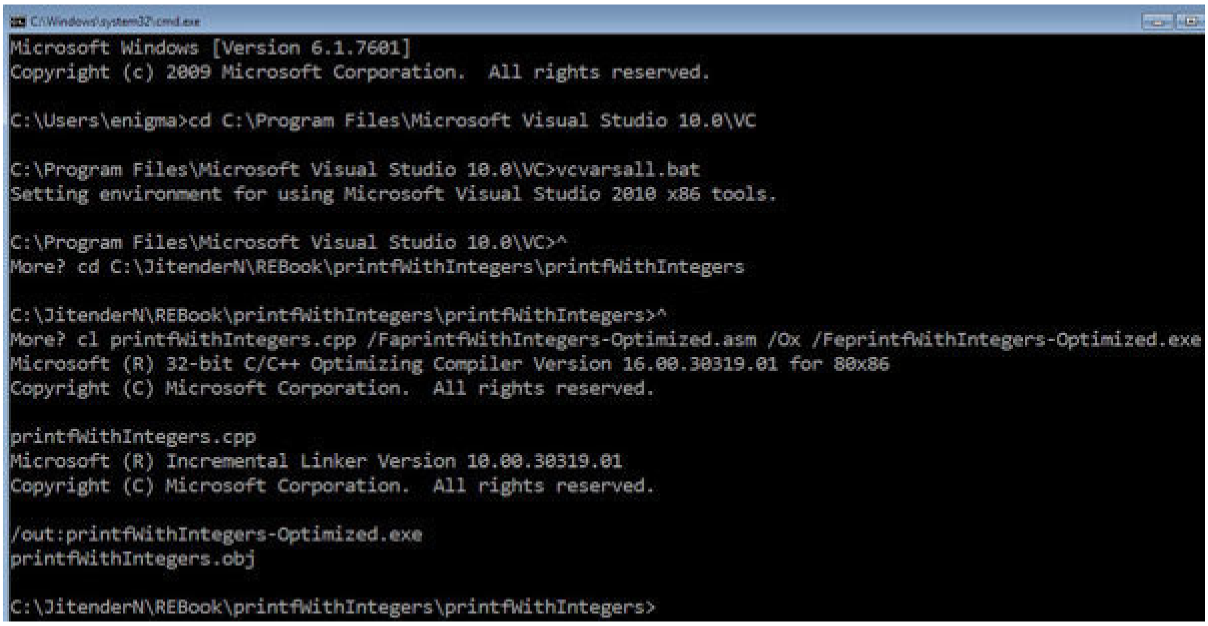
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

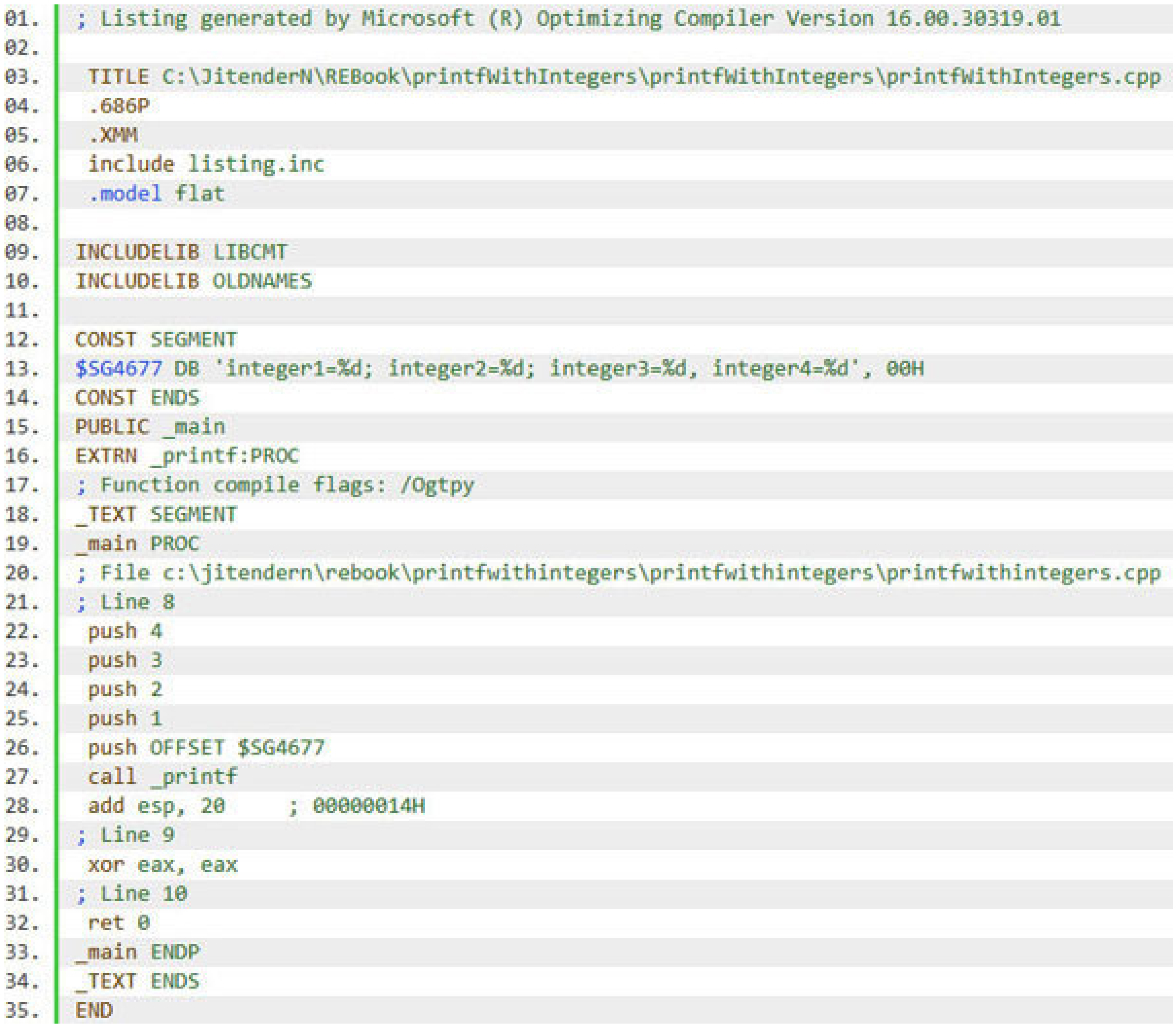
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers

C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers>^
More? cl printfWithIntegers.cpp /FaprintfWithIntegers-Optimized.asm /Ox /FeprintfWithIntegers-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithIntegers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithIntegers-Optimized.exe
printfWithIntegers.obj

C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers>
```

Figure 7.9: *Function printf printing integers with optimization*

The generated assembly code will be as follows:



```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\printfWithIntegers\printfWithIntegers\printfWithIntegers.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4677 DB 'integer1=%d; integer2=%d; integer3=%d, integer4=%d', 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_main PROC
; File c:\jitendern\rebook\printfwithintegers\printfwithintegers\printfwithintegers.cpp
; Line 8
 push 4
 push 3
 push 2
 push 1
 push OFFSET $SG4677
 call _printf
 add esp, 20     ; 00000014H
; Line 9
 xor eax, eax
; Line 10
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 7.10: *printfWithIntegers-Optimized.asm*

In the optimized code, everything is the same except the **main** function prologue and the epilogue code is removed by compilers to consume fewer resources.

Function printf with Float

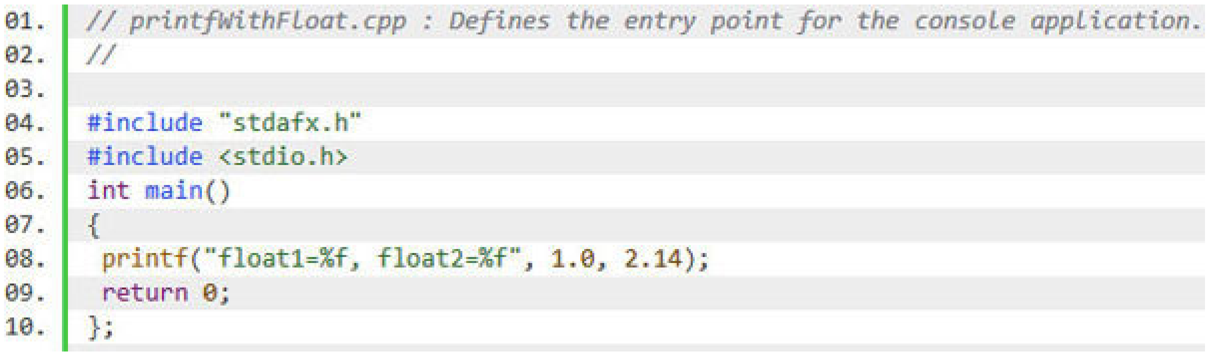
Most of the calculations are done using integers, but when it comes to accuracy, floating point plays an important role. The earlier x86 processor family has separate coprocessors for mathematical calculations that process floating point numbers. But later on, the capability of handling floating point numbers was integrated into the microprocessor itself. This unit which was integrated into the microprocessor to handle the floating point numbers is called the Floating Point Unit (FPU). Now to handle the floating point, two things are required:

There should be space to store the floating point numbers.

There must be instructions to handle and do operations on the floating point numbers.

Now, regarding the space to store floating point numbers, FPU has 8 registers that forms a stack, i.e., from ST0 to ST7. FPU is also referred to as the “x87” section or “FPU Register Stack” the "x87 Stack". Instructions to handle the floating point numbers are referred to as the "x87 instruction set".

The floating point numbers are generally 32-bit long for float type and 64-bit long for double type. So, to maintain maximum accuracy of the floating numbers, the FPU stack registers are 80-bit wide. To understand more about floating point, we will take a simple C/C++ code to print two floating numbers on the console. We are using printf to print the floating numbers.



```cpp
// printfWithFloat.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
int main()
{
 printf("float1=%f, float2=%f", 1.0, 2.14);
 return 0;
};
```

Figure 7.11: *printfWithFloat.cpp*

Function printf Printing Float without Optimization

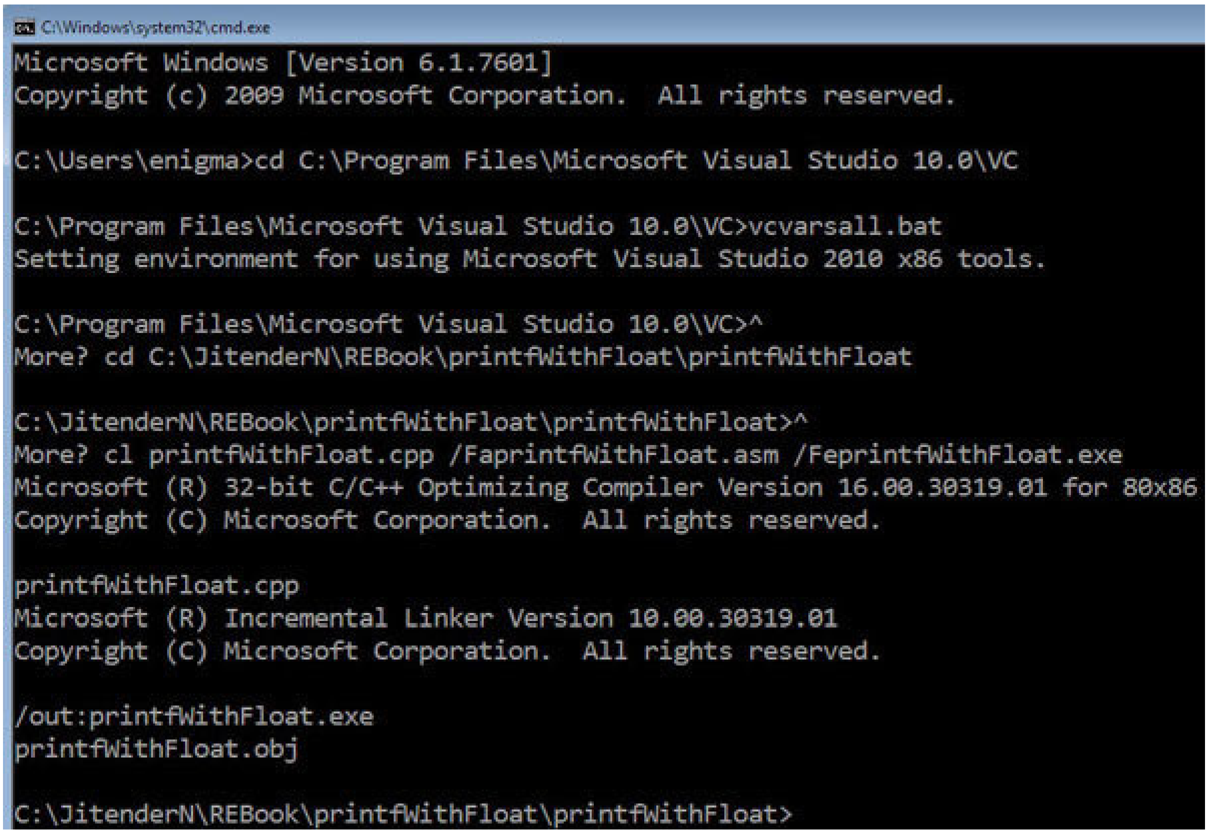
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithFloat\printfWithFloat

C:\JitenderN\REBook\printfWithFloat\printfWithFloat>^
More? cl printfWithFloat.cpp /FaprintfWithFloat.asm /FeprintfWithFloat.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithFloat.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithFloat.exe
printfWithFloat.obj

C:\JitenderN\REBook\printfWithFloat\printfWithFloat>
```

Figure 7.12: *Function printf printing Float without Optimization*

The generated assembly code will be as follows:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\printfWithFloat\printfWithFloat\printfWithFloat.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4677 DB 'float1=%f, float2=%f', 00H
CONST ENDS
PUBLIC __real@3ff0000000000000
PUBLIC __real@40011eb851eb851f
PUBLIC _main
EXTRN _printf:PROC
EXTRN __fltused:DWORD
; COMDAT __real@3ff0000000000000
; File c:\jitendern\rebook\printfwithfloat\printfwithfloat\printfwithfloat.cpp
CONST SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r ; 1
CONST ENDS
; COMDAT __real@40011eb851eb851f
CONST SEGMENT
__real@40011eb851eb851f DQ 040011eb851eb851fr ; 2.14
; Function compile flags: /Odtp
CONST ENDS
_TEXT SEGMENT
_main PROC
; Line 7
 push ebp
 mov ebp, esp
; Line 8
 sub esp, 8
 fld QWORD PTR __real@40011eb851eb851f
 fstp QWORD PTR [esp]
 sub esp, 8
 fld1
 fstp QWORD PTR [esp]
 push OFFSET $SG4677
 call _printf
 add esp, 20     ; 00000014H
; Line 9
 xor eax, eax
; Line 10
 pop ebp
 ret 0
_main ENDP
_TEXT ENDS
END
```

***Figure 7.13:** printfWithFloat.asm*

Let’s start understanding the assembly code generated line by line:

▼Line 1-10

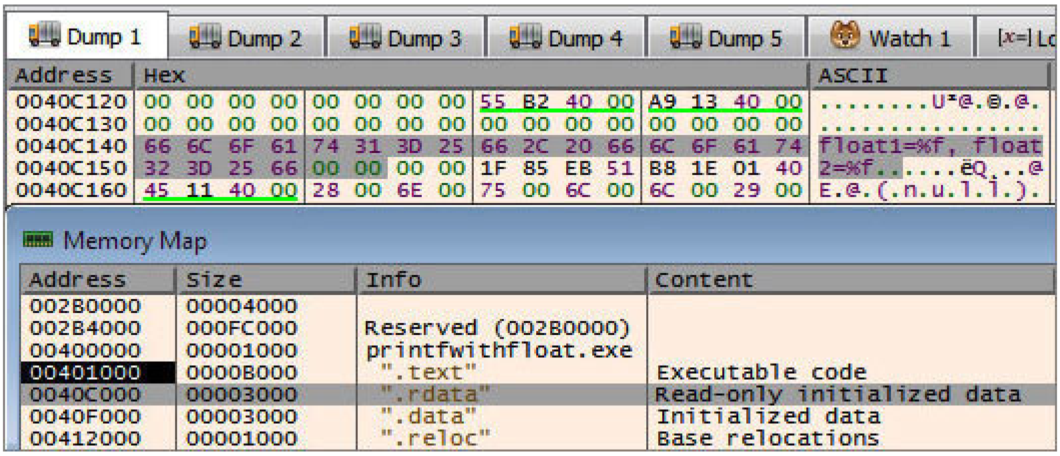
```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
TITLE C:\JitenderN\REBook\printfWithFloat\printfWithFloat\printfWithFloat.cpp
.686P
.XMM
include listing.inc
.model flat
INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES
```

We have already discussed this in the earlier section, so we will move on to the next instruction.

▼Line 12-14

```
CONST SEGMENT
$SG4677 DB 'float1=%f, float2=%f', 00H
CONST ENDS
```

This is the start of **CONST SEGMENT** named by the linker as The compiler is using the **$SG4677** name to handle the string constant. DB is the data type to define byte and the string is terminated with null, The .rdata can be dumped to view the constant string.



```
Address   Hex                                              ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f......ëQ...@
0040C160  45 11 40 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  E.@.(.n.u.l.l.).
```

```
Address   Size      Info                      Content
002B0000  00004000
002B4000  000FC000  Reserved (002B0000)
00400000  00001000  printfwithfloat.exe
00401000  0000B000   ".text"                  Executable code
0040C000  00003000   ".rdata"                 Read-only initialized data
0040F000  00003000   ".data"                  Initialized data
00412000  00001000   ".reloc"                 Base relocations
```

Figure 7.14: *.rdata*

▼Line 15-16

PUBLIC __real@3ff0000000000000

PUBLIC __real@40011eb851eb851f

Floating point numbers can be represented in a binary number format and this binary numbering format is standardized. Real number format, short/single, and long/double real numbers are available in three sizes:

REAL4, 32-bit, short real or single precision

REAL8, 64-bit, long real or double precision

REAL10, 80-bit, temporary real or extended precision

REAL4, REAL8, REAL10 have different formats: The format for REAL4 is:

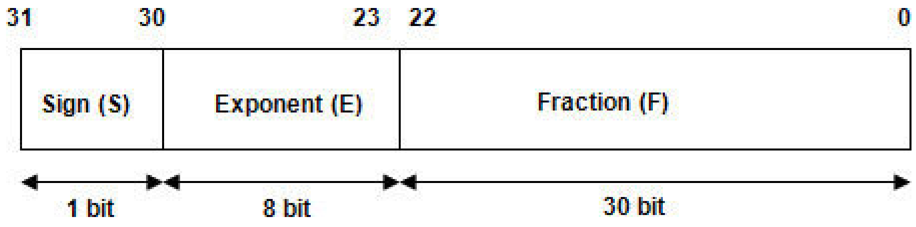


Figure 7.15: *REAL4 format*

Where:

S = sign bit (0=positive, 1=negative)

E = exponent bits

F = fraction bits of the significand

The format for REAL8 is:

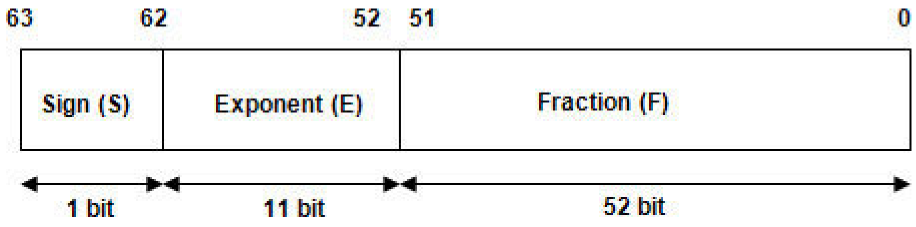


Figure 7.16: *REAL8 format*

The format for REAL10 is:

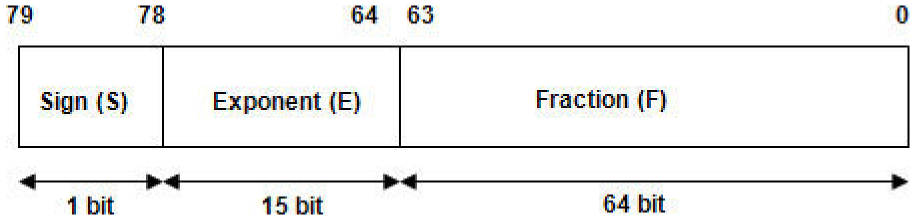
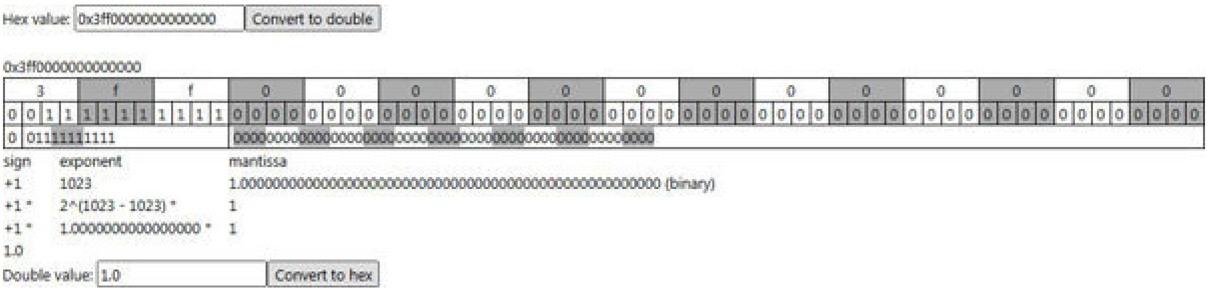


Figure 7.17: REAL10 format

To keep it simple and easy to understand, we will not discuss in detail the individual formats. Keeping the real numbering format (REAL8, which is 64 bit for long real or double precision is used in our code) in mind, we will move to the same instructions:

PUBLIC __real@3ff0000000000000
PUBLIC __real@40011eb851eb851f

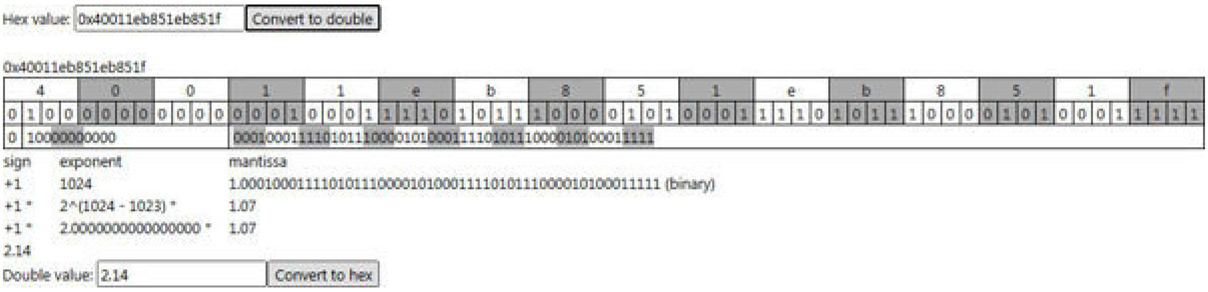
Public is a derivative that makes a variable public and make it available across modules. **real** is representing the real number format followed by the hexadecimal value of our function parameter. Let's convert this hex values into floating number using an online line converter



```
Hex value: 0x3ff0000000000000   Convert to double

0x3ff0000000000000
3       f       f       0       0       0       0       0       0       0       0       0       0       0       0       0
0011    1111    1111    0000    0000    0000    0000    0000    0000    0000    0000    0000    0000    0000    0000    0000
0 01111111111   0000000000000000000000000000000000000000000000000000
sign    exponent              mantissa
+1      1023                  1.0000000000000000000000000000000000000000000000000000 (binary)
+1 *    2^(1023 - 1023) *     1
+1 *    1.0000000000000000 *  1
1.0
Double value: 1.0   Convert to hex
```

Figure 7.18: Hex value into float1

This hexadecimal 3ff0000000000000 is equivalent to our *float1(1.0)* argument in C/C++ code.



```
Hex value: 0x40011eb851eb851f   [Convert to double]

0x40011eb851eb851f
|   4   |   0   |   0   |   1   |   1   |   e   |   b   |   8   |   5   |   1   |   e   |   b   |   8   |   5   |   1   |   f   |
|0|1|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|1|1|1|1|0|1|0|1|1|1|0|0|0|0|1|0|1|0|0|0|1|1|1|1|0|1|0|1|1|1|0|0|0|0|1|0|1|0|0|0|1|1|1|1|1|
|0| 10000000000 | 0001000111101011100001010001111010111000010100011111 |
sign   exponent              mantissa
+1     1024                  1.0001000111101011100001010001111010111000010100011111 (binary)
+1 *   2^(1024 - 1023) *     1.07
+1 *   2.0000000000000000 *  1.07
2.14
Double value: 2.14   [Convert to hex]
```

Figure 7.19: *Hex value into float2*

This hexadecimal 40011eb851eb851f is equivalent to our *float2(2.14)* argument in C/C++ code.

▼Line 17

PUBLIC _main

The **main** function is labeled as a public function, which means it can be accessed by other modules.

▼Line 18-20

EXTRN _printf:PROC
EXTRN __fltused:DWORD
; COMDAT __real@3ff0000000000000

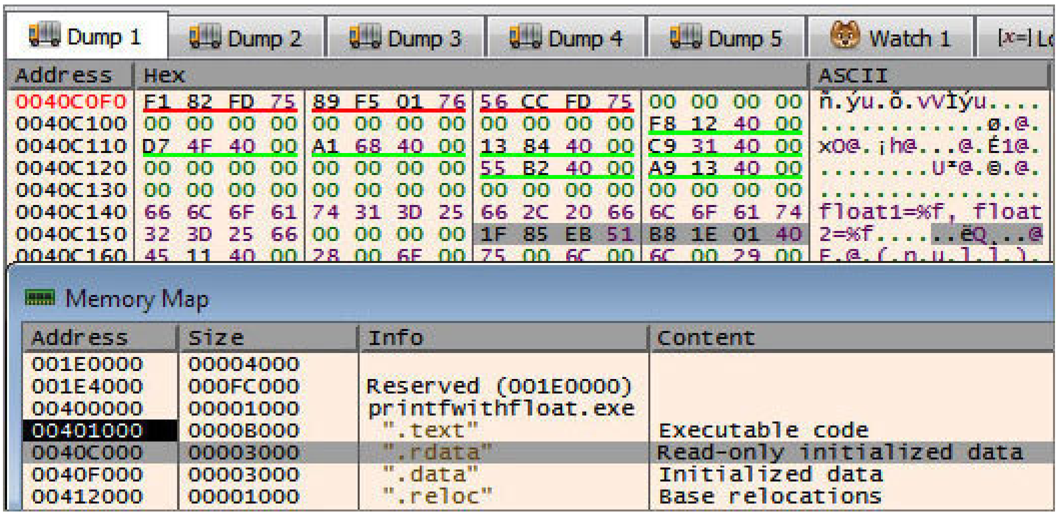
EXTRN derivative declares the external function, which is printf in our case. All functions begin with an underscore.

▼Line 22-29

```
CONST SEGMENT
__real@3ff0000000000000 DQ 03ff0000000000000r ; 1
CONST ENDS
; COMDAT __real@40011eb851eb851f
CONST SEGMENT
__real@40011eb851eb851f DQ 040011eb851eb851fr ; 2.14
; Function compile flags: /Odtp
CONST ENDS
```

This is the start/end of **CONST** Within this segment, our function argument 2.14 represented by 40011eb851eb851f hexadecimal notation is stored in **CONST SEGMENT** or we can say Let's dump **.rdata** to view our float2 argument in C/C++ code using any debugger. As we deal with little-endian, the floating point argument's hexadecimal representation will be stored in a reverse order.

40011eb851eb851f will be stored as **1F 85 EB 51 B8 1E 01 40** in the .rdata segment, as we can see in x32dbg at 0x0040C158 memory location. If you are thinking about other floating variables in .rdata segment, then wait. Things will be clear after a few instructions.



```
Address  Hex                                                ASCII
0040C0F0 F1 82 FD 75 89 F5 01 76 56 CC FD 75 00 00 00 00 ñ.ýu.õ.vVÌýu....
0040C100 00 00 00 00 00 00 00 00 00 00 00 00 F8 12 40 00 ............ø.@.
0040C110 D7 4F 40 00 A1 68 40 00 13 84 40 00 C9 31 40 00 ×O@.¡h@...@.É1@.
0040C120 00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00 ........U²@.©.@.
0040C130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
0040C140 66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74 float1=%f, float
0040C150 32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40 2=%f......ëQ¸..@
0040C160 45 11 40 00 28 00 6E 00 75 00 6C 00 6C 00 29 00 E.@.(.n.u.l.l.).
```

```
Memory Map
Address  Size     Info                 Content
001E0000 00004000
001E4000 000FC000 Reserved (001E0000)
00400000 00001000 printfwithfloat.exe
00401000 0000B000  ".text"             Executable code
0040C000 00003000  ".rdata"            Read-only initialized data
0040F000 00003000  ".data"             Initialized data
00412000 00001000  ".reloc"            Base relocations
```

Figure 7.20: *Floating point argument in .rdata*

▼Line 30-34

_TEXT SEGMENT

_main PROC

; Line 7

push ebp

mov ebp, esp

In the **TEXT** segment, we have the **main** procedure which starts with a function prologue.

▼Line 36

sub esp, 8

Allocating 8 bytes on the stack for the **main** function local variable, which is third argument (2.14) to the printf function.

▼Line 37

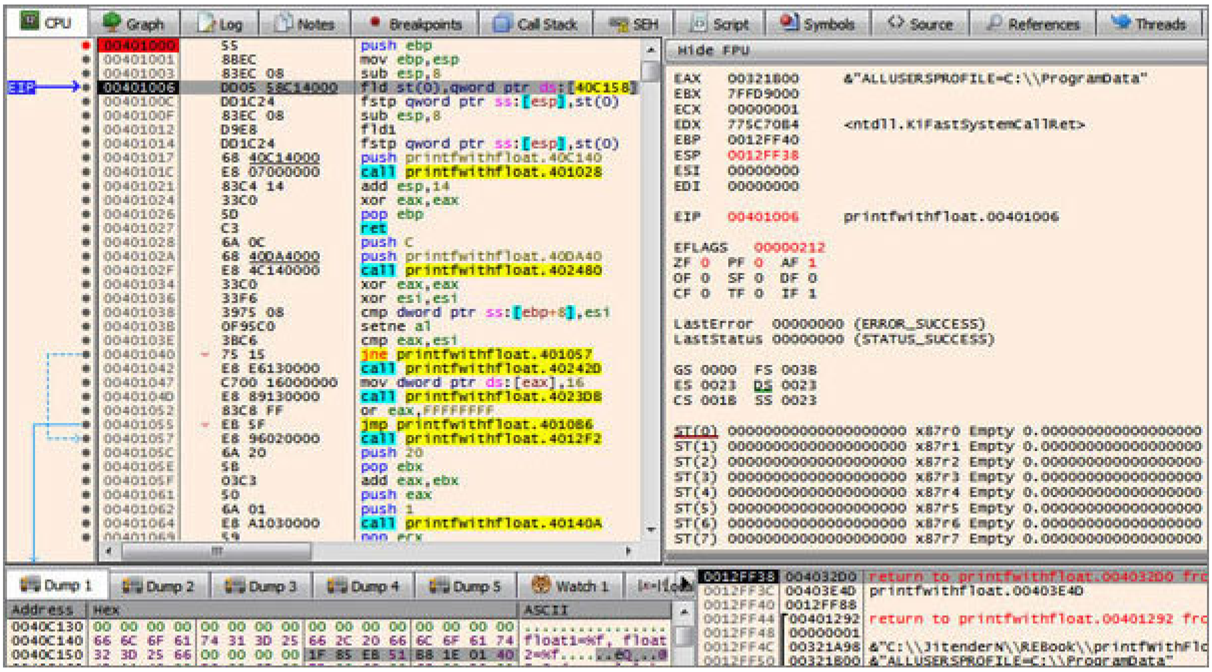
fld QWORD PTR __real@40011eb851eb851f

FLD stands for Floating Point Load. This instruction pushes the floating point value on FPU stack, which is from ST0 to ST7. You can see the debugger output before and after running this instruction as follows:

You can insert breakpoint at the start of To insert breakpoint, scroll to the top of .text segment, you will find the same set of instructions as in our assembly file Once breakpoint is set you can *step into* the instructions one by one.

x32dbg output before instruction execution of this instruction:

fld QWORD PTR __real@40011eb851eb851f



```
     00401000   55             push ebp
     00401001   8BEC           mov ebp,esp
     00401003   83EC 08        sub esp,8
EIP→ 00401006   DD05 58C14000  fld st(0),qword ptr ds:[40C158]
     0040100C   DD1C24         fstp qword ptr ss:[esp],st(0)
     0040100F   83EC 08        sub esp,8
     00401012   D9E8           fld1
     00401014   DD1C24         fstp qword ptr ss:[esp],st(0)
     00401017   68 40C14000    push printfwithfloat.40C140
     0040101C   E8 07000000    call printfwithfloat.401028
     00401021   83C4 14        add esp,14
     00401024   33C0           xor eax,eax
     00401026   5D             pop ebp
     00401027   C3             ret
     00401028   6A 0C          push C
     0040102A   68 40DA4000    push printfwithfloat.40DA40
     0040102F   E8 4C140000    call printfwithfloat.402480
     00401034   33C0           xor eax,eax
     00401036   33F6           xor esi,esi
     00401038   3975 08        cmp dword ptr ss:[ebp+8],esi
     0040103B   0F95C0         setne al
     0040103E   3BC6           cmp eax,esi
     00401040   75 15          jne printfwithfloat.401057
     00401042   E8 E6130000    call printfwithfloat.40242D
     00401047   C700 16000000  mov dword ptr ds:[eax],16
     0040104D   E8 89130000    call printfwithfloat.4023DB
     00401052   83C8 FF        or eax,FFFFFFFF
     00401055   EB 5F          jmp printfwithfloat.401086
     00401057   E8 96020000    call printfwithfloat.4012F2
     0040105C   6A 20          push 20
     0040105E   5B             pop ebx
     0040105F   03C3           add eax,ebx
     00401061   50             push eax
     00401062   6A 01          push 1
     00401064   E8 A1030000    call printfwithfloat.40140A
```

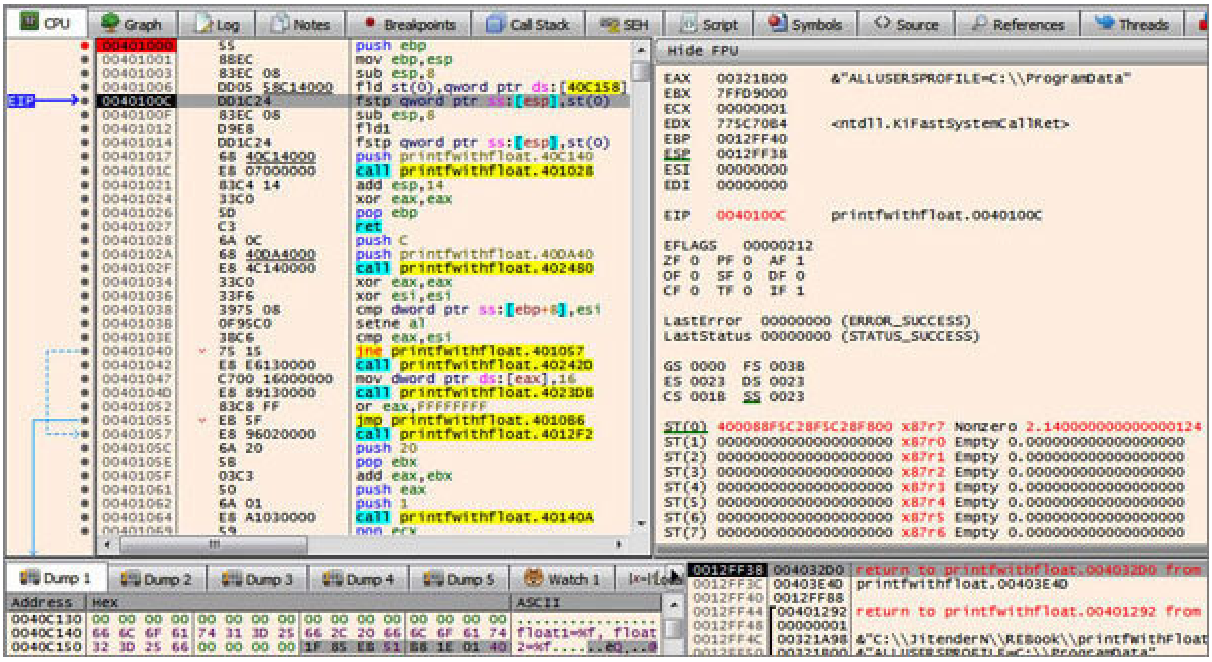
```
EAX     00321800     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX     7FFD9000
ECX     00000001
EDX     775C70B4     <ntdll.KiFastSystemCallRet>
EBP     0012FF40
ESP     0012FF38
ESI     00000000
EDI     00000000

EIP     00401006     printfwithfloat.00401006

EFLAGS    00000212
ZF 0   PF 0   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 0038
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r7 Empty 0.000000000000000000
```

```
Address   Hex                                              ASCII
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f......ëQ...@
```

```
0012FF38  004032D0  return to printfwithfloat.004032D0 fro
0012FF3C  00403E4D  printfwithfloat.00403E4D
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
0012FF4C  00321A98  &"C:\\JitenderN\\REBook\\printfWithFlo
0012FF50  00321800  &"ALLUSERSPROFILE=C:\\ProgramData"
```

Figure 7.21: *ST0 before*

After instruction execution:



```
00401000  55                push ebp
00401001  8BEC              mov ebp,esp
00401003  83EC 08           sub esp,8
00401006  DD05 58C14000     fld st(0),qword ptr ds:[40C158]
0040100C  DD1C24            fstp qword ptr ss:[esp],st(0)
0040100F  83EC 08           sub esp,8
00401012  D9E8              fld1
00401014  DD1C24            fstp qword ptr ss:[esp],st(0)
00401017  68 40C14000       push printfwithfloat.40C140
0040101C  E8 07000000       call printfwithfloat.401028
00401021  83C4 14           add esp,14
00401024  33C0              xor eax,eax
00401026  5D                pop ebp
00401027  C3                ret
00401028  6A 0C             push C
0040102A  68 40DA4000       push printfwithfloat.40DA40
0040102F  E8 4C140000       call printfwithfloat.402480
00401034  33C0              xor eax,eax
00401036  33F6              xor esi,esi
00401038  3975 08           cmp dword ptr ss:[ebp+8],esi
0040103B  0F95C0            setne al
0040103E  3BC6              cmp eax,esi
00401040  75 15             jne printfwithfloat.401057
00401042  E8 E6130000       call printfwithfloat.40242D
00401047  C700 16000000     mov dword ptr ds:[eax],16
0040104D  E8 89130000       call printfwithfloat.4023DB
00401052  83C8 FF           or eax,FFFFFFFF
00401055  EB 5F             jmp printfwithfloat.4010B6
00401057  E8 96020000       call printfwithfloat.4012F2
0040105C  6A 20             push 20
0040105E  5B                pop ebx
0040105F  03C3              add eax,ebx
00401061  50                push eax
00401062  6A 01             push 1
00401064  E8 A1030000       call printfwithfloat.40140A
```

```
EAX    00321B00     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX    7FFD9000
ECX    00000001
EDX    775C70B4     <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    0040100C     printfwithfloat.0040100C

EFLAGS    00000212
ZF 0   PF 0   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 400088F5C28F5C28F800 x87r7 Nonzero 2.1400000000000000124
ST(1) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r6 Empty 0.000000000000000000
```

```
Address   Hex                                              ASCII
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....ëQ...@
```

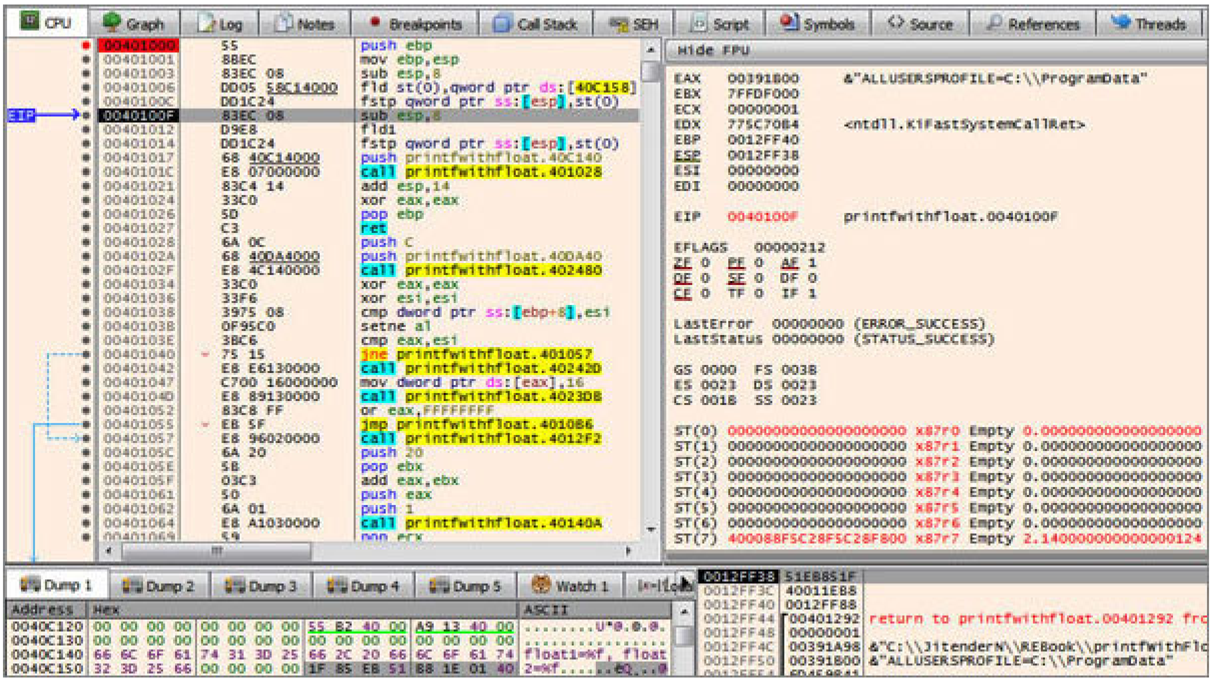
```
0012FF38  004032D0  return to printfwithfloat.004032D0 from
0012FF3C  00403E4D  printfwithfloat.00403E4D
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 from
0012FF48  00000001
0012FF4C  00321A98  &"C:\\JitenderN\\REBook\\printfwithFloat
```

Figure 7.22: *ST0 after*

▼Line 38

fstp QWORD PTR [esp]

FSTP means Floating Point Store and POP. It moves the floating point value from ST0 to the top of the stack PTR [esp] and POP the value from ST0 completely.



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   83EC 08         sub esp,8
00401006   DD05 58C14000   fld st(0),qword ptr ds:[40C158]
0040100C   DD1C24          fstp qword ptr ss:[esp],st(0)
0040100F   83EC 08         sub esp,8
00401012   D9E8            fld1
00401014   DD1C24          fstp qword ptr ss:[esp],st(0)
00401017   68 40C14000     push printfwithfloat.40C140
0040101C   E8 07000000     call printfwithfloat.401028
00401021   83C4 14         add esp,14
00401024   33C0            xor eax,eax
00401026   5D              pop ebp
00401027   C3              ret
00401028   6A 0C           push C
0040102A   68 40DA4000     push printfwithfloat.40DA40
0040102F   E8 4C140000     call printfwithfloat.402480
00401034   33C0            xor eax,eax
00401036   33F6            xor esi,esi
00401038   3975 08         cmp dword ptr ss:[ebp+8],esi
0040103B   0F95C0          setne al
0040103E   3BC6            cmp eax,esi
00401040   75 15           jne printfwithfloat.401057
00401042   E8 E6130000     call printfwithfloat.40242D
00401047   C700 16000000   mov dword ptr ds:[eax],16
0040104D   E8 89130000     call printfwithfloat.4023DB
00401052   83C8 FF         or eax,FFFFFFFF
00401055   EB 5F           jmp printfwithfloat.4010B6
00401057   E8 96020000     call printfwithfloat.4012F2
0040105C   6A 20           push 20
0040105E   5B              pop ebx
0040105F   03C3            add eax,ebx
00401061   50              push eax
00401062   6A 01           push 1
00401064   E8 A1030000     call printfwithfloat.40140A
```

```
EAX    00391800    &"ALLUSERSPROFILE=C:\\ProgramData"
EBX    7FFDF000
ECX    00000001
EDX    775C70B4    <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    0040100F    printfwithfloat.0040100F

EFLAGS    00000212
ZF 0  PF 0  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 400088F5C28F5C28F800 x87r7 Empty 2.140000000000000124
```

```
Address   Hex                                              ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....ëQ¸..@
```

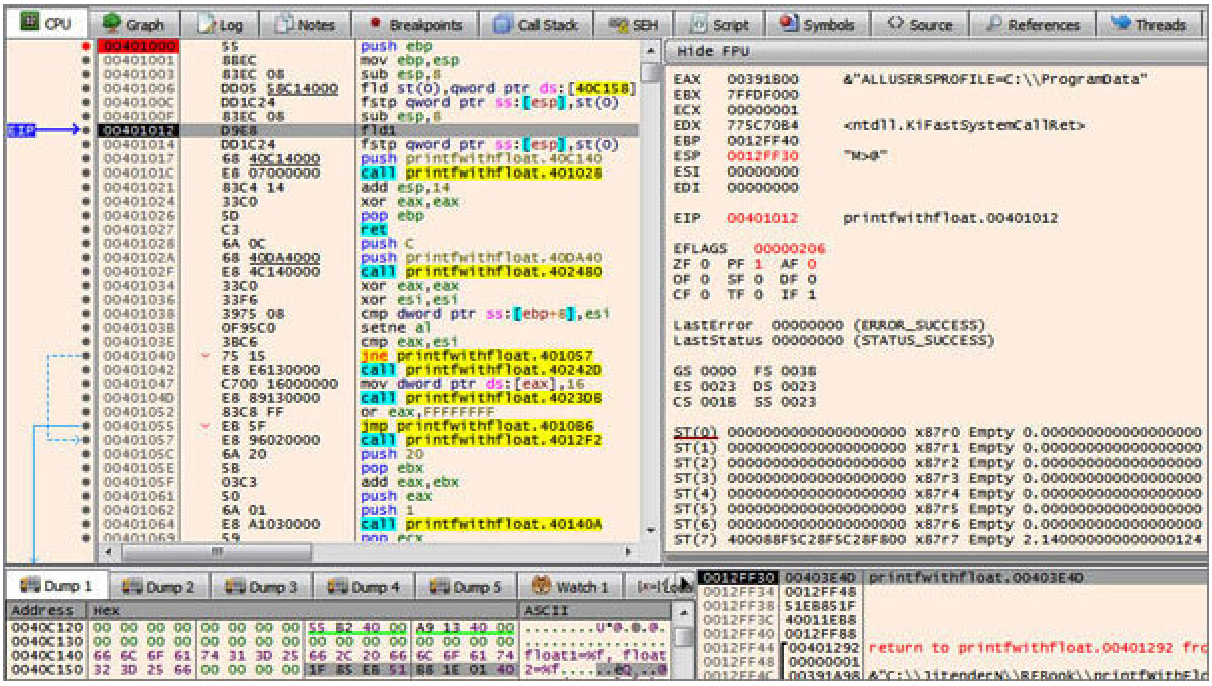
```
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
0012FF4C  00391A98  &"C:\\JitenderN\\REBook\\printfwithFlo
0012FF50  00391800  &"ALLUSERSPROFILE=C:\\ProgramData"
```

***Figure 7.23:** Floating point Store and POP*

▼Line 39

sub esp, 8

Again allocating 8 bytes on the stack for the **main** function local variable, which is the second parameter 1.0 to the printf function. Let's see what it looks like in x32dbg.



```
00401000   55               push ebp
00401001   8BEC             mov ebp,esp
00401003   83EC 08          sub esp,8
00401006   DD05 58C14000    fld st(0),qword ptr ds:[40C158]
0040100C   DD1C24           fstp qword ptr ss:[esp],st(0)
0040100F   83EC 08          sub esp,8
00401012   D9E8             fld1
00401014   DD1C24           fstp qword ptr ss:[esp],st(0)
00401017   68 40C14000      push printfwithfloat.40C140
0040101C   E8 07000000      call printfwithfloat.401028
00401021   83C4 14          add esp,14
00401024   33C0             xor eax,eax
00401026   5D               pop ebp
00401027   C3               ret
00401028   6A 0C            push C
0040102A   68 400A4000      push printfwithfloat.400A40
0040102F   E8 4C140000      call printfwithfloat.402480
00401034   33C0             xor eax,eax
00401036   33F6             xor esi,esi
00401038   3975 08          cmp dword ptr ss:[ebp+8],esi
0040103B   0F95C0           setne al
0040103E   3BC6             cmp eax,esi
00401040   75 15            jne printfwithfloat.401057
00401042   E8 E6130000      call printfwithfloat.40242D
00401047   C700 16000000    mov dword ptr ds:[eax],16
0040104D   E8 89130000      call printfwithfloat.4023DB
00401052   83C8 FF          or eax,FFFFFFFF
00401055   EB 5F            jmp printfwithfloat.401086
00401057   E8 96020000      call printfwithfloat.4012F2
0040105C   6A 20            push 20
0040105E   5B               pop ebx
0040105F   03C3             add eax,ebx
00401061   50               push eax
00401062   6A 01            push 1
00401064   E8 A1030000      call printfwithfloat.40140A
00401069   59               pop ecx
```

```
EAX    00391800     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX    7FFDF000
ECX    00000001
EDX    775C70B4     <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF30     "M>@"
ESI    00000000
EDI    00000000

EIP    00401012     printfwithfloat.00401012

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 400088F5C28F5C28F800 x87r7 Empty 2.140000000000000124
```

```
Address   Hex                                              ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f......ëQ¸..@
```

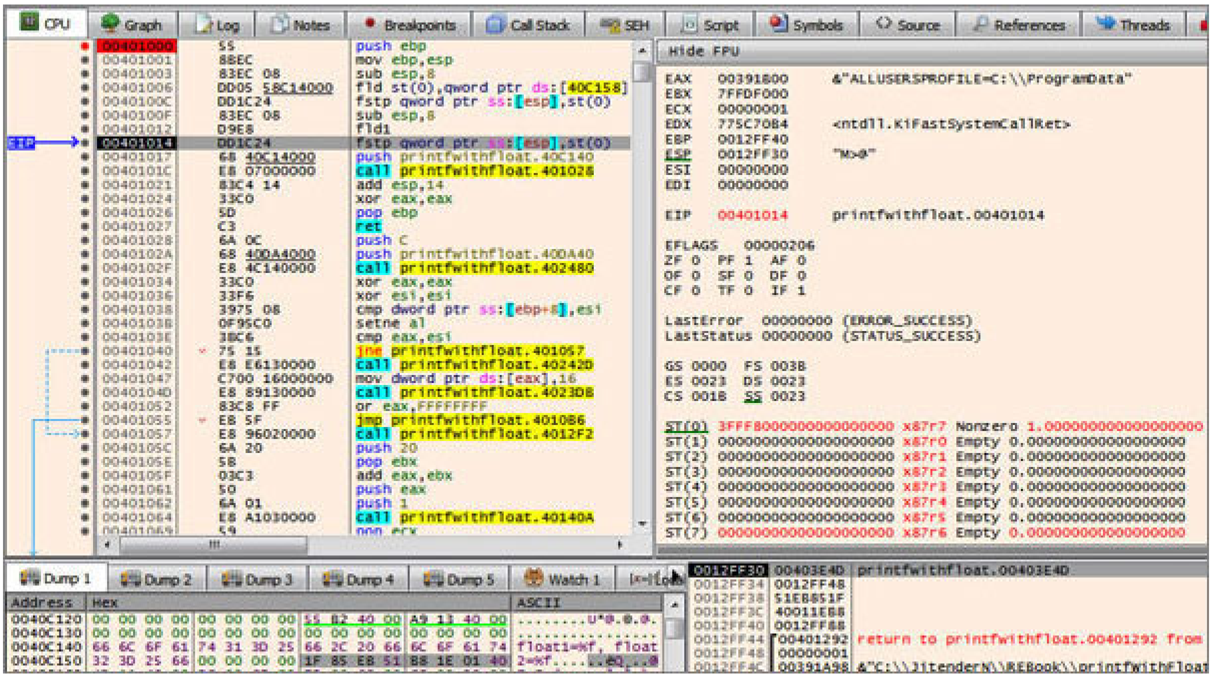
```
0012FF30  00403E4D  printfwithfloat.00403E4D
0012FF34  0012FF48
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
0012FF4C  00391A98  &"C:\\JitenderN\\REBook\\printfwithFlo
```

Figure 7.24: Allocating space on stack

▼Line 40

fld1

This instruction loads the floating point value 1.0 on the FPU stack.



```
00401000  55              push ebp
00401001  8BEC            mov ebp,esp
00401003  83EC 08         sub esp,8
00401006  DD05 58C14000   fld st(0),qword ptr ds:[40C158]
0040100C  DD1C24          fstp qword ptr ss:[esp],st(0)
0040100F  83EC 08         sub esp,8
00401012  D9E8            fld1
00401014  DD1C24          fstp qword ptr ss:[esp],st(0)
00401017  68 40C14000     push printfwithfloat.40C140
0040101C  E8 07000000     call printfwithfloat.401028
00401021  83C4 14         add esp,14
00401024  33C0            xor eax,eax
00401026  5D              pop ebp
00401027  C3              ret
00401028  6A 0C           push C
0040102A  68 400A4000     push printfwithfloat.40DA40
0040102F  E8 4C140000     call printfwithfloat.402480
00401034  33C0            xor eax,eax
00401036  33F6            xor esi,esi
00401038  3975 08         cmp dword ptr ss:[ebp+8],esi
0040103B  0F95C0          setne al
0040103E  3BC6            cmp eax,esi
00401040  75 15           jne printfwithfloat.401057
00401042  E8 E6130000     call printfwithfloat.40242D
00401047  C700 16000000   mov dword ptr ds:[eax],16
0040104D  E8 89130000     call printfwithfloat.4023DB
00401052  83C8 FF         or eax,FFFFFFFF
00401055  EB 5F           jmp printfwithfloat.4010B6
00401057  E8 96020000     call printfwithfloat.4012F2
0040105C  6A 20           push 20
0040105E  5B              pop ebx
0040105F  03C3            add eax,ebx
00401061  50              push eax
00401062  6A 01           push 1
00401064  E8 A1030000     call printfwithfloat.40140A
00401069  59              pop ecx
```

```
EAX   00391800     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX   7FFDF000
ECX   00000001
EDX   775C70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF40
ESP   0012FF30     "M>@"
ESI   00000000
EDI   00000000

EIP   00401014     printfwithfloat.00401014

EFLAGS    00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 3FFF8000000000000000 x87r7 Nonzero 1.000000000000000000
ST(1) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r6 Empty 0.000000000000000000
```

```
Address   Hex                                               ASCII
0040C120  00 00 00 00 00 00 00 00 55 82 40 00 A9 11 40 00  ........U‚@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....…ëQ¸..@
```

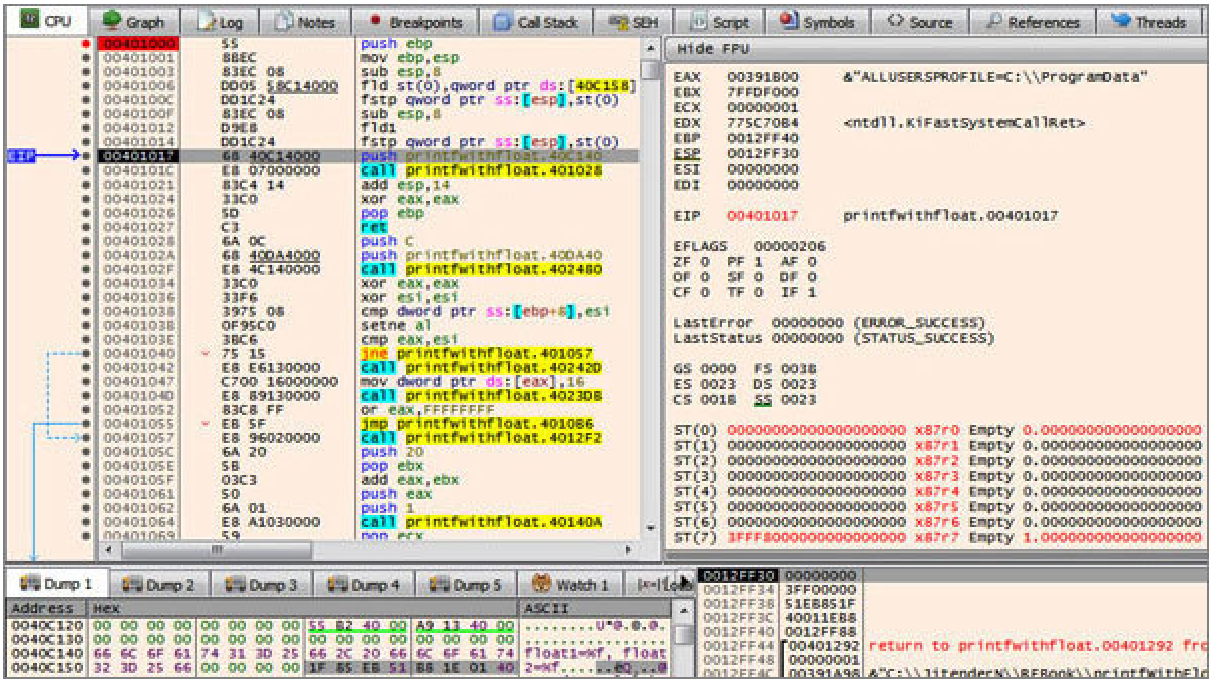
```
0012FF30  00403E4D  printfwithfloat.00403E4D
0012FF34  0012FF48
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 from
0012FF48  00000001
0012FF4C  00391A98  &"C:\\JitenderN\\REBook\\printfwithFloat
```

Figure 7.25: *Loads floating point value 1.0*

▼Line 41

fstp QWORD PTR [esp]

FSTP means Floating Point Store and POP, it moves the floating point value which is 1.0 from ST0 to the top of the stack PTR [esp] and POP the value from ST0 completely.



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   83EC 08         sub esp,8
00401006   DD05 58C14000   fld st(0),qword ptr ds:[40C158]
0040100C   DD1C24          fstp qword ptr ss:[esp],st(0)
0040100F   83EC 08         sub esp,8
00401012   D9E8            fld1
00401014   DD1C24          fstp qword ptr ss:[esp],st(0)
00401017   68 40C14000     push printfwithfloat.40C140
0040101C   E8 07000000     call printfwithfloat.401028
00401021   83C4 14         add esp,14
00401024   33C0            xor eax,eax
00401026   5D              pop ebp
00401027   C3              ret
00401028   6A 0C           push C
0040102A   68 40DA4000     push printfwithfloat.40DA40
0040102F   E8 4C140000     call printfwithfloat.402480
00401034   33C0            xor eax,eax
00401036   33F6            xor esi,esi
00401038   3975 08         cmp dword ptr ss:[ebp+8],esi
0040103B   0F95C0          setne al
0040103E   3BC6            cmp eax,esi
00401040   75 15           jne printfwithfloat.401057
00401042   E8 E6130000     call printfwithfloat.40242D
00401047   C700 16000000   mov dword ptr ds:[eax],16
0040104D   E8 89130000     call printfwithfloat.4023DB
00401052   83C8 FF         or eax,FFFFFFFF
00401055   EB 5F           jmp printfwithfloat.401086
00401057   E8 96020000     call printfwithfloat.4012F2
0040105C   6A 20           push 20
0040105E   5B              pop ebx
0040105F   03C3            add eax,ebx
00401061   50              push eax
00401062   6A 01           push 1
00401064   E8 A1030000     call printfwithfloat.40140A
00401069   59              pop ecx
```

```
EAX    00391800   &"ALLUSERSPROFILE=C:\\ProgramData"
EBX    7FFDF000
ECX    00000001
EDX    775C7084   <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF30
ESI    00000000
EDI    00000000

EIP    00401017   printfwithfloat.00401017

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 0000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 0000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 0000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 0000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 0000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 0000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 3FFF800000000000000 x87r7 Empty 1.000000000000000000
```

```
Address   Hex                                                ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....ëQ¸..@
```

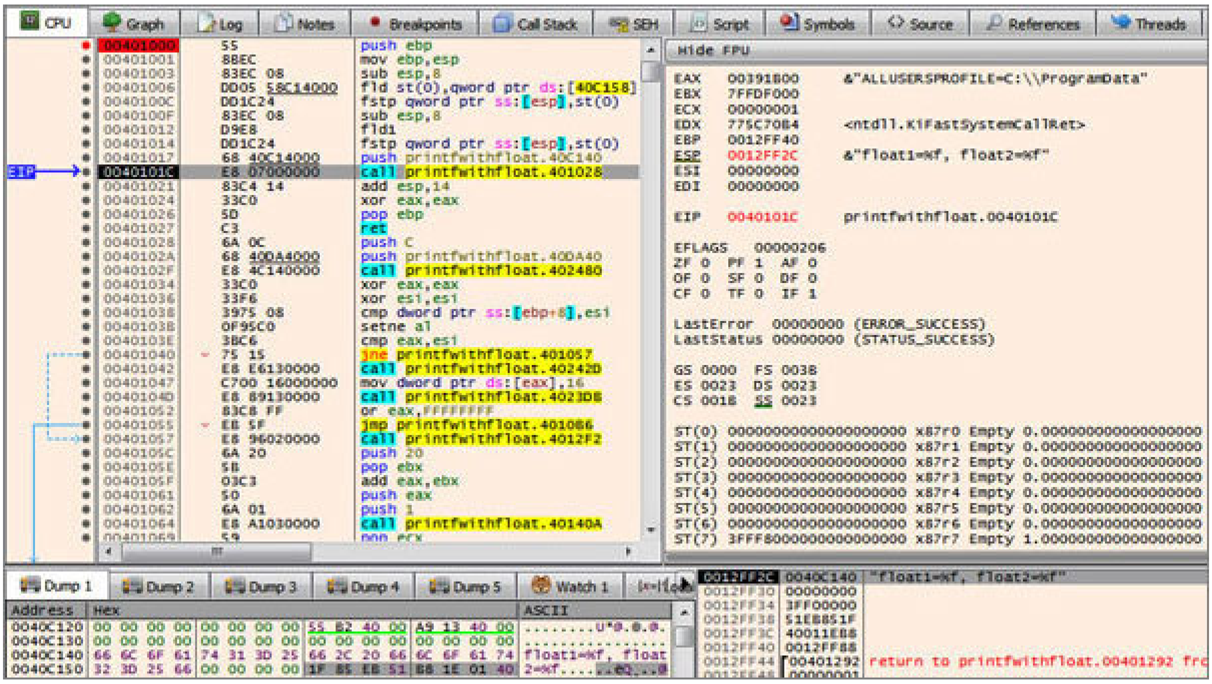
```
0012FF30  00000000
0012FF34  3FF00000
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
0012FF4C  00391A98  &"C:\\JitenderN\\REBook\\printfwithFlo
```

Figure 7.26: *Floating Point Store and POP*

▼Line 42

push OFFSET $SG4677

Now before calling the printf function, we have to pass the remaining string constant to the stack. This PUSH instruction is pushing the string constant $SG4677 onto the stack.



```
00401000  55             push ebp
00401001  8BEC           mov ebp,esp
00401003  83EC 08        sub esp,8
00401006  DD05 58C14000  fld st(0),qword ptr ds:[40C158]
0040100C  DD1C24         fstp qword ptr ss:[esp],st(0)
0040100F  83EC 08        sub esp,8
00401012  D9E8           fld1
00401014  DD1C24         fstp qword ptr ss:[esp],st(0)
00401017  68 40C14000    push printfwithfloat.40C140
0040101C  E8 07000000    call printfwithfloat.401028
00401021  83C4 14        add esp,14
00401024  33C0           xor eax,eax
00401026  5D             pop ebp
00401027  C3             ret
00401028  6A 0C          push C
0040102A  68 40DA4000    push printfwithfloat.40DA40
0040102F  E8 4C140000    call printfwithfloat.402480
00401034  33C0           xor eax,eax
00401036  33F6           xor esi,esi
00401038  3975 08        cmp dword ptr ss:[ebp+8],esi
0040103B  0F95C0         setne al
0040103E  3BC6           cmp eax,esi
00401040  75 15          jne printfwithfloat.401057
00401042  E8 E6130000    call printfwithfloat.40242D
00401047  C700 16000000  mov dword ptr ds:[eax],16
0040104D  E8 89130000    call printfwithfloat.4023DB
00401052  83C8 FF        or eax,FFFFFFFF
00401055  EB 5F          jmp printfwithfloat.401086
00401057  E8 96020000    call printfwithfloat.4012F2
0040105C  6A 20          push 20
0040105E  5B             pop ebx
0040105F  03C3           add eax,ebx
00401061  50             push eax
00401062  6A 01          push 1
00401064  E8 A1030000    call printfwithfloat.40140A
00401069  59             pop ecx
```

```
EAX     00191800     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX     7FFDF000
ECX     00000001
EDX     775C70B4     <ntdll.KiFastSystemCallRet>
EBP     0012FF40
ESP     0012FF2C     &"float1=%f, float2=%f"
ESI     00000000
EDI     00000000

EIP     0040101C     printfwithfloat.0040101C

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 3FFF8000000000000000 x87r7 Empty 1.000000000000000000
```

```
Address   Hex                                              ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U².@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....ëQ¸..@
```

```
0012FF2C  0040C140  "float1=%f, float2=%f"
0012FF30  00000000
0012FF34  3FF00000
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
```

Figure 7.27: *Push string*

▼Line 43

call _printf

All the parameters to the printf function are pushed onto the stack before the **CALL** instruction to the printf function. After this instruction execution, both the variables will be printed on the console.

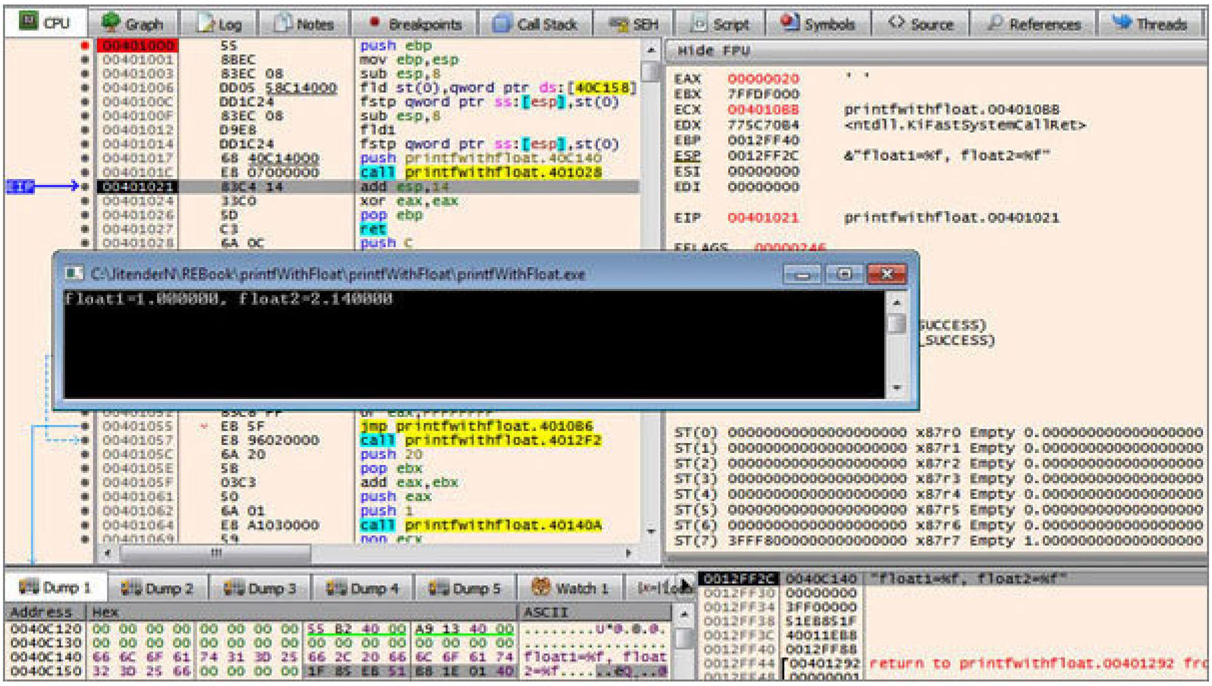
▼Line 44

add esp, 20   ; 00000014H

As we are following the CDECL calling convention, it's the caller who cleans the stack. On returning from the printf function, the main cleans the stack by moving the stack point back by 20 bytes (we used 4 bytes for pushing the string constant, 8 bytes for pushing the 1.0 floating hex value and another 8 bytes were used for pushing the 2.14 floating hex value).

Before this instruction, the x32dbg screen looks as follows:

To move directly to *add* instruction, you can use *step over* to execute *print* function and stop at *add* instruction. Or you can insert breakpoint at the *add* instruction to stop execution at *add* instruction.



```
00401000  55             push ebp
00401001  8BEC           mov ebp,esp
00401003  83EC 08        sub esp,8
00401006  DD05 58C14000  fld st(0),qword ptr ds:[40C158]
0040100C  DD1C24         fstp qword ptr ss:[esp],st(0)
0040100F  83EC 08        sub esp,8
00401012  D9E8           fld1
00401014  DD1C24         fstp qword ptr ss:[esp],st(0)
00401017  68 40C14000    push printfwithfloat.40C140
0040101C  E8 07000000    call printfwithfloat.401028
00401021  83C4 14        add esp,14
00401024  33C0           xor eax,eax
00401026  5D             pop ebp
00401027  C3             ret
00401028  6A 0C          push C
```

```
EAX  00000020     ' '
EBX  7FFDF000
ECX  00401088     printfwithfloat.00401088
EDX  775C7084     <ntdll.KiFastSystemCallRet>
EBP  0012FF40
ESP  0012FF2C     &"float1=%f, float2=%f"
ESI  00000000
EDI  00000000

EIP  00401021     printfwithfloat.00401021
```

C:\JitenderN\REBook\printfWithFloat\printfWithFloat\printfWithFloat.exe

```
Float1=1.000000, Float2=2.140000
```

```
00401055  EB 5F          jmp printfwithfloat.401086
00401057  E8 96020000    call printfwithfloat.4012F2
0040105C  6A 20          push 20
0040105E  5B             pop ebx
0040105F  03C3           add eax,ebx
00401061  50             push eax
00401062  6A 01          push 1
00401064  E8 A1030000    call printfwithfloat.40140A
```

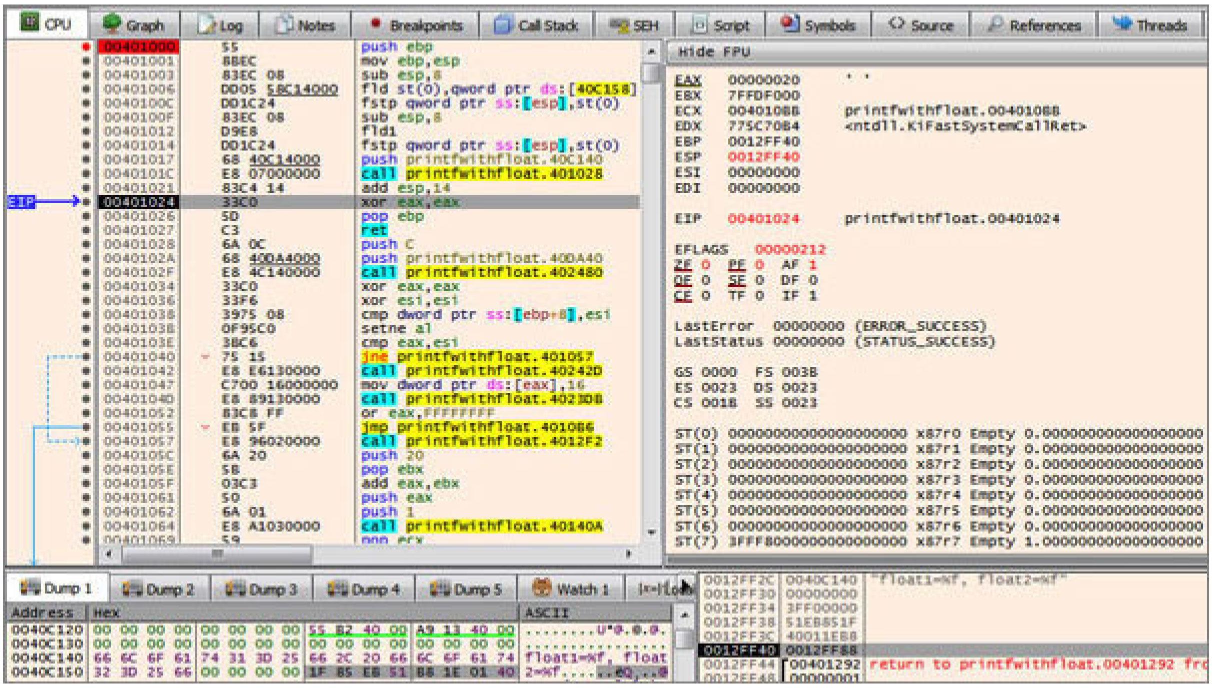
```
ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 3FFF8000000000000000 x87r7 Empty 1.000000000000000000
```

```
Address   Hex                                              ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f.....ëQ¸..@
```

```
0012FF2C  0040C140  "float1=%f, float2=%f"
0012FF30  00000000
0012FF34  3FF00000
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fr
```

Figure 7.28: After call printf

After this instruction:



```
00401000   55               push ebp
00401001   8BEC             mov ebp,esp
00401003   83EC 08          sub esp,8
00401006   DD05 58C14000    fld st(0),qword ptr ds:[40C158]
0040100C   DD1C24           fstp qword ptr ss:[esp],st(0)
0040100F   83EC 08          sub esp,8
00401012   D9E8             fld1
00401014   DD1C24           fstp qword ptr ss:[esp],st(0)
00401017   68 40C14000      push printfwithfloat.40C140
0040101C   E8 07000000      call printfwithfloat.401028
00401021   83C4 14          add esp,14
00401024   33C0             xor eax,eax
00401026   5D               pop ebp
00401027   C3               ret
00401028   6A 0C            push C
0040102A   68 40DA4000      push printfwithfloat.40DA40
0040102F   E8 4C140000      call printfwithfloat.402480
00401034   33C0             xor eax,eax
00401036   33F6             xor esi,esi
00401038   3975 08          cmp dword ptr ss:[ebp+8],esi
0040103B   0F95C0           setne al
0040103E   3BC6             cmp eax,esi
00401040   75 15            jne printfwithfloat.401057
00401042   E8 E6130000      call printfwithfloat.40242D
00401047   C700 16000000    mov dword ptr ds:[eax],16
0040104D   E8 89130000      call printfwithfloat.4023DB
00401052   83C8 FF          or eax,FFFFFFFF
00401055   EB 5F            jmp printfwithfloat.4010B6
00401057   E8 96020000      call printfwithfloat.4012F2
0040105C   6A 20            push 20
0040105E   5B               pop ebx
0040105F   03C3             add eax,ebx
00401061   50               push eax
00401062   6A 01            push 1
00401064   E8 A1030000      call printfwithfloat.40140A
00401069   59               pop ecx
```

```
EAX   00000020     ' '
EBX   7FFDF000
ECX   004010B8     printfwithfloat.004010B8
EDX   775C70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF40
ESP   0012FF40
ESI   00000000
EDI   00000000

EIP   00401024     printfwithfloat.00401024

EFLAGS    00000212
ZF 0   PF 0   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 3FFF8000000000000000 x87r7 Empty 1.000000000000000000
```

```
Address   Hex                                                ASCII
0040C120  00 00 00 00 00 00 00 00 55 B2 40 00 A9 13 40 00  ........U²@.©.@.
0040C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040C140  66 6C 6F 61 74 31 3D 25 66 2C 20 66 6C 6F 61 74  float1=%f, float
0040C150  32 3D 25 66 00 00 00 00 1F 85 EB 51 B8 1E 01 40  2=%f....…ëQ¸..@
```

```
0012FF2C  0040C140  "float1=%f, float2=%f"
0012FF30  00000000
0012FF34  3FF00000
0012FF38  51EB851F
0012FF3C  40011EB8
0012FF40  0012FF88
0012FF44  00401292  return to printfwithfloat.00401292 fro
0012FF48  00000001
```

Figure 7.29: Stack cleaned

▼Line 45-52

```
; Line 9

xor eax, eax
; Line 10
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

The remaining instructions are making EAX zero as **main** is returning 0 in the C/C++ code. In the last it is calling function epilogue to end main, TEXT segment, and code.

Function printf Printing Float with Optimization

Compile the code with optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

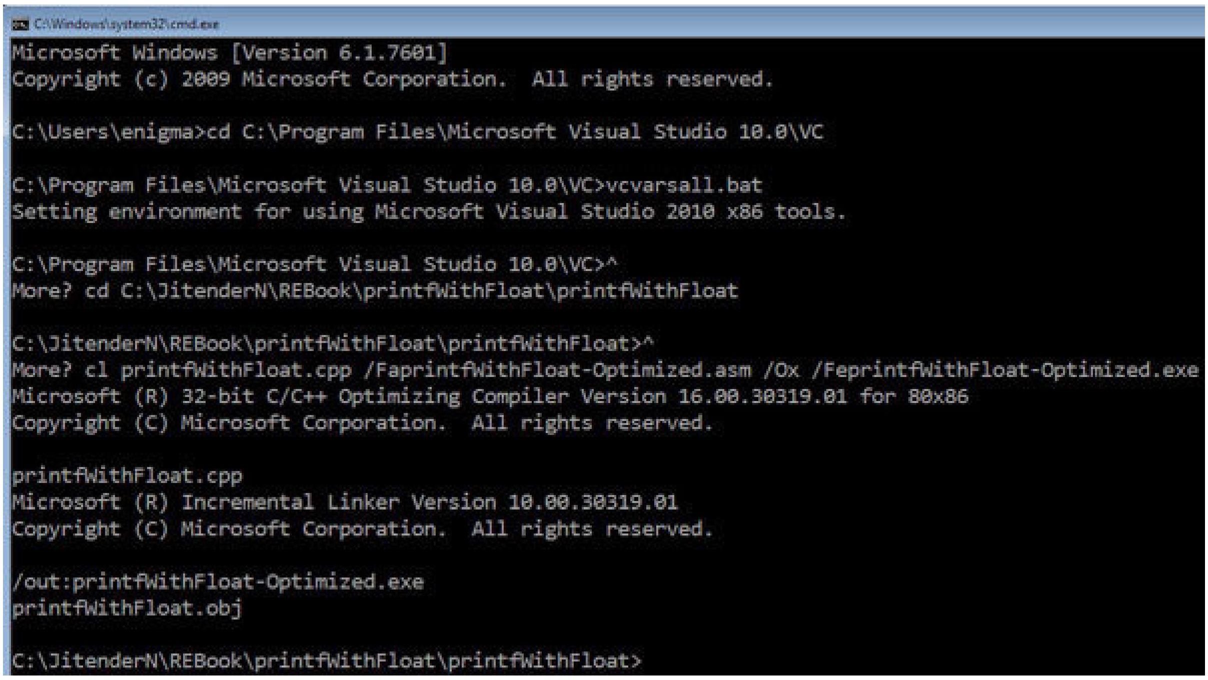
/Ox: Enable maximum optimization

/Fa: Name of the output assembly listing file

/Fe: Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithFloat\printfWithFloat

C:\JitenderN\REBook\printfWithFloat\printfWithFloat>^
More? cl printfWithFloat.cpp /FaprintfWithFloat-Optimized.asm /Ox /FeprintfWithFloat-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithFloat.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithFloat-Optimized.exe
printfWithFloat.obj

C:\JitenderN\REBook\printfWithFloat\printfWithFloat>
```

Figure 7.30: *Function printf printing Float with Optimization*

The generated assembly code will be as follows:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\printfWithFloat\printfWithFloat\printfWithFloat.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4677 DB 'float1=%f, float2=%f', 00H
14. CONST ENDS
15. PUBLIC __real@3ff0000000000000
16. PUBLIC __real@40011eb851eb851f
17. PUBLIC _main
18. EXTRN _printf:PROC
19. EXTRN __fltused:DWORD
20. ; COMDAT __real@3ff0000000000000
21. ; File c:\jitendern\rebook\printfwithfloat\printfwithfloat\printfwithfloat.cpp
22. CONST SEGMENT
23. __real@3ff0000000000000 DQ 03ff0000000000000r ; 1
24. CONST ENDS
25. ; COMDAT __real@40011eb851eb851f
26. CONST SEGMENT
27. __real@40011eb851eb851f DQ 040011eb851eb851fr ; 2.14
28. ; Function compile flags: /Ogtpy
29. CONST ENDS
30. _TEXT SEGMENT
31. _main PROC
32. ; Line 8
33.  fld QWORD PTR __real@40011eb851eb851f
34.  sub esp, 16      ; 00000010H
35.  fstp QWORD PTR [esp+8]
36.  fld1
37.  fstp QWORD PTR [esp]
38.  push OFFSET $SG4677
39.  call _printf
40.  add esp, 20      ; 00000014H
41. ; Line 9
42.  xor eax, eax
43. ; Line 10
44.  ret 0
45. _main ENDP
46. _TEXT ENDS
47. END
```

Figure 7.31: *printfWithFloat-Optimized.asm*

All the instructions in the listing are the same except line 33. As the code is optimized, the function prologue and epilogue are removed, which we will discuss in this section.

▼Line 33

```
fld QWORD PTR __real@40011eb851eb851f
```

It is the same as before. This instruction pushes the floating point value 2.14 on the FPU stack, ST0.

▼Line 34

```
sub esp, 16       ; 00000010H
```

As we saw in the non-optimized code, **SUB** is called twice to allocate memory for the **main** function local variable. But in optimization, both instructions are combined to subtract 16 bytes from ESP to allocate 8+8 bytes for floating variables 1.0 and 2.14.

▼Line 35

```
fstp QWORD PTR [esp+8]
```

It moves the floating point value, which is 2.14 from ST0 to the top of the stack PTR [esp+8] and POP the value from ST0 completely.

▼Line 36

```
fld1
```

It is the same as earlier. This instruction loads the floating point value 1.0 on the FPU stack.

▼Line 37

```
fstp QWORD PTR [esp]
```

It moves the floating point value, which is 1.0 from ST0 to the top of the stack PTR [esp] and POP the value from ST0 completely.

▼Line 38-39

```
push OFFSET $SG4677
call _printf
```

Pushes the string constant offset on the stack to call the **printf** function.

▼Line 40-47

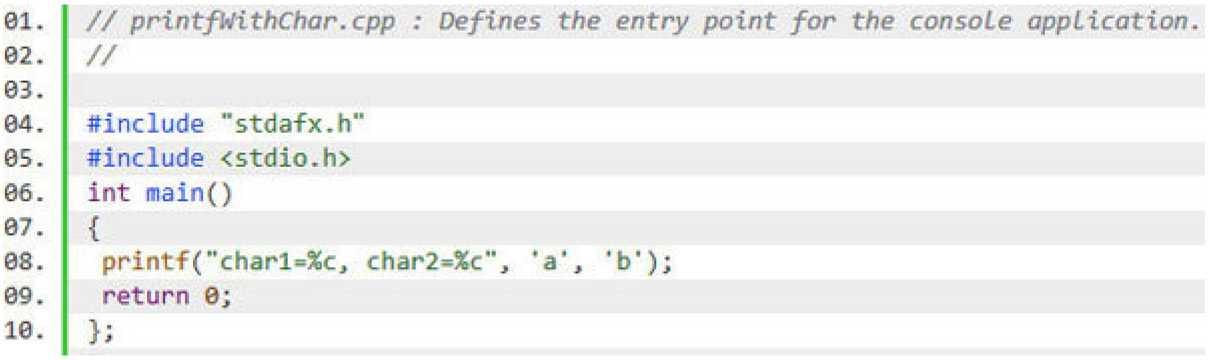
```
add esp, 20        ; 00000014H

; Line 9
xor eax, eax
; Line 10
ret 0
_main ENDP
```

```
_TEXT  ENDS
END
```

The caller cleans the stack as per the CDECL calling convention. The remaining instructions return 0 by XORing the EAX register, and end the main procedure, text segment, and code in line 45-47.

Function printf with char

In this simple C/C++ code, we are printing 2 char on the console. We are using the printf function to print the char.



```
// printfWithChar.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
int main()
{
 printf("char1=%c, char2=%c", 'a', 'b');
 return 0;
};
```

Figure 7.32: *printfWithChar.cpp*

Function printf Printing Char without Optimization

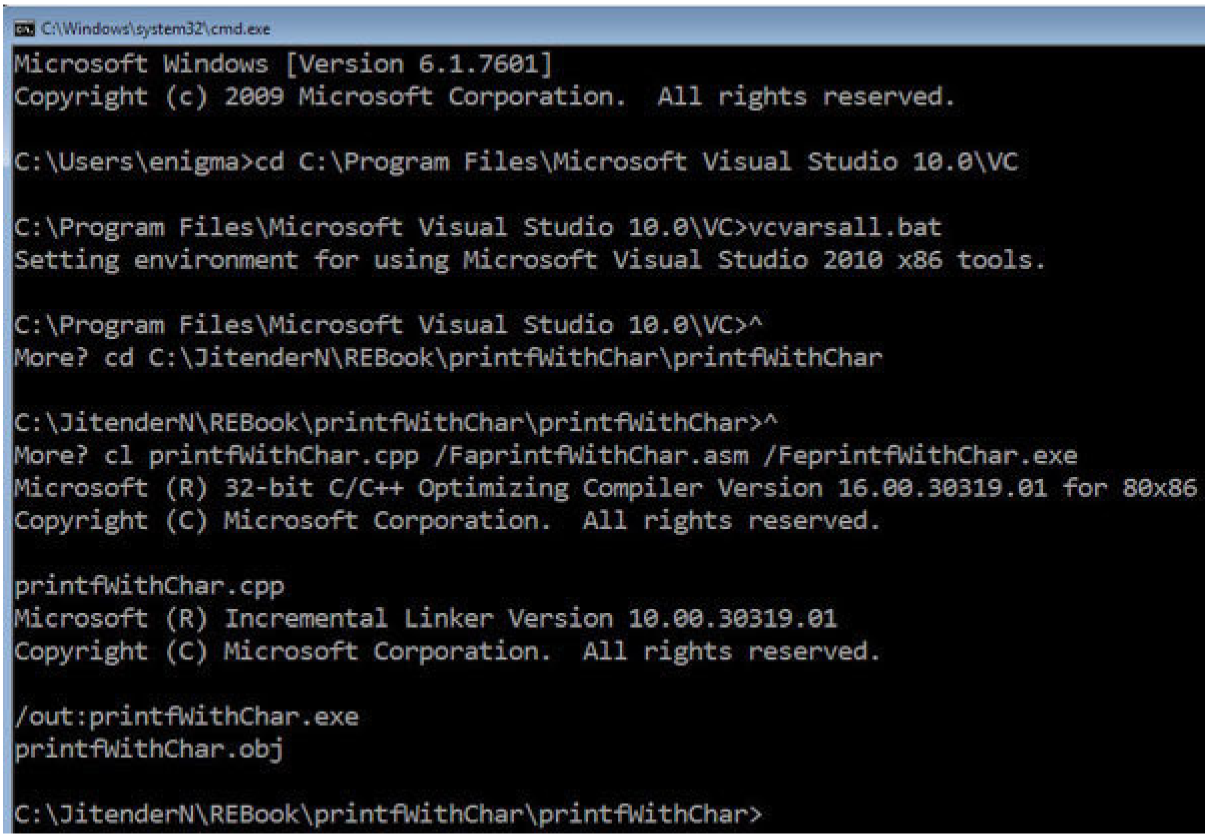
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithChar\printfWithChar

C:\JitenderN\REBook\printfWithChar\printfWithChar>^
More? cl printfWithChar.cpp /FaprintfWithChar.asm /FeprintfWithChar.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithChar.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithChar.exe
printfWithChar.obj

C:\JitenderN\REBook\printfWithChar\printfWithChar>
```

Figure 7.33: *Function printf printing Char without Optimization*

The generated assembly code will be as follows:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\printfWithChar\printfWithChar\printfWithChar.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4677 DB 'char1=%c, char2=%c', 00H
14. CONST ENDS
15. PUBLIC _main
16. EXTRN _printf:PROC
17. ; Function compile flags: /Odtp
18. _TEXT SEGMENT
19. _main PROC
20. ; File c:\jitendern\rebook\printfwithchar\printfwithchar\printfwithchar.cpp
21. ; Line 7
22.  push ebp
23.  mov ebp, esp
24. ; Line 8
25.  push 98      ; 00000062H
26.  push 97      ; 00000061H
27.  push OFFSET $SG4677
28.  call _printf
29.  add esp, 12      ; 0000000cH
30. ; Line 9
31.  xor eax, eax
32. ; Line 10
33.  pop ebp
34.  ret 0
35. _main ENDP
36. _TEXT ENDS
37. END
```

Figure 7.34: *printfWithChar.asm*

Most of the part of the assembly listing has already been discussed in the preceding section. Let's move on to line 25.

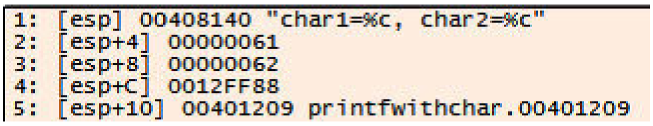
▼Line 25-28

push 98 ; 00000062H
push 97 ; 00000061H
push OFFSET $SG4677
call _printf

Before calling the printf function, three parameters are pushed onto the stack. The first **PUSH** instruction is **PUSH** where 98 is the ASCII value of char ‘b’ and the hex equivalent is 62H. Refer to the ASCII table in the appendix for a complete ASCII listing.

The second **PUSH** instruction is **PUSH** where 97 is the ASCII value of ‘a’ and the hex equivalent is 61H.

And the third parameter to printf is the string constant defined in the **CONST** This is how the stack looks in **x32dbg** before the **CALL** instruction. You can check the stack state by inserting breakpoint at the **CALL** to the printf function.



```
1: [esp] 00408140 "char1=%c, char2=%c"
2: [esp+4] 00000061
3: [esp+8] 00000062
4: [esp+C] 0012FF88
5: [esp+10] 00401209 printfwithchar.00401209
```

Figure 7.35: *Stack state*

1: [esp] 00408140 Memory location of $SG4677, "char1=%c, char2=%c"
2: [esp+4] 00000061 Hex value of char ‘a’ is PUSHed
3: [esp+8] 00000062 Hex value of char ‘b’ is PUSHed
4: [esp+C] 0012FF88 [EBP]
5: [esp+10] 00401209 return to printfwithchar.00401209 from printfwithchar.00401000

After returning from the caller cleans the stack with the **ADD ESP, 12** instruction as we discussed in the CDECL calling convention.

Further, the code is ended with 0 in EAX, which is achieved by XOR'ing EAX. As we discussed earlier, that function return value is stored in EAX.

Function printf printing Char with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

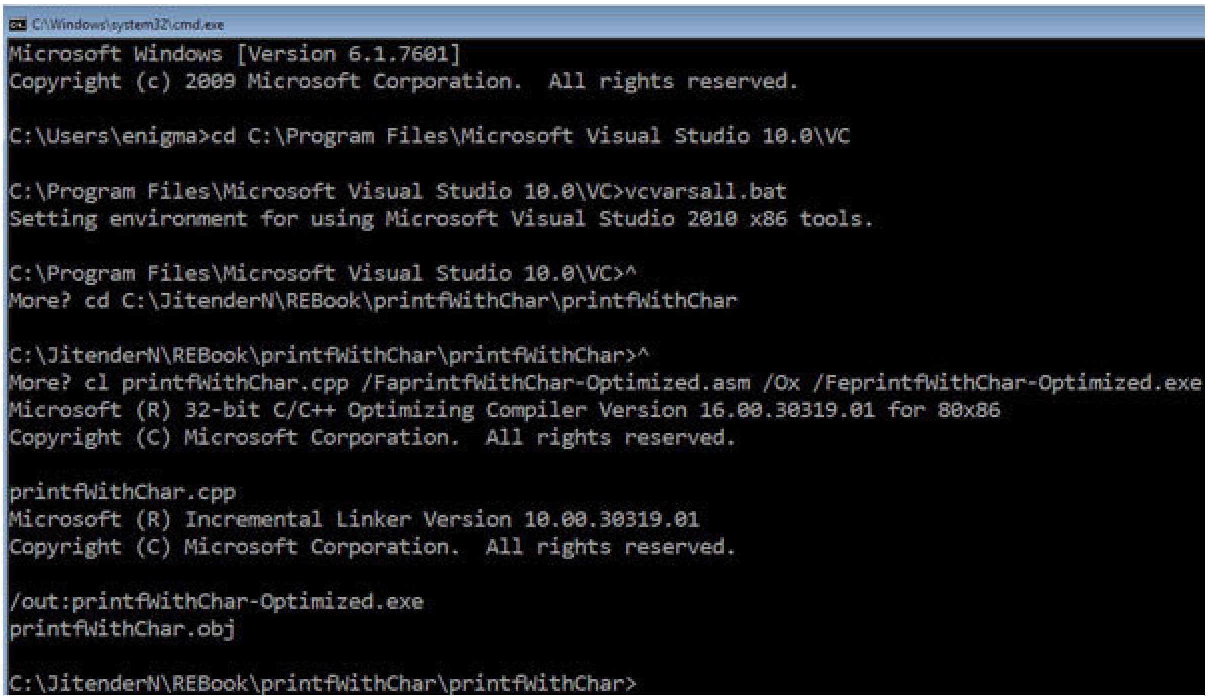
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\printfWithChar\printfWithChar

C:\JitenderN\REBook\printfWithChar\printfWithChar>^
More? cl printfWithChar.cpp /FaprintfWithChar-Optimized.asm /Ox /FeprintfWithChar-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

printfWithChar.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:printfWithChar-Optimized.exe
printfWithChar.obj

C:\JitenderN\REBook\printfWithChar\printfWithChar>
```

Figure 7.36: *Function printf printing Char with Optimization*

The generated assembly code will be as follows:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\printfWithChar\printfWithChar\printfWithChar.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4677 DB 'char1=%c, char2=%c', 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_main PROC
; File c:\jitendern\rebook\printfwithchar\printfwithchar\printfwithchar.cpp
; Line 8
 push 98      ; 00000062H
 push 97      ; 00000061H
 push OFFSET $SG4677
 call _printf
 add esp, 12      ; 0000000cH
; Line 9
 xor eax, eax
; Line 10
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 7.37: *printfWithChar-Optimized.asm*

Everything in the optimized code is the same except that the function prologue and epilogue are eliminated in the optimized code.

Conclusion

In this chapter, we learned to reverse engineer programs or applications with the printf function. We also spoke about the code optimization concept in programs with the printf function. We also discussed how integer, float, and char variables are stored in memory. The floating point variable takes a different approach in working as compared to integer or char. In the next chapter, we will talk about pointers and how they are handled in reverse engineering.

CHAPTER 8

Reverse Engineering Pattern of Pointer Program

Most of us find it difficult to understand pointers, but it is the one of the most interesting subjects in programming. In our real life, have you ever imagined that pointers are everywhere? When we watch the television with a cable connection, we have so many channels to watch. Each channel is associated with a number we often call the channel number. When this channel number is pressed on the remote of our cable modem, the respective channel broadcasting starts on the television. In the context of pointers, this channel number is the pointer to the channel. This is the number where the channel is stored and played when pressed on the remote control.

So now, to understand the concept of pointers, we will walk through its concept in programming. In C/C++, there are various types of variables to hold the different types of values. We have the Integer variables that store the Integer value, the Floating variables to store real numbers, Char to store characters, and others. Similarly, a pointer is a variable that stores the memory address of other variables. In this chapter, we will understand the pattern of pointers in assembly code.

Structure

In this chapter, we will cover the following topics:

Pointers

Pointers without Optimization

Pointers with Optimization

Objective

After studying this chapter, we should be able to understand how pointers are used in programming. We will study the basics of pointers from integer to float and char. This will help you understand the way pointers are handled with respect to memory allocation. After understanding pointers, we will write a simple C/C++ program to generate optimized and non-optimized ASM code and check pointer assembly pattern with and without optimization.

Pointers

To understand pointers, let's take a simple declaration:

int iNumber = 3;

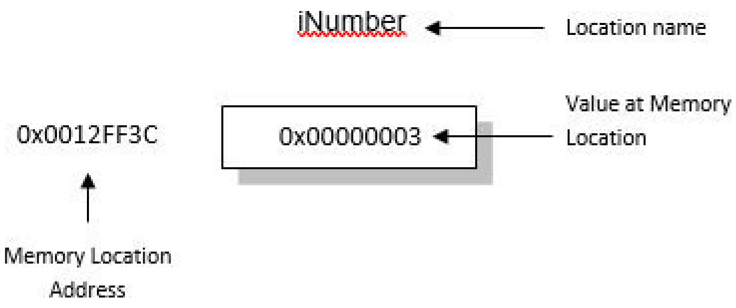
Here, we are defining an integer variable of the name, We are using the Hungarian Notation as the variable naming convention. This declaration is used by the compiler to:

Reverse space for an integer variable in memory

Associate the name with the memory location

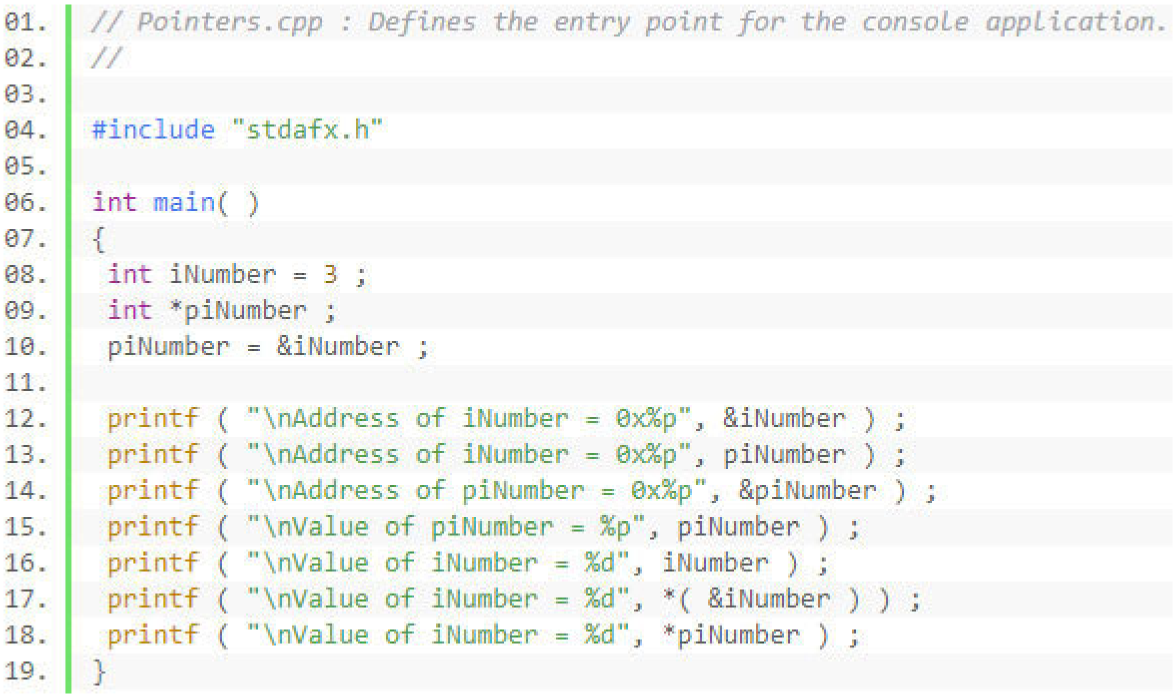
Store 3 on the reserved memory location

You can imagine this as:



0x0012FF3C is the memory location address which is holding **0x00000003** as a value.

So we saw how a variable is stored in memory. Now, we will take a C/C++ program to understand pointers and the concepts associated with them:



```
01. // Pointers.cpp : Defines the entry point for the console application.
02. //
03.
04. #include "stdafx.h"
05.
06. int main( )
07. {
08.  int iNumber = 3 ;
09.  int *piNumber ;
10.  piNumber = &iNumber ;
11.
12.  printf ( "\nAddress of iNumber = 0x%p", &iNumber ) ;
13.  printf ( "\nAddress of iNumber = 0x%p", piNumber ) ;
14.  printf ( "\nAddress of piNumber = 0x%p", &piNumber ) ;
15.  printf ( "\nValue of piNumber = %p", piNumber ) ;
16.  printf ( "\nValue of iNumber = %d", iNumber ) ;
17.  printf ( "\nValue of iNumber = %d", *( &iNumber ) ) ;
18.  printf ( "\nValue of iNumber = %d", *piNumber ) ;
19. }
```

Figure 8.1: *Pointers.cpp*

Line 8 of C/C++ code defines a variable of type integer. In lines 9 and 10, we see two operators, * and

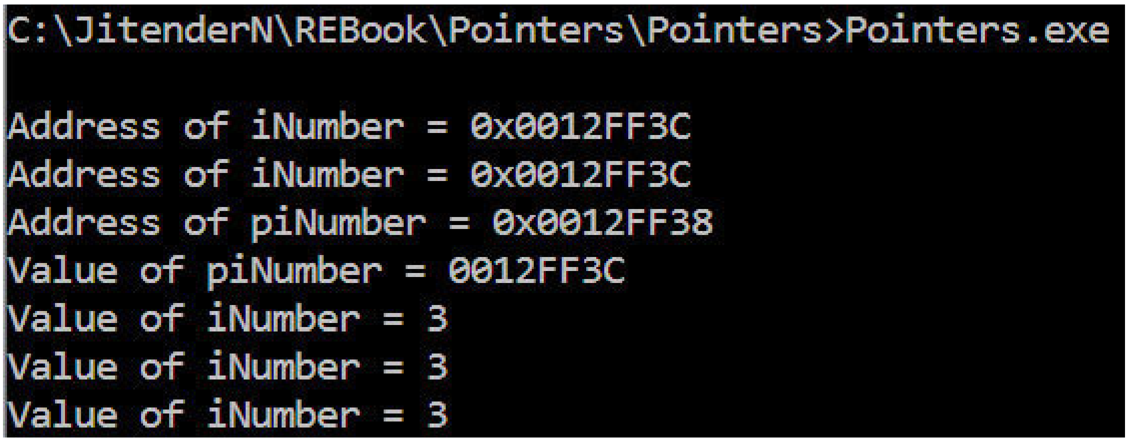
First, we will discuss the & operator. It means the address of the operator. So, **&iNumber** returns the memory address of the variable which in the preceding case was

Second is the * operator. It is called the value at address operator. It gives the value stored at the particular address. So, *

(&iNumber) returns the value at **0x0012FF3C** memory location, which is 3.

On line 9, **piNumber** is declared as the pointer variable, which means it is capable of holding memory addresses. Declaring **int *piNumber** does not mean that **piNumber** contains an integer value. What it means is that **piNumber** will hold the memory address of an integer variable. Similarly, **Float *pf** means that **pf** will hold the address of a floating point variable. The output of the preceding C/C++ program is:

```
Address of iNumber = 0x0012FF3C

Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF38
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
Value of iNumber = 3
```



```
C:\JitenderN\REBook\Pointers\Pointers>Pointers.exe

Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF38
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
Value of iNumber = 3
```

Figure 8.2: *Pointers.exe output*

You will get different output while executing the preceding code. Let us see what we get on compiling the code with and without optimization.

Pointer without Optimization

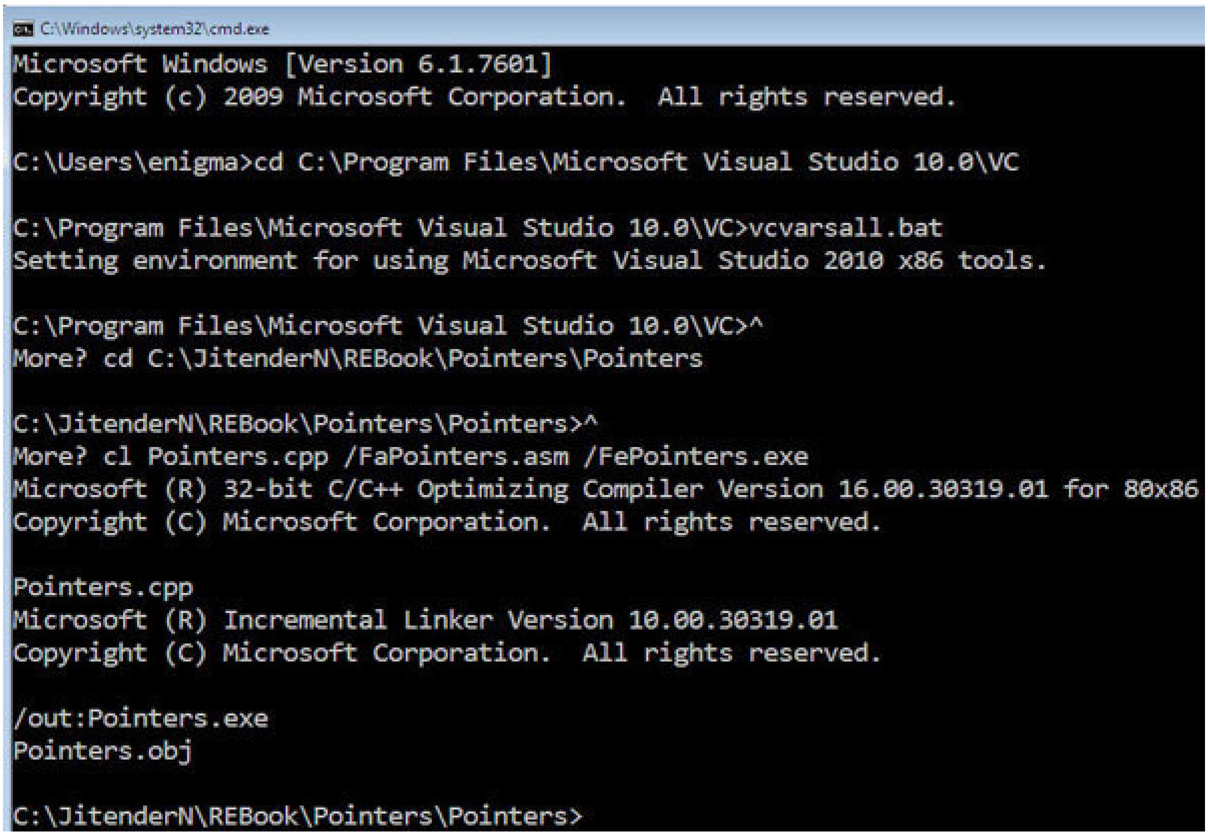
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

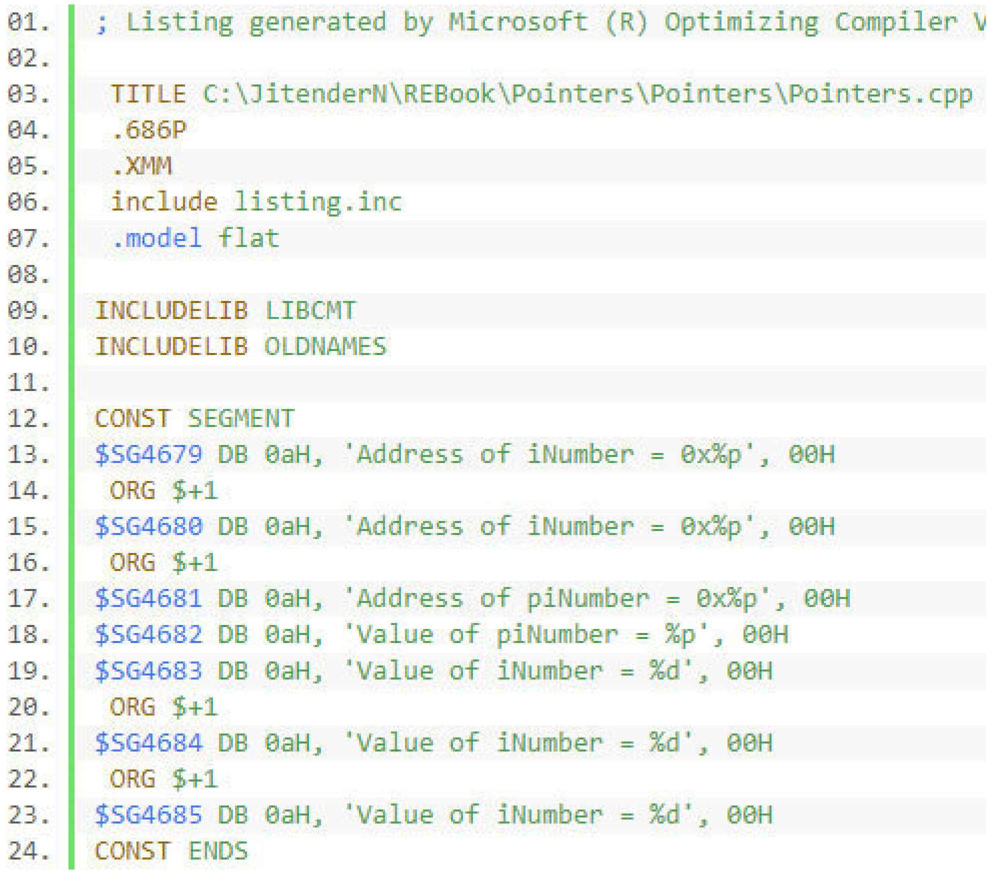
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Pointers\Pointers

C:\JitenderN\REBook\Pointers\Pointers>^
More? cl Pointers.cpp /FaPointers.asm /FePointers.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Pointers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Pointers.exe
Pointers.obj

C:\JitenderN\REBook\Pointers\Pointers>
```

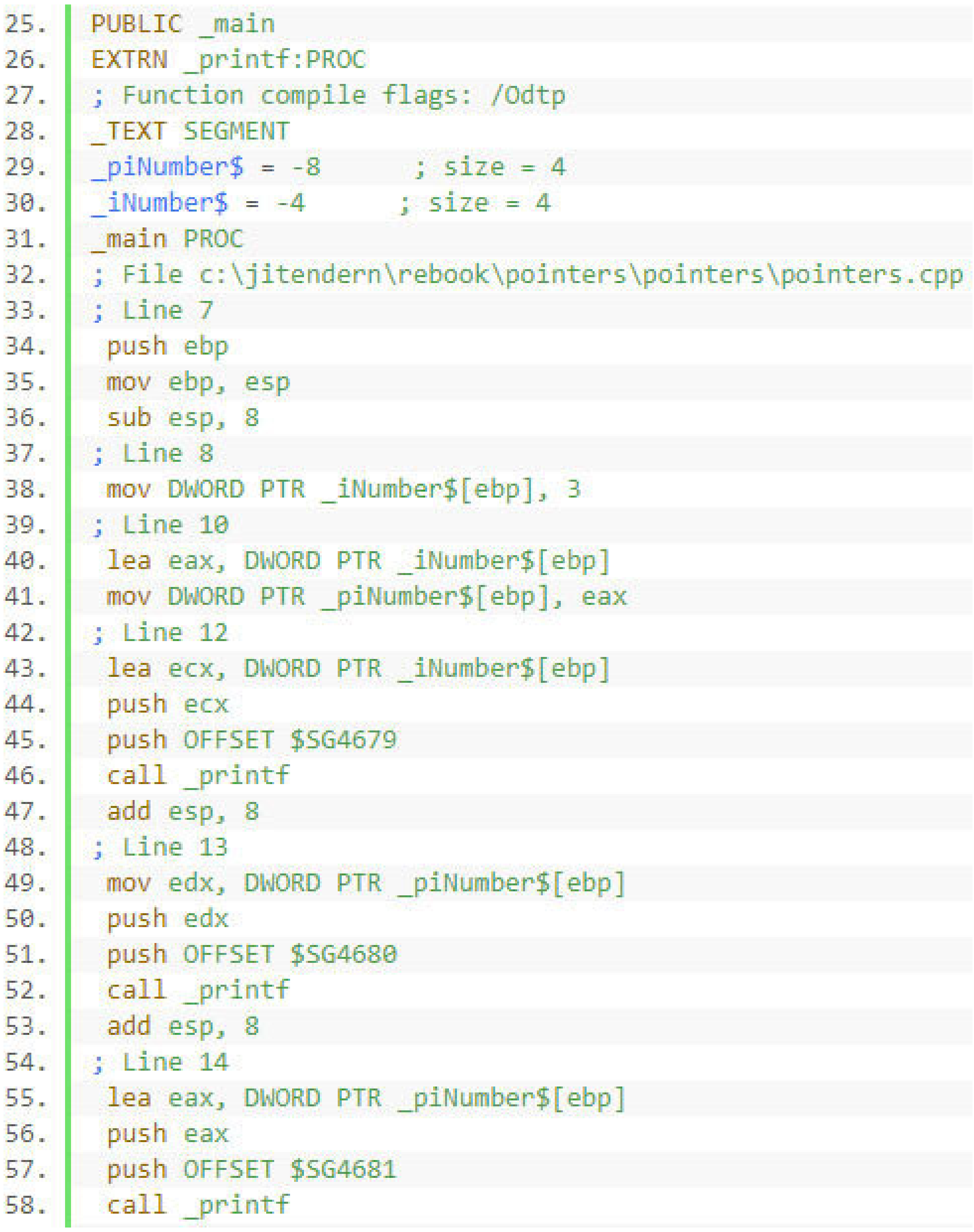
Figure 8.3: *Pointer without Optimization*

The generated assembly code will be as follows:



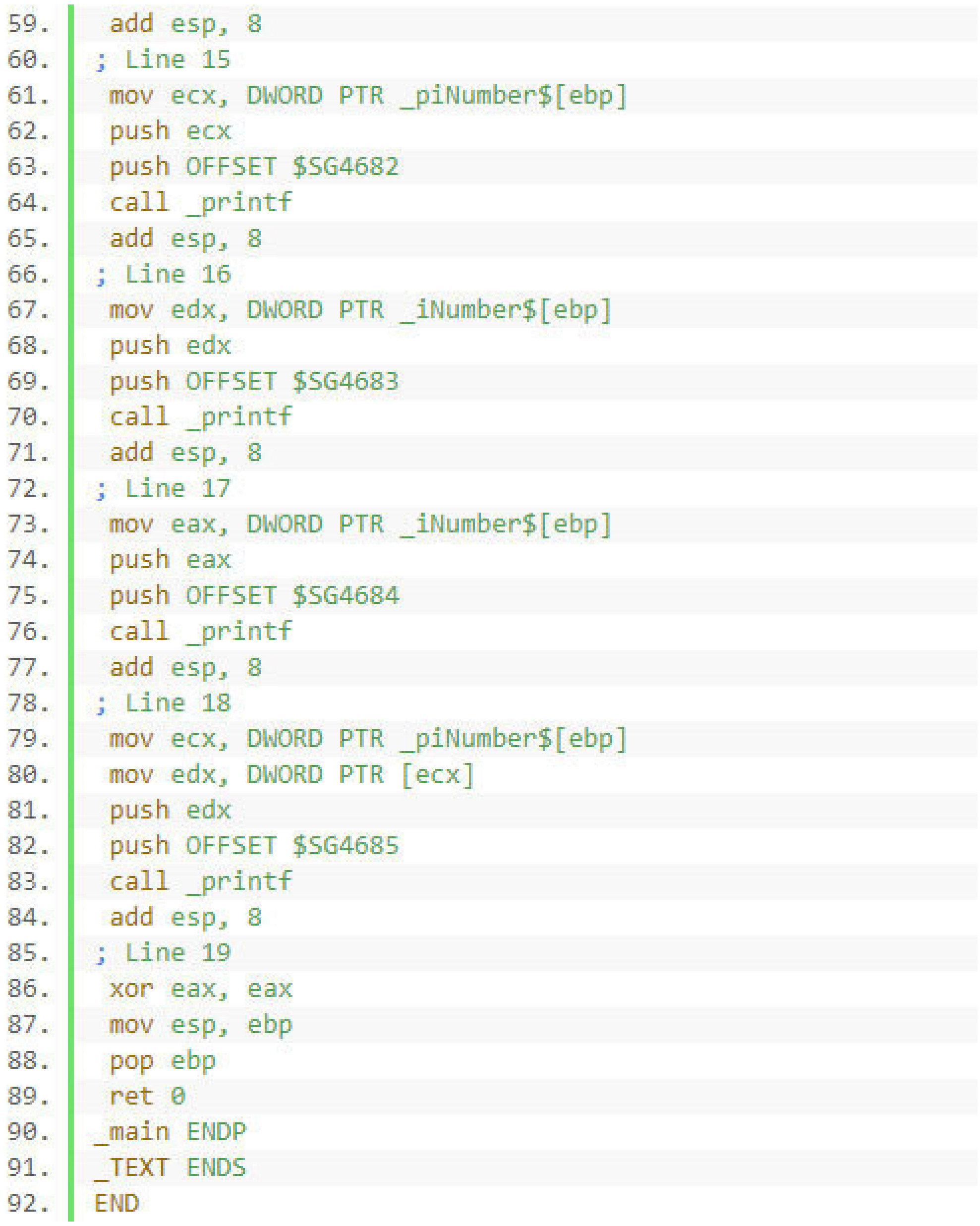
```
; Listing generated by Microsoft (R) Optimizing Compiler V

 TITLE C:\JitenderN\REBook\Pointers\Pointers\Pointers.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4679 DB 0aH, 'Address of iNumber = 0x%p', 00H
 ORG $+1
$SG4680 DB 0aH, 'Address of iNumber = 0x%p', 00H
 ORG $+1
$SG4681 DB 0aH, 'Address of piNumber = 0x%p', 00H
$SG4682 DB 0aH, 'Value of piNumber = %p', 00H
$SG4683 DB 0aH, 'Value of iNumber = %d', 00H
 ORG $+1
$SG4684 DB 0aH, 'Value of iNumber = %d', 00H
 ORG $+1
$SG4685 DB 0aH, 'Value of iNumber = %d', 00H
CONST ENDS
```

Figure 8.4: Pointers.asm-Part 1



```
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_piNumber$ = -8       ; size = 4
_iNumber$ = -4       ; size = 4
_main PROC
; File c:\jitendern\rebook\pointers\pointers\pointers.cpp
; Line 7
 push ebp
 mov ebp, esp
 sub esp, 8
; Line 8
 mov DWORD PTR _iNumber$[ebp], 3
; Line 10
 lea eax, DWORD PTR _iNumber$[ebp]
 mov DWORD PTR _piNumber$[ebp], eax
; Line 12
 lea ecx, DWORD PTR _iNumber$[ebp]
 push ecx
 push OFFSET $SG4679
 call _printf
 add esp, 8
; Line 13
 mov edx, DWORD PTR _piNumber$[ebp]
 push edx
 push OFFSET $SG4680
 call _printf
 add esp, 8
; Line 14
 lea eax, DWORD PTR _piNumber$[ebp]
 push eax
 push OFFSET $SG4681
 call _printf
```

Figure 8.5: *Pointers.asm-Part 2*



```
59.   add esp, 8
60.  ; Line 15
61.   mov ecx, DWORD PTR _piNumber$[ebp]
62.   push ecx
63.   push OFFSET $SG4682
64.   call _printf
65.   add esp, 8
66.  ; Line 16
67.   mov edx, DWORD PTR _iNumber$[ebp]
68.   push edx
69.   push OFFSET $SG4683
70.   call _printf
71.   add esp, 8
72.  ; Line 17
73.   mov eax, DWORD PTR _iNumber$[ebp]
74.   push eax
75.   push OFFSET $SG4684
76.   call _printf
77.   add esp, 8
78.  ; Line 18
79.   mov ecx, DWORD PTR _piNumber$[ebp]
80.   mov edx, DWORD PTR [ecx]
81.   push edx
82.   push OFFSET $SG4685
83.   call _printf
84.   add esp, 8
85.  ; Line 19
86.   xor eax, eax
87.   mov esp, ebp
88.   pop ebp
89.   ret 0
90.  _main ENDP
91.  _TEXT ENDS
92.  END
```

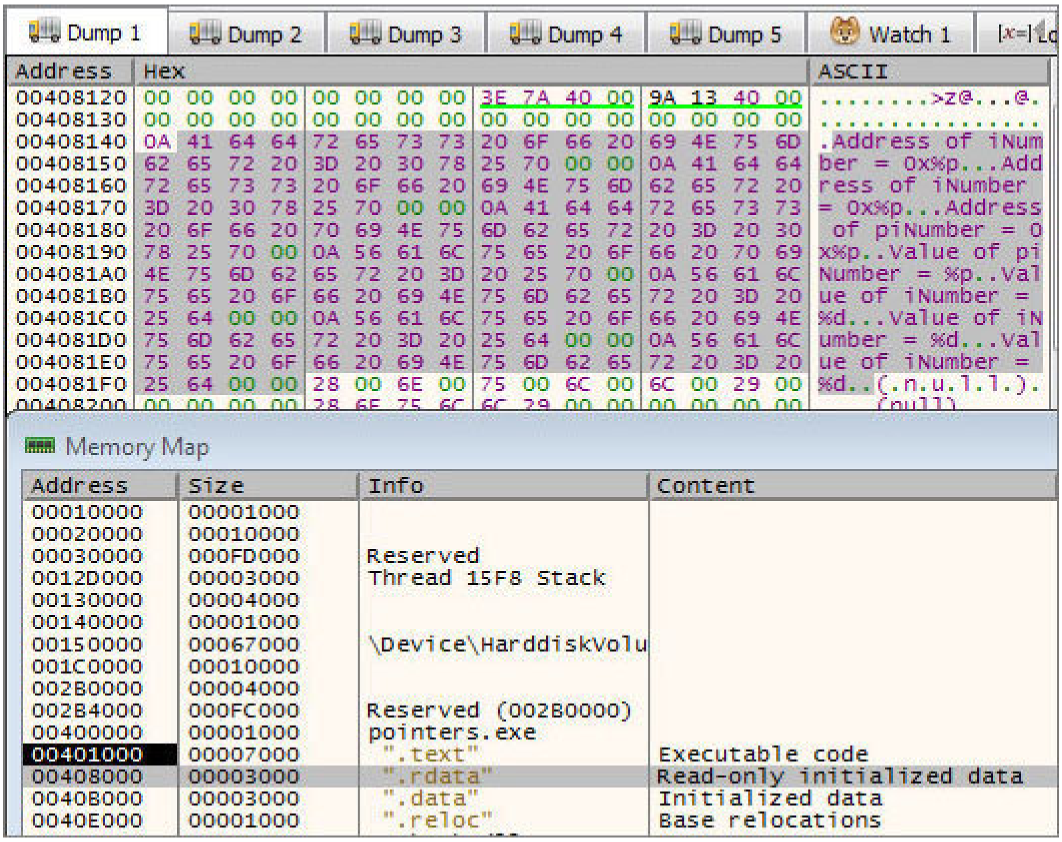
Figure 8.6: *Pointers.asm-Part 3*

The code generated contains two segments: **CONST** named by linker) and **_TEXT** segment (for code). Let's walk through the segments in the assembly listing:

▼Line 12-24
CONST SEGMENT
$SG4679 DB 0aH, 'Address of iNumber = 0x%p', 00H

```
ORG $+1

$SG4680 DB 0aH, 'Address of iNumber = 0x%p', 00H
ORG $+1
$SG4681 DB 0aH, 'Address of piNumber = 0x%p', 00H
$SG4682 DB 0aH, 'Value of piNumber = %p', 00H
$SG4683 DB 0aH, 'Value of iNumber = %d', 00H
ORG $+1
$SG4684 DB 0aH, 'Value of iNumber = %d', 00H
ORG $+1
$SG4685 DB 0aH, 'Value of iNumber = %d', 00H
CONST ENDS
```

Constant strings in the code are terminated by zero byte and are allocated in the **CONST** segment, which can be seen by dumping **.rdata** using x32dbg.



```
Address   Hex                                               ASCII
00408120  00 00 00 00 00 00 00 00 3E 7A 40 00 9A 13 40 00  ........>z@...@.
00408130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00408140  0A 41 64 64 72 65 73 73 20 6F 66 20 69 4E 75 6D  .Address of iNum
00408150  62 65 72 20 3D 20 30 78 25 70 00 00 0A 41 64 64  ber = 0x%p...Add
00408160  72 65 73 73 20 6F 66 20 69 4E 75 6D 62 65 72 20  ress of iNumber 
00408170  3D 20 30 78 25 70 00 00 0A 41 64 64 72 65 73 73  = 0x%p...Address
00408180  20 6F 66 20 70 69 4E 75 6D 62 65 72 20 3D 20 30   of piNumber = 0
00408190  78 25 70 00 0A 56 61 6C 75 65 20 6F 66 20 70 69  x%p..Value of pi
004081A0  4E 75 6D 62 65 72 20 3D 20 25 70 00 0A 56 61 6C  Number = %p..Val
004081B0  75 65 20 6F 66 20 69 4E 75 6D 62 65 72 20 3D 20  ue of iNumber = 
004081C0  25 64 00 00 0A 56 61 6C 75 65 20 6F 66 20 69 4E  %d...Value of iN
004081D0  75 6D 62 65 72 20 3D 20 25 64 00 00 0A 56 61 6C  umber = %d...Val
004081E0  75 65 20 6F 66 20 69 4E 75 6D 62 65 72 20 3D 20  ue of iNumber = 
004081F0  25 64 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  %d..(.n.u.l.l.).
00408200  00 00 00 00 28 6F 75 6C 6C 29 00 00 00 00 00 00      (null)
```

```
Address   Size      Info                   Content
00010000  00001000
00020000  00010000
00030000  000FD000  Reserved
0012D000  00003000  Thread 15F8 Stack
00130000  00004000
00140000  00001000
00150000  00067000  \Device\HarddiskVolu
001C0000  00010000
002B0000  00004000
002B4000  000FC000  Reserved (002B0000)
00400000  00001000  pointers.exe
00401000  00007000   ".text"               Executable code
00408000  00003000   ".rdata"              Read-only initialized data
0040B000  00003000   ".data"               Initialized data
0040E000  00001000   ".reloc"              Base relocations
```

Figure 8.7: .rdata

▼Line 28-30

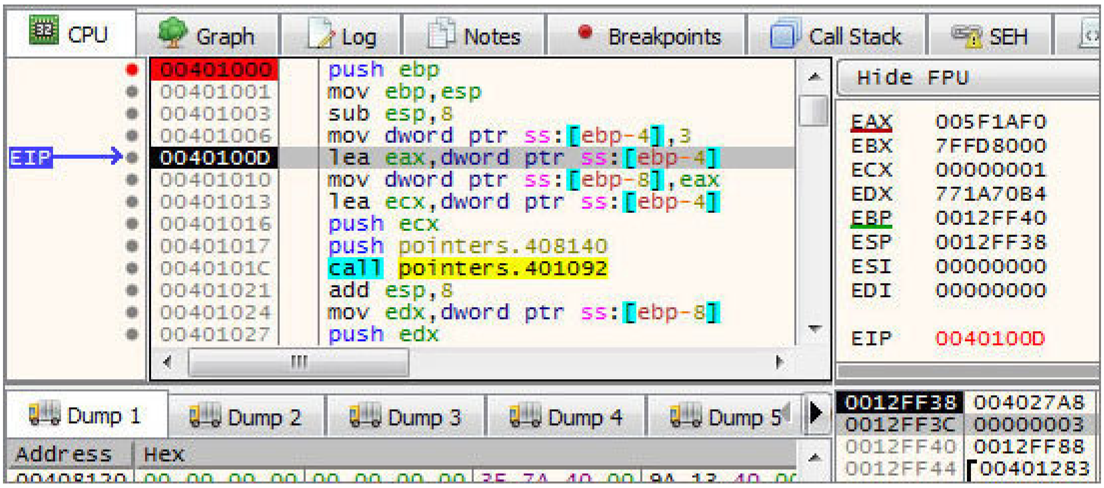
```
_TEXT SEGMENT
_piNumber$ = -8        ; size = 4
_iNumber$ = -4         ; size = 4
```

To access the local variable on the stack frame, we have to add **_$** to the EBP address. So, to access the **piNumber** variable on stack, we have to add -8 to the EBP address and add -4 to EBP to access the **iNumber** variable.

▼Line 31-38

```
_main PROC
; File c:\jitendern\rebook\pointers\pointers\pointers.cpp
; Line 7
push ebp
mov ebp, esp
sub esp, 8
; Line 8
mov DWORD PTR _iNumber$[ebp], 3
```

Let us understand the **main** procedure by placing the breakpoint at the start of the **main** call can be located by scrolling to the top of the disassembled code). Once the breakpoint is set, run the program and step into the instructions one by one. The **main** procedure starts with a usual function prologue and then with a **SUB** instruction. The **SUB** instruction is allocating 8 bytes on the stack for the **main** function local variables. Once the space is allocated on stack, the **MOV** instruction will push 3 on the main stack frame at [EBP-0x4] as shown in the following screenshot:



```
00401000   push ebp
00401001   mov ebp,esp
00401003   sub esp,8
00401006   mov dword ptr ss:[ebp-4],3
0040100D   lea eax,dword ptr ss:[ebp-4]
00401010   mov dword ptr ss:[ebp-8],eax
00401013   lea ecx,dword ptr ss:[ebp-4]
00401016   push ecx
00401017   push pointers.408140
0040101C   call pointers.401092
00401021   add esp,8
00401024   mov edx,dword ptr ss:[ebp-8]
00401027   push edx
```

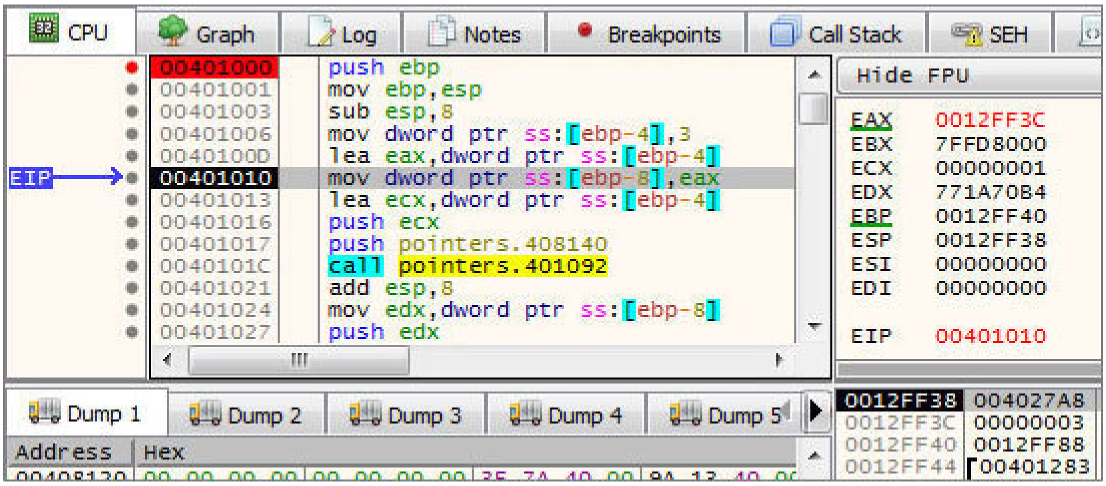
```
EAX   005F1AF0
EBX   7FFD8000
ECX   00000001
EDX   771A70B4
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   0040100D
```

```
0012FF38  004027A8
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.8: *Stack state*

Line 39-40
; Line 10
lea eax, DWORD PTR _iNumber$[ebp]

Load Effective Address loads the address of [EBP-0x4] into EAX. As we know that **iNumber** = which is stored on stack at **[EBP-0x4] = 0x0012FF3C** memory location. This instruction loads EAX with as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
```

```
EAX 0012FF3C
EBX 7FFD8000
ECX 00000001
EDX 771A70B4
EBP 0012FF40
ESP 0012FF38
ESI 00000000
EDI 00000000

EIP 00401010
```

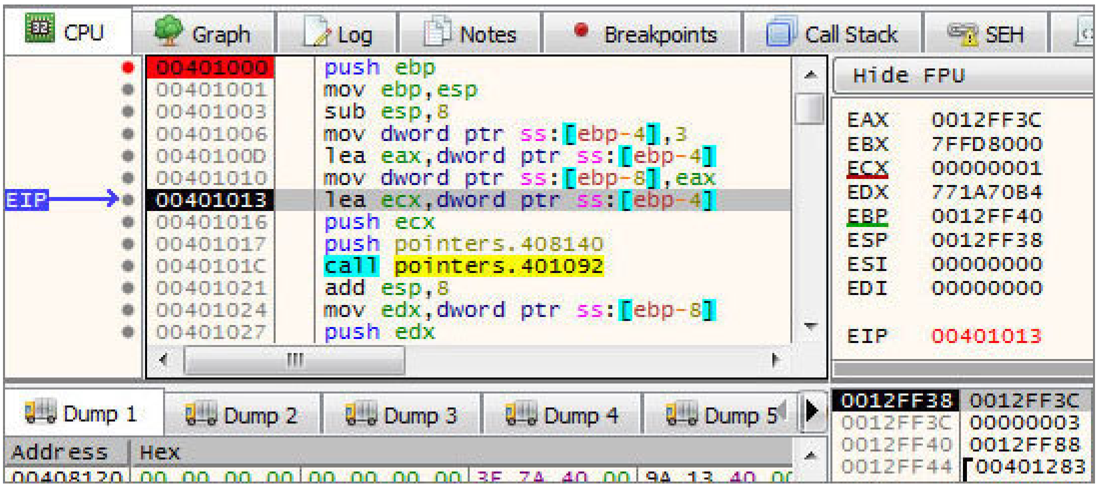
```
0012FF38  004027A8
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.9: *LEA output*

▼Line 41

mov DWORD PTR _piNumber$[ebp], eax

This will move the memory location stored in EAX onto the stack at [EBP-0x8]. Now we have both values stored on the stack, integer value which is 0x00000003 and the pointer to the integer value, as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
```

```
EAX 0012FF3C
EBX 7FFD8000
ECX 00000001
EDX 771A70B4
EBP 0012FF40
ESP 0012FF38
ESI 00000000
EDI 00000000

EIP 00401013
```

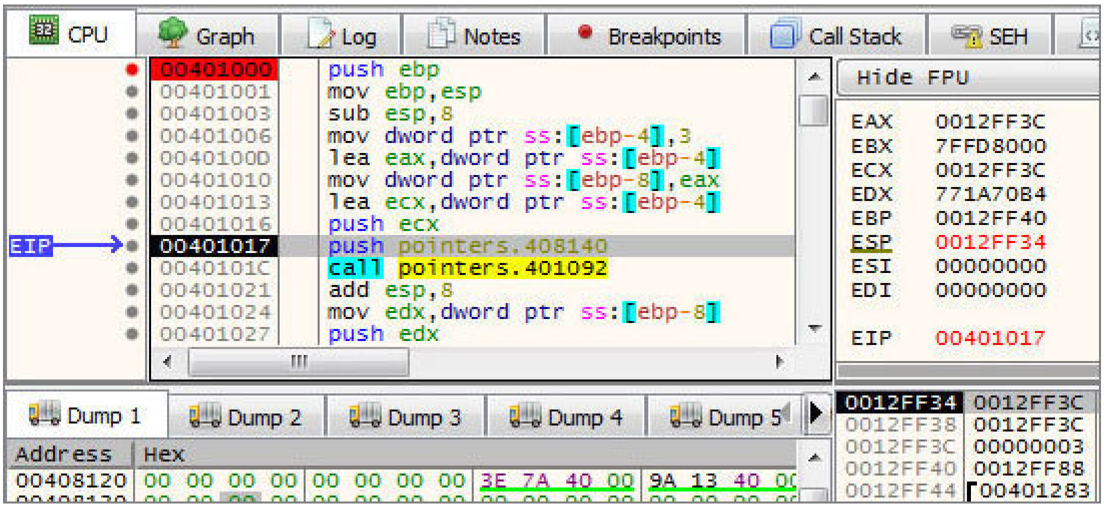
```
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.10: *piNumber on stack*

▼Line 42-45

```
; Line 12
lea ecx, DWORD PTR _iNumber$[ebp]
push ecx
```

Before calling the first **printf** function, we will have to push the arguments onto the stack. The first parameter that will be pushed on the stack will be the address of LEA will load the address of **iNumber** into ECX, which is later pushed on the stack, as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
```

```
EAX   0012FF3C
EBX   7FFD8000
ECX   0012FF3C
EDX   771A70B4
EBP   0012FF40
ESP   0012FF34
ESI   00000000
EDI   00000000

EIP   00401017
```

```
00408120  00 00 00 00 00 00 00 00 3E 7A 40 00 9A 13 40 0
```

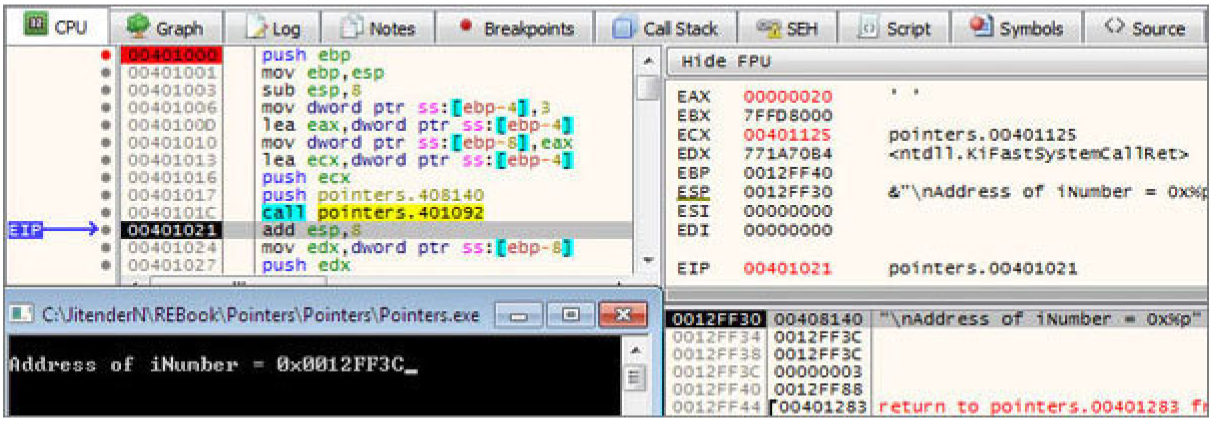
```
0012FF34  0012FF3C
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.11: *Before calling first printf function*

▼Line 45-46

```
push OFFSET $SG4679
call _printf
```

This push instruction is pushing another argument to which is a string constant referred to by Once both the parameters are pushed, call to the **printf** function is made. While debugging in x32dbg, we are stepping over (using **Debug** > **Step Over** option) during the **printf** call. Return from the **printf** function call will print on the console the **Address of** as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
```

```
EAX    00000020      ' '
EBX    7FFD8000
ECX    00401125      pointers.00401125
EDX    771A70B4      <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF30      &"\nAddress of iNumber = 0x%p

EIP    00401021      pointers.00401021
```

```
0012FF30  00408140  "\nAddress of iNumber = 0x%p"
0012FF34  0012FF3C
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283  return to pointers.00401283 fr
```

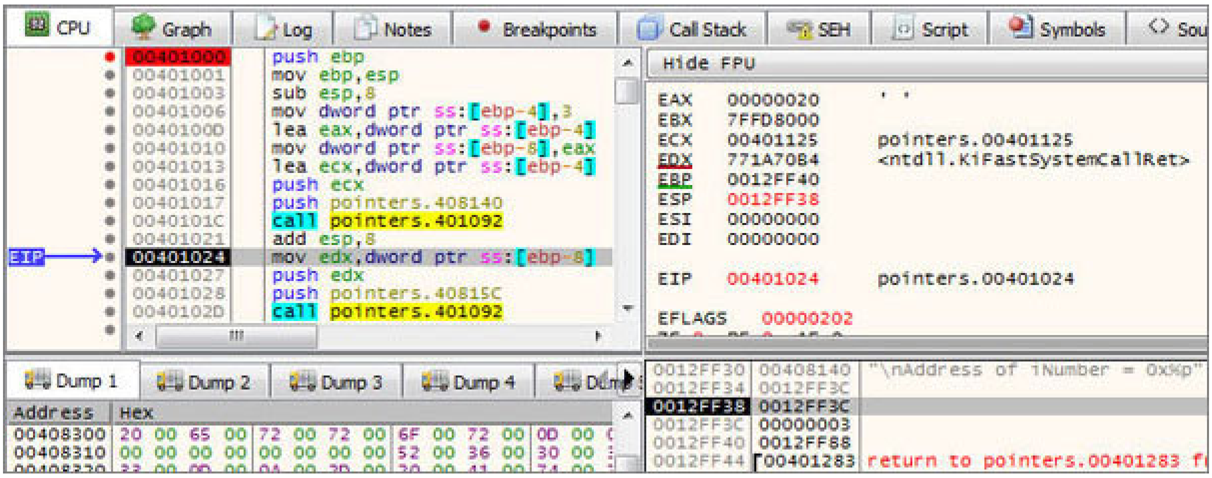
```
Address of iNumber = 0x0012FF3C_
```

Figure 8.12: *Pointer.exe output*

▼Line 47

add esp, 8

On returning from the **printf** procedure, the stack is cleaned by adding 8 bytes to ESP, as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
00401028  push pointers.40815C
0040102D  call pointers.401092
```

```
EAX     00000020     ' '
EBX     7FFD8000
ECX     00401125     pointers.00401125
EDX     771A7084     <ntdll.KiFastSystemCallRet>
EBP     0012FF40
ESP     0012FF38
ESI     00000000
EDI     00000000

EIP     00401024     pointers.00401024

EFLAGS    00000202
```

```
Address   Hex
00408300  20 00 65 00|72 00 72 00|6F 00 72 00|0D 00
00408310  00 00 00 00|00 00 00 00|52 00 36 00|30 00
```

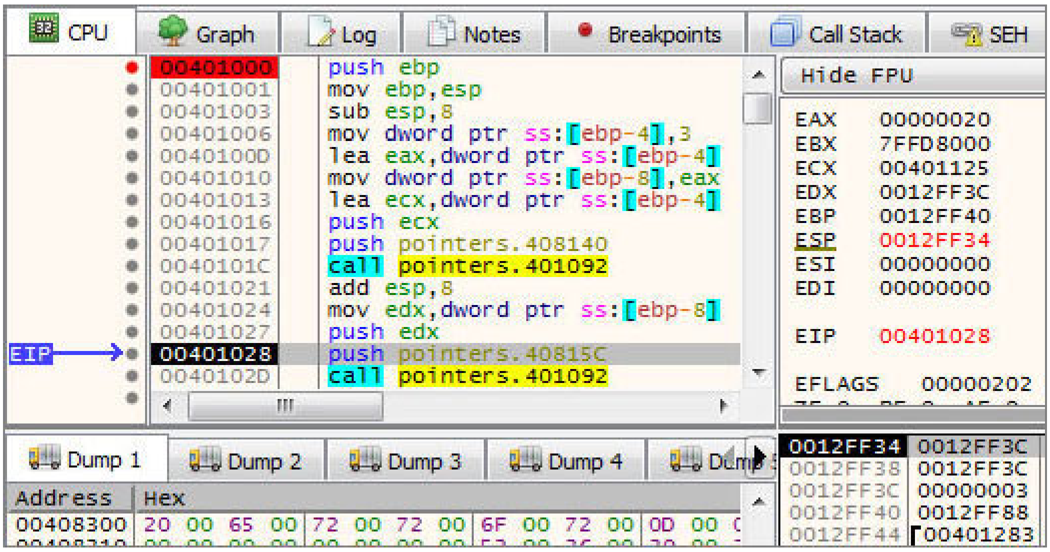
```
0012FF30  00408140  "\nAddress of iNumber = 0x%p"
0012FF34  0012FF3C
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283  return to pointers.00401283 f
```

Figure 8.13: Stack clean

▼Line 48-50

```
; Line 13
mov edx, DWORD PTR _piNumber$[ebp]
push edx
```

Now this is preparing the stack for the second **printf** function call. It will first **MOV** the value stored at [EBP-0x8] to the EDX register and then push EDX onto the stack, as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
00401028  push pointers.40815C
0040102D  call pointers.401092
```

```
EAX     00000020
EBX     7FFD8000
ECX     00401125
EDX     0012FF3C
EBP     0012FF40
ESP     0012FF34
ESI     00000000
EDI     00000000

EIP     00401028

EFLAGS    00000202
```

```
Address   Hex
00408300  20 00 65 00|72 00 72 00|6F 00 72 00|0D 00
```

```
0012FF34  0012FF3C
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.14: EDX onto the stack

▼Line 51-53

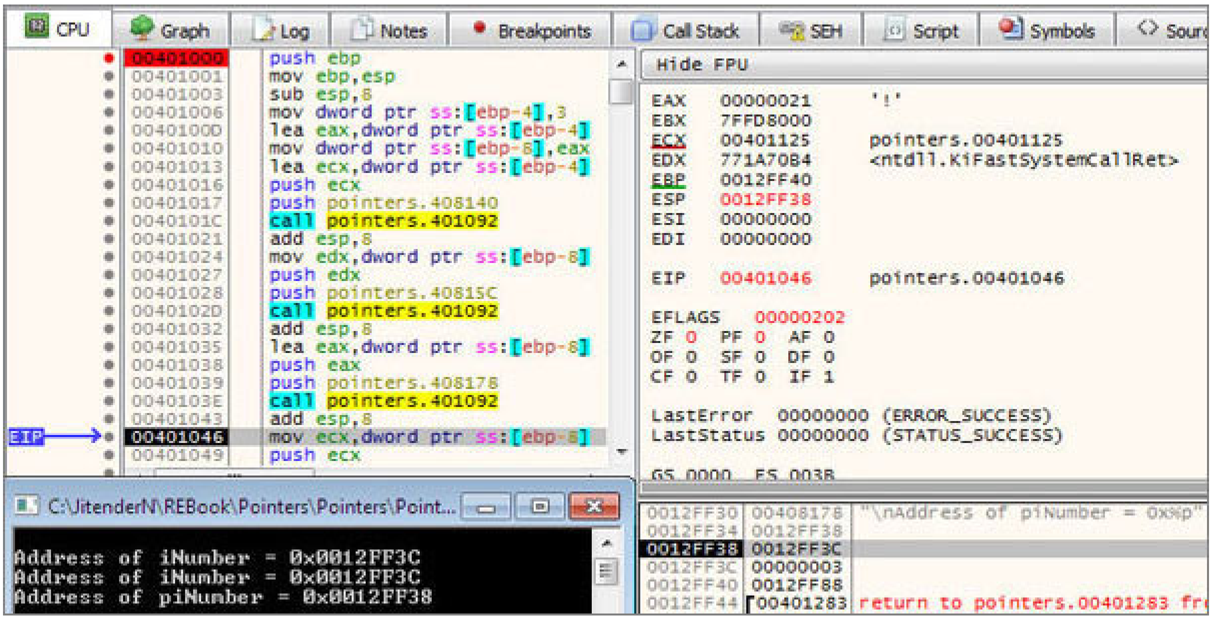
```
push OFFSET $SG4680
call _printf
add esp, 8
```

This is pushing another argument to the **printf** function, which is a string constant onto the stack. Now again, both the arguments to **printf** are pushed on the stack. The call to the **printf** function is made. On return, **printf** will print **Address of iNumber = piNumber** and the stack will again be cleaned after the return.

▼Line 54-59

```
; Line 14
lea eax, DWORD PTR _piNumber$[ebp]
push eax
push OFFSET $SG4681
call _printf
add esp, 8
```

In third the **printf** call, we have to print the address of which is For this, LEA is used to load the address of [EBP-0x8] to EAX and then pushed on the stack before the **printf** call. The string constant, represented by is also pushed on the stack before another **printf** call, as shown in the following screenshot:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  mov dword ptr ss:[ebp-4],3
0040100D  lea eax,dword ptr ss:[ebp-4]
00401010  mov dword ptr ss:[ebp-8],eax
00401013  lea ecx,dword ptr ss:[ebp-4]
00401016  push ecx
00401017  push pointers.408140
0040101C  call pointers.401092
00401021  add esp,8
00401024  mov edx,dword ptr ss:[ebp-8]
00401027  push edx
00401028  push pointers.40815C
0040102D  call pointers.401092
00401032  add esp,8
00401035  lea eax,dword ptr ss:[ebp-8]
00401038  push eax
00401039  push pointers.408178
0040103E  call pointers.401092
00401043  add esp,8
00401046  mov ecx,dword ptr ss:[ebp-8]
00401049  push ecx
```

```
EAX    00000021      '!'
EBX    7FFD8000
ECX    00401125      pointers.00401125
EDX    771A7084      <ntdll.KiFastSystemCallRet>
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    00401046      pointers.00401046

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
```

```
0012FF30  00408178  "\nAddress of piNumber = 0x%p"
0012FF34  0012FF38
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283  return to pointers.00401283 fr
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF38
```

Figure 8.15: Print address of piNumber

▼Line 60-62

```
; Line 15
mov ecx, DWORD PTR _piNumber$[ebp]
push ecx
```

Now we have to print, value of So the **MOV** instruction will move the **piNumber** variable value at [EBP-ox8] into ECX. In the next instruction, it will push ECX onto the stack before the **printf** call.

▼Line 63-65

```
push OFFSET $SG4682
call _printf
```

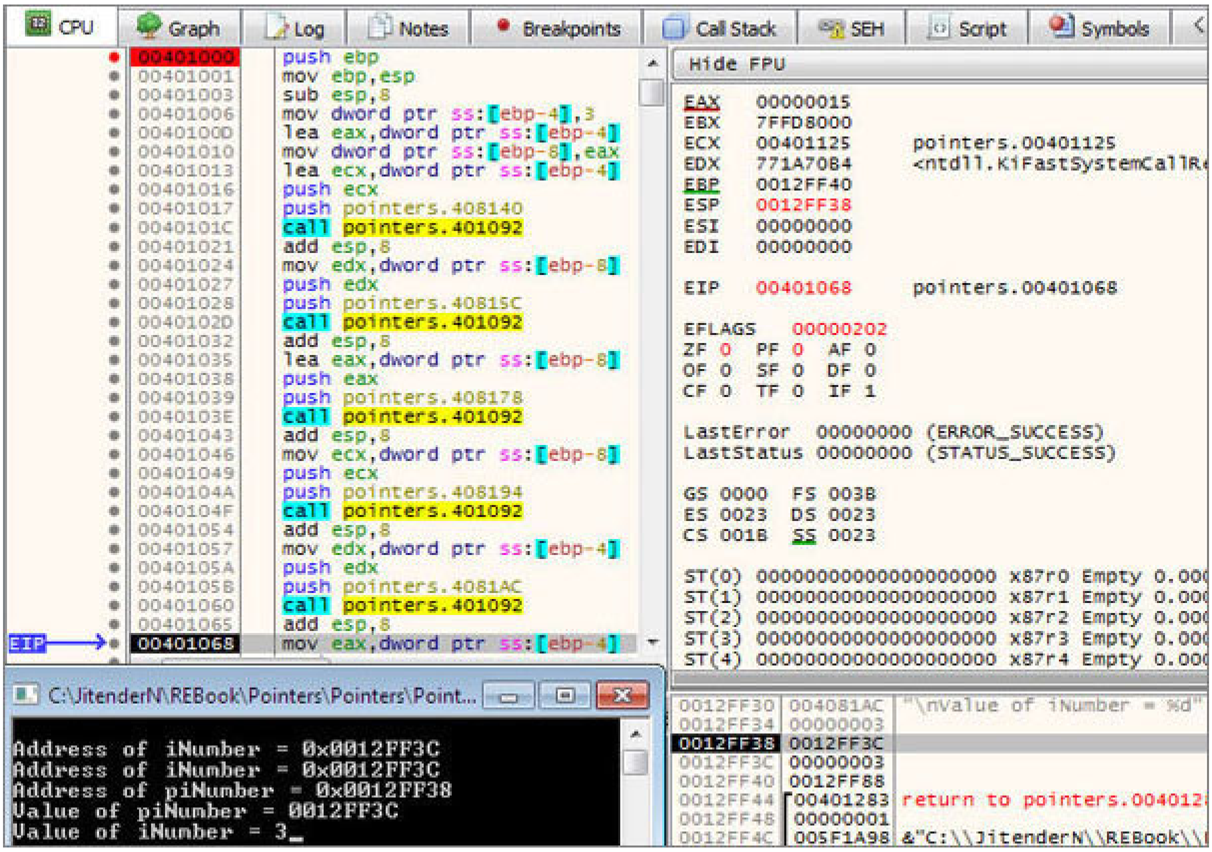
```
add esp, 8
```

These are the same as we did in the earlier **printf** call. It then cleans up the stack with the **ADD** instruction.

▼Line 66-71

```
; Line 16
mov edx, DWORD PTR _iNumber$[ebp]
push edx

push OFFSET $SG4683
call _printf
add esp, 8
```

This instruction will move the **iNumber** variable value at [EBP-0x4] into EDX. Before the **printf** call, it pushes both the arguments (string constant and EDX value) on the stack. Whatever is pushed onto the stack before pushing the string constant will be printed on the output console, as shown in the following screenshot:



```
00401000   push ebp
00401001   mov ebp,esp
00401003   sub esp,8
00401006   mov dword ptr ss:[ebp-4],3
0040100D   lea eax,dword ptr ss:[ebp-4]
00401010   mov dword ptr ss:[ebp-8],eax
00401013   lea ecx,dword ptr ss:[ebp-4]
00401016   push ecx
00401017   push pointers.408140
0040101C   call pointers.401092
00401021   add esp,8
00401024   mov edx,dword ptr ss:[ebp-8]
00401027   push edx
00401028   push pointers.40815C
0040102D   call pointers.401092
00401032   add esp,8
00401035   lea eax,dword ptr ss:[ebp-8]
00401038   push eax
00401039   push pointers.408178
0040103E   call pointers.401092
00401043   add esp,8
00401046   mov ecx,dword ptr ss:[ebp-8]
00401049   push ecx
0040104A   push pointers.408194
0040104F   call pointers.401092
00401054   add esp,8
00401057   mov edx,dword ptr ss:[ebp-4]
0040105A   push edx
0040105B   push pointers.4081AC
00401060   call pointers.401092
00401065   add esp,8
00401068   mov eax,dword ptr ss:[ebp-4]
```

```
EAX    00000015
EBX    7FFD8000
ECX    00401125     pointers.00401125
EDX    771A7084     <ntdll.KiFastSystemCallR
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    00401068     pointers.00401068

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000000 x87r0 Empty 0.000
ST(1) 0000000000000000000000 x87r1 Empty 0.000
ST(2) 0000000000000000000000 x87r2 Empty 0.000
ST(3) 0000000000000000000000 x87r3 Empty 0.000
ST(4) 0000000000000000000000 x87r4 Empty 0.000
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF38
Value of piNumber = 0012FF3C
Value of iNumber = 3_
```

```
0012FF30  004081AC  "\nValue of iNumber = %d"
0012FF34  00000003
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283  return to pointers.004012
0012FF48  00000001
0012FF4C  005F1A98  &"C:\\JitenderN\\REBook\\
```

Figure 8.16: *Printing iNumber*

▼Line 72-77

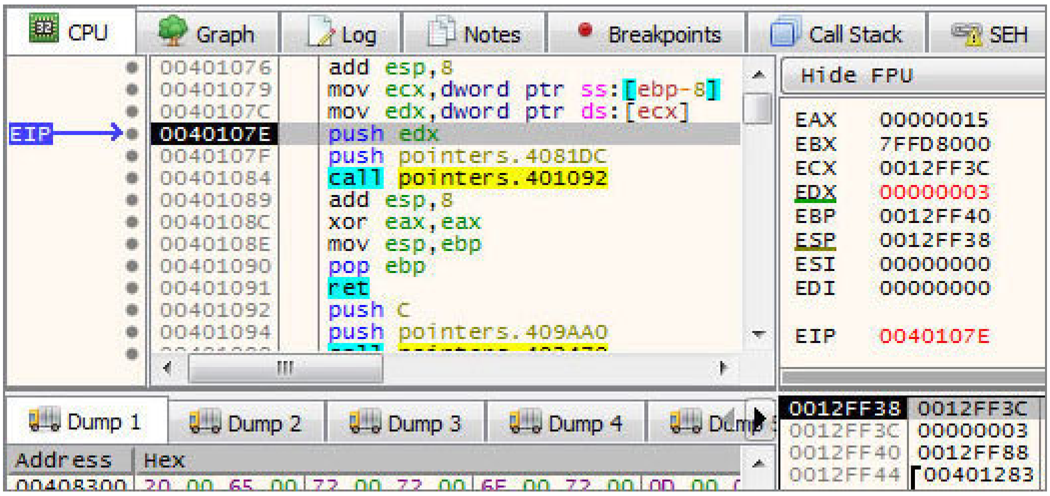
```
; Line 17
mov eax, DWORD PTR _iNumber$[ebp]
push eax
push OFFSET $SG4684
call _printf

add esp, 8
```

These instructions are doing the same as the earlier ones, except that they are using EAX for pushing the **iNumber** variable value at [EBP-0x4] onto the stack.

▼Line 78-80

```
; Line 18
mov ecx, DWORD PTR _piNumber$[ebp]
mov edx, DWORD PTR [ecx]
```

The first **MOV** instruction will move the **piNumber** variable value at [EBP-0x8] into ECX. The second instruction will move the value stored at a memory location pointed by ECX into EDX, as shown in the following screenshot:



```
00401076    add esp,8
00401079    mov ecx,dword ptr ss:[ebp-8]
0040107C    mov edx,dword ptr ds:[ecx]
0040107E    push edx
0040107F    push pointers.4081DC
00401084    call pointers.401092
00401089    add esp,8
0040108C    xor eax,eax
0040108E    mov esp,ebp
00401090    pop ebp
00401091    ret
00401092    push C
00401094    push pointers.409AA0
```

```
EAX    00000015
EBX    7FFD8000
ECX    0012FF3C
EDX    00000003
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    0040107E
```

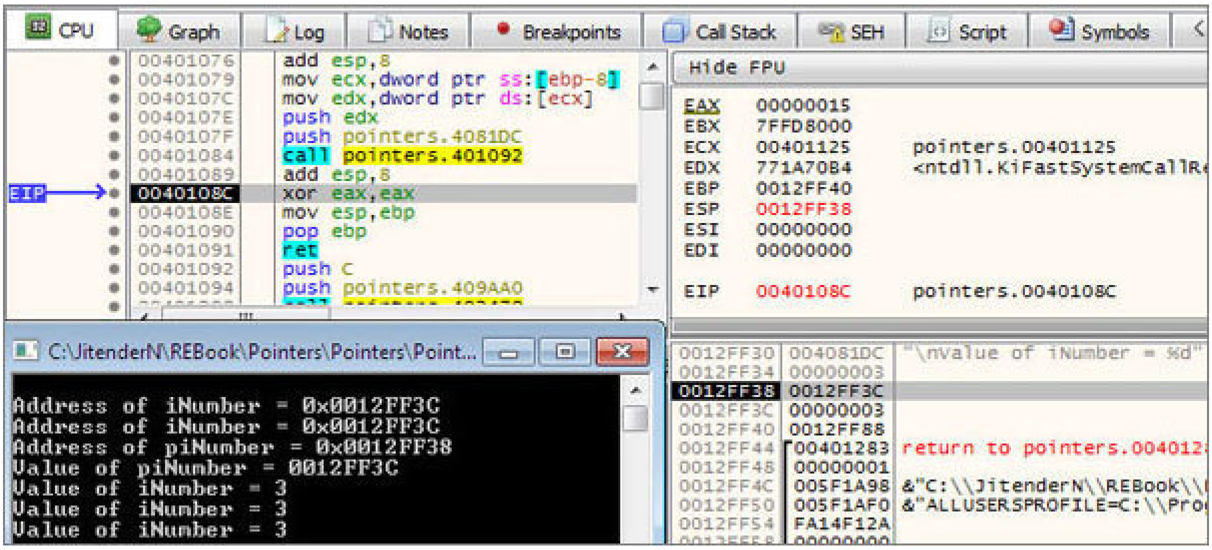
```
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283
```

Figure 8.17: *MOV [ECX] into EDX*

▼Line 81-84

```
push edx
push OFFSET $SG4685
call _printf
add esp, 8
```

The push instruction is pushing the arguments on the stack before the **printf** call. On return from the **printf** function, the value of **iNumber** will be printed on the console. Stack cleaning is done with the **ADD** instruction, as shown in the following screenshot:



```
00401076   add esp,8
00401079   mov ecx,dword ptr ss:[ebp-8]
0040107C   mov edx,dword ptr ds:[ecx]
0040107E   push edx
0040107F   push pointers.4081DC
00401084   call pointers.401092
00401089   add esp,8
0040108C   xor eax,eax
0040108E   mov esp,ebp
00401090   pop ebp
00401091   ret
00401092   push C
00401094   push pointers.409AA0
```

```
EAX   00000015
EBX   7FFD8000
ECX   00401125   pointers.00401125
EDX   771A70B4   <ntdll.KiFastSystemCallR
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   0040108C   pointers.0040108C
```

```
0012FF30  004081DC  "\nvalue of iNumber = %d"
0012FF34  00000003
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  00401283  return to pointers.004012
0012FF48  00000001
0012FF4C  005F1A98  &"C:\\JitenderN\\REBook\\
0012FF50  005F1AF0  &"ALLUSERSPROFILE=C:\\Pro
0012FF54  FA14F12A
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF38
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
Value of iNumber = 3
```

Figure 8.18: *iNumber will be printed*

▼Line 85-92

```
; Line 19
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

The remaining instructions are the same as what we discussed in the earlier sections. The main function epilogue is called to end the segment and code.

Pointer with Optimization

Compile the code with optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

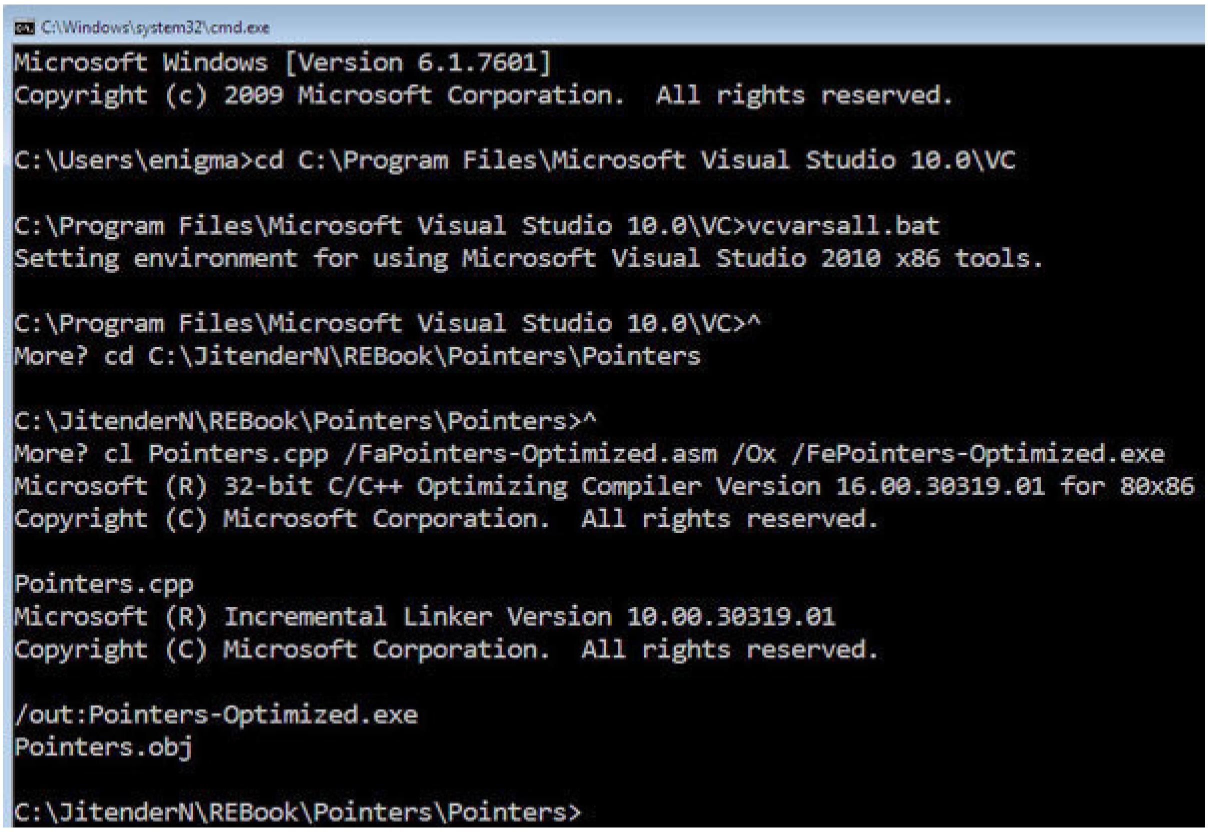
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

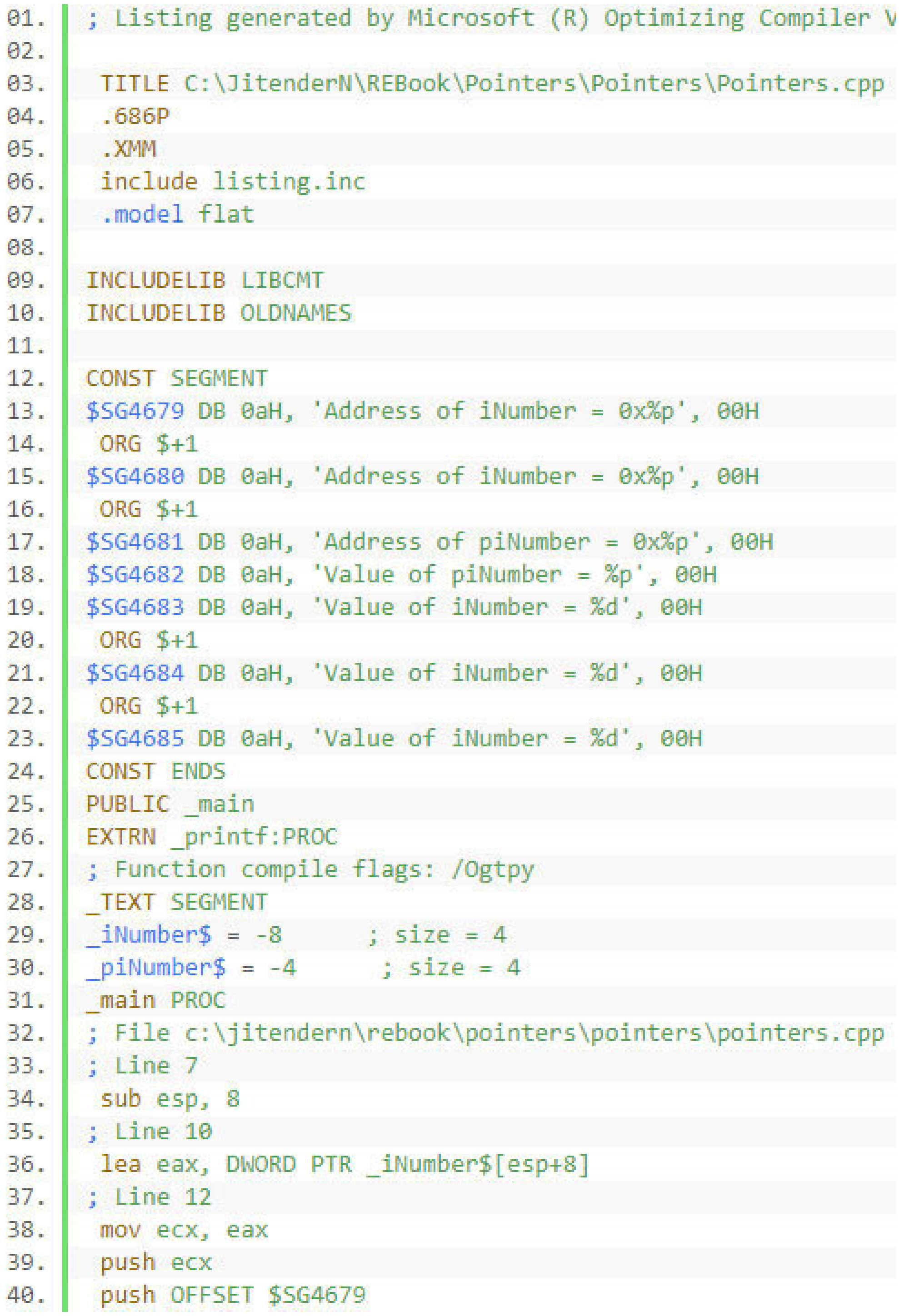
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Pointers\Pointers

C:\JitenderN\REBook\Pointers\Pointers>^
More? cl Pointers.cpp /FaPointers-Optimized.asm /Ox /FePointers-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Pointers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Pointers-Optimized.exe
Pointers.obj

C:\JitenderN\REBook\Pointers\Pointers>
```

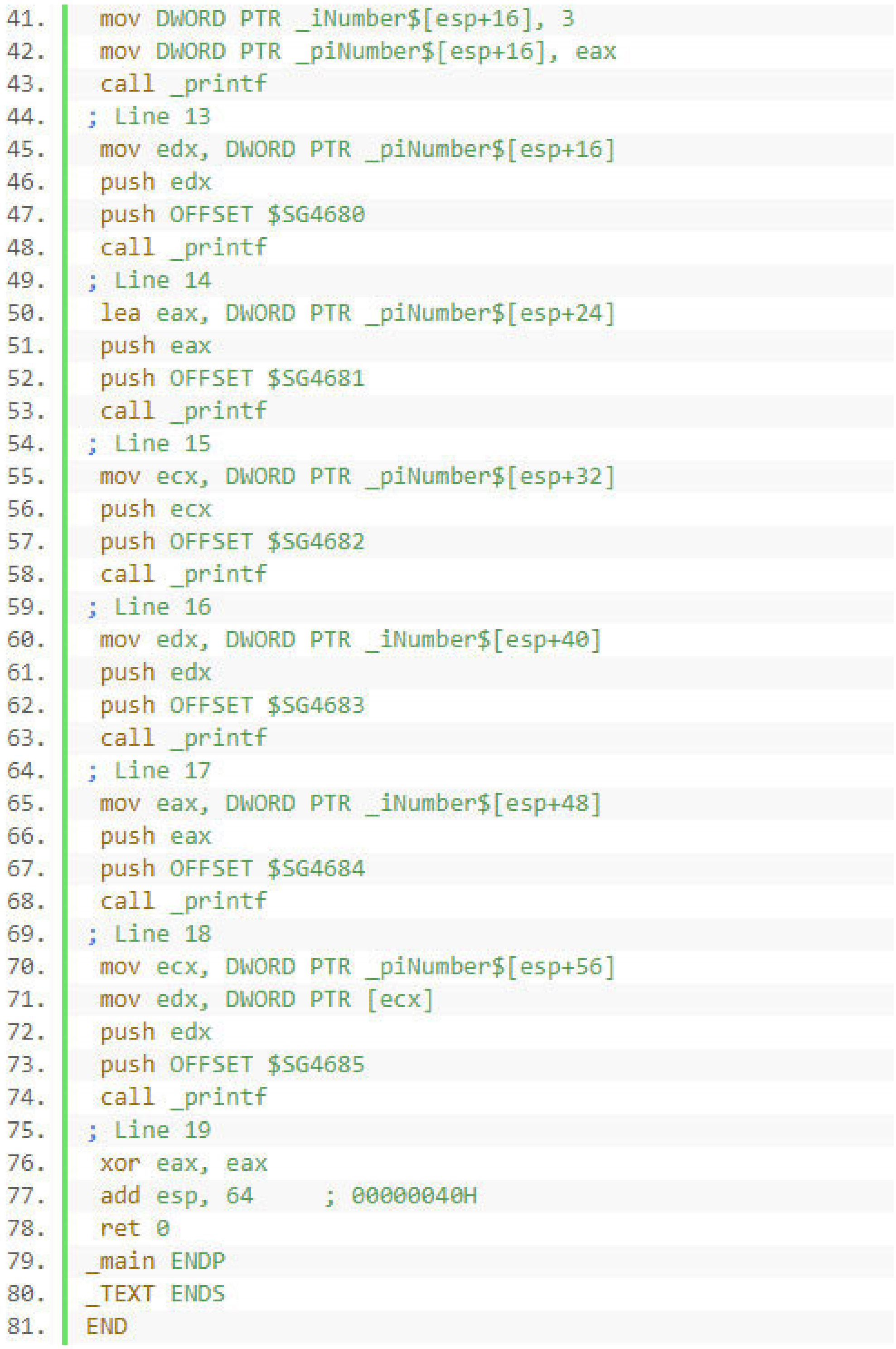
Figure 8.19: *Pointer with Optimization*

The generated assembly code will be as follows:



```
; Listing generated by Microsoft (R) Optimizing Compiler V

 TITLE C:\JitenderN\REBook\Pointers\Pointers\Pointers.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4679 DB 0aH, 'Address of iNumber = 0x%p', 00H
 ORG $+1
$SG4680 DB 0aH, 'Address of iNumber = 0x%p', 00H
 ORG $+1
$SG4681 DB 0aH, 'Address of piNumber = 0x%p', 00H
$SG4682 DB 0aH, 'Value of piNumber = %p', 00H
$SG4683 DB 0aH, 'Value of iNumber = %d', 00H
 ORG $+1
$SG4684 DB 0aH, 'Value of iNumber = %d', 00H
 ORG $+1
$SG4685 DB 0aH, 'Value of iNumber = %d', 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_iNumber$ = -8      ; size = 4
_piNumber$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\pointers\pointers\pointers.cpp
; Line 7
 sub esp, 8
; Line 10
 lea eax, DWORD PTR _iNumber$[esp+8]
; Line 12
 mov ecx, eax
 push ecx
 push OFFSET $SG4679
```

Figure 8.20: Pointers-Optimized.asm-Part 1



```
 mov DWORD PTR _iNumber$[esp+16], 3
 mov DWORD PTR _piNumber$[esp+16], eax
 call _printf
; Line 13
 mov edx, DWORD PTR _piNumber$[esp+16]
 push edx
 push OFFSET $SG4680
 call _printf
; Line 14
 lea eax, DWORD PTR _piNumber$[esp+24]
 push eax
 push OFFSET $SG4681
 call _printf
; Line 15
 mov ecx, DWORD PTR _piNumber$[esp+32]
 push ecx
 push OFFSET $SG4682
 call _printf
; Line 16
 mov edx, DWORD PTR _iNumber$[esp+40]
 push edx
 push OFFSET $SG4683
 call _printf
; Line 17
 mov eax, DWORD PTR _iNumber$[esp+48]
 push eax
 push OFFSET $SG4684
 call _printf
; Line 18
 mov ecx, DWORD PTR _piNumber$[esp+56]
 mov edx, DWORD PTR [ecx]
 push edx
 push OFFSET $SG4685
 call _printf
; Line 19
 xor eax, eax
 add esp, 64     ; 00000040H
 ret 0
_main ENDP
_TEXT ENDS
END
```

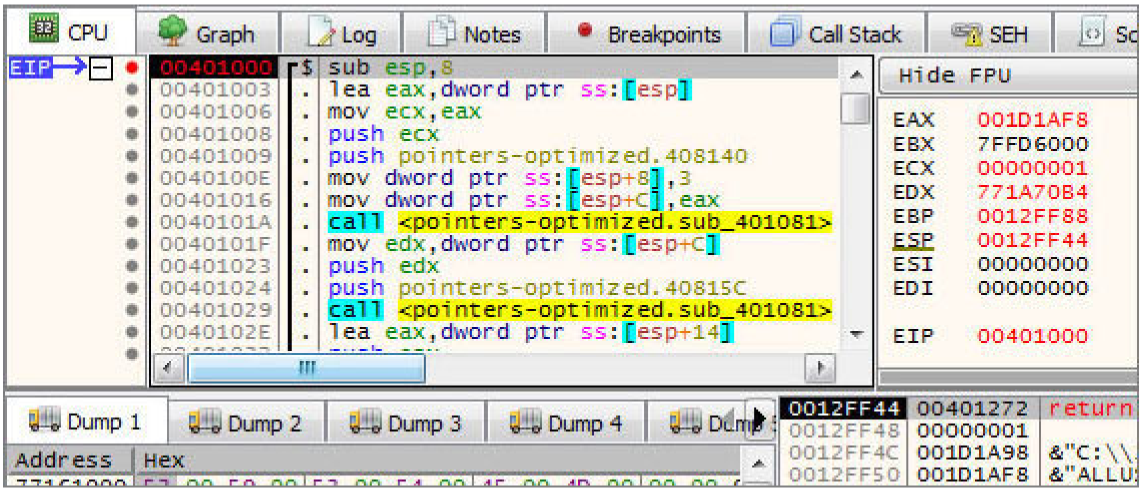
Figure 8.21: *Pointers-Optimized.asm-Part 2*

First, we will talk about the difference in the optimized and non-optimized code generated. In optimized code, the function prologue is removed, so all the places where EBP is referred is replaced with ESP. So, in the optimized assembly listing, we will observe the ESP reference in place of EBP as we observed in the non-optimized assembly listing. To explain this concept, we will check the stack state in detail for a better understanding. Let's walk through the assembly listing by placing breakpoint at the start of the **main** procedure:

▼Line 33-34

```
; Line 7
sub esp, 8
```

This is the start of the **main** procedure, where ESP is subtracted by 8 bytes to create room for the **main** function local variables, which are integer variable and pointer to that integer variable Before the execution of this instruction, the stack will look as follows:



```
00401000  $ sub esp,8
00401003  . lea eax,dword ptr ss:[esp]
00401006  . mov ecx,eax
00401008  . push ecx
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
0040102E  . lea eax,dword ptr ss:[esp+14]
```

```
EAX   001D1AF8
EBX   7FFD6000
ECX   00000001
EDX   771A70B4
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   00401000
```

```
0012FF44  00401272  return
0012FF48  00000001
0012FF4C  001D1A98  &"C:\\
0012FF50  001D1AF8  &"ALLU
```

Figure 8.22: Start of the main procedure

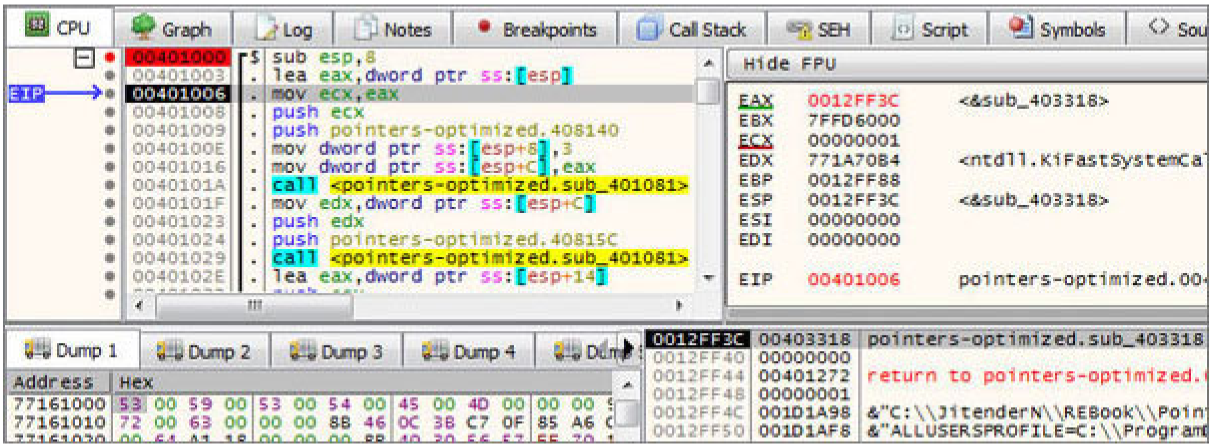
▼Line 35-36

; Line 10
lea eax, DWORD PTR _iNumber$[esp+8]

The **Load effective address** will load EAX with the address of which is evaluated to:

lea eax, DWORD PTR ss:[esp]
as _iNumber$ = -8

As we can see, EAX is filled with 0x0012FF3C, which is ESP itself. Now, EAX also points to the top of the stack as follows:



```
00401000  $ sub esp,8
00401003  . lea eax,dword ptr ss:[esp]
00401006  . mov ecx,eax
00401008  . push ecx
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
0040102E  . lea eax,dword ptr ss:[esp+14]
```

```
EAX   0012FF3C     <&sub_403318>
EBX   7FFD6000
ECX   00000001
EDX   771A70B4     <ntdll.KiFastSystemCa
EBP   0012FF88
ESP   0012FF3C     <&sub_403318>
ESI   00000000
EDI   00000000

EIP   00401006     pointers-optimized.00
```

```
Address   Hex
77161000  53 00 59 00 53 00 54 00 45 00 4D 00 00 00
77161010  72 00 63 00 00 00 8B 46 0C 3B C7 0F 85 A6
```

```
0012FF3C  00403318  pointers-optimized.sub_403318
0012FF40  00000000
0012FF44  00401272  return to pointers-optimized.
0012FF48  00000001
0012FF4C  001D1A98  &"C:\\JitenderN\\REBook\\Poin
0012FF50  001D1AF8  &"ALLUSERSPROFILE=C:\\Program
```

__Figure 8.23:__ EAX also points to the top of stack

▼Line 37-39

; Line 12
mov ecx, eax

```
push ecx
```

Move EAX to ECX. Now, ECX is also filled with which is pushed on the stack with the push **ecx** instruction. With this, our one argument to the **printf** function is pushed on the stack.

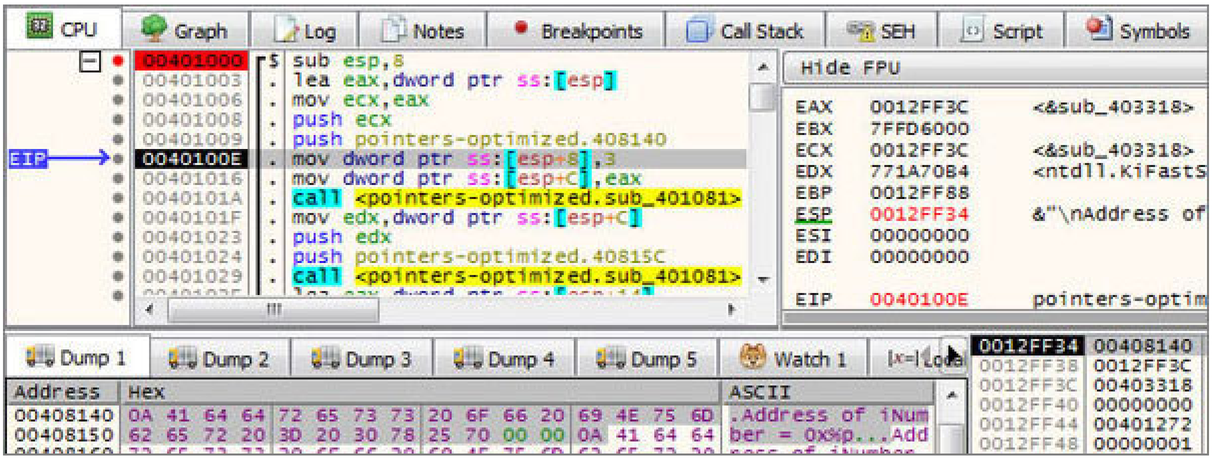
▼Line 40

```
push OFFSET $SG4679
```

This will push the string constant on the stack, which is another argument to the **printf** function. As we can see, both the arguments to the **printf** function are pushed on the stack. The stack state is shown in the following screenshot and can be understood in the following manner:

[ESP] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", arg to printf()

[ESP+0x4] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()
[ESP+0x8] 0012FF3C JUNKJUNK This where we will store iNumber
[ESP+0xC] 0012FF40 JUNKJUNK This where we will store &iNumber
[ESP+0x10] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
00401000  $ sub esp,8
00401003  . lea eax,dword ptr ss:[esp]
00401006  . mov ecx,eax
00401008  . push ecx
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
```

```
EAX   0012FF3C    <&sub_403318>
EBX   7FFD6000
ECX   0012FF3C    <&sub_403318>
EDX   771A70B4    <ntdll.KiFastS
EBP   0012FF88
ESP   0012FF34    &"\nAddress of
ESI   00000000
EDI   00000000

EIP   0040100E    pointers-optim
```

```
Address   Hex                                               ASCII
00408140  0A 41 64 64 72 65 73 73 20 6F 66 20 69 4E 75 6D  .Address of iNum
00408150  62 65 72 20 3D 20 30 78 25 70 00 00 0A 41 64 64  ber = 0x%p...Add
```

```
0012FF34  00408140
0012FF38  0012FF3C
0012FF3C  00403318
0012FF40  00000000
0012FF44  00401272
0012FF48  00000001
```

Figure 8.24: *Stack state*

▼Line 41-42

mov DWORD PTR _iNumber$[esp+16], 3

mov DWORD PTR _piNumber$[esp+16], eax

Both instructions are evaluated to:

mov dword ptr ss:[esp+0x8], 0x3

mov dword ptr ss:[esp+0xC], eax

Earlier, in the stack state, we allocated room for local variables of the **main** function on the stack (shown as in stack state). Now, using these instructions, we are storing the integer variable **0x00000003** at [ESP+0x8] and the pointer to this variable at [ESP+0xC]. Now, the stack will look like as follows:

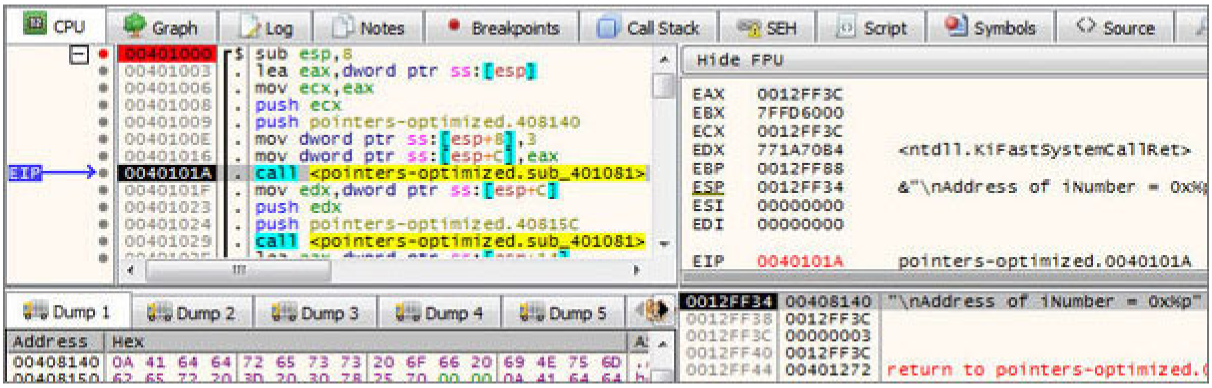
[ESP] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", argument to printf()

[ESP+0x4] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()

[ESP+0x8] 0012FF3C 00000003 iNumber is stored here

[ESP+0xC] 0012FF40 0012FF3C &iNumber is stored here

[ESP+0x10] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
00401000  $ sub esp,8
00401003  . lea eax,dword ptr ss:[esp]
00401006  . mov ecx,eax
00401008  . push ecx
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
```

```
EAX   0012FF3C
EBX   7FFD6000
ECX   0012FF3C
EDX   771A70B4    <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF34    &"\nAddress of iNumber = 0x%
ESI   00000000
EDI   00000000

EIP   0040101A    pointers-optimized.0040101A
```

```
Address   Hex
00408140  0A 41 64 64 72 65 73 73 20 6F 66 20 69 4E 75 6D
```

```
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.
```

Figure 8.25: *After Line 41-42 execution*

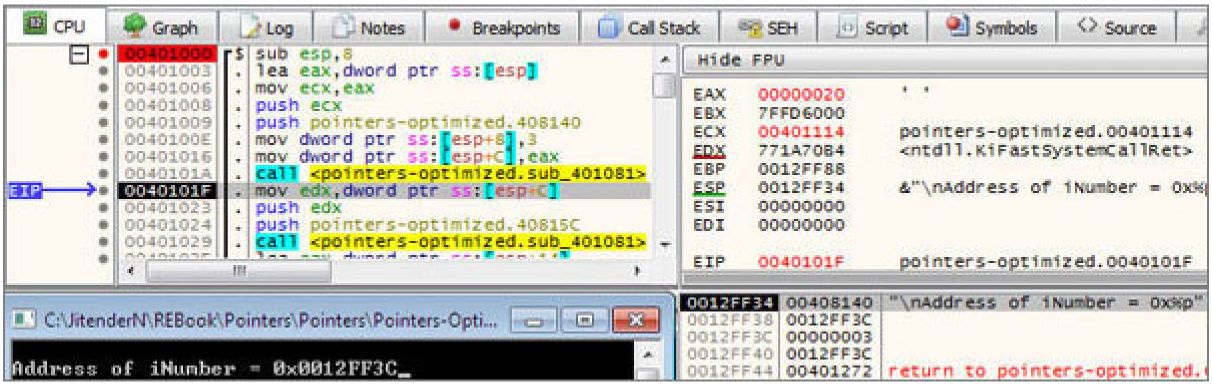
▼Line 43

call _printf

Once the arguments are pushed on the stack, call to **printf** is made. As per the C/C++ code at line 12:

printf ("\nAddress of iNumber = 0x%p", &iNumber) ;

So, on returning from the **printf** call, the following will be printed on the console:



```
00401000  $ sub esp,8
00401003  . lea eax,dword ptr ss:[esp]
00401006  . mov ecx,eax
00401008  . push ecx
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
```

```
EAX   00000020   ' '
EBX   7FFD6000
ECX   00401114   pointers-optimized.00401114
EDX   771A7084   <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF34   &"\nAddress of iNumber = 0x%

EIP   0040101F   pointers-optimized.0040101F
```

```
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.
```

```
Address of iNumber = 0x0012FF3C_
```

Figure 8.26: After printf execution

▼Line 44-48

```
; Line 13
mov edx, DWORD PTR _piNumber$[esp+16]
push edx
push OFFSET $SG4680
call _printf
```

The C/C++ code on line 13 prints which is the memory location of

printf ("\nAddress of iNumber = 0x%p", piNumber) ;

In our assembly code, arguments to the **printf** function are pushed on the stack by first moving EDX with the pointer to the Integer variable. The same EDX is then pushed on to stack with another push of string constant. Once we have both arguments to **printf** function on stack, call to **printf** function is made. As shown in the stack state and the following screenshot:

[ESP] 0012FF2C 0040815C "\nAddress of iNumber = 0x%p", argument to printf()

[ESP+0x04] 0012FF30 0012FF3C piNumber is stored here, argument to printf()

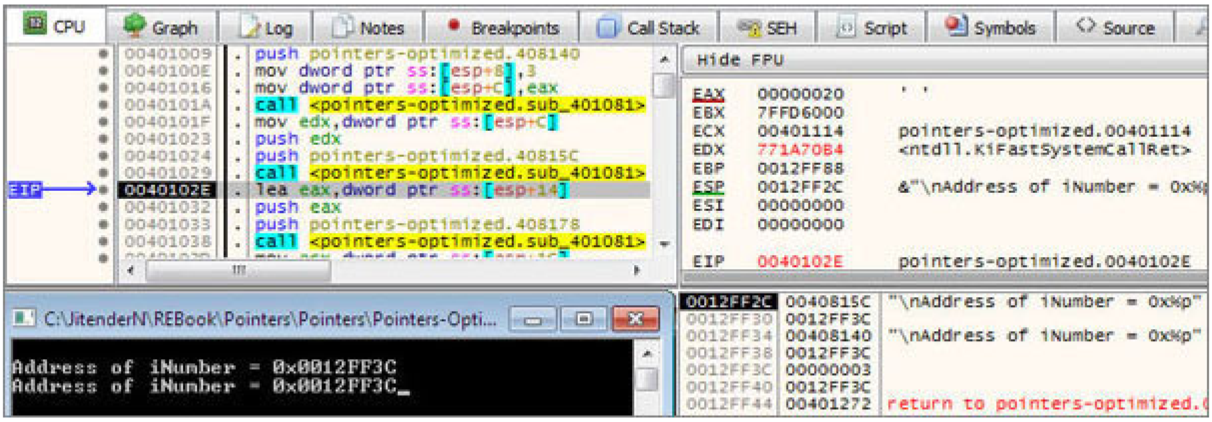
[ESP+0x08] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", argument to printf()

[ESP+0x0C] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()

[ESP+0x10] 0012FF3C 00000003 iNumber is stored here

[ESP+0x14] 0012FF40 0012FF3C &iNumber is stored here

[ESP+0x18] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
00401009  . push pointers-optimized.408140
0040100E  . mov dword ptr ss:[esp+8],3
00401016  . mov dword ptr ss:[esp+C],eax
0040101A  . call <pointers-optimized.sub_401081>
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
0040102E  . lea eax,dword ptr ss:[esp+14]
00401032  . push eax
00401033  . push pointers-optimized.408178
00401038  . call <pointers-optimized.sub_401081>
```

```
EAX   00000020     ' '
EBX   7FFD6000
ECX   00401114     pointers-optimized.00401114
EDX   771A70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF2C     &"\nAddress of iNumber = 0x%

EIP   0040102E     pointers-optimized.0040102E
```

```
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.
```

C:\JitenderN\REBook\Pointers\Pointers\Pointers-Opti...

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C_
```

***Figure 8.27:** After the second printf execution*

▼Line 49-53

```
; Line 14
lea eax, DWORD PTR _piNumber$[esp+24]
push eax
```

```
push OFFSET $SG4681
call _printf
```

The C/C++ code on line 14 prints which is the memory location of

```
printf ("\nAddress of piNumber = 0x%p", &piNumber) ;
```

In our assembly code, the LEA instruction will be evaluated to:

```
lea eax, ss:[esp+0x14]
```

LEA will load EAX with the memory address of which is the address of Once EAX is loaded with the address, it is pushed onto the stack with another push of the string constant. Once we have both arguments to the **printf** function on the stack, call to the **printf** function is made. This is shown in the following screenshot:

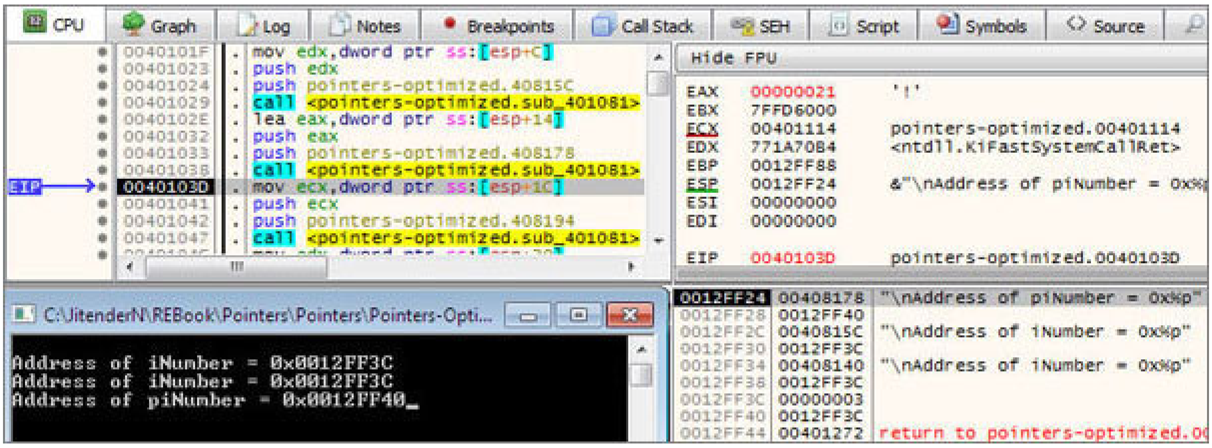
```
[ESP]  0012FF24  00408178  "\nAddress of piNumber = 0x%p",
argument to printf()
[ESP+0x04] 0012FF28  0012FF40  &piNumber is stored here,
argument to printf()

[ESP+0x08] 0012FF2C  0040815C  "\nAddress of iNumber =
0x%p", argument to printf()
[ESP+0x0C] 0012FF30  0012FF3C  piNumber is stored here,
argument to printf()
[ESP+0x10] 0012FF34  00408140  "\nAddress of iNumber =
0x%p", argument to printf()
```

[ESP+0x14] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()

[ESP+0x18] 0012FF3C 00000003 iNumber is stored here

[ESP+0x1C] 0012FF40 0012FF3C &iNumber is stored here

[ESP+0x20] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
0040101F  . mov edx,dword ptr ss:[esp+C]
00401023  . push edx
00401024  . push pointers-optimized.40815C
00401029  . call <pointers-optimized.sub_401081>
0040102E  . lea eax,dword ptr ss:[esp+14]
00401032  . push eax
00401033  . push pointers-optimized.408178
00401038  . call <pointers-optimized.sub_401081>
0040103D  . mov ecx,dword ptr ss:[esp+1C]
00401041  . push ecx
00401042  . push pointers-optimized.408194
00401047  . call <pointers-optimized.sub_401081>
```

```
EAX   00000021    '!'
EBX   7FFD6000
ECX   00401114    pointers-optimized.00401114
EDX   771A70B4    <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF24    &"\nAddress of piNumber = 0x%
ESI   00000000
EDI   00000000

EIP   0040103D    pointers-optimized.0040103D
```

```
0012FF24 00408178 "\nAddress of piNumber = 0x%p"
0012FF28 0012FF40
0012FF2C 0040815C "\nAddress of iNumber = 0x%p"
0012FF30 0012FF3C
0012FF34 00408140 "\nAddress of iNumber = 0x%p"
0012FF38 0012FF3C
0012FF3C 00000003
0012FF40 0012FF3C
0012FF44 00401272 return to pointers-optimized.0
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40_
```

Figure 8.28: *After thrid printf execution*

▼Line 54-58

```
; Line 15
mov ecx, DWORD PTR _piNumber$[esp+32]
push ecx
push OFFSET $SG4682
call _printf
```

The C/C++ code on line 15 prints which is the memory location of

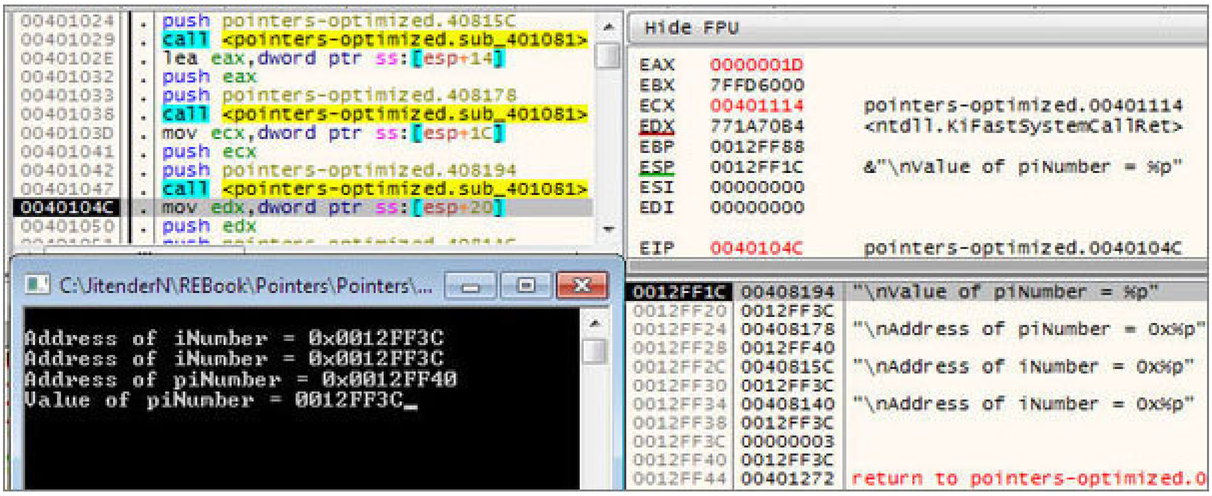
```
printf ("\nValue of piNumber = %p", piNumber) ;
```

In our assembly code, the **MOV** instruction will be evaluated to:

```
mov ecx, dword ptr ss:[esp+0x1C]
```

MOV will move the value at **ss:[esp+0x1C]** to ECX, which points to the memory location of Now, the arguments are again pushed to the stack for call to the **printf** function. This is shown in the following screenshot:

```
[ESP] 0012FF1C 00408194 "\nValue of piNumber = %p", argument to printf()
[ESP+0x04] 0012FF20 0012FF3C piNumber is stored here, argument to printf()
[ESP+0x08] 0012FF24 00408178 "\nAddress of piNumber = 0x%p", argument to printf()
[ESP+0x0C] 0012FF28 0012FF40 &piNumber is stored here, argument to printf()
[ESP+0x10] 0012FF2C 0040815C "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x14] 0012FF30 0012FF3C piNumber is stored here, argument to printf()
[ESP+0x18] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x1C] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()
[ESP+0x20] 0012FF3C 00000003 iNumber is stored here
[ESP+0x24] 0012FF40 0012FF3C &iNumber is stored here
[ESP+0x28] 0012FF44 00401272 return to 0x00401272 from 0x401000
```



```
00401024  .  push pointers-optimized.40815C
00401029  .  call <pointers-optimized.sub_401081>
0040102E  .  lea eax,dword ptr ss:[esp+14]
00401032  .  push eax
00401033  .  push pointers-optimized.408178
00401038  .  call <pointers-optimized.sub_401081>
0040103D  .  mov ecx,dword ptr ss:[esp+1C]
00401041  .  push ecx
00401042  .  push pointers-optimized.408194
00401047  .  call <pointers-optimized.sub_401081>
0040104C  .  mov edx,dword ptr ss:[esp+20]
00401050  .  push edx
```

```
EAX   0000001D
EBX   7FFD6000
ECX   00401114    pointers-optimized.00401114
EDX   771A7084    <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF1C    &"\nValue of piNumber = %p"
ESI   00000000
EDI   00000000

EIP   0040104C    pointers-optimized.0040104C
```

```
0012FF1C  00408194  "\nValue of piNumber = %p"
0012FF20  0012FF3C
0012FF24  00408178  "\nAddress of piNumber = 0x%p"
0012FF28  0012FF40
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.0
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40
Value of piNumber = 0012FF3C_
```

__Figure 8.29:__ After fourth printf execution

▼Line 59-63

```
; Line 16
mov edx, DWORD PTR _iNumber$[esp+40]
push edx
push OFFSET $SG4683
call _printf
```

The C/C++ code on line 16 prints which is the value of

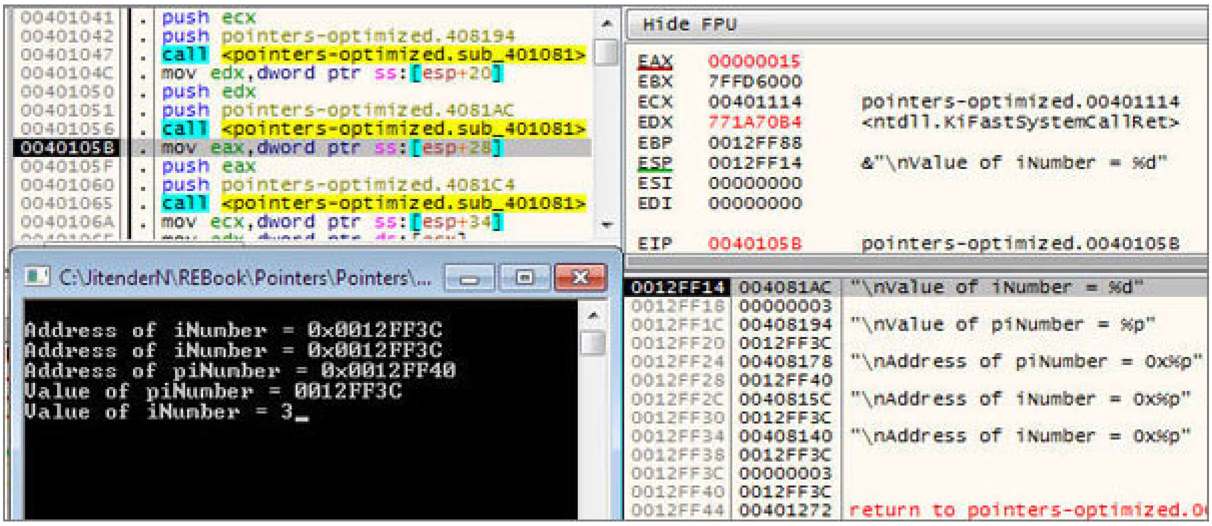
printf ("\nValue of iNumber = %d", iNumber) ;

In our assembly code, the **MOV** instruction will be evaluated to:

mov edx, dword ptr ss:[esp+0x20]

MOV will move the value at **ss:[esp+0x20]** to EDX, which is the value of Now, the arguments are again pushed to the stack for call to the **printf** function. This is shown in the following screenshot:

```
[ESP] 0012FF14 004081AC "\nValue of iNumber = %d",
argument to printf()
[ESP+0x04] 0012FF18 00000003 iNumber is stored here,
argument to printf()

[ESP+0x08] 0012FF1C 00408194 "\nValue of piNumber = %p",
argument to printf()
[ESP+0x0C] 0012FF20 0012FF3C piNumber is stored here,
argument to printf()
[ESP+0x10] 0012FF24 00408178 "\nAddress of piNumber =
0x%p", argument to printf()
[ESP+0x14] 0012FF28 0012FF40 &piNumber is stored here,
argument to printf()
[ESP+0x18] 0012FF2C 0040815C "\nAddress of iNumber =
0x%p", argument to printf()
[ESP+0x1C] 0012FF30 0012FF3C piNumber is stored here,
argument to printf()
[ESP+0x20] 0012FF34 00408140 "\nAddress of iNumber =
0x%p", argument to printf()
[ESP+0x24] 0012FF38 0012FF3C &iNumber is stored here,
argument to printf()
[ESP+0x28] 0012FF3C 00000003 iNumber is stored here
[ESP+0x2C] 0012FF40 0012FF3C &iNumber is stored here
[ESP+0x30] 0012FF44 00401272 return to 0x00401272 from
0x401000
```



```
00401041  .  push ecx
00401042  .  push pointers-optimized.408194
00401047  .  call <pointers-optimized.sub_401081>
0040104C  .  mov edx,dword ptr ss:[esp+20]
00401050  .  push edx
00401051  .  push pointers-optimized.4081AC
00401056  .  call <pointers-optimized.sub_401081>
0040105B  .  mov eax,dword ptr ss:[esp+28]
0040105F  .  push eax
00401060  .  push pointers-optimized.4081C4
00401065  .  call <pointers-optimized.sub_401081>
0040106A  .  mov ecx,dword ptr ss:[esp+34]
```

```
EAX  00000015
EBX  7FFD6000
ECX  00401114     pointers-optimized.00401114
EDX  771A70B4     <ntdll.KiFastSystemCallRet>
EBP  0012FF88
ESP  0012FF14     &"\nValue of iNumber = %d"
ESI  00000000
EDI  00000000

EIP  0040105B     pointers-optimized.0040105B
```

```
0012FF14  004081AC  "\nValue of iNumber = %d"
0012FF18  00000003
0012FF1C  00408194  "\nValue of piNumber = %p"
0012FF20  0012FF3C
0012FF24  00408178  "\nAddress of piNumber = 0x%p"
0012FF28  0012FF40
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.0
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40
Value of piNumber = 0012FF3C
Value of iNumber = 3_
```

Figure 8.30: *After fifth printf execution*

▼Line 64-68

```
; Line 17
mov eax, DWORD PTR _iNumber$[esp+48]
push eax
push OFFSET $SG4684
call _printf
```

The C/C++ code on line 17 prints which itself is the value of

```
printf ("\nValue of iNumber = %d", *(&iNumber)) ;
```

In our assembly code, the **MOV** instruction will be evaluated to:

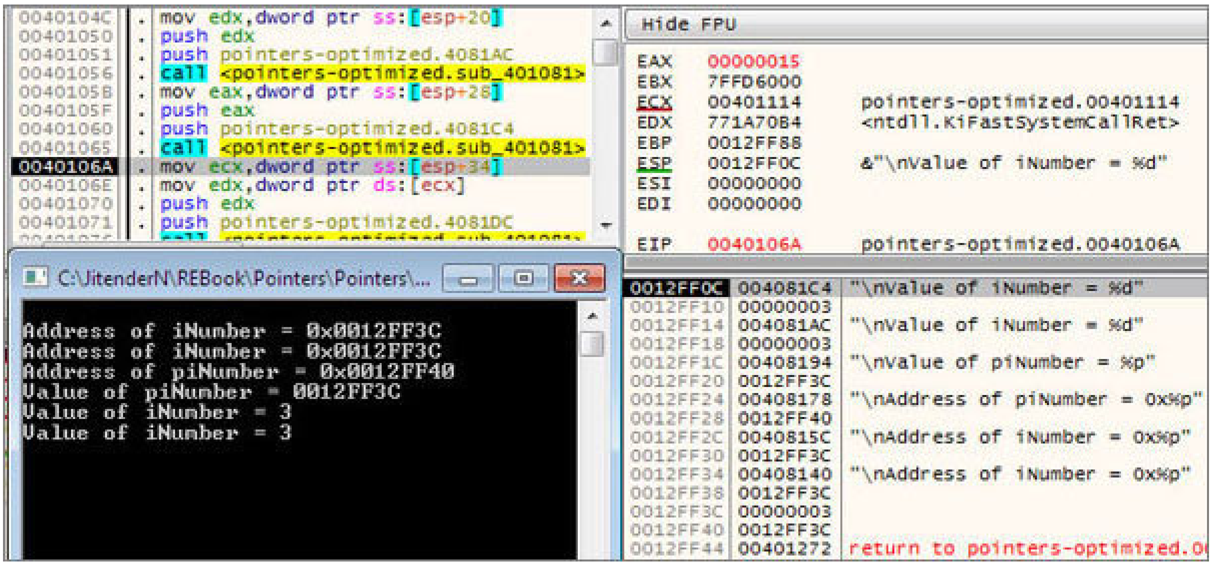
```
mov eax, dword ptr ss:[esp+0x28]
```

MOV will move the value at **ss:[esp+0x28]** to EAX, which is the value of Now, the arguments are again pushed to the stack for call to the **printf** function. As shown in the stack state and the following screenshot:

[ESP] 0012FF0C 004081C4 "\nValue of iNumber = %d", argument to printf()
[ESP+0x04] 0012FF10 00000003 *(&iNumber) is stored here, argument to printf()
[ESP+0x08] 0012FF14 004081AC "\nValue of iNumber = %d", argument to printf()
[ESP+0x0C] 0012FF18 00000003 iNumber is stored here, argument to printf()
[ESP+0x10] 0012FF1C 00408194 "\nValue of piNumber = %p", argument to printf()
[ESP+0x14] 0012FF20 0012FF3C piNumber is stored here, argument to printf()
[ESP+0x18] 0012FF24 00408178 "\nAddress of piNumber = 0x%p", argument to printf()

[ESP+0x1C] 0012FF28 0012FF40 &piNumber is stored here, argument to printf()
[ESP+0x20] 0012FF2C 0040815C "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x24] 0012FF30 0012FF3C piNumber is stored here, argument to printf()
[ESP+0x28] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x2C] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()
[ESP+0x30] 0012FF3C 00000003 iNumber is stored here
[ESP+0x34] 0012FF40 0012FF3C &iNumber is stored here

[ESP+0x38] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
0040104C  .  mov edx,dword ptr ss:[esp+20]
00401050  .  push edx
00401051  .  push pointers-optimized.4081AC
00401056  .  call <pointers-optimized.sub_401081>
0040105B  .  mov eax,dword ptr ss:[esp+28]
0040105F  .  push eax
00401060  .  push pointers-optimized.4081C4
00401065  .  call <pointers-optimized.sub_401081>
0040106A  .  mov ecx,dword ptr ss:[esp+34]
0040106E  .  mov edx,dword ptr ds:[ecx]
00401070  .  push edx
00401071  .  push pointers-optimized.4081DC
```

```
EAX   00000015
EBX   7FFD6000
ECX   00401114     pointers-optimized.00401114
EDX   771A70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF0C     &"\nValue of iNumber = %d"
ESI   00000000
EDI   00000000

EIP   0040106A     pointers-optimized.0040106A
```

```
0012FF0C  004081C4  "\nValue of iNumber = %d"
0012FF10  00000003
0012FF14  004081AC  "\nValue of iNumber = %d"
0012FF18  00000003
0012FF1C  00408194  "\nValue of piNumber = %p"
0012FF20  0012FF3C
0012FF24  00408178  "\nAddress of piNumber = 0x%p"
0012FF28  0012FF40
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.0
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
```

Figure 8.31: After sixth printf execution

▼Line 69-74

```
; Line 18
mov ecx, DWORD PTR _piNumber$[esp+56]
mov edx, DWORD PTR [ecx]
push edx

push OFFSET $SG4685
call _printf
```

The C/C++ code on line 17 prints which itself is the value of

```
printf ("\nValue of iNumber = %d", *piNumber) ;
```

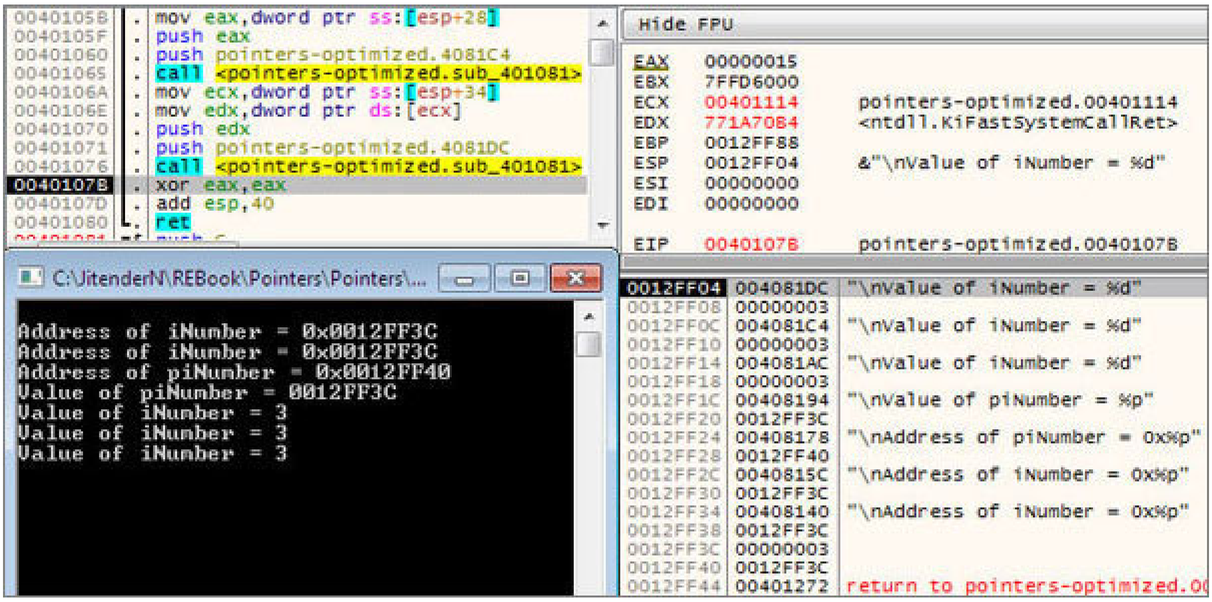
In our assembly code, we see two **MOV** instructions, where one move instruction takes the memory location of **iNumber** and the other fetches the value stored at that memory address. The first **MOV** instruction is resolved to:

```
mov ecx, dword ptr ss:[esp+0x34]
```

This **MOV** will move the value at **ss:[esp+0x34]** to ECX, which is the memory location of The second **MOV** fetches the value stored at the memory address in ECX and moves it to EDX. Now, the arguments are again pushed to the stack for call to the **printf** function. This is shown in the following screenshot:

```
[ESP] 0012FF04 004081DC "\nValue of iNumber = %d",
argument to printf()
[ESP+0x04] 0012FF08 00000003 *piNumber is stored here,
argument to printf()
[ESP+0x08] 0012FF0C 004081C4 "\nValue of iNumber = %d",
argument to printf()
[ESP+0x0C] 0012FF10 00000003 *(&iNumber) is stored here,
argument to printf()
[ESP+0x10] 0012FF14 004081AC "\nValue of iNumber = %d",
argument to printf()
[ESP+0x14] 0012FF18 00000003 iNumber is stored here,
argument to printf()
[ESP+0x18] 0012FF1C 00408194 "\nValue of piNumber = %p",
argument to printf()


[ESP+0x1C] 0012FF20 0012FF3C piNumber is stored here,
argument to printf()
```

[ESP+0x20] 0012FF24 00408178 "\nAddress of piNumber = 0x%p", argument to printf()
[ESP+0x24] 0012FF28 0012FF40 &piNumber is stored here, argument to printf()
[ESP+0x28] 0012FF2C 0040815C "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x2C] 0012FF30 0012FF3C piNumber is stored here, argument to printf()
[ESP+0x30] 0012FF34 00408140 "\nAddress of iNumber = 0x%p", argument to printf()
[ESP+0x34] 0012FF38 0012FF3C &iNumber is stored here, argument to printf()
[ESP+0x38] 0012FF3C 00000003 iNumber is stored here
[ESP+0x3C] 0012FF40 0012FF3C &iNumber is stored here
[ESP+0x40] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
0040105B  .  mov eax,dword ptr ss:[esp+28]
0040105F  .  push eax
00401060  .  push pointers-optimized.4081C4
00401065  .  call <pointers-optimized.sub_401081>
0040106A  .  mov ecx,dword ptr ss:[esp+34]
0040106E  .  mov edx,dword ptr ds:[ecx]
00401070  .  push edx
00401071  .  push pointers-optimized.4081DC
00401076  .  call <pointers-optimized.sub_401081>
0040107B  .  xor eax,eax
0040107D  .  add esp,40
00401080  .  ret
```

```
EAX 00000015
EBX 7FFD6000
ECX 00401114   pointers-optimized.00401114
EDX 771A70B4   <ntdll.KiFastSystemCallRet>
EBP 0012FF88
ESP 0012FF04   &"\nValue of iNumber = %d"
ESI 00000000
EDI 00000000

EIP 0040107B   pointers-optimized.0040107B
```

```
0012FF04  004081DC  "\nValue of iNumber = %d"
0012FF08  00000003
0012FF0C  004081C4  "\nValue of iNumber = %d"
0012FF10  00000003
0012FF14  004081AC  "\nValue of iNumber = %d"
0012FF18  00000003
0012FF1C  00408194  "\nValue of piNumber = %p"
0012FF20  0012FF3C
0012FF24  00408178  "\nAddress of piNumber = 0x%p"
0012FF28  0012FF40
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.00
```

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
Value of iNumber = 3
```

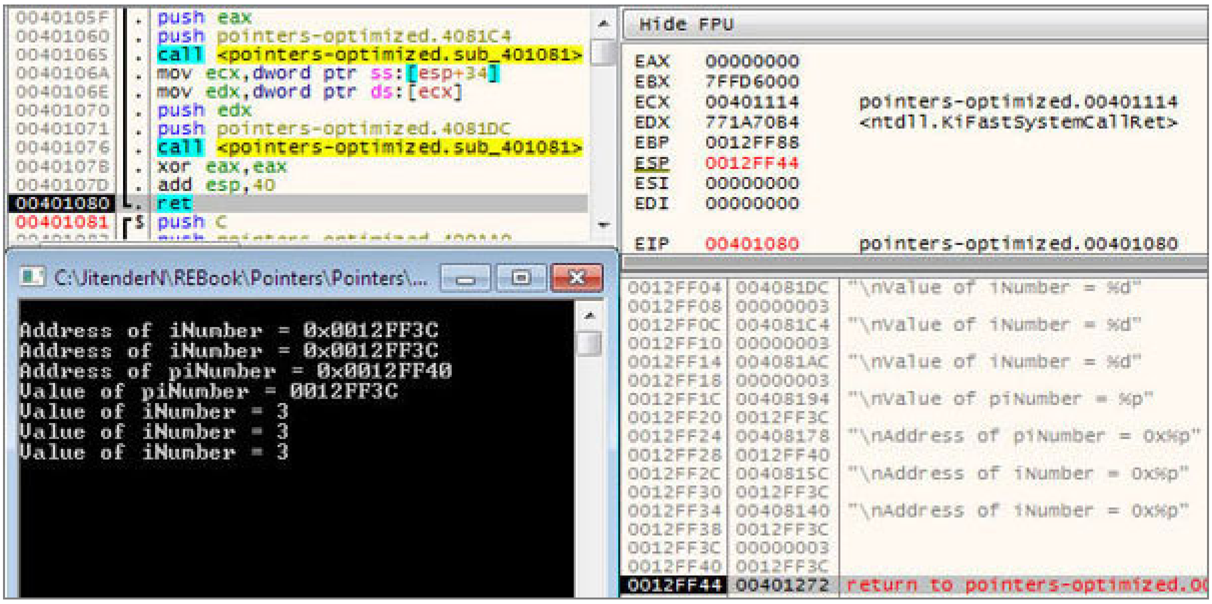
Figure 8.32: *After seventh printf execution*

▼Line 75-81

```
; Line 19
xor eax, eax
add esp, 64          ; 00000040H
ret 0
_main ENDP
_TEXT ENDS
END
```

XOR and **ADD** will clean up the EAX and stack, respectively and END the code by returning 0. The **ADD** instruction cleaned up the stack in one go, which is different from that in the non-optimized code, wherein we cleaned the stack after each printf call. This is shown in the stack state below and the following screenshot:

[ESP] 0012FF44 00401272 return to 0x00401272 from 0x401000



```
0040105F  .  push eax
00401060  .  push pointers-optimized.4081C4
00401065  .  call <pointers-optimized.sub_401081>
0040106A  .  mov ecx,dword ptr ss:[esp+34]
0040106E  .  mov edx,dword ptr ds:[ecx]
00401070  .  push edx
00401071  .  push pointers-optimized.4081DC
00401076  .  call <pointers-optimized.sub_401081>
0040107B  .  xor eax,eax
0040107D  .  add esp,40
00401080  L. ret
00401081  r$ push C
```

```
EAX   00000000
EBX   7FFD6000
ECX   00401114     pointers-optimized.00401114
EDX   771A7084     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   00401080     pointers-optimized.00401080
```

```
0012FF04  004081DC  "\nValue of iNumber = %d"
0012FF08  00000003
0012FF0C  004081C4  "\nValue of iNumber = %d"
0012FF10  00000003
0012FF14  004081AC  "\nValue of iNumber = %d"
0012FF18  00000003
0012FF1C  00408194  "\nValue of piNumber = %p"
0012FF20  0012FF3C
0012FF24  00408178  "\nAddress of piNumber = 0x%p"
0012FF28  0012FF40
0012FF2C  0040815C  "\nAddress of iNumber = 0x%p"
0012FF30  0012FF3C
0012FF34  00408140  "\nAddress of iNumber = 0x%p"
0012FF38  0012FF3C
0012FF3C  00000003
0012FF40  0012FF3C
0012FF44  00401272  return to pointers-optimized.0
```

C:\JitenderN\REBook\Pointers\Pointers\...

```
Address of iNumber = 0x0012FF3C
Address of iNumber = 0x0012FF3C
Address of piNumber = 0x0012FF40
Value of piNumber = 0012FF3C
Value of iNumber = 3
Value of iNumber = 3
Value of iNumber = 3
```

Figure 8.33: *Cleaning stack*

Conclusion

In this chapter, we learned the conceptual knowledge about pointers. We also learned how pointers are stored in memory with respect to integer, float, and char pointers. We also found out how non-optimized code is different from optimized assembly code. The non-optimized code uses EBP to refer to arguments on the stack and the stack cleaning is done after every printf call. But in case of optimized code, ESP is used to refer to the arguments and the stack cleaning is done in one go towards the END of the code. In the next chapter, we will talk about another interesting topic, which is decision. We will see how decisions statements are handled in assembly programming.

CHAPTER 9

Reverse Engineering Pattern of Decision Control Structure

In our day-to-day life, we take many decisions and every decision is related to an outcome. Let's take an example. Before leaving home for office, we all prefer to check Google maps and if the traffic is high, we may ask our boss for permission to work from home. Another example is if I earn more, I will purchase a nice luxury car and a big house. All these decisions are linked with some condition and result in certain outcomes.

Similar to this, computers are also programmed to make decisions based on conditions. C/C++ programming also provides decision-making statements, which help programmers to write code that involves decisions based on the conditions. Different conditions are linked to the outcomes to behave in the natural manner. This chapter will help you understand the pattern of these conditions in assembling listing from the reverse engineering point of view.

Structure

In this chapter, we will cover the following topics:

If-else statement

If-else statement without Optimization

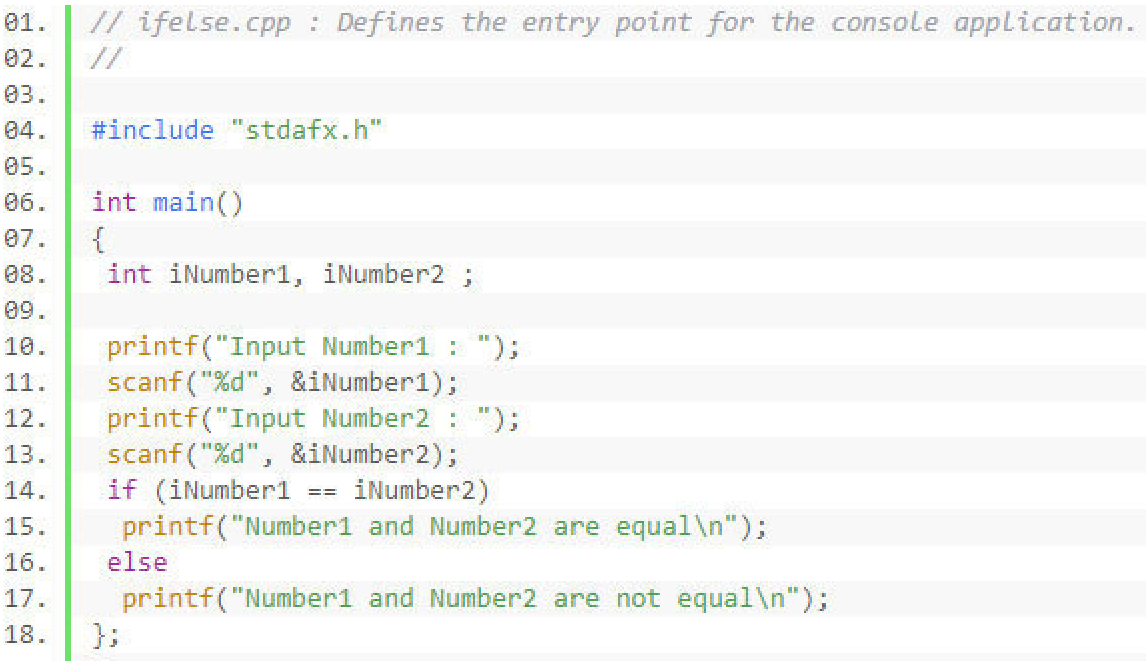
If-else statement with Optimization

Objective

The objective of this chapter is to understand the concept of assembly instructions used to make decisions in a program flow. We will learn about CMP and Jump instructions in the assembly code. As we tend to take several decisions for a single problem in our day-to-day life, similarly to handle problems or conditions in assembly we have CMP and jump instructions in assembly. We will also understand the differentiation between optimized and non-optimized assembly code of decision control structures.

If-else statement

In this section, we will cover simple C/C++ code with if-else statement, wherein we will ask the user to enter two integer numbers. If both the numbers are equal, it prints saying that both the numbers are equal on the console. But if both the entered numbers are not equal, the code will use else statement to print, the numbers are not equal. This code also includes **scanf** and **printf** in our C/C++ code. It is recommended to go through *Chapter 7, Reverse Engineering Pattern of Printf* for a clear understanding.



```
// ifelse.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int main()
{
 int iNumber1, iNumber2 ;

 printf("Input Number1 : ");
 scanf("%d", &iNumber1);
 printf("Input Number2 : ");
 scanf("%d", &iNumber2);
 if (iNumber1 == iNumber2)
  printf("Number1 and Number2 are equal\n");
 else
  printf("Number1 and Number2 are not equal\n");
};
```

Figure 9.1: *ifelse.cpp*

If-else statement without Optimization

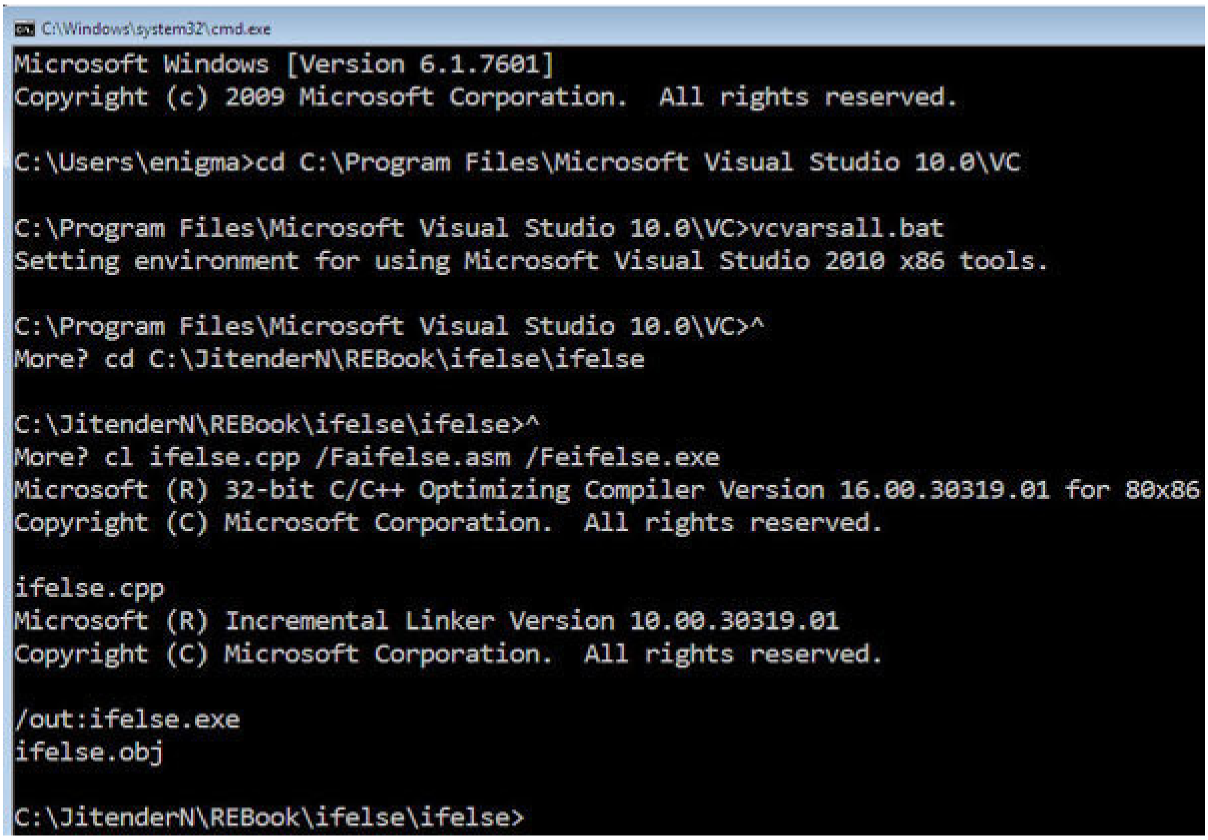
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\ifelse\ifelse

C:\JitenderN\REBook\ifelse\ifelse>^
More? cl ifelse.cpp /Faifelse.asm /Feifelse.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ifelse.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ifelse.exe
ifelse.obj

C:\JitenderN\REBook\ifelse\ifelse>
```

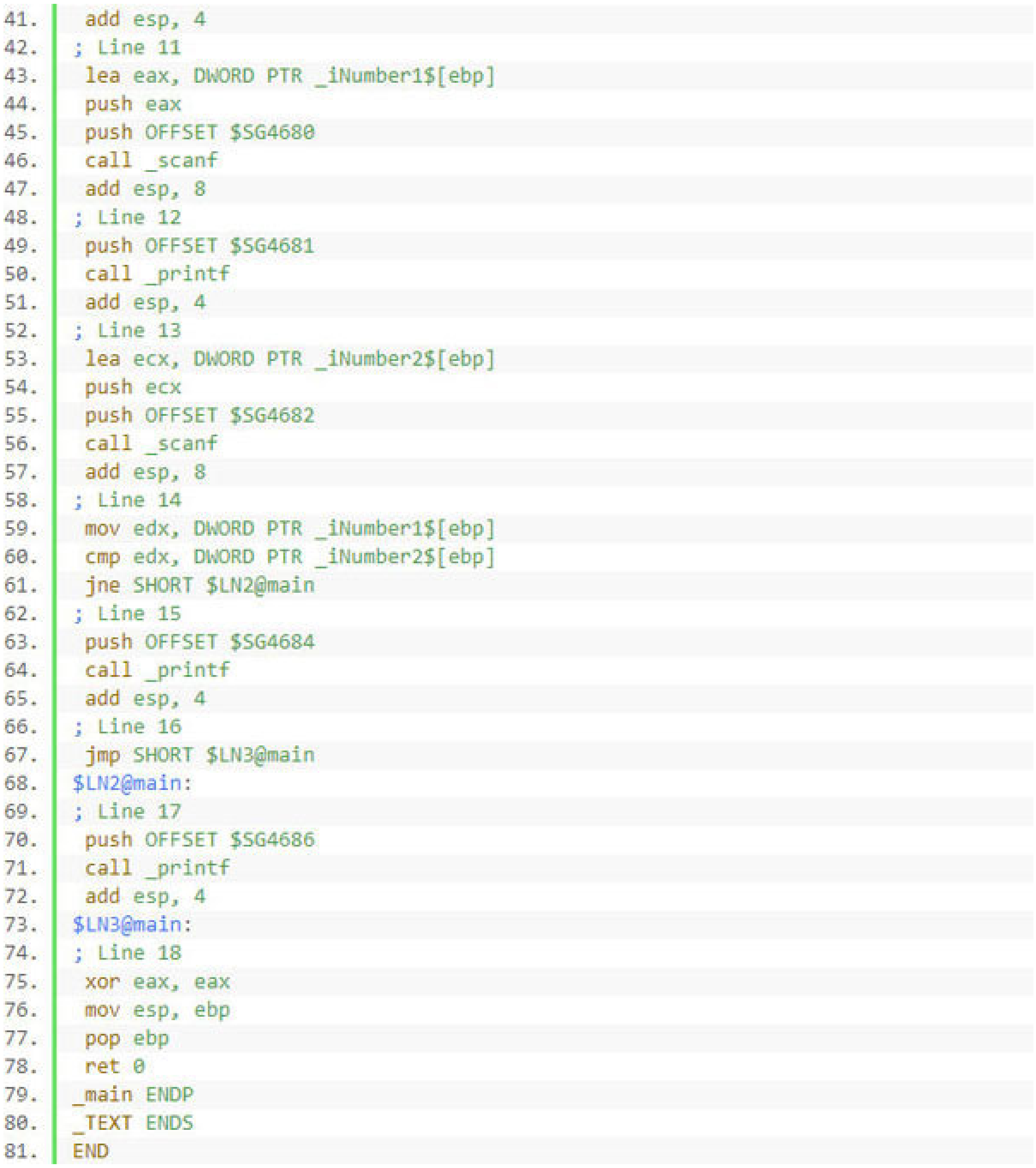
Figure 9.2: *If-else statement without Optimization*

This will generate the assembly code and the EXE file. For analysis purpose, we will disable the **Address Space Layout Randomization** It is a security mechanism by which the base address of the PE file is randomized on every load of the **Portable Executable** file generated with our MSVC compiler. To disable ASLR, follow the same steps by using the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can** For more details, refer to the Appendix.

Now, let us move on to the generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\ifelse\ifelse\ifelse.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4679 DB 'Input Number1 : ', 00H
 ORG $+3
$SG4680 DB '%d', 00H
 ORG $+1
$SG4681 DB 'Input Number2 : ', 00H
 ORG $+3
$SG4682 DB '%d', 00H
 ORG $+1
$SG4684 DB 'Number1 and Number2 are equal', 0aH, 00H
 ORG $+1
$SG4686 DB 'Number1 and Number2 are not equal', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_iNumber1$ = -8      ; size = 4
_iNumber2$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\ifelse\ifelse\ifelse.cpp
; Line 7
 push ebp
 mov ebp, esp
 sub esp, 8
; Line 10
 push OFFSET $SG4679
 call _printf
```

Figure 9.3: *ifelse.asm-Part-1*



```
41.   add esp, 4
42.  ; Line 11
43.   lea eax, DWORD PTR _iNumber1$[ebp]
44.   push eax
45.   push OFFSET $SG4680
46.   call _scanf
47.   add esp, 8
48.  ; Line 12
49.   push OFFSET $SG4681
50.   call _printf
51.   add esp, 4
52.  ; Line 13
53.   lea ecx, DWORD PTR _iNumber2$[ebp]
54.   push ecx
55.   push OFFSET $SG4682
56.   call _scanf
57.   add esp, 8
58.  ; Line 14
59.   mov edx, DWORD PTR _iNumber1$[ebp]
60.   cmp edx, DWORD PTR _iNumber2$[ebp]
61.   jne SHORT $LN2@main
62.  ; Line 15
63.   push OFFSET $SG4684
64.   call _printf
65.   add esp, 4
66.  ; Line 16
67.   jmp SHORT $LN3@main
68.  $LN2@main:
69.  ; Line 17
70.   push OFFSET $SG4686
71.   call _printf
72.   add esp, 4
73.  $LN3@main:
74.  ; Line 18
75.   xor eax, eax
76.   mov esp, ebp
77.   pop ebp
78.   ret 0
79.  _main ENDP
80.  _TEXT ENDS
81.  END
```

Figure 9.4: *ifelse.asm-Part-2*

Let's walk through the assembly code line by line. As we have discussed most of the initial assembly listing in the earlier chapters, we will move on to the instructions from the start of the **main** procedure.

▼Line 34-37

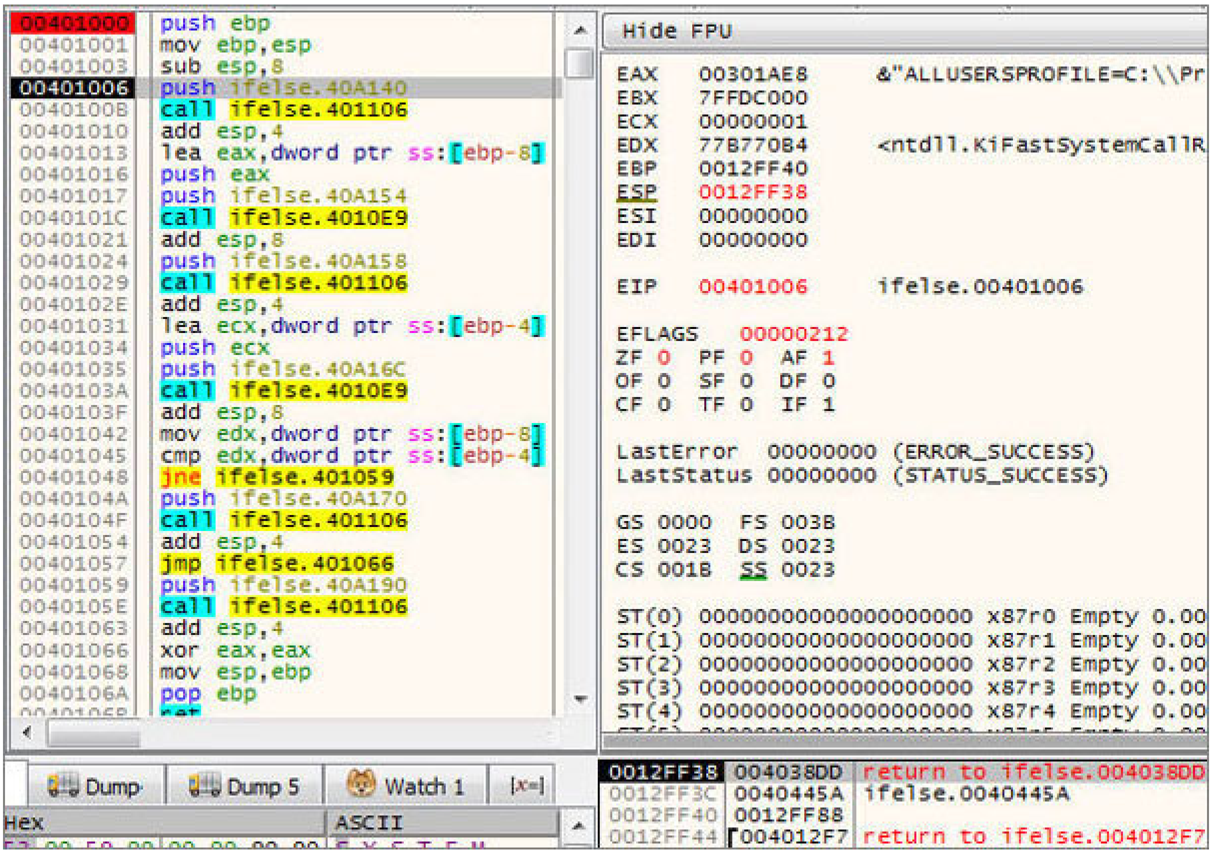
```
; Line 7
push ebp

mov ebp, esp
sub esp, 8
```

It starts with the **main** function prologue. With the **SUB** instruction, we are creating room for the local variables of which are **iNumber1** and Each number will occupy 4 bytes of space, making it a total of 8 bytes.

To analyze the stack state, let us open the PE file generated in x32dbg and place a breakpoint at the start of the **main** procedure. We will step into the code to see the stack state after the **SUB** instruction.

```
[ESP]  0012FF38  004038DD  [EBP-8] now JUNK, Input placeholder for iNumber1
[ESP+0x4] 0012FF3C  0040445A  [EBP-4] now JUNK, Input placeholder for iNumber2
[ESP+0x8] 0012FF40  0012FF88  [EBP]
[ESP+0xC] 0012FF44  004012F7  return to ifelse.004012F7 from ifelse.00401000
```

iNumber1 can be accessed with the help of the **_iNumber1$** macro, which is equal to -8. So, **iNumber1** can be accessed by adding **EBP** with the **_iNumber1$** macro, which is equal to **[EBP-8].**



```
00401000  push ebp
00401001  mov ebp,esp
00401003  sub esp,8
00401006  push ifelse.40A140
0040100B  call ifelse.401106
00401010  add esp,4
00401013  lea eax,dword ptr ss:[ebp-8]
00401016  push eax
00401017  push ifelse.40A154
0040101C  call ifelse.4010E9
00401021  add esp,8
00401024  push ifelse.40A158
00401029  call ifelse.401106
0040102E  add esp,4
00401031  lea ecx,dword ptr ss:[ebp-4]
00401034  push ecx
00401035  push ifelse.40A16C
0040103A  call ifelse.4010E9
0040103F  add esp,8
00401042  mov edx,dword ptr ss:[ebp-8]
00401045  cmp edx,dword ptr ss:[ebp-4]
00401048  jne ifelse.401059
0040104A  push ifelse.40A170
0040104F  call ifelse.401106
00401054  add esp,4
00401057  jmp ifelse.401066
00401059  push ifelse.40A190
0040105E  call ifelse.401106
00401063  add esp,4
00401066  xor eax,eax
00401068  mov esp,ebp
0040106A  pop ebp
```

```
EAX   00301AE8     &"ALLUSERSPROFILE=C:\\Pr
EBX   7FFDC000
ECX   00000001
EDX   77B770B4     <ntdll.KiFastSystemCallR
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   00401006     ifelse.00401006

EFLAGS    00000212
ZF 0  PF 0  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000000 x87r0 Empty 0.00
ST(1) 0000000000000000000000 x87r1 Empty 0.00
ST(2) 0000000000000000000000 x87r2 Empty 0.00
ST(3) 0000000000000000000000 x87r3 Empty 0.00
ST(4) 0000000000000000000000 x87r4 Empty 0.00
```

```
0012FF38  004038DD  return to ifelse.004038DD
0012FF3C  0040445A  ifelse.0040445A
0012FF40  0012FF88
0012FF44  004012F7  return to ifelse.004012F7
```

Figure 9.5: Creating room for our local variable

▼Line 38-41

```
; Line 10
push OFFSET $SG4679
call _printf
add esp, 4
```

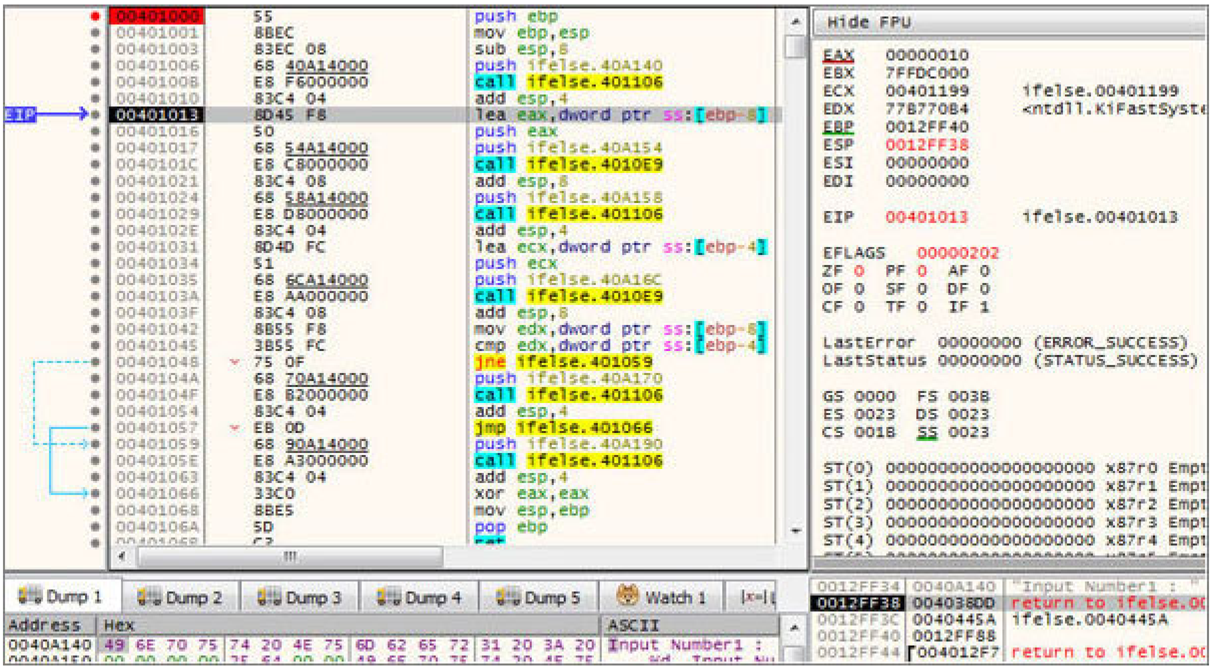
The C/C++ code on line 10 prints the string on the console:

```
printf("Input Number1 : ");
```

In our assembly code, the argument to the **printf** function is first pushed onto the stack. The argument of **printf** is a string constant stored in the **.rdata** segment and internally defined as A call to **printf** will print **$SG4679** on the console and the **ADD** instruction after call to **printf** is cleaning the stack by 4 bytes.

[ESP-0x4] 0012FF34 0040A140 "Input Number1 : ", parameter to 1st printf()

[ESP] 0012FF38 004038DD [EBP-8] now JUNK, Input placeholder for iNumber1
[ESP+0x4] 0012FF3C 0040445A [EBP-4] now JUNK, Input placeholder for iNumber2
[ESP+0x8] 0012FF40 0012FF88 [EBP]
[ESP+0xC] 0012FF44 004012F7 return to ifelse.004012F7 from ifelse.00401000



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   83EC 08         sub esp,8
00401006   68 40A14000     push ifelse.40A140
0040100B   E8 F6000000     call ifelse.401106
00401010   83C4 04         add esp,4
00401013   8D45 F8         lea eax,dword ptr ss:[ebp-8]
00401016   50              push eax
00401017   68 54A14000     push ifelse.40A154
0040101C   E8 C8000000     call ifelse.4010E9
00401021   83C4 08         add esp,8
00401024   68 58A14000     push ifelse.40A158
00401029   E8 D8000000     call ifelse.401106
0040102E   83C4 04         add esp,4
00401031   8D4D FC         lea ecx,dword ptr ss:[ebp-4]
00401034   51              push ecx
00401035   68 6CA14000     push ifelse.40A16C
0040103A   E8 AA000000     call ifelse.4010E9
0040103F   83C4 08         add esp,8
00401042   8B55 F8         mov edx,dword ptr ss:[ebp-8]
00401045   3B55 FC         cmp edx,dword ptr ss:[ebp-4]
00401048   75 0F           jne ifelse.401059
0040104A   68 70A14000     push ifelse.40A170
0040104F   E8 B2000000     call ifelse.401106
00401054   83C4 04         add esp,4
00401057   EB 0D           jmp ifelse.401066
00401059   68 90A14000     push ifelse.40A190
0040105E   E8 A3000000     call ifelse.401106
00401063   83C4 04         add esp,4
00401066   33C0            xor eax,eax
00401068   8BE5            mov esp,ebp
0040106A   5D              pop ebp
```

```
EAX    00000010
EBX    7FFDC000
ECX    00401199     ifelse.00401199
EDX    77B770B4     <ntdll.KiFastSyste
EBP    0012FF40
ESP    0012FF38
ESI    00000000
EDI    00000000

EIP    00401013     ifelse.00401013

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000000 x87r0 Empt
ST(1) 0000000000000000000000 x87r1 Empt
ST(2) 0000000000000000000000 x87r2 Empt
ST(3) 0000000000000000000000 x87r3 Empt
ST(4) 0000000000000000000000 x87r4 Empt
```

```
Address   Hex                                               ASCII
0040A140  49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 : 
```

```
0012FF34  0040A140  "Input Number1 : "
0012FF38  004038DD  return to ifelse.00
0012FF3C  0040445A  ifelse.0040445A
0012FF40  0012FF88
0012FF44  004012F7  return to ifelse.00
```

Figure 9.6: *Call to printf*

▼Line 42-47

```
; Line 11
lea eax, DWORD PTR _iNumber1$[ebp]
push eax
push OFFSET $SG4680
call _scanf
add esp, 8
```

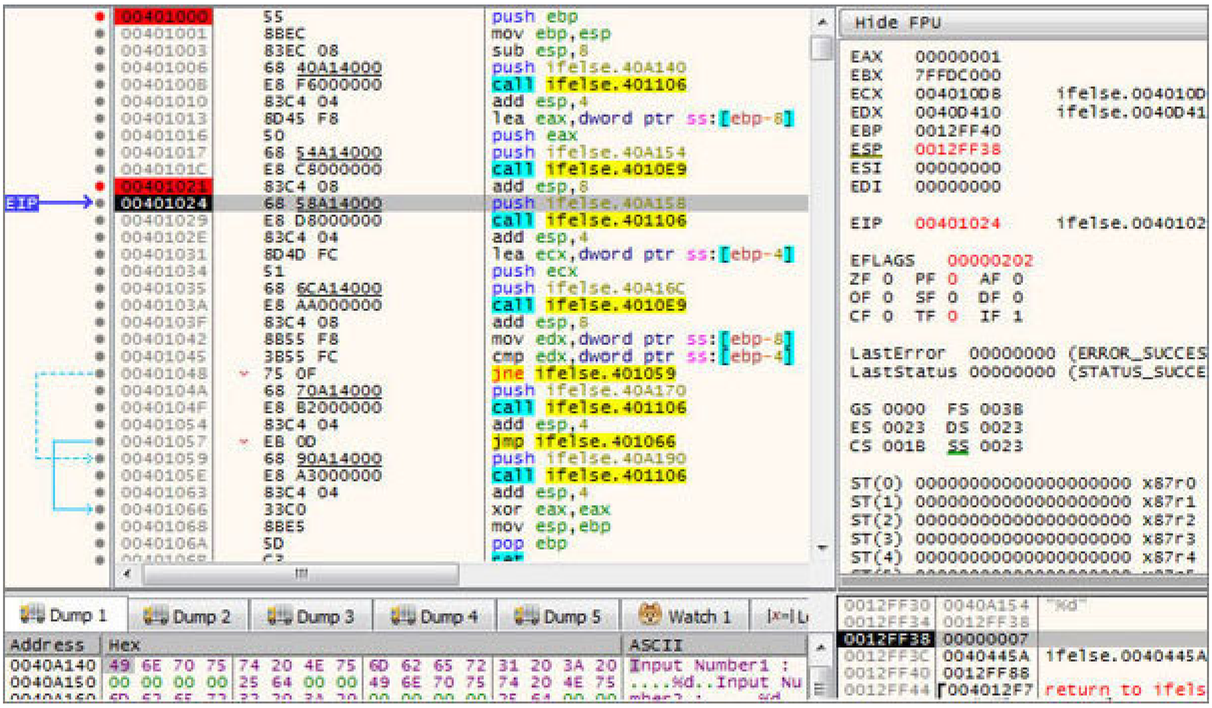
The C/C++ code on line 11 takes the input integer from the user and stores it at the **&iNumber1** memory location:

```
scanf("%d", &iNumber1);
```

In our assembly code, LEA will effectively load the address of which is into the **EAX** register. The purpose of loading **EAX** with the address of **_iNumber1$[ebp]** is to push the **EAX** register on the stack along with the string constant Both the address and string **$SG4680** are arguments to the **scanf** function.

To check the stack state after the **scanf** function call, put another breakpoint at **ADD** instruction after the **scanf** call. Now step into the instructions one by one and on call to do a step over. It will prompt you to enter the first number on the console. We will enter number 7 and press After pressing number 7 will be saved onto the stack at the input placeholder for the **iNumber1** memory location, With the **scanf** call, the number input by the user is saved onto the stack. After the call, it is time to again clean up the stack with the **ADD** instruction. **ADD** will move the stack pointer by 8 bytes for cleaning. The stack state is shown in the following screenshot:

[ESP-0x8] 0012FF30 0040A154 "%d", parameter to 1st scanf()
[ESP-0x4] 0012FF34 0012FF38 Memory location of placeholder for iNumber1
[ESP] 0012FF38 00000007 [EBP-8] 7 is stored, Input placeholder for iNumber1
[ESP+0xC] 0012FF3C 0040445A [EBP-4] now JUNK, Input placeholder for iNumber2
[ESP+0x10] 0012FF40 0012FF88 [EBP]
[ESP+0x14] 0012FF44 004012F7 return to ifelse.004012F7 from ifelse.00401000



```
EIP 00401000  55              push ebp
    00401001  8BEC            mov ebp,esp
    00401003  83EC 08         sub esp,8
    00401006  68 40A14000     push ifelse.40A140
    0040100B  E8 F6000000     call ifelse.401106
    00401010  83C4 04         add esp,4
    00401013  8D45 F8         lea eax,dword ptr ss:[ebp-8]
    00401016  50              push eax
    00401017  68 54A14000     push ifelse.40A154
    0040101C  E8 C8000000     call ifelse.4010E9
    00401021  83C4 08         add esp,8
    00401024  68 58A14000     push ifelse.40A158
    00401029  E8 D8000000     call ifelse.401106
    0040102E  83C4 04         add esp,4
    00401031  8D4D FC         lea ecx,dword ptr ss:[ebp-4]
    00401034  51              push ecx
    00401035  68 6CA14000     push ifelse.40A16C
    0040103A  E8 AA000000     call ifelse.4010E9
    0040103F  83C4 08         add esp,8
    00401042  8B55 F8         mov edx,dword ptr ss:[ebp-8]
    00401045  3B55 FC         cmp edx,dword ptr ss:[ebp-4]
    00401048  75 0F           jne ifelse.401059
    0040104A  68 70A14000     push ifelse.40A170
    0040104F  E8 B2000000     call ifelse.401106
    00401054  83C4 04         add esp,4
    00401057  EB 0D           jmp ifelse.401066
    00401059  68 90A14000     push ifelse.40A190
    0040105E  E8 A3000000     call ifelse.401106
    00401063  83C4 04         add esp,4
    00401066  33C0            xor eax,eax
    00401068  8BE5            mov esp,ebp
    0040106A  5D              pop ebp
```

```
Hide FPU

EAX   00000001
EBX   7FFDC000
ECX   004010D8      ifelse.004010D
EDX   0040D410      ifelse.0040D41
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   00401024      ifelse.0040102

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCES
LastStatus 00000000 (STATUS_SUCCE

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0
ST(1) 00000000000000000000 x87r1
ST(2) 00000000000000000000 x87r2
ST(3) 00000000000000000000 x87r3
ST(4) 00000000000000000000 x87r4
```

```
Address   Hex                                               ASCII
0040A140  49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
0040A150  00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75  ....%d..Input Nu
```

```
0012FF30  0040A154  "%d"
0012FF34  0012FF38
0012FF38  00000007
0012FF3C  0040445A  ifelse.0040445A
0012FF40  0012FF88
0012FF44  004012F7  return to ifels
```

Figure 9.7: After scanf

▼**Line 48-51**

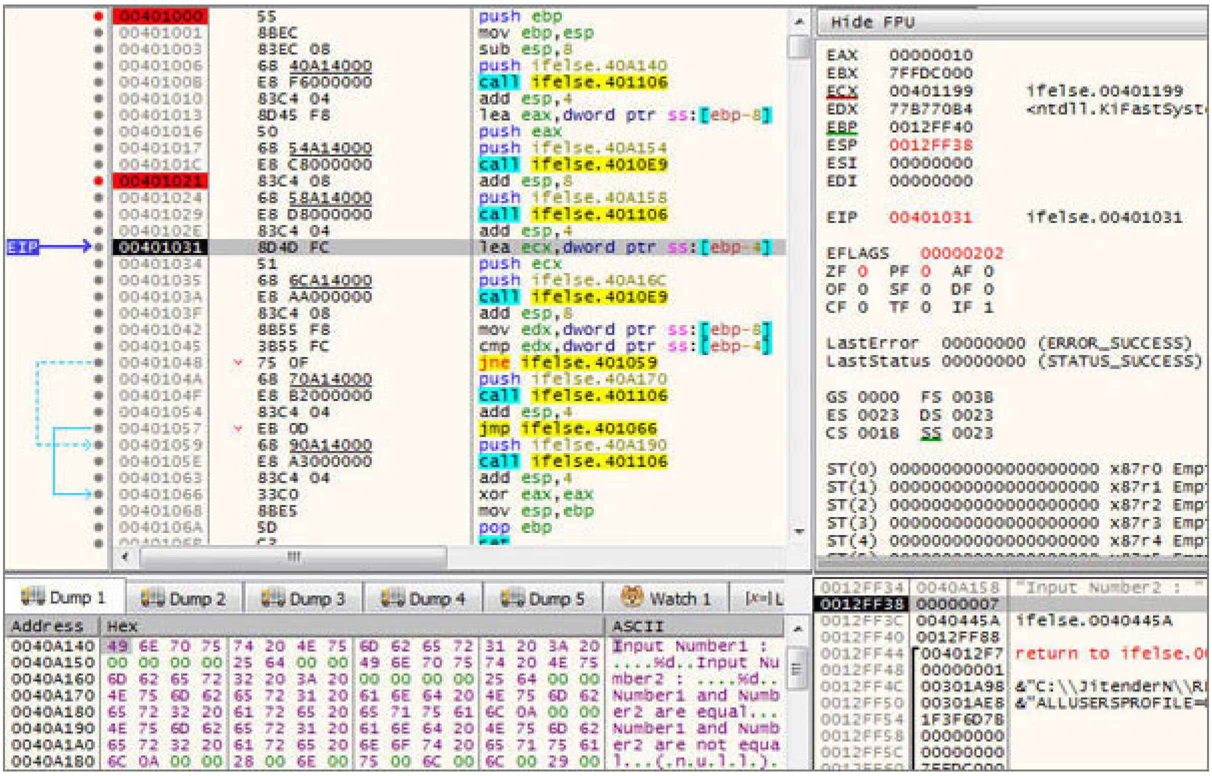
```
; Line 12
push OFFSET $SG4681
call _printf
add esp, 4
```

The C/C++ code on line 12 prints the string on the console:

```
printf("Input Number2 : ");
```

It is the same as the earlier. The argument to the **printf** function is first pushed onto the stack. The argument of **printf** is a string constant stored in the **.rdata** segment and internally defined as A call to **printf** will print **$SG4681** on the console and the **ADD** instruction following the call to **printf** will clean the stack by 4 bytes. The stack state is shown in the following screenshot:

```
[ESP-0x4] 0012FF34  0040A158  "Input Number2 : ", parameter to 2nd printf()

[ESP]  0012FF38  00000007  [EBP-8] 7 is stored, Input placeholder for iNumber1
[ESP+0xC] 0012FF3C  0040445A  [EBP-4] now JUNK, Input placeholder for iNumber2
[ESP+0x10] 0012FF40  0012FF88  [EBP]
[ESP+0x14] 0012FF44  004012F7  return to ifelse.004012F7 from ifelse.00401000
```



```
00401000   55             push ebp
00401001   8BEC           mov ebp,esp
00401003   83EC 08        sub esp,8
00401006   68 40A14000    push ifelse.40A140
0040100B   E8 F6000000    call ifelse.401106
00401010   83C4 04        add esp,4
00401013   8D45 F8        lea eax,dword ptr ss:[ebp-8]
00401016   50             push eax
00401017   68 54A14000    push ifelse.40A154
0040101C   E8 C8000000    call ifelse.4010E9
00401021   83C4 08        add esp,8
00401024   68 58A14000    push ifelse.40A158
00401029   E8 D8000000    call ifelse.401106
0040102E   83C4 04        add esp,4
00401031   8D4D FC        lea ecx,dword ptr ss:[ebp-4]
00401034   51             push ecx
00401035   68 6CA14000    push ifelse.40A16C
0040103A   E8 AA000000    call ifelse.4010E9
0040103F   83C4 08        add esp,8
00401042   8B55 F8        mov edx,dword ptr ss:[ebp-8]
00401045   3B55 FC        cmp edx,dword ptr ss:[ebp-4]
00401048   75 0F          jne ifelse.401059
0040104A   68 70A14000    push ifelse.40A170
0040104F   E8 B2000000    call ifelse.401106
00401054   83C4 04        add esp,4
00401057   EB 0D          jmp ifelse.401066
00401059   68 90A14000    push ifelse.40A190
0040105E   E8 A3000000    call ifelse.401106
00401063   83C4 04        add esp,4
00401066   33C0           xor eax,eax
00401068   8BE5           mov esp,ebp
0040106A   5D             pop ebp
```

```
EAX  00000010
EBX  7FFDC000
ECX  00401199    ifelse.00401199
EDX  77B77084    <ntdll.KiFastSyst
EBP  0012FF40
ESP  0012FF38
ESI  00000000
EDI  00000000

EIP  00401031    ifelse.00401031

EFLAGS  00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000000 x87r0 Emp
ST(1) 0000000000000000000000 x87r1 Emp
ST(2) 0000000000000000000000 x87r2 Emp
ST(3) 0000000000000000000000 x87r3 Emp
ST(4) 0000000000000000000000 x87r4 Emp
```

```
Address   Hex                                              ASCII
0040A140  49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
0040A150  00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75  ....%d..Input Nu
0040A160  6D 62 65 72 32 20 3A 20 00 00 00 00 25 64 00 00  mber2 : ....%d..
0040A170  4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
0040A180  65 72 32 20 61 72 65 20 65 71 75 61 6C 0A 00 00  er2 are equal...
0040A190  4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
0040A1A0  65 72 32 20 61 72 65 20 6E 6F 74 20 65 71 75 61  er2 are not equa
0040A1B0  6C 0A 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  l...(.n.u.l.l.).
```

```
0012FF34  0040A158  "Input Number2 : "
0012FF38  00000007
0012FF3C  0040445A  ifelse.0040445A
0012FF40  0012FF88
0012FF44  004012F7  return to ifelse.0
0012FF48  00000001
0012FF4C  00301A98  &"C:\\JitenderN\\R
0012FF50  00301AE8  &"ALLUSERSPROFILE=
0012FF54  1F3F6D7B
0012FF58  00000000
0012FF5C  00000000
```

Figure 9.8: Again call to printf

▼Line 52-57

```
; Line 13
lea ecx, DWORD PTR _iNumber2$[ebp]
push ecx
push OFFSET $SG4682
call _scanf
add esp, 8
```

The C/C++ code on line 13 takes the input integer from the user and stores it at the **&iNumber2** memory location:

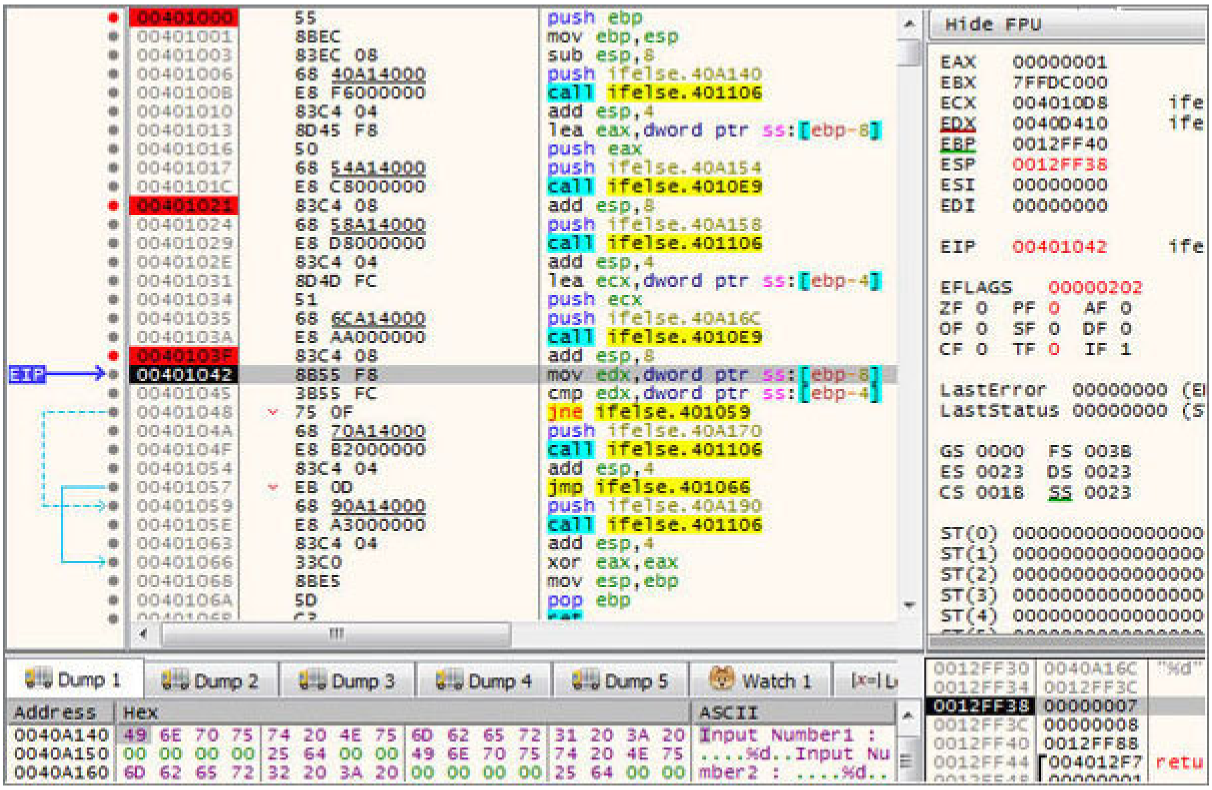
```
scanf("%d", &iNumber2);
```

It is same as the earlier **scanf** code. LEA will effectively load the address of which is into the **ECX** register. The purpose of loading **ECX** with the address of **_iNumber2$[ebp]** is to push the **ECX** register on the stack along with the string constant Both the address and string **$SG4682** are arguments to the **scanf** function.

To check the stack state after the **scanf** function call, put another breakpoint at **ADD** instruction after the **scanf** call. Now step into the instructions one by one and on call to do a step over. It will prompt you to enter the second number on the console. We will enter number 8 and press After pressing number 8 will be saved onto the stack at the input placeholder for the **iNumber2** memory location, With the **scanf** call, the number input by the user is saved onto the stack. After the call, it is time to again clean up the stack with the **ADD** instruction. **ADD** will move the stack pointer by 8 bytes for cleaning. The stack state is shown in the following screenshot:

[ESP-0x8] 0012FF30 0040A16C "%d", parameter to 2nd scanf()
[ESP-0x4] 0012FF34 0012FF3C Memory location of placeholder for iNumber2
[ESP] 0012FF38 00000007 [EBP-8] 7 is stored, Input placeholder for iNumber1
[ESP+0xC] 0012FF3C 00000008 [EBP-4] 8 is stored, Input placeholder for iNumber2
[ESP+0x10] 0012FF40 0012FF88 [EBP]

[ESP+0x14] 0012FF44 004012F7 return to ifelse.004012F7 from ifelse.00401000



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   83EC 08         sub esp,8
00401006   68 40A14000     push ifelse.40A140
0040100B   E8 F6000000     call ifelse.401106
00401010   83C4 04         add esp,4
00401013   8D45 F8         lea eax,dword ptr ss:[ebp-8]
00401016   50              push eax
00401017   68 54A14000     push ifelse.40A154
0040101C   E8 C8000000     call ifelse.4010E9
00401021   83C4 08         add esp,8
00401024   68 58A14000     push ifelse.40A158
00401029   E8 D8000000     call ifelse.401106
0040102E   83C4 04         add esp,4
00401031   8D4D FC         lea ecx,dword ptr ss:[ebp-4]
00401034   51              push ecx
00401035   68 6CA14000     push ifelse.40A16C
0040103A   E8 AA000000     call ifelse.4010E9
0040103F   83C4 08         add esp,8
00401042   8B55 F8         mov edx,dword ptr ss:[ebp-8]
00401045   3B55 FC         cmp edx,dword ptr ss:[ebp-4]
00401048   75 0F           jne ifelse.401059
0040104A   68 70A14000     push ifelse.40A170
0040104F   E8 B2000000     call ifelse.401106
00401054   83C4 04         add esp,4
00401057   EB 0D           jmp ifelse.401066
00401059   68 90A14000     push ifelse.40A190
0040105E   E8 A3000000     call ifelse.401106
00401063   83C4 04         add esp,4
00401066   33C0            xor eax,eax
00401068   8BE5            mov esp,ebp
0040106A   5D              pop ebp
```

```
EAX     00000001
EBX     7FFDC000
ECX     004010D8     ife
EDX     0040D410     ife
EBP     0012FF40
ESP     0012FF38
ESI     00000000
EDI     00000000

EIP     00401042     ife

EFLAGS     00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (E
LastStatus  00000000 (S

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000
ST(1) 0000000000000000
ST(2) 0000000000000000
ST(3) 0000000000000000
ST(4) 0000000000000000
```

```
Address  Hex                                              ASCII
0040A140 49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20 Input Number1 :
0040A150 00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75 ....%d..Input Nu
0040A160 6D 62 65 72 32 20 3A 20 00 00 00 00 25 64 00 00 mber2 : ....%d..
```

```
0012FF30  0040A16C  "%d"
0012FF34  0012FF3C
0012FF38  00000007
0012FF3C  00000008
0012FF40  0012FF88
0012FF44  004012F7  retu
```

Figure 9.9: Again after another scanf

▼Line 58-61

; Line 14
mov edx, DWORD PTR _iNumber1$[ebp]
cmp edx, DWORD PTR _iNumber2$[ebp]
jne SHORT $LN2@main

The C/C++ code on line 14 compares the two numbers:

if (iNumber1 == iNumber2)

Now we have both the numbers **iNumber1** and **iNumber2** stored at **[EBP-8]** and **[EBP-4],** respectively. To compare these two numbers, the first **MOV** instruction will move the value stored at

[EBP-8] to the **EDX** register, which is number **0x00000007** and then use the CMP instruction to compare the value stored at **[EBP-4]** with that of the **EDX** register.

In our case, **EDX** = **0x00000007** is compared with **[EBP-8]** =

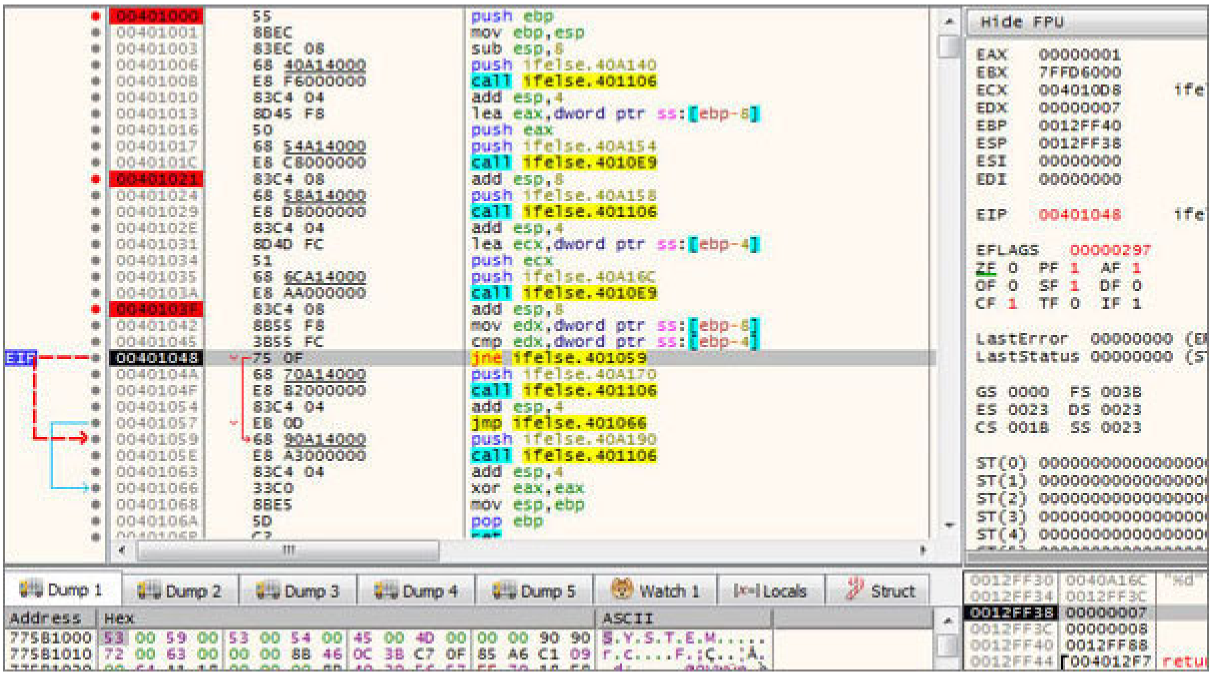
The **CMP** instruction can be visualized as a **SUB** instruction and as it is a conditional it will work as follows:

[EDX] - [EBP-8], based on the subtraction result, the control flag is set

0x00000007 - 0x00000008 != 0, so it sets ZF (Zero Flag) = 0

Following the **CMP** is the which is **Jump if Not** As both numbers are not equal and the jump will occur to the location specified after the **JNE** instruction, which is Let's check the stack state and a screenshot of x32dbg before the **JNE** instruction:

[ESP-0x8] 0012FF30 0040A16C "%d", parameter to 2nd scanf()
[ESP-0x4] 0012FF34 0012FF3C Memory location of placeholder for iNumber2
[ESP] 0012FF38 00000007 [EBP-8] 7 is stored, Input placeholder for iNumber1
[ESP+0xC] 0012FF3C 00000008 [EBP-4] 8 is stored, Input placeholder for iNumber2
[ESP+0x10] 0012FF40 0012FF88 [EBP]
[ESP+0x14] 0012FF44 004012F7 return to ifelse.004012F7 from ifelse.00401000



```
00401000   55          push ebp
00401001   8BEC        mov ebp,esp
00401003   83EC 08     sub esp,8
00401006   68 40A14000 push ifelse.40A140
0040100B   E8 F6000000 call ifelse.401106
00401010   83C4 04     add esp,4
00401013   8D45 F8     lea eax,dword ptr ss:[ebp-8]
00401016   50          push eax
00401017   68 54A14000 push ifelse.40A154
0040101C   E8 C8000000 call ifelse.4010E9
00401021   83C4 08     add esp,8
00401024   68 58A14000 push ifelse.40A158
00401029   E8 D8000000 call ifelse.401106
0040102E   83C4 04     add esp,4
00401031   8D4D FC     lea ecx,dword ptr ss:[ebp-4]
00401034   51          push ecx
00401035   68 6CA14000 push ifelse.40A16C
0040103A   E8 AA000000 call ifelse.4010E9
0040103F   83C4 08     add esp,8
00401042   8B55 F8     mov edx,dword ptr ss:[ebp-8]
00401045   3B55 FC     cmp edx,dword ptr ss:[ebp-4]
00401048   75 0F       jne ifelse.401059
0040104A   68 70A14000 push ifelse.40A170
0040104F   E8 B2000000 call ifelse.401106
00401054   83C4 04     add esp,4
00401057   EB 0D       jmp ifelse.401066
00401059   68 90A14000 push ifelse.40A190
0040105E   E8 A3000000 call ifelse.401106
00401063   83C4 04     add esp,4
00401066   33C0        xor eax,eax
00401068   8BE5        mov esp,ebp
0040106A   5D          pop ebp
```

```
EAX   00000001
EBX   7FFD6000
ECX   004010D8
EDX   00000007
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   00401048

EFLAGS   00000297
ZF 0   PF 1   AF 1
OF 0   SF 1   DF 0
CF 1   TF 0   IF 1

LastError   00000000
LastStatus  00000000

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023
```

```
Address   Hex                                               ASCII
77581000  53 00 59 00 53 00 54 00 45 00 4D 00 00 00 90 90  S.Y.S.T.E.M.....
77581010  72 00 63 00 00 00 8B 46 0C 3B C7 0F 85 A6 C1 09  r.c....F.;Ç..¦Á.
```

```
0012FF30  0040A16C  "%d"
0012FF34  0012FF3C
0012FF38  00000007
0012FF3C  00000008
0012FF40  0012FF88
0012FF44  004012F7
```

Figure 9.10: Before JNE instruction

In our case, **JNE** will move the instruction pointer to the **$LN2@main** location. If in case the user enters the same numbers, the CMP instruction will result in **ZF (Zero Flag)** = That will move the instruction pointer to print **"Number1 and Number2 are equal"** on the console, followed by the unconditional jump instruction to **$LN3@main SHORT $LN3@main** at line 67), where **$LN3@main** is the closing of the function and the code. The stack state is shown in the following screenshot:

▼Line 68-72

$LN2@main:
; Line 17
push OFFSET $SG4686

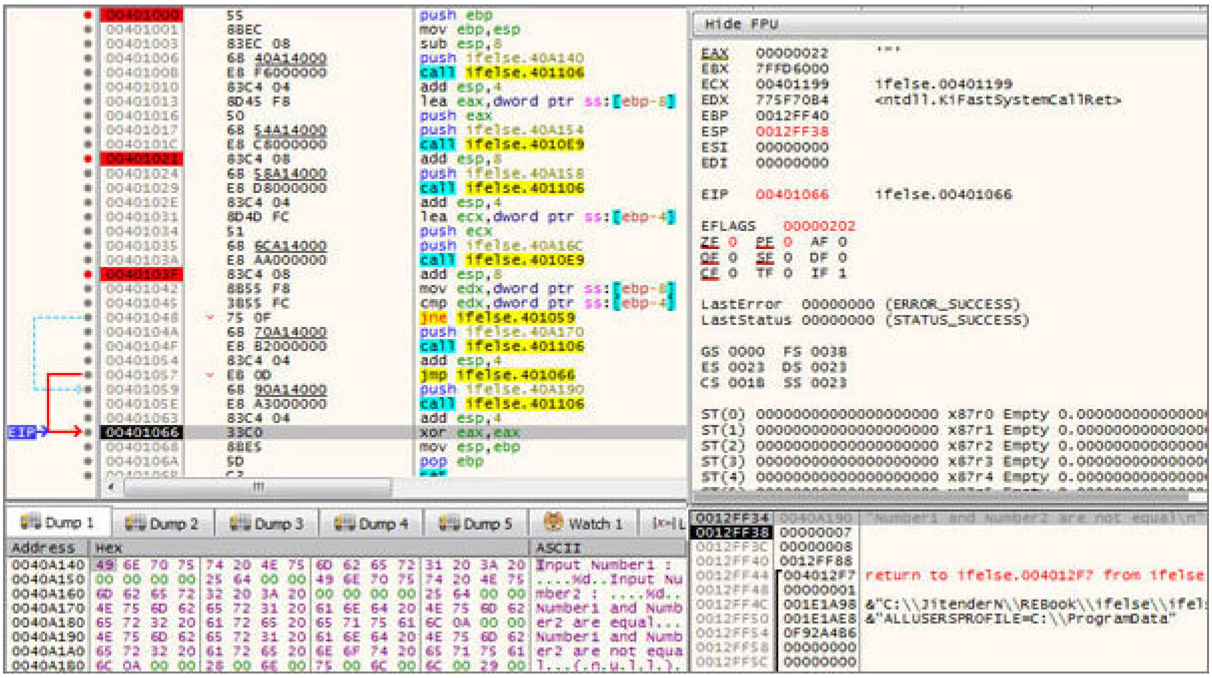
```
call _printf
add esp, 4
```

The C/C++ equivalent is:

```
printf("Number1 and Number2 are not equal\n");
```

It prints ‘Number1 and Numver2 are not equal’. The ASM code pushes the string constant **$SG4686** on the stack to be used by the **printf** function as an argument. This will print **‘Number1 and Number2 are not equal’** on the console. Following this is the **ADD** instruction which will clean the stack by 4 bytes, which was used by string constant **$SG4686** in **PUSH** operation. The stack state after this will be as follows:

```
[ESP-0x4] 0012FF34  0040A190  "Number1 and Number2 are not equal\n", arg to printf
[ESP]  0012FF38  00000007  [EBP-8] 7 is stored, Input placeholder for iNumber1
[ESP+0xC] 0012FF3C  00000008  [EBP-4] 8 is stored, Input placeholder for iNumber2
[ESP+0x10] 0012FF40  0012FF88  [EBP]
[ESP+0x14] 0012FF44  004012F7  return to ifelse.004012F7 from ifelse.00401000
```



```
00401000   55             push ebp
00401001   8BEC           mov ebp,esp
00401003   83EC 08        sub esp,8
00401006   68 40A14000    push ifelse.40A140
0040100B   E8 F6000000    call ifelse.401106
00401010   83C4 04        add esp,4
00401013   8D45 F8        lea eax,dword ptr ss:[ebp-8]
00401016   50             push eax
00401017   68 54A14000    push ifelse.40A154
0040101C   E8 C8000000    call ifelse.4010E9
00401021   83C4 08        add esp,8
00401024   68 58A14000    push ifelse.40A158
00401029   E8 D8000000    call ifelse.401106
0040102E   83C4 04        add esp,4
00401031   8D4D FC        lea ecx,dword ptr ss:[ebp-4]
00401034   51             push ecx
00401035   68 6CA14000    push ifelse.40A16C
0040103A   E8 AA000000    call ifelse.4010E9
0040103F   83C4 08        add esp,8
00401042   8B55 F8        mov edx,dword ptr ss:[ebp-8]
00401045   3B55 FC        cmp edx,dword ptr ss:[ebp-4]
00401048   75 0F          jne ifelse.401059
0040104A   68 70A14000    push ifelse.40A170
0040104F   E8 B2000000    call ifelse.401106
00401054   83C4 04        add esp,4
00401057   EB 0D          jmp ifelse.401066
00401059   68 90A14000    push ifelse.40A190
0040105E   E8 A3000000    call ifelse.401106
00401063   83C4 04        add esp,4
00401066   33C0           xor eax,eax
00401068   8BE5           mov esp,ebp
0040106A   5D             pop ebp
```

```
Hide FPU

EAX   00000022     '"'
EBX   7FFD6000
ECX   00401199     ifelse.00401199
EDX   775F70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF40
ESP   0012FF38
ESI   00000000
EDI   00000000

EIP   00401066     ifelse.00401066

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.00000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.00000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.00000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.00000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.00000000000000
```

```
Address   Hex                                               ASCII
0040A140  49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
0040A150  00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75  ....%d..Input Nu
0040A160  6D 62 65 72 32 20 3A 20 00 00 00 00 25 64 00 00  mber2 : ....%d..
0040A170  4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
0040A180  65 72 32 20 61 72 65 20 65 71 75 61 6C 0A 00 00  er2 are equal...
0040A190  4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
0040A1A0  65 72 32 20 61 72 65 20 6E 6F 74 20 65 71 75 61  er2 are not equa
0040A1B0  6C 0A 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  l...(.n.u.l.l.).
```

```
0012FF34  0040A190  "Number1 and Number2 are not equal\n"
0012FF38  00000007
0012FF3C  00000008
0012FF40  0012FF88
0012FF44  004012F7  return to ifelse.004012F7 from ifelse
0012FF48  00000001
0012FF4C  001E1A98  &"C:\\JitenderN\\REBook\\ifelse\\ifel
0012FF50  001E1AE8  &"ALLUSERSPROFILE=C:\\ProgramData"
0012FF54  0F92A4B6
0012FF58  00000000
0012FF5C  00000000
```

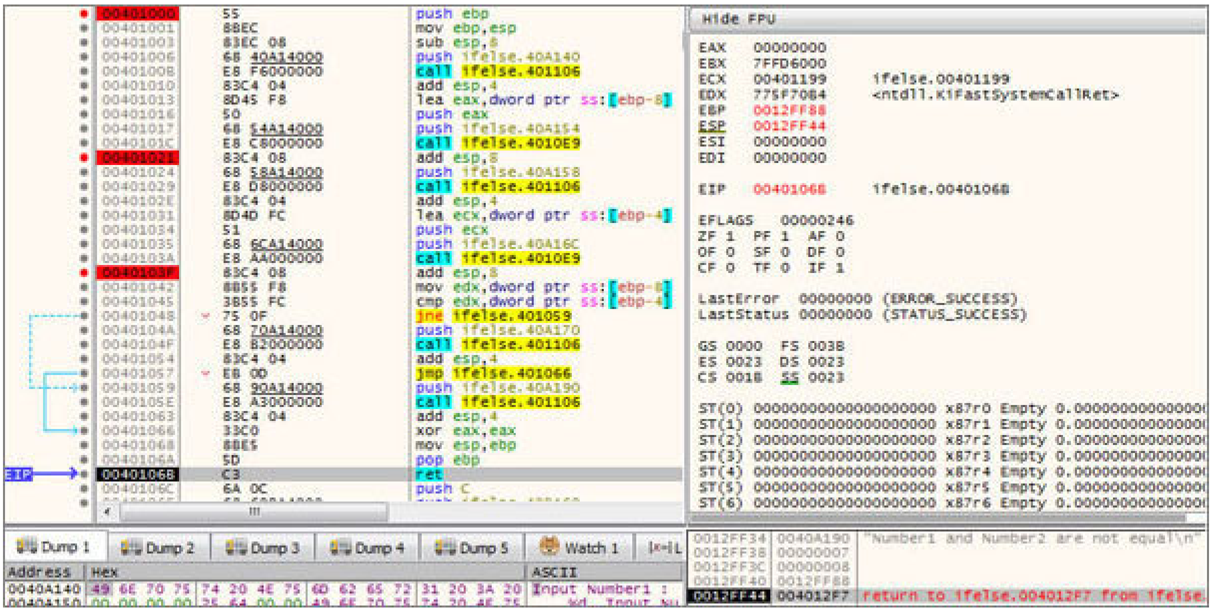
Figure 9.11: *Print "Number1 and Number2 are not equal"*

▼Line 73-81

```
$LN3@main:

; Line 18
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

XOR and the function epilogue will clean up **EAX** and stack, respectively. **END** will end the code by returning 0. The stack state is shown in the following screenshot:

[ESP-0x10] 0012FF34 0040A190 JUNK
[ESP-0xC] 0012FF38 00000007 JUNK
[ESP-0x8] 0012FF3C 00000008 JUNK
[ESP-0x4] 0012FF40 0012FF88 [EBP] popped up
[ESP] 0012FF44 004012F7 return to ifelse.004012F7 from ifelse.00401000



```
00401000  55           push ebp
00401001  8BEC         mov ebp,esp
00401003  83EC 08      sub esp,8
00401006  68 40A14000  push ifelse.40A140
0040100B  E8 F6000000  call ifelse.401106
00401010  83C4 04      add esp,4
00401013  8D45 F8      lea eax,dword ptr ss:[ebp-8]
00401016  50           push eax
00401017  68 54A14000  push ifelse.40A154
0040101C  E8 C8000000  call ifelse.4010E9
00401021  83C4 08      add esp,8
00401024  68 58A14000  push ifelse.40A158
00401029  E8 D8000000  call ifelse.401106
0040102E  83C4 04      add esp,4
00401031  8D4D FC      lea ecx,dword ptr ss:[ebp-4]
00401034  51           push ecx
00401035  68 6CA14000  push ifelse.40A16C
0040103A  E8 AA000000  call ifelse.4010E9
0040103F  83C4 08      add esp,8
00401042  8B55 F8      mov edx,dword ptr ss:[ebp-8]
00401045  3B55 FC      cmp edx,dword ptr ss:[ebp-4]
00401048  75 0F        jne ifelse.401059
0040104A  68 70A14000  push ifelse.40A170
0040104F  E8 B2000000  call ifelse.401106
00401054  83C4 04      add esp,4
00401057  EB 0D        jmp ifelse.401066
00401059  68 90A14000  push ifelse.40A190
0040105E  E8 A3000000  call ifelse.401106
00401063  83C4 04      add esp,4
00401066  33C0         xor eax,eax
00401068  8BE5         mov esp,ebp
0040106A  5D           pop ebp
0040106B  C3           ret
0040106C  6A 0C        push C
```

```
EAX   00000000
EBX   7FFD6000
ECX   00401199     ifelse.00401199
EDX   775F70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   0040106B     ifelse.0040106B

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000
```

```
Address   Hex                                              ASCII
0040A140  49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
```

```
0012FF34  0040A190  "Number1 and Number2 are not equal\n"
0012FF38  00000007
0012FF3C  00000008
0012FF40  0012FF88
0012FF44  004012F7  return to ifelse.004012F7 from ifelse.
```

Figure 9.12: *Stack cleaned*

If-else statement with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

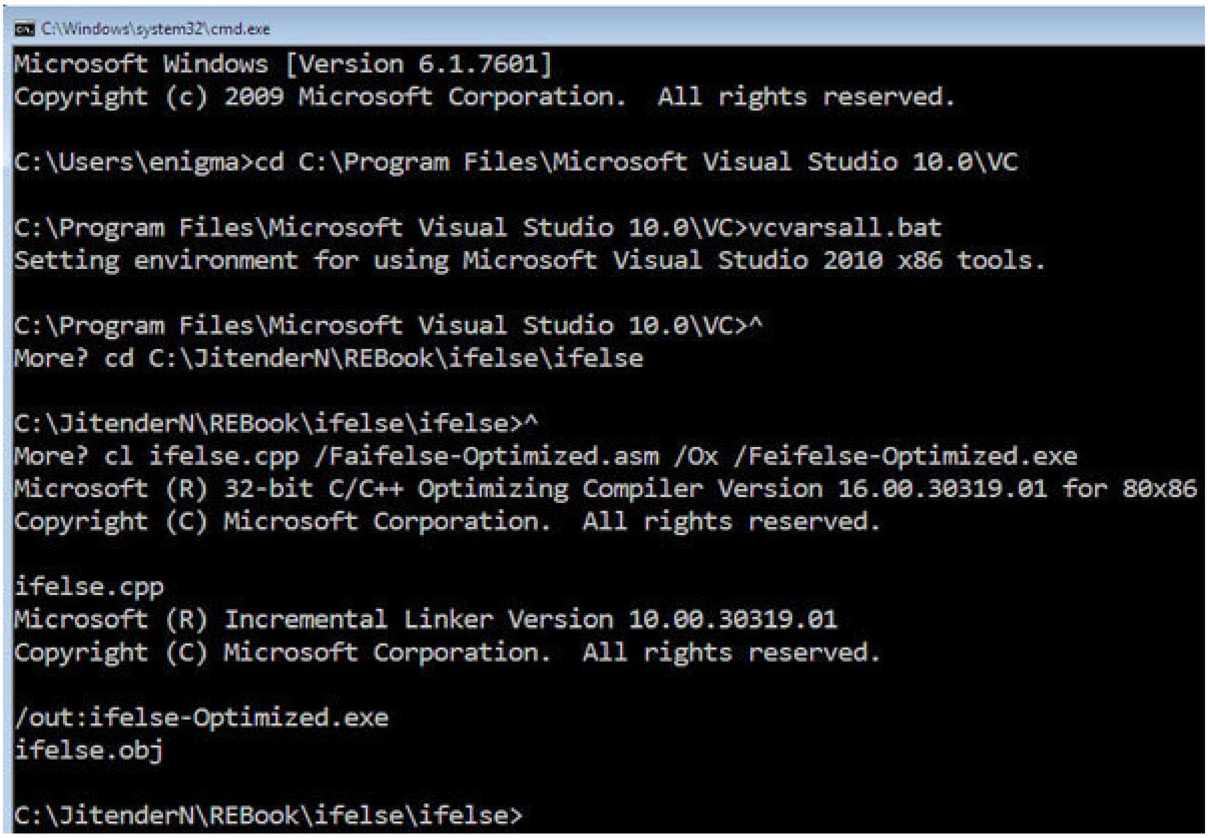
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\ifelse\ifelse

C:\JitenderN\REBook\ifelse\ifelse>^
More? cl ifelse.cpp /Faifelse-Optimized.asm /Ox /Feifelse-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ifelse.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:ifelse-Optimized.exe
ifelse.obj

C:\JitenderN\REBook\ifelse\ifelse>
```

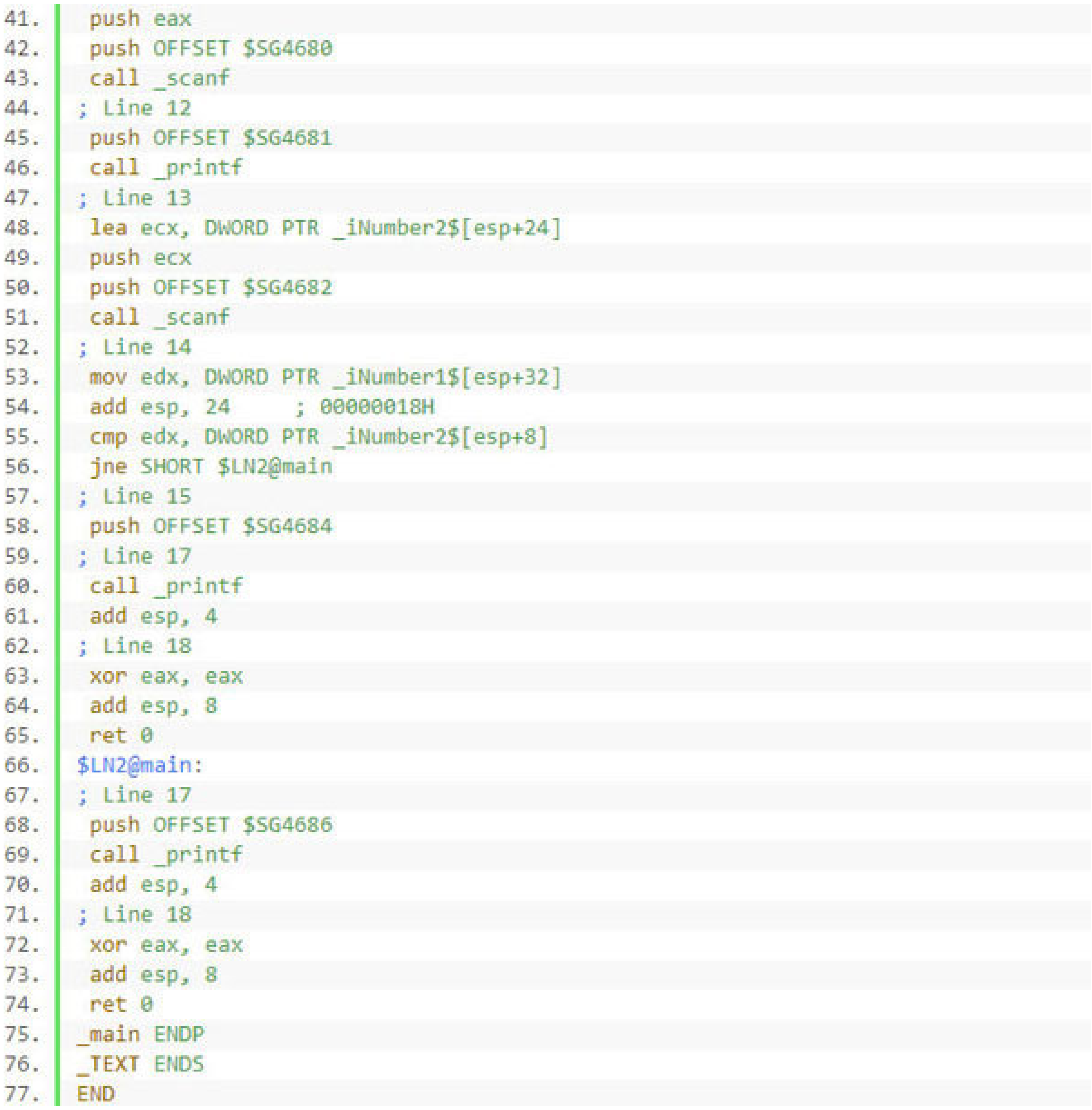
Figure 9.13: *If-else statement with Optimization*

This will generate the assembly code and the EXE file. For analysis purpose, we will disable the ASLR. It is a security mechanism by which the base address of the PE file is randomized on every load of the PE file generated with our MSVC compiler. To disable ASLR, follow the same steps by using the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can** For step-by-step reference to disable ASLR, refer to the Appendix.

Now, let's move on to the generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\ifelse\ifelse\ifelse.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4679 DB 'Input Number1 : ', 00H
 ORG $+3
$SG4680 DB '%d', 00H
 ORG $+1
$SG4681 DB 'Input Number2 : ', 00H
 ORG $+3
$SG4682 DB '%d', 00H
 ORG $+1
$SG4684 DB 'Number1 and Number2 are equal', 0aH, 00H
 ORG $+1
$SG4686 DB 'Number1 and Number2 are not equal', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_iNumber1$ = -8       ; size = 4
_iNumber2$ = -4       ; size = 4
_main PROC
; File c:\jitendern\rebook\ifelse\ifelse\ifelse.cpp
; Line 7
 sub esp, 8
; Line 10
 push OFFSET $SG4679
 call _printf
; Line 11
 lea eax, DWORD PTR _iNumber1$[esp+12]
```

Figure 9.14: *ifelse-Optimized.asm-Part-1*



```
41.   push eax
42.   push OFFSET $SG4680
43.   call _scanf
44.  ; Line 12
45.   push OFFSET $SG4681
46.   call _printf
47.  ; Line 13
48.   lea ecx, DWORD PTR _iNumber2$[esp+24]
49.   push ecx
50.   push OFFSET $SG4682
51.   call _scanf
52.  ; Line 14
53.   mov edx, DWORD PTR _iNumber1$[esp+32]
54.   add esp, 24      ; 00000018H
55.   cmp edx, DWORD PTR _iNumber2$[esp+8]
56.   jne SHORT $LN2@main
57.  ; Line 15
58.   push OFFSET $SG4684
59.  ; Line 17
60.   call _printf
61.   add esp, 4
62.  ; Line 18
63.   xor eax, eax
64.   add esp, 8
65.   ret 0
66.  $LN2@main:
67.  ; Line 17
68.   push OFFSET $SG4686
69.   call _printf
70.   add esp, 4
71.  ; Line 18
72.   xor eax, eax
73.   add esp, 8
74.   ret 0
75.  _main ENDP
76.  _TEXT ENDS
77.  END
```

Figure 9.15: *ifelse-Optimized.asm-Part-2*

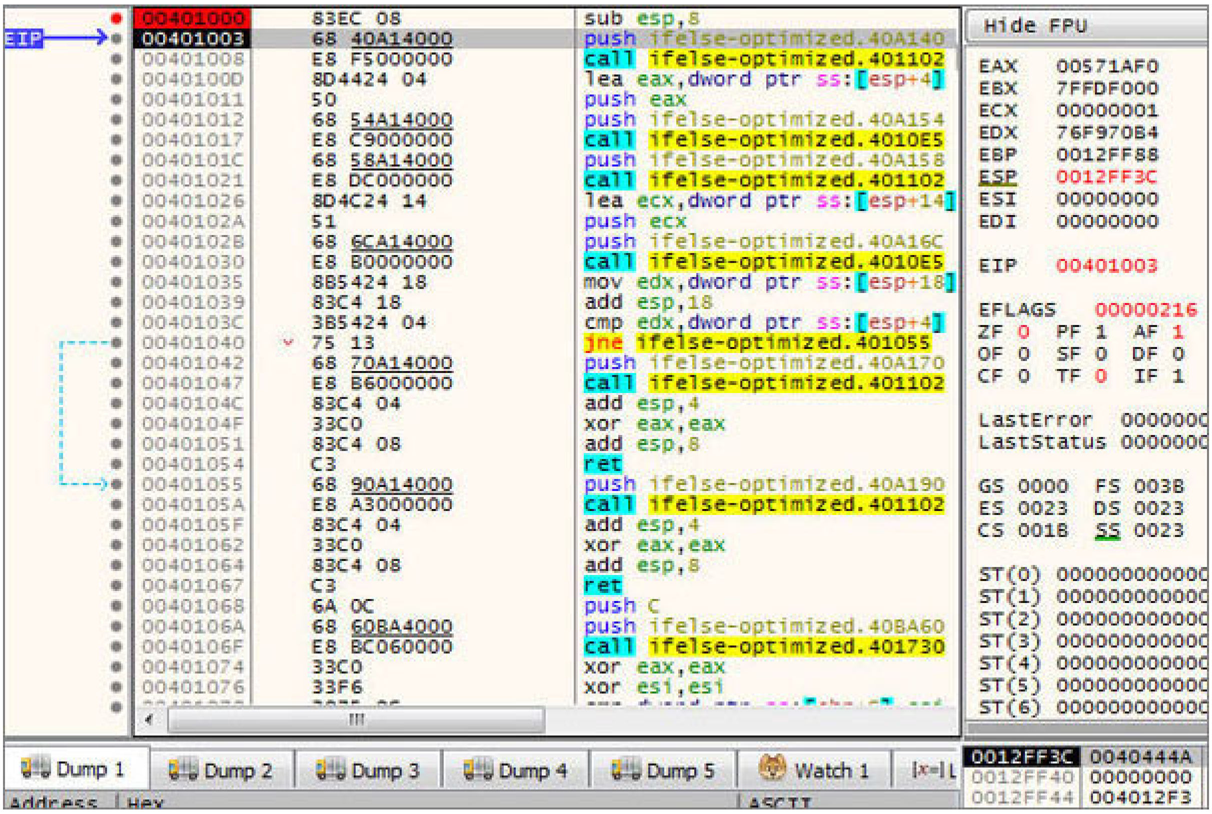
This assembly code is optimized by removing unnecessary lines to consume fewer resources. In this code, we will not see the function prologue and epilogue. We can also observe that ESP is used as a reference rather than EBP; this is because of the absence of the function prologue and epilogue. Let us start the analysis and this time, we will enter the same number.

▼Line 34-35

```
; Line 7
sub esp, 8
```

The **SUB** instruction is creating room for the local variables of the **main** function on the stack, **iNumber1** and To check the stack state, we will open the PE file in x32dbg and place our first breakpoint at the start of the **main** procedure. The stack state and the x32dbg screenshot after the **SUB** instruction is as follows:

[ESP] 0012FF3C 0040444A right now is JUNK, Input placeholder for iNumber1
[ESP+0x4] 0012FF40 00000000 right now is JUNK, Input placeholder for iNumber2
[ESP+0x8] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000



```
00401000  83EC 08        sub esp,8
00401003  68 40A14000    push ifelse-optimized.40A140
00401008  E8 F5000000    call ifelse-optimized.401102
0040100D  8D4424 04      lea eax,dword ptr ss:[esp+4]
00401011  50             push eax
00401012  68 54A14000    push ifelse-optimized.40A154
00401017  E8 C9000000    call ifelse-optimized.4010E5
0040101C  68 58A14000    push ifelse-optimized.40A158
00401021  E8 DC000000    call ifelse-optimized.401102
00401026  8D4C24 14      lea ecx,dword ptr ss:[esp+14]
0040102A  51             push ecx
0040102B  68 6CA14000    push ifelse-optimized.40A16C
00401030  E8 B0000000    call ifelse-optimized.4010E5
00401035  8B5424 18      mov edx,dword ptr ss:[esp+18]
00401039  83C4 18        add esp,18
0040103C  3B5424 04      cmp edx,dword ptr ss:[esp+4]
00401040  75 13          jne ifelse-optimized.401055
00401042  68 70A14000    push ifelse-optimized.40A170
00401047  E8 B6000000    call ifelse-optimized.401102
0040104C  83C4 04        add esp,4
0040104F  33C0           xor eax,eax
00401051  83C4 08        add esp,8
00401054  C3             ret
00401055  68 90A14000    push ifelse-optimized.40A190
0040105A  E8 A3000000    call ifelse-optimized.401102
0040105F  83C4 04        add esp,4
00401062  33C0           xor eax,eax
00401064  83C4 08        add esp,8
00401067  C3             ret
00401068  6A 0C          push C
0040106A  68 60BA4000    push ifelse-optimized.40BA60
0040106F  E8 BC060000    call ifelse-optimized.401730
00401074  33C0           xor eax,eax
00401076  33F6           xor esi,esi
```

```
EAX 00571AF0
EBX 7FFDF000
ECX 00000001
EDX 76F970B4
EBP 0012FF88
ESP 0012FF3C
ESI 00000000
EDI 00000000

EIP 00401003

EFLAGS   00000216
ZF 0  PF 1  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000000C
LastStatus 000000C

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000C
ST(1) 00000000000C
ST(2) 00000000000C
ST(3) 00000000000C
ST(4) 00000000000C
ST(5) 00000000000C
ST(6) 00000000000C
```

```
0012FF3C 0040444A
0012FF40 00000000
0012FF44 004012F3
```

Figure 9.16: Creating room for our local variable

▼Line 36-38

; Line 10
push OFFSET $SG4679
call _printf

The C/C++ equivalent of the ASM is:

printf("Input Number1 : ");

It will print the string constant **$SG4679** on the console. **$SG4679** was pushed on the stack before the **printf** function call. The stack state after the **CALL** instruction is as follows:

[ESP] 0012FF38 0040A140 “Input Number1 : “, parameter to printf()
[ESP+0x4] 0012FF3C 0040444A right now is JUNK, Input placeholder for iNumber1
[ESP+0x8] 0012FF40 00000000 right now is JUNK, Input placeholder for iNumber2
[ESP+0xC] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000

▼Line 39-43

```
; Line 11
lea eax, DWORD PTR _iNumber1$[esp+12]
push eax
push OFFSET $SG4680
call _scanf
```

The C/C++ code on line 11 asks the user to input the first number, This number will be stored at the **&iNumber1** location.

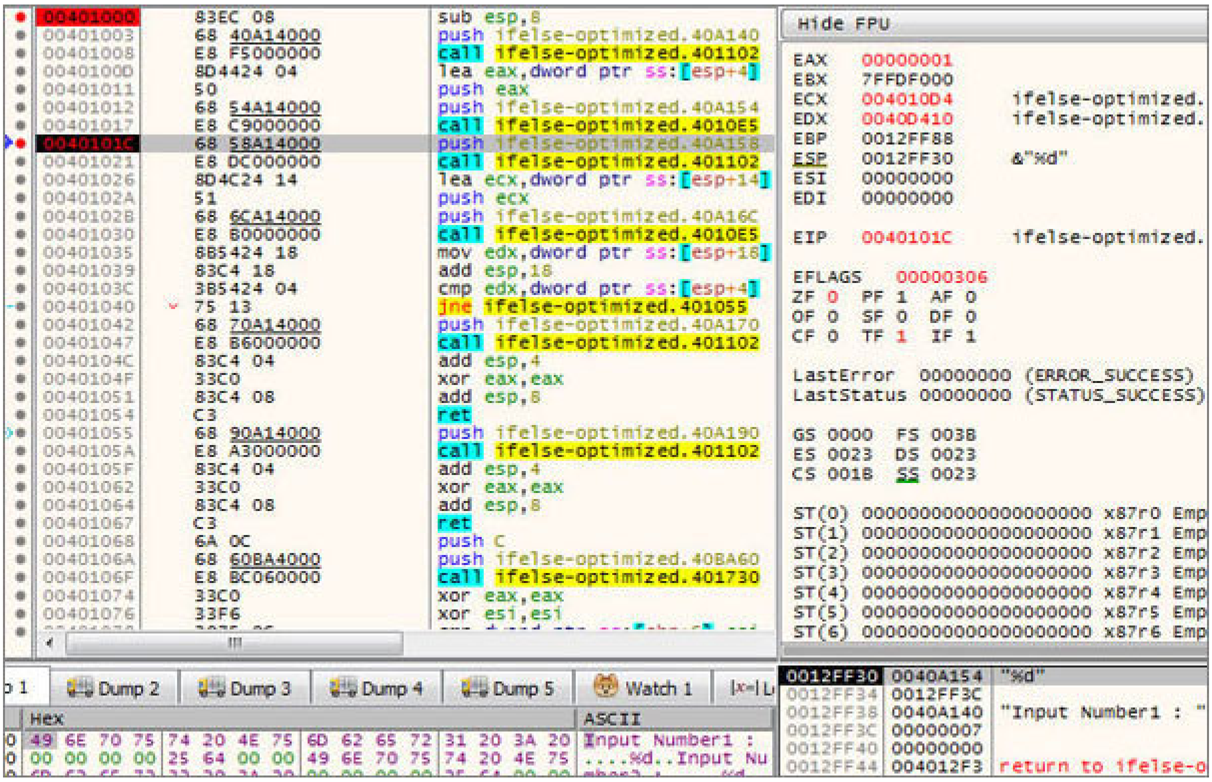
```
scanf("%d", &iNumber1);
```

In ASM, the LEA instruction is evaluated to:

```
lea eax, ss:[esp+0x4]
```

This will load the effective address of **ESP+0x4** to This is the memory location on the stack where the **iNumber1** input by the user will be stored. As the **scanf** function expects two arguments, so both the arguments to **scanf** are pushed onto the stack before the **CALL** instruction. Arguments to **scanf** are memory location, on which **iNumber1** is stored, and the string constant When a call to **scanf** is done, it will ask the user to enter the number. We have inserted one more breakpoint after the **scanf** function to view the stack after the **scanf** call. Refer to the stack state and x32dbg in the following screenshot:

[ESP] 0012FF30 0040A154 “%d”, parameter to scanf()
[ESP+0x4] 0012FF34 0012FF3C Memory location of Input placeholder for iNumber1
[ESP+0x8] 0012FF38 0040A140 “Input Number1 : “, parameter to printf()
[ESP+0xC] 0012FF3C 00000007 7 is stored here, Input placeholder for iNumber1
[ESP+0x10] 0012FF40 00000000 right now is JUNK, Input placeholder for iNumber2
[ESP+0x14] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000



```
00401000   83EC 08         sub esp,8
00401003   68 40A14000     push ifelse-optimized.40A140
00401008   E8 F5000000     call ifelse-optimized.401102
0040100D   8D4424 04       lea eax,dword ptr ss:[esp+4]
00401011   50              push eax
00401012   68 54A14000     push ifelse-optimized.40A154
00401017   E8 C9000000     call ifelse-optimized.4010E5
0040101C   68 58A14000     push ifelse-optimized.40A158
00401021   E8 DC000000     call ifelse-optimized.401102
00401026   8D4C24 14       lea ecx,dword ptr ss:[esp+14]
0040102A   51              push ecx
0040102B   68 6CA14000     push ifelse-optimized.40A16C
00401030   E8 B0000000     call ifelse-optimized.4010E5
00401035   8B5424 18       mov edx,dword ptr ss:[esp+18]
00401039   83C4 18         add esp,18
0040103C   3B5424 04       cmp edx,dword ptr ss:[esp+4]
00401040   75 13           jne ifelse-optimized.401055
00401042   68 70A14000     push ifelse-optimized.40A170
00401047   E8 B6000000     call ifelse-optimized.401102
0040104C   83C4 04         add esp,4
0040104F   33C0            xor eax,eax
00401051   83C4 08         add esp,8
00401054   C3              ret
00401055   68 90A14000     push ifelse-optimized.40A190
0040105A   E8 A3000000     call ifelse-optimized.401102
0040105F   83C4 04         add esp,4
00401062   33C0            xor eax,eax
00401064   83C4 08         add esp,8
00401067   C3              ret
00401068   6A 0C           push C
0040106A   68 60BA4000     push ifelse-optimized.40BA60
0040106F   E8 BC060000     call ifelse-optimized.401730
00401074   33C0            xor eax,eax
00401076   33F6            xor esi,esi
```

```
EAX   00000001
EBX   7FFDF000
ECX   004010D4      ifelse-optimized.
EDX   0040D410      ifelse-optimized.
EBP   0012FF88
ESP   0012FF30      &"%d"
ESI   00000000
EDI   00000000

EIP   0040101C      ifelse-optimized.

EFLAGS   00000306
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Emp
ST(1) 00000000000000000000 x87r1 Emp
ST(2) 00000000000000000000 x87r2 Emp
ST(3) 00000000000000000000 x87r3 Emp
ST(4) 00000000000000000000 x87r4 Emp
ST(5) 00000000000000000000 x87r5 Emp
ST(6) 00000000000000000000 x87r6 Emp
```

```
Hex                                                  ASCII
49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20     Input Number1 :
00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75     ....%d..Input Nu
```

```
0012FF30  0040A154  "%d"
0012FF34  0012FF3C
0012FF38  0040A140  "Input Number1 : "
0012FF3C  00000007
0012FF40  00000000
0012FF44  004012F3  return to ifelse-o
```

Figure 9.17: *Breakpoint after the first scanf*

▼Line 44-46

```
; Line 12
push OFFSET $SG4681
call _printf
```

The C/C++ code on line 12 prints the following message on the console:

```
printf("Input Number2 : ");
```

In ASM, it does the same by pushing the string constant **$SG4681** on the stack and the call **printf** function.

```
[ESP] 0012FF2C 0040A158 "Input Number2 : ", parameter to printf()
[ESP+0x4] 0012FF30 0040A154 "%d", parameter to scanf()
[ESP+0x8] 0012FF34 0012FF3C Memory location of Input placeholder for iNumber1
[ESP+0xC] 0012FF38 0040A140 "Input Number1 : ", parameter to printf()
[ESP+0x10] 0012FF3C 00000007 7 is stored here, Input placeholder for iNumber1
[ESP+0x14] 0012FF40 00000000 right now is JUNK, Input placeholder for iNumber2
[ESP+0x18] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000
```

▼Line 47-51

```
; Line 13
lea ecx, DWORD PTR _iNumber2$[esp+24]
push ecx
push OFFSET $SG4682
call _scanf
```

The C/C++ code on line 13 asks the user to input the second number, This number will be stored at the **&iNumber2** location.

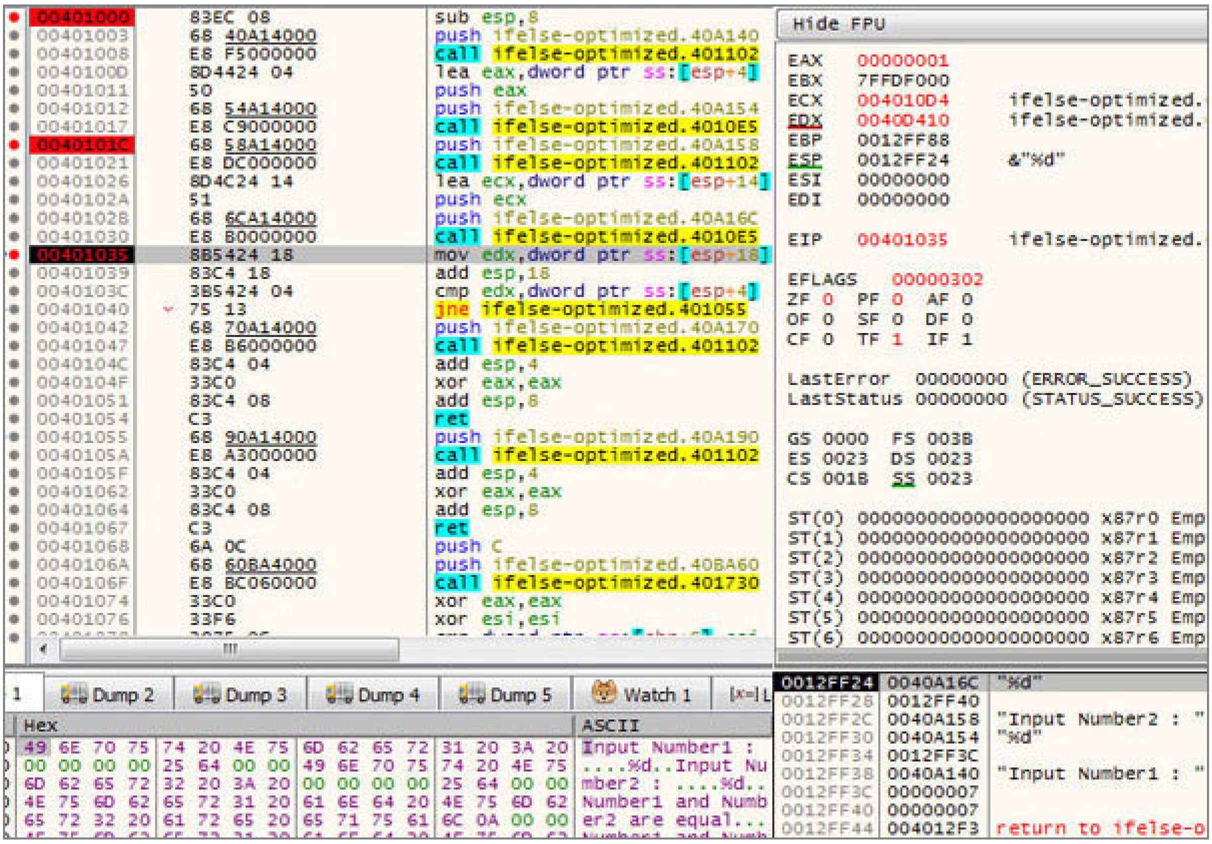
```
scanf("%d", &iNumber2);
```

In ASM, the LEA instruction is evaluated to:

lea ecx, ss:[esp+0x14]

This will load the effective address of **ESP+0x14** to This is memory location on the stack where the **iNumber2** input by the user will be stored. As the **scanf** function expects two arguments, so both the arguments to **scanf** are pushed onto the stack before the **CALL** to the **scanf** function. Arguments to scanf are **iNumber2** and the string constant When a call to **scanf** is done, it will ask the user to enter the number. This time, we have entered the same number, 7. We have inserted one more breakpoint after the **scanf** function to view the stack after the **scanf** call. Refer to the stack state and x32dbg in the following screenshot:.

```
[ESP]  0012FF24  0040A16C  "%d"
[ESP+0x4] 0012FF28  0012FF40
[ESP+0x8] 0012FF2C  0040A158  "Input Number2 : ", parameter to printf()
[ESP+0xC] 0012FF30  0040A154  "%d", parameter to scanf()
[ESP+0x10] 0012FF34  0012FF3C  Memory location of Input placeholder for iNumber1
[ESP+0x14] 0012FF38  0040A140  "Input Number1 : ", parameter to printf()
[ESP+0x18] 0012FF3C  00000007  7 is stored here, Input placeholder for iNumber1
[ESP+0x1C] 0012FF40  00000007  7 is stored here, Input placeholder for iNumber2

[ESP+0x20] 0012FF44  004012F3  return to 0x004012F3 from 0x00401000
```



```
00401000   83EC 08          sub esp,8
00401003   68 40A14000      push ifelse-optimized.40A140
00401008   E8 F5000000      call ifelse-optimized.401102
0040100D   8D4424 04        lea eax,dword ptr ss:[esp+4]
00401011   50               push eax
00401012   68 54A14000      push ifelse-optimized.40A154
00401017   E8 C9000000      call ifelse-optimized.4010E5
0040101C   68 58A14000      push ifelse-optimized.40A158
00401021   E8 DC000000      call ifelse-optimized.401102
00401026   8D4C24 14        lea ecx,dword ptr ss:[esp+14]
0040102A   51               push ecx
0040102B   68 6CA14000      push ifelse-optimized.40A16C
00401030   E8 B0000000      call ifelse-optimized.4010E5
00401035   8B5424 18        mov edx,dword ptr ss:[esp+18]
00401039   83C4 18          add esp,18
0040103C   3B5424 04        cmp edx,dword ptr ss:[esp+4]
00401040   75 13            jne ifelse-optimized.401055
00401042   68 70A14000      push ifelse-optimized.40A170
00401047   E8 B6000000      call ifelse-optimized.401102
0040104C   83C4 04          add esp,4
0040104F   33C0             xor eax,eax
00401051   83C4 08          add esp,8
00401054   C3               ret
00401055   68 90A14000      push ifelse-optimized.40A190
0040105A   E8 A3000000      call ifelse-optimized.401102
0040105F   83C4 04          add esp,4
00401062   33C0             xor eax,eax
00401064   83C4 08          add esp,8
00401067   C3               ret
00401068   6A 0C            push C
0040106A   68 60BA4000      push ifelse-optimized.40BA60
0040106F   E8 BC060000      call ifelse-optimized.401730
00401074   33C0             xor eax,eax
00401076   33F6             xor esi,esi
```

```
EAX 00000001
EBX 7FFDF000
ECX 004010D4     ifelse-optimized.
EDX 0040D410     ifelse-optimized.
EBP 0012FF88
ESP 0012FF24     &"%d"
ESI 00000000
EDI 00000000

EIP 00401035     ifelse-optimized.

EFLAGS   00000302
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Emp
ST(1) 00000000000000000000 x87r1 Emp
ST(2) 00000000000000000000 x87r2 Emp
ST(3) 00000000000000000000 x87r3 Emp
ST(4) 00000000000000000000 x87r4 Emp
ST(5) 00000000000000000000 x87r5 Emp
ST(6) 00000000000000000000 x87r6 Emp
```

```
49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75  ....%d..Input Nu
6D 62 65 72 32 20 3A 20 00 00 00 00 25 64 00 00  mber2 : ....%d..
4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
65 72 32 20 61 72 65 20 65 71 75 61 6C 0A 00 00  er2 are equal...
```

```
0012FF24  0040A16C  "%d"
0012FF28  0012FF40
0012FF2C  0040A158  "Input Number2 : "
0012FF30  0040A154  "%d"
0012FF34  0012FF3C
0012FF38  0040A140  "Input Number1 : "
0012FF3C  00000007
0012FF40  00000007
0012FF44  004012F3  return to ifelse-o
```

Figure 9.18: *Breakpoint after the second scanf*

▼Line 51-56

```
; Line 14
mov edx, DWORD PTR _iNumber1$[esp+32]
add esp, 24          ; 00000018H
cmp edx, DWORD PTR _iNumber2$[esp+8]
jne SHORT $LN2@main
```

The C/C++ code on line 14 compares the two numbers:

```
if (iNumber1 == iNumber2)
```

In ASM, the **MOV** instruction will move **iNumber1** stored at **ss:[esp+0x18]** to The **ADD** instruction will clean the stack by adding 24 bytes to Once the stack is cleaned, the **CMP** instruction compares the value at **EDX** with **ss:[esp+0x4].** As both the numbers are equal, it will set As **Jump if Not Equal** condition is not met, the instruction pointer will move to the next instruction following the Refer to the stack state after the **JNE** instruction execution and x32dbg in the following screenshot:

[ESP-0x18] 0012FF24 0040A16C JUNK

[ESP-0x14] 0012FF28 0012FF40 JUNK

[ESP-0x10] 0012FF2C 0040A158 JUNK

[ESP-0xC] 0012FF30 0040A154 JUNK

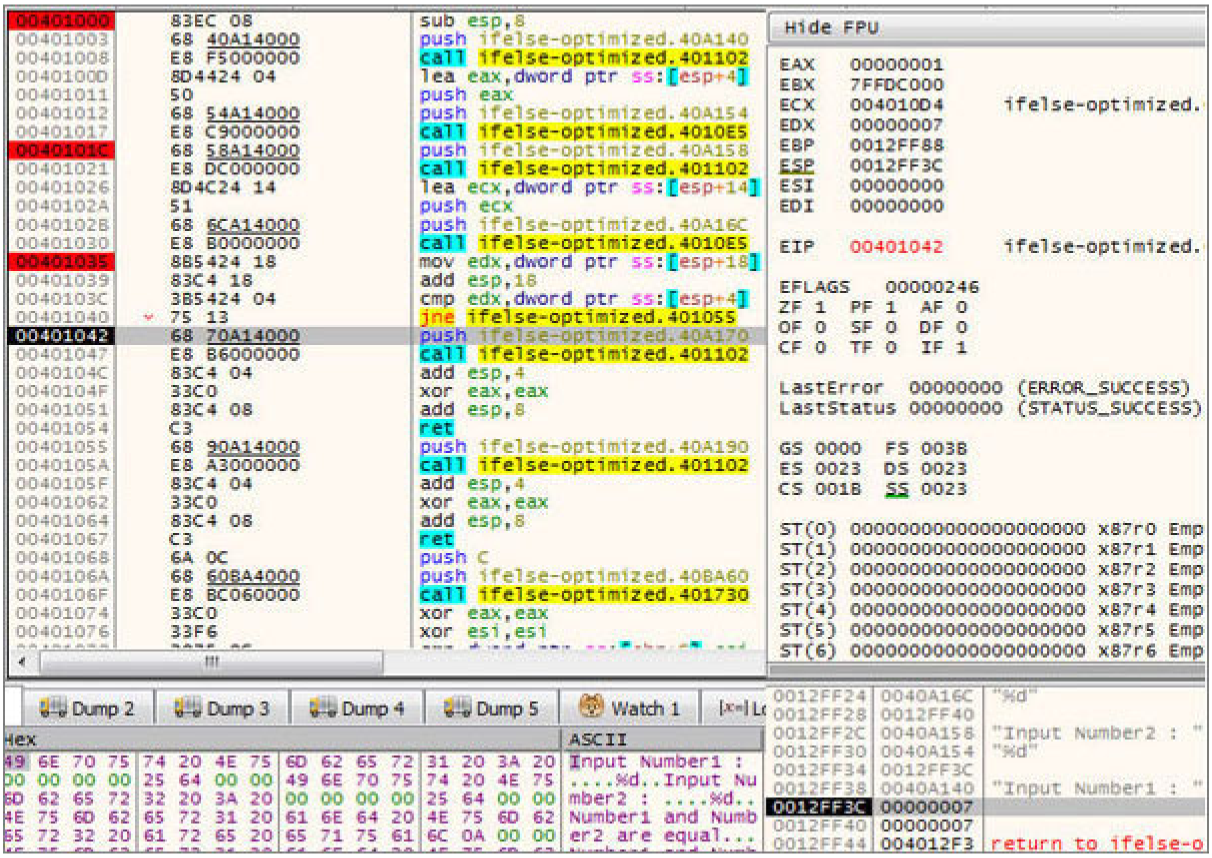
[ESP-0x8] 0012FF34 0012FF3C JUNK

[ESP-0x4] 0012FF38 0040A140 JUNK

[ESP] 0012FF3C 00000007 7 is stored here, Input placeholder for iNumber1

[ESP+0x4] 0012FF40 00000007 7 is stored here, Input placeholder for iNumber2

[ESP+0x8] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000



```
00401000    83EC 08         sub esp,8
00401003    68 40A14000     push ifelse-optimized.40A140
00401008    E8 F5000000     call ifelse-optimized.401102
0040100D    8D4424 04       lea eax,dword ptr ss:[esp+4]
00401011    50              push eax
00401012    68 54A14000     push ifelse-optimized.40A154
00401017    E8 C9000000     call ifelse-optimized.4010E5
0040101C    68 58A14000     push ifelse-optimized.40A158
00401021    E8 DC000000     call ifelse-optimized.401102
00401026    8D4C24 14       lea ecx,dword ptr ss:[esp+14]
0040102A    51              push ecx
0040102B    68 6CA14000     push ifelse-optimized.40A16C
00401030    E8 B0000000     call ifelse-optimized.4010E5
00401035    8B5424 18       mov edx,dword ptr ss:[esp+18]
00401039    83C4 18         add esp,18
0040103C    3B5424 04       cmp edx,dword ptr ss:[esp+4]
00401040    75 13           jne ifelse-optimized.401055
00401042    68 70A14000     push ifelse-optimized.40A170
00401047    E8 B6000000     call ifelse-optimized.401102
0040104C    83C4 04         add esp,4
0040104F    33C0            xor eax,eax
00401051    83C4 08         add esp,8
00401054    C3              ret
00401055    68 90A14000     push ifelse-optimized.40A190
0040105A    E8 A3000000     call ifelse-optimized.401102
0040105F    83C4 04         add esp,4
00401062    33C0            xor eax,eax
00401064    83C4 08         add esp,8
00401067    C3              ret
00401068    6A 0C           push C
0040106A    68 60BA4000     push ifelse-optimized.40BA60
0040106F    E8 BC060000     call ifelse-optimized.401730
00401074    33C0            xor eax,eax
00401076    33F6            xor esi,esi
```

```
EAX   00000001
EBX   7FFDC000
ECX   004010D4     ifelse-optimized.
EDX   00000007
EBP   0012FF88
ESP   0012FF3C
ESI   00000000
EDI   00000000

EIP   00401042     ifelse-optimized.

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Emp
ST(1) 00000000000000000000 x87r1 Emp
ST(2) 00000000000000000000 x87r2 Emp
ST(3) 00000000000000000000 x87r3 Emp
ST(4) 00000000000000000000 x87r4 Emp
ST(5) 00000000000000000000 x87r5 Emp
ST(6) 00000000000000000000 x87r6 Emp
```

```
49 6E 70 75 74 20 4E 75 6D 62 65 72 31 20 3A 20  Input Number1 :
00 00 00 00 25 64 00 00 49 6E 70 75 74 20 4E 75  ....%d..Input Nu
6D 62 65 72 32 20 3A 20 00 00 00 00 25 64 00 00  mber2 : ....%d..
4E 75 6D 62 65 72 31 20 61 6E 64 20 4E 75 6D 62  Number1 and Numb
65 72 32 20 61 72 65 20 65 71 75 61 6C 0A 00 00  er2 are equal...
```

```
0012FF24  0040A16C  "%d"
0012FF28  0012FF40
0012FF2C  0040A158  "Input Number2 : "
0012FF30  0040A154  "%d"
0012FF34  0012FF3C
0012FF38  0040A140  "Input Number1 : "
0012FF3C  00000007
0012FF40  00000007
0012FF44  004012F3  return to ifelse-o
```

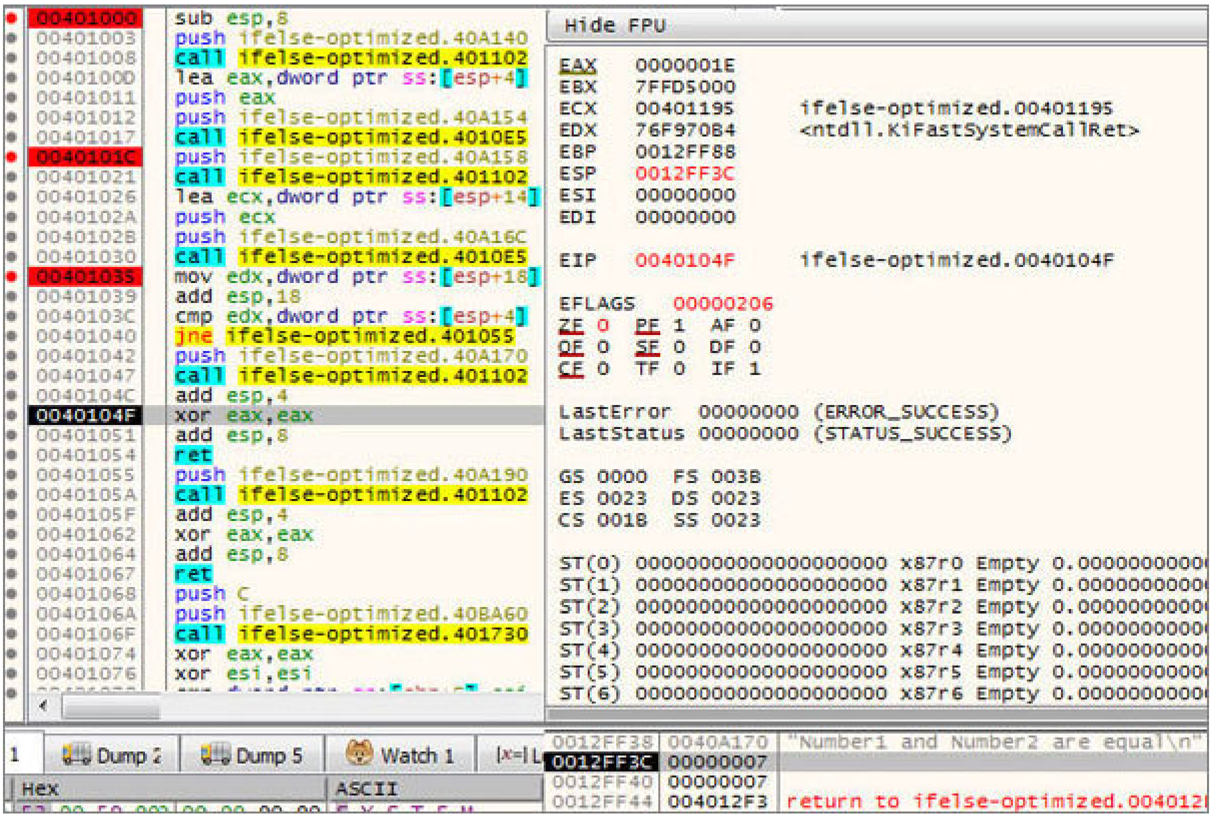
Figure 9.19: *Stack state after JNE*

▼Line 57-61

```
; Line 15
push OFFSET $SG4684
; Line 17
call _printf
add esp, 4
```

This will push the string constant **$SG4684** on the stack for the **printf** function to print on the console. The **ADD** instruction will perform the stack cleaning. Refer to the stack state and x32dbg after the **ADD** instruction in the following screenshot:

[ESP-0x4] 0012FF38 0040A170 Now JUNK, "Number1 and Number2 are equal\n"
[ESP] 0012FF3C 00000007 7 is stored here, Input placeholder for iNumber1
[ESP+0x4] 0012FF40 00000007 7 is stored here, Input placeholder for iNumber2
[ESP+0x8] 0012FF44 004012F3 return to 0x004012F3 from 0x00401000



```
00401000  sub esp,8
00401003  push ifelse-optimized.40A140
00401008  call ifelse-optimized.401102
0040100D  lea eax,dword ptr ss:[esp+4]
00401011  push eax
00401012  push ifelse-optimized.40A154
00401017  call ifelse-optimized.4010E5
0040101C  push ifelse-optimized.40A158
00401021  call ifelse-optimized.401102
00401026  lea ecx,dword ptr ss:[esp+14]
0040102A  push ecx
0040102B  push ifelse-optimized.40A16C
00401030  call ifelse-optimized.4010E5
00401035  mov edx,dword ptr ss:[esp+18]
00401039  add esp,18
0040103C  cmp edx,dword ptr ss:[esp+4]
00401040  jne ifelse-optimized.401055
00401042  push ifelse-optimized.40A170
00401047  call ifelse-optimized.401102
0040104C  add esp,4
0040104F  xor eax,eax
00401051  add esp,8
00401054  ret
00401055  push ifelse-optimized.40A190
0040105A  call ifelse-optimized.401102
0040105F  add esp,4
00401062  xor eax,eax
00401064  add esp,8
00401067  ret
00401068  push C
0040106A  push ifelse-optimized.40BA60
0040106F  call ifelse-optimized.401730
00401074  xor eax,eax
00401076  xor esi,esi
```

```
EAX    0000001E
EBX    7FFD5000
ECX    00401195     ifelse-optimized.00401195
EDX    76F970B4     <ntdll.KiFastSystemCallRet>
EBP    0012FF88
ESP    0012FF3C
ESI    00000000
EDI    00000000

EIP    0040104F     ifelse-optimized.0040104F

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.0000000000
ST(1) 00000000000000000000 x87r1 Empty 0.0000000000
ST(2) 00000000000000000000 x87r2 Empty 0.0000000000
ST(3) 00000000000000000000 x87r3 Empty 0.0000000000
ST(4) 00000000000000000000 x87r4 Empty 0.0000000000
ST(5) 00000000000000000000 x87r5 Empty 0.0000000000
ST(6) 00000000000000000000 x87r6 Empty 0.0000000000
```

```
0012FF38  0040A170  "Number1 and Number2 are equal\n"
0012FF3C  00000007
0012FF40  00000007
0012FF44  004012F3  return to ifelse-optimized.004012
```

Figure 9.20: *Stack state after the ADD instruction*

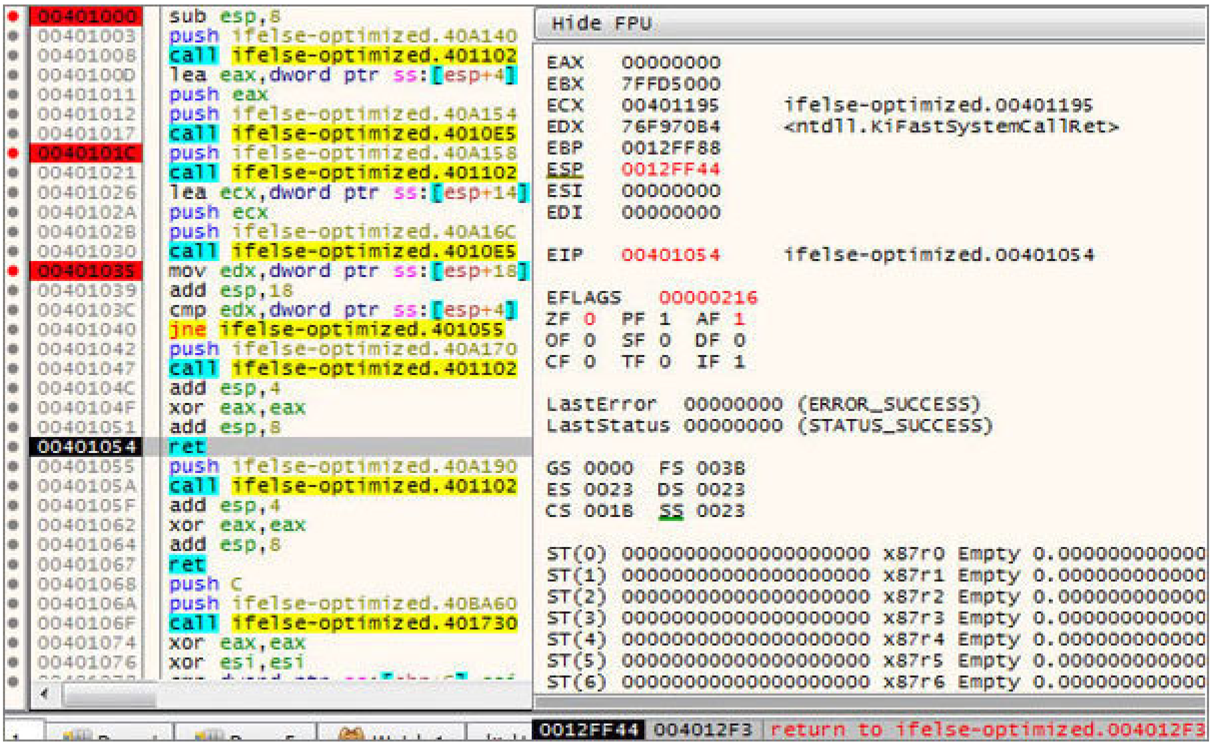
▼Line 62-65

; Line 18

```
xor eax, eax
add esp, 8
ret 0
_main ENDP
_TEXT ENDS
END
```

XOR and the function epilogue will clean up the **EAX** and stack, respectively. **END** will end the code by returning 0. Refer to the stack state and x32dbg after the stack cleaning in the following screenshot:

```
[ESP-0xC] 0012FF38   0040A170   Now JUNK, "Number1 and Number2 are equal\n"
[ESP-0x8] 0012FF3C   00000007   Now JUNK, iNumber1 number 7 entered was stored here
[ESP-0x4] 0012FF40   00000007   Now JUNK, iNumber1 number 7 entered was stored here
[ESP]     0012FF44   004012F3   return to 0x004012F3 from 0x00401000
```



```
00401000  sub esp,8
00401003  push ifelse-optimized.40A140
00401008  call ifelse-optimized.401102
0040100D  lea eax,dword ptr ss:[esp+4]
00401011  push eax
00401012  push ifelse-optimized.40A154
00401017  call ifelse-optimized.4010E5
0040101C  push ifelse-optimized.40A158
00401021  call ifelse-optimized.401102
00401026  lea ecx,dword ptr ss:[esp+14]
0040102A  push ecx
0040102B  push ifelse-optimized.40A16C
00401030  call ifelse-optimized.4010E5
00401035  mov edx,dword ptr ss:[esp+18]
00401039  add esp,18
0040103C  cmp edx,dword ptr ss:[esp+4]
00401040  jne ifelse-optimized.401055
00401042  push ifelse-optimized.40A170
00401047  call ifelse-optimized.401102
0040104C  add esp,4
0040104F  xor eax,eax
00401051  add esp,8
00401054  ret
00401055  push ifelse-optimized.40A190
0040105A  call ifelse-optimized.401102
0040105F  add esp,4
00401062  xor eax,eax
00401064  add esp,8
00401067  ret
00401068  push C
0040106A  push ifelse-optimized.40BA60
0040106F  call ifelse-optimized.401730
00401074  xor eax,eax
00401076  xor esi,esi
```

```
EAX    00000000
EBX    7FFD5000
ECX    00401195     ifelse-optimized.00401195
EDX    76F970B4     <ntdll.KiFastSystemCallRet>
EBP    0012FF88
ESP    0012FF44
ESI    00000000
EDI    00000000

EIP    00401054     ifelse-optimized.00401054

EFLAGS    00000216
ZF 0   PF 1   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000
```

```
0012FF44 004012F3 return to ifelse-optimized.004012F3
```

Figure 9.21: *Stack cleaned*

Conclusion

In this chapter, we learned the concept of assembly instructions used to make decisions in a program flow. We also learned about the usage of CMP and Jump instructions in assembly code. Along with this, we also understood the differentiating pattern between optimized and non-optimized assembly code of decision control structures. In the next chapter, we will learn about the disassembly of programs with looping control statements.

CHAPTER 10

Reverse Engineering Pattern of Loop Control Structure

In the last chapter, we discussed about the programs that are associated with some decision control statements. In our day-to-day life, we tend to do so many things over and over again. Like we get ready every morning and go to sleep in the night around the same time every day. This is where we see our routine happening in a loop every day.

Every programming language is equipped with a similar loop control structure, where the programs are coded in such a way that they run in a loop with some conditions. As a reverse engineer, this topic can be understood from a malware writer's point of view, wherein malware's are coded in such a way that they infect all the files on the target computer. When a malware gets executed, files in the computer are run through a loop condition so as to target the whole computer data. Understanding to identify loop patterns in assembly will help you decode programs using loop control structures. Most of the computer programs are coded with these loop control structures. It is important for us to should understand the pattern of these programs when reverse engineered.

Structure

In this chapter, we will cover the following topics:

Understanding about a loop control structure

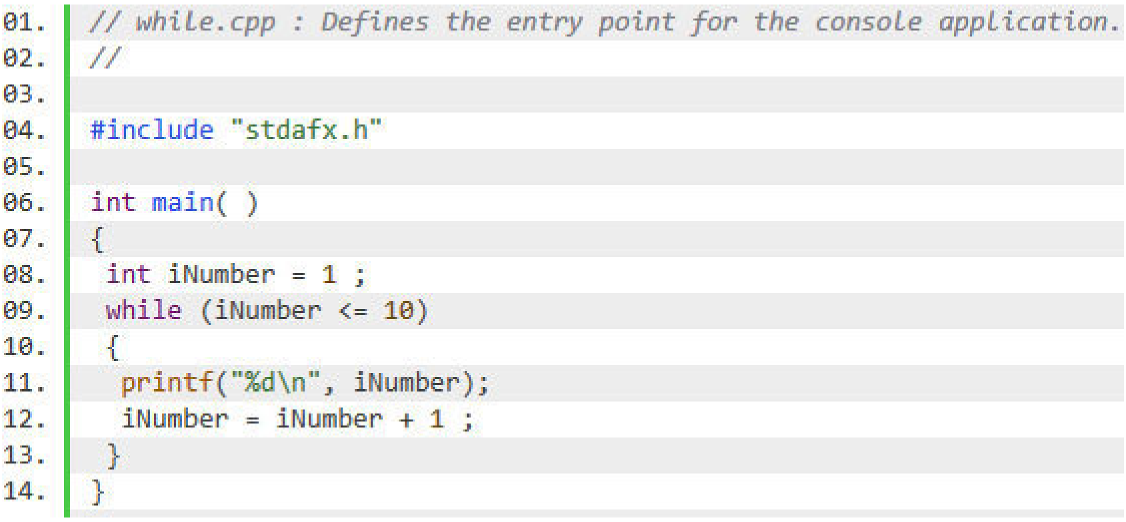
How loops are handled in assembly

Objective

The objective of this chapter is to learn about different loop statements in C/C++ and understanding the code pattern in a disassembled code. We will also learn to put a conditional breakpoint in code execution. In an assembly listing generated from loop statement programs, we will check the **CMP** and **JMP** instruction patterns in assembly and note their differences in optimized and non-optimized code.

While Condition

In this C/C++ code, we will use the **while** condition to print numbers from 1 to 10 on the console/screen. This will also include the **printf** function in our C/C++ code.



```
// while.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int main( )
{
 int iNumber = 1 ;
 while (iNumber <= 10)
 {
  printf("%d\n", iNumber);
  iNumber = iNumber + 1 ;
 }
}
```

Figure 10.1: while.cpp

While condition without Optimization

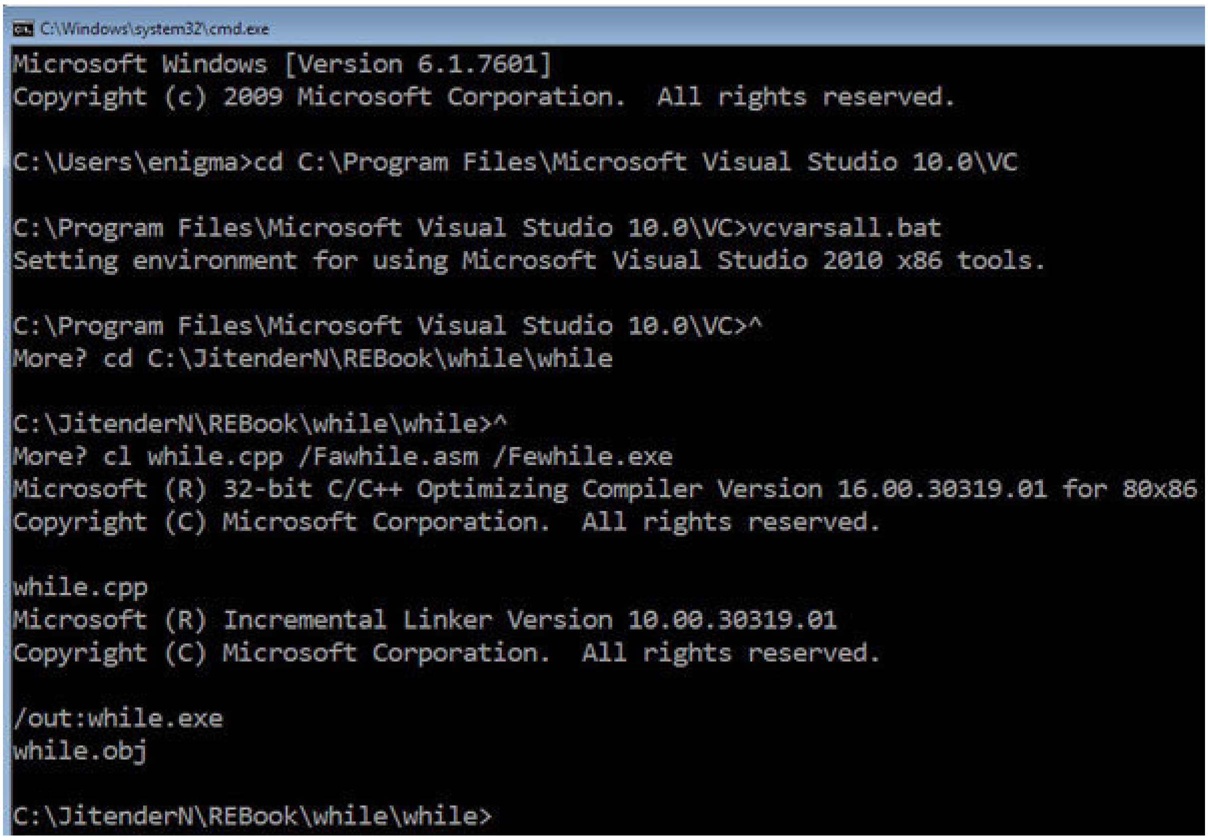
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\while\while

C:\JitenderN\REBook\while\while>^
More? cl while.cpp /Fawhile.asm /Fewhile.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

while.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:while.exe
while.obj

C:\JitenderN\REBook\while\while>
```

Figure 10.2: *While condition without optimization*

This will generate the assembly code and the EXE file. For further analysis, we have disabled the ASLR manually. To disable ASLR, follow the same steps by using the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can** For step-by-step instructions, refer to the

Now let’s move on to the generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\while\while\while.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4681 DB '%d', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_iNumber$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\while\while\while.cpp
; Line 7
 push ebp
 mov ebp, esp
 push ecx
; Line 8
 mov DWORD PTR _iNumber$[ebp], 1
$LN2@main:
; Line 9
 cmp DWORD PTR _iNumber$[ebp], 10  ; 0000000aH
 jg SHORT $LN3@main
; Line 11
 mov eax, DWORD PTR _iNumber$[ebp]
 push eax
 push OFFSET $SG4681
 call _printf
 add esp, 8
; Line 12
 mov ecx, DWORD PTR _iNumber$[ebp]
 add ecx, 1
 mov DWORD PTR _iNumber$[ebp], ecx
; Line 13
 jmp SHORT $LN2@main
$LN3@main:
; Line 14
 xor eax, eax
 mov esp, ebp
 pop ebp
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 10.3: *while.asm*

Let's walk through the assembly code line by line. As we have discussed most of the initial instructions of the assembly listing in the earlier chapters, we will move on to the instructions from the start of the **main** procedure.

▼Line 22-25

```
; Line 7
push ebp
mov ebp, esp
push ecx
```

The assembly code starts with the **main** function prologue first with the **PUSH** instruction and then with the **MOV** instruction. **ECX** is pushed onto the stack to create room for the **iNumber** integer variable of size 4 bytes.

▼Line 26-27

```
; Line 8
mov DWORD PTR _iNumber$[ebp], 1
```

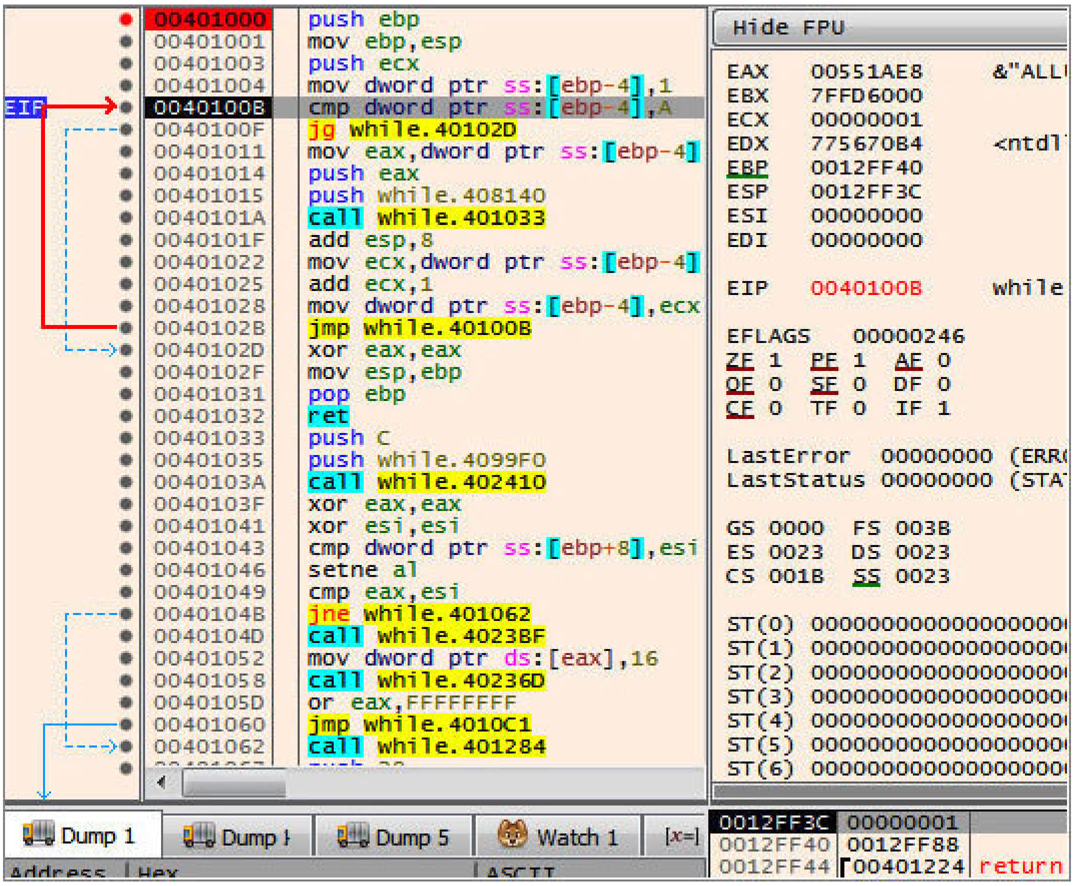
The C/C++ code for this is as follows:

```
int iNumber = 1 ;
```

The **iNumber** integer variable is initialized to 1.

In the ASM code, we created room for the **iNumber** local variable to **main** function by pushing **ECX** onto the stack. 1 is moved to this memory location on stack **[EBP-4]** (We get this by adding **iNumber** macro, which is **_iNumber$ = -4** with Let's check the stack state by placing the breakpoint at the start of the **main** procedure and then stepping into the instructions. Refer to the following stack state and a screenshot of x32dbg:

[ESP] 0012FF3C 00000001 iNumber is stored here
[ESP+0x4] 0012FF40 0012FF88 [EBP]
[ESP+0x4] 0012FF44 00401224 return to while.00401224 from while.00401000



```
00401000   push ebp
00401001   mov ebp,esp
00401003   push ecx
00401004   mov dword ptr ss:[ebp-4],1
0040100B   cmp dword ptr ss:[ebp-4],A
0040100F   jg while.40102D
00401011   mov eax,dword ptr ss:[ebp-4]
00401014   push eax
00401015   push while.408140
0040101A   call while.401033
0040101F   add esp,8
00401022   mov ecx,dword ptr ss:[ebp-4]
00401025   add ecx,1
00401028   mov dword ptr ss:[ebp-4],ecx
0040102B   jmp while.40100B
0040102D   xor eax,eax
0040102F   mov esp,ebp
00401031   pop ebp
00401032   ret
00401033   push C
00401035   push while.4099F0
0040103A   call while.402410
0040103F   xor eax,eax
00401041   xor esi,esi
00401043   cmp dword ptr ss:[ebp+8],esi
00401046   setne al
00401049   cmp eax,esi
0040104B   jne while.401062
0040104D   call while.4023BF
00401052   mov dword ptr ds:[eax],16
00401058   call while.40236D
0040105D   or eax,FFFFFFFF
00401060   jmp while.4010C1
00401062   call while.401284
```

```
EAX    00551AE8     &"ALL
EBX    7FFD6000
ECX    00000001
EDX    775670B4     <ntdl
EBP    0012FF40
ESP    0012FF3C
ESI    00000000
EDI    00000000

EIP    0040100B     while

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF3C 00000001
0012FF40 0012FF88
0012FF44 00401224 return
```

Figure 10.4: *iNumber is stored on stack*

▼Line 28-31

```
$LN2@main:
; Line 9
cmp DWORD PTR _iNumber$[ebp], 10   ; 0000000aH
jg SHORT $LN3@main
```

The C/C++ code for this is as follows:

```
while (iNumber <= 10)
```

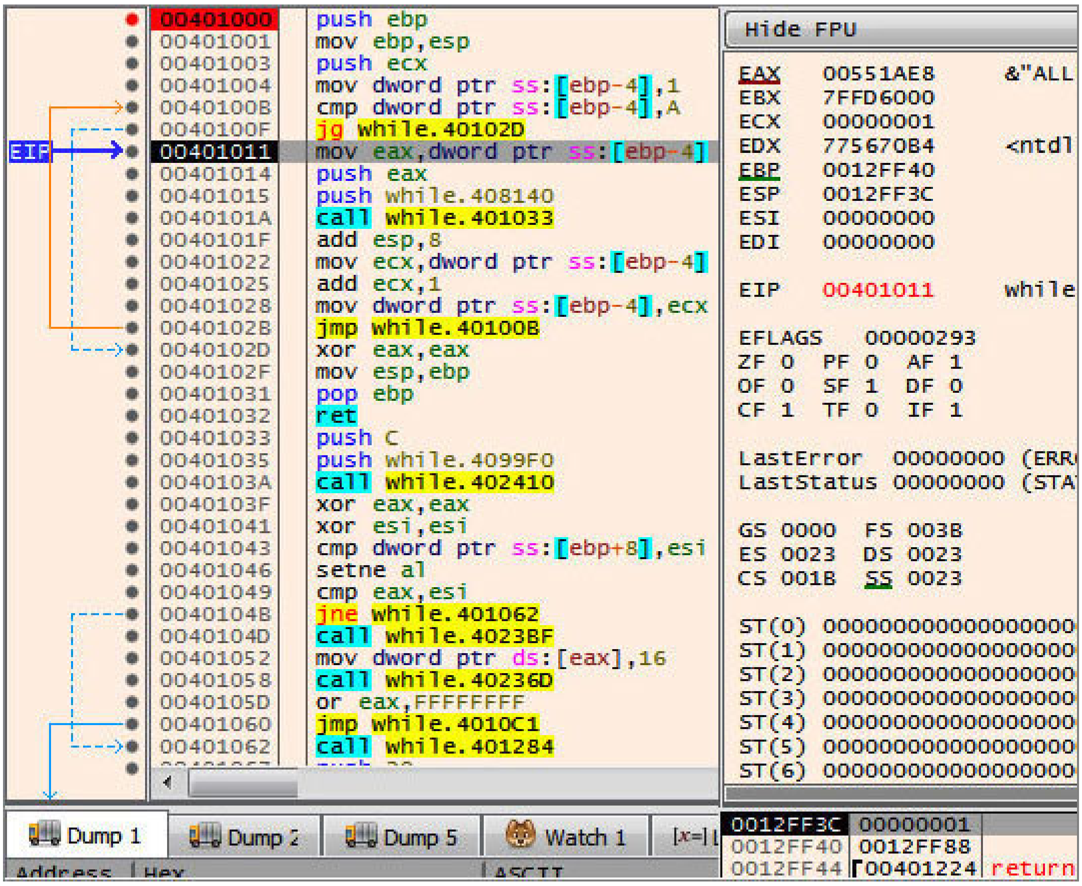
As we see in the C/C++ code **while** condition checks **iNumber** value, for less than or equal to 10 (0x0A). In the ASM code, the value stored on the stack at **[EBP-4]** is compared with the value 10 with the **CMP** instruction. As both the operands are not equal in the current state, executing the **CMP** instruction will set ZF=0.

When the value at **[EBP-4]** will be greater than 10, then **JG (Jump Greater)** will jump to the label **$LN3@main** and the label **$LN3@main** will point towards the closure of the **main** function.

Considering the current case where **iNumber** is 1, the stack state and the screenshot of x32dbg will be as follows:

```
[ESP]     0012FF3C   00000001   iNumber is stored here
[ESP+0x4] 0012FF40   0012FF88   [EBP]
```

[ESP+0x4] 0012FF44 00401224 return to while.00401224 from while.00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  cmp dword ptr ss:[ebp-4],A
0040100F  jg while.40102D
00401011  mov eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push while.408140
0040101A  call while.401033
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.40100B
0040102D  xor eax,eax
0040102F  mov esp,ebp
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while.4099F0
0040103A  call while.402410
0040103F  xor eax,eax
00401041  xor esi,esi
00401043  cmp dword ptr ss:[ebp+8],esi
00401046  setne al
00401049  cmp eax,esi
0040104B  jne while.401062
0040104D  call while.4023BF
00401052  mov dword ptr ds:[eax],16
00401058  call while.40236D
0040105D  or eax,FFFFFFFF
00401060  jmp while.4010C1
00401062  call while.401284
```

```
EAX   00551AE8     &"ALL
EBX   7FFD6000
ECX   00000001
EDX   775670B4     <ntdl
EBP   0012FF40
ESP   0012FF3C
ESI   00000000
EDI   00000000

EIP   00401011     while

EFLAGS   00000293
ZF 0  PF 0  AF 1
OF 0  SF 1  DF 0
CF 1  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF3C  00000001
0012FF40  0012FF88
0012FF44  00401224  return
```

Figure 10.5: After compare instruction when iNumber=1

▼Line 32-37

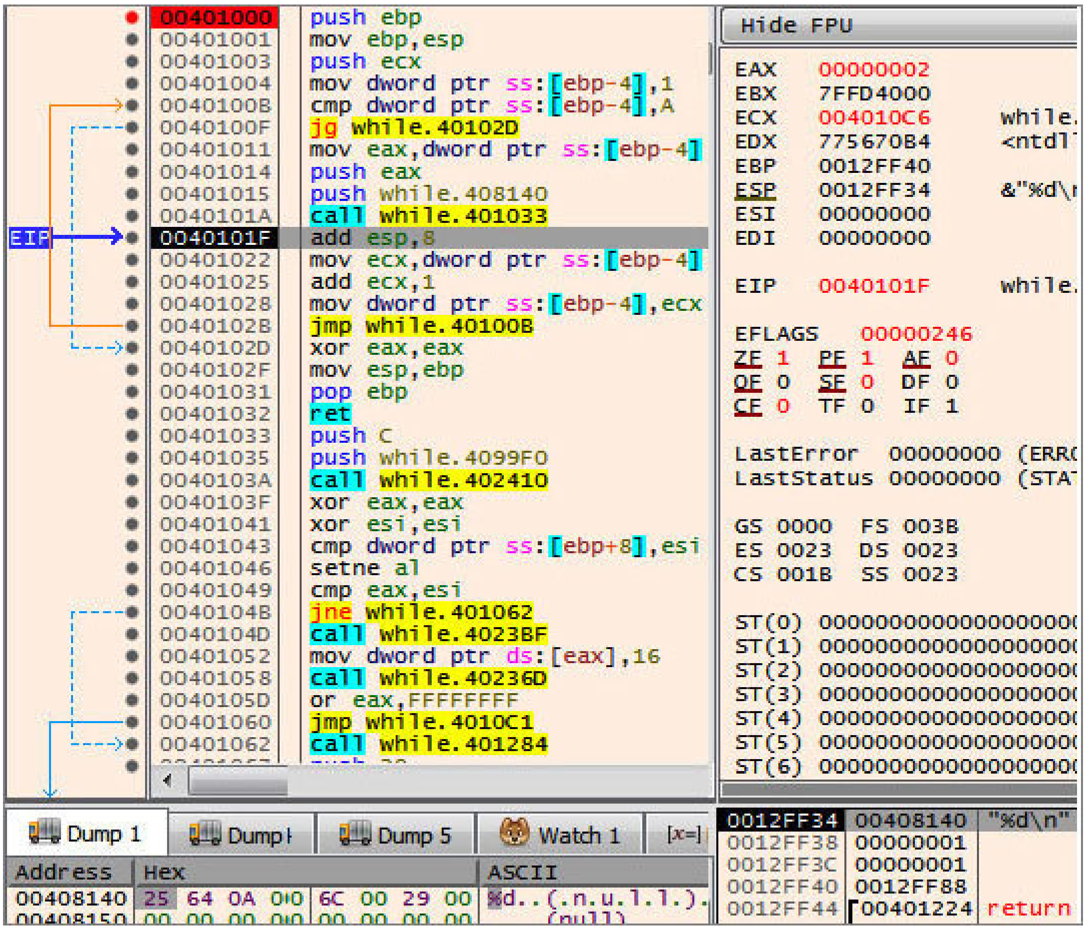
```
; Line 11
mov eax, DWORD PTR _iNumber$[ebp]
push eax
push OFFSET $SG4681

call _printf
add esp, 8
```

On line 11 of the C/C++ code is the **printf** function to print **iNumber** on the console.

The ASM code moves the value stored on the stack at **[EBP-4]** to the **EAX** register, which is later in the next instruction pushed onto the stack along with the other argument, string constant When both the arguments to the **printf** function are passed onto the stack, the call to **printf** is made. On return from the **printf** function, **ADD** is executed to clean the stack by adding 8 bytes to

The stack state and x32dbg screenshot before the **ADD** instruction will be as follows:

[ESP] 0012FF34 00408140 "%d\n", as a parameter to printf()
[ESP+0x4] 0012FF38 00000001 iNumber is pushed as a parameter to printf()
[ESP+0x8] 0012FF3C 00000001 iNumber is stored here
[ESP+0xC] 0012FF40 0012FF88 [EBP]
[ESP+0x10] 0012FF44 00401224 return to while.00401224 from while.00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  cmp dword ptr ss:[ebp-4],A
0040100F  jg while.40102D
00401011  mov eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push while.408140
0040101A  call while.401033
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.40100B
0040102D  xor eax,eax
0040102F  mov esp,ebp
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while.4099F0
0040103A  call while.402410
0040103F  xor eax,eax
00401041  xor esi,esi
00401043  cmp dword ptr ss:[ebp+8],esi
00401046  setne al
00401049  cmp eax,esi
0040104B  jne while.401062
0040104D  call while.4023BF
00401052  mov dword ptr ds:[eax],16
00401058  call while.40236D
0040105D  or eax,FFFFFFFF
00401060  jmp while.4010C1
00401062  call while.401284
```

```
EAX    00000002
EBX    7FFD4000
ECX    004010C6      while.
EDX    775670B4      <ntdl
EBP    0012FF40
ESP    0012FF34      &"%d\
ESI    00000000
EDI    00000000

EIP    0040101F      while.

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
Address   Hex                       ASCII
00408140  25 64 0A 00 6C 00 29 00   %d..(.n.u.l.l.).
00408150  00 00 00 00 00 00 00 00     (null)
```

```
0012FF34  00408140  "%d\n"
0012FF38  00000001
0012FF3C  00000001
0012FF40  0012FF88
0012FF44  00401224  return
```

Figure 10.6: After printf

▼Line 38-41

```
; Line 12
mov ecx, DWORD PTR _iNumber$[ebp]
add ecx, 1
mov DWORD PTR _iNumber$[ebp], ecx
```

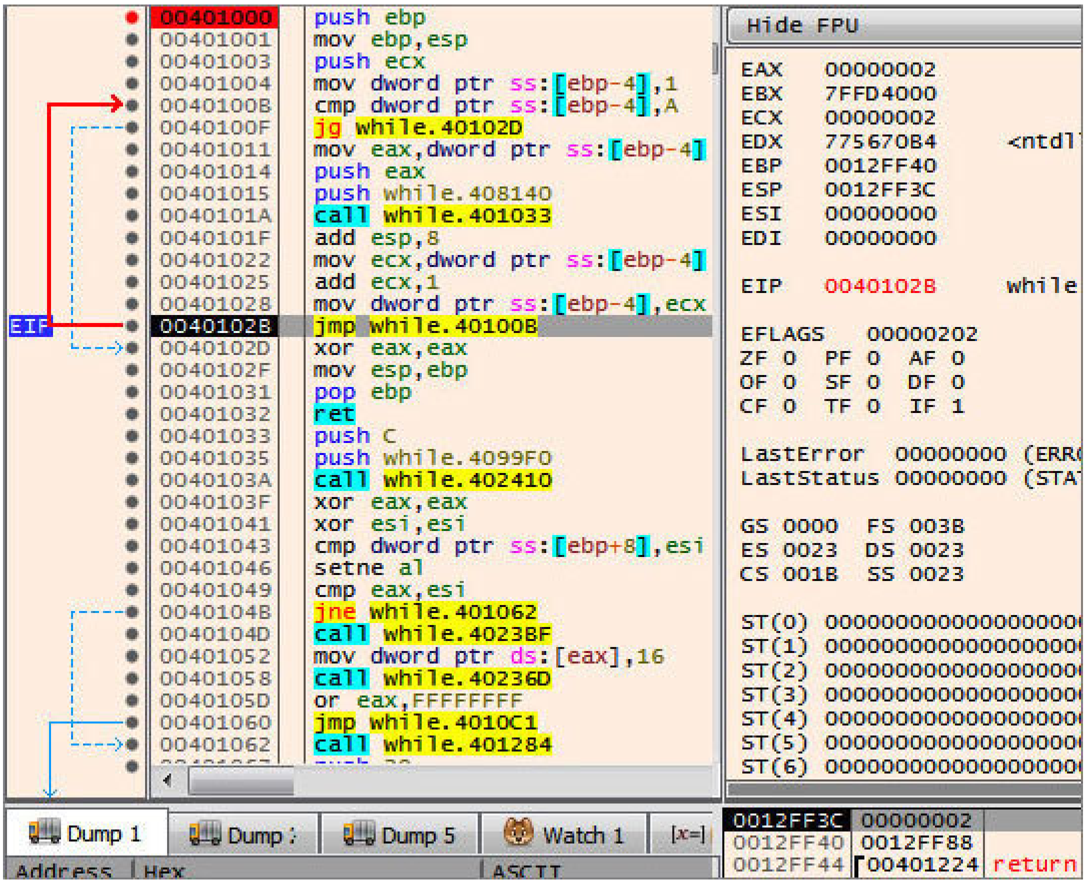
Line 12 of the C/C++ code increments the **iNumber** value by 1.

```
iNumber = iNumber + 1 ;
```

In ASM, we are incrementing the **iNumber** value by first moving the **iNumber** value stored at **[EBP-4]** to and then incrementing **ECX** by 1. Once incremented, we will move **ECX** back to **[EBP-4]** memory location. This process or iteration is repeated every time until **iNumber** is less than or equal to 10 (0x0A).

The stack state and the screenshot of x32dbg when **iNumber** is incremented by 1 will be as follows:

[ESP] 0012FF3C 00000002 iNumber is stored here, now its incremented by 1
[ESP+0x4] 0012FF40 0012FF88 [EBP]
[ESP+0x4] 0012FF44 00401224 return to while.00401224 from while.00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  cmp dword ptr ss:[ebp-4],A
0040100F  jg while.40102D
00401011  mov eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push while.408140
0040101A  call while.401033
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.40100B
0040102D  xor eax,eax
0040102F  mov esp,ebp
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while.4099F0
0040103A  call while.402410
0040103F  xor eax,eax
00401041  xor esi,esi
00401043  cmp dword ptr ss:[ebp+8],esi
00401046  setne al
00401049  cmp eax,esi
0040104B  jne while.401062
0040104D  call while.4023BF
00401052  mov dword ptr ds:[eax],16
00401058  call while.40236D
0040105D  or eax,FFFFFFFF
00401060  jmp while.4010C1
00401062  call while.401284
```

```
EAX    00000002
EBX    7FFD4000
ECX    00000002
EDX    775670B4      <ntdl
EBP    0012FF40
ESP    0012FF3C
ESI    00000000
EDI    00000000

EIP    0040102B      while

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 000000000000000000
ST(1) 000000000000000000
ST(2) 000000000000000000
ST(3) 000000000000000000
ST(4) 000000000000000000
ST(5) 000000000000000000
ST(6) 000000000000000000
```

```
0012FF3C  00000002
0012FF40  0012FF88
0012FF44  00401224  return
```

Figure 10.7: *When iNumber is incremented by 1*

▼Line 42-43

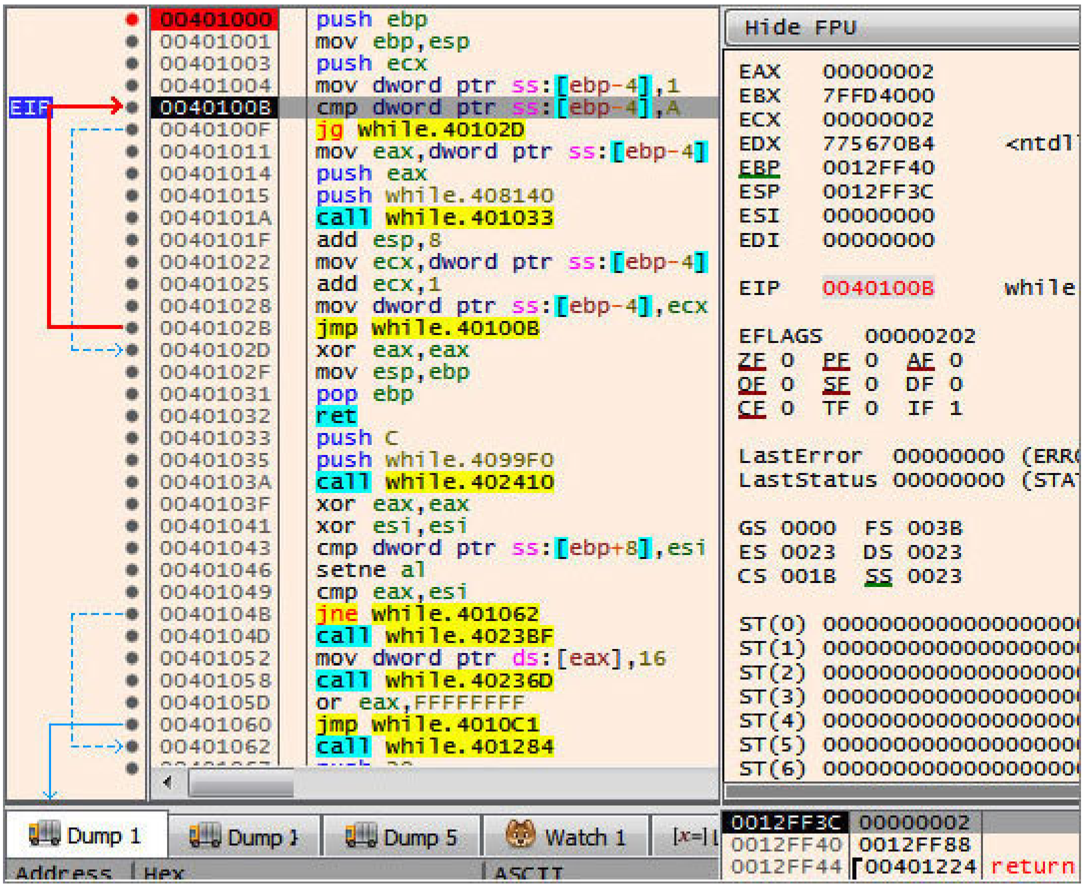
```
; Line 13
jmp SHORT $LN2@main
```

This is an unconditional jump to the **$LN2@main** label, where the value stored at **[EBP-4]** is again compared with the value 10 (0x0A). If the value is less than or equal to 10, then the same instructions will print **iNumber** on screen, along with incrementing the **iNumber** value at **[EBP-4]** by 1.

The stack state will be the same as earlier after

[ESP] 0012FF3C 00000002 iNumber is stored here, now its incremented by 1
[ESP+0x4] 0012FF40 0012FF88 [EBP]
[ESP+0x4] 0012FF44 00401224 return to while.00401224 from while.00401000

Instruction pointer will be pointed to **0x0040100B** as follows:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  cmp dword ptr ss:[ebp-4],A
0040100F  jg while.40102D
00401011  mov eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push while.408140
0040101A  call while.401033
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.40100B
0040102D  xor eax,eax
0040102F  mov esp,ebp
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while.4099F0
0040103A  call while.402410
0040103F  xor eax,eax
00401041  xor esi,esi
00401043  cmp dword ptr ss:[ebp+8],esi
00401046  setne al
00401049  cmp eax,esi
0040104B  jne while.401062
0040104D  call while.4023BF
00401052  mov dword ptr ds:[eax],16
00401058  call while.40236D
0040105D  or eax,FFFFFFFF
00401060  jmp while.4010C1
00401062  call while.401284
```

```
EAX     00000002
EBX     7FFD4000
ECX     00000002
EDX     775670B4      <ntdl
EBP     0012FF40
ESP     0012FF3C
ESI     00000000
EDI     00000000

EIP     0040100B      while

EFLAGS     00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF3C  00000002
0012FF40  0012FF88
0012FF44  00401224  return
```

__Figure 10.8:__ unconditional jump to $LN2@main

Now, we will see what happens when **iNumber** becomes 10 (0x0A). There are two ways to analyze the code:

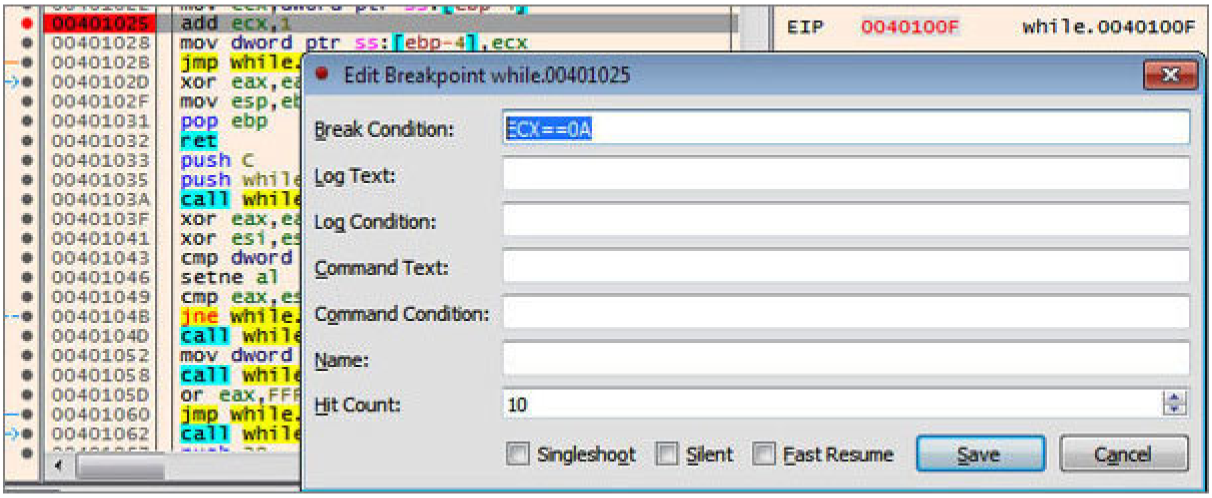
Manually **step into** the code instruction by instruction. This is a time-consuming process.

Set the breakpoint on the condition when either the value stored at **[EBP-4]** becomes **0x0A** or **ECX** == We will set the conditional breakpoint in x32dbg, when **ECX** ==

Now, to set the conditional breakpoint, set the software breakpoint (press key first at:

0x00401025 add ecx, 1

Then right-click on the instruction and select the **Edit Breakpoint** command from the context menu. Fill in the following conditional expression and then confirm and close the dialog box, as shown in the following screenshot:



```
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.
0040102D  xor eax,ea
0040102F  mov esp,eb
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while
0040103A  call while
0040103F  xor eax,ea
00401041  xor esi,es
00401043  cmp dword
00401046  setne al
00401049  cmp eax,es
0040104B  jne while.
0040104D  call while
00401052  mov dword
00401058  call while
0040105D  or eax,FFF
00401060  jmp while.
00401062  call while
```

```
EIP  0040100F   while.0040100F
```

Edit Breakpoint while.00401025

Break Condition: ECX==0A

Log Text:

Log Condition:

Command Text:

Command Condition:

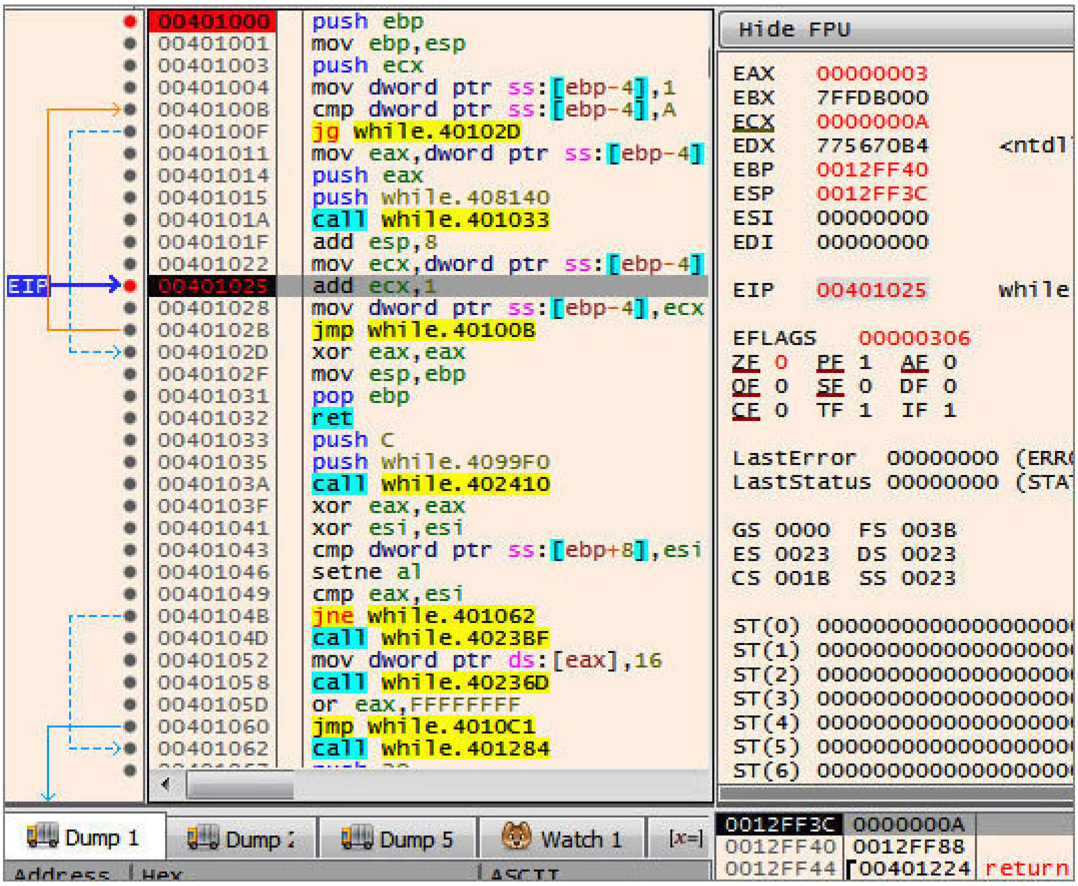
Name:

Hit Count: 10

Figure 10.9: *Set the breakpoint on the condition*

Run the code until you hit the conditional breakpoint. From there, we can manually **step into** the instructions. The stack state and the screenshot of x32dbg at this point will be as follows:

[ESP] 0012FF3C 0000000A iNumber is stored here, now its incremented to 0xA
[ESP+0x4] 0012FF40 0012FF88 [EBP]
[ESP+0x4] 0012FF44 00401224 return to while.00401224 from while.00401000



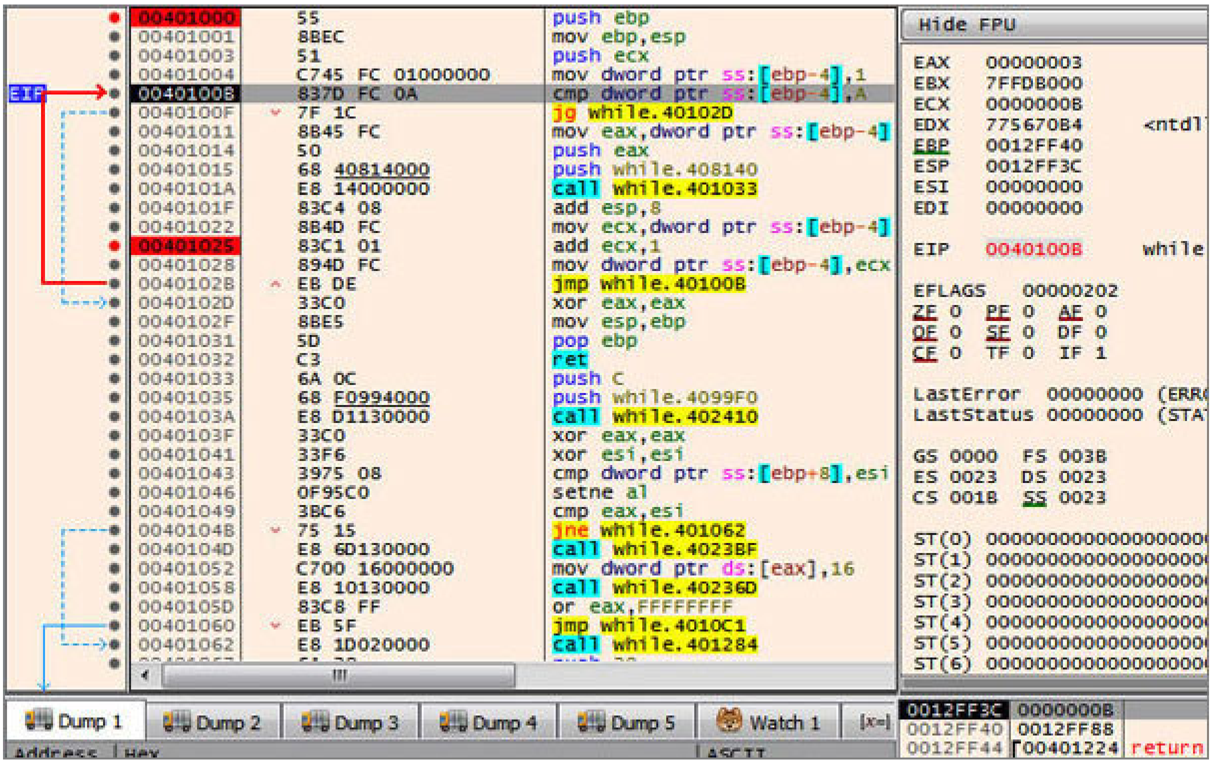
```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  cmp dword ptr ss:[ebp-4],A
0040100F  jg while.40102D
00401011  mov eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push while.408140
0040101A  call while.401033
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  add ecx,1
00401028  mov dword ptr ss:[ebp-4],ecx
0040102B  jmp while.40100B
0040102D  xor eax,eax
0040102F  mov esp,ebp
00401031  pop ebp
00401032  ret
00401033  push C
00401035  push while.4099F0
0040103A  call while.402410
0040103F  xor eax,eax
00401041  xor esi,esi
00401043  cmp dword ptr ss:[ebp+8],esi
00401046  setne al
00401049  cmp eax,esi
0040104B  jne while.401062
0040104D  call while.4023BF
00401052  mov dword ptr ds:[eax],16
00401058  call while.40236D
0040105D  or eax,FFFFFFFF
00401060  jmp while.4010C1
00401062  call while.401284
```

```
EAX   00000003
EBX   7FFDB000
ECX   0000000A
EDX   775670B4      <ntdl
EBP   0012FF40
ESP   0012FF3C
ESI   00000000
EDI   00000000

EIP   00401025      while

EFLAGS   00000306
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 000000000000000000
ST(1) 000000000000000000
ST(2) 000000000000000000
ST(3) 000000000000000000
ST(4) 000000000000000000
ST(5) 000000000000000000
ST(6) 000000000000000000
```

```
0012FF3C  0000000A
0012FF40  0012FF88
0012FF44  00401224  return
```

Figure 10.10: *When iNumber is incremented to 0x0A*

As we **step ECX** and **[EBP-4]** will become 0x0B.



```
00401000  55                 push ebp
00401001  8BEC               mov ebp,esp
00401003  51                 push ecx
00401004  C745 FC 01000000   mov dword ptr ss:[ebp-4],1
0040100B  837D FC 0A         cmp dword ptr ss:[ebp-4],A
0040100F  7F 1C              jg while.40102D
00401011  8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401014  50                 push eax
00401015  68 40814000        push while.408140
0040101A  E8 14000000        call while.401033
0040101F  83C4 08            add esp,8
00401022  8B4D FC            mov ecx,dword ptr ss:[ebp-4]
00401025  83C1 01            add ecx,1
00401028  894D FC            mov dword ptr ss:[ebp-4],ecx
0040102B  EB DE              jmp while.40100B
0040102D  33C0               xor eax,eax
0040102F  8BE5               mov esp,ebp
00401031  5D                 pop ebp
00401032  C3                 ret
00401033  6A 0C              push C
00401035  68 F0994000        push while.4099F0
0040103A  E8 D1130000        call while.402410
0040103F  33C0               xor eax,eax
00401041  33F6               xor esi,esi
00401043  3975 08            cmp dword ptr ss:[ebp+8],esi
00401046  0F95C0             setne al
00401049  3BC6               cmp eax,esi
0040104B  75 15              jne while.401062
0040104D  E8 6D130000        call while.4023BF
00401052  C700 16000000      mov dword ptr ds:[eax],16
00401058  E8 10130000        call while.40236D
0040105D  83C8 FF            or eax,FFFFFFFF
00401060  EB 5F              jmp while.4010C1
00401062  E8 1D020000        call while.401284
```

```
EAX   00000003
EBX   7FFDB000
ECX   0000000B
EDX   775670B4      <ntdl
EBP   0012FF40
ESP   0012FF3C
ESI   00000000
EDI   00000000

EIP   0040100B      while

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF3C  0000000B
0012FF40  0012FF88
0012FF44  00401224  return
```

__Figure 10.11:__ When iNumber is incremented to 0x0B

In the next iteration, the **JG** instruction will jump the instruction pointer to the **$LN3@main** label, which we will discuss next.

▼Line 44-52

```
$LN3@main:
; Line 14
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

This will zero the **EAX** and call the **main** function epilogue. With this, the TEXT segment and code is ended.

While condition with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

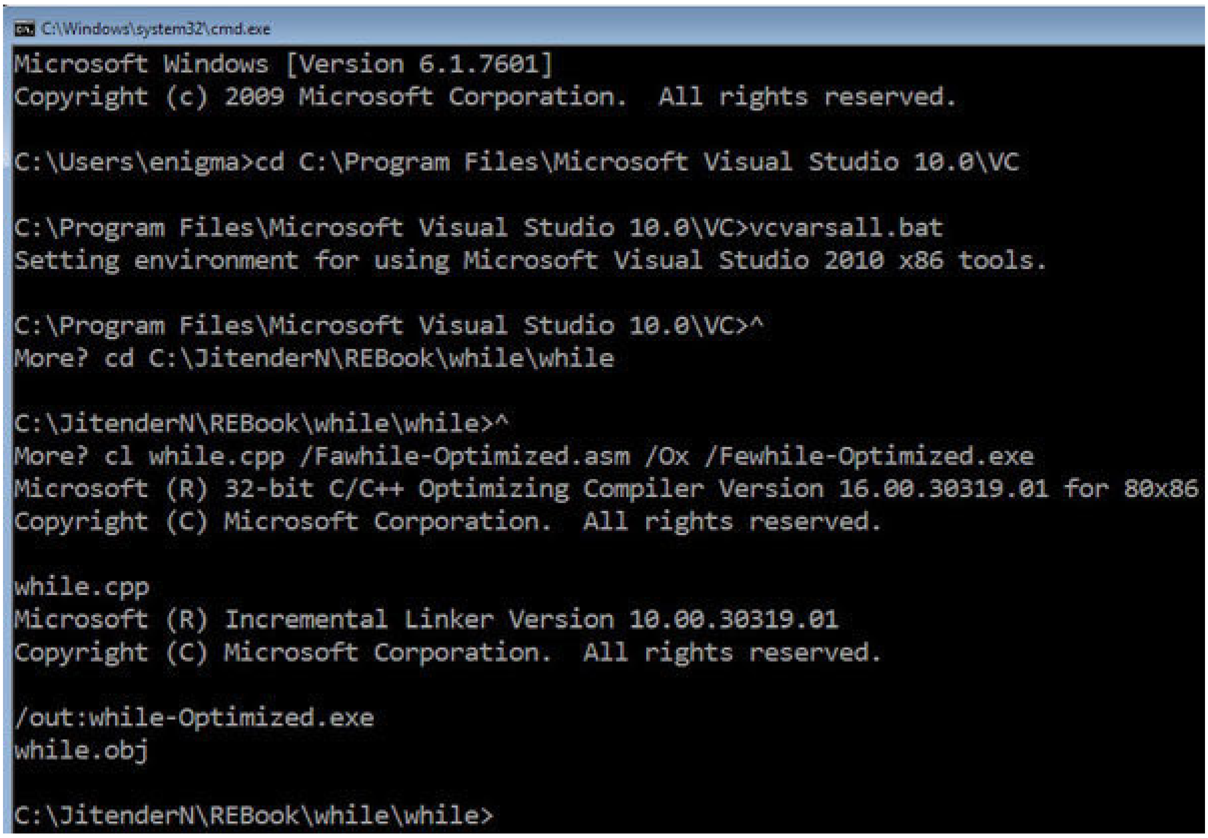
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\while\while

C:\JitenderN\REBook\while\while>^
More? cl while.cpp /Fawhile-Optimized.asm /Ox /Fewhile-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

while.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:while-Optimized.exe
while.obj

C:\JitenderN\REBook\while\while>
```

Figure 10.12: *While condition with Optimization*

It will generate the assembly code and the EXE file. Follow the same process to disable the ASLR manually. To disable ASLR, follow the same steps by using the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can** For a step-by-step process to disable ASLR, refer to the

Now, let’s move on to the generated assembly listing:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\while\while\while.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4681 DB '%d', 0aH, 00H
14. CONST ENDS
15. PUBLIC _main
16. EXTRN _printf:PROC
17. ; Function compile flags: /Ogtpy
18. _TEXT SEGMENT
19. _main PROC
20. ; File c:\jitendern\rebook\while\while\while.cpp
21. ; Line 7
22.  push esi
23. ; Line 8
24.  mov esi, 1
25.  npad 10
26. $LL2@main:
27. ; Line 11
28.  push esi
29.  push OFFSET $SG4681
30.  call _printf
31. ; Line 12
32.  inc esi
33.  add esp, 8
34.  cmp esi, 10      ; 0000000aH
35.  jle SHORT $LL2@main
36. ; Line 14
37.  xor eax, eax
38.  pop esi
39.  ret 0
40. _main ENDP
41. _TEXT ENDS
42. END
```

Figure 10.13: *while-Optimized.asm*

We will walk through the generated assembly code instruction by instruction:

▼Line 21-22

```
; Line 7
push esi
```

The **main** procedure starts by pushing the **ESI** register. By pushing **ESI** onto the stack, we are preserving the **ESI** register value. **ESI** is restored back with **POP ESI** instruction at the end of the **main** procedure. So, it is basically persevering and restoring the register value at the start and end of the function with the use of the **PUSH & POP** instructions, respectively.

▼Line 23-25

```
; Line 8
mov esi, 1
npad 10
```

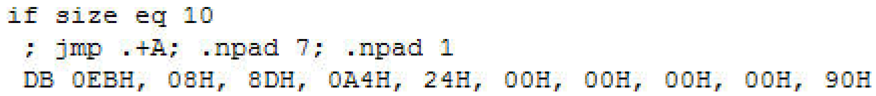
Line 8 of the C/C++ code initializes the **iNumber** variable:

```
int iNumber = 1 ;
```

In the ASM code, the **ESI** register will be used to hold the **iNumber** value and for subsequent operations. The **MOV** instruction initializes **iNumber** by moving 0x01 into the **ESI** register.

Next is the **npad** macro which inserts non-destructive and non-operational instructions. It means that it will insert an instruction that does nothing. For information on please refer to the *Appendix* section. Our assembly listing is using **npad** which is defined in **LISTING.INC** as **jmp .+A; .npad 7; .npad** where

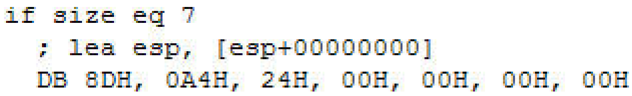
npad 10 is



```
if size eq 10
 ; jmp .+A; .npad 7; .npad 1
 DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
```

Figure 10.14: *npad 10*

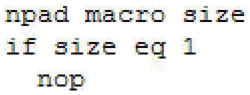
npad 7 is



```
if size eq 7
  ; lea esp, [esp+00000000]
  DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
```

Figure 10.15: *npad 7*

npad 1 is



```
npad macro size
if size eq 1
  nop
```

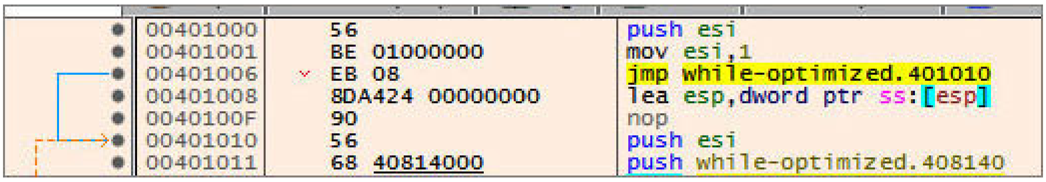
Figure 10.16: *npad 1*

Together, **npad 10** makes:

jmp .+A -> It will add 0x0A to instruction location (0x00401006 + 0x0A)
lea esp, [esp+00000000]

```
nop
```

Let's see how the **npad 10** macro is resolved in the x32dbg screenshot:



```
00401000    56                  push esi
00401001    BE 01000000         mov esi,1
00401006    EB 08               jmp while-optimized.401010
00401008    8DA424 00000000     lea esp,dword ptr ss:[esp]
0040100F    90                  nop
00401010    56                  push esi
00401011    68 40814000         push while-optimized.408140
```

Figure 10.17: npad 10 in x32dbg

All the three instructions marked in RED have no effect on the code flow; they are just equivalent to a series of NOPs of size 10 bytes. The compiler pads 10 bytes between the **mov esi,1** instruction and the label **$LL2@main** (defined next).

▼Line 26-30

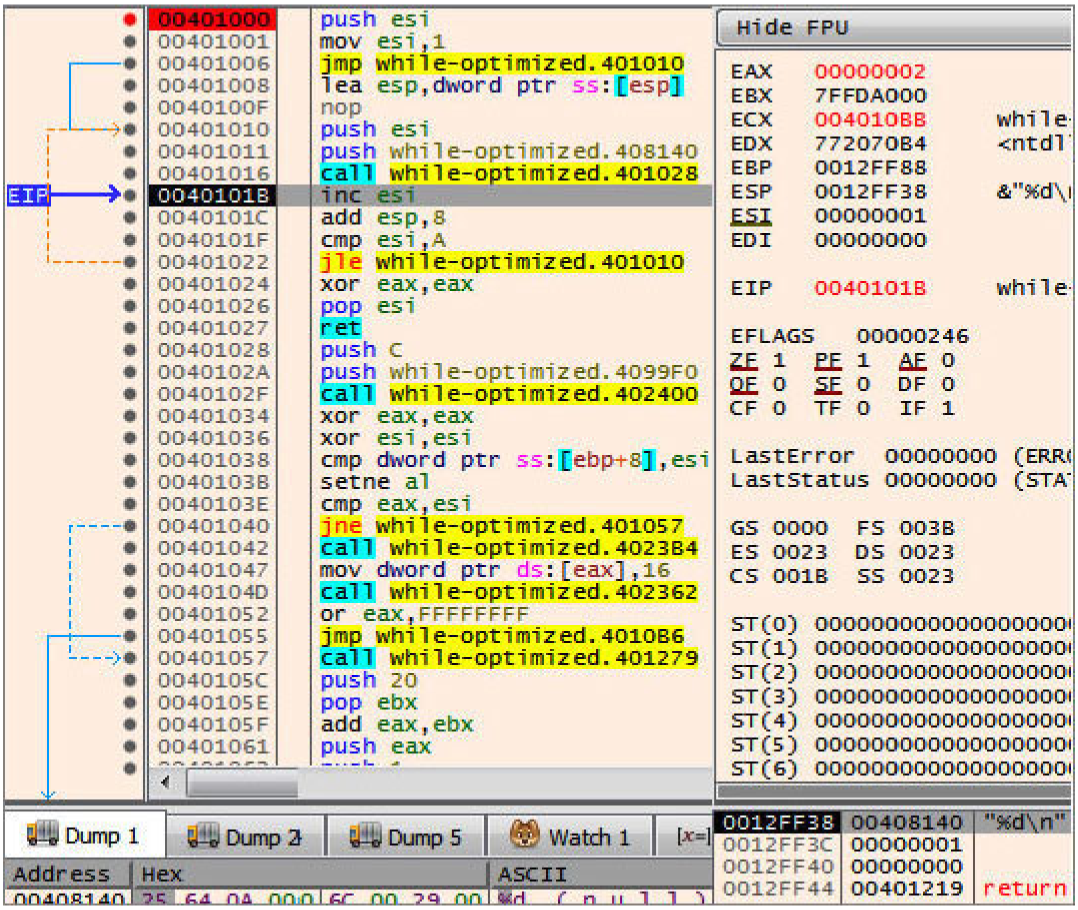
```
$LL2@main:
; Line 11
push esi


push OFFSET $SG4681
call _printf
```

The **$LL2@main** label is defined followed by the comment as line 11. Line 11 of the C/C++ code prints the **iNumber** on the screen using the **printf** function.

```
printf("%d\n", iNumber);
```

In the ASM code, before calling the **printf** function, two arguments to **printf** are pushed onto stack. The first one is **ESI** holding the **iNumber** and the other is the string constant defined by The stack state after the **printf** call is shown as follows, along with the x32dbg screenshot for a better understanding:

[ESP] 0012FF38 00408140 "%d\n", parameter to printf()
[ESP+0x4] 0012FF3C 00000001 iNumber is pushed as a parameter to printf()
[ESP+0x8] 0012FF40 00000000 Old value of ESI is preserved
[ESP+0xC] 0012FF44 00401219 return to 0x00401219 from 0x00401000



```
00401000  push esi
00401001  mov esi,1
00401006  jmp while-optimized.401010
00401008  lea esp,dword ptr ss:[esp]
0040100F  nop
00401010  push esi
00401011  push while-optimized.408140
00401016  call while-optimized.401028
0040101B  inc esi
0040101C  add esp,8
0040101F  cmp esi,A
00401022  jle while-optimized.401010
00401024  xor eax,eax
00401026  pop esi
00401027  ret
00401028  push C
0040102A  push while-optimized.4099F0
0040102F  call while-optimized.402400
00401034  xor eax,eax
00401036  xor esi,esi
00401038  cmp dword ptr ss:[ebp+8],esi
0040103B  setne al
0040103E  cmp eax,esi
00401040  jne while-optimized.401057
00401042  call while-optimized.4023B4
00401047  mov dword ptr ds:[eax],16
0040104D  call while-optimized.402362
00401052  or eax,FFFFFFFF
00401055  jmp while-optimized.4010B6
00401057  call while-optimized.401279
0040105C  push 20
0040105E  pop ebx
0040105F  add eax,ebx
00401061  push eax
```

```
EAX    00000002
EBX    7FFDA000
ECX    004010BB     while
EDX    772070B4     <ntdl
EBP    0012FF88
ESP    0012FF38     &"%d\
ESI    00000001
EDI    00000000

EIP    0040101B     while

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF38  00408140  "%d\n"
0012FF3C  00000001
0012FF40  00000000
0012FF44  00401219  return
```

Figure 10.18: After printf

▼Line 31-35

```
; Line 12
inc esi
add esp, 8
cmp esi, 10        ; 0000000aH
jle SHORT $LL2@main
```

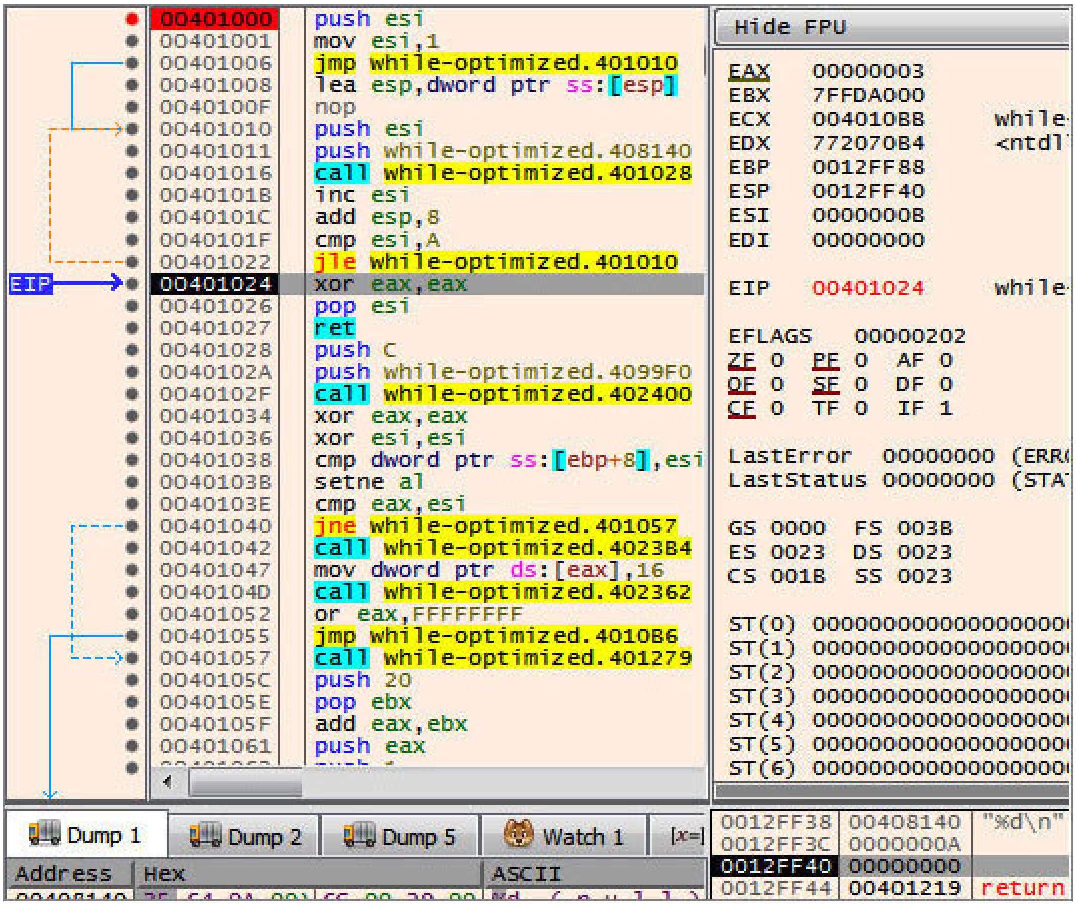
The C/C++ code on line 12 increments the **iNumber** by 1.

```
iNumber = iNumber + 1 ;
```

In ASM, we are incrementing the **iNumber** stored in **ESI** by using the **INC** instruction. **INC** adds 1 to **ESI** and stores the result back in the **ESI** register. The **ADD** instruction cleans the stack by adding 8 bytes to Once the stack is cleaned, the value in **ESI** is compared with 10 (0x0A). It will perform a signed comparison jump after a **CMP** instruction, if the value in **ESI** is less than or equal to 10 (0x0A).

In our case, the current value of **ESI = 0x02** after incrementing. So, the instruction pointer will be jumped to the **$LL2@main** label. The **ESI** is again printed on the console. This loop iterates until **ESI** becomes 0x0B. When **ESI** is incremented to 0x0B, a signed comparison jump after a **CMP** instruction will not take place. So, the instruction pointer will move to the end of the assembly code. The stack state at this point:

[ESP-0x8] 0012FF38 00408140 NOW JUNK
[ESP-0x4] 0012FF3C 0000000A NOW JUNK
[ESP] 0012FF40 00000000 Old value of ESI is preserved
[ESP+0xC] 0012FF44 00401219 return to 0x00401219 from 0x00401000



```
00401000  push esi
00401001  mov esi,1
00401006  jmp while-optimized.401010
00401008  lea esp,dword ptr ss:[esp]
0040100F  nop
00401010  push esi
00401011  push while-optimized.408140
00401016  call while-optimized.401028
0040101B  inc esi
0040101C  add esp,8
0040101F  cmp esi,A
00401022  jle while-optimized.401010
00401024  xor eax,eax
00401026  pop esi
00401027  ret
00401028  push C
0040102A  push while-optimized.4099F0
0040102F  call while-optimized.402400
00401034  xor eax,eax
00401036  xor esi,esi
00401038  cmp dword ptr ss:[ebp+8],esi
0040103B  setne al
0040103E  cmp eax,esi
00401040  jne while-optimized.401057
00401042  call while-optimized.4023B4
00401047  mov dword ptr ds:[eax],16
0040104D  call while-optimized.402362
00401052  or eax,FFFFFFFF
00401055  jmp while-optimized.4010B6
00401057  call while-optimized.401279
0040105C  push 20
0040105E  pop ebx
0040105F  add eax,ebx
00401061  push eax
```

```
Hide FPU
EAX    00000003
EBX    7FFDA000
ECX    004010BB      while
EDX    772070B4      <ntdl
EBP    0012FF88
ESP    0012FF40
ESI    0000000B
EDI    00000000

EIP    00401024      while

EFLAGS    00000202
ZF 0  PE 0  AF 0
OF 0  SE 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 000000000000000000
ST(1) 000000000000000000
ST(2) 000000000000000000
ST(3) 000000000000000000
ST(4) 000000000000000000
ST(5) 000000000000000000
ST(6) 000000000000000000
```

```
0012FF38 | 00408140 | "%d\n"
0012FF3C | 0000000A |
0012FF40 | 00000000 |
0012FF44 | 00401219 | return
```

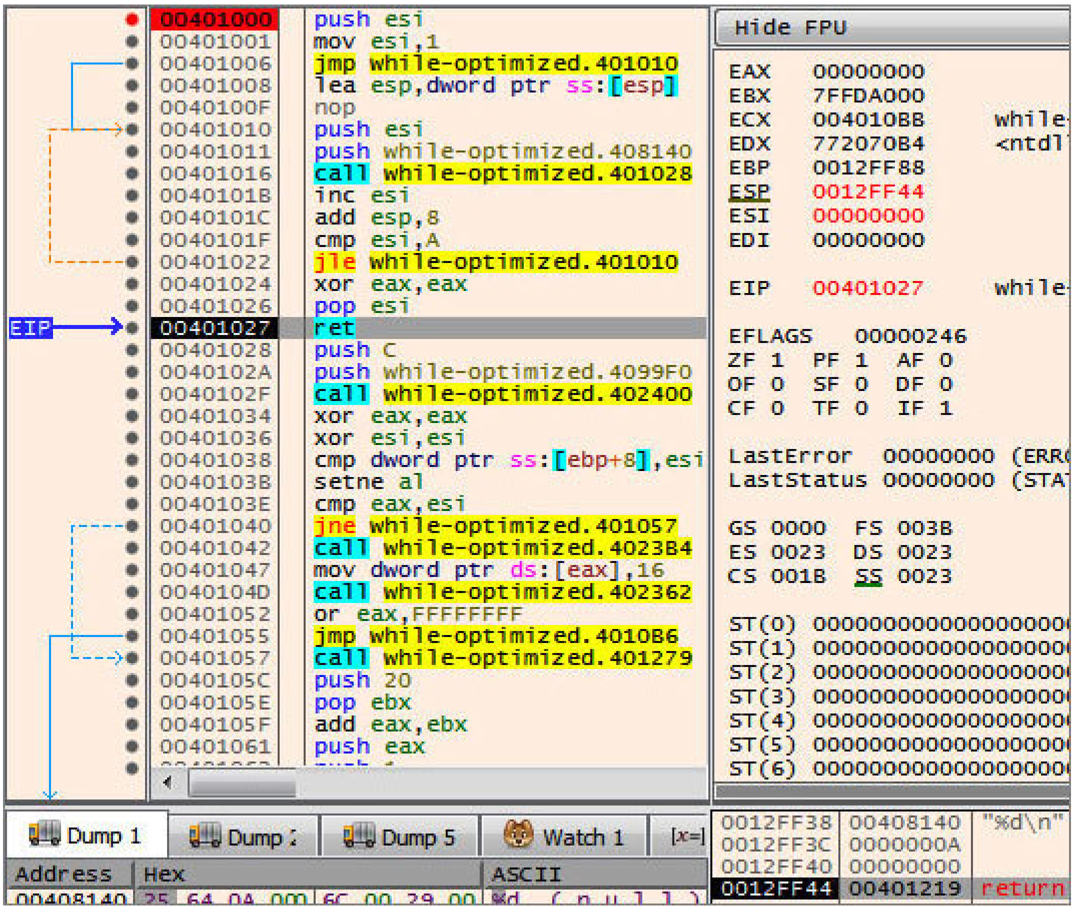
Figure 10.19: ESI is incremented to 0x0B

▼Line 36-42

```
; Line 14
xor eax, eax
pop esi
ret 0
_main ENDP
_TEXT ENDS
END
```

It is the same as explained in the other sections. **EAX** is XOR'ed to return 0. A point to note here is that the value of **ESI** is restored back using the **POP ESI** instruction. With this, our **main** procedure, TEXT segment, and code ends. The stack state after popping **ESI** is as follows:

```
[ESP-0x8] 0012FF38   00408140   NOW JUNK
[ESP-0x4] 0012FF3C   0000000A   NOW JUNK
[ESP]   0012FF40   00000000   NOW JUNK
[ESP+0xC] 0012FF44   00401219   return to 0x00401219 from
0x00401000
```



```
00401000  push esi
00401001  mov esi,1
00401006  jmp while-optimized.401010
00401008  lea esp,dword ptr ss:[esp]
0040100F  nop
00401010  push esi
00401011  push while-optimized.408140
00401016  call while-optimized.401028
0040101B  inc esi
0040101C  add esp,8
0040101F  cmp esi,A
00401022  jle while-optimized.401010
00401024  xor eax,eax
00401026  pop esi
00401027  ret
00401028  push C
0040102A  push while-optimized.4099F0
0040102F  call while-optimized.402400
00401034  xor eax,eax
00401036  xor esi,esi
00401038  cmp dword ptr ss:[ebp+8],esi
0040103B  setne al
0040103E  cmp eax,esi
00401040  jne while-optimized.401057
00401042  call while-optimized.4023B4
00401047  mov dword ptr ds:[eax],16
0040104D  call while-optimized.402362
00401052  or eax,FFFFFFFF
00401055  jmp while-optimized.4010B6
00401057  call while-optimized.401279
0040105C  push 20
0040105E  pop ebx
0040105F  add eax,ebx
00401061  push eax
```

```
EAX     00000000
EBX     7FFDA000
ECX     004010BB     while
EDX     772070B4     <ntdl
EBP     0012FF88
ESP     0012FF44
ESI     00000000
EDI     00000000

EIP     00401027     while

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

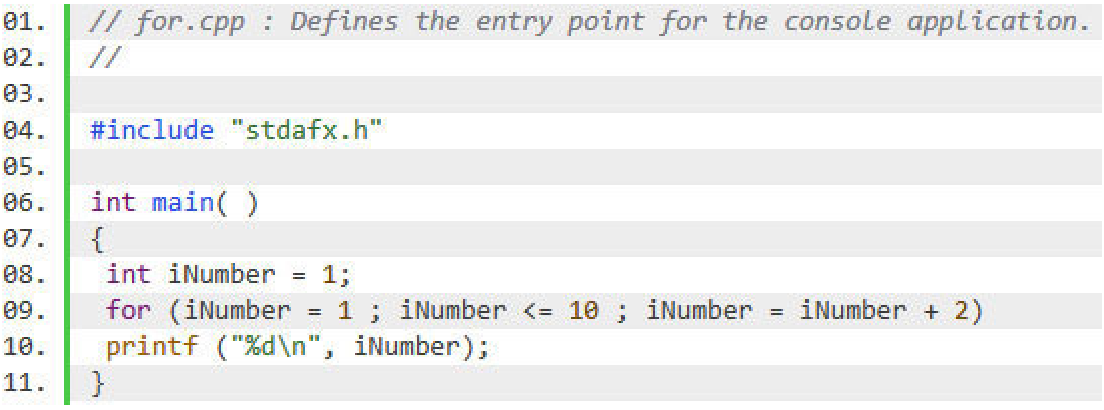
```
Address   Hex                                    ASCII
```

```
0012FF38  00408140  "%d\n"
0012FF3C  0000000A
0012FF40  00000000
0012FF44  00401219  return
```

Figure 10.20: Stack cleaned

For Loop

In this section, we will use **for** loop in the C/C++ code, which initializes an integer variable and increments it by 2 to print it on the screen or console.



```
// for.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int main( )
{
 int iNumber = 1;
 for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
 printf ("%d\n", iNumber);
}
```

Figure 10.21: *for.cpp*

For Loop without Optimization

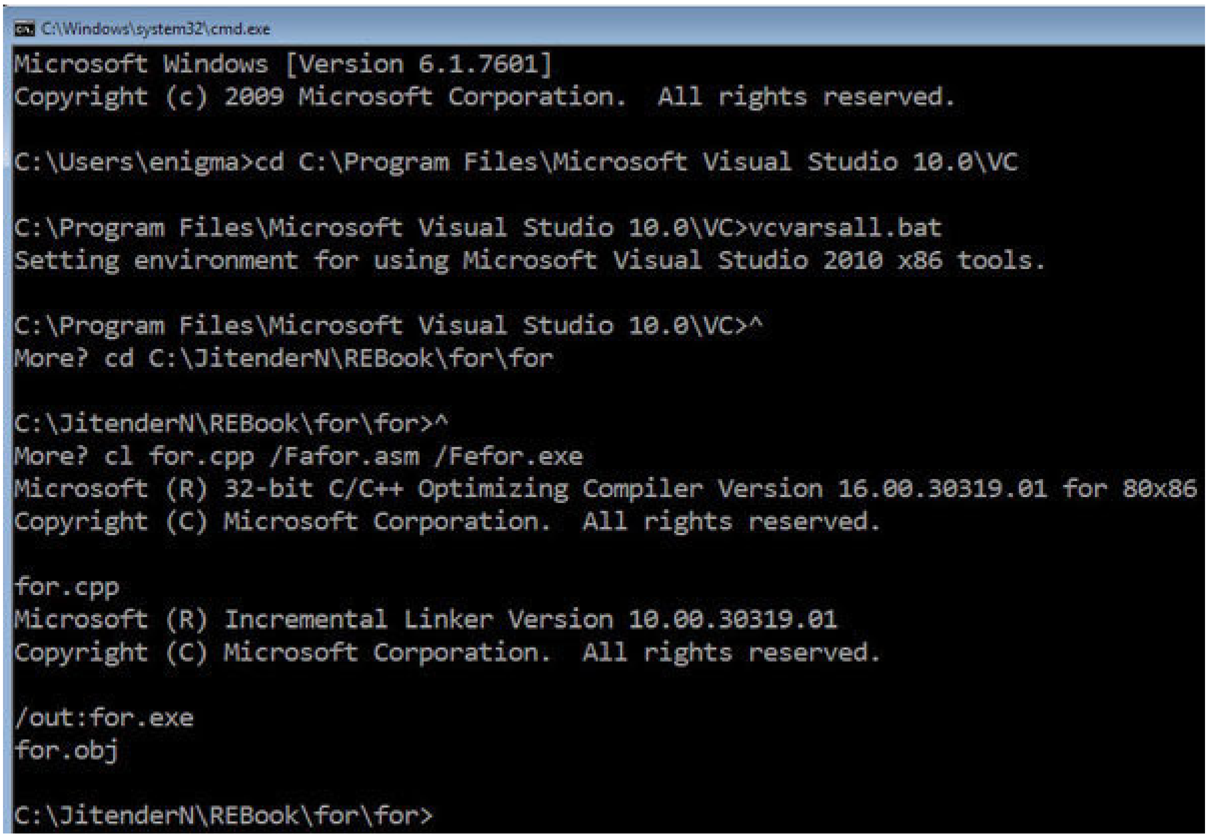
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\for\for

C:\JitenderN\REBook\for\for>^
More? cl for.cpp /Fafor.asm /Fefor.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

for.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:for.exe
for.obj

C:\JitenderN\REBook\for\for>
```

Figure 10.22: *For Loop without Optimization*

The compilation generates the assembly code and the EXE file. To manually disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let’s move on to the generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\for\for\for.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4681 DB '%d', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_iNumber$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\for\for\for.cpp
; Line 7
 push ebp
 mov ebp, esp
 push ecx
; Line 8
 mov DWORD PTR _iNumber$[ebp], 1
; Line 9
 mov DWORD PTR _iNumber$[ebp], 1
 jmp SHORT $LN3@main
$LN2@main:
 mov eax, DWORD PTR _iNumber$[ebp]
 add eax, 2
 mov DWORD PTR _iNumber$[ebp], eax
$LN3@main:
 cmp DWORD PTR _iNumber$[ebp], 10  ; 0000000aH
 jg SHORT $LN4@main
; Line 10
 mov ecx, DWORD PTR _iNumber$[ebp]
 push ecx
 push OFFSET $SG4681
 call _printf
 add esp, 8
 jmp SHORT $LN2@main
$LN4@main:
; Line 11
 xor eax, eax
 mov esp, ebp
 pop ebp
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 10.23: *for.asm*

▼Line 22-25

```
; Line 7
push ebp
mov ebp, esp
push ecx
```

The ASM code starts with the **main** function prologue. The first **PUSH ECX** will create room for the **iNumber** integer variable on the stack.

▼Line 26-27

```
; Line 8
mov DWORD PTR _iNumber$[ebp], 1
```

The following line 8 of the C/C++ code initializes the **iNumber** variable to 1, which corresponds to the preceding ASM code:

```
int iNumber = 1;
```

In the ASM code, **iNumber** macro is defined as:

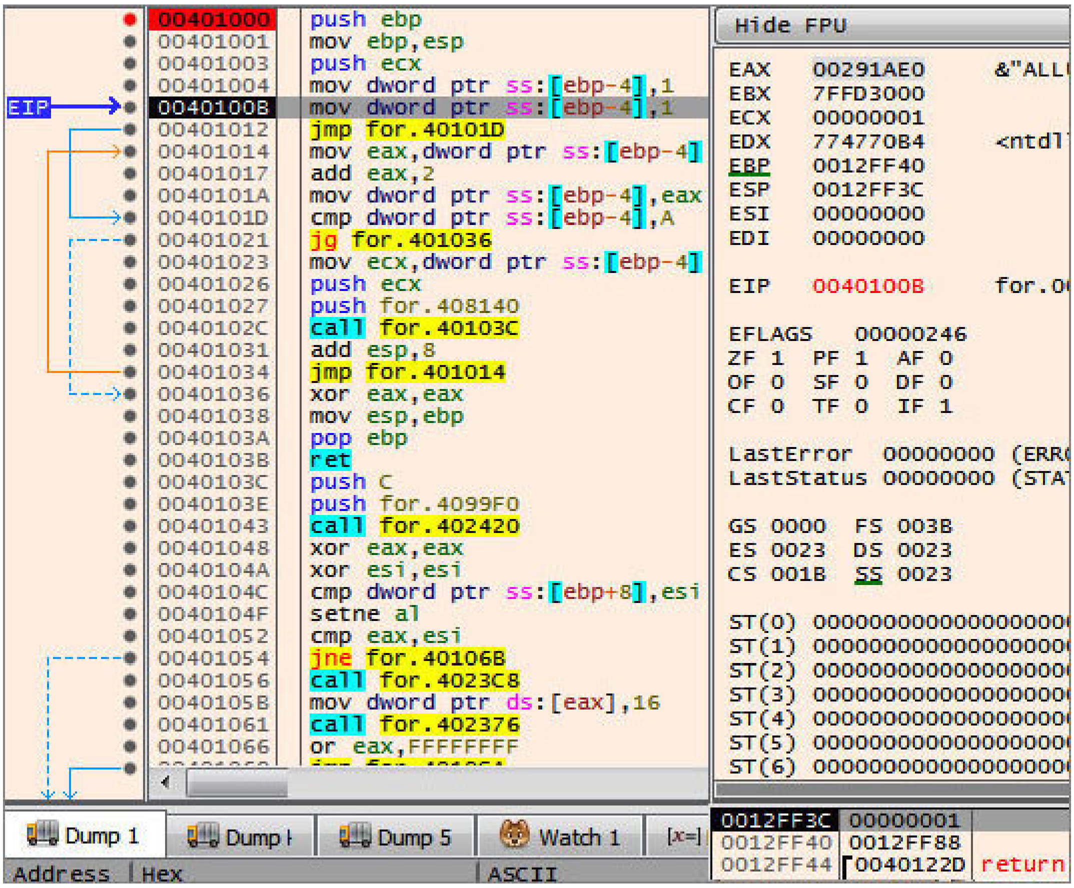
```
_iNumber$ = -4; size = 4
```

So, the preceding ASM code resolves to the following instruction in x32dbg:

mov dword ptr ss:[ebp-0x4], 0x1

MOV initializes the **iNumber** variable on the stack at **ss:[ebp-0x4]** to 0x01. This is the same place in memory where **ECX** was pushed for creating room for the **iNumber** variable. The stack state after this instruction execution is as follows:

[ESP] 0012FF3C 00000001 iNumber variable is stored here
[ESP+0x4] 0012FF40 0012FF88 [EBP]
[ESP+0x8] 0012FF44 0040122D return to for.0040122D from for.00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  mov dword ptr ss:[ebp-4],1
00401012  jmp for.40101D
00401014  mov eax,dword ptr ss:[ebp-4]
00401017  add eax,2
0040101A  mov dword ptr ss:[ebp-4],eax
0040101D  cmp dword ptr ss:[ebp-4],A
00401021  jg for.401036
00401023  mov ecx,dword ptr ss:[ebp-4]
00401026  push ecx
00401027  push for.408140
0040102C  call for.40103C
00401031  add esp,8
00401034  jmp for.401014
00401036  xor eax,eax
00401038  mov esp,ebp
0040103A  pop ebp
0040103B  ret
0040103C  push C
0040103E  push for.4099F0
00401043  call for.402420
00401048  xor eax,eax
0040104A  xor esi,esi
0040104C  cmp dword ptr ss:[ebp+8],esi
0040104F  setne al
00401052  cmp eax,esi
00401054  jne for.40106B
00401056  call for.4023C8
0040105B  mov dword ptr ds:[eax],16
00401061  call for.402376
00401066  or eax,FFFFFFFF
```

```
EAX     00291AE0      &"ALL
EBX     7FFD3000
ECX     00000001
EDX     774770B4      <ntdl
EBP     0012FF40
ESP     0012FF3C
ESI     00000000
EDI     00000000

EIP     0040100B      for.0

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF3C  00000001
0012FF40  0012FF88
0012FF44  0040122D  return
```

Figure 10.24: iNumber on stack

▼Line 28-30

```
; Line 9
mov DWORD PTR _iNumber$[ebp], 1
jmp SHORT $LN3@main
```

This ASM code corresponds to the C/C++ code line 9, where **iNumber** is again initialized and the loop is started.

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
```

In ASM, the **MOV** instruction again initializes the **iNumber** variable. As the code is not optimized, the compiler does not remove the unwanted or duplicate instructions. **JMP** is an unconditional jump to **$LN3@main** label, where the **iNumber** is compared to 10(0x0A) to get into the loop. We will see the **$LN3@main** label in the next ASM explanation. The stack state after this unconditional JMP will be the same:

```
[ESP]  0012FF3C  00000001  iNumber variable is stored here
[ESP+0x4] 0012FF40  0012FF88  [EBP]
[ESP+0x8] 0012FF44  0040122D  return to for.0040122D from
for.00401000
```

▼Line 35-44

```
$LN3@main:
cmp DWORD PTR _iNumber$[ebp], 10   ; 0000000aH
jg SHORT $LN4@main
; Line 10
mov ecx, DWORD PTR _iNumber$[ebp]
push ecx
push OFFSET $SG4681
call _printf
add esp, 8
jmp SHORT $LN2@main
```

The preceding ASM code corresponds to the following C/C++ code lines 9-10, which compare **iNumber** with 10 (0x0A).

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
printf ("%d\n", iNumber);
```

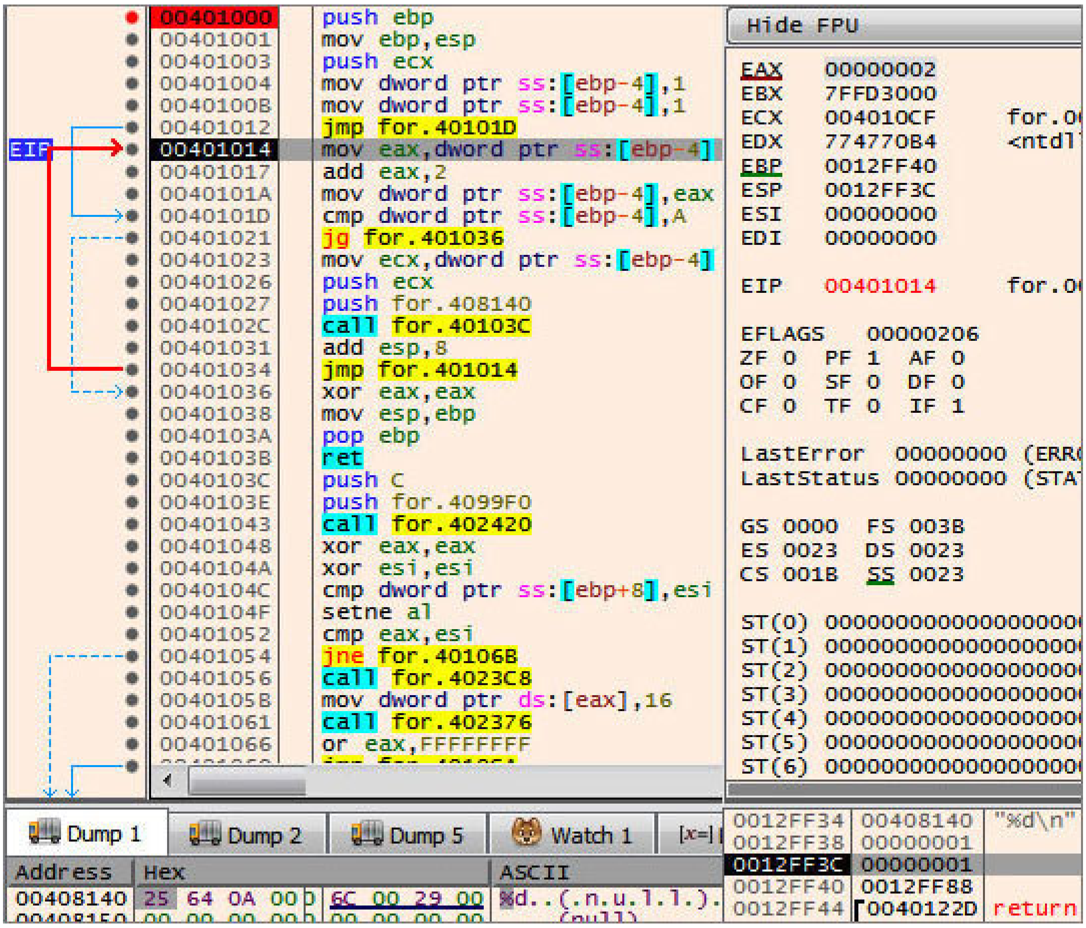
Let's walk through the ASM code. The **CMP** instruction compares the **iNumber** value stored on the stack with 10 (0x0A). It will perform a signed comparison jump if the **iNumber** at **[EBP-0x04]** is greater than 10 (0x0A). Currently, the **iNumber** is 1 so the jump will not happen and the instruction pointer will move to the next section after the line 10 comment in the assembly listing. The **CMP** instruction also sets ZF=0.

Instructions after the line 10 comment move the **iNumber** value at **[EBP-4]** to the **ECX** register so that **ECX** can be pushed back to the stack along with the string constant **$SG4681** as argument to

On call to the **printf** function, **iNumber** value = 1 will be printed on the screen.

On return from the **printf** call, the stack is cleaned with the **ADD** instruction by adding 8 bytes to The which is an unconditional jump, makes the instruction pointer move to the **$LN2@main** label, where the **iNumber** is incremented by 2. The stack state after the unconditional jump is as follows:

```
[ESP-0x8] 0012FF34  00408140  Now JUNK, "%d\n", parameter to print() was pushed
[ESP-0x4] 0012FF38  00000001  Now JUNK, iNumber parameter to print() was pushed
[ESP]  0012FF3C  00000001  iNumber variable is stored here
[ESP+0x4] 0012FF40  0012FF88  [EBP]
[ESP+0x8] 0012FF44  0040122D  return to for.0040122D from for.00401000
```



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  mov dword ptr ss:[ebp-4],1
00401012  jmp for.40101D
00401014  mov eax,dword ptr ss:[ebp-4]
00401017  add eax,2
0040101A  mov dword ptr ss:[ebp-4],eax
0040101D  cmp dword ptr ss:[ebp-4],A
00401021  jg for.401036
00401023  mov ecx,dword ptr ss:[ebp-4]
00401026  push ecx
00401027  push for.408140
0040102C  call for.40103C
00401031  add esp,8
00401034  jmp for.401014
00401036  xor eax,eax
00401038  mov esp,ebp
0040103A  pop ebp
0040103B  ret
0040103C  push C
0040103E  push for.4099F0
00401043  call for.402420
00401048  xor eax,eax
0040104A  xor esi,esi
0040104C  cmp dword ptr ss:[ebp+8],esi
0040104F  setne al
00401052  cmp eax,esi
00401054  jne for.40106B
00401056  call for.4023C8
0040105B  mov dword ptr ds:[eax],16
00401061  call for.402376
00401066  or eax,FFFFFFFF
```

```
EAX    00000002
EBX    7FFD3000
ECX    004010CF       for.0
EDX    774770B4       <ntdl
EBP    0012FF40
ESP    0012FF3C
ESI    00000000
EDI    00000000

EIP    00401014       for.0

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
Address   Hex                          ASCII
00408140  25 64 0A 00  6C 00 29 00     %d..(.n.u.l.l.).
```

```
0012FF34  00408140  "%d\n"
0012FF38  00000001
0012FF3C  00000001
0012FF40  0012FF88
0012FF44  0040122D  return
```

Figure 10.25: Stack state after unconditional jump

▼Line 31-34

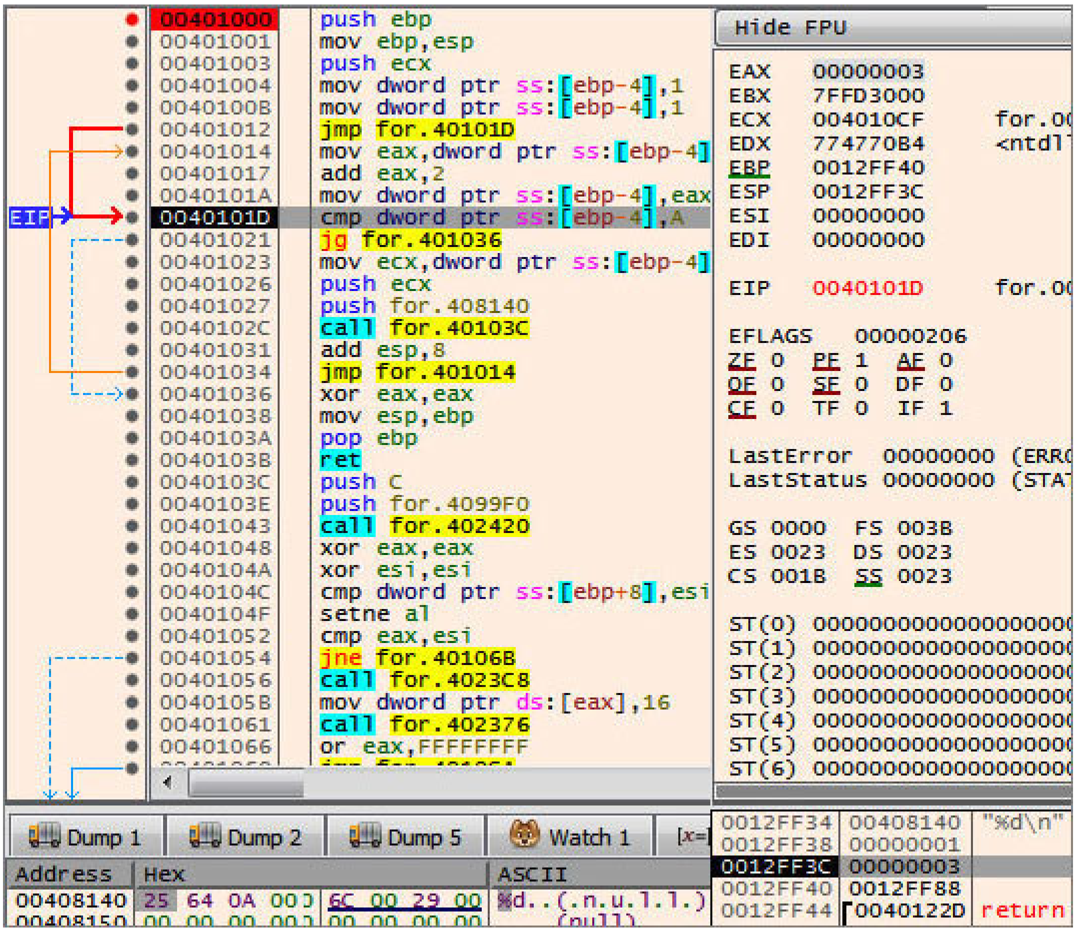
```
$LN2@main:
mov eax, DWORD PTR _iNumber$[ebp]
add eax, 2
mov DWORD PTR _iNumber$[ebp], eax
```

This ASM code corresponds to the following C/C++ code line 9, where the **iNumber** is incremented by 2.

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
```

In ASM, the **MOV** instruction takes the **iNumber** value from **[EBP-0x4]** to the **EAX** register, where **EAX** is incremented by 2 using the **ADD** instruction and moved back to the memory placeholder for Following this instruction is the same instruction when **iNumber** is compared with 10 (0x0A), as discussed previously. The stack state after the increment is as follows:

```
[ESP-0x8] 0012FF34   00408140   Now JUNK, "%d\n", parameter to print() was pushed
[ESP-0x4] 0012FF38   00000001   Now JUNK, iNumber parameter to print() was pushed
[ESP]   0012FF3C   00000003   iNumber variable is stored here
[ESP+0x4] 0012FF40   0012FF88   [EBP]
[ESP+0x8] 0012FF44   0040122D   return to for.0040122D from for.00401000
```



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  mov dword ptr ss:[ebp-4],1
00401012  jmp for.40101D
00401014  mov eax,dword ptr ss:[ebp-4]
00401017  add eax,2
0040101A  mov dword ptr ss:[ebp-4],eax
0040101D  cmp dword ptr ss:[ebp-4],A
00401021  jg for.401036
00401023  mov ecx,dword ptr ss:[ebp-4]
00401026  push ecx
00401027  push for.408140
0040102C  call for.40103C
00401031  add esp,8
00401034  jmp for.401014
00401036  xor eax,eax
00401038  mov esp,ebp
0040103A  pop ebp
0040103B  ret
0040103C  push C
0040103E  push for.4099F0
00401043  call for.402420
00401048  xor eax,eax
0040104A  xor esi,esi
0040104C  cmp dword ptr ss:[ebp+8],esi
0040104F  setne al
00401052  cmp eax,esi
00401054  jne for.40106B
00401056  call for.4023C8
0040105B  mov dword ptr ds:[eax],16
00401061  call for.402376
00401066  or eax,FFFFFFFF
```

```
EAX     00000003
EBX     7FFD3000
ECX     004010CF      for.00
EDX     774770B4      <ntdl
EBP     0012FF40
ESP     0012FF3C
ESI     00000000
EDI     00000000

EIP     0040101D      for.00

EFLAGS     00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
Address   Hex                        ASCII
00408140  25 64 0A 00 6C 00 29 00   %d..(.n.u.l.l.)
00408150  00 00 00 00 00 00 00 00       (null)
```

```
0012FF34  00408140  "%d\n"
0012FF38  00000001
0012FF3C  00000003
0012FF40  0012FF88
0012FF44  0040122D  return
```

Figure 10.26: *iNumber on stack is incremented*

▼Line 35-37

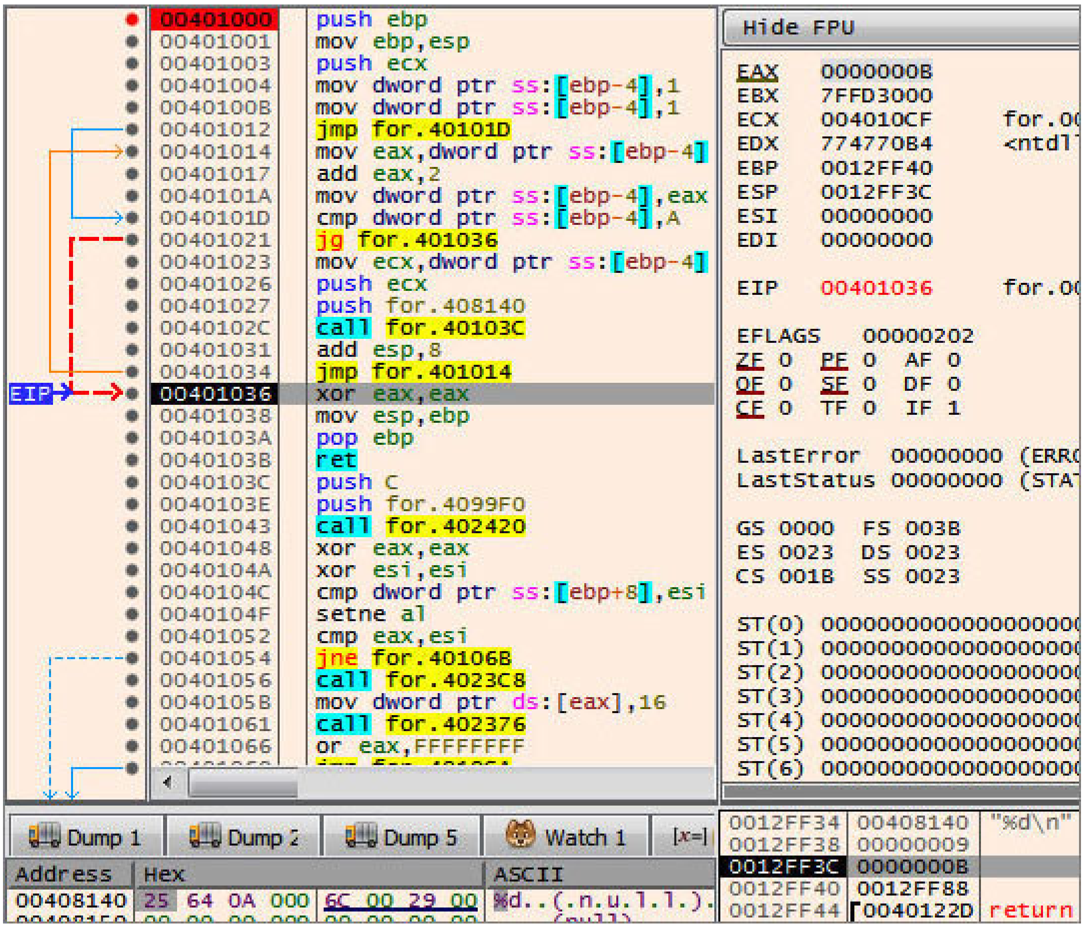
```
$LN3@main:
cmp DWORD PTR _iNumber$[ebp], 10   ; 0000000aH
jg SHORT $LN4@main
```

This ASM code corresponds to the following C/C++ code line 9, which compares the **iNumber** with the 10 (0x0A).

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
```

Now, we will take that iteration wherein the **iNumber** is incremented to 11 (0xB). At this point, it will perform a signed comparison jump as the **iNumber** at **[EBP-0x04]** is greater than 10 (0x0A). The **iNumber** is 11 (0xB), so the jump will happen to the **$LN4@main** label. The stack state after the **JG (JUMP if Greater)** is as follows:

```
[ESP-0x8] 0012FF34   00408140   Now JUNK, "%d\n", parameter to print() was pushed
[ESP-0x4] 0012FF38   00000009   Now JUNK, last iNumber value pushed as arg to print
[ESP]   0012FF3C   0000000B   iNumber variable is stored here
[ESP+0x4] 0012FF40   0012FF88   [EBP]
[ESP+0x8] 0012FF44   0040122D   return to for.0040122D from for.00401000
```



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  mov dword ptr ss:[ebp-4],1
00401012  jmp for.40101D
00401014  mov eax,dword ptr ss:[ebp-4]
00401017  add eax,2
0040101A  mov dword ptr ss:[ebp-4],eax
0040101D  cmp dword ptr ss:[ebp-4],A
00401021  jg for.401036
00401023  mov ecx,dword ptr ss:[ebp-4]
00401026  push ecx
00401027  push for.408140
0040102C  call for.40103C
00401031  add esp,8
00401034  jmp for.401014
00401036  xor eax,eax
00401038  mov esp,ebp
0040103A  pop ebp
0040103B  ret
0040103C  push C
0040103E  push for.4099F0
00401043  call for.402420
00401048  xor eax,eax
0040104A  xor esi,esi
0040104C  cmp dword ptr ss:[ebp+8],esi
0040104F  setne al
00401052  cmp eax,esi
00401054  jne for.40106B
00401056  call for.4023C8
0040105B  mov dword ptr ds:[eax],16
00401061  call for.402376
00401066  or eax,FFFFFFFF
```

```
EAX  0000000B
EBX  7FFD3000
ECX  004010CF    for.0
EDX  774770B4    <ntdl
EBP  0012FF40
ESP  0012FF3C
ESI  00000000
EDI  00000000

EIP  00401036    for.0

EFLAGS   00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 000000000000000000
ST(1) 000000000000000000
ST(2) 000000000000000000
ST(3) 000000000000000000
ST(4) 000000000000000000
ST(5) 000000000000000000
ST(6) 000000000000000000
```

```
0012FF34  00408140  "%d\n"
0012FF38  00000009
0012FF3C  0000000B
0012FF40  0012FF88
0012FF44  0040122D  return
```

```
Address   Hex                        ASCII
00408140  25 64 0A 000 6C 00 29 00   %d..(.n.u.l.l.).
```

Figure 10.27: *iNumber is incremented to 11 (0x0B)*

▼Line 45-53

```
$LN4@main:
; Line 11
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

In this **$LN4@main** label, **EAX** is zeroed to return 0. The function epilogue is called to END the **main** function, TEXT segment, and code. The stack state will be as follows:

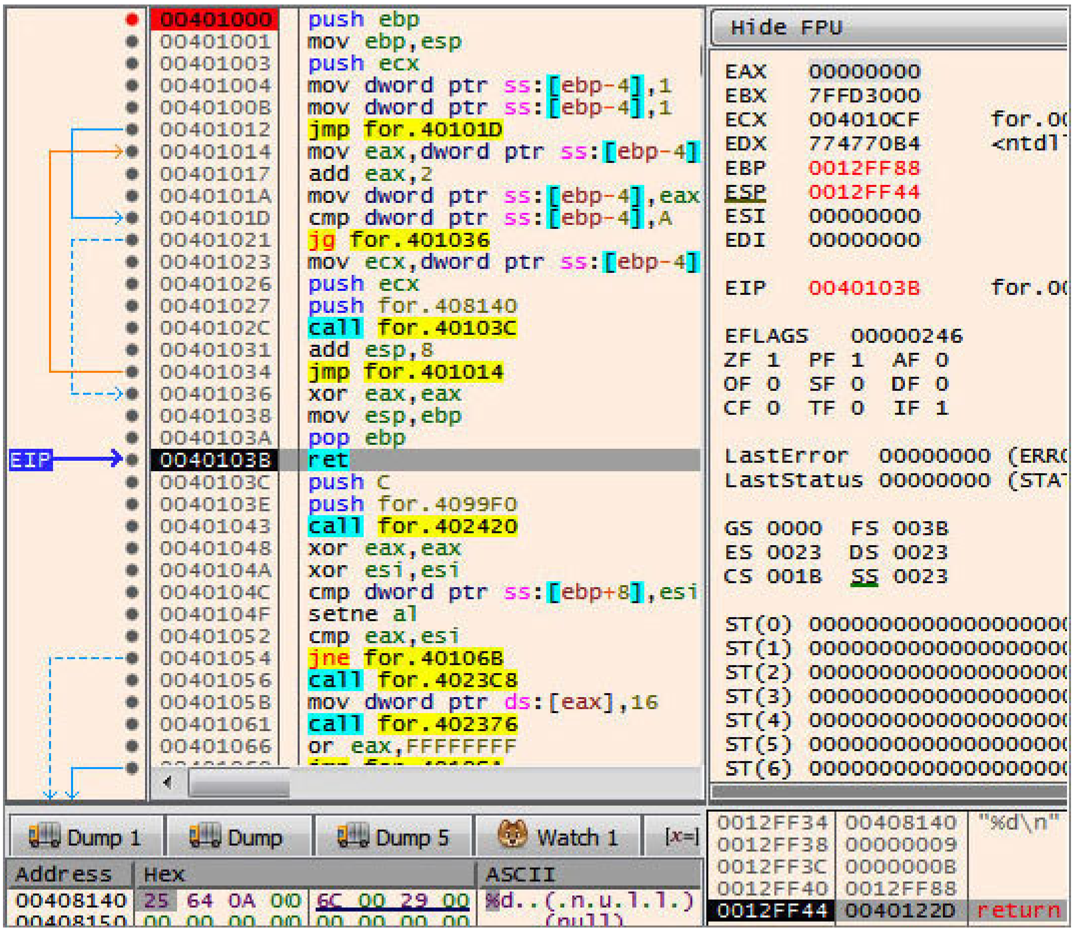
[ESP-0x10] 0012FF34 00408140 Now JUNK, "%d\n", parameter to print() was pushed

[ESP-0xC] 0012FF38 00000009 Now JUNK, last iNumber value pushed as arg to print

[ESP-0x8] 0012FF3C 0000000B Now JUNK, iNumber variable is stored here

[ESP-0x4] 0012FF40 0012FF88 Now JUNK, EBP popped

[ESP] 0012FF44 0040122D return to for.0040122D from for.00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  mov dword ptr ss:[ebp-4],1
0040100B  mov dword ptr ss:[ebp-4],1
00401012  jmp for.40101D
00401014  mov eax,dword ptr ss:[ebp-4]
00401017  add eax,2
0040101A  mov dword ptr ss:[ebp-4],eax
0040101D  cmp dword ptr ss:[ebp-4],A
00401021  jg for.401036
00401023  mov ecx,dword ptr ss:[ebp-4]
00401026  push ecx
00401027  push for.408140
0040102C  call for.40103C
00401031  add esp,8
00401034  jmp for.401014
00401036  xor eax,eax
00401038  mov esp,ebp
0040103A  pop ebp
0040103B  ret
0040103C  push C
0040103E  push for.4099F0
00401043  call for.402420
00401048  xor eax,eax
0040104A  xor esi,esi
0040104C  cmp dword ptr ss:[ebp+8],esi
0040104F  setne al
00401052  cmp eax,esi
00401054  jne for.40106B
00401056  call for.4023C8
0040105B  mov dword ptr ds:[eax],16
00401061  call for.402376
00401066  or eax,FFFFFFFF
```

```
EAX     00000000
EBX     7FFD3000
ECX     004010CF      for.0
EDX     774770B4      <ntdl
EBP     0012FF88
ESP     0012FF44
ESI     00000000
EDI     00000000

EIP     0040103B      for.0

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
Address   Hex                        ASCII
00408140  25 64 0A 00  6C 00 29 00   %d..(.n.u.l.l.)
00408150  00 00 00 00  00 00 00 00       (null)
```

```
0012FF34  00408140  "%d\n"
0012FF38  00000009
0012FF3C  0000000B
0012FF40  0012FF88
0012FF44  0040122D  return
```

Figure 10.28: *Stack cleaned*

For Loop with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

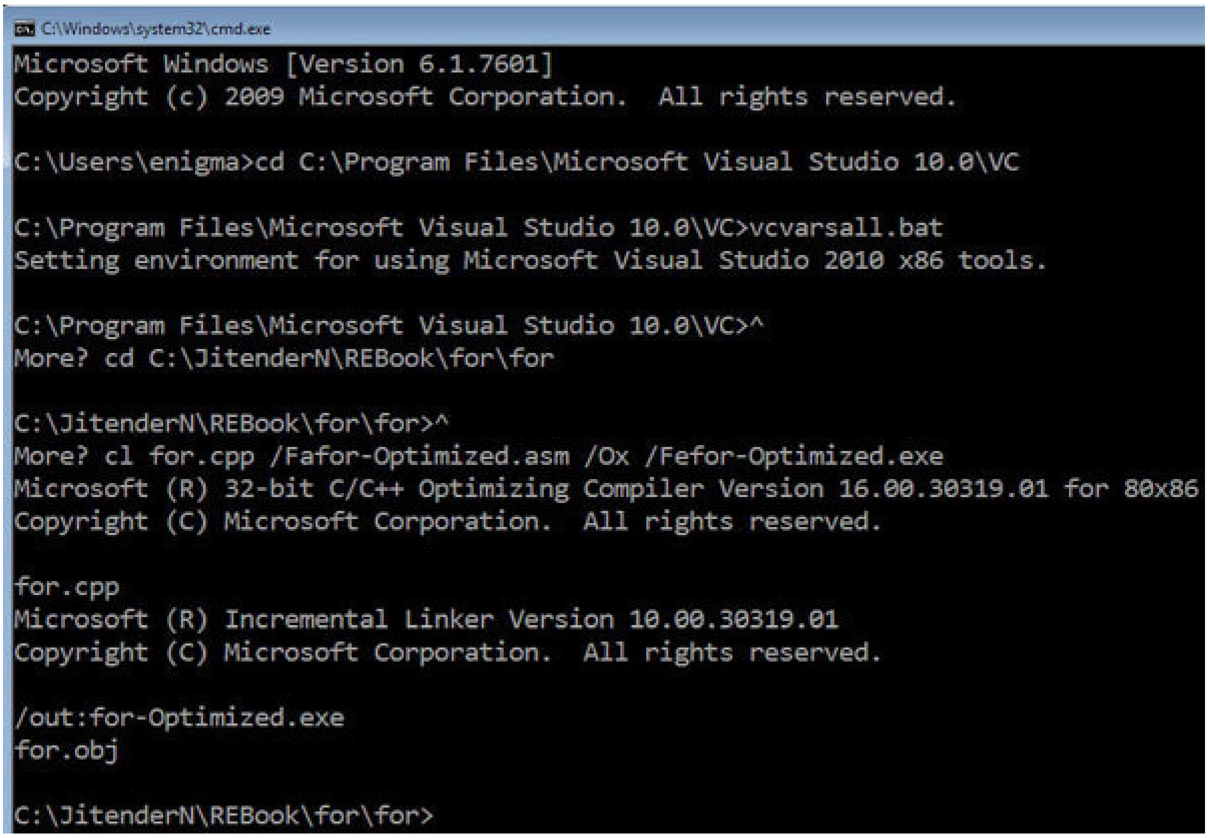
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\for\for

C:\JitenderN\REBook\for\for>^
More? cl for.cpp /Fafor-Optimized.asm /Ox /Fefor-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

for.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:for-Optimized.exe
for.obj

C:\JitenderN\REBook\for\for>
```

Figure 10.29: *For Loop with Optimization*

To manually disable the ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let’s move on to the generated assembly listing:

```
01. ; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01
02.
03.  TITLE C:\JitenderN\REBook\for\for\for.cpp
04.  .686P
05.  .XMM
06.  include listing.inc
07.  .model flat
08.
09. INCLUDELIB LIBCMT
10. INCLUDELIB OLDNAMES
11.
12. CONST SEGMENT
13. $SG4681 DB '%d', 0aH, 00H
14. CONST ENDS
15. PUBLIC _main
16. EXTRN _printf:PROC
17. ; Function compile flags: /Ogtpy
18. _TEXT SEGMENT
19. _main PROC
20. ; File c:\jitendern\rebook\for\for\for.cpp
21. ; Line 7
22.  push esi
23. ; Line 9
24.  mov esi, 1
25.  npad 10
26. $LL3@main:
27. ; Line 10
28.  push esi
29.  push OFFSET $SG4681
30.  call _printf
31.  add esi, 2
32.  add esp, 8
33.  cmp esi, 10     ; 0000000aH
34.  jle SHORT $LL3@main
35. ; Line 11
36.  xor eax, eax
37.  pop esi
38.  ret 0
39. _main ENDP
40. _TEXT ENDS
41. END
```

Figure 10.30: *For-Optimized.asm*

Let's walk through the generated ASM code using the optimization flag.

▼Line 21-22

```
; Line 7
push esi
```

In the function prologue, **ESI** is pushed onto the stack and in the function epilogue, **ESI** is popped up. **ESI** will be used as the **iNumber** placeholder. So before using it as a placeholder for the the old value **ESI** is preserved onto the stack and in the **main** procedure epilogue, the **ESI** value is restored to the old value using **POP**

▼Line 23-25

```
; Line 9
mov esi, 1
npad 10
```

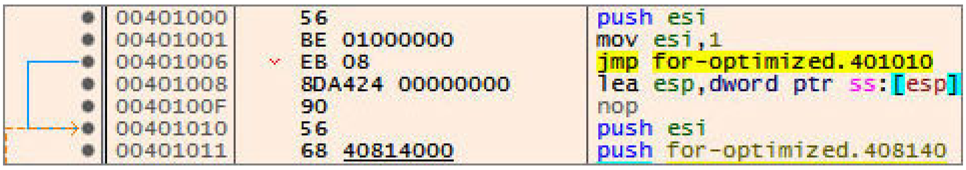
Line 9 of the C/C++ code initializes the **iNumber** variable:

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
```

In the ASM code, the **ESI** register will be used as a placeholder for the **iNumber** value. So, 0x01 is moved to the **ESI** register to initialize the

npad macro inserts non-destructive non-operational instructions. For information on please refer to the *Appendix* section.

The compiler inserts three instructions that will have no effect on the code flow; they are just equivalent to a series of NOPs of size 10 bytes. In x32dbg, **npad** macro is resolved as marked within the RED box:



```
00401000    56                  push esi
00401001    BE 01000000         mov esi,1
00401006    EB 08               jmp for-optimized.401010
00401008    8DA424 00000000     lea esp,dword ptr ss:[esp]
0040100F    90                  nop
00401010    56                  push esi
00401011    68 40814000         push for-optimized.408140
```

Figure 10.31: *npad 10*

The compiler pads 10 bytes between the **mov esi,1** instruction and the label **$LL3@main** (defined next).

▼Line 26-34

```
$LL3@main:

; Line 10
push esi
push OFFSET $SG4681
call _printf
add esi, 2
add esp, 8
cmp esi, 10        ; 0000000aH
jle SHORT $LL3@main
```

Line 9-10 of the C/C++ code initializes the **iNumber** value to enter into the loop to print **iNumber** and increment every **iNumber** iteration by 2.

```
for (iNumber = 1 ; iNumber <= 10 ; iNumber = iNumber + 2)
printf ("%d\n", iNumber);
```

In ASM, it is done by pushing **ESI** and the string constant **$SG4681** onto the stack. Once we have both the arguments to the **printf** function on the stack, the call to the **printf** function is made. This will print the current value of **iNumber** stored in the **ESI** register on the screen.

Now, **ESI** is incremented by 2 by adding 0x02 to the **ESI** register. This is to increment the **iNumber** value by 2.

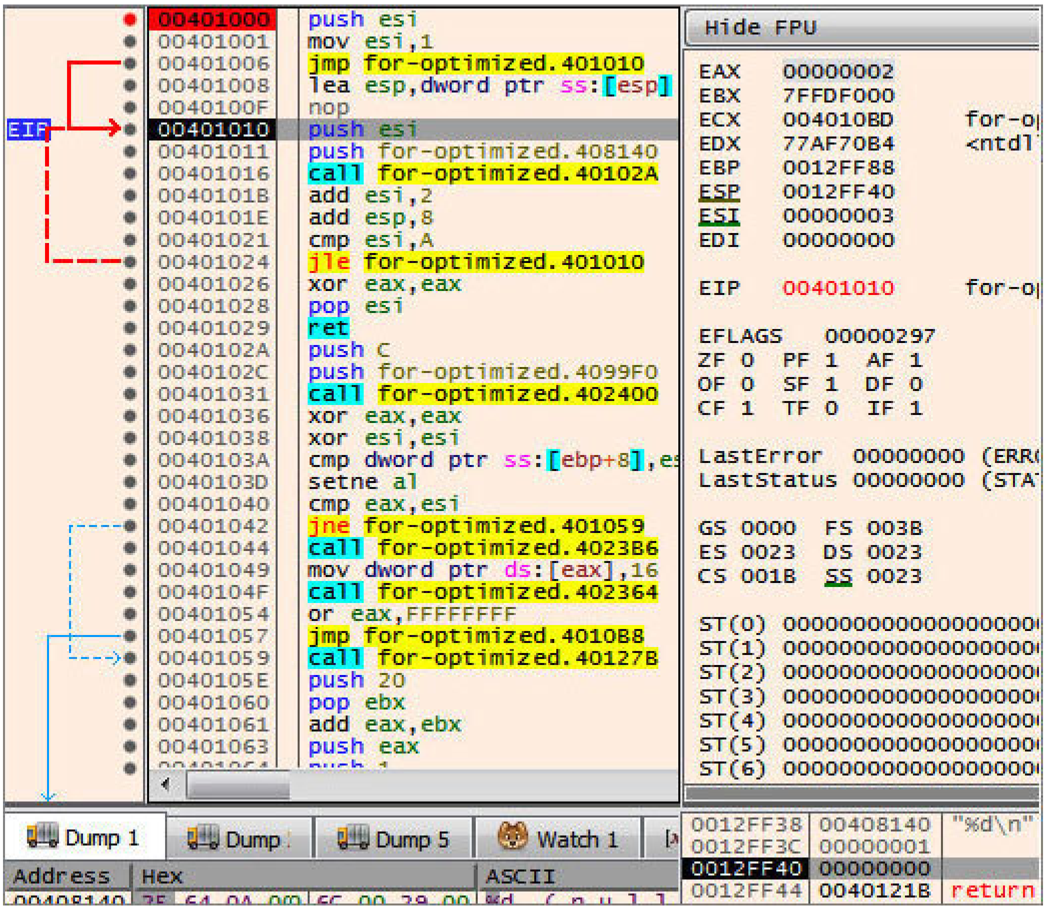
Next, the **ADD** instruction cleans up the stack by adding 8 bytes to the **ESP** register.

CMP will compare the value in **ESI** with 0x10. It will perform a signed comparison jump after a **CMP** instruction if the value in **ESI** is less than or equal to 10 (0x0A).

In our case, the current value of **ESI = 0x03** after incrementing, which is less than 10 (0x0A). So, the instruction pointer will jump to the **$LL3@main** label. The stack state and x32dbg screenshot at this point will be as follows:

[ESP-0x8] 0012FF38 00408140 "%d\n", parameter to printf()
[ESP-0x4] 0012FF3C 00000001 ESI holding iNumber is pushed, as arg to printf

[ESP] 0012FF40 00000000 Old value of ESI is preserved on stack
[ESP+0x4] 0012FF44 0040121B return to 0x0040121B from 0x00401000



```
00401000  push esi
00401001  mov esi,1
00401006  jmp for-optimized.401010
00401008  lea esp,dword ptr ss:[esp]
0040100F  nop
00401010  push esi
00401011  push for-optimized.408140
00401016  call for-optimized.40102A
0040101B  add esi,2
0040101E  add esp,8
00401021  cmp esi,A
00401024  jle for-optimized.401010
00401026  xor eax,eax
00401028  pop esi
00401029  ret
0040102A  push C
0040102C  push for-optimized.4099F0
00401031  call for-optimized.402400
00401036  xor eax,eax
00401038  xor esi,esi
0040103A  cmp dword ptr ss:[ebp+8],es
0040103D  setne al
00401040  cmp eax,esi
00401042  jne for-optimized.401059
00401044  call for-optimized.4023B6
00401049  mov dword ptr ds:[eax],16
0040104F  call for-optimized.402364
00401054  or eax,FFFFFFFF
00401057  jmp for-optimized.4010B8
00401059  call for-optimized.40127B
0040105E  push 20
00401060  pop ebx
00401061  add eax,ebx
00401063  push eax
```

```
EAX    00000002
EBX    7FFDF000
ECX    004010BD      for-o
EDX    77AF70B4      <ntdl
EBP    0012FF88
ESP    0012FF40
ESI    00000003
EDI    00000000

EIP    00401010      for-o

EFLAGS    00000297
ZF 0   PF 1   AF 1
OF 0   SF 1   DF 0
CF 1   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
0012FF38  00408140  "%d\n"
0012FF3C  00000001
0012FF40  00000000
0012FF44  0040121B  return
```

```
Address   Hex                                 ASCII
```

Figure 10.32: After JMP

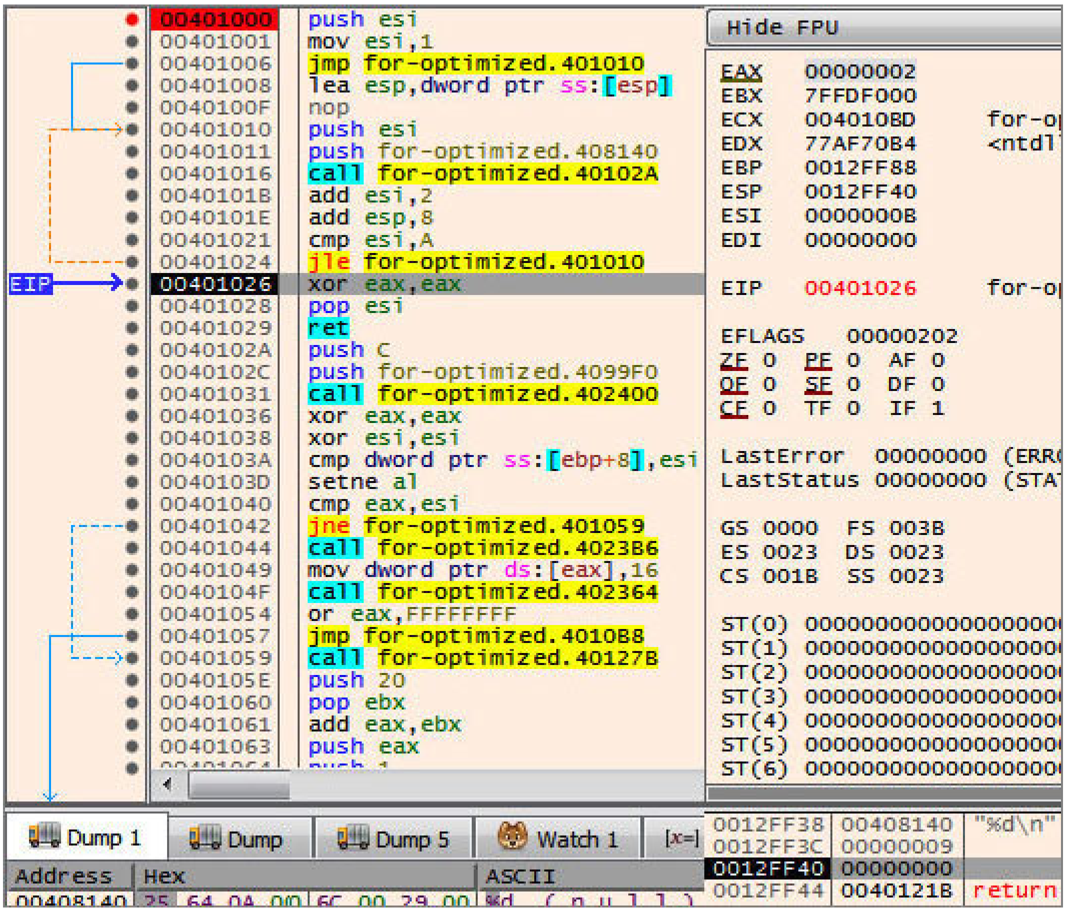
This loop iterates until **ESI** becomes 0x0B. When **ESI** is incremented to 0x0B, a signed comparison jump after a **CMP** instruction will not take place. So, the instruction pointer will move to the end of the assembly code. The stack state at this point is as follows:

[ESP-0x8] 0012FF38 00408140 "%d\n", parameter to printf()

[ESP-0x4] 0012FF3C 00000009 ESI holding iNumber is pushed, as arg to printf

[ESP] 0012FF40 00000000 Old value of ESI is preserved on stack

[ESP+0x4] 0012FF44 0040121B return to 0x0040121B from 0x00401000



```
00401000  push esi
00401001  mov esi,1
00401006  jmp for-optimized.401010
00401008  lea esp,dword ptr ss:[esp]
0040100F  nop
00401010  push esi
00401011  push for-optimized.408140
00401016  call for-optimized.40102A
0040101B  add esi,2
0040101E  add esp,8
00401021  cmp esi,A
00401024  jle for-optimized.401010
00401026  xor eax,eax
00401028  pop esi
00401029  ret
0040102A  push C
0040102C  push for-optimized.4099F0
00401031  call for-optimized.402400
00401036  xor eax,eax
00401038  xor esi,esi
0040103A  cmp dword ptr ss:[ebp+8],esi
0040103D  setne al
00401040  cmp eax,esi
00401042  jne for-optimized.401059
00401044  call for-optimized.4023B6
00401049  mov dword ptr ds:[eax],16
0040104F  call for-optimized.402364
00401054  or eax,FFFFFFFF
00401057  jmp for-optimized.4010B8
00401059  call for-optimized.40127B
0040105E  push 20
00401060  pop ebx
00401061  add eax,ebx
00401063  push eax
```

```
EAX   00000002
EBX   7FFDF000
ECX   004010BD    for-o
EDX   77AF70B4    <ntdl
EBP   0012FF88
ESP   0012FF40
ESI   0000000B
EDI   00000000

EIP   00401026    for-o

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023
```

```
0012FF38  00408140  "%d\n"
0012FF3C  00000009
0012FF40  00000000
0012FF44  0040121B  return
```

Figure 10.33: *ESI is incremented to 0x0B*

▼Line 35-41

; Line 11
xor eax, eax
pop esi
ret 0

```
_main ENDP
_TEXT ENDS
END
```

With this, the **main** function, TEXT segment, and code is ended by XOR'ing **EAX** to return 0 and the old value of **ESI** is restored from the stack by the **POP ESI** instruction. The stack state at this point will be as follows:

```
[ESP-0xC] 0012FF38   00408140   Now JUNK
[ESP-0x8] 0012FF3C   00000009   Now JUNK
[ESP-0x4] 0012FF40   00000000   Now JUNK
[ESP]   0012FF44   0040121B   return to 0x0040121B from
0x00401000
```

Conclusion

In this chapter, we learned about the different loop statements in C/C++ and how the code pattern of different loop statements can be identified in a disassembled code. We also found out how to putt a conditional breakpoint to check the stack state and the major flag bits. In the assembly listing generated from loop statement programs, we checked CMP and Jump instruction patterns in assembly and learned the difference between optimized and non-optimized code. In the subsequent chapters, we will talk about real-world problems and their solutions.

CHAPTER 11

Array Code Pattern in Reverse Engineering

Imagine you are a programmer working in some company to develop software to mark the attendance of 50 students in a class. Now, as a software developer, you have an option to define separate variables for students, which is quite difficult to manage and not appropriate in case the student count increases in the future. To manage these situations where you have similar data type variables, every programming language is equipped with the concept of arrays. Arrays are sets of similar elements stored in contiguous memory locations. So, in developing the attendance software for a class, an array will be the most appropriate.

We have already studied the pattern of pointers in the earlier chapter. Pointers and array are correlated to each other. The importance of array in reverse engineering is very important. Sometimes, malware writers code malware in such a way that they infect the list of files with a specific extension. In most of these cases, all the specific file extensions are defined in an array. So, adding any other extension in the future to the malware becomes easy for malware coders. In this chapter, we will work on this real-world problem by coding the same program using array and then reversing it to understand its pattern.

Structure

In this chapter, we will cover the following topics:

Understanding an array

Array loop without Optimization

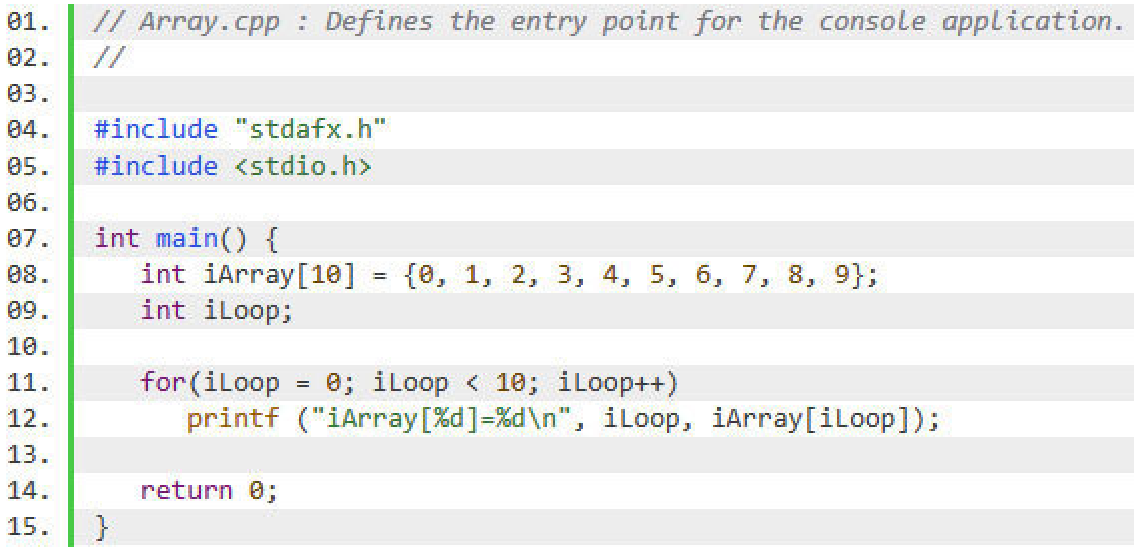
Array loop with Optimization

Objective

The objective of this chapter is to understand the working of an array with respect to reverse engineering. We will talk about the array code pattern in a disassembled code and how arrays are stored in memory. Arrays are stored in contiguous memory locations and depending on the data types of the array, the storage is allocated in memory. We will also learn to put a conditional breakpoint in code execution. We will also cover an array program pattern when the code is optimized and not optimized.

Understanding an array

In this example, we have declared and defined an array named We will iterate through the array up to its length and print each element along with the index on the screen.



```
// Array.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>

int main() {
   int iArray[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
   int iLoop;

   for(iLoop = 0; iLoop < 10; iLoop++)
      printf ("iArray[%d]=%d\n", iLoop, iArray[iLoop]);

   return 0;
}
```

Figure 11.1: *Array.cpp*

Array Loop without Optimization

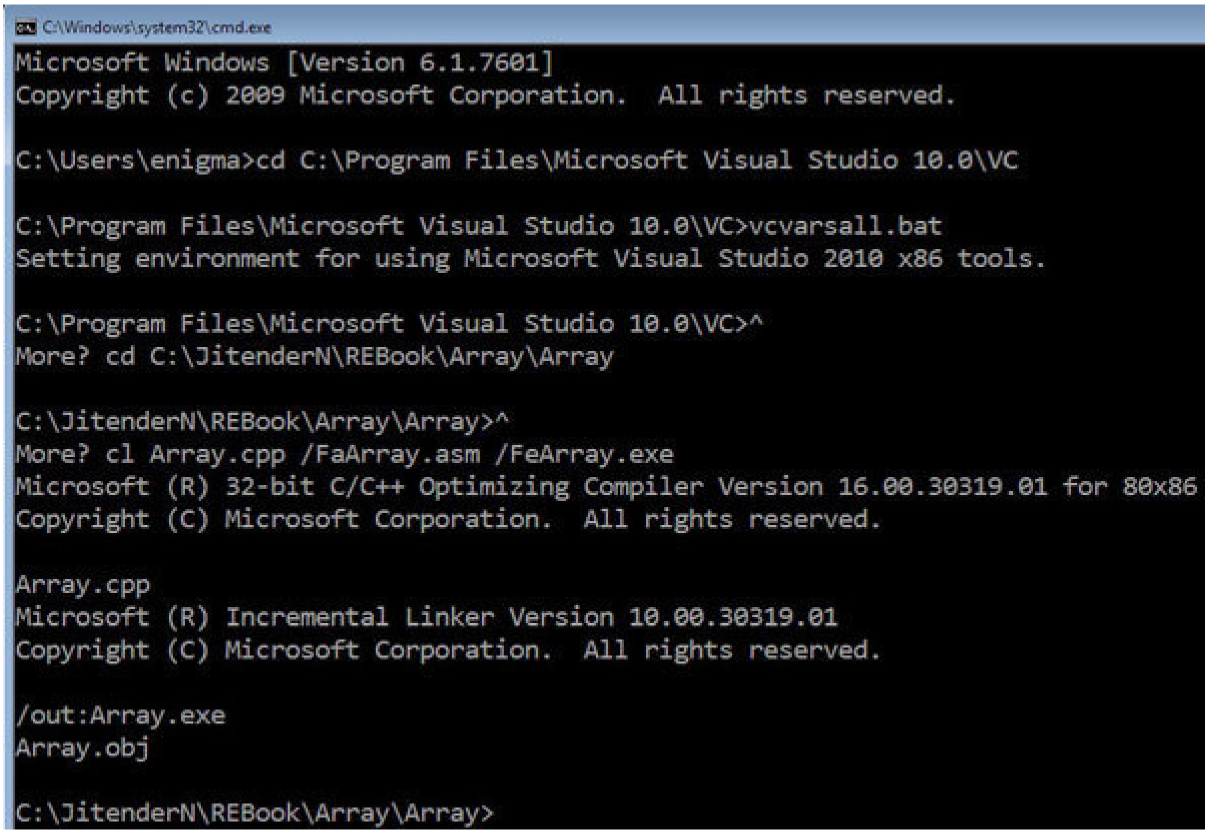
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:

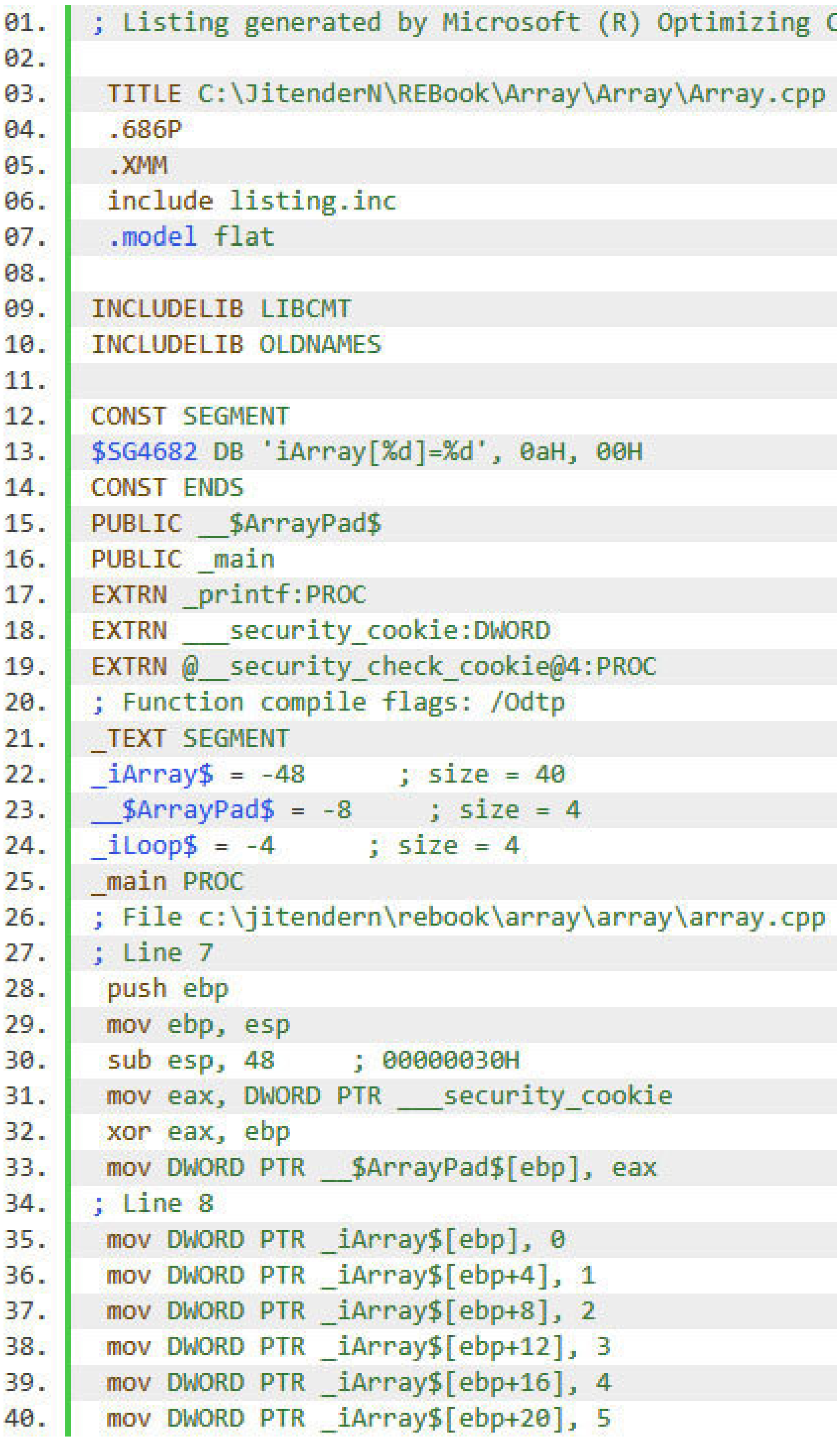


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Array\Array

C:\JitenderN\REBook\Array\Array>^
More? cl Array.cpp /FaArray.asm /FeArray.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Array.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Array.exe
Array.obj

C:\JitenderN\REBook\Array\Array>
```

Figure 11.2: *Array Loop without Optimization*

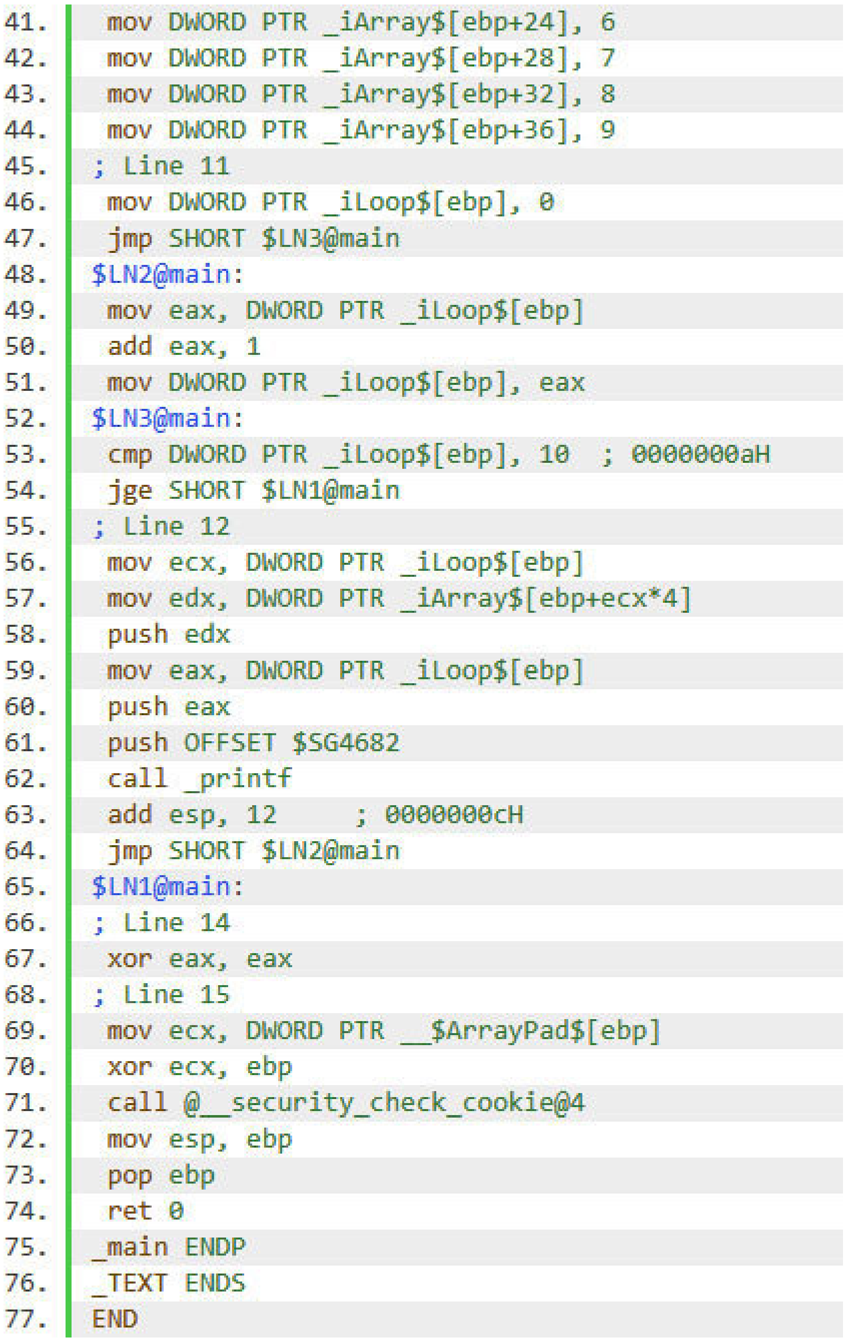
Compilation generates the EXE file and assembly code. Disable ASLR manually. To disable ASLR, use the CFF explorer and change **the DllCharacteristics** parameter to uncheck **DLL can** For detailed steps to understand how to disable ASLR, please refer to the Appendix.

Now, let's move on to the generated assembly listing:



```
; Listing generated by Microsoft (R) Optimizing C

 TITLE C:\JitenderN\REBook\Array\Array\Array.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4682 DB 'iArray[%d]=%d', 0aH, 00H
CONST ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_iArray$ = -48      ; size = 40
__$ArrayPad$ = -8      ; size = 4
_iLoop$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\array\array\array.cpp
; Line 7
 push ebp
 mov ebp, esp
 sub esp, 48     ; 00000030H
 mov eax, DWORD PTR ___security_cookie
 xor eax, ebp
 mov DWORD PTR __$ArrayPad$[ebp], eax
; Line 8
 mov DWORD PTR _iArray$[ebp], 0
 mov DWORD PTR _iArray$[ebp+4], 1
 mov DWORD PTR _iArray$[ebp+8], 2
 mov DWORD PTR _iArray$[ebp+12], 3
 mov DWORD PTR _iArray$[ebp+16], 4
 mov DWORD PTR _iArray$[ebp+20], 5
```

Figure 11.3: *Array.asm-Part-1*



```
 mov DWORD PTR _iArray$[ebp+24], 6
 mov DWORD PTR _iArray$[ebp+28], 7
 mov DWORD PTR _iArray$[ebp+32], 8
 mov DWORD PTR _iArray$[ebp+36], 9
; Line 11
 mov DWORD PTR _iLoop$[ebp], 0
 jmp SHORT $LN3@main
$LN2@main:
 mov eax, DWORD PTR _iLoop$[ebp]
 add eax, 1
 mov DWORD PTR _iLoop$[ebp], eax
$LN3@main:
 cmp DWORD PTR _iLoop$[ebp], 10  ; 0000000aH
 jge SHORT $LN1@main
; Line 12
 mov ecx, DWORD PTR _iLoop$[ebp]
 mov edx, DWORD PTR _iArray$[ebp+ecx*4]
 push edx
 mov eax, DWORD PTR _iLoop$[ebp]
 push eax
 push OFFSET $SG4682
 call _printf
 add esp, 12      ; 0000000cH
 jmp SHORT $LN2@main
$LN1@main:
; Line 14
 xor eax, eax
; Line 15
 mov ecx, DWORD PTR __$ArrayPad$[ebp]
 xor ecx, ebp
 call @__security_check_cookie@4
 mov esp, ebp
 pop ebp
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 11.4: *Array.asm-Part-2*

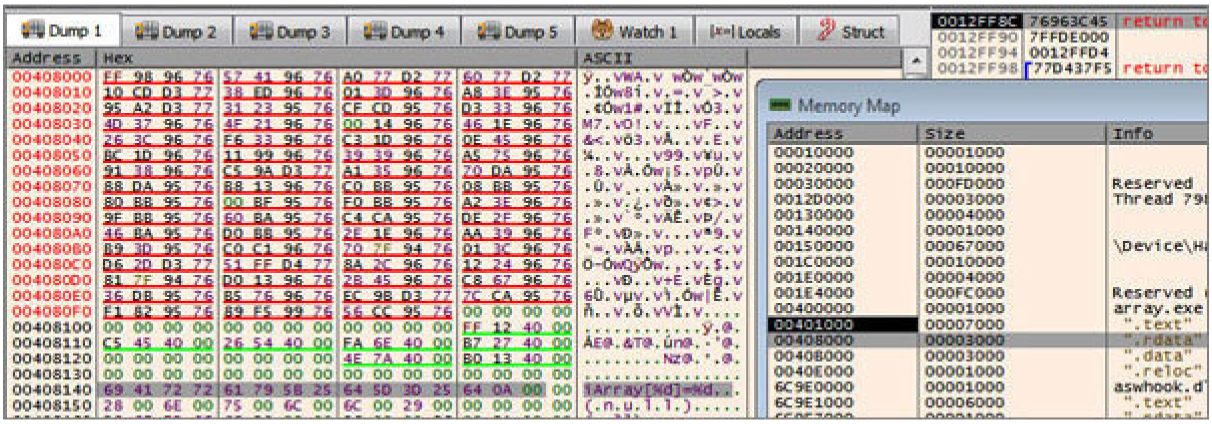
▼Line 12-14

CONST SEGMENT

$SG4682 DB 'iArray[%d]=%d', 0aH, 00H

CONST ENDS

The code defines two segments, one being **CONST** This derivative is used to define the start of the constant segment in the memory. The linker renames **CONST SEGMENT** to **.rdata** the code is placed in the **.code** segment, the constant string is placed in the **CONST(.rdata)** segment and if not constant, it is placed in the **.data** segment], which can be dumped using any debugger. Below screenshot shows **$SG4682** in the memory dump:



```
Address   Hex                                              ASCII
00408000  FF 98 96 76 57 41 96 76 A0 77 D2 77 60 77 D2 77  ÿ..vWA.v wÒw`wÒw
00408010  10 CD D3 77 38 ED 96 76 01 3D 96 76 A8 3E 95 76  .ÍÓw8í.v.=.v¨>.v
00408020  95 A2 D3 77 31 23 95 76 CF CD 95 76 D3 33 96 76  .¢Ów1#.vÏÍ.vÓ3.v
00408030  4D 37 96 76 4F 21 96 76 00 14 96 76 46 1E 96 76  M7.vO!.v...vF..v
00408040  26 3C 96 76 F6 33 96 76 C3 1D 96 76 0E 45 96 76  &<.vö3.vÃ..v.E.v
00408050  BC 1D 96 76 11 99 96 76 39 39 96 76 A5 75 96 76  ¼..v...v99.v¥u.v
00408060  91 38 96 76 C5 9A D3 77 A1 35 96 76 70 DA 95 76  .8.vÅ.Ów¡5.vpÚ.v
00408070  B8 DA 95 76 B8 13 96 76 C0 BB 95 76 08 BB 95 76  ¸Ú.v¸..vÀ».v.».v
00408080  80 BB 95 76 00 BF 95 76 F0 BB 95 76 A2 3E 96 76  .».v.¿.vð».v¢>.v
00408090  9F BB 95 76 60 BA 95 76 C4 CA 95 76 DE 2F 96 76  .».v`º.vÄÊ.vÞ/.v
004080A0  46 BA 95 76 D0 BB 95 76 2E 1E 96 76 AA 39 96 76  Fº.vÐ».v...vª9.v
004080B0  B9 3D 95 76 C0 C1 96 76 70 7F 94 76 D1 3C 96 76  ¹=.vÀÁ.vp..vÑ<.v
004080C0  D6 2D D3 77 51 FF D4 77 8A 2C 96 76 12 24 96 76  Ö-ÓwQÿÔw.,.v.$.v
004080D0  81 7F 94 76 D0 13 96 76 28 45 96 76 C8 67 96 76  ...vÐ..v+E.vÈg.v
004080E0  36 DB 95 76 B5 76 96 76 EC 9B D3 77 7C CA 95 76  6Û.vµv.vì.Ów|Ê.v
004080F0  F1 82 95 76 89 F5 99 76 56 CC 95 76 00 00 00 00  ñ..v.õ.vVÌ.v....
00408100  00 00 00 00 00 00 00 00 00 00 00 00 FF 12 40 00  ............ÿ.@.
00408110  C5 45 40 00 26 54 40 00 FA 6E 40 00 B7 27 40 00  ÅE@.&T@.ún@.·'@.
00408120  00 00 00 00 00 00 00 00 4E 7A 40 00 B0 13 40 00  ........Nz@.°.@.
00408130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00408140  69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
00408150  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
```

```
0012FF8C  76963C45  return t
0012FF90  7FFDE000
0012FF94  0012FFD4
0012FF98  77D437F5  return t
```

```
Memory Map
Address   Size      Info
00010000  00001000
00020000  00010000
00030000  000FD000  Reserved
0012D000  00003000  Thread 79
00130000  00004000
00140000  00001000
00150000  00067000  \Device\H
001C0000  00010000
001E0000  00004000
001E4000  000FC000  Reserved
00400000  00001000  array.exe
00401000  00007000  ".text"
00408000  00003000  ".rdata"
0040B000  00003000  ".data"
0040E000  00001000  ".reloc"
6C9E0000  00001000  aswhook.d
6C9E1000  00006000  ".text"
```

Figure 11.5: *.rdata*

The **$SG4682** is the internal name given by the compiler to handle the string constant. DB, defines the byte which is the data type. **'iArray[%d]=%d', 0aH, 00H** is the string data, which is null terminated ASCII string.

By **CONST** the constant segment is ended.

▼Line 15-16

PUBLIC __$ArrayPad$

```
PUBLIC _main
```

PUBLIC is the derivative which makes the **_main** procedure and **__$ArrayPad$** macro public, which makes it accessible to other modules.

▼Line 17-19

```
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
```

The **EXTRN** derivative declares the extern function, which is **printf** in our case. All functions begin with an underscore.

▼Line 21-24

```
_TEXT SEGMENT
_iArray$ = -48        ; size = 40
__$ArrayPad$ = -8       ; size = 4
_iLoop$ = -4        ; size = 4
```

This is the start of the **_TEXT** segment, where our main function code resides. After we have the different variable macro defined. To access the local variable on the stack frame, we have to add **_$** to the EBP address.

▼Line 25

```
_main PROC
```

This is the start of the **main** procedure.

▼Line 27-30

```
; Line 7
push ebp
mov ebp, esp
sub esp, 48; 00000030H
```

Line 7 of the C/C++ code starts the **main** function.

```
int main() {
```

The ASM code starts with the **main** function prologue of **PUSH** and **SUB** is creating room for variables on the stack by subtracting 48 (0x30) from ESP.

▼Line 31-33

```
mov eax, DWORD PTR ___security_cookie
xor eax, ebp
mov DWORD PTR __$ArrayPad$[ebp], eax
```

To understand all these three instructions, we will first have to understand the concept of stack cookie. A stack cookie is the protection mechanism against buffer overflow attacks. To understand buffer overflow in layman terms, imagine somebody pours more coffee in a cup; the coffee poured in excess will spill on the table. Similarly, when a buffer stored in memory is filled with more data than its size, the excess data will be spilled causing the critical memory locations to be overwritten. A stack cookie is a random value generated at each execution. This value is XOR'ed with EBP and then placed on the stack. This value stored on the stack is placed between local variables (buffer or array in our case) and EBP.

Figure 11.6: *Stack cookie*

Once this value is stored on the stack after the function prologue, to prevent against buffer overflow attacks, this value is checked before the function epilogue. If this value is not the same before the function epilogue, then it means that a buffer overflow exploit has occurred.

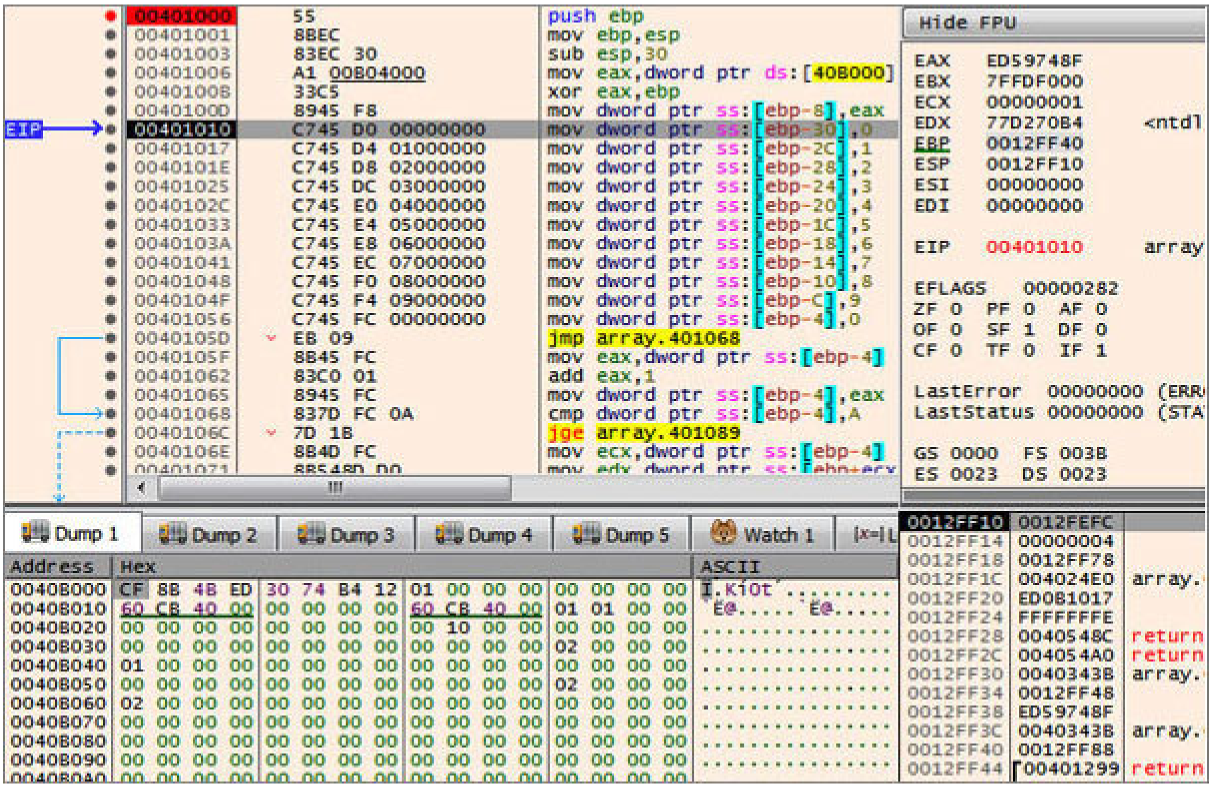
Now coming back to the first **MOV** instruction, it moves the value stored at the stack cookie location to EAX. EAX is XOR'ed with EBP and the result of XOR is stored in EAX. This XOR value stored in EAX is moved to the stack at **ss:[ebp-ox8]** location. This is where our stack cookie is stored on the stack. Let's see the stack state and the stack cookie value in the memory dump.

Stack Cookie stored at **0x0040B000** location is = **0xED4B8BCF**

EBP = **0x0012FF40**

Stack Cookie XOR EBP = **0xED4B8BCF** XOR **0x0012FF40** = **0xED59748F**

```
[ESP]   0012FF10  0012FEFC [EBP-0x30] JUNK Right Now
[ESP+0x4] 0012FF14  00000004 [EBP-0x2C] JUNK Right Now
[ESP+0x8] 0012FF18  0012FF78 [EBP-0x28] JUNK Right Now
[ESP+0xC] 0012FF1C  004024E0 [EBP-0x24] JUNK Right Now
[ESP+0x10] 0012FF20  ED0B1017 [EBP-0x20] JUNK Right Now
[ESP+0x14] 0012FF24  FFFFFFFE [EBP-0x1C] JUNK Right Now
[ESP+0x18] 0012FF28  0040548C [EBP-0x18] JUNK Right Now
[ESP+0x1C] 0012FF2C  004054A0 [EBP-0x14]  JUNK Right Now
[ESP+0x20] 0012FF30  0040343B [EBP-0x10] JUNK Right Now
[ESP+0x24] 0012FF34  0012FF48 [EBP-0x0C] JUNK Right Now
[ESP+0x28] 0012FF38  ED59748F [EBP-0x08] Stack Cookie xor EBP value placed here
[ESP+0x2C] 0012FF3C  0040343B [EBP-0x04] JUNK Right Now
[ESP+0x30] 0012FF40  0012FF88 [EBP]
[ESP+0x34] 0012FF44  00401299 return to array.00401299 from array.00401000
```



```
00401000  55                 push ebp
00401001  8BEC               mov ebp,esp
00401003  83EC 30            sub esp,30
00401006  A1 00B04000        mov eax,dword ptr ds:[40B000]
0040100B  33C5               xor eax,ebp
0040100D  8945 F8            mov dword ptr ss:[ebp-8],eax
00401010  C745 D0 00000000   mov dword ptr ss:[ebp-30],0
00401017  C745 D4 01000000   mov dword ptr ss:[ebp-2C],1
0040101E  C745 D8 02000000   mov dword ptr ss:[ebp-28],2
00401025  C745 DC 03000000   mov dword ptr ss:[ebp-24],3
0040102C  C745 E0 04000000   mov dword ptr ss:[ebp-20],4
00401033  C745 E4 05000000   mov dword ptr ss:[ebp-1C],5
0040103A  C745 E8 06000000   mov dword ptr ss:[ebp-18],6
00401041  C745 EC 07000000   mov dword ptr ss:[ebp-14],7
00401048  C745 F0 08000000   mov dword ptr ss:[ebp-10],8
0040104F  C745 F4 09000000   mov dword ptr ss:[ebp-C],9
00401056  C745 FC 00000000   mov dword ptr ss:[ebp-4],0
0040105D  EB 09              jmp array.401068
0040105F  8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401062  83C0 01            add eax,1
00401065  8945 FC            mov dword ptr ss:[ebp-4],eax
00401068  837D FC 0A         cmp dword ptr ss:[ebp-4],A
0040106C  7D 1B              jge array.401089
0040106E  8B4D FC            mov ecx,dword ptr ss:[ebp-4]
```

```
EAX   ED59748F
EBX   7FFDF000
ECX   00000001
EDX   77D270B4      <ntdl
EBP   0012FF40
ESP   0012FF10
ESI   00000000
EDI   00000000

EIP   00401010      array

EFLAGS   00000282
ZF 0  PF 0  AF 0
OF 0  SF 1  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
```

```
Address   Hex                                              ASCII
0040B000  CF 8B 4B ED 30 74 B4 12 01 00 00 00 00 00 00 00  Ï.KíOt´.........
0040B010  60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00  `Ë@.....`Ë@.....
0040B020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
0040B030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FF10  0012FEFC
0012FF14  00000004
0012FF18  0012FF78
0012FF1C  004024E0  array.
0012FF20  ED081017
0012FF24  FFFFFFFE
0012FF28  0040548C  return
0012FF2C  004054A0  return
0012FF30  0040343B  array.
0012FF34  0012FF48
0012FF38  ED59748F
0012FF3C  0040343B  array.
0012FF40  0012FF88
0012FF44  00401299  return
```

__Figure 11.7:__ Stack cookie on stack

▼Line 34-44

```
; Line 8
mov DWORD PTR _iArray$[ebp], 0
mov DWORD PTR _iArray$[ebp+4], 1
mov DWORD PTR _iArray$[ebp+8], 2
mov DWORD PTR _iArray$[ebp+12], 3
mov DWORD PTR _iArray$[ebp+16], 4
mov DWORD PTR _iArray$[ebp+20], 5
mov DWORD PTR _iArray$[ebp+24], 6
mov DWORD PTR _iArray$[ebp+28], 7
mov DWORD PTR _iArray$[ebp+32], 8
mov DWORD PTR _iArray$[ebp+36], 9
```

Line 8 of the C/C++ code defines **iArray** of type

```
int iArray[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

In ASM, **iArray** is of type integer, so each element of the array will occupy 4 bytes of space as required for that integer. All these move instructions will place the elements of **iArray** on the stack. **iArray** macro is defined as:

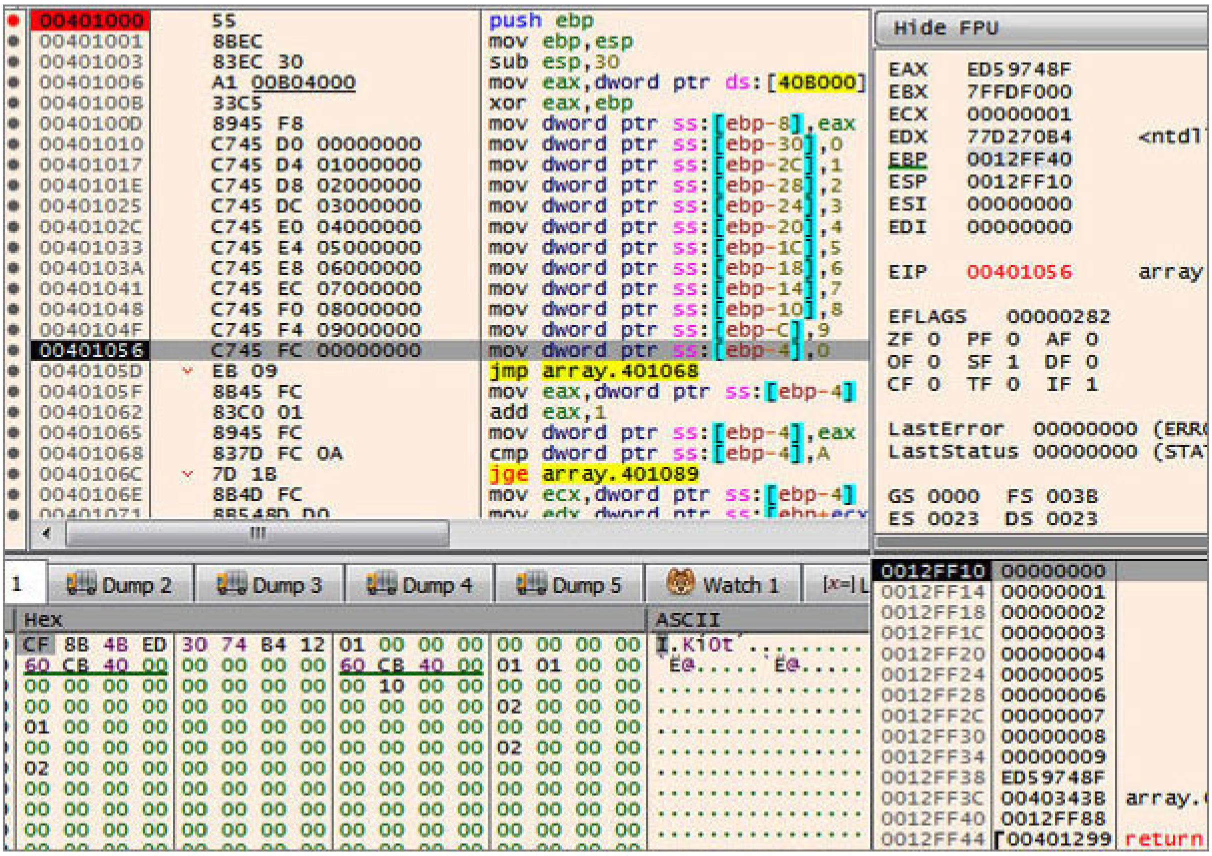
_iArray$ = -48 (-0x30)

So **_iArray$[ebp]** will correspond to **ss:[ebp-0x30].** Similarly, other **iArray** elements can be accessed on the stack by adding **_$** to the EBP address. The stack state after the execution of the preceding instructions is as follows:

[ESP] 0012FF10 00000000 [EBP-0x30] iArray first element
[ESP+0x4] 0012FF14 00000001 [EBP-0x2C] iArray second element
[ESP+0x8] 0012FF18 00000002 [EBP-0x28] iArray third element
[ESP+0xC] 0012FF1C 00000003 [EBP-0x24] iArray forth element
[ESP+0x10] 0012FF20 00000004 [EBP-0x20] iArray fifth element
[ESP+0x14] 0012FF24 00000005 [EBP-0x1C] iArray sixth element
[ESP+0x18] 0012FF28 00000006 [EBP-0x18] iArray seventh element
[ESP+0x1C] 0012FF2C 00000007 [EBP-0x14] iArray eighth element
[ESP+0x20] 0012FF30 00000008 [EBP-0x10] iArray ninth element
[ESP+0x24] 0012FF34 00000009 [EBP-0x0C] iArray tenth element
[ESP+0x28] 0012FF38 ED59748F [EBP-0x08] Stack Cookie xor EBP value placed

[ESP+0x2C] 0012FF3C 0040343B [EBP-0x04] JUNK Right Now

```
[ESP+0x30] 0012FF40   0012FF88   [EBP]
[ESP+0x34] 0012FF44   00401299   return to array.00401299 from array.00401000
```



```
00401000    55                  push ebp
00401001    8BEC                mov ebp,esp
00401003    83EC 30             sub esp,30
00401006    A1 00B04000         mov eax,dword ptr ds:[40B000]
0040100B    33C5                xor eax,ebp
0040100D    8945 F8             mov dword ptr ss:[ebp-8],eax
00401010    C745 D0 00000000    mov dword ptr ss:[ebp-30],0
00401017    C745 D4 01000000    mov dword ptr ss:[ebp-2C],1
0040101E    C745 D8 02000000    mov dword ptr ss:[ebp-28],2
00401025    C745 DC 03000000    mov dword ptr ss:[ebp-24],3
0040102C    C745 E0 04000000    mov dword ptr ss:[ebp-20],4
00401033    C745 E4 05000000    mov dword ptr ss:[ebp-1C],5
0040103A    C745 E8 06000000    mov dword ptr ss:[ebp-18],6
00401041    C745 EC 07000000    mov dword ptr ss:[ebp-14],7
00401048    C745 F0 08000000    mov dword ptr ss:[ebp-10],8
0040104F    C745 F4 09000000    mov dword ptr ss:[ebp-C],9
00401056    C745 FC 00000000    mov dword ptr ss:[ebp-4],0
0040105D  v EB 09               jmp array.401068
0040105F    8B45 FC             mov eax,dword ptr ss:[ebp-4]
00401062    83C0 01             add eax,1
00401065    8945 FC             mov dword ptr ss:[ebp-4],eax
00401068    837D FC 0A          cmp dword ptr ss:[ebp-4],A
0040106C  v 7D 1B               jge array.401089
0040106E    8B4D FC             mov ecx,dword ptr ss:[ebp-4]
```

```
Hide FPU

EAX   ED59748F
EBX   7FFDF000
ECX   00000001
EDX   77D270B4       <ntdl
EBP   0012FF40
ESP   0012FF10
ESI   00000000
EDI   00000000

EIP   00401056       array

EFLAGS     00000282
ZF 0  PF 0  AF 0
OF 0  SF 1  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
```

```
Hex                                                  ASCII
CF 8B 4B ED 30 74 B4 12 01 00 00 00 00 00 00 00      I.K10t'.........
60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00      `E@.....`E@.....
00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00      ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00      ................
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00      ................
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      ................
```

```
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  0040343B  array.
0012FF40  0012FF88
0012FF44  00401299  return
```

Figure 11.8: *iArray on stack*

▼Line 35-47

```
; Line 11
mov DWORD PTR _iLoop$[ebp], 0
jmp SHORT $LN3@main
```

Line 11 of the C/C++ code initializes the **iLoop** variable to iterate over the **iArray** elements. The **MOV** instruction is placing **0x00** on the **iLoop** variable placeholder on the stack at **ss:[ebp-0x4].** The

JMP instruction makes an unconditional jump to the **$LN3@main** label.

▼Line 52-54

```
$LN3@main:

cmp DWORD PTR _iLoop$[ebp], 10   ; 0000000aH
jge SHORT $LN1@main
```

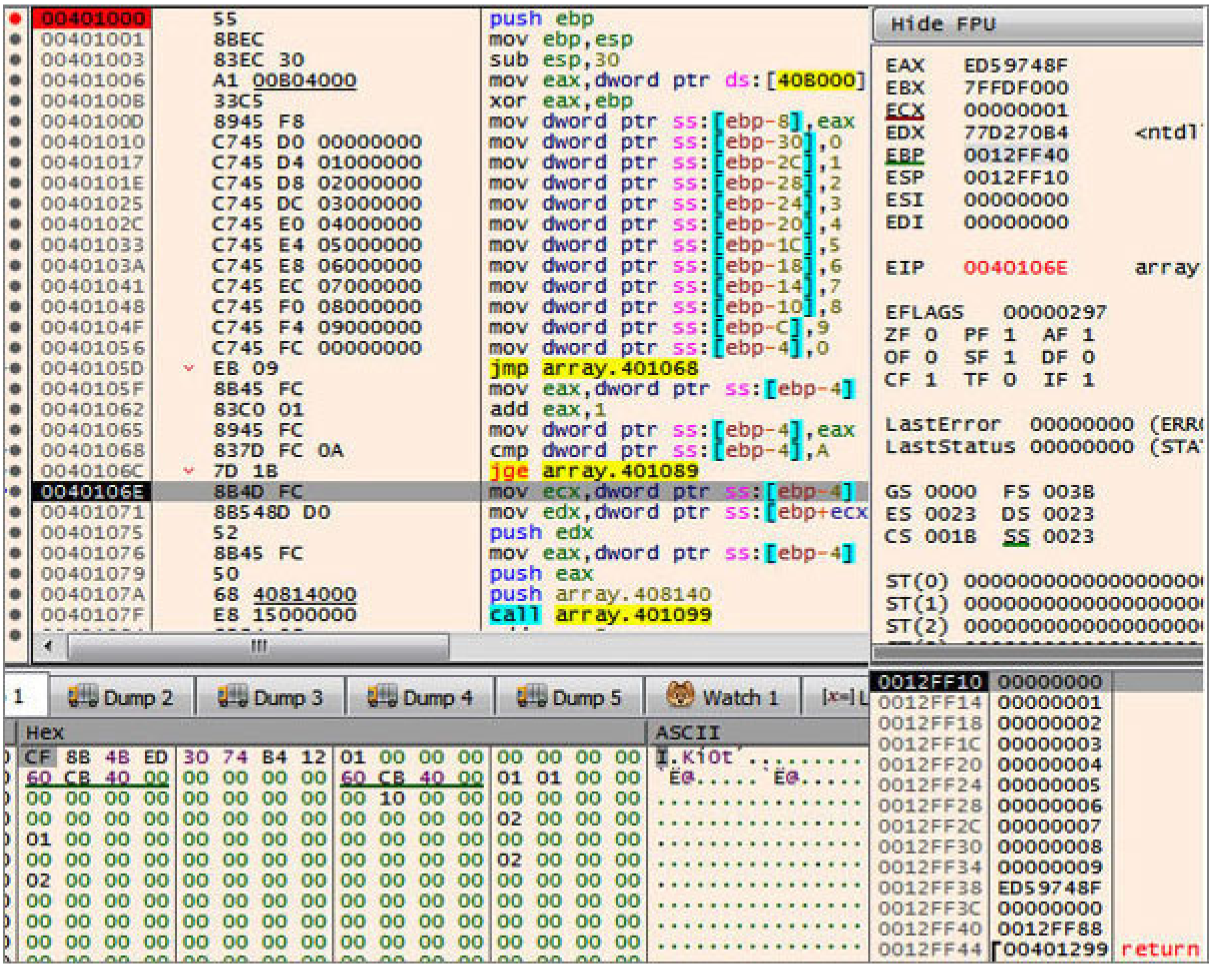
The **CMP** instruction compares the **iLoop** value stored on the stack with **10(0x0A).** It will perform a signed comparison jump if the **iLoop** at **[EBP-0x04]** is greater than or equal to **10(0x0A).** Currently, **iLoop** is 0, so the jump will not happen and the instruction pointer will move to the next instruction. The stack state will be as follows:

[ESP] 0012FF10 00000000 [EBP-0x30] iArray first element
[ESP+0x4] 0012FF14 00000001 [EBP-0x2C] iArray second element
[ESP+0x8] 0012FF18 00000002 [EBP-0x28] iArray third element
[ESP+0xC] 0012FF1C 00000003 [EBP-0x24] iArray forth element
[ESP+0x10] 0012FF20 00000004 [EBP-0x20] iArray fifth element
[ESP+0x14] 0012FF24 00000005 [EBP-0x1C] iArray sixth element
[ESP+0x18] 0012FF28 00000006 [EBP-0x18] iArray seventh element
[ESP+0x1C] 0012FF2C 00000007 [EBP-0x14] iArray eighth element
[ESP+0x20] 0012FF30 00000008 [EBP-0x10] iArray ninth element
[ESP+0x24] 0012FF34 00000009 [EBP-0x0C] iArray tenth element
[ESP+0x28] 0012FF38 ED59748F [EBP-0x08] Stack Cookie xor EBP value placed

[ESP+0x2C] 0012FF3C 00000000 [EBP-0x04] iLoop placeholder here

[ESP+0x30] 0012FF40 0012FF88 [EBP]

[ESP+0x34] 0012FF44 00401299 return to array.00401299 from array.00401000



```
00401000   55                 push ebp
00401001   8BEC               mov ebp,esp
00401003   83EC 30            sub esp,30
00401006   A1 00804000        mov eax,dword ptr ds:[408000]
0040100B   33C5               xor eax,ebp
0040100D   8945 F8            mov dword ptr ss:[ebp-8],eax
00401010   C745 D0 00000000   mov dword ptr ss:[ebp-30],0
00401017   C745 D4 01000000   mov dword ptr ss:[ebp-2C],1
0040101E   C745 D8 02000000   mov dword ptr ss:[ebp-28],2
00401025   C745 DC 03000000   mov dword ptr ss:[ebp-24],3
0040102C   C745 E0 04000000   mov dword ptr ss:[ebp-20],4
00401033   C745 E4 05000000   mov dword ptr ss:[ebp-1C],5
0040103A   C745 E8 06000000   mov dword ptr ss:[ebp-18],6
00401041   C745 EC 07000000   mov dword ptr ss:[ebp-14],7
00401048   C745 F0 08000000   mov dword ptr ss:[ebp-10],8
0040104F   C745 F4 09000000   mov dword ptr ss:[ebp-C],9
00401056   C745 FC 00000000   mov dword ptr ss:[ebp-4],0
0040105D   EB 09              jmp array.401068
0040105F   8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401062   83C0 01            add eax,1
00401065   8945 FC            mov dword ptr ss:[ebp-4],eax
00401068   837D FC 0A         cmp dword ptr ss:[ebp-4],A
0040106C   7D 1B              jge array.401089
0040106E   8B4D FC            mov ecx,dword ptr ss:[ebp-4]
00401071   8B548D D0          mov edx,dword ptr ss:[ebp+ecx
00401075   52                 push edx
00401076   8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401079   50                 push eax
0040107A   68 40814000        push array.408140
0040107F   E8 15000000        call array.401099
```

```
EAX   ED59748F
EBX   7FFDF000
ECX   00000001
EDX   77D270B4      <ntdl
EBP   0012FF40
ESP   0012FF10
ESI   00000000
EDI   00000000

EIP   0040106E      array

EFLAGS    00000297
ZF 0   PF 1   AF 1
OF 0   SF 1   DF 0
CF 1   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
```

```
CF 8B 4B ED 30 74 B4 12 01 00 00 00 00 00 00 00  I.Kíot´........
60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00  `Ë@.....`Ë@.....
00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  00000000
0012FF40  0012FF88
0012FF44  00401299  return
```

Figure 11.9: *iLoop is 0*

▼Line 55-64

```
; Line 12
mov ecx, DWORD PTR _iLoop$[ebp]
mov edx, DWORD PTR _iArray$[ebp+ecx*4]
```

```
push edx
mov eax, DWORD PTR _iLoop$[ebp]
push eax
push OFFSET $SG4682

call _printf
add esp, 12        ; 0000000cH
jmp SHORT $LN2@main
```

Line 12 of the C/C++ code prints the element of **iArray** array along with the index on the screen.

```
printf ("iArray[%d]=%d\n", iLoop, iArray[iLoop]);
```

As the **printf** function takes three arguments to print on the screen, all these three arguments need to be pushed onto the stack before the call to the **printf** function. In the ASM code, the first argument to be pushed on the stack will be **iArray[iLoop].** To push this value on the stack, the **iLoop** value stored at **[EBP-0x04]** is moved to **ECX** is the current value of array index. It is multiplied by 4 (the size of the integer) and then added to the **iArray** macro to calculate the memory location of the element stored for the particular array index. This will give us the value of the first array element in the **EDX** register. So, by pushing **EDX** onto the stack, we pushed the first argument to **printf** onto the stack.

Next argument, **iLoop** is pushed onto the stack by first pushing the **iLoop** value stored at **[EBP-0x04]** to and the push **EAX** which is second argument to

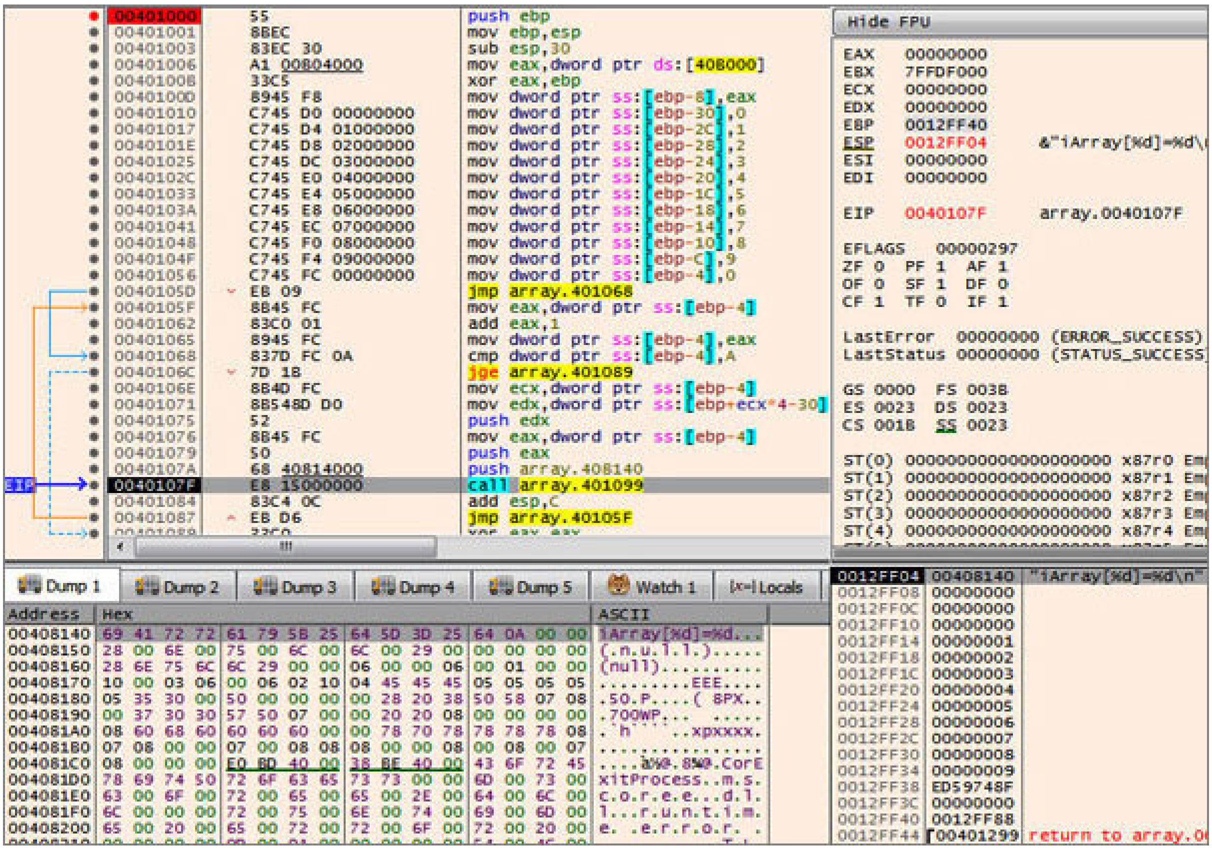
Last argument, the string constant is pushed by **push OFFSET**

Now, the three arguments are on the stack. So a call to **printf** is made. The stack state at this point in time is as follows:

[ESP] 0012FF04 00408140 [EBP-0x40] "iArray[%d]=%d\n", argument to printf
[ESP+0x4] 0012FF08 00000000 [EBP-0x38] iLoop value is pushed as printf argument

[ESP+0x8] 0012FF0C 00000000 [EBP-0x34] iArray first element pushed as printf arg
ESP+0xC] 0012FF10 00000000 [EBP-0x30] iArray first element
[ESP+0x10] 0012FF14 00000001 [EBP-0x2C] iArray second element
[ESP+0x14] 0012FF18 00000002 [EBP-0x28] iArray third element
[ESP+0x18] 0012FF1C 00000003 [EBP-0x24] iArray forth element
[ESP+0x1C] 0012FF20 00000004 [EBP-0x20] iArray fifth element
[ESP+0x20] 0012FF24 00000005 [EBP-0x1C] iArray sixth element
[ESP+0x24] 0012FF28 00000006 [EBP-0x18] iArray seventh element
[ESP+0x28] 0012FF2C 00000007 [EBP-0x14] iArray eighth element
[ESP+0x2C] 0012FF30 00000008 [EBP-0x10] iArray ninth element
[ESP+0x30] 0012FF34 00000009 [EBP-0x0C] iArray tenth element
[ESP+0x34] 0012FF38 ED59748F [EBP-0x08] Stack Cookie xor EBP value placed
[ESP+0x38] 0012FF3C 00000000 [EBP-0x04] iLoop placeholder here
[ESP+0x3C] 0012FF40 0012FF88 [EBP]

[ESP+0x40] 0012FF44 00401299 return to array.00401299 from array.00401000



```
00401000   55                 push ebp
00401001   8BEC               mov ebp,esp
00401003   83EC 30            sub esp,30
00401006   A1 00804000        mov eax,dword ptr ds:[408000]
0040100B   33C5               xor eax,ebp
0040100D   8945 F8            mov dword ptr ss:[ebp-8],eax
00401010   C745 D0 00000000   mov dword ptr ss:[ebp-30],0
00401017   C745 D4 01000000   mov dword ptr ss:[ebp-2C],1
0040101E   C745 D8 02000000   mov dword ptr ss:[ebp-28],2
00401025   C745 DC 03000000   mov dword ptr ss:[ebp-24],3
0040102C   C745 E0 04000000   mov dword ptr ss:[ebp-20],4
00401033   C745 E4 05000000   mov dword ptr ss:[ebp-1C],5
0040103A   C745 E8 06000000   mov dword ptr ss:[ebp-18],6
00401041   C745 EC 07000000   mov dword ptr ss:[ebp-14],7
00401048   C745 F0 08000000   mov dword ptr ss:[ebp-10],8
0040104F   C745 F4 09000000   mov dword ptr ss:[ebp-C],9
00401056   C745 FC 00000000   mov dword ptr ss:[ebp-4],0
0040105D   EB 09              jmp array.401068
0040105F   8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401062   83C0 01            add eax,1
00401065   8945 FC            mov dword ptr ss:[ebp-4],eax
00401068   837D FC 0A         cmp dword ptr ss:[ebp-4],A
0040106C   7D 1B              jge array.401089
0040106E   8B4D FC            mov ecx,dword ptr ss:[ebp-4]
00401071   8B548D D0          mov edx,dword ptr ss:[ebp+ecx*4-30]
00401075   52                 push edx
00401076   8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401079   50                 push eax
0040107A   68 40814000        push array.408140
0040107F   E8 15000000        call array.401099
00401084   83C4 0C            add esp,C
00401087   EB D6              jmp array.40105F
```

```
EAX     00000000
EBX     7FFDF000
ECX     00000000
EDX     00000000
EBP     0012FF40
ESP     0012FF04      &"iArray[%d]=%d\
ESI     00000000
EDI     00000000

EIP     0040107F      array.0040107F

EFLAGS    00000297
ZF 0  PF 1  AF 1
OF 0  SF 1  DF 0
CF 1  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000000 x87r0 Em
ST(1) 0000000000000000000000 x87r1 Em
ST(2) 0000000000000000000000 x87r2 Em
ST(3) 0000000000000000000000 x87r3 Em
ST(4) 0000000000000000000000 x87r4 Em
```

```
Address   Hex                                               ASCII
00408140  69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
00408150  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
00408160  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
00408170  10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
00408180  05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00408190  00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
004081A0  08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
004081B0  07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
004081C0  08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
004081D0  78 69 74 50 72 6F 63 65 73 73 00 00 6D 00 73 00  xitProcess..m.s.
004081E0  63 00 6F 00 72 00 65 00 65 00 2E 00 64 00 6C 00  c.o.r.e.e...d.l.
004081F0  6C 00 00 00 72 00 75 00 6E 00 74 00 69 00 6D 00  l...r.u.n.t.i.m.
00408200  65 00 20 00 65 00 72 00 72 00 6F 00 72 00 20 00  e. .e.r.r.o.r. .
```

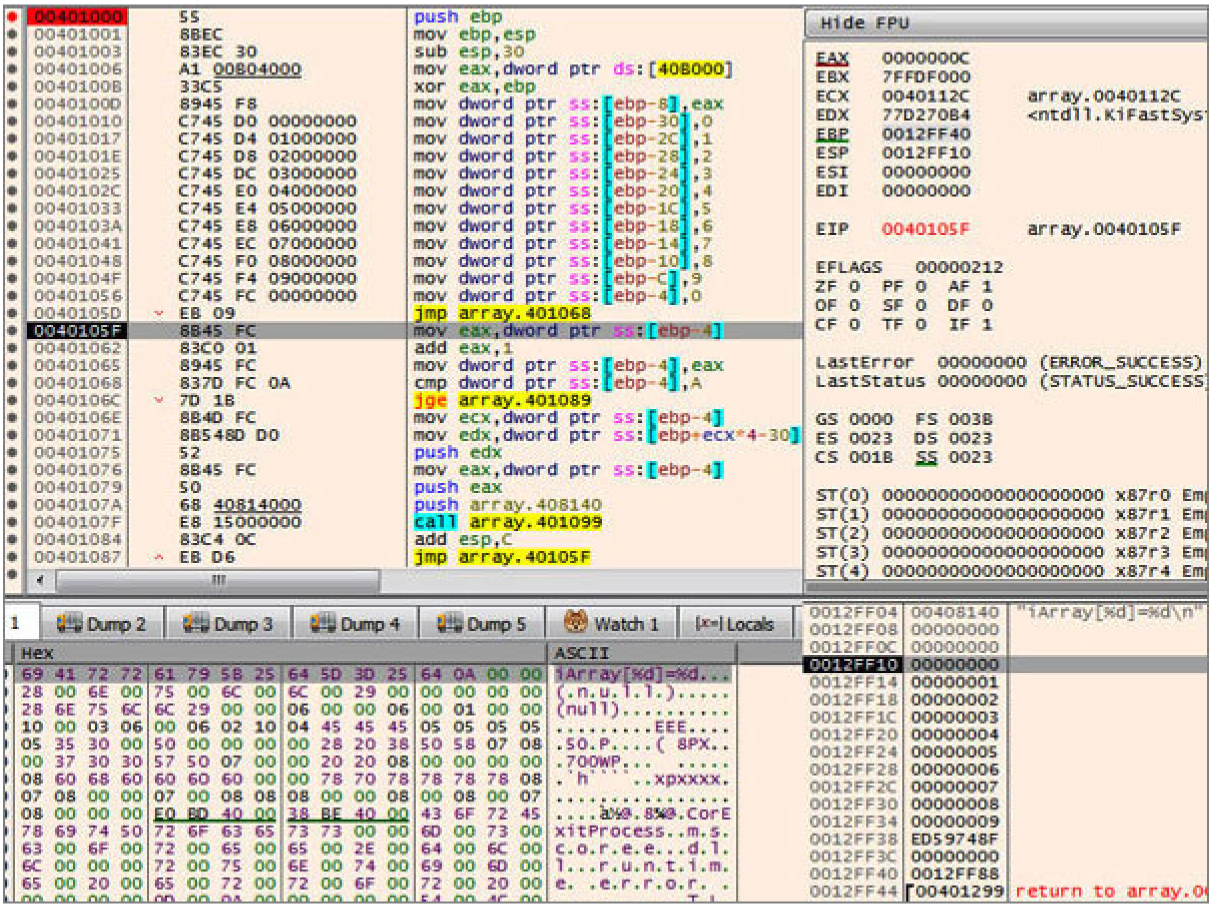
```
0012FF04  00408140  "iArray[%d]=%d\n"
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  00000000
0012FF40  0012FF88
0012FF44  00401299  return to array.0
```

Figure 11.10: Printf arg on stack

On return from call to the **printf** function, the stack is cleaned by 12 bytes by the **ADD** instruction. Next, an unconditional jump to **$LN2@main** label is made. The stack state after the unconditional jump is as follows:

[ESP-0x0C] 0012FF04 00408140 [EBP-0x40] NOW JUNK
[ESP-0x8] 0012FF08 00000000 [EBP-0x38] NOW JUNK
[ESP-0x4] 0012FF0C 00000000 [EBP-0x34] NOW JUNK
[ESP] 0012FF10 00000000 [EBP-0x30] iArray first element
[ESP+0x4] 0012FF14 00000001 [EBP-0x2C] iArray second element

```
[ESP+0x8] 0012FF18   00000002   [EBP-0x28] iArray third element
[ESP+0xC] 0012FF1C   00000003   [EBP-0x24] iArray forth element
[ESP+0x10] 0012FF20   00000004   [EBP-0x20] iArray fifth element

[ESP+0x14] 0012FF24   00000005   [EBP-0x1C] iArray sixth element
[ESP+0x18] 0012FF28   00000006   [EBP-0x18] iArray seventh element
[ESP+0x1C] 0012FF2C   00000007   [EBP-0x14]  iArray eighth element
[ESP+0x20] 0012FF30   00000008   [EBP-0x10] iArray ninth element
[ESP+0x24] 0012FF34   00000009   [EBP-0x0C] iArray tenth element
[ESP+0x28] 0012FF38   ED59748F   [EBP-0x08] Stack Cookie xor EBP value placed
[ESP+0x2C] 0012FF3C   00000000   [EBP-0x04] iLoop placeholder here
[ESP+0x30] 0012FF40   0012FF88   [EBP]
[ESP+0x34] 0012FF44   00401299   return to array.00401299 from array.00401000
```



```
00401000    55              push ebp
00401001    8BEC            mov ebp,esp
00401003    83EC 30         sub esp,30
00401006    A1 00804000     mov eax,dword ptr ds:[408000]
0040100B    33C5            xor eax,ebp
0040100D    8945 F8         mov dword ptr ss:[ebp-8],eax
00401010    C745 D0 00000000 mov dword ptr ss:[ebp-30],0
00401017    C745 D4 01000000 mov dword ptr ss:[ebp-2C],1
0040101E    C745 D8 02000000 mov dword ptr ss:[ebp-28],2
00401025    C745 DC 03000000 mov dword ptr ss:[ebp-24],3
0040102C    C745 E0 04000000 mov dword ptr ss:[ebp-20],4
00401033    C745 E4 05000000 mov dword ptr ss:[ebp-1C],5
0040103A    C745 E8 06000000 mov dword ptr ss:[ebp-18],6
00401041    C745 EC 07000000 mov dword ptr ss:[ebp-14],7
00401048    C745 F0 08000000 mov dword ptr ss:[ebp-10],8
0040104F    C745 F4 09000000 mov dword ptr ss:[ebp-C],9
00401056    C745 FC 00000000 mov dword ptr ss:[ebp-4],0
0040105D  v EB 09           jmp array.401068
0040105F    8B45 FC         mov eax,dword ptr ss:[ebp-4]
00401062    83C0 01         add eax,1
00401065    8945 FC         mov dword ptr ss:[ebp-4],eax
00401068    837D FC 0A      cmp dword ptr ss:[ebp-4],A
0040106C  v 7D 1B           jge array.401089
0040106E    8B4D FC         mov ecx,dword ptr ss:[ebp-4]
00401071    8B548D D0       mov edx,dword ptr ss:[ebp+ecx*4-30]
00401075    52              push edx
00401076    8B45 FC         mov eax,dword ptr ss:[ebp-4]
00401079    50              push eax
0040107A    68 40814000     push array.408140
0040107F    E8 15000000     call array.401099
00401084    83C4 0C         add esp,C
00401087  ^ EB D6           jmp array.40105F
```

```
Hide FPU

EAX   0000000C
EBX   7FFDF000
ECX   0040112C     array.0040112C
EDX   77D270B4     <ntdll.KiFastSys
EBP   0012FF40
ESP   0012FF10
ESI   00000000
EDI   00000000

EIP   0040105F     array.0040105F

EFLAGS    00000212
ZF 0  PF 0  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Em
ST(1) 00000000000000000000 x87r1 Em
ST(2) 00000000000000000000 x87r2 Em
ST(3) 00000000000000000000 x87r3 Em
ST(4) 00000000000000000000 x87r4 Em
```

```
Hex                                                    ASCII
69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
78 69 74 50 72 6F 63 65 73 73 00 00 6D 00 73 00  xitProcess..m.s.
63 00 6F 00 72 00 65 00 65 00 2E 00 64 00 6C 00  c.o.r.e.e...d.l.
6C 00 00 00 72 00 75 00 6E 00 74 00 69 00 6D 00  l...r.u.n.t.i.m.
65 00 20 00 65 00 72 00 72 00 6F 00 72 00 20 00  e. .e.r.r.o.r. .
```

```
0012FF04  00408140  "iArray[%d]=%d\n"
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  00000000
0012FF40  0012FF88
0012FF44 Γ00401299  return to array.0
```

Figure 11.11: Stack cleaned

▼Line 48-51

```
$LN2@main:
mov eax, DWORD PTR _iLoop$[ebp]
add eax, 1
mov DWORD PTR _iLoop$[ebp], eax
```

At this label, the **iLoop** value from **[EBP-0x04]** is moved to **EAX** and incremented by 1. This increment is placed back to **[EBP-0x4].** The instruction pointer will again iterate over the label **$LN3@main** to compare the **iLoop** value with **10(0x0A).**

▼Line 52-54

```
$LN3@main:
cmp DWORD PTR _iLoop$[ebp], 10  ; 0000000aH
jge SHORT $LN1@main
```

Now, imagine that we iterated 9 times over the instructions and the **iLoop** value has become **10** At this point, the **CMP** instruction will perform a signed comparison jump as the **iLoop** at **[EBP-0x04]** is equal to **10(0x0A)** and it will also set ZF=1. As **iLoop** is **10(0xA),** so the jump will happen to the **$LN1@main** label. The stack state after **JUMP if greater equal** is as follows:

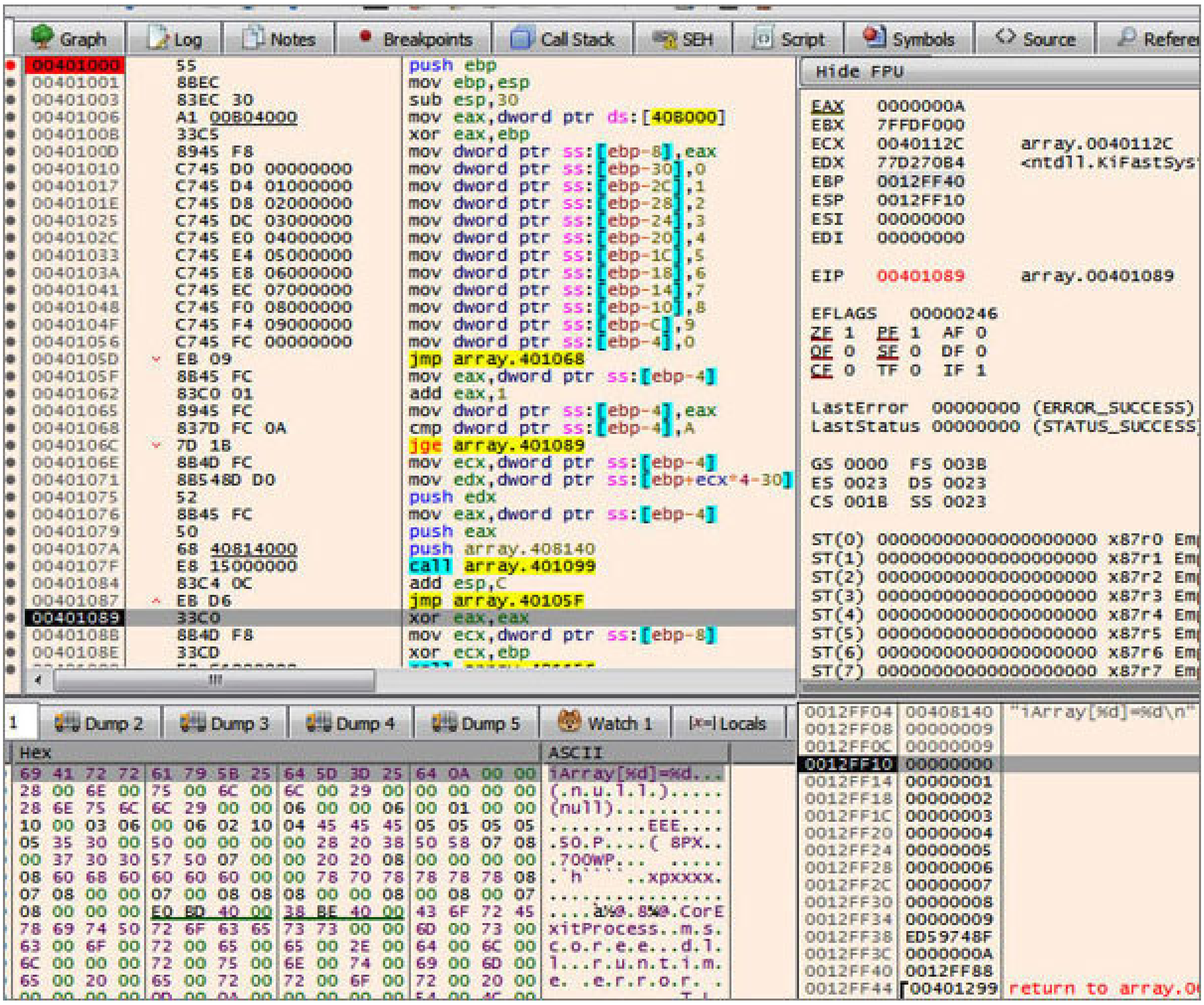
```
[ESP-0x0C] 0012FF04  00408140  [EBP-0x40] NOW JUNK
[ESP-0x8] 0012FF08  00000009  [EBP-0x38] NOW JUNK
[ESP-0x4] 0012FF0C  00000009  [EBP-0x34] NOW JUNK
[ESP]  0012FF10  00000000  [EBP-0x30] iArray first element
[ESP+0x4] 0012FF14  00000001  [EBP-0x2C] iArray second element
[ESP+0x8] 0012FF18  00000002  [EBP-0x28] iArray third element

[ESP+0xC] 0012FF1C  00000003  [EBP-0x24] iArray forth element
[ESP+0x10] 0012FF20  00000004  [EBP-0x20] iArray fifth element
[ESP+0x14] 0012FF24  00000005  [EBP-0x1C] iArray sixth element
[ESP+0x18] 0012FF28  00000006  [EBP-0x18] iArray seventh element
[ESP+0x1C] 0012FF2C  00000007  [EBP-0x14] iArray eighth element
[ESP+0x20] 0012FF30  00000008  [EBP-0x10] iArray ninth element
[ESP+0x24] 0012FF34  00000009  [EBP-0x0C] iArray tenth element
```

[ESP+0x28] 0012FF38 ED59748F [EBP-0x08] Stack Cookie xor EBP value placed

[ESP+0x2C] 0012FF3C 0000000A [EBP-0x04] iLoop placeholder here

[ESP+0x30] 0012FF40 0012FF88 [EBP]

[ESP+0x34] 0012FF44 00401299 return to array.00401299 from array.00401000



```
00401000     55                push ebp
00401001     8BEC              mov ebp,esp
00401003     83EC 30           sub esp,30
00401006     A1 00804000       mov eax,dword ptr ds:[408000]
0040100B     33C5              xor eax,ebp
0040100D     8945 F8           mov dword ptr ss:[ebp-8],eax
00401010     C745 D0 00000000  mov dword ptr ss:[ebp-30],0
00401017     C745 D4 01000000  mov dword ptr ss:[ebp-2C],1
0040101E     C745 D8 02000000  mov dword ptr ss:[ebp-28],2
00401025     C745 DC 03000000  mov dword ptr ss:[ebp-24],3
0040102C     C745 E0 04000000  mov dword ptr ss:[ebp-20],4
00401033     C745 E4 05000000  mov dword ptr ss:[ebp-1C],5
0040103A     C745 E8 06000000  mov dword ptr ss:[ebp-18],6
00401041     C745 EC 07000000  mov dword ptr ss:[ebp-14],7
00401048     C745 F0 08000000  mov dword ptr ss:[ebp-10],8
0040104F     C745 F4 09000000  mov dword ptr ss:[ebp-C],9
00401056     C745 FC 00000000  mov dword ptr ss:[ebp-4],0
0040105D   v EB 09             jmp array.401068
0040105F     8B45 FC           mov eax,dword ptr ss:[ebp-4]
00401062     83C0 01           add eax,1
00401065     8945 FC           mov dword ptr ss:[ebp-4],eax
00401068     837D FC 0A        cmp dword ptr ss:[ebp-4],A
0040106C   v 7D 1B             jge array.401089
0040106E     8B4D FC           mov ecx,dword ptr ss:[ebp-4]
00401071     8B548D D0         mov edx,dword ptr ss:[ebp+ecx*4-30]
00401075     52                push edx
00401076     8B45 FC           mov eax,dword ptr ss:[ebp-4]
00401079     50                push eax
0040107A     68 40814000       push array.408140
0040107F     E8 15000000       call array.401099
00401084     83C4 0C           add esp,C
00401087   ^ EB D6             jmp array.40105F
00401089     33C0              xor eax,eax
0040108B     8B4D F8           mov ecx,dword ptr ss:[ebp-8]
0040108E     33CD              xor ecx,ebp
```

```
EAX    0000000A
EBX    7FFDF000
ECX    0040112C      array.0040112C
EDX    77D27084      <ntdll.KiFastSys
EBP    0012FF40
ESP    0012FF10
ESI    00000000
EDI    00000000

EIP    00401089      array.00401089

EFLAGS     00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Em
ST(1) 00000000000000000000 x87r1 Em
ST(2) 00000000000000000000 x87r2 Em
ST(3) 00000000000000000000 x87r3 Em
ST(4) 00000000000000000000 x87r4 Em
ST(5) 00000000000000000000 x87r5 Em
ST(6) 00000000000000000000 x87r6 Em
ST(7) 00000000000000000000 x87r7 Em
```

```
69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
78 69 74 50 72 6F 63 65 73 73 00 00 6D 00 73 00  xitProcess..m.s.
63 00 6F 00 72 00 65 00 65 00 2E 00 64 00 6C 00  c.o.r.e.e...d.l.
6C 00 00 00 72 00 75 00 6E 00 74 00 69 00 6D 00  l...r.u.n.t.i.m.
65 00 20 00 65 00 72 00 72 00 6F 00 72 00 20 00  e. .e.r.r.o.r. .
```

```
0012FF04  00408140  "iArray[%d]=%d\n"
0012FF08  00000009
0012FF0C  00000009
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  0000000A
0012FF40  0012FF88
0012FF44  00401299  return to array.0
```

Figure 11.12: The stack state after JGE

▼Line 65-77

```
$LN1@main:
; Line 14
xor eax, eax
; Line 15
mov ecx, DWORD PTR __$ArrayPad$[ebp]
xor ecx, ebp
call @__security_check_cookie@4
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS

END
```

At this label, EAX is zeroed using XOR to return 0 as per line 14 of the C/C++ code.

Security Cookie is checked before the function epilogue by first moving the Security Cookie stored at **[EBP-0x08]** to ECX and then XOR ECX with EBP. A call to the **security_check_cookie** procedure is made to prevent the buffer overflow attack.

Security Cookie stored at **[EBP-0x08]** = **0xED59748F**

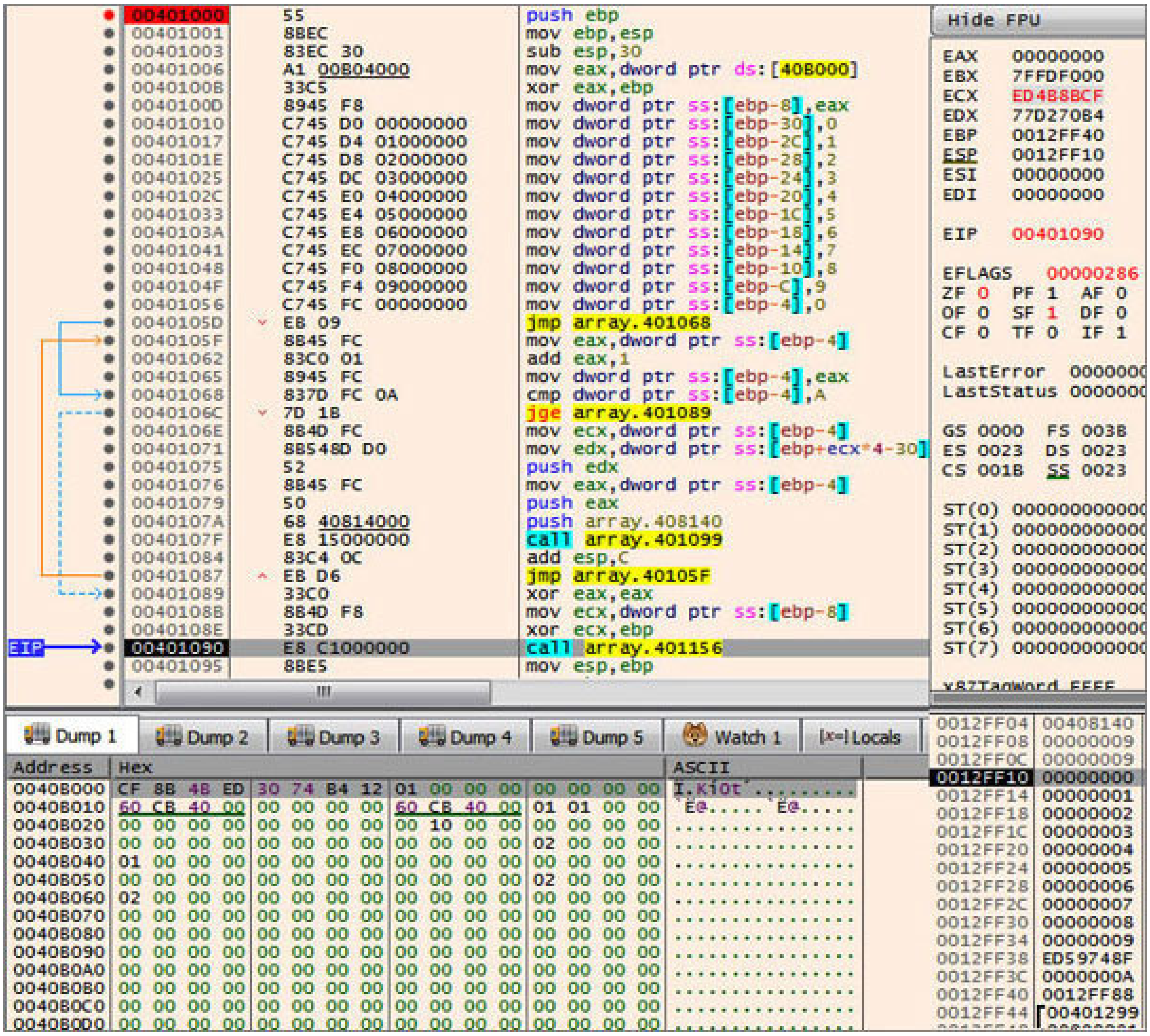
EBP = **0x0012FF40**

ECX XOR EBP = **0xED59748F** XOR **0x0012FF40** = **0xED4B8BCF**

The result will be saved in ECX.

As we can see, this value is the same as the value generated and stored at the **0x0040B000** memory location. As we are dealing with little-endian, the value is stored in the reversed order. The stack state at this point is the same as earlier:

```
[ESP-0x0C] 0012FF04   00408140   [EBP-0x40] NOW JUNK
[ESP-0x8] 0012FF08   00000009   [EBP-0x38] NOW JUNK
[ESP-0x4] 0012FF0C   00000009   [EBP-0x34] NOW JUNK
[ESP]   0012FF10   00000000   [EBP-0x30] iArray first element
[ESP+0x4] 0012FF14   00000001   [EBP-0x2C] iArray second element
[ESP+0x8] 0012FF18   00000002   [EBP-0x28] iArray third element
[ESP+0xC] 0012FF1C   00000003   [EBP-0x24] iArray forth element
[ESP+0x10] 0012FF20   00000004   [EBP-0x20] iArray fifth element
[ESP+0x14] 0012FF24   00000005   [EBP-0x1C] iArray sixth element

[ESP+0x18] 0012FF28   00000006   [EBP-0x18] iArray seventh element
[ESP+0x1C] 0012FF2C   00000007   [EBP-0x14]  iArray eighth element
[ESP+0x20] 0012FF30   00000008   [EBP-0x10] iArray ninth element
[ESP+0x24] 0012FF34   00000009   [EBP-0x0C] iArray tenth element
[ESP+0x28] 0012FF38   ED59748F   [EBP-0x08] Stack Cookie xor EBP value placed
[ESP+0x2C] 0012FF3C   0000000A   [EBP-0x04] iLoop placeholder here
[ESP+0x30] 0012FF40   0012FF88   [EBP]
[ESP+0x34] 0012FF44   00401299   return to array.00401299 from array.00401000
```



```
00401000    55                push ebp
00401001    8BEC              mov ebp,esp
00401003    83EC 30           sub esp,30
00401006    A1 00B04000       mov eax,dword ptr ds:[40B000]
0040100B    33C5              xor eax,ebp
0040100D    8945 F8           mov dword ptr ss:[ebp-8],eax
00401010    C745 D0 00000000  mov dword ptr ss:[ebp-30],0
00401017    C745 D4 01000000  mov dword ptr ss:[ebp-2C],1
0040101E    C745 D8 02000000  mov dword ptr ss:[ebp-28],2
00401025    C745 DC 03000000  mov dword ptr ss:[ebp-24],3
0040102C    C745 E0 04000000  mov dword ptr ss:[ebp-20],4
00401033    C745 E4 05000000  mov dword ptr ss:[ebp-1C],5
0040103A    C745 E8 06000000  mov dword ptr ss:[ebp-18],6
00401041    C745 EC 07000000  mov dword ptr ss:[ebp-14],7
00401048    C745 F0 08000000  mov dword ptr ss:[ebp-10],8
0040104F    C745 F4 09000000  mov dword ptr ss:[ebp-C],9
00401056    C745 FC 00000000  mov dword ptr ss:[ebp-4],0
0040105D    EB 09             jmp array.401068
0040105F    8B45 FC           mov eax,dword ptr ss:[ebp-4]
00401062    83C0 01           add eax,1
00401065    8945 FC           mov dword ptr ss:[ebp-4],eax
00401068    837D FC 0A        cmp dword ptr ss:[ebp-4],A
0040106C    7D 1B             jge array.401089
0040106E    8B4D FC           mov ecx,dword ptr ss:[ebp-4]
00401071    8B548D D0         mov edx,dword ptr ss:[ebp+ecx*4-30]
00401075    52                push edx
00401076    8B45 FC           mov eax,dword ptr ss:[ebp-4]
00401079    50                push eax
0040107A    68 40814000       push array.408140
0040107F    E8 15000000       call array.401099
00401084    83C4 0C           add esp,C
00401087    EB D6             jmp array.40105F
00401089    33C0              xor eax,eax
0040108B    8B4D F8           mov ecx,dword ptr ss:[ebp-8]
0040108E    33CD              xor ecx,ebp
00401090    E8 C1000000       call array.401156
00401095    8BE5              mov esp,ebp
```

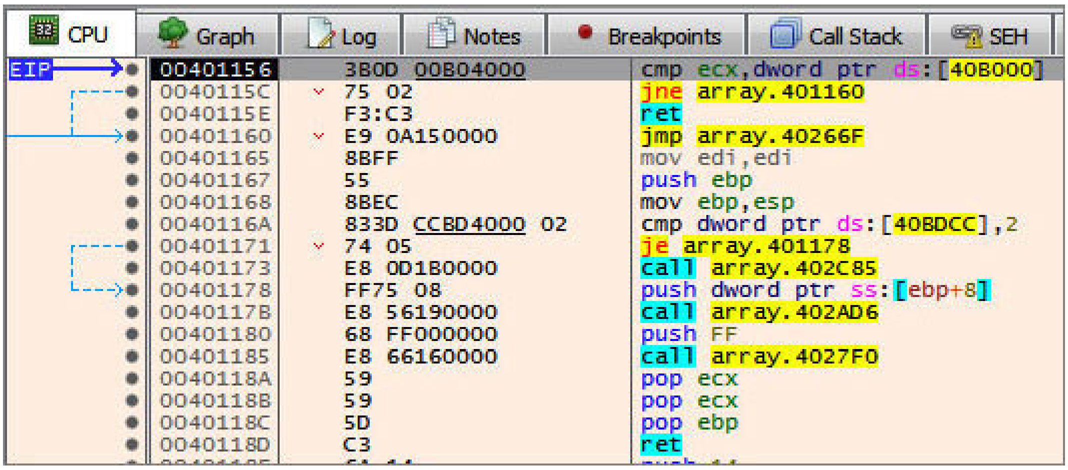
```
EAX    00000000
EBX    7FFDF000
ECX    ED4B8BCF
EDX    77D270B4
EBP    0012FF40
ESP    0012FF10
ESI    00000000
EDI    00000000

EIP    00401090

EFLAGS    00000286
ZF 0   PF 1   AF 0
OF 0   SF 1   DF 0
CF 0   TF 0   IF 1

LastError   0000000
LastStatus  0000000

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000
ST(1) 00000000000
ST(2) 00000000000
ST(3) 00000000000
ST(4) 00000000000
ST(5) 00000000000
ST(6) 00000000000
ST(7) 00000000000

x87TagWord FFFF
```

```
Address   Hex                                               ASCII
0040B000  CF 8B 4B ED 30 74 B4 12 01 00 00 00 00 00 00 00  Ï.KíOt´.........
0040B010  60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00  `Ë@.....`Ë@.....
0040B020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
0040B030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FF04  00408140
0012FF08  00000009
0012FF0C  00000009
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F
0012FF3C  0000000A
0012FF40  0012FF88
0012FF44  00401299
```

Figure 11.13: *Call to security_check_cookie*

If we step into the **call** we can see that our XOR result saved in ECX is compared with the security cookie stored at



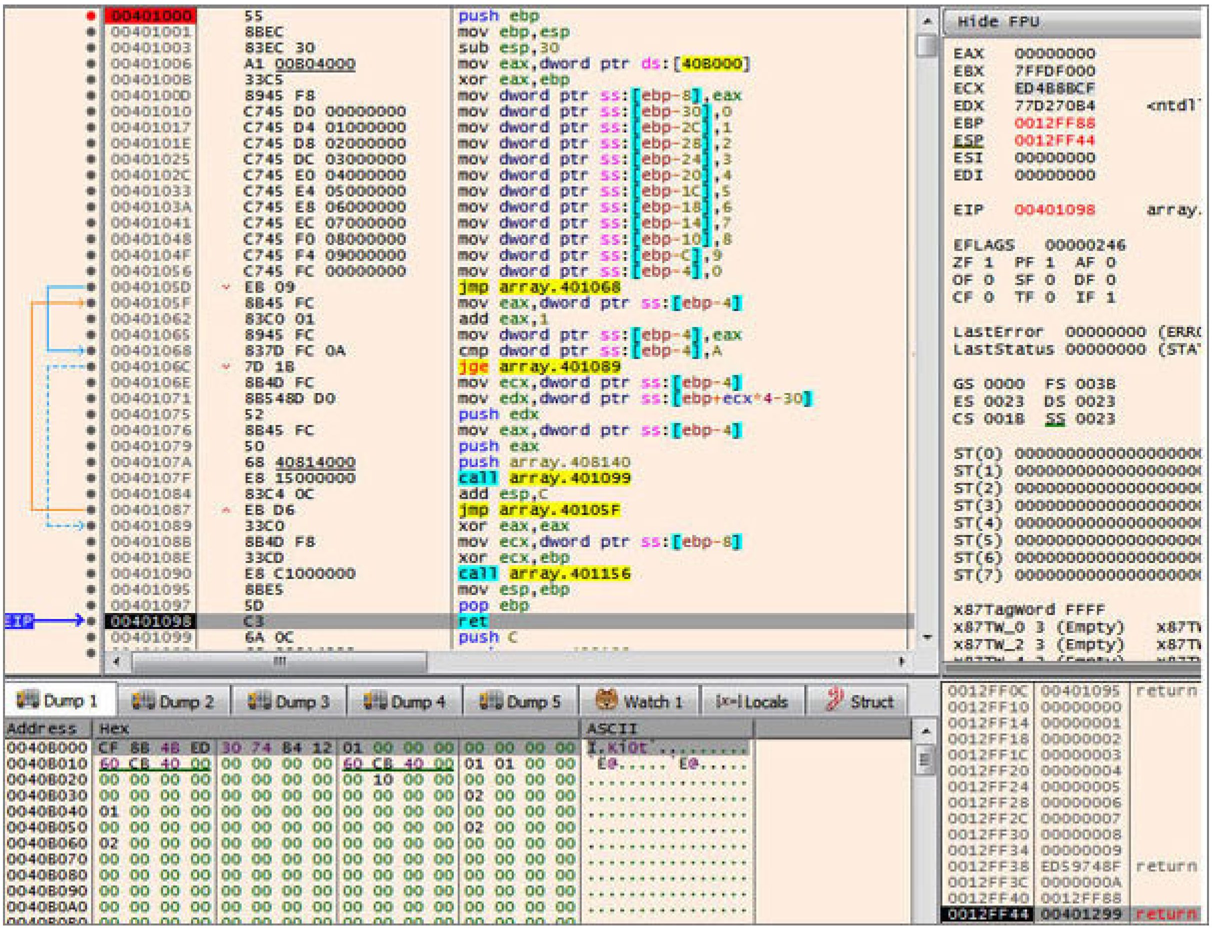
```
00401156    3B0D 00B04000     cmp ecx,dword ptr ds:[40B000]
0040115C  ˅ 75 02             jne array.401160
0040115E    F3:C3             ret
00401160  ˅ E9 0A150000       jmp array.40266F
00401165    8BFF              mov edi,edi
00401167    55                push ebp
00401168    8BEC              mov ebp,esp
0040116A    833D CCBD4000 02  cmp dword ptr ds:[40BDCC],2
00401171  ˅ 74 05             je array.401178
00401173    E8 0D1B0000       call array.402C85
00401178    FF75 08           push dword ptr ss:[ebp+8]
0040117B    E8 56190000       call array.402AD6
00401180    68 FF000000       push FF
00401185    E8 66160000       call array.4027F0
0040118A    59                pop ecx
0040118B    59                pop ecx
0040118C    5D                pop ebp
0040118D    C3                ret
```

Figure 11.14: Security_check_cookie function

As our security cookie stored at **0x0040B000** is the same as that in ECX, ZF=1, the instruction pointer will return back to the **main** function. On return, the function epilogue is called to end the **main** function, TEXT segment, and code. Below is the stack state explained before return instruction:

[ESP-0x34] 0012FF10 00000000 NOW JUNK
[ESP-0x30] 0012FF14 00000001 NOW JUNK
[ESP-0x2C] 0012FF18 00000002 NOW JUNK
[ESP-0x28] 0012FF1C 00000003 NOW JUNK
[ESP-0x24] 0012FF20 00000004 NOW JUNK
[ESP-0x20] 0012FF24 00000005 NOW JUNK
[ESP-0x1C] 0012FF28 00000006 NOW JUNK
[ESP-0x18] 0012FF2C 00000007 NOW JUNK
[ESP-0x14] 0012FF30 00000008 NOW JUNK
[ESP-0x10] 0012FF34 00000009 NOW JUNK
[ESP-0x0C] 0012FF38 ED59748F NOW JUNK
[ESP-0x08] 0012FF3C 0000000A NOW JUNK
[ESP-0x04] 0012FF40 0012FF88 NOW JUNK

```
[ESP] 0012FF44 00401299 return to array.00401299 from array.00401000
```



```
00401000  55                 push ebp
00401001  8BEC               mov ebp,esp
00401003  83EC 30            sub esp,30
00401006  A1 00B04000        mov eax,dword ptr ds:[408000]
0040100B  33C5               xor eax,ebp
0040100D  8945 F8            mov dword ptr ss:[ebp-8],eax
00401010  C745 D0 00000000   mov dword ptr ss:[ebp-30],0
00401017  C745 D4 01000000   mov dword ptr ss:[ebp-2C],1
0040101E  C745 D8 02000000   mov dword ptr ss:[ebp-28],2
00401025  C745 DC 03000000   mov dword ptr ss:[ebp-24],3
0040102C  C745 E0 04000000   mov dword ptr ss:[ebp-20],4
00401033  C745 E4 05000000   mov dword ptr ss:[ebp-1C],5
0040103A  C745 E8 06000000   mov dword ptr ss:[ebp-18],6
00401041  C745 EC 07000000   mov dword ptr ss:[ebp-14],7
00401048  C745 F0 08000000   mov dword ptr ss:[ebp-10],8
0040104F  C745 F4 09000000   mov dword ptr ss:[ebp-C],9
00401056  C745 FC 00000000   mov dword ptr ss:[ebp-4],0
0040105D  EB 09              jmp array.401068
0040105F  8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401062  83C0 01            add eax,1
00401065  8945 FC            mov dword ptr ss:[ebp-4],eax
00401068  837D FC 0A         cmp dword ptr ss:[ebp-4],A
0040106C  7D 1B              jge array.401089
0040106E  8B4D FC            mov ecx,dword ptr ss:[ebp-4]
00401071  8B548D D0          mov edx,dword ptr ss:[ebp+ecx*4-30]
00401075  52                 push edx
00401076  8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401079  50                 push eax
0040107A  68 40814000        push array.408140
0040107F  E8 15000000        call array.401099
00401084  83C4 0C            add esp,C
00401087  EB D6              jmp array.40105F
00401089  33C0               xor eax,eax
0040108B  8B4D F8            mov ecx,dword ptr ss:[ebp-8]
0040108E  33CD               xor ecx,ebp
00401090  E8 C1000000        call array.401156
00401095  8BE5               mov esp,ebp
00401097  5D                 pop ebp
00401098  C3                 ret
00401099  6A 0C              push C
```

```
EAX    00000000
EBX    7FFDF000
ECX    ED4B8BCF
EDX    77D270B4    <ntdl
EBP    0012FF88
ESP    0012FF44
ESI    00000000
EDI    00000000

EIP    00401098    array.

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
ST(7) 0000000000000000000

x87TagWord FFFF
x87TW_0 3 (Empty)    x87T
x87TW_2 3 (Empty)    x87T
```

```
Address   Hex                                               ASCII
00408000  CF 8B 4B ED 30 74 B4 12 01 00 00 00 00 00 00 00   I.K10t´.........
00408010  60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00    Ë@..... Ë@.....
00408020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00   ................
00408030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00   ................
00408040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00   ................
00408060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
004080A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
```

```
0012FF0C  00401095  return
0012FF10  00000000
0012FF14  00000001
0012FF18  00000002
0012FF1C  00000003
0012FF20  00000004
0012FF24  00000005
0012FF28  00000006
0012FF2C  00000007
0012FF30  00000008
0012FF34  00000009
0012FF38  ED59748F  return
0012FF3C  0000000A
0012FF40  0012FF88
0012FF44  00401299  return
```

Figure 11.15: Stack cleaned

Array Loop with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

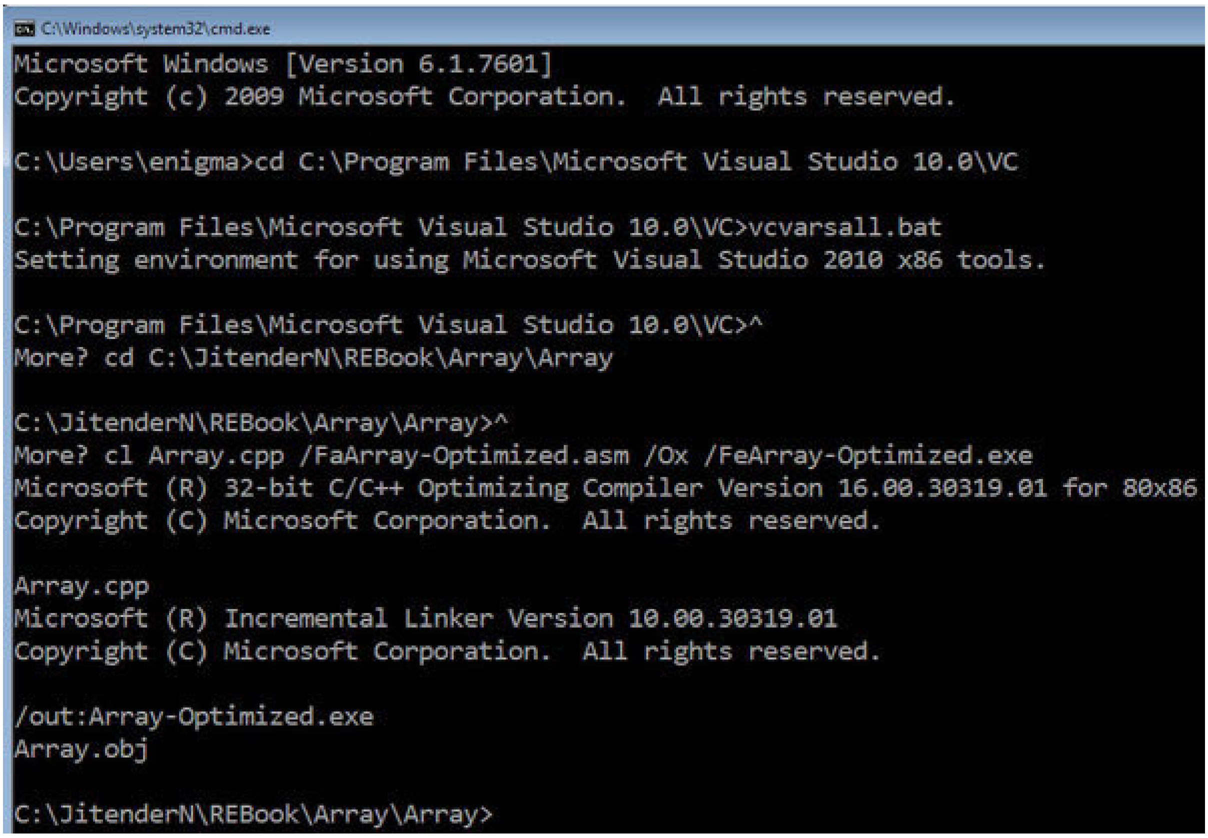
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Array\Array

C:\JitenderN\REBook\Array\Array>^
More? cl Array.cpp /FaArray-Optimized.asm /Ox /FeArray-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Array.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Array-Optimized.exe
Array.obj

C:\JitenderN\REBook\Array\Array>
```

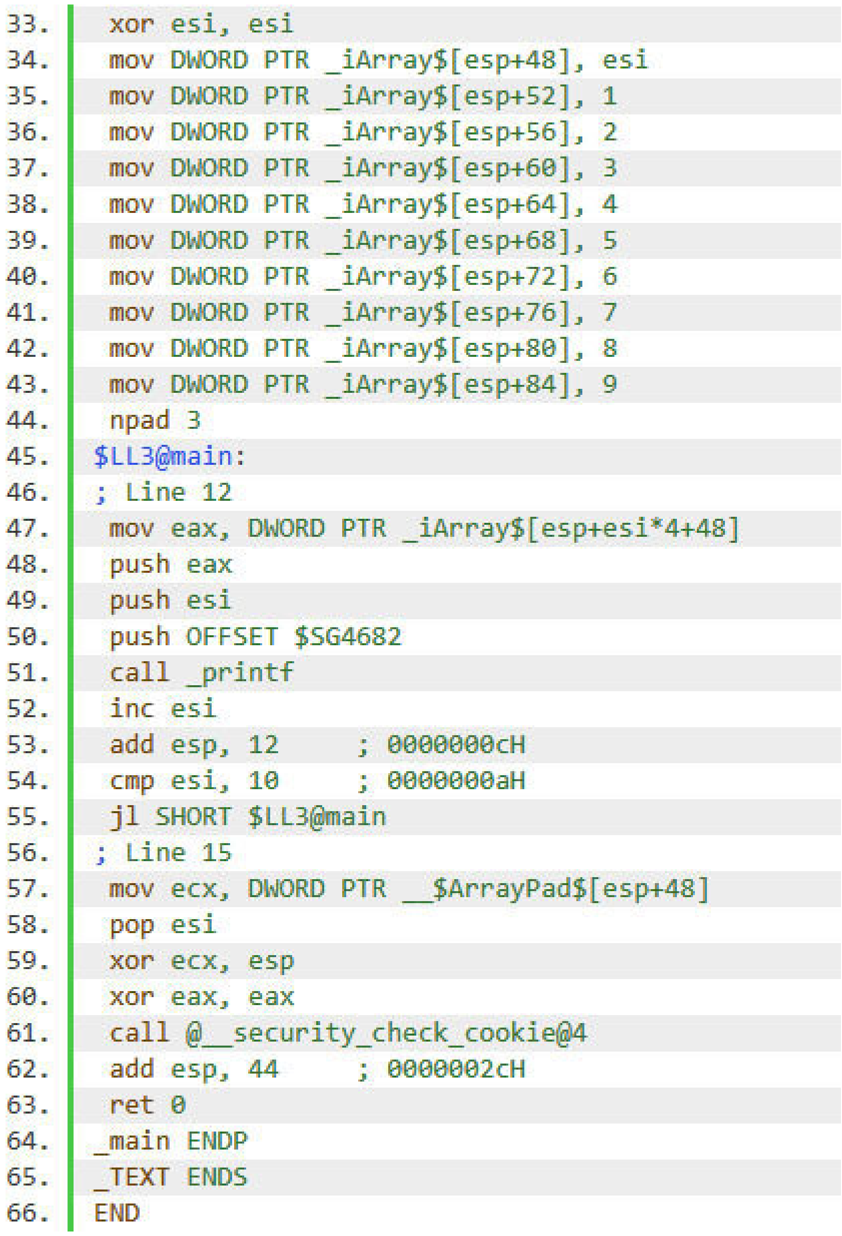
Figure 11.16: *Array Loop with Optimization*

The compilation generates the EXE file and assembly code. Disable ASLR manually. To disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can** For detailed steps to disable ASLR, please refer to the

Now, let us move on to generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing

 TITLE C:\JitenderN\REBook\Array\Array\Array.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4682 DB 'iArray[%d]=%d', 0aH, 00H
CONST ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_iArray$ = -44      ; size = 40
__$ArrayPad$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\array\array\array.cpp
; Line 7
 sub esp, 44      ; 0000002cH
 mov eax, DWORD PTR ___security_cookie
 xor eax, esp
 mov DWORD PTR __$ArrayPad$[esp+44], eax
 push esi
; Line 8
```

Figure 11.17: *Array-Optimized.asm-Part-1*



```
  xor esi, esi
  mov DWORD PTR _iArray$[esp+48], esi
  mov DWORD PTR _iArray$[esp+52], 1
  mov DWORD PTR _iArray$[esp+56], 2
  mov DWORD PTR _iArray$[esp+60], 3
  mov DWORD PTR _iArray$[esp+64], 4
  mov DWORD PTR _iArray$[esp+68], 5
  mov DWORD PTR _iArray$[esp+72], 6
  mov DWORD PTR _iArray$[esp+76], 7
  mov DWORD PTR _iArray$[esp+80], 8
  mov DWORD PTR _iArray$[esp+84], 9
  npad 3
$LL3@main:
; Line 12
  mov eax, DWORD PTR _iArray$[esp+esi*4+48]
  push eax
  push esi
  push OFFSET $SG4682
  call _printf
  inc esi
  add esp, 12      ; 0000000cH
  cmp esi, 10      ; 0000000aH
  jl SHORT $LL3@main
; Line 15
  mov ecx, DWORD PTR __$ArrayPad$[esp+48]
  pop esi
  xor ecx, esp
  xor eax, eax
  call @__security_check_cookie@4
  add esp, 44      ; 0000002cH
  ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 11.18: *Array-Optimized.asm-Part-2*

With optimization enabled, all of the unwanted code is removed in the ASM listing. As we walk through the ASM code, we see that the standard function prologue and the epilogue are not in the optimized code. So, the EBP reference is replaced with ESP in all the instructions.

We will directly jump to the instructions as we have covered some of the common ASM listings in the earlier section.

▼Line 26-31

```
; Line 7
sub esp, 44        ; 0000002cH
mov eax, DWORD PTR ___security_cookie
xor eax, esp
mov DWORD PTR __$ArrayPad$[esp+44], eax
push esi
```

The **SUB** instruction creates room for the local variables on the stack by adding **44 (0x2C)** bytes to ESP. 44 bytes are coming from the 10 elements of **iArray** wherein each element is of 4 bytes each and 4 bytes are used for storing the stack cookie on the stack to prevent buffer overflow as explained earlier.

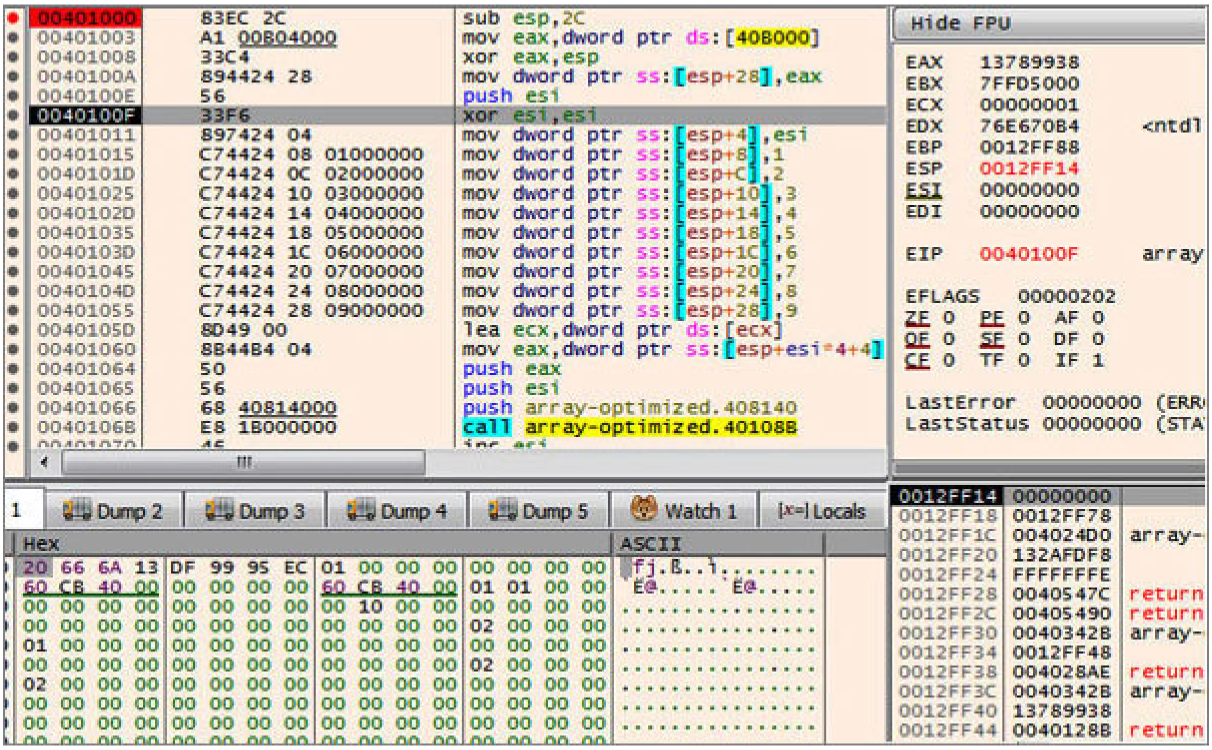
The instruction moves the stack cookie stored in the **.data** segment to the **EAX** register, where it is XOR'ed with The XOR'ed result is placed on the stack to prevent a buffer overflow exploit to happen. This XOR value will be checked before the **main** function epilogue.

ESI will be used for the placeholder of the **iLoop** variable; the old value of **ESI** is preserved on the stack using the **PUSH** instruction. The stack state after pushing the **ESI** register is as follows:

[ESP] 0012FF14 00000000 old ESI value is preserved here

[ESP+0x04] 0012FF18 0012FF78 JUNK
[ESP+0x08] 0012FF1C 004024D0 JUNK
[ESP+0x0C] 0012FF20 132AFDF8 JUNK
[ESP+0x010] 0012FF24 FFFFFFFE JUNK

[ESP+0x014] 0012FF28 0040547C JUNK
[ESP+0x18] 0012FF2C 00405490 JUNK
[ESP+0x1C] 0012FF30 0040342B JUNK
[ESP+0x20] 0012FF34 0012FF48 JUNK
[ESP+0x24] 0012FF38 004028AE JUNK
[ESP+0x28] 0012FF3C 0040342B JUNK
[ESP+0x2C] 0012FF40 13789938 XOR of stack cookie and ESP is stored here
[ESP+0x30] 0012FF44 0040128B ESP before 'sub esp, 44' instruction


```
00401000  83EC 2C             sub esp,2C
00401003  A1 00B04000         mov eax,dword ptr ds:[40B000]
00401008  33C4                xor eax,esp
0040100A  894424 28           mov dword ptr ss:[esp+28],eax
0040100E  56                  push esi
0040100F  33F6                xor esi,esi
00401011  897424 04           mov dword ptr ss:[esp+4],esi
00401015  C74424 08 01000000  mov dword ptr ss:[esp+8],1
0040101D  C74424 0C 02000000  mov dword ptr ss:[esp+C],2
00401025  C74424 10 03000000  mov dword ptr ss:[esp+10],3
0040102D  C74424 14 04000000  mov dword ptr ss:[esp+14],4
00401035  C74424 18 05000000  mov dword ptr ss:[esp+18],5
0040103D  C74424 1C 06000000  mov dword ptr ss:[esp+1C],6
00401045  C74424 20 07000000  mov dword ptr ss:[esp+20],7
0040104D  C74424 24 08000000  mov dword ptr ss:[esp+24],8
00401055  C74424 28 09000000  mov dword ptr ss:[esp+28],9
0040105D  8D49 00             lea ecx,dword ptr ds:[ecx]
00401060  8B44B4 04           mov eax,dword ptr ss:[esp+esi*4+4]
00401064  50                  push eax
00401065  56                  push esi
00401066  68 40814000         push array-optimized.408140
0040106B  E8 1B000000         call array-optimized.40108B
```

```
EAX   13789938
EBX   7FFD5000
ECX   00000001
EDX   76E670B4      <ntdl
EBP   0012FF88
ESP   0012FF14
ESI   00000000
EDI   00000000

EIP   0040100F      array

EFLAGS    00000202
ZF 0   PE 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA
```

```
Hex                                                                  ASCII
20 66 6A 13 DF 99 95 EC 01 00 00 00 00 00 00 00  fj.ß..ì.........
60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00  `Ë@.....`Ë@.....
00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FF14  00000000
0012FF18  0012FF78
0012FF1C  004024D0  array-
0012FF20  132AFDF8
0012FF24  FFFFFFFE
0012FF28  0040547C  return
0012FF2C  00405490  return
0012FF30  0040342B  array-
0012FF34  0012FF48
0012FF38  004028AE  return
0012FF3C  0040342B  array-
0012FF40  13789938
0012FF44  0040128B  return
```

Figure 11.19: Old ESI value is preserved

▼Line 32-44

```
; Line 8
xor esi, esi
mov DWORD PTR _iArray$[esp+48], esi
mov DWORD PTR _iArray$[esp+52], 1
mov DWORD PTR _iArray$[esp+56], 2

mov DWORD PTR _iArray$[esp+60], 3
mov DWORD PTR _iArray$[esp+64], 4
mov DWORD PTR _iArray$[esp+68], 5
mov DWORD PTR _iArray$[esp+72], 6
mov DWORD PTR _iArray$[esp+76], 7
mov DWORD PTR _iArray$[esp+80], 8
mov DWORD PTR _iArray$[esp+84], 9
npad 3
```

Line 8 of the C/C++ code initializes

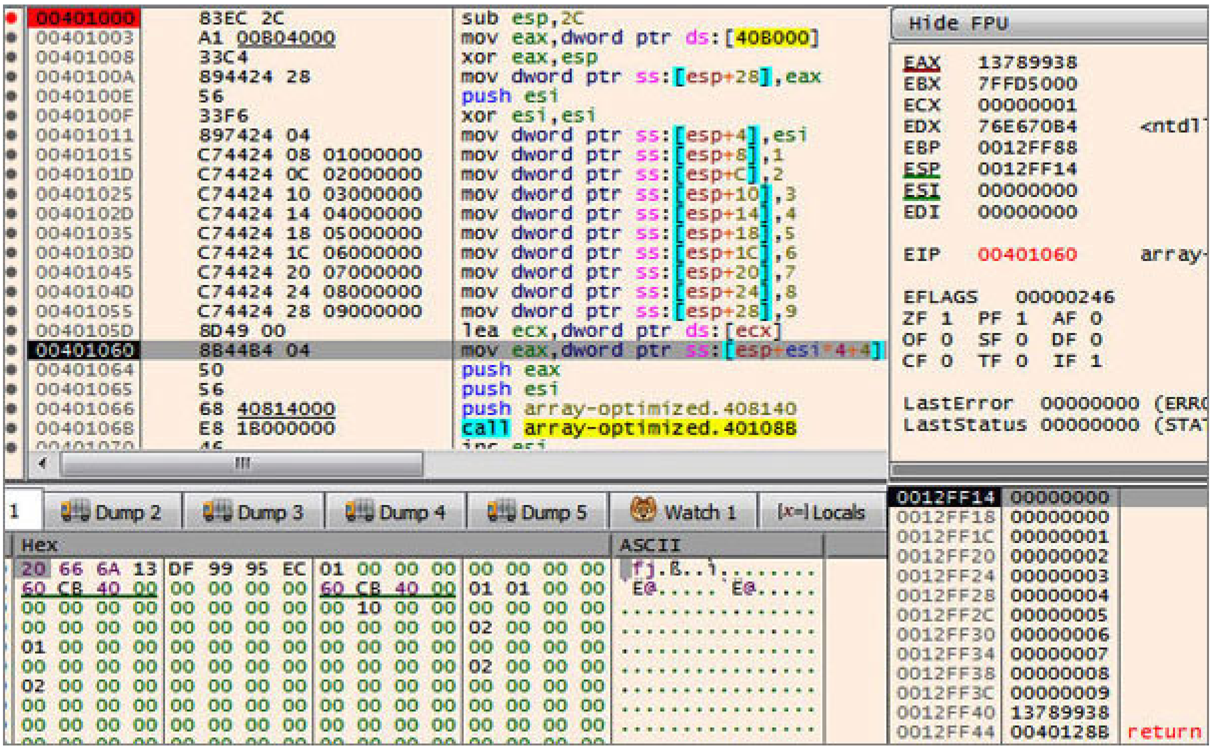
```
int iArray[10] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
```

In the ASM code, the XOR instruction resets the **ESI** register to **0x00** and then moves **ESI** to the **[ESP+0x04]** location as the first element of iArray. All the other **MOV** instructions push the remaining elements of **iArray** on the stack.

We see the **npad** macro, which inserts the non-destructive and non-operational instructions rather than a series of NOP instructions. **Npad 3** corresponds to **lea ecx,** For more details related to refer to the Appendix. The stack state immediately after the **npad** macro is as follows:

[ESP] 0012FF14 00000000 old ESI value is preserved here
[ESP+0x04] 0012FF18 00000000 iArray first element pushed here
[ESP+0x08] 0012FF1C 00000001 iArray second element pushed here
[ESP+0x0C] 0012FF20 00000002 iArray third element pushed here
[ESP+0x010] 0012FF24 00000003 iArray forth element pushed here

[ESP+0x014] 0012FF28 00000004 iArray fifth element pushed here
[ESP+0x18] 0012FF2C 00000005 iArray sixth element pushed here
[ESP+0x1C] 0012FF30 00000006 iArray seventh element pushed here
[ESP+0x20] 0012FF34 00000007 iArray eighth element pushed here
[ESP+0x24] 0012FF38 00000008 iArray ninth element pushed here
[ESP+0x28] 0012FF3C 00000009 iArray tenth element pushed here
[ESP+0x2C] 0012FF40 13789938 XOR of stack cookie and ESP is stored here
[ESP+0x30] 0012FF44 0040128B ESP before ‘sub esp, 44’ instruction



```
00401000   83EC 2C              sub esp,2C
00401003   A1 00B04000          mov eax,dword ptr ds:[40B000]
00401008   33C4                 xor eax,esp
0040100A   894424 28            mov dword ptr ss:[esp+28],eax
0040100E   56                   push esi
0040100F   33F6                 xor esi,esi
00401011   897424 04            mov dword ptr ss:[esp+4],esi
00401015   C74424 08 01000000   mov dword ptr ss:[esp+8],1
0040101D   C74424 0C 02000000   mov dword ptr ss:[esp+C],2
00401025   C74424 10 03000000   mov dword ptr ss:[esp+10],3
0040102D   C74424 14 04000000   mov dword ptr ss:[esp+14],4
00401035   C74424 18 05000000   mov dword ptr ss:[esp+18],5
0040103D   C74424 1C 06000000   mov dword ptr ss:[esp+1C],6
00401045   C74424 20 07000000   mov dword ptr ss:[esp+20],7
0040104D   C74424 24 08000000   mov dword ptr ss:[esp+24],8
00401055   C74424 28 09000000   mov dword ptr ss:[esp+28],9
0040105D   8D49 00              lea ecx,dword ptr ds:[ecx]
00401060   8B44B4 04            mov eax,dword ptr ss:[esp+esi*4+4]
00401064   50                   push eax
00401065   56                   push esi
00401066   68 40814000          push array-optimized.408140
0040106B   E8 18000000          call array-optimized.401088
```

```
EAX 13789938
EBX 7FFD5000
ECX 00000001
EDX 76E670B4     <ntdl
EBP 0012FF88
ESP 0012FF14
ESI 00000000
EDI 00000000

EIP 00401060     array-

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA
```

```
20 66 6A 13|DF 99 95 EC|01 00 00 00|00 00 00 00  fj.ß..ì.........
60 CB 40 00|00 00 00 00|60 CB 40 00|01 01 00 00   E@.....`E@.....
00 00 00 00|00 00 00 00|00 10 00 00|00 00 00 00  ................
00 00 00 00|00 00 00 00|00 00 00 00|02 00 00 00  ................
01 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00  ................
00 00 00 00|00 00 00 00|00 00 00 00|02 00 00 00  ................
02 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00  ................
00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00  ................
00 00 00 00|00 00 00 00|00 00 00 00|00 00 00 00  ................
```

```
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000001
0012FF20  00000002
0012FF24  00000003
0012FF28  00000004
0012FF2C  00000005
0012FF30  00000006
0012FF34  00000007
0012FF38  00000008
0012FF3C  00000009
0012FF40  13789938
0012FF44  0040128B  return
```

Figure 11.20: *The stack state after npad*

▼Line 45-55

```
$LL3@main:
; Line 12
mov eax, DWORD PTR _iArray$[esp+esi*4+48]
push eax
push esi
push OFFSET $SG4682
call _printf
inc esi
add esp, 12          ; 0000000cH
cmp esi, 10          ; 0000000aH
jl SHORT $LL3@main
```

Line 12 of the C/C++ code prints the elements of **iArray** along with the index on the screen.

```
printf ("iArray[%d]=%d\n", iLoop, iArray[iLoop]);
```

This same concept was discussed in the previous section without optimization. The **printf** function takes three arguments to the print **iArray** elements and indexes on the screen. So, all these three arguments need to be pushed onto the stack before the call to the **printf** function.

In the ASM code, the first argument to be pushed on the stack will be To push this value on the stack, the **iLoop** value stored in the **ESI** register is multiplied by 4 (the size of the integer) and then added to the **iArray** macro and offset of 48 bytes to calculate the memory location of the element stored for the particular **iLoop** value. So, during the first iteration when the **iLoop** stored at ESI = 0, the first **MOV** instruction will result in:

```
mov eax, DWORD PTR _iArray$[esp+esi*4+48]
mov eax, dword ptr ss:[esp+esi*4+0x4], as ESI = 0
```

```
mov eax, dword ptr ss:[esp+0x4]
```

This will give us the value of the first array element in the **EAX** register. So, by pushing **EAX** onto the stack, we pushed the first argument to **printf** onto the stack.

The next argument to **printf** is the iLoop value. It is pushed onto the stack by pushing the **ESI** register.

Last argument, which is string constant is pushed by **push OFFSET**

Now that the three arguments are on the stack, a call to **printf** is made. Stack at this point in time is as follows:

[ESP] 0012FF08 00408140 "iArray[%d]=%d\n", string argument to printf

[ESP+0x04] 0012FF0C 00000000 iLoop value, ESI is pushed as printf argument

[ESP+0x08] 0012FF10 00000000 iArray first element is pushed as printf argument

[ESP+0x0C] 0012FF14 00000000 old ESI value is preserved here

[ESP+0x10] 0012FF18 00000000 iArray first element pushed here

[ESP+0x14] 0012FF1C 00000001 iArray second element pushed here

[ESP+0x18] 0012FF20 00000002 iArray third element pushed here

[ESP+0x1C] 0012FF24 00000003 iArray forth element pushed here

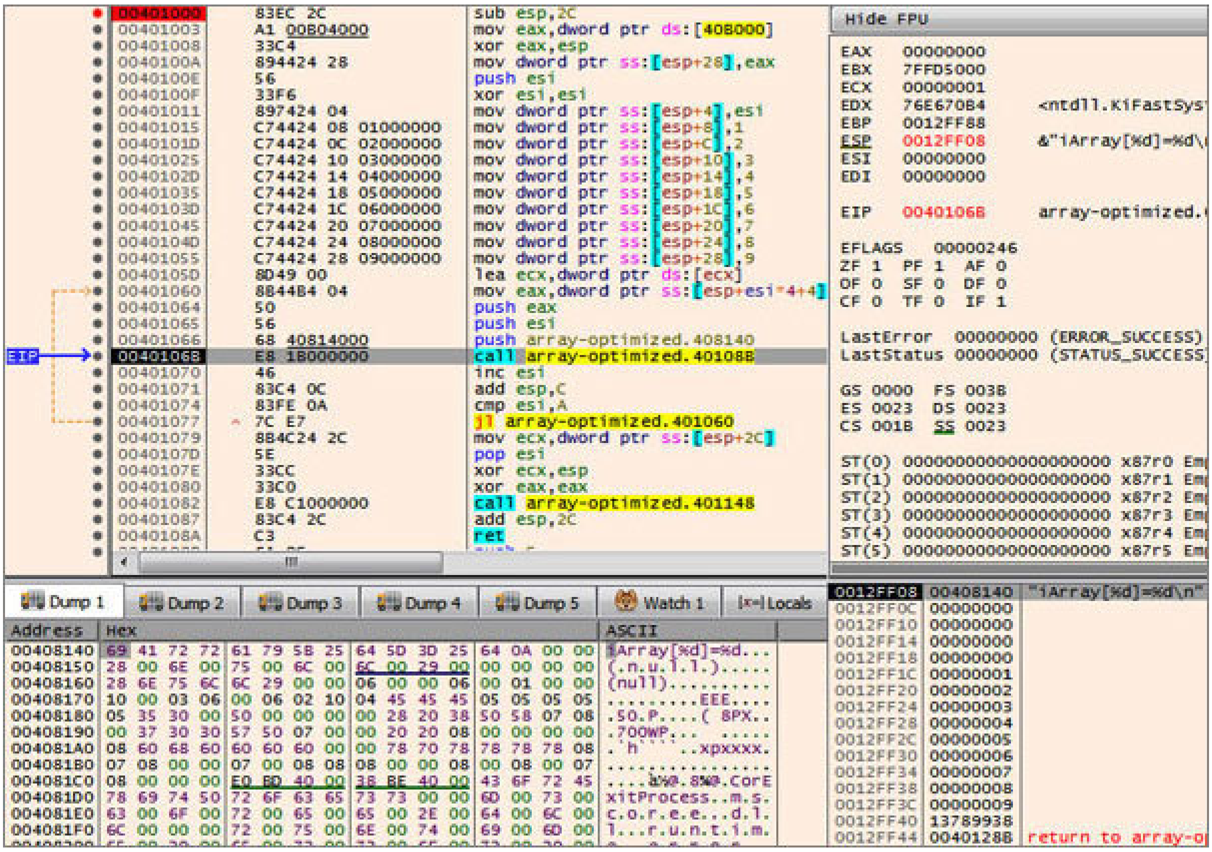
[ESP+0x20] 0012FF28 00000004 iArray fifth element pushed here

[ESP+0x24] 0012FF2C 00000005 iArray sixth element pushed here

[ESP+0x2C] 0012FF30 00000006 iArray seventh element pushed here

[ESP+0x30] 0012FF34 00000007 iArray eighth element pushed here

[ESP+0x34] 0012FF38 00000008 iArray ninth element pushed here



```
00401000   83EC 2C              sub esp,2C
00401003   A1 00B04000          mov eax,dword ptr ds:[408000]
00401008   33C4                 xor eax,esp
0040100A   894424 28            mov dword ptr ss:[esp+28],eax
0040100E   56                   push esi
0040100F   33F6                 xor esi,esi
00401011   897424 04            mov dword ptr ss:[esp+4],esi
00401015   C74424 08 01000000   mov dword ptr ss:[esp+8],1
0040101D   C74424 0C 02000000   mov dword ptr ss:[esp+C],2
00401025   C74424 10 03000000   mov dword ptr ss:[esp+10],3
0040102D   C74424 14 04000000   mov dword ptr ss:[esp+14],4
00401035   C74424 18 05000000   mov dword ptr ss:[esp+18],5
0040103D   C74424 1C 06000000   mov dword ptr ss:[esp+1C],6
00401045   C74424 20 07000000   mov dword ptr ss:[esp+20],7
0040104D   C74424 24 08000000   mov dword ptr ss:[esp+24],8
00401055   C74424 28 09000000   mov dword ptr ss:[esp+28],9
0040105D   8D49 00              lea ecx,dword ptr ds:[ecx]
00401060   8B44B4 04            mov eax,dword ptr ss:[esp+esi*4+4]
00401064   50                   push eax
00401065   56                   push esi
00401066   68 40814000          push array-optimized.408140
0040106B   E8 18000000          call array-optimized.401088
00401070   46                   inc esi
00401071   83C4 0C              add esp,C
00401074   83FE 0A              cmp esi,A
00401077   7C E7                jl array-optimized.401060
00401079   8B4C24 2C            mov ecx,dword ptr ss:[esp+2C]
0040107D   5E                   pop esi
0040107E   33CC                 xor ecx,esp
00401080   33C0                 xor eax,eax
00401082   E8 C1000000          call array-optimized.401148
00401087   83C4 2C              add esp,2C
0040108A   C3                   ret
```

```
EAX     00000000
EBX     7FFD5000
ECX     00000001
EDX     76E670B4      <ntdll.KiFastSys
EBP     0012FF88
ESP     0012FF08      &"iArray[%d]=%d\
ESI     00000000
EDI     00000000

EIP     0040106B      array-optimized.

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Em
ST(1) 00000000000000000000 x87r1 Em
ST(2) 00000000000000000000 x87r2 Em
ST(3) 00000000000000000000 x87r3 Em
ST(4) 00000000000000000000 x87r4 Em
ST(5) 00000000000000000000 x87r5 Em
```

```
Address   Hex                                               ASCII
00408140  69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
00408150  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
00408160  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
00408170  10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
00408180  05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00408190  00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
004081A0  08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
004081B0  07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
004081C0  08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
004081D0  78 69 74 50 72 6F 63 65 73 73 00 00 6D 00 73 00  xitProcess..m.s.
004081E0  63 00 6F 00 72 00 65 00 65 00 2E 00 64 00 6C 00  c.o.r.e.e...d.l.
004081F0  6C 00 00 00 72 00 75 00 6E 00 74 00 69 00 6D 00  l...r.u.n.t.i.m.
```

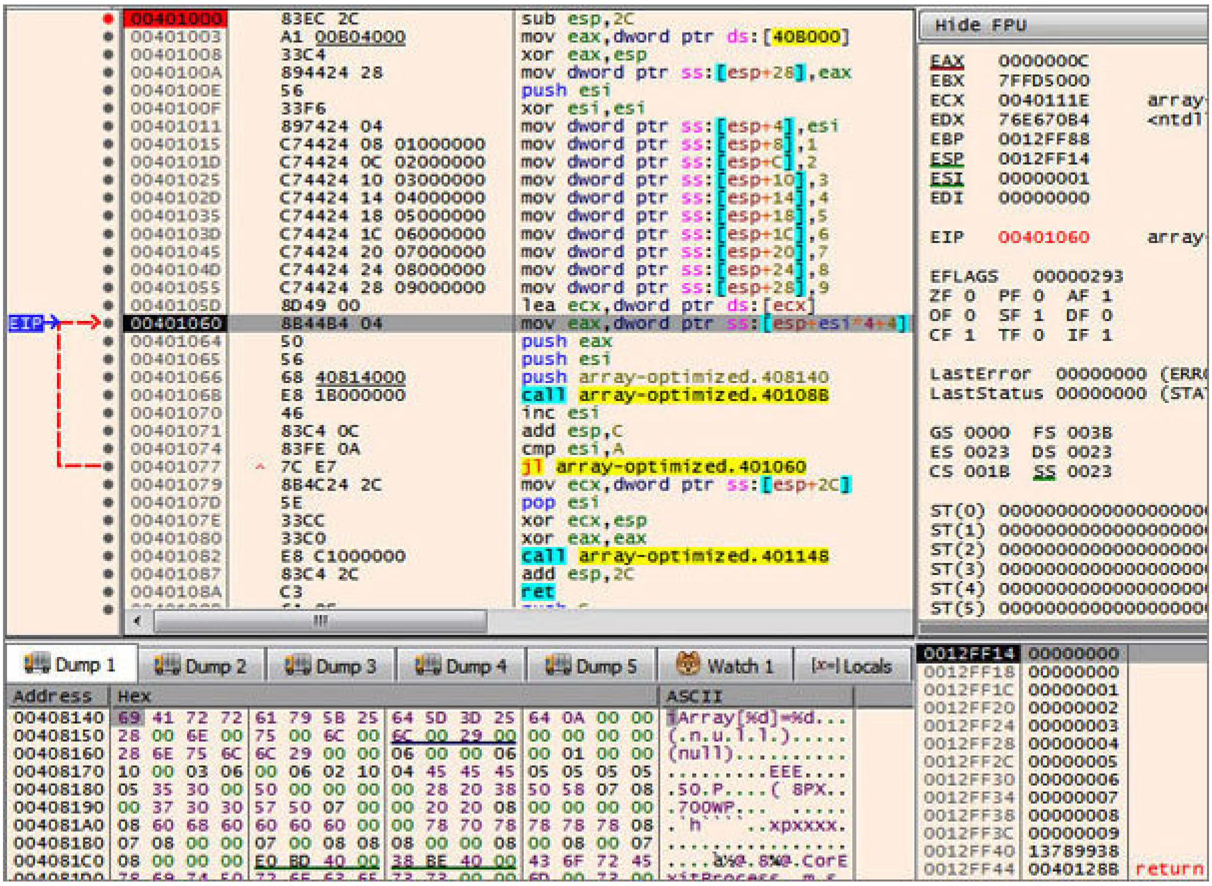
```
0012FF08  00408140  "iArray[%d]=%d\n"
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000001
0012FF20  00000002
0012FF24  00000003
0012FF28  00000004
0012FF2C  00000005
0012FF30  00000006
0012FF34  00000007
0012FF38  00000008
0012FF3C  00000009
0012FF40  13789938
0012FF44  0040128B  return to array-o
```

Figure 11.21: *Arg to printf on stack*

The **INC** instruction will increment the **ESI** register. This means that it increments the **iLoop** value stored in ESI.

The **ADD** instruction cleans the stack by 12 bytes. Next, **ESI** is compared with **10 (0x0A).** At this point, the **CMP** instruction will perform a signed comparison jump to the label **$LL3@main** as the **iLoop** at **ESI** less than **10(0x0A).** The stack state after **JUMP if Less than** is as follows:

```
[ESP-0x0C] 0012FF08   00408140    Now JUNK
[ESP-0x08] 0012FF0C   00000000    Now JUNK

[ESP-0x04] 0012FF10   00000000    Now JUNK
[ESP]   0012FF14   00000000   old ESI value is preserved here
[ESP+0x04] 0012FF18   00000000    iArray first element pushed
here
[ESP+0x08] 0012FF1C   00000001    iArray second element pushed
here
[ESP+0x0C] 0012FF20   00000002    iArray third element pushed
here
[ESP+0x010] 0012FF24   00000003    iArray forth element pushed
here
[ESP+0x014] 0012FF28   00000004    iArray fifth element pushed
here
[ESP+0x18] 0012FF2C   00000005    iArray sixth element pushed
here
[ESP+0x1C] 0012FF30   00000006    iArray seventh element pushed
here
[ESP+0x20] 0012FF34   00000007    iArray eighth element pushed
here
[ESP+0x24] 0012FF38   00000008    iArray ninth element pushed
here
[ESP+0x28] 0012FF3C   00000009    iArray tenth element pushed
here
[ESP+0x2C] 0012FF40   13789938    XOR of stack cookie and ESP
is stored here
[ESP+0x30] 0012FF44   0040128B    ESP before ‘sub esp, 44’
instruction
```



```
00401000    83EC 2C                 sub esp,2C
00401003    A1 00B04000             mov eax,dword ptr ds:[408000]
00401008    33C4                    xor eax,esp
0040100A    894424 28               mov dword ptr ss:[esp+28],eax
0040100E    56                      push esi
0040100F    33F6                    xor esi,esi
00401011    897424 04               mov dword ptr ss:[esp+4],esi
00401015    C74424 08 01000000      mov dword ptr ss:[esp+8],1
0040101D    C74424 0C 02000000      mov dword ptr ss:[esp+C],2
00401025    C74424 10 03000000      mov dword ptr ss:[esp+10],3
0040102D    C74424 14 04000000      mov dword ptr ss:[esp+14],4
00401035    C74424 18 05000000      mov dword ptr ss:[esp+18],5
0040103D    C74424 1C 06000000      mov dword ptr ss:[esp+1C],6
00401045    C74424 20 07000000      mov dword ptr ss:[esp+20],7
0040104D    C74424 24 08000000      mov dword ptr ss:[esp+24],8
00401055    C74424 28 09000000      mov dword ptr ss:[esp+28],9
0040105D    8D49 00                 lea ecx,dword ptr ds:[ecx]
00401060    8B44B4 04               mov eax,dword ptr ss:[esp+esi*4+4]
00401064    50                      push eax
00401065    56                      push esi
00401066    68 40814000             push array-optimized.408140
0040106B    E8 1B000000             call array-optimized.40108B
00401070    46                      inc esi
00401071    83C4 0C                 add esp,C
00401074    83FE 0A                 cmp esi,A
00401077  ^ 7C E7                   jl array-optimized.401060
00401079    8B4C24 2C               mov ecx,dword ptr ss:[esp+2C]
0040107D    5E                      pop esi
0040107E    33CC                    xor ecx,esp
00401080    33C0                    xor eax,eax
00401082    E8 C1000000             call array-optimized.401148
00401087    83C4 2C                 add esp,2C
0040108A    C3                      ret
```

```
EAX    0000000C
EBX    7FFD5000
ECX    0040111E      array
EDX    76E67084      <ntdl
EBP    0012FF88
ESP    0012FF14
ESI    00000001
EDI    00000000

EIP    00401060      array

EFLAGS    00000293
ZF 0   PF 0   AF 1
OF 0   SF 1   DF 0
CF 1   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
```

```
Address   Hex                                               ASCII
00408140  69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00  iArray[%d]=%d...
00408150  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
00408160  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
00408170  10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
00408180  05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00408190  00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
004081A0  08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
004081B0  07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
004081C0  08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
```

```
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000001
0012FF20  00000002
0012FF24  00000003
0012FF28  00000004
0012FF2C  00000005
0012FF30  00000006
0012FF34  00000007
0012FF38  00000008
0012FF3C  00000009
0012FF40  13789938
0012FF44  00401288  return
```

Figure 11.22: *The stack state after JL*

Now, imagine that we iterated 10 times over the instructions and the **iLoop** value at **ESI** has become **10 (0x0A).** At this point, the CMP instruction will result in ZF=1 and the jump will not happen to the **$LL3@main** label. The instruction pointer will move to the next instruction. The stack state at this stage will be:

[ESP-0x0C] 0012FF08 00408140 Now JUNK
[ESP-0x08] 0012FF0C 00000009 Now JUNK
[ESP-0x04] 0012FF10 00000009 Now JUNK
[ESP] 0012FF14 00000000 old ESI value is preserved here
[ESP+0x04] 0012FF18 00000000 iArray first element pushed here

[ESP+0x08] 0012FF1C 00000001 iArray second element pushed here

[ESP+0x0C] 0012FF20 00000002 iArray third element pushed here

[ESP+0x010] 0012FF24 00000003 iArray forth element pushed here

[ESP+0x014] 0012FF28 00000004 iArray fifth element pushed here

[ESP+0x18] 0012FF2C 00000005 iArray sixth element pushed here

[ESP+0x1C] 0012FF30 00000006 iArray seventh element pushed here

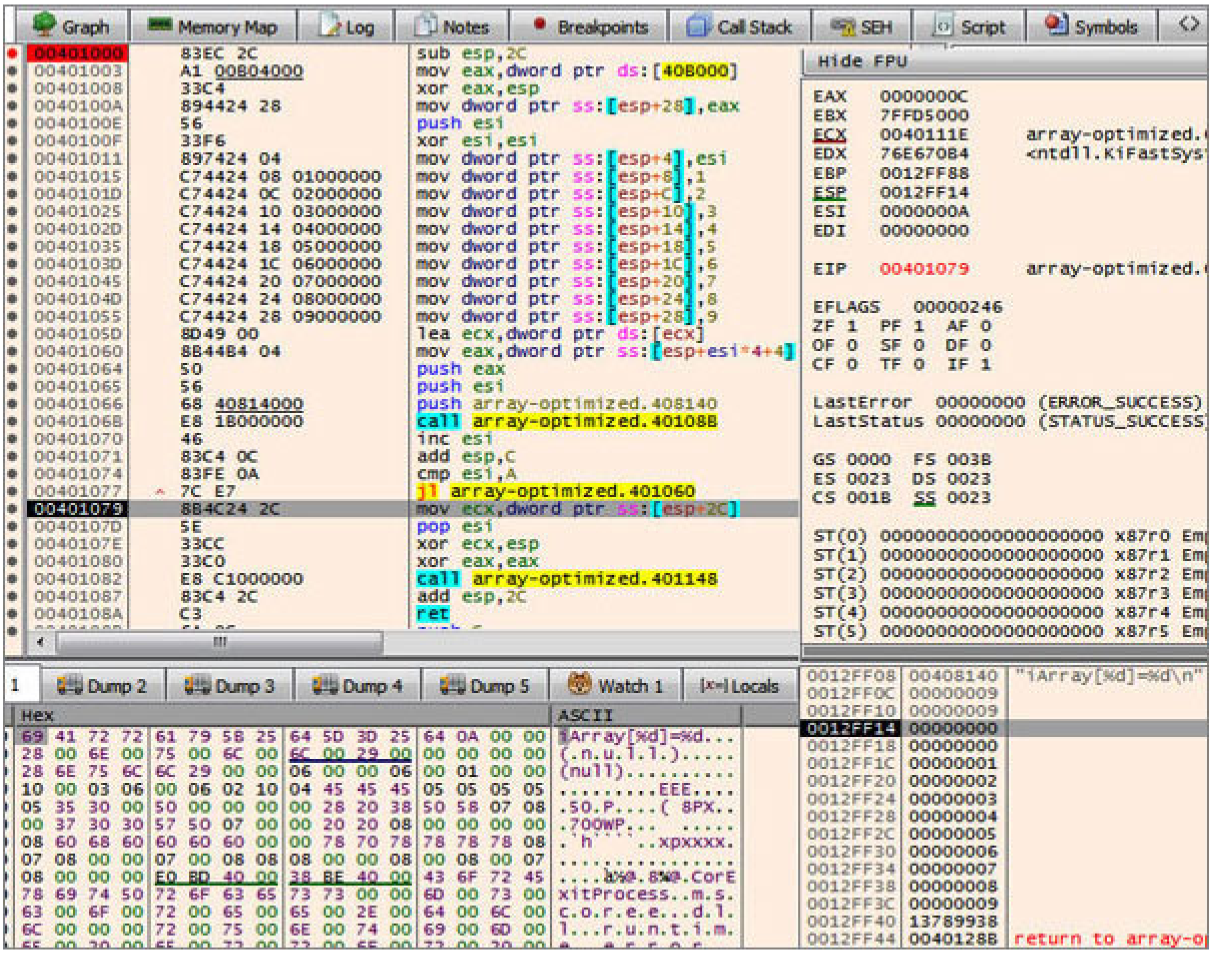
[ESP+0x20] 0012FF34 00000007 iArray eighth element pushed here

[ESP+0x24] 0012FF38 00000008 iArray ninth element pushed here

[ESP+0x28] 0012FF3C 00000009 iArray tenth element pushed here

[ESP+0x2C] 0012FF40 13789938 XOR of stack cookie and ESP is stored here

[ESP+0x30] 0012FF44 0040128B ESP before ‘sub esp, 44’ instruction



```
00401000   83EC 2C               sub esp,2C
00401003   A1 00804000           mov eax,dword ptr ds:[408000]
00401008   33C4                  xor eax,esp
0040100A   894424 28             mov dword ptr ss:[esp+28],eax
0040100E   56                    push esi
0040100F   33F6                  xor esi,esi
00401011   897424 04             mov dword ptr ss:[esp+4],esi
00401015   C74424 08 01000000    mov dword ptr ss:[esp+8],1
0040101D   C74424 0C 02000000    mov dword ptr ss:[esp+C],2
00401025   C74424 10 03000000    mov dword ptr ss:[esp+10],3
0040102D   C74424 14 04000000    mov dword ptr ss:[esp+14],4
00401035   C74424 18 05000000    mov dword ptr ss:[esp+18],5
0040103D   C74424 1C 06000000    mov dword ptr ss:[esp+1C],6
00401045   C74424 20 07000000    mov dword ptr ss:[esp+20],7
0040104D   C74424 24 08000000    mov dword ptr ss:[esp+24],8
00401055   C74424 28 09000000    mov dword ptr ss:[esp+28],9
0040105D   8D49 00               lea ecx,dword ptr ds:[ecx]
00401060   8B44B4 04             mov eax,dword ptr ss:[esp+esi*4+4]
00401064   50                    push eax
00401065   56                    push esi
00401066   68 40814000           push array-optimized.408140
0040106B   E8 1B000000           call array-optimized.40108B
00401070   46                    inc esi
00401071   83C4 0C               add esp,C
00401074   83FE 0A               cmp esi,A
00401077 ^ 7C E7                 jl array-optimized.401060
00401079   8B4C24 2C             mov ecx,dword ptr ss:[esp+2C]
0040107D   5E                    pop esi
0040107E   33CC                  xor ecx,esp
00401080   33C0                  xor eax,eax
00401082   E8 C1000000           call array-optimized.401148
00401087   83C4 2C               add esp,2C
0040108A   C3                    ret
```

```
EAX   0000000C
EBX   7FFD5000
ECX   0040111E     array-optimized.
EDX   76E67084     <ntdll.KiFastSys
EBP   0012FF88
ESP   0012FF14
ESI   0000000A
EDI   00000000

EIP   00401079     array-optimized.

EFLAGS   00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Em
ST(1) 00000000000000000000 x87r1 Em
ST(2) 00000000000000000000 x87r2 Em
ST(3) 00000000000000000000 x87r3 Em
ST(4) 00000000000000000000 x87r4 Em
ST(5) 00000000000000000000 x87r5 Em
```

```
Hex                                                  ASCII
69 41 72 72 61 79 5B 25 64 5D 3D 25 64 0A 00 00     iArray[%d]=%d...
28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00     (.n.u.l.l.).....
28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00     (null)..........
10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05     .........EEE....
05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08     .50.P....( 8PX..
00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00     .700WP...  .....
08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08     .`h````..xpxxxx.
07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07     ................
08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45     ....à½@.8¾@.CorE
78 69 74 50 72 6F 63 65 73 73 00 00 6D 00 73 00     xitProcess..m.s.
63 00 6F 00 72 00 65 00 65 00 2E 00 64 00 6C 00     c.o.r.e.e...d.l.
6C 00 00 00 72 00 75 00 6E 00 74 00 69 00 6D 00     l...r.u.n.t.i.m.
```

```
0012FF08  00408140  "iArray[%d]=%d\n"
0012FF0C  00000009
0012FF10  00000009
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000001
0012FF20  00000002
0012FF24  00000003
0012FF28  00000004
0012FF2C  00000005
0012FF30  00000006
0012FF34  00000007
0012FF38  00000008
0012FF3C  00000009
0012FF40  13789938
0012FF44  0040128B  return to array-o
```

Figure 11.23: *The stack state after ESI is 0x0A*

▼Line 56-66

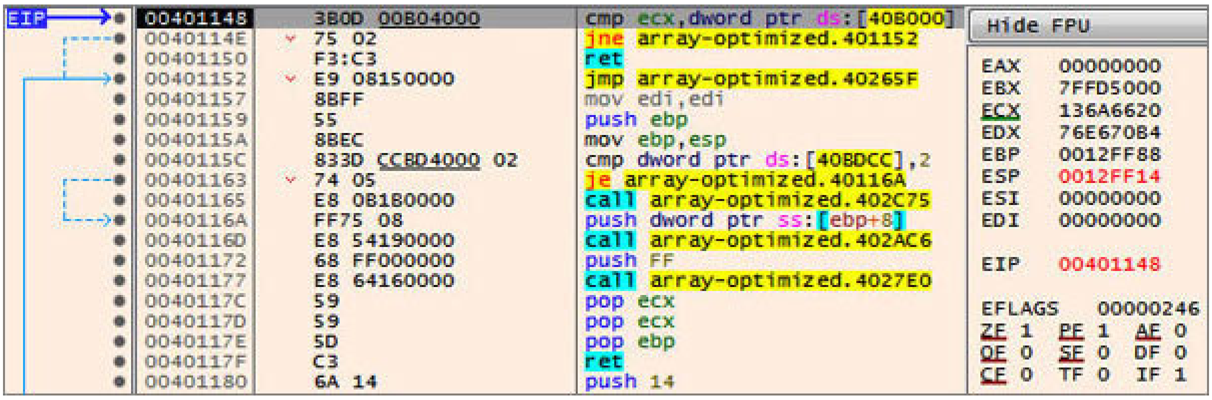
```
; Line 15
mov ecx, DWORD PTR __$ArrayPad$[esp+48]
pop esi
xor ecx, esp
xor eax, eax
call @__security_check_cookie@4
add esp, 44        ; 0000002cH
ret 0
_main ENDP
_TEXT ENDS
```

END

The **MOV** instruction will move the stack cookie stored at **[ESP+0x2C]** to

The **POP** instruction restores the value of **ESI** from the stack.

The XOR instruction will XOR the stack cookie moved to **ECX** with and the result of XOR will be stored back in the **ECX** register. On the call to the **security_check_cookie** procedure, this **ECX** value is compared with the stack cookie stored in the **.data** section.



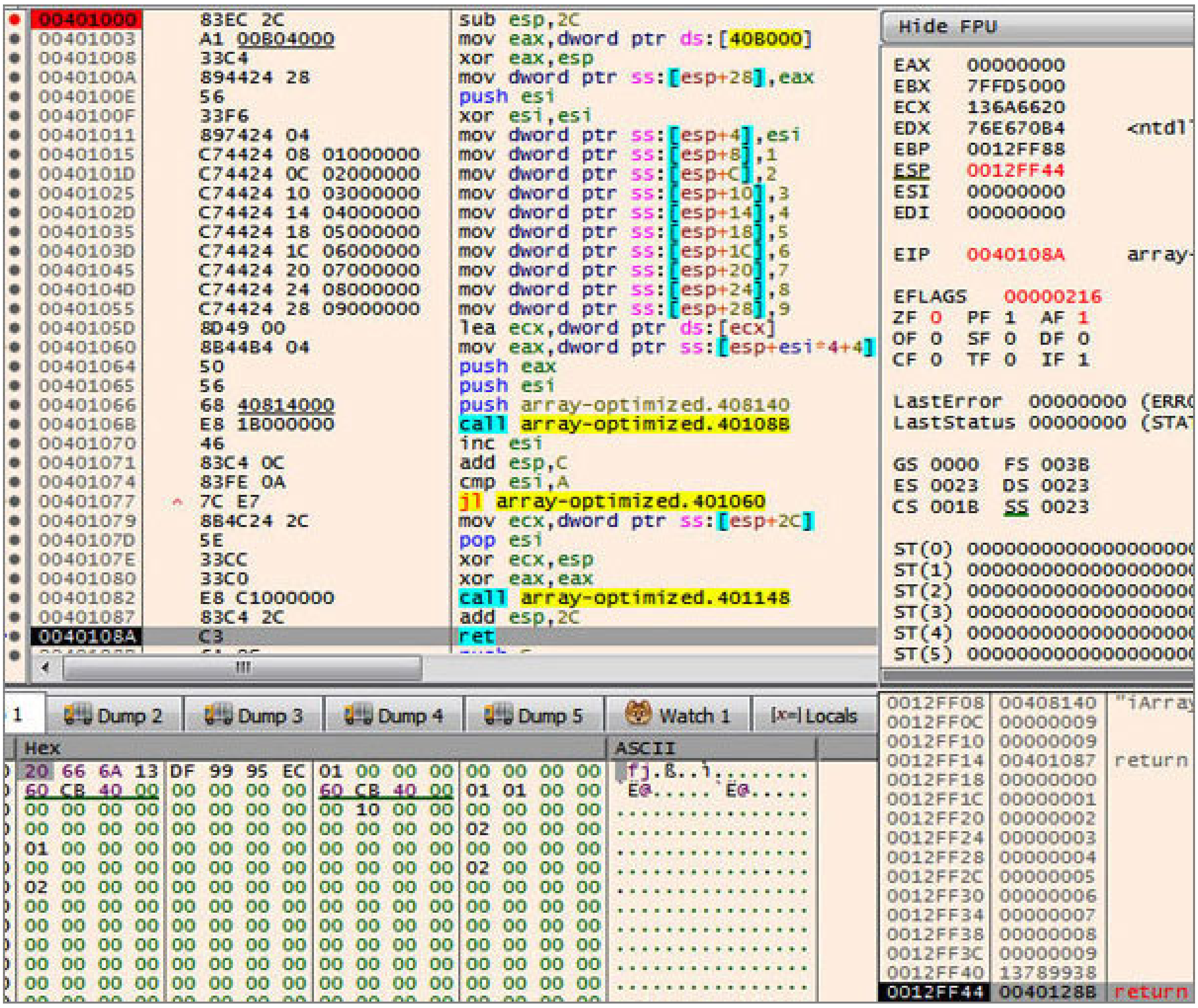
```
00401148    3B0D 00B04000       cmp ecx,dword ptr ds:[40B000]
0040114E    75 02               jne array-optimized.401152
00401150    F3:C3               ret
00401152    E9 08150000         jmp array-optimized.40265F
00401157    8BFF                mov edi,edi
00401159    55                  push ebp
0040115A    8BEC                mov ebp,esp
0040115C    833D CCBD4000 02    cmp dword ptr ds:[40BDCC],2
00401163    74 05               je array-optimized.40116A
00401165    E8 0B1B0000         call array-optimized.402C75
0040116A    FF75 08             push dword ptr ss:[ebp+8]
0040116D    E8 54190000         call array-optimized.402AC6
00401172    68 FF000000         push FF
00401177    E8 64160000         call array-optimized.4027E0
0040117C    59                  pop ecx
0040117D    59                  pop ecx
0040117E    5D                  pop ebp
0040117F    C3                  ret
00401180    6A 14               push 14
```

```
EAX     00000000
EBX     7FFD5000
ECX     136A6620
EDX     76E67084
EBP     0012FF88
ESP     0012FF14
ESI     00000000
EDI     00000000

EIP     00401148

EFLAGS     00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1
```

__Figure 11.24:__ Call to security_check_cookie

As the stack cookie value is unchanged, the instruction pointer will return back to On return, **EAX** is XOR with **EAX** to return 0 and the **ADD** instruction will clean up the stack to end the **main** procedure, TEXT segment, and code. The stack state in the end will be as follows:

```
[ESP-0x3C] 0012FF08   00408140    Now JUNK
[ESP-0x38] 0012FF0C   00000009    Now JUNK
[ESP-0x34] 0012FF10   00000009    Now JUNK
ESP-0x30] 0012FF14   00401087    Now JUNK
[ESP-0x2C] 0012FF18   00000000    Now JUNK
[ESP-0x28] 0012FF1C   00000001    Now JUNK
[ESP-0x24] 0012FF20   00000002    Now JUNK
[ESP-0x20] 0012FF24   00000003    Now JUNK
[ESP-0x1C] 0012FF28   00000004    Now JUNK
[ESP-0x18] 0012FF2C   00000005    Now JUNK
[ESP-0x14] 0012FF30   00000006    Now JUNK

[ESP-0x10] 0012FF34   00000007    Now JUNK
[ESP-0x0C] 0012FF38   00000008    Now JUNK
[ESP-0x08] 0012FF3C   00000009    Now JUNK
[ESP-0x04] 0012FF40   13789938    Now JUNK
[ESP]   0012FF44   0040128B    ESP at the start of main function
```



```
00401000   83EC 2C            sub esp,2C
00401003   A1 00B04000        mov eax,dword ptr ds:[40B000]
00401008   33C4               xor eax,esp
0040100A   894424 28          mov dword ptr ss:[esp+28],eax
0040100E   56                 push esi
0040100F   33F6               xor esi,esi
00401011   897424 04          mov dword ptr ss:[esp+4],esi
00401015   C74424 08 01000000 mov dword ptr ss:[esp+8],1
0040101D   C74424 0C 02000000 mov dword ptr ss:[esp+C],2
00401025   C74424 10 03000000 mov dword ptr ss:[esp+10],3
0040102D   C74424 14 04000000 mov dword ptr ss:[esp+14],4
00401035   C74424 18 05000000 mov dword ptr ss:[esp+18],5
0040103D   C74424 1C 06000000 mov dword ptr ss:[esp+1C],6
00401045   C74424 20 07000000 mov dword ptr ss:[esp+20],7
0040104D   C74424 24 08000000 mov dword ptr ss:[esp+24],8
00401055   C74424 28 09000000 mov dword ptr ss:[esp+28],9
0040105D   8D49 00            lea ecx,dword ptr ds:[ecx]
00401060   8B44B4 04          mov eax,dword ptr ss:[esp+esi*4+4]
00401064   50                 push eax
00401065   56                 push esi
00401066   68 40814000        push array-optimized.408140
0040106B   E8 1B000000        call array-optimized.40108B
00401070   46                 inc esi
00401071   83C4 0C            add esp,C
00401074   83FE 0A            cmp esi,A
00401077 ^ 7C E7              jl array-optimized.401060
00401079   8B4C24 2C          mov ecx,dword ptr ss:[esp+2C]
0040107D   5E                 pop esi
0040107E   33CC               xor ecx,esp
00401080   33C0               xor eax,eax
00401082   E8 C1000000        call array-optimized.401148
00401087   83C4 2C            add esp,2C
0040108A   C3                 ret
```

```
EAX    00000000
EBX    7FFD5000
ECX    136A6620
EDX    76E670B4      <ntdl
EBP    0012FF88
ESP    0012FF44
ESI    00000000
EDI    00000000

EIP    0040108A      array-

EFLAGS    00000216
ZF 0   PF 1   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STAT

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
```

```
Hex                                                ASCII
20 66 6A 13 DF 99 95 EC 01 00 00 00 00 00 00 00     fj.ß..ì........
60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00    `Ë@.....`Ë@.....
00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00    ................
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00    ................
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00    ................
```

```
0012FF08  00408140  "iArray
0012FF0C  00000009
0012FF10  00000009
0012FF14  00401087  return
0012FF18  00000000
0012FF1C  00000001
0012FF20  00000002
0012FF24  00000003
0012FF28  00000004
0012FF2C  00000005
0012FF30  00000006
0012FF34  00000007
0012FF38  00000008
0012FF3C  00000009
0012FF40  13789938
0012FF44  0040128B  return
```

Figure 11.25: *The stack state in the end*

Conclusion

In this chapter, we discussed about the working of an array with respect to reverse engineering. We saw the array code pattern in a disassembled code and understood how arrays are stored in contiguous memory locations. As an integer occupies 4 bytes of memory, so the integer array occupies 4 bytes multiplied by the number of elements in an array. We saw how contiguous memory locations are allocated in stack. We also covered the array program pattern when a code is optimized and not optimized. In the next chapter, we will talk about reversing structures that can handle dissimilar data types.

CHAPTER 12

Structure Code Pattern in Reverse Engineering

In the real world, we can describe an individual using several attributes. These attributes, or we can say parameters or characteristics, help us identify an individual uniquely. The attributes using which we can uniquely identify an individual can be their name, age, sex, height, weight, nationality, and many more. All these attributes correlate to different types of data. It means that the age is an integer, the sex is a string, the weight can be float, and the nationality is a string.

Now, as a computer programmer, if we have to code an application to record the details of all the individuals present in a geographical location, then we have to write an application in such a way that it can handle the data of the individuals in a well-managed and easy manner. In the earlier chapter, we have already seen ordinary variables that can hold a single piece of information. We have also seen how an array can hold data of a similar data type. But in the case of recording individual data, we have to use structures, which are used to record data of dissimilar data types. In this chapter, we will be reversing a structure which is used in many applications.

Structure

In this chapter, we will cover the following topics:

Understanding of structures

Structure without Optimization

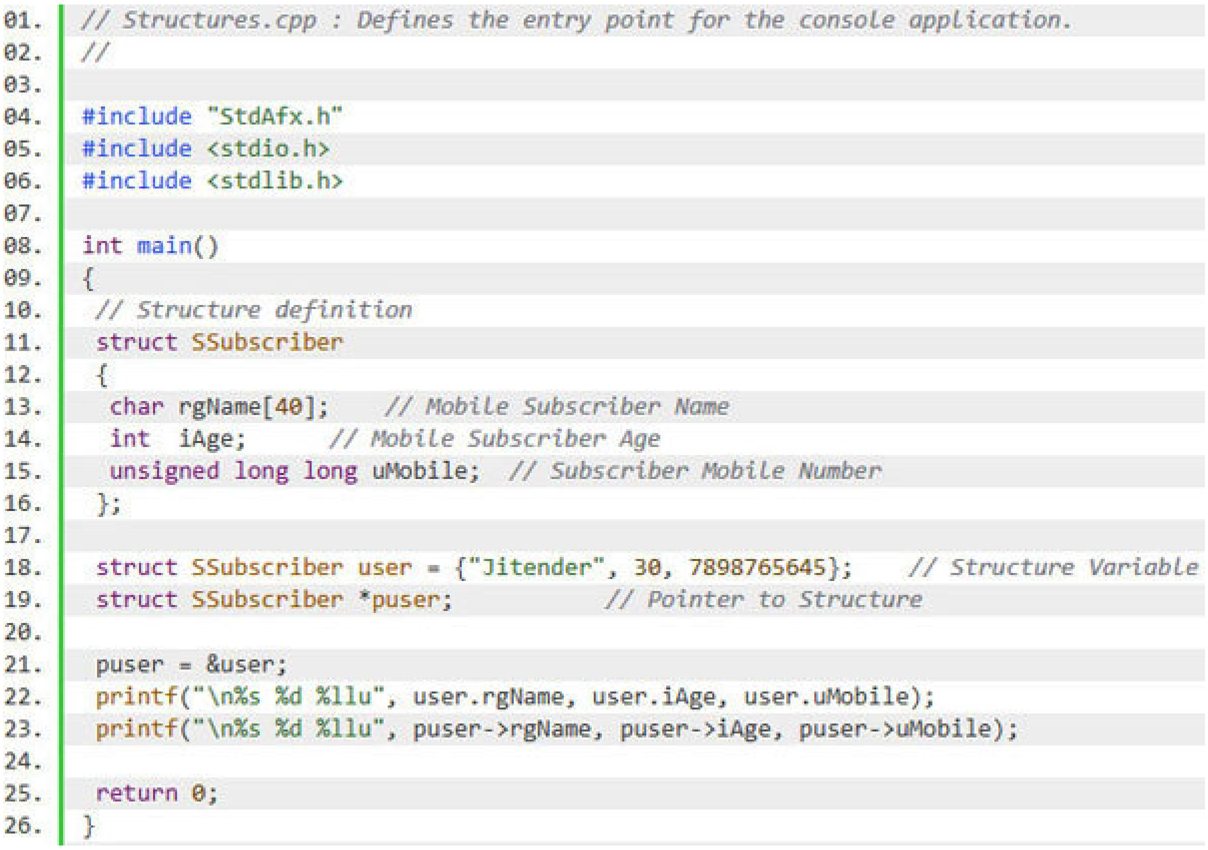
Structure with Optimization

Objective

In this chapter, we will study about pointers to structures with respect to reverse engineering. We will talk about structures code pattern in disassembled code and how structures are stored in memory. We will also cover structures program with optimized and not optimized code.

Understanding of structures

In this example, let's demonstrate the structure pointer. As we have a pointer to an integer or a pointer to a char, similarly, we have a pointer to structures. In C, we have an arrow operator that refers to the elements of a structure.



```
// Structures.cpp : Defines the entry point for the console application.
//

#include "StdAfx.h"
#include <stdio.h>
#include <stdlib.h>

int main()
{
 // Structure definition
 struct SSubscriber
 {
  char rgName[40];    // Mobile Subscriber Name
  int  iAge;      // Mobile Subscriber Age
  unsigned long long uMobile;  // Subscriber Mobile Number
 };

 struct SSubscriber user = {"Jitender", 30, 7898765645};    // Structure Variable
 struct SSubscriber *puser;           // Pointer to Structure

 puser = &user;
 printf("\n%s %d %llu", user.rgName, user.iAge, user.uMobile);
 printf("\n%s %d %llu", puser->rgName, puser->iAge, puser->uMobile);

 return 0;
}
```

Figure 12.1: *Structures.cpp*

Structure without Optimization

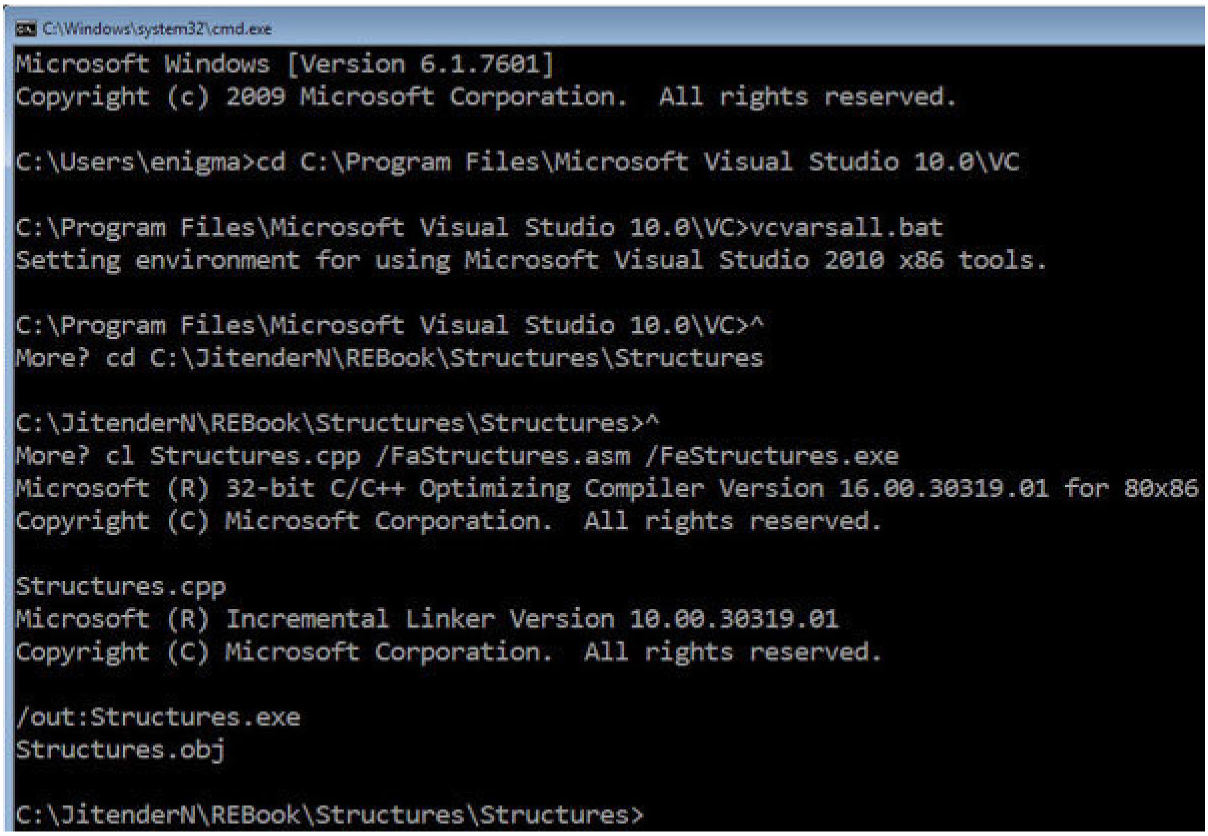
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

file file file file file file file file file file file file file file file

The following is the output of running the preceding commands:

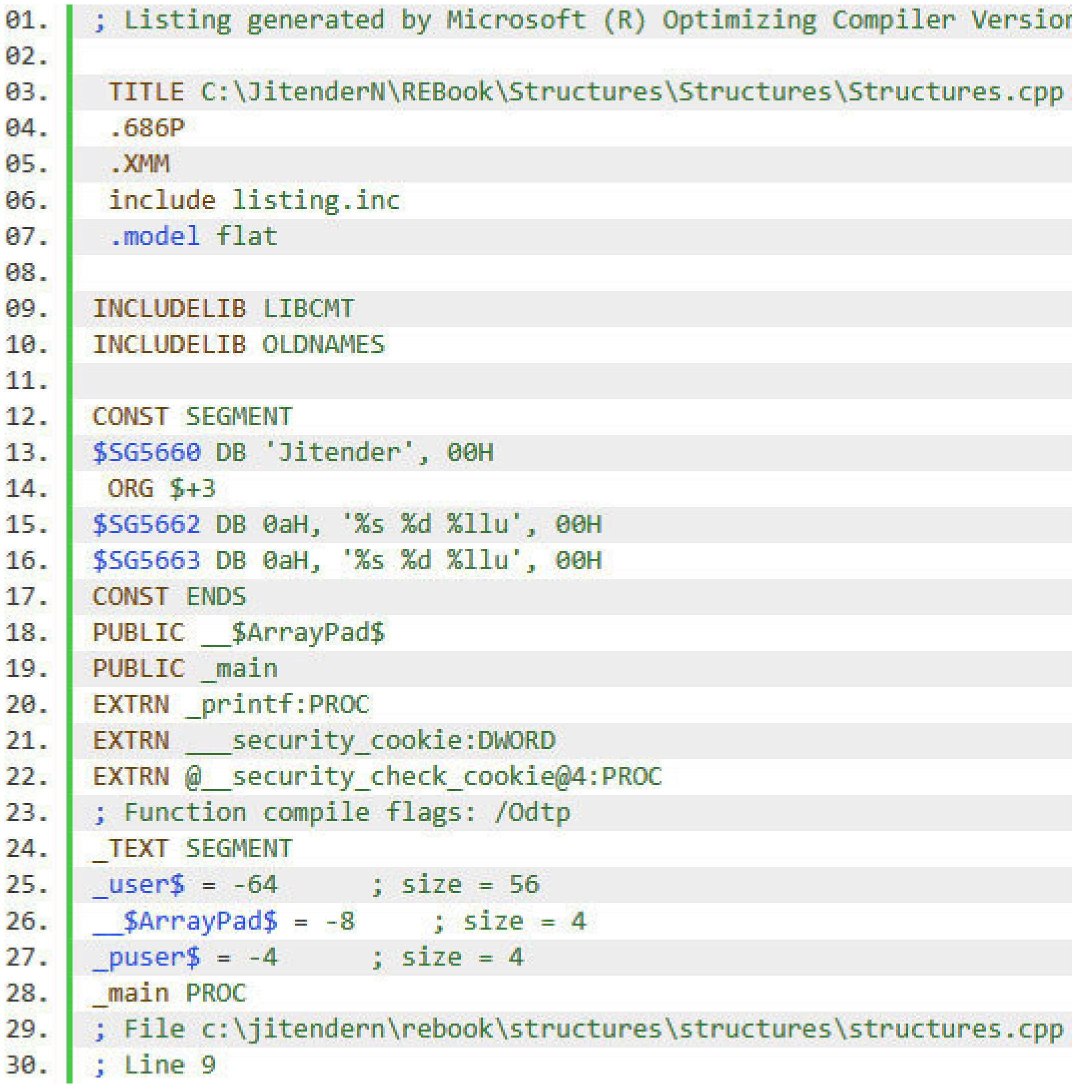


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Structures\Structures

C:\JitenderN\REBook\Structures\Structures>^
More? cl Structures.cpp /FaStructures.asm /FeStructures.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Structures.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Structures.exe
Structures.obj

C:\JitenderN\REBook\Structures\Structures>
```

Figure 12.2: *Structure without Optimization*

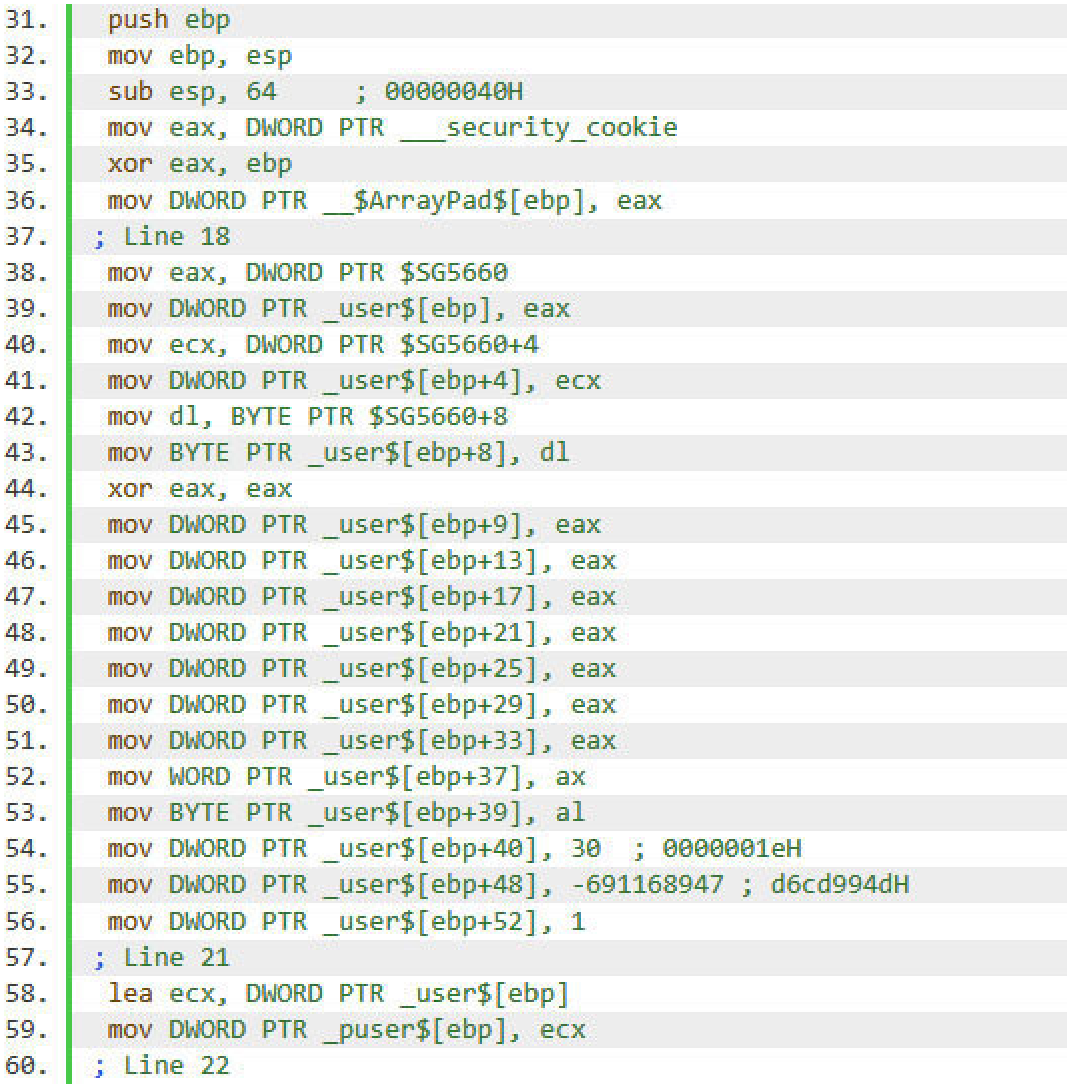
The compilation generates the EXE file and assembly code. Disable ASLR manually. To disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let us move onto the generated assembly listing:



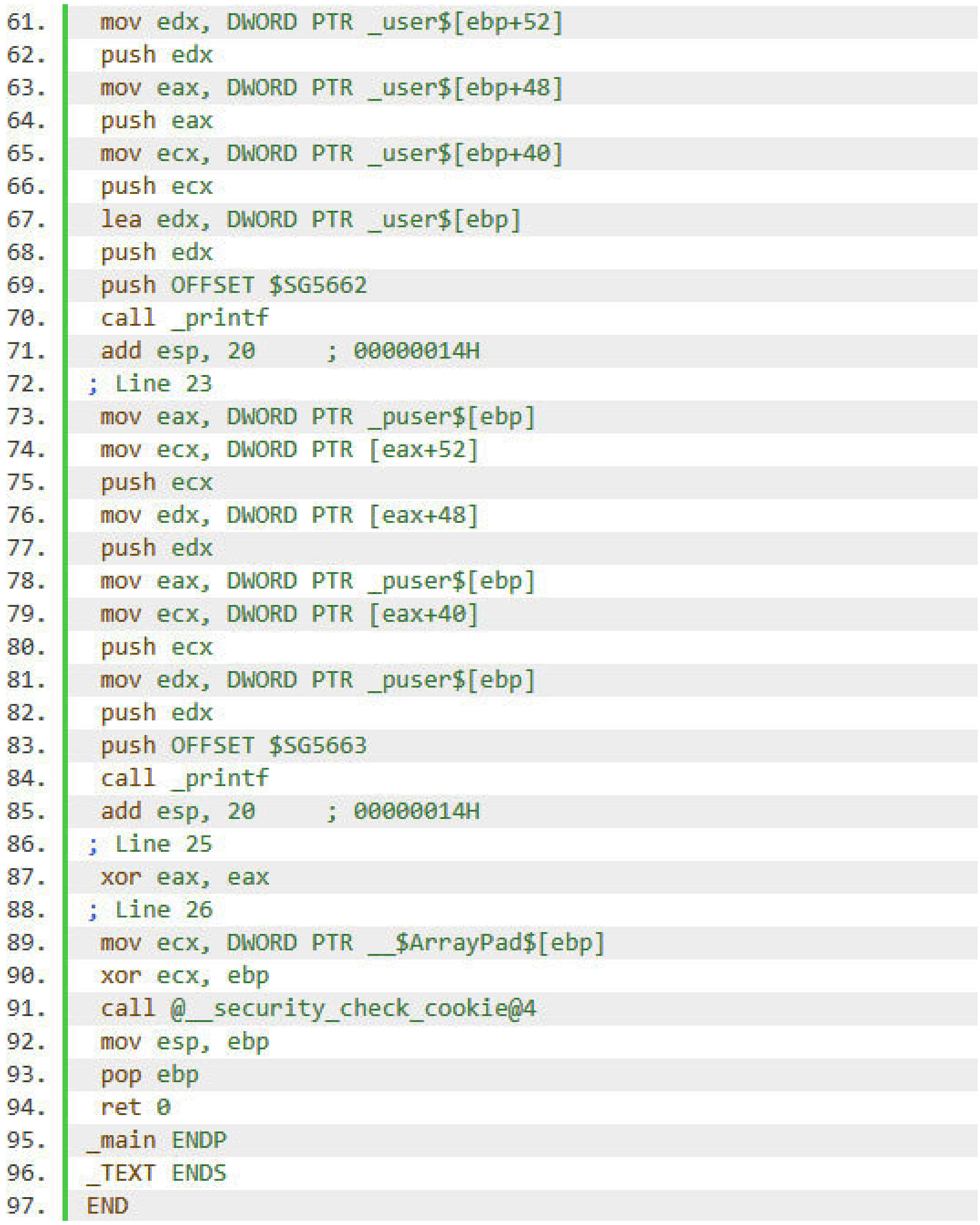
```
; Listing generated by Microsoft (R) Optimizing Compiler Versio

 TITLE C:\JitenderN\REBook\Structures\Structures\Structures.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG5660 DB 'Jitender', 00H
 ORG $+3
$SG5662 DB 0aH, '%s %d %llu', 00H
$SG5663 DB 0aH, '%s %d %llu', 00H
CONST ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_user$ = -64      ; size = 56
__$ArrayPad$ = -8      ; size = 4
_puser$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\structures\structures\structures.cpp
; Line 9
```

Figure 12.3: *Structures.asm-Part-1*



```
 push ebp
 mov ebp, esp
 sub esp, 64      ; 00000040H
 mov eax, DWORD PTR ___security_cookie
 xor eax, ebp
 mov DWORD PTR __$ArrayPad$[ebp], eax
; Line 18
 mov eax, DWORD PTR $SG5660
 mov DWORD PTR _user$[ebp], eax
 mov ecx, DWORD PTR $SG5660+4
 mov DWORD PTR _user$[ebp+4], ecx
 mov dl, BYTE PTR $SG5660+8
 mov BYTE PTR _user$[ebp+8], dl
 xor eax, eax
 mov DWORD PTR _user$[ebp+9], eax
 mov DWORD PTR _user$[ebp+13], eax
 mov DWORD PTR _user$[ebp+17], eax
 mov DWORD PTR _user$[ebp+21], eax
 mov DWORD PTR _user$[ebp+25], eax
 mov DWORD PTR _user$[ebp+29], eax
 mov DWORD PTR _user$[ebp+33], eax
 mov WORD PTR _user$[ebp+37], ax
 mov BYTE PTR _user$[ebp+39], al
 mov DWORD PTR _user$[ebp+40], 30  ; 0000001eH
 mov DWORD PTR _user$[ebp+48], -691168947 ; d6cd994dH
 mov DWORD PTR _user$[ebp+52], 1
; Line 21
 lea ecx, DWORD PTR _user$[ebp]
 mov DWORD PTR _puser$[ebp], ecx
; Line 22
```

Figure 12.4: *Structures.asm-Part-2*



```
61.  mov edx, DWORD PTR _user$[ebp+52]
62.  push edx
63.  mov eax, DWORD PTR _user$[ebp+48]
64.  push eax
65.  mov ecx, DWORD PTR _user$[ebp+40]
66.  push ecx
67.  lea edx, DWORD PTR _user$[ebp]
68.  push edx
69.  push OFFSET $SG5662
70.  call _printf
71.  add esp, 20      ; 00000014H
72. ; Line 23
73.  mov eax, DWORD PTR _puser$[ebp]
74.  mov ecx, DWORD PTR [eax+52]
75.  push ecx
76.  mov edx, DWORD PTR [eax+48]
77.  push edx
78.  mov eax, DWORD PTR _puser$[ebp]
79.  mov ecx, DWORD PTR [eax+40]
80.  push ecx
81.  mov edx, DWORD PTR _puser$[ebp]
82.  push edx
83.  push OFFSET $SG5663
84.  call _printf
85.  add esp, 20      ; 00000014H
86. ; Line 25
87.  xor eax, eax
88. ; Line 26
89.  mov ecx, DWORD PTR __$ArrayPad$[ebp]
90.  xor ecx, ebp
91.  call @__security_check_cookie@4
92.  mov esp, ebp
93.  pop ebp
94.  ret 0
95. _main ENDP
96. _TEXT ENDS
97. END
```

Figure 12.5: Structures.asm-Part-3

Let's move onto the explanation of the ASM code:

▼Line 30-33

; Line 9

```
push ebp
mov ebp, esp
sub esp, 64         ; 00000040H
```

It starts with the **main** function prologue, wherein the old **EBP** is pushed onto the stack and the current **ESP** is moved to a new

The **SUB** instruction creates room for the security cookie and the local variable by subtracting 64 bytes from the **ESP** register. Space allocated for variables on stack can be segregated as follows:

char rgName[40]; = 40 bytes
int iAge; = 4 bytes
unsigned long long uMobile; = 8 bytes
struct SSubscriber *puser; = 4 bytes
Stack Cookie = 4 bytes

▼Line 34-36

```
mov eax, DWORD PTR ___security_cookie
xor eax, ebp
mov DWORD PTR __$ArrayPad$[ebp], eax
```

As shown in the following screenshot, the stack cookie stored at **0x0040B000** is moved to **EAX** where it is XOR'ed with The result of XOR from **EAX** is moved back to which is **[EBP-0x08].** The stack at this point in time is as follows:

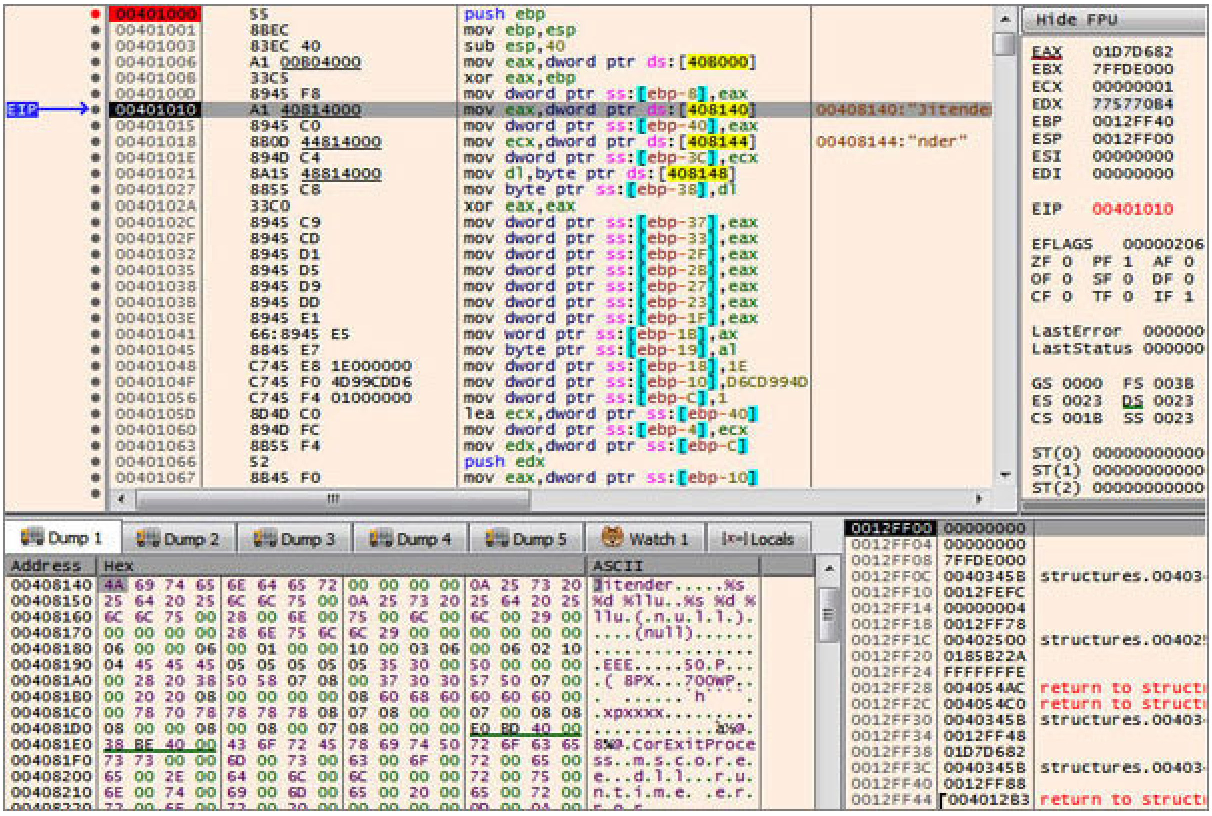
[ESP] 0012FF00 00000000 Right now JUNK, space for local variables
[ESP+0x04] 0012FF04 00000000 Right now JUNK, space for local variables
[ESP+0x08] 0012FF08 7FFDE000 Right now JUNK, space for local variables
[ESP+0x0C] 0012FF0C 0040345B Right now JUNK, space for local variables
[ESP+0x10] 0012FF10 0012FEFC Right now JUNK, space for local variables

[ESP+0x14] 0012FF14 00000004 Right now JUNK, space for local variables
[ESP+0x18] 0012FF18 0012FF78 Right now JUNK, space for local variables
[ESP+0x1C] 0012FF1C 00402500 Right now JUNK, space for local variables
[ESP+0x20] 0012FF20 0185B22A Right now JUNK, space for local variables
[ESP+0x24] 0012FF24 FFFFFFFE Right now JUNK, space for local variables
[ESP+0x28] 0012FF28 004054AC Right now JUNK, space for local variables
[ESP+0x2C] 0012FF2C 004054C0 Right now JUNK, space for local variables
[ESP+0x30] 0012FF30 0040345B Right now JUNK, space for local variables
[ESP+0x34] 0012FF34 0012FF48 Right now JUNK, space for local variables
[ESP+0x38] 0012FF38 01D7D682 XOR of Stack Cookie and EAX is stored here

[ESP+0x3C] 0012FF3C   0040345B   Right now JUNK, space for local variables

[ESP+0x40] 0012FF40   0012FF88   [EBP]

[ESP+0x44] 0012FF44   004012B3   return to structures.004012B3



```
00401000   55               push ebp
00401001   8BEC             mov ebp,esp
00401003   83EC 40          sub esp,40
00401006   A1 00804000      mov eax,dword ptr ds:[408000]
0040100B   33C5             xor eax,ebp
0040100D   8945 F8          mov dword ptr ss:[ebp-8],eax
00401010   A1 40814000      mov eax,dword ptr ds:[408140]     00408140:"Jitende
00401015   8945 C0          mov dword ptr ss:[ebp-40],eax
00401018   8B0D 44814000    mov ecx,dword ptr ds:[408144]     00408144:"nder"
0040101E   894D C4          mov dword ptr ss:[ebp-3C],ecx
00401021   8A15 48814000    mov dl,byte ptr ds:[408148]
00401027   8855 C8          mov byte ptr ss:[ebp-38],dl
0040102A   33C0             xor eax,eax
0040102C   8945 C9          mov dword ptr ss:[ebp-37],eax
0040102F   8945 CD          mov dword ptr ss:[ebp-33],eax
00401032   8945 D1          mov dword ptr ss:[ebp-2F],eax
00401035   8945 D5          mov dword ptr ss:[ebp-2B],eax
00401038   8945 D9          mov dword ptr ss:[ebp-27],eax
0040103B   8945 DD          mov dword ptr ss:[ebp-23],eax
0040103E   8945 E1          mov dword ptr ss:[ebp-1F],eax
00401041   66:8945 E5       mov word ptr ss:[ebp-1B],ax
00401045   8845 E7          mov byte ptr ss:[ebp-19],al
00401048   C745 E8 1E000000 mov dword ptr ss:[ebp-18],1E
0040104F   C745 F0 4D99CDD6 mov dword ptr ss:[ebp-10],D6CD994D
00401056   C745 F4 01000000 mov dword ptr ss:[ebp-C],1
0040105D   8D4D C0          lea ecx,dword ptr ss:[ebp-40]
00401060   894D FC          mov dword ptr ss:[ebp-4],ecx
00401063   8B55 F4          mov edx,dword ptr ss:[ebp-C]
00401066   52               push edx
00401067   8B45 F0          mov eax,dword ptr ss:[ebp-10]
```

```
EAX 01D7D682
EBX 7FFDE000
ECX 00000001
EDX 77577084
EBP 0012FF40
ESP 0012FF00
ESI 00000000
EDI 00000000

EIP 00401010

EFLAGS  00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  000000
LastStatus 000000

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000
ST(1) 00000000000
ST(2) 00000000000
```

```
Address  Hex                                              ASCII
00408140 4A 69 74 65 6E 64 65 72 00 00 00 00 0A 25 73 20 Jitender.....%s
00408150 25 64 20 25 6C 6C 75 00 0A 25 73 20 25 64 20 25 %d %llu..%s %d %
00408160 6C 6C 75 00 28 00 6E 00 75 00 6C 00 6C 00 29 00 llu.(.n.u.l.l.).
00408170 00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00 ....(null)......
00408180 06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10 ................
00408190 04 45 45 45 05 05 05 05 05 35 30 00 50 00 00 00 .EEE.....50.P...
004081A0 00 28 20 38 50 58 07 08 00 37 30 30 57 50 07 00 .( 8PX...700WP..
004081B0 00 20 20 08 00 00 00 00 08 60 68 60 60 60 60 00 .  .....`h````.
004081C0 00 78 70 78 78 78 78 08 07 08 00 00 07 00 08 08 .xpxxxx.........
004081D0 08 00 00 08 00 08 00 07 08 00 00 00 E0 BD 40 00 ............à½@.
004081E0 38 BE 40 00 43 6F 72 45 78 69 74 50 72 6F 63 65 8¾@.CorExitProce
004081F0 73 73 00 00 6D 00 73 00 63 00 6F 00 72 00 65 00 ss..m.s.c.o.r.e.
00408200 65 00 2E 00 64 00 6C 00 6C 00 00 00 72 00 75 00 e...d.l.l...r.u.
00408210 6E 00 74 00 69 00 6D 00 65 00 20 00 65 00 72 00 n.t.i.m.e. .e.r.
```

```
0012FF00 00000000
0012FF04 00000000
0012FF08 7FFDE000
0012FF0C 0040345B structures.00403
0012FF10 0012FEFC
0012FF14 00000004
0012FF18 0012FF78
0012FF1C 00402500 structures.00402
0012FF20 0185B22A
0012FF24 FFFFFFFE
0012FF28 004054AC return to struct
0012FF2C 004054C0 return to struct
0012FF30 0040345B structures.00403
0012FF34 0012FF48
0012FF38 01D7D682
0012FF3C 0040345B structures.00403
0012FF40 0012FF88
0012FF44 004012B3 return to struct
```

Figure 12.6: Stack cookie

▼Line 37-56

; Line 18

mov eax, DWORD PTR $SG5660

mov DWORD PTR _user$[ebp], eax

mov ecx, DWORD PTR $SG5660+4

mov DWORD PTR _user$[ebp+4], ecx

mov dl, BYTE PTR $SG5660+8

```
mov BYTE PTR _user$[ebp+8], dl
xor eax, eax
mov DWORD PTR _user$[ebp+9], eax
mov DWORD PTR _user$[ebp+13], eax
mov DWORD PTR _user$[ebp+17], eax
mov DWORD PTR _user$[ebp+21], eax
mov DWORD PTR _user$[ebp+25], eax
mov DWORD PTR _user$[ebp+29], eax
mov DWORD PTR _user$[ebp+33], eax

mov WORD PTR _user$[ebp+37], ax
mov BYTE PTR _user$[ebp+39], al
mov DWORD PTR _user$[ebp+40], 30   ; 0000001eH
mov DWORD PTR _user$[ebp+48], -691168947 ; d6cd994dH
mov DWORD PTR _user$[ebp+52], 1
; Line 18
```

This is a comment which states that the ASM instructions preceding it will represent line 18 of the C/C++ code, which is:

```
struct SSubscriber user = {"Jitender", 30, 7898765645};      // Structure Variable
```

In the ASM code, we see several **MOV** instructions. To understand these instructions, we will have to understand that the elements of structure are always stored in contiguous memory locations. All the **MOV** instructions will store the elements of structure onto the stack. To understand the stack state, let's first take each element of structure in a hex representation as follows:

char rgName[40]; = 40 bytes = Jitender (0x4A6974656E646572)

(0x4A6974656E646572)

(0x4A6974656E646572)

int iAge; = 4 bytes = 30 (0x0000001E)

(0x0000001E)

unsigned long long uMobile; = 8 bytes = 7898765645 (0x00000001D6CD994D)

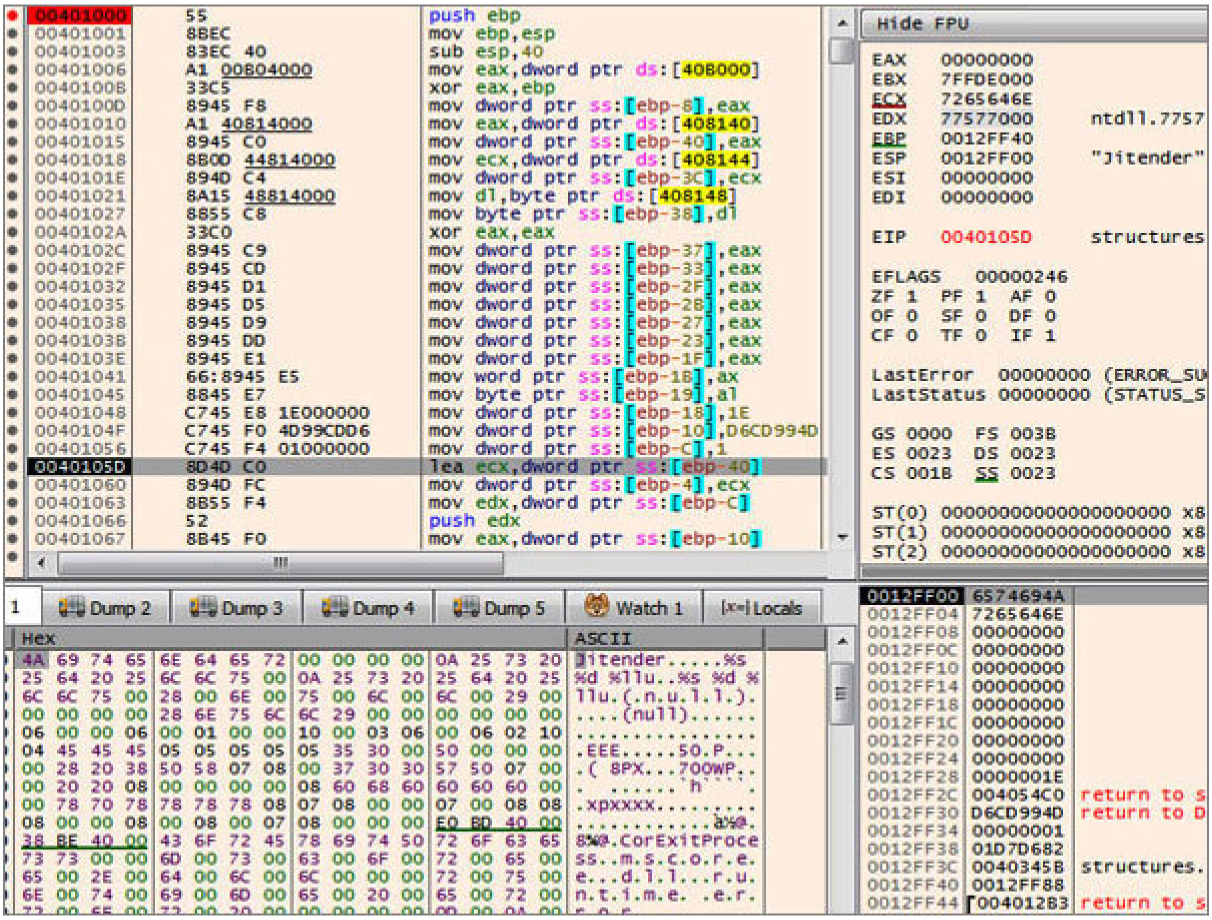
(0x00000001D6CD994D)

The stack state after all the MOV instructions will be as follows:

[ESP] 0012FF00 6574694A 4 bytes of array, rgName in little-endian etiJ

[ESP+0x04] 0012FF04 7265646E 4 bytes of array, rgName in little-endian redn
[ESP+0x08] 0012FF08 00000000 String terminator with NULL in remaining array
[ESP+0x0C] 0012FF0C 00000000 NULL values in char array of size 40 bytes
[ESP+0x10] 0012FF10 00000000 NULL values in char array of size 40 bytes

```
[ESP+0x14] 0012FF14   00000000   NULL values in char array of
size 40 bytes
[ESP+0x18] 0012FF18   00000000   NULL values in char array of
size 40 bytes
[ESP+0x1C] 0012FF1C   00000000   NULL values in char array of
size 40 bytes
[ESP+0x20] 0012FF20   00000000   NULL values in char array of
size 40 bytes
[ESP+0x24] 0012FF24   00000000   array from 0x0012FF00 to
0x0012FF28, total 40 bytes
[ESP+0x28] 0012FF28   0000001E   Second element of Structure,
iAge is stored here
[ESP+0x2C] 0012FF2C   004054C0   return to 0x004054C0 from
0x00405477
[ESP+0x30] 0012FF30   D6CD994D   4 bytes of uMobile stored
here
[ESP+0x34] 0012FF34   00000001   remaining 4 bytes of uMobile
stored here
[ESP+0x38] 0012FF38   01D7D682   XOR of Stack Cookie and EAX
is stored here
[ESP+0x3C] 0012FF3C   0040345B   structures.0040345B

[ESP+0x40] 0012FF40   0012FF88   [EBP]
[ESP+0x44] 0012FF44   004012B3   return to 0x004012B3 from
0x00401000
```



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   83EC 40         sub esp,40
00401006   A1 00804000     mov eax,dword ptr ds:[408000]
0040100B   33C5            xor eax,ebp
0040100D   8945 F8         mov dword ptr ss:[ebp-8],eax
00401010   A1 40814000     mov eax,dword ptr ds:[408140]
00401015   8945 C0         mov dword ptr ss:[ebp-40],eax
00401018   8B0D 44814000   mov ecx,dword ptr ds:[408144]
0040101E   894D C4         mov dword ptr ss:[ebp-3C],ecx
00401021   8A15 48814000   mov dl,byte ptr ds:[408148]
00401027   8855 C8         mov byte ptr ss:[ebp-38],dl
0040102A   33C0            xor eax,eax
0040102C   8945 C9         mov dword ptr ss:[ebp-37],eax
0040102F   8945 CD         mov dword ptr ss:[ebp-33],eax
00401032   8945 D1         mov dword ptr ss:[ebp-2F],eax
00401035   8945 D5         mov dword ptr ss:[ebp-2B],eax
00401038   8945 D9         mov dword ptr ss:[ebp-27],eax
0040103B   8945 DD         mov dword ptr ss:[ebp-23],eax
0040103E   8945 E1         mov dword ptr ss:[ebp-1F],eax
00401041   66:8945 E5      mov word ptr ss:[ebp-1B],ax
00401045   8845 E7         mov byte ptr ss:[ebp-19],al
00401048   C745 E8 1E000000 mov dword ptr ss:[ebp-18],1E
0040104F   C745 F0 4D99CDD6 mov dword ptr ss:[ebp-10],D6CD994D
00401056   C745 F4 01000000 mov dword ptr ss:[ebp-C],1
0040105D   8D4D C0         lea ecx,dword ptr ss:[ebp-40]
00401060   894D FC         mov dword ptr ss:[ebp-4],ecx
00401063   8B55 F4         mov edx,dword ptr ss:[ebp-C]
00401066   52              push edx
00401067   8B45 F0         mov eax,dword ptr ss:[ebp-10]
```

```
EAX   00000000
EBX   7FFDE000
ECX   7265646E
EDX   77577000      ntdll.7757
EBP   0012FF40
ESP   0012FF00      "Jitender"
ESI   00000000
EDI   00000000

EIP   0040105D      structures

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SU
LastStatus 00000000 (STATUS_S

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000000 x8
ST(1) 0000000000000000000000 x8
ST(2) 0000000000000000000000 x8
```

```
Hex                                                ASCII
4A 69 74 65 6E 64 65 72 00 00 00 00 0A 25 73 20   Jitender.....%s
25 64 20 25 6C 6C 75 00 0A 25 73 20 25 64 20 25   %d %llu..%s %d %
6C 6C 75 00 28 00 6E 00 75 00 6C 00 6C 00 29 00   llu.(.n.u.l.l.).
00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00   ....(null)......
06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10   ................
04 45 45 45 05 05 05 05 05 35 30 00 50 00 00 00   .EEE.....50.P...
00 28 20 38 50 58 07 08 00 37 30 30 57 50 07 00   .( 8PX...700WP..
00 20 20 08 00 00 00 00 08 60 68 60 60 60 60 00   .  .....`h````.
00 78 70 78 78 78 78 08 07 08 00 00 07 00 08 08   .xpxxxx.........
08 00 00 08 00 08 00 07 08 00 00 00 E0 BD 40 00   ............à½@.
38 BE 40 00 43 6F 72 45 78 69 74 50 72 6F 63 65   8¾@.CorExitProce
73 73 00 00 6D 00 73 00 63 00 6F 00 72 00 65 00   ss..m.s.c.o.r.e.
65 00 2E 00 64 00 6C 00 6C 00 00 00 72 00 75 00   e...d.l.l...r.u.
6E 00 74 00 69 00 6D 00 65 00 20 00 65 00 72 00   n.t.i.m.e. .e.r.
```

```
0012FF00  6574694A
0012FF04  7265646E
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  0000001E
0012FF2C  004054C0  return to s
0012FF30  D6CD994D  return to D
0012FF34  00000001
0012FF38  01D7D682
0012FF3C  0040345B  structures.
0012FF40  0012FF88
0012FF44  00401283  return to s
```

Figure 12.7: Stack after all the MOV instructions

▼Line 57-59

; Line 21
lea ecx, DWORD PTR _user$[ebp]
mov DWORD PTR _puser$[ebp], ecx

This is a comment which states that the ASM instructions preceding it will represent line 21 of the C/C++ code, which is:

puser = &user;

In the ASM code, the pointer to structure is pushed onto the **ECX** register using the Load Effective Address instruction. The pointer to the structure points to the first element of the structure, as structures are always stored in contiguous memory locations.

The **MOV** instruction will push the pointer to structure on the stack at **[EBP-0x04].** The stack state after this instruction will be as follows:

[ESP] 0012FF00 6574694A 4 bytes of array, rgName in little-endian etiJ

[ESP+0x04] 0012FF04 7265646E 4 bytes of array, rgName in little-endian redn

[ESP+0x08] 0012FF08 00000000 String terminator with NULL in remaining array

[ESP+0x0C] 0012FF0C 00000000 NULL values in char array of size 40 bytes

[ESP+0x10] 0012FF10 00000000 NULL values in char array of size 40 bytes

[ESP+0x14] 0012FF14 00000000 NULL values in char array of size 40 bytes

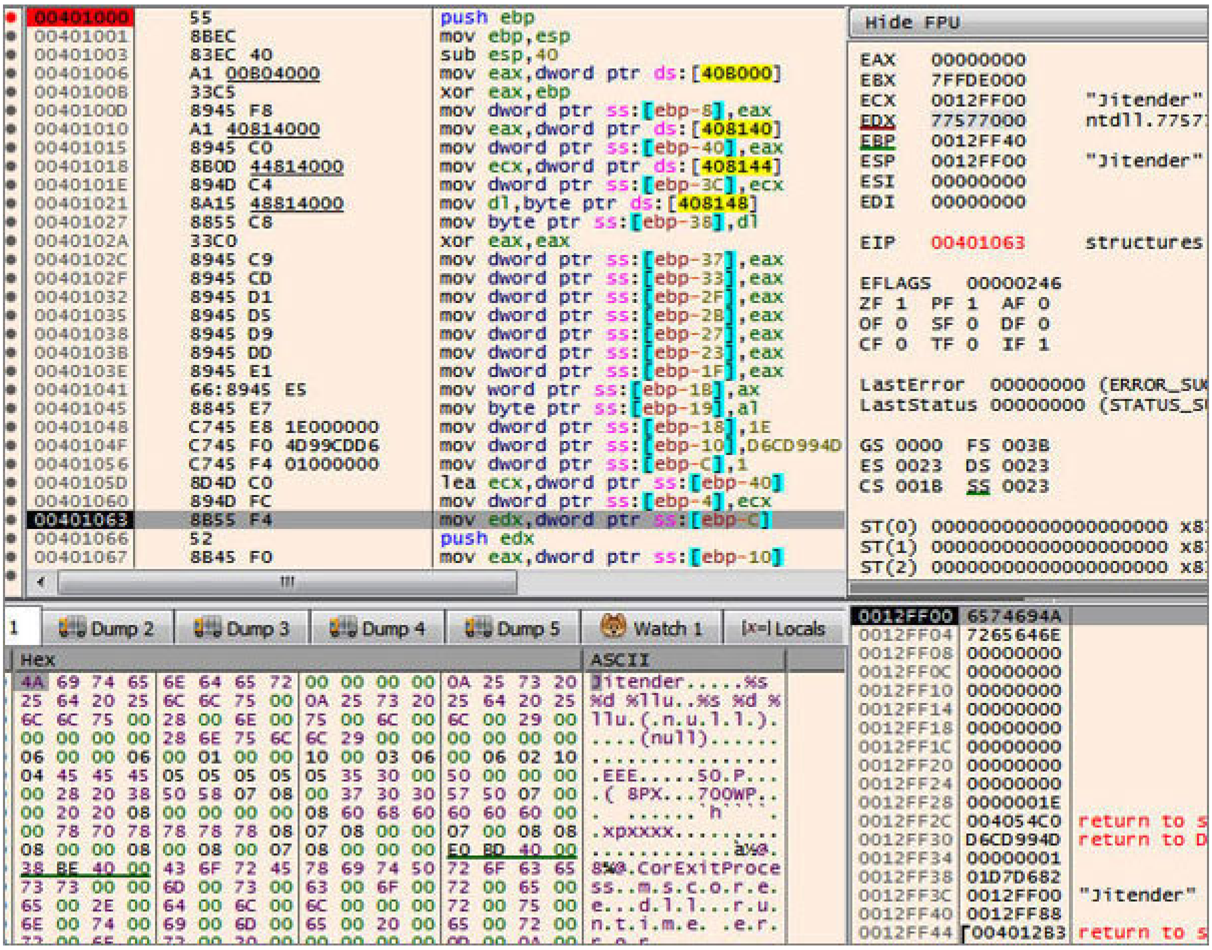
[ESP+0x18] 0012FF18 00000000 NULL values in char array of size 40 bytes

[ESP+0x1C] 0012FF1C 00000000 NULL values in char array of size 40 bytes

[ESP+0x20] 0012FF20 00000000 NULL values in char array of size 40 bytes

[ESP+0x24] 0012FF24 00000000 rgName from 0x0012FF00 to 0x0012FF28, total 40 bytes

```
[ESP+0x28] 0012FF28   0000001E   Second element of Structure,
iAge is stored here
[ESP+0x2C] 0012FF2C   004054C0   return to 0x004054C0 from
0x00405477

[ESP+0x30] 0012FF30   D6CD994D   4 bytes of uMobile stored
here
[ESP+0x34] 0012FF34   00000001   remaining 4 bytes of uMobile
stored here
[ESP+0x38] 0012FF38   01D7D682   XOR of Stack Cookie and EAX
is stored here
[ESP+0x3C] 0012FF3C   0012FF00   Pointer to structure is stored
here, puser
[ESP+0x40] 0012FF40   0012FF88   [EBP]
[ESP+0x44] 0012FF44   004012B3    return to 0x004012B3 from
0x00401000
```



```
00401000   55                  push ebp
00401001   8BEC                mov ebp,esp
00401003   83EC 40             sub esp,40
00401006   A1 00804000         mov eax,dword ptr ds:[408000]
0040100B   33C5                xor eax,ebp
0040100D   8945 F8             mov dword ptr ss:[ebp-8],eax
00401010   A1 40814000         mov eax,dword ptr ds:[408140]
00401015   8945 C0             mov dword ptr ss:[ebp-40],eax
00401018   8B0D 44814000       mov ecx,dword ptr ds:[408144]
0040101E   894D C4             mov dword ptr ss:[ebp-3C],ecx
00401021   8A15 48814000       mov dl,byte ptr ds:[408148]
00401027   8855 C8             mov byte ptr ss:[ebp-38],dl
0040102A   33C0                xor eax,eax
0040102C   8945 C9             mov dword ptr ss:[ebp-37],eax
0040102F   8945 CD             mov dword ptr ss:[ebp-33],eax
00401032   8945 D1             mov dword ptr ss:[ebp-2F],eax
00401035   8945 D5             mov dword ptr ss:[ebp-2B],eax
00401038   8945 D9             mov dword ptr ss:[ebp-27],eax
0040103B   8945 DD             mov dword ptr ss:[ebp-23],eax
0040103E   8945 E1             mov dword ptr ss:[ebp-1F],eax
00401041   66:8945 E5          mov word ptr ss:[ebp-1B],ax
00401045   8845 E7             mov byte ptr ss:[ebp-19],al
00401048   C745 E8 1E000000    mov dword ptr ss:[ebp-18],1E
0040104F   C745 F0 4D99CDD6    mov dword ptr ss:[ebp-10],D6CD994D
00401056   C745 F4 01000000    mov dword ptr ss:[ebp-C],1
0040105D   8D4D C0             lea ecx,dword ptr ss:[ebp-40]
00401060   894D FC             mov dword ptr ss:[ebp-4],ecx
00401063   8B55 F4             mov edx,dword ptr ss:[ebp-C]
00401066   52                  push edx
00401067   8B45 F0             mov eax,dword ptr ss:[ebp-10]
```

```
EAX   00000000
EBX   7FFDE000
ECX   0012FF00      "Jitender"
EDX   77577000      ntdll.7757
EBP   0012FF40
ESP   0012FF00      "Jitender"
ESI   00000000
EDI   00000000

EIP   00401063      structures

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SU
LastStatus 00000000 (STATUS_S

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x8
ST(1) 00000000000000000000 x8
ST(2) 00000000000000000000 x8
```

```
Hex                                                  ASCII
4A 69 74 65 6E 64 65 72 00 00 00 00 0A 25 73 20   Jitender.....%s
25 64 20 25 6C 6C 75 00 0A 25 73 20 25 64 20 25   %d %llu..%s %d %
6C 6C 75 00 28 00 6E 00 75 00 6C 00 6C 00 29 00   llu.(.n.u.l.l.).
00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00   ....(null)......
06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10   ................
04 45 45 45 05 05 05 05 05 35 30 00 50 00 00 00   .EEE.....50.P...
00 28 20 38 50 58 07 08 00 37 30 30 57 50 07 00   .( 8PX...700WP..
00 20 20 08 00 00 00 00 08 60 68 60 60 60 60 00   .  .....`h````.
00 78 70 78 78 78 78 08 07 08 00 00 07 00 08 08   .xpxxxx.........
08 00 00 08 00 08 00 07 08 00 00 00 E0 BD 40 00   ............à½@.
38 BE 40 00 43 6F 72 45 78 69 74 50 72 6F 63 65   8¾@.CorExitProce
73 73 00 00 6D 00 73 00 63 00 6F 00 72 00 65 00   ss..m.s.c.o.r.e.
65 00 2E 00 64 00 6C 00 6C 00 00 00 72 00 75 00   e...d.l.l...r.u.
6E 00 74 00 69 00 6D 00 65 00 20 00 65 00 72 00   n.t.i.m.e. .e.r.
```

```
0012FF00  6574694A
0012FF04  7265646E
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  0000001E
0012FF2C  004054C0  return to s
0012FF30  D6CD994D  return to D
0012FF34  00000001
0012FF38  01D7D682
0012FF3C  0012FF00  "Jitender"
0012FF40  0012FF88
0012FF44  00401283  return to s
```

Figure 12.8: *Pointer to structure on stack*

▼Line 60-71

```
; Line 22
mov edx, DWORD PTR _user$[ebp+52]
push edx

mov eax, DWORD PTR _user$[ebp+48]
push eax
mov ecx, DWORD PTR _user$[ebp+40]
push ecx
lea edx, DWORD PTR _user$[ebp]
push edx
```

```
push OFFSET $SG5662
call _printf
add esp, ; 00000014H
```

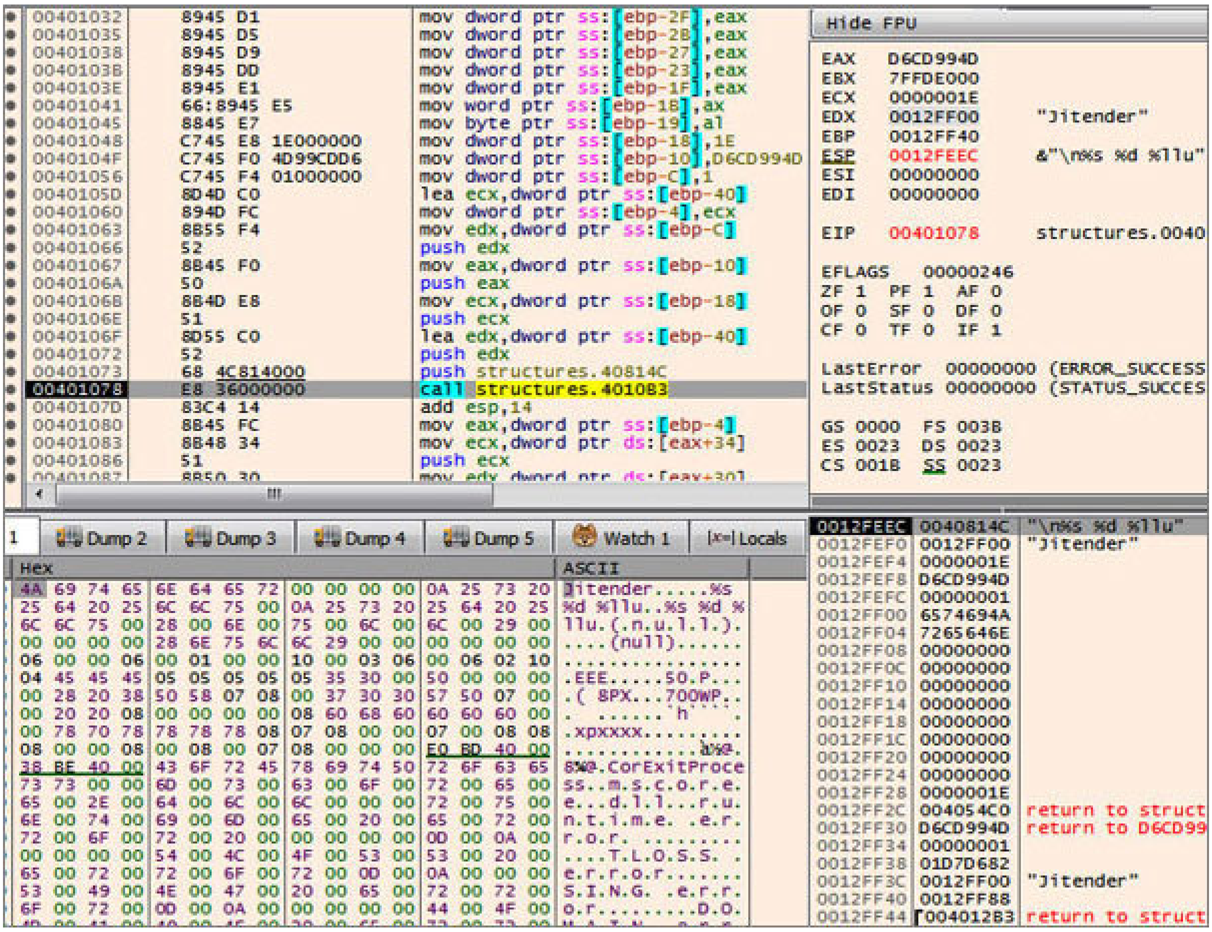
This is a comment which states that the ASM instructions preceding it will represent line 22 of the C/C++ code, which is:

```
printf("\n%s %d %llu", user.rgName, user.iAge, user.uMobile);
```

In the ASM code, we are pushing all the arguments to the stack one by one and then a call to the **printf** function is made. The stack state just before the call to the **printf** function is as follows:

```
[ESP]  0012FEEC  0040814C  "\n%s %d %llu", argument to printf() is pushed here
[ESP+0x04] 0012FEF0  0012FF00  pointer to rgName array is pushed here
[ESP+0x08] 0012FEF4  0000001E  iAge value is pushed here
[ESP+0x0C] 0012FEF8  D6CD994D  uMobile value is pushed here
[ESP+0x10] 0012FEFC  00000001  uMobile value is pushed here
[ESP+0x14] 0012FF00  6574694A  4 bytes of array, rgName in little-endian etiJ
[ESP+0x18] 0012FF04  7265646E  4 bytes of array, rgName in little-endian redn

[ESP+0x1C] 0012FF08  00000000  String terminator with NULL in remaining array
[ESP+0x20] 0012FF0C  00000000  NULL values in char array of size 40 bytes
[ESP+0x24] 0012FF10  00000000  NULL values in char array of size 40 bytes
```

```
[ESP+0x28] 0012FF14   00000000   NULL values in char array of size 40 bytes
[ESP+0x2C] 0012FF18   00000000   NULL values in char array of size 40 bytes
[ESP+0x30] 0012FF1C   00000000   NULL values in char array of size 40 bytes
[ESP+0x34] 0012FF20   00000000   NULL values in char array of size 40 bytes
[ESP+0x38] 0012FF24   00000000   rgName from 0x0012FF00 to 0x0012FF28, total 40 bytes
[ESP+0x3C] 0012FF28   0000001E   Second element of Structure, iAge is stored here
[ESP+0x40] 0012FF2C   004054C0   return to 0x004054C0 from 0x00405477
[ESP+0x44] 0012FF30   D6CD994D   4 bytes of uMobile stored here
[ESP+0x48] 0012FF34   00000001   remaining 4 bytes of uMobile stored here
[ESP+0x50] 0012FF38   01D7D682   XOR of Stack Cookie and EAX is stored here
[ESP+0x54] 0012FF3C   0012FF00   Pointer to structure is stored here, puser
[ESP+0x58] 0012FF40   0012FF88   [EBP]

[ESP+0x5C] 0012FF44   004012B3    return to 0x004012B3 from 0x00401000
```



```
00401032   8945 D1            mov dword ptr ss:[ebp-2F],eax
00401035   8945 D5            mov dword ptr ss:[ebp-2B],eax
00401038   8945 D9            mov dword ptr ss:[ebp-27],eax
0040103B   8945 DD            mov dword ptr ss:[ebp-23],eax
0040103E   8945 E1            mov dword ptr ss:[ebp-1F],eax
00401041   66:8945 E5         mov word ptr ss:[ebp-1B],ax
00401045   8845 E7            mov byte ptr ss:[ebp-19],al
00401048   C745 E8 1E000000   mov dword ptr ss:[ebp-18],1E
0040104F   C745 F0 4D99CDD6   mov dword ptr ss:[ebp-10],D6CD994D
00401056   C745 F4 01000000   mov dword ptr ss:[ebp-C],1
0040105D   8D4D C0            lea ecx,dword ptr ss:[ebp-40]
00401060   894D FC            mov dword ptr ss:[ebp-4],ecx
00401063   8B55 F4            mov edx,dword ptr ss:[ebp-C]
00401066   52                 push edx
00401067   8B45 F0            mov eax,dword ptr ss:[ebp-10]
0040106A   50                 push eax
0040106B   8B4D E8            mov ecx,dword ptr ss:[ebp-18]
0040106E   51                 push ecx
0040106F   8D55 C0            lea edx,dword ptr ss:[ebp-40]
00401072   52                 push edx
00401073   68 4C814000        push structures.40814C
00401078   E8 36000000        call structures.401083
0040107D   83C4 14            add esp,14
00401080   8B45 FC            mov eax,dword ptr ss:[ebp-4]
00401083   8B48 34            mov ecx,dword ptr ds:[eax+34]
00401086   51                 push ecx
```

```
Hide FPU

EAX   D6CD994D
EBX   7FFDE000
ECX   0000001E
EDX   0012FF00      "Jitender"
EBP   0012FF40
ESP   0012FEEC      &"\n%s %d %llu"
ESI   00000000
EDI   00000000

EIP   00401078      structures.0040

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS
LastStatus  00000000 (STATUS_SUCCES

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023
```

```
Hex                                                  ASCII
4A 69 74 65 6E 64 65 72 00 00 00 00 0A 25 73 20   Jitender.....%s
25 64 20 25 6C 6C 75 00 0A 25 73 20 25 64 20 25   %d %llu..%s %d %
6C 6C 75 00 28 00 6E 00 75 00 6C 00 6C 00 29 00   llu.(.n.u.l.l.).
00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00   ....(null)......
06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10   ................
04 45 45 45 05 05 05 05 05 35 30 00 50 00 00 00   .EEE.....50.P...
00 28 20 38 50 58 07 08 00 37 30 30 57 50 07 00   .( 8PX...700WP..
00 20 20 08 00 00 00 00 08 60 68 60 60 60 60 00   .  .....`h````.
00 78 70 78 78 78 78 08 07 08 00 00 07 00 08 08   .xpxxxx.........
08 00 00 08 00 08 00 07 08 00 00 00 E0 BD 40 00   ............à½@.
38 BE 40 00 43 6F 72 45 78 69 74 50 72 6F 63 65   8¾@.CorExitProce
73 73 00 00 6D 00 73 00 63 00 6F 00 72 00 65 00   ss..m.s.c.o.r.e.
65 00 2E 00 64 00 6C 00 6C 00 00 00 72 00 75 00   e...d.l.l...r.u.
6E 00 74 00 69 00 6D 00 65 00 20 00 65 00 72 00   n.t.i.m.e. .e.r.
72 00 6F 00 72 00 20 00 00 00 00 00 0D 00 0A 00   r.o.r. .........
00 00 00 00 54 00 4C 00 4F 00 53 00 53 00 20 00   ....T.L.O.S.S. .
65 00 72 00 72 00 6F 00 72 00 0D 00 0A 00 00 00   e.r.r.o.r.......
53 00 49 00 4E 00 47 00 20 00 65 00 72 00 72 00   S.I.N.G. .e.r.r.
6F 00 72 00 0D 00 0A 00 00 00 00 00 44 00 4F 00   o.r.........D.O.
```

```
0012FEEC  0040814C  "\n%s %d %llu"
0012FEF0  0012FF00  "Jitender"
0012FEF4  0000001E
0012FEF8  D6CD994D
0012FEFC  00000001
0012FF00  6574694A
0012FF04  7265646E
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  0000001E
0012FF2C  004054C0  return to struct
0012FF30  D6CD994D  return to D6CD99
0012FF34  00000001
0012FF38  01D7D682
0012FF3C  0012FF00  "Jitender"
0012FF40  0012FF88
0012FF44  004012B3  return to struct
```

Figure 12.9: *Before the call to printf*

After the call to the **printf** function, the stack is cleaned using the **ADD** instruction.

▼Line 72-85

```
; Line 23
mov eax, DWORD PTR _puser$[ebp]
mov ecx, DWORD PTR [eax+52]
push ecx
mov edx, DWORD PTR [eax+48]
push edx
mov eax, DWORD PTR _puser$[ebp]
```

```
mov ecx, DWORD PTR [eax+40]
push ecx
mov edx, DWORD PTR _puser$[ebp]
push edx

push OFFSET $SG5663
call _printf
add esp, 20       ; 00000014H
```

This is a comment which states that the ASM instructions preceding it will represent line 23 of the C/C++ code, which is:

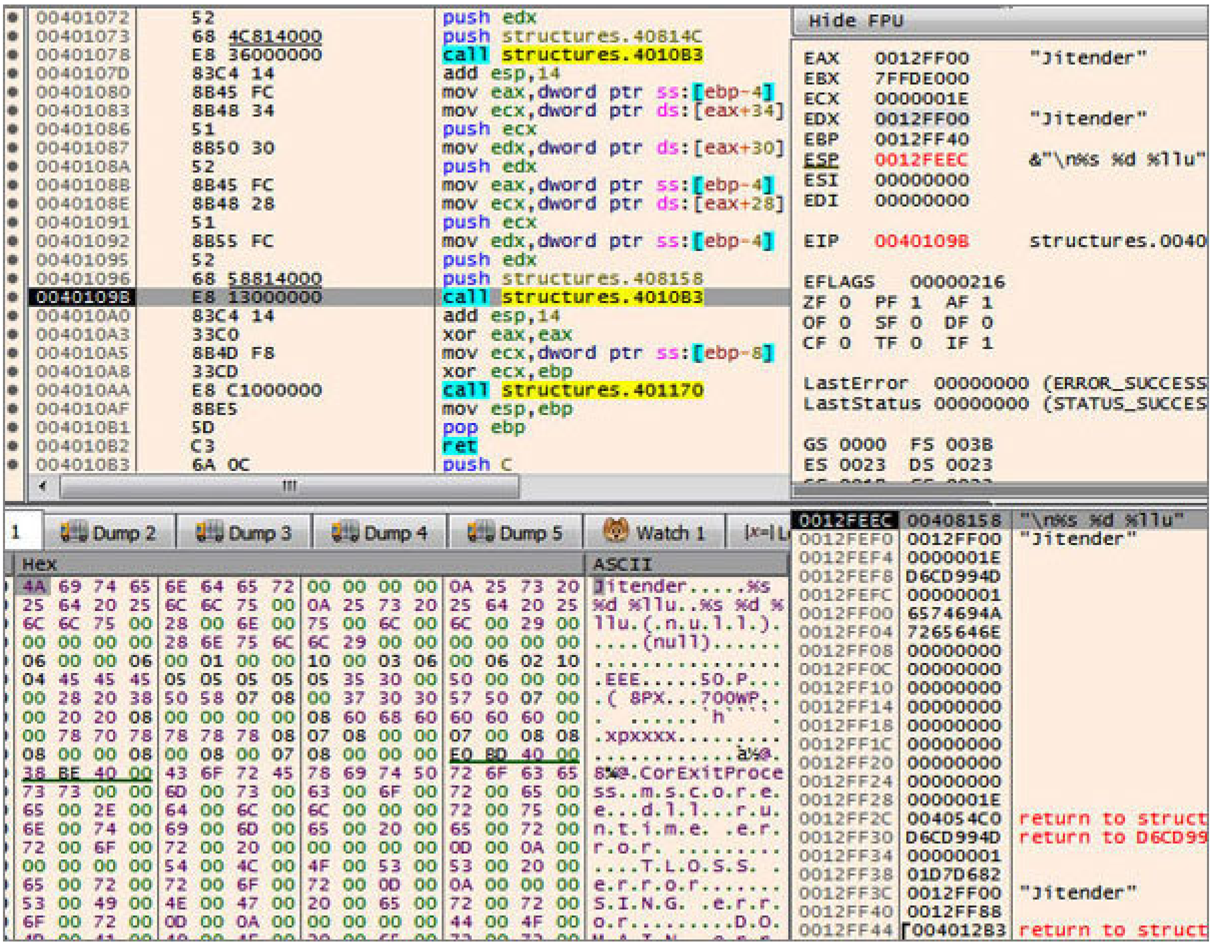
```
printf("\n%s %d %llu", puser->rgName, puser->iAge, puser-
>uMobile);
```

In the C/C++ code, we are accessing all the elements of structure using pointer to the structure and using **->** operator. So, in ASM, we can observe that in the first **MOV** instruction, pointer to structure is pushed onto the **EAX** register and then all the subsequent **MOV** instructions are referring to the elements using the **EAX** register, as **EAX** points to the start of the structure memory. In all the subsequent instructions, we are pushing the structure elements onto the stack using a combination of **MOV** and **PUSH** instructions.

All the **MOV** instructions are accessing variables using **EAX+offset** (where offset is the number of bytes. When added to it gives the location of the element value stored on the stack) and moving it to an available register. Next, the **PUSH** instruction pushes the structure element value onto the stack. Once all the elements are pushed onto the stack, a call to the **printf** function is made. The stack state before the call to the second **printf** function is as follows:

```
[ESP] 0012FEEC 00408158 "\n%s %d %llu", argument to printf() is pushed here
[ESP+0x04] 0012FEF0 0012FF00 pointer to rgName array is pushed here
[ESP+0x08] 0012FEF4 0000001E iAge value is pushed here

[ESP+0x0C] 0012FEF8 D6CD994D uMobile value is pushed here
[ESP+0x10] 0012FEFC 00000001 uMobile value is pushed here
[ESP+0x14] 0012FF00 6574694A 4 bytes of array, rgName in little-endian etiJ
[ESP+0x18] 0012FF04 7265646E 4 bytes of array, rgName in little-endian redn
[ESP+0x1C] 0012FF08 00000000 String terminator with NULL in remaining array
[ESP+0x20] 0012FF0C 00000000 NULL values in char array of size 40 bytes
[ESP+0x24] 0012FF10 00000000 NULL values in char array of size 40 bytes
[ESP+0x28] 0012FF14 00000000 NULL values in char array of size 40 bytes
[ESP+0x2C] 0012FF18 00000000 NULL values in char array of size 40 bytes
[ESP+0x30] 0012FF1C 00000000 NULL values in char array of size 40 bytes
[ESP+0x34] 0012FF20 00000000 NULL values in char array of size 40 bytes
[ESP+0x38] 0012FF24 00000000 rgName from 0x0012FF00 to 0x0012FF28, total 40 bytes
```

```
[ESP+0x3C] 0012FF28   0000001E   Second element of Structure, iAge is stored here
[ESP+0x40] 0012FF2C   004054C0   return to 0x004054C0 from 0x00405477

[ESP+0x44] 0012FF30   D6CD994D   4 bytes of uMobile stored here
[ESP+0x48] 0012FF34   00000001   remaining 4 bytes of uMobile stored here
[ESP+0x50] 0012FF38   01D7D682   XOR of Stack Cookie and EAX is stored here
[ESP+0x54] 0012FF3C   0012FF00   Pointer to structure is stored here, puser
[ESP+0x58] 0012FF40   0012FF88   [EBP]
[ESP+0x5C] 0012FF44   004012B3     return to 0x004012B3 from 0x00401000
```



```
00401072   52          push edx
00401073   68 4C814000 push structures.40814C
00401078   E8 36000000 call structures.4010B3
0040107D   83C4 14     add esp,14
00401080   8B45 FC     mov eax,dword ptr ss:[ebp-4]
00401083   8B48 34     mov ecx,dword ptr ds:[eax+34]
00401086   51          push ecx
00401087   8B50 30     mov edx,dword ptr ds:[eax+30]
0040108A   52          push edx
0040108B   8B45 FC     mov eax,dword ptr ss:[ebp-4]
0040108E   8B48 28     mov ecx,dword ptr ds:[eax+28]
00401091   51          push ecx
00401092   8B55 FC     mov edx,dword ptr ss:[ebp-4]
00401095   52          push edx
00401096   68 58814000 push structures.408158
0040109B   E8 13000000 call structures.4010B3
004010A0   83C4 14     add esp,14
004010A3   33C0        xor eax,eax
004010A5   8B4D F8     mov ecx,dword ptr ss:[ebp-8]
004010A8   33CD        xor ecx,ebp
004010AA   E8 C1000000 call structures.401170
004010AF   8BE5        mov esp,ebp
004010B1   5D          pop ebp
004010B2   C3          ret
004010B3   6A 0C       push C
```

```
EAX   0012FF00   "Jitender"
EBX   7FFDE000
ECX   0000001E
EDX   0012FF00   "Jitender"
EBP   0012FF40
ESP   0012FEEC   &"\n%s %d %llu"
ESI   00000000
EDI   00000000

EIP   0040109B   structures.0040

EFLAGS   00000216
ZF 0  PF 1  AF 1
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS
LastStatus 00000000 (STATUS_SUCCES

GS 0000  FS 003B
ES 0023  DS 0023
```

```
Hex                                                  ASCII
4A 69 74 65 6E 64 65 72 00 00 00 00 0A 25 73 20  Jitender.....%s
25 64 20 25 6C 6C 75 00 0A 25 73 20 25 64 20 25  %d %llu..%s %d %
6C 6C 75 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  llu.(.n.u.l.l.).
00 00 00 00 28 6E 75 6C 6C 29 00 00 00 00 00 00  ....(null)......
06 00 00 06 00 01 00 00 10 00 03 06 00 06 02 10  ................
04 45 45 45 05 05 05 05 05 35 30 00 50 00 00 00  .EEE.....50.P...
00 28 20 38 50 58 07 08 00 37 30 30 57 50 07 00  .( 8PX...700WP..
00 20 20 08 00 00 00 00 08 60 68 60 60 60 60 00  .  .......h    .
00 78 70 78 78 78 78 08 07 08 00 00 07 00 08 08  .xpxxxx.........
08 00 00 08 00 08 00 07 08 00 00 00 E0 BD 40 00  ............à½@.
38 BE 40 00 43 6F 72 45 78 69 74 50 72 6F 63 65  8¾@.CorExitProce
73 73 00 00 6D 00 73 00 63 00 6F 00 72 00 65 00  ss..m.s.c.o.r.e.
65 00 2E 00 64 00 6C 00 6C 00 00 00 72 00 75 00  e...d.l.l...r.u.
6E 00 74 00 69 00 6D 00 65 00 20 00 65 00 72 00  n.t.i.m.e. .e.r.
72 00 6F 00 72 00 20 00 00 00 00 00 0D 00 0A 00  r.o.r. .........
00 00 00 00 54 00 4C 00 4F 00 53 00 53 00 20 00  ....T.L.O.S.S. .
65 00 72 00 72 00 6F 00 72 00 0D 00 0A 00 00 00  e.r.r.o.r.......
53 00 49 00 4E 00 47 00 20 00 65 00 72 00 72 00  S.I.N.G. .e.r.r.
6F 00 72 00 0D 00 0A 00 00 00 00 00 44 00 4F 00  o.r.........D.O.
```

```
0012FEEC  00408158  "\n%s %d %llu"
0012FEF0  0012FF00  "Jitender"
0012FEF4  0000001E
0012FEF8  D6CD994D
0012FEFC  00000001
0012FF00  6574694A
0012FF04  7265646E
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  0000001E
0012FF2C  004054C0  return to struct
0012FF30  D6CD994D  return to D6CD99
0012FF34  00000001
0012FF38  01D7D682
0012FF3C  0012FF00  "Jitender"
0012FF40  0012FF88
0012FF44  004012B3  return to struct
```

Figure 12.10: *Before call to second printf*

After the call to the **printf** function, the stack is cleaned using the ADD instruction.

▼Line 86-87

```
; Line 25
xor eax, eax
```

This is a comment which states that the ASM instructions preceding it will represent line 25 of the C/C++ code, which is:

```
return 0;
```

In the ASM code, **EAX** is zeroed to return 0, as we discussed earlier that the return value of a function is stored in the **EAX** register.

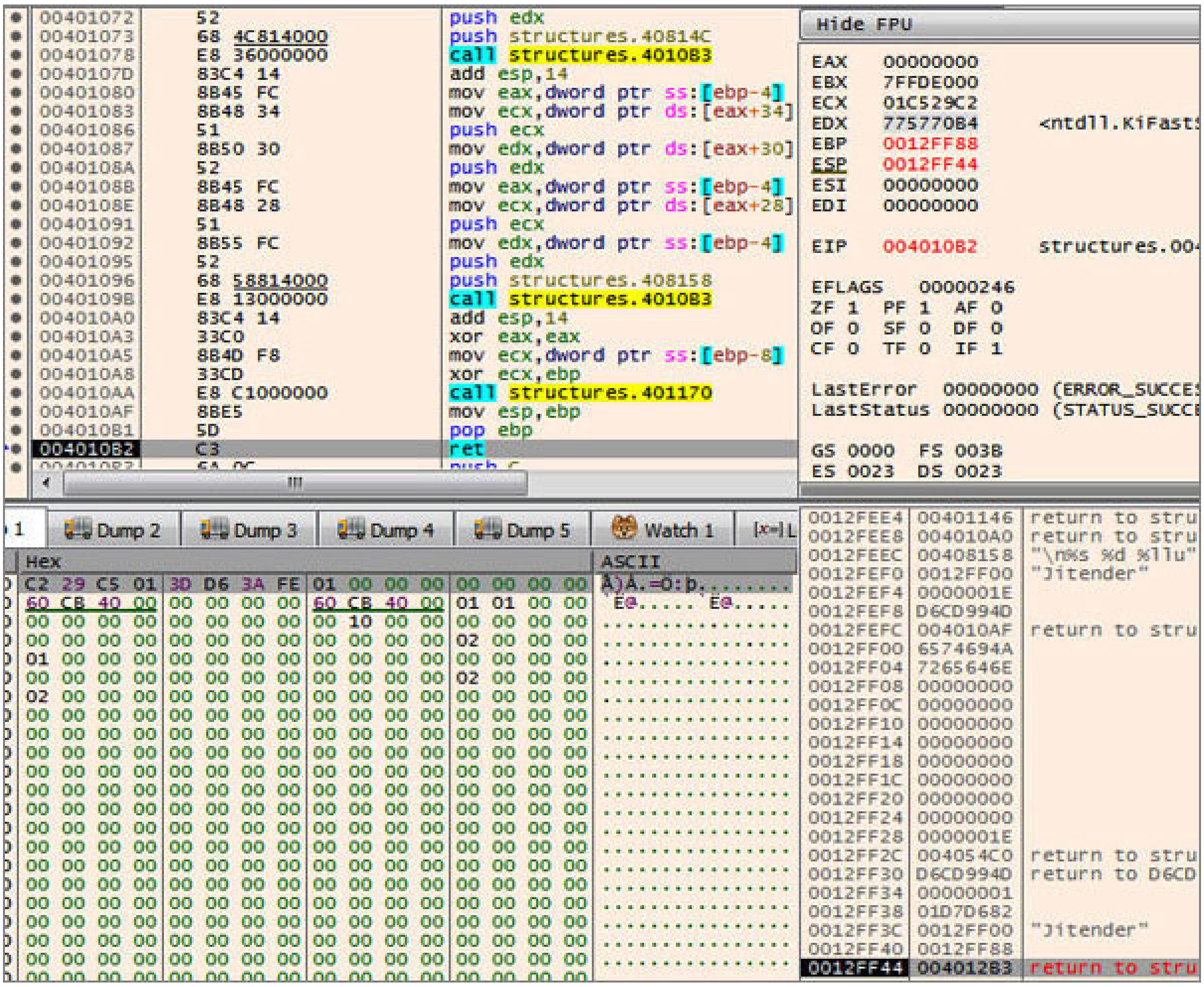
▼Line 88-97

```
; Line 26
mov ecx, DWORD PTR __$ArrayPad$[ebp]
xor ecx, ebp
call @__security_check_cookie@4
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

This ASM code that retrieves the stack cookie value from **[EBP-0x08]** and XOR with **EBP** to match the stack cookie value stored at On return from the **security_check_cookie** procedure, the function epilogue is called to return 0 and end the **main** procedure, TEXT segment, and code. The stack state before the return is as follows:

```
[ESP-0x44] 0012FF00   6574694A    Now JUNK
[ESP-0x40] 0012FF04   7265646E    Now JUNK
[ESP-0x3C] 0012FF08   00000000    Now JUNK
```

```
[ESP-0x38]  0012FF0C   00000000    Now  JUNK
[ESP-0x34]  0012FF10   00000000    Now  JUNK
[ESP-0x30]  0012FF14   00000000    Now  JUNK

[ESP-0x2C]  0012FF18   00000000    Now  JUNK
[ESP-0x28]  0012FF1C   00000000    Now  JUNK
[ESP-0x24]  0012FF20   00000000    Now  JUNK
[ESP-0x20]  0012FF24   00000000    Now  JUNK
[ESP-0x1C]  0012FF28   0000001E    Now  JUNK
[ESP-0x18]  0012FF2C   004054C0    return  to  0x004054C0  from
0x00405477
[ESP-0x14]  0012FF30   D6CD994D    Now  JUNK
[ESP-0x10]  0012FF34   00000001    Now  JUNK
[ESP-0x0C]  0012FF38   01D7D682    Now  JUNK
[ESP-0x08]  0012FF3C   0012FF00    Now  JUNK
[ESP-0x04]  0012FF40   0012FF88    Now  JUNK
[ESP]   0012FF44   004012B3    return  to  0x004012B3  from
0x00401000
```



```
00401072   52              push edx
00401073   68 4C814000     push structures.40814C
00401078   E8 36000000     call structures.401083
0040107D   83C4 14         add esp,14
00401080   8B45 FC         mov eax,dword ptr ss:[ebp-4]
00401083   8B48 34         mov ecx,dword ptr ds:[eax+34]
00401086   51              push ecx
00401087   8B50 30         mov edx,dword ptr ds:[eax+30]
0040108A   52              push edx
0040108B   8B45 FC         mov eax,dword ptr ss:[ebp-4]
0040108E   8B48 28         mov ecx,dword ptr ds:[eax+28]
00401091   51              push ecx
00401092   8B55 FC         mov edx,dword ptr ss:[ebp-4]
00401095   52              push edx
00401096   68 58814000     push structures.408158
0040109B   E8 13000000     call structures.401083
004010A0   83C4 14         add esp,14
004010A3   33C0            xor eax,eax
004010A5   8B4D F8         mov ecx,dword ptr ss:[ebp-8]
004010A8   33CD            xor ecx,ebp
004010AA   E8 C1000000     call structures.401170
004010AF   8BE5            mov esp,ebp
004010B1   5D              pop ebp
004010B2   C3              ret
```

```
EAX   00000000
EBX   7FFDE000
ECX   01C529C2
EDX   775770B4      <ntdll.KiFasts
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   004010B2      structures.00

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCES
LastStatus  00000000 (STATUS_SUCCE

GS 0000  FS 003B
ES 0023  DS 0023
```

```
Hex                                                  ASCII
C2 29 C5 01 3D D6 3A FE 01 00 00 00 00 00 00 00  Â)Å.=Ö:þ........
60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00   Ë@.....  Ë@.....
00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FEE4  00401146  return to stru
0012FEE8  004010A0  return to stru
0012FEEC  00408158  "\n%s %d %llu"
0012FEF0  0012FF00  "Jitender"
0012FEF4  0000001E
0012FEF8  D6CD994D
0012FEFC  004010AF  return to stru
0012FF00  6574694A
0012FF04  7265646E
0012FF08  00000000
0012FF0C  00000000
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  0000001E
0012FF2C  004054C0  return to stru
0012FF30  D6CD994D  return to D6CD
0012FF34  00000001
0012FF38  01D7D682
0012FF3C  0012FF00  "Jitender"
0012FF40  0012FF88
0012FF44  00401283  return to stru
```

Figure 12.11: Stack cleaned

Structure with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

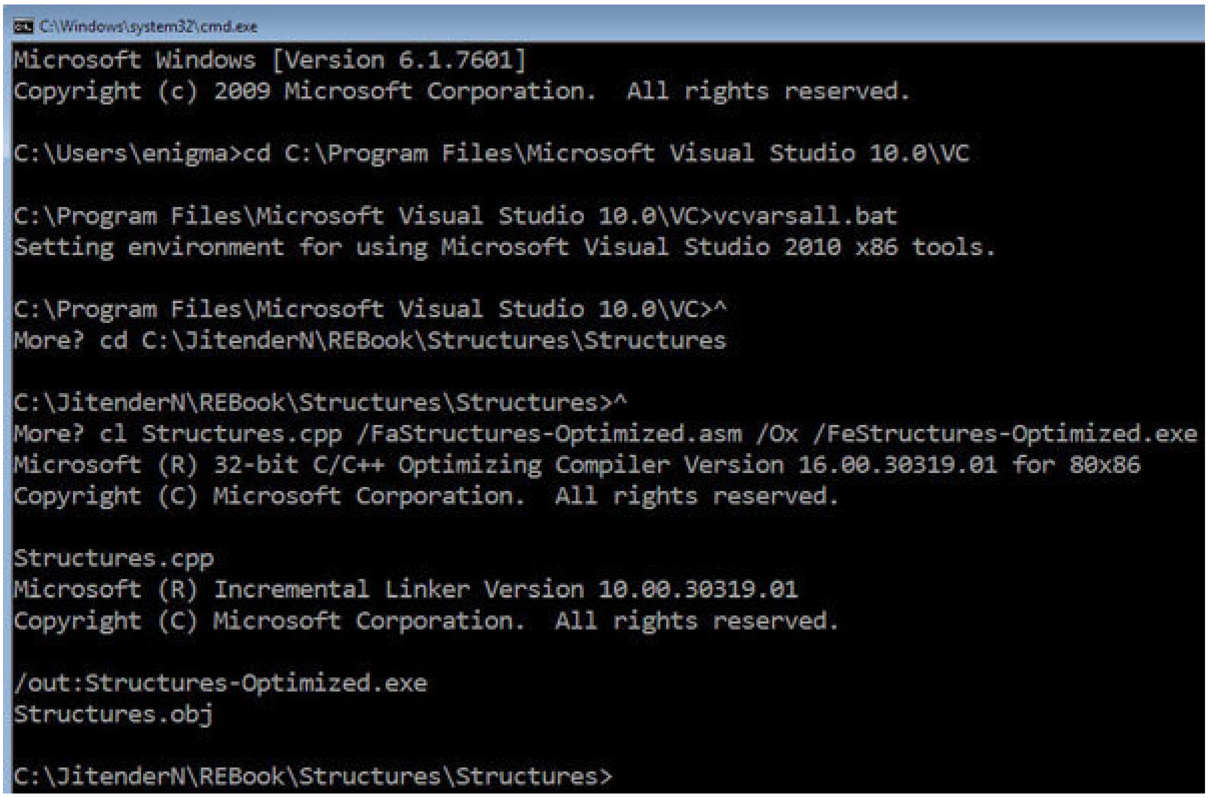
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:

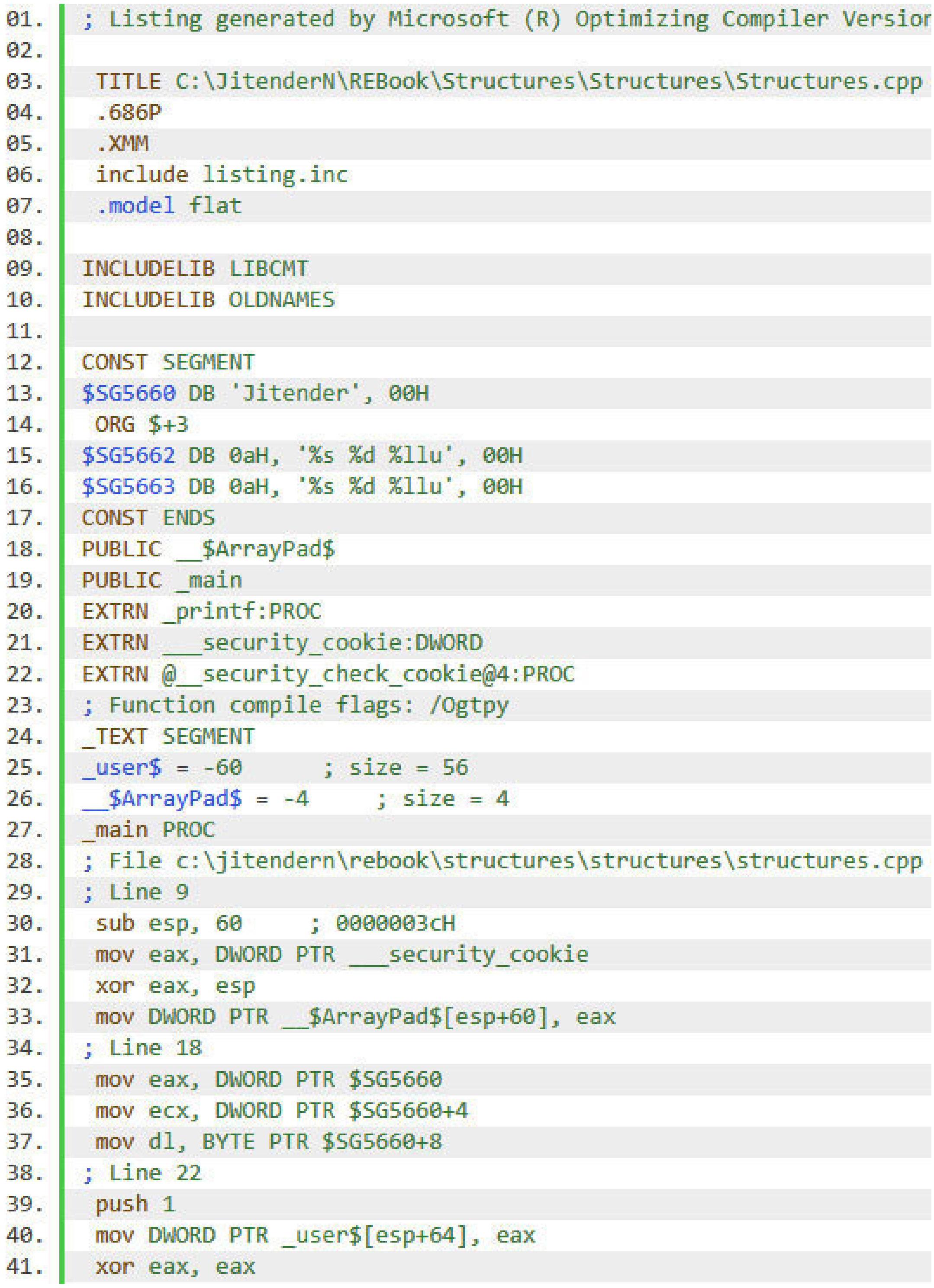


```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\Structures\Structures

C:\JitenderN\REBook\Structures\Structures>^
More? cl Structures.cpp /FaStructures-Optimized.asm /Ox /FeStructures-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

Structures.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:Structures-Optimized.exe
Structures.obj

C:\JitenderN\REBook\Structures\Structures>
```

Figure 12.12: *Structure with Optimization*

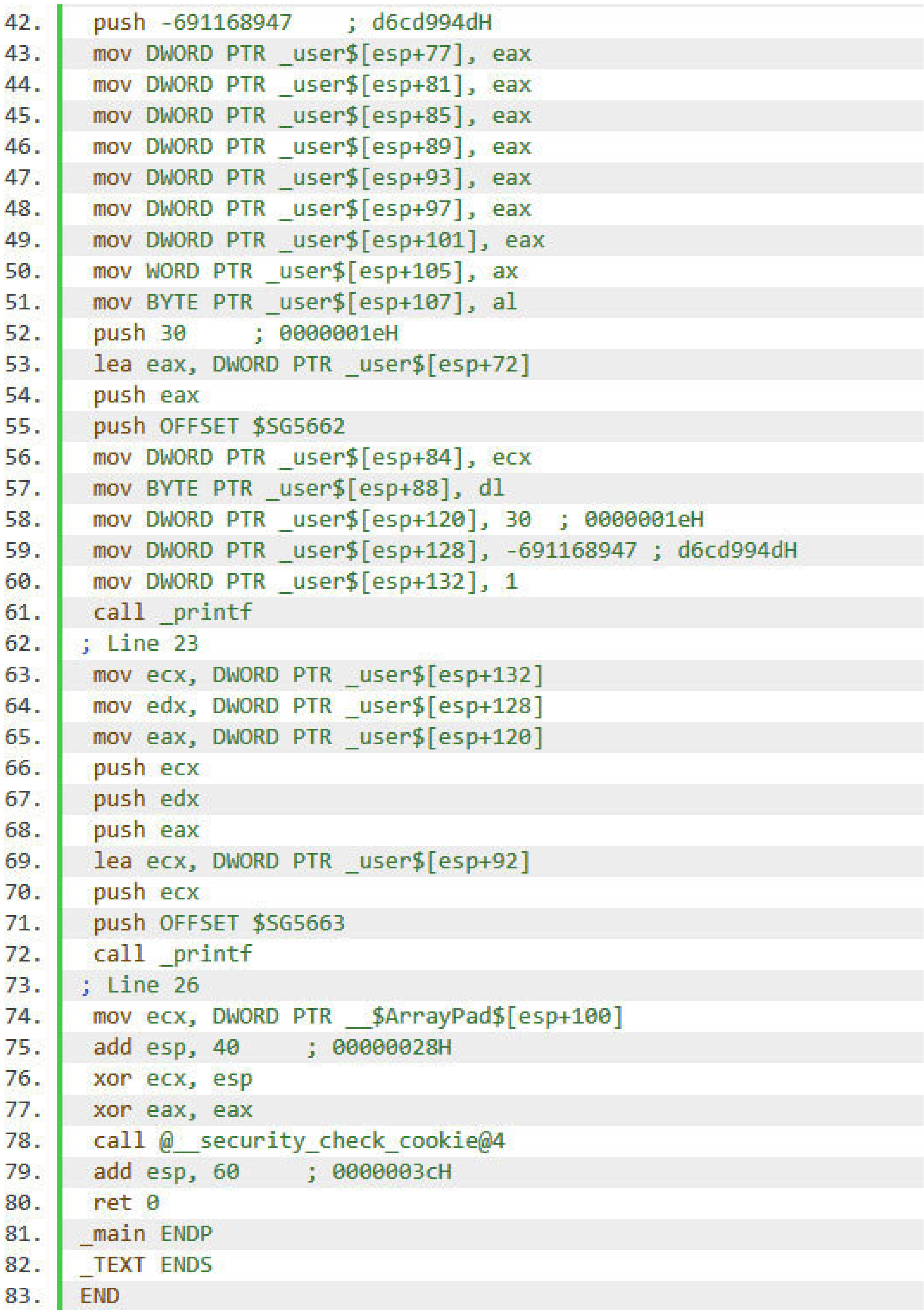
The compilation generates the EXE file and assembly code. Disable ASLR manually. To disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let’s move onto the generated assembly listing:



```
; Listing generated by Microsoft (R) Optimizing Compiler Versio

 TITLE C:\JitenderN\REBook\Structures\Structures\Structures.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG5660 DB 'Jitender', 00H
 ORG $+3
$SG5662 DB 0aH, '%s %d %llu', 00H
$SG5663 DB 0aH, '%s %d %llu', 00H
CONST ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_user$ = -60      ; size = 56
__$ArrayPad$ = -4     ; size = 4
_main PROC
; File c:\jitendern\rebook\structures\structures\structures.cpp
; Line 9
 sub esp, 60     ; 0000003cH
 mov eax, DWORD PTR ___security_cookie
 xor eax, esp
 mov DWORD PTR __$ArrayPad$[esp+60], eax
; Line 18
 mov eax, DWORD PTR $SG5660
 mov ecx, DWORD PTR $SG5660+4
 mov dl, BYTE PTR $SG5660+8
; Line 22
 push 1
 mov DWORD PTR _user$[esp+64], eax
 xor eax, eax
```

Figure 12.13: *Structures-Optimized.asm-Part-1*



```
42.   push -691168947     ; d6cd994dH
43.   mov DWORD PTR _user$[esp+77], eax
44.   mov DWORD PTR _user$[esp+81], eax
45.   mov DWORD PTR _user$[esp+85], eax
46.   mov DWORD PTR _user$[esp+89], eax
47.   mov DWORD PTR _user$[esp+93], eax
48.   mov DWORD PTR _user$[esp+97], eax
49.   mov DWORD PTR _user$[esp+101], eax
50.   mov WORD PTR _user$[esp+105], ax
51.   mov BYTE PTR _user$[esp+107], al
52.   push 30      ; 0000001eH
53.   lea eax, DWORD PTR _user$[esp+72]
54.   push eax
55.   push OFFSET $SG5662
56.   mov DWORD PTR _user$[esp+84], ecx
57.   mov BYTE PTR _user$[esp+88], dl
58.   mov DWORD PTR _user$[esp+120], 30  ; 0000001eH
59.   mov DWORD PTR _user$[esp+128], -691168947 ; d6cd994dH
60.   mov DWORD PTR _user$[esp+132], 1
61.   call _printf
62.  ; Line 23
63.   mov ecx, DWORD PTR _user$[esp+132]
64.   mov edx, DWORD PTR _user$[esp+128]
65.   mov eax, DWORD PTR _user$[esp+120]
66.   push ecx
67.   push edx
68.   push eax
69.   lea ecx, DWORD PTR _user$[esp+92]
70.   push ecx
71.   push OFFSET $SG5663
72.   call _printf
73.  ; Line 26
74.   mov ecx, DWORD PTR __$ArrayPad$[esp+100]
75.   add esp, 40      ; 00000028H
76.   xor ecx, esp
77.   xor eax, eax
78.   call @__security_check_cookie@4
79.   add esp, 60      ; 0000003cH
80.   ret 0
81.  _main ENDP
82.  _TEXT ENDS
83.  END
```

Figure 12.14: *Structures-Optimized.asm-Part-2*

Let's analyze the ASM code:

▼Line 29-33

```
; Line 9
sub esp, 60       ; 0000003cH
mov eax, DWORD PTR ___security_cookie
xor eax, esp
mov DWORD PTR __$ArrayPad$[esp+60], eax
```

In the ASM code, 60 bytes are subtracted from **ESP** to create room for the variables on the stack. The **MOV** instruction moves the stack cookie stored at **0x0040B000** to where it is XOR'ed with **ESP** and the result is moved to the stack at **[ESP+0x38].** As instruction:

```
mov DWORD PTR __$ArrayPad$[esp+60], eax
```

When this instruction is viewed in x32dbg it will be shown as below:

```
mov dword ptr ss:[esp+0x38], eax
```

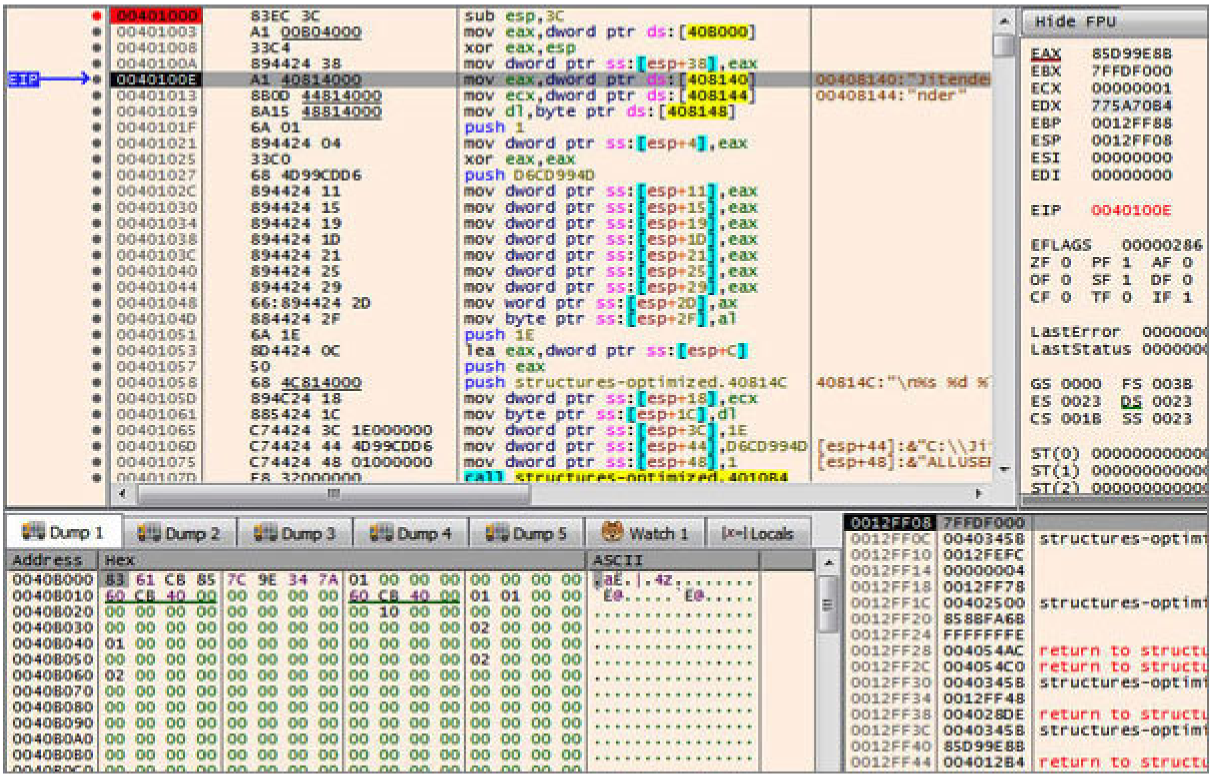
The stack state after this instruction is as follows:

```
[ESP]   0012FF08   7FFDF000   JUNK
[ESP+0x4] 0012FF0C   0040345B   JUNK
[ESP+0x8] 0012FF10   0012FEFC   JUNK
[ESP+0xC] 0012FF14   00000004   JUNK
[ESP+0x10] 0012FF18   0012FF78   JUNK
[ESP+0x14] 0012FF1C   00402500   JUNK
[ESP+0x18] 0012FF20   858BFA6B   JUNK
```

```
[ESP+0x1C] 0012FF24   FFFFFFFE   JUNK
[ESP+0x20] 0012FF28   004054AC   JUNK
[ESP+0x24] 0012FF2C   004054C0   JUNK
[ESP+0x28] 0012FF30   0040345B   JUNK
[ESP+0x2C] 0012FF34   0012FF48   JUNK
[ESP+0x30] 0012FF38   004028DE   JUNK
[ESP+0x34] 0012FF3C   0040345B   JUNK
```

[ESP+0x38] 0012FF40 85D99E8B XOR of Stack Cookie and ESP is stored here

[ESP+0x3C] 0012FF44 004012B4 return to structures-optimized.004012B4



```
00401000   83EC 3C             sub esp,3C
00401003   A1 00804000         mov eax,dword ptr ds:[408000]
00401008   33C4                xor eax,esp
0040100A   894424 38           mov dword ptr ss:[esp+38],eax
0040100E   A1 40814000         mov eax,dword ptr ds:[408140]        00408140:"Jitende
00401013   8B0D 44814000       mov ecx,dword ptr ds:[408144]        00408144:"nder"
00401019   8A15 48814000       mov dl,byte ptr ds:[408148]
0040101F   6A 01               push 1
00401021   894424 04           mov dword ptr ss:[esp+4],eax
00401025   33C0                xor eax,eax
00401027   68 4D99CDD6         push D6CD994D
0040102C   894424 11           mov dword ptr ss:[esp+11],eax
00401030   894424 15           mov dword ptr ss:[esp+15],eax
00401034   894424 19           mov dword ptr ss:[esp+19],eax
00401038   894424 1D           mov dword ptr ss:[esp+1D],eax
0040103C   894424 21           mov dword ptr ss:[esp+21],eax
00401040   894424 25           mov dword ptr ss:[esp+25],eax
00401044   894424 29           mov dword ptr ss:[esp+29],eax
00401048   66:894424 2D        mov word ptr ss:[esp+2D],ax
0040104D   884424 2F           mov byte ptr ss:[esp+2F],al
00401051   6A 1E               push 1E
00401053   8D4424 0C           lea eax,dword ptr ss:[esp+C]
00401057   50                  push eax
00401058   68 4C814000         push structures-optimized.40814C     40814C:"\n%s %d %
0040105D   894C24 18           mov dword ptr ss:[esp+18],ecx
00401061   885424 1C           mov byte ptr ss:[esp+1C],dl
00401065   C74424 3C 1E000000  mov dword ptr ss:[esp+3C],1E
0040106D   C74424 44 4D99CDD6  mov dword ptr ss:[esp+44],D6CD994D  [esp+44]:&"C:\\Ji
00401075   C74424 48 01000000  mov dword ptr ss:[esp+48],1         [esp+48]:&"ALLUSEF
0040107D   E8 32000000         call structures-optimized.401084
```

```
EAX   85D99E8B
EBX   7FFDF000
ECX   00000001
EDX   775A70B4
EBP   0012FF88
ESP   0012FF08
ESI   00000000
EDI   00000000

EIP   0040100E

EFLAGS   00000286
ZF 0   PF 1   AF 0
OF 0   SF 1   DF 0
CF 0   TF 0   IF 1

LastError   000000
LastStatus  000000

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 000000000000
ST(1) 000000000000
ST(2) 000000000000
```

```
Address   Hex                                               ASCII
0040B000  83 61 CB 85 7C 9E 34 7A 01 00 00 00 00 00 00 00  .aË.|.4z........
0040B010  60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00  `Ë@.....`Ë@.....
0040B020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00  ................
0040B030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00  ................
0040B060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0040B0B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

```
0012FF08  7FFDF000
0012FF0C  0040345B  structures-optimi
0012FF10  0012FEFC
0012FF14  00000004
0012FF18  0012FF78
0012FF1C  00402500  structures-optimi
0012FF20  858BFA6B
0012FF24  FFFFFFFE
0012FF28  004054AC  return to structu
0012FF2C  004054C0  return to structu
0012FF30  0040345B  structures-optimi
0012FF34  0012FF48
0012FF38  004028DE  return to structu
0012FF3C  0040345B  structures-optimi
0012FF40  85D99E8B
0012FF44  004012B4  return to structu
```

Figure 12.15: Stack cookie

▼Line 34-37

```
; Line 18
mov eax, DWORD PTR $SG5660
mov ecx, DWORD PTR $SG5660+4
mov dl, BYTE PTR $SG5660+8
```

These instructions move the first 4 bytes of the **rgName** array into the **EAX** register in the little-endian format and the next 4 bytes in the **ECX** register. The null terminated character is moved to the **DL** register.

▼Line 38-61

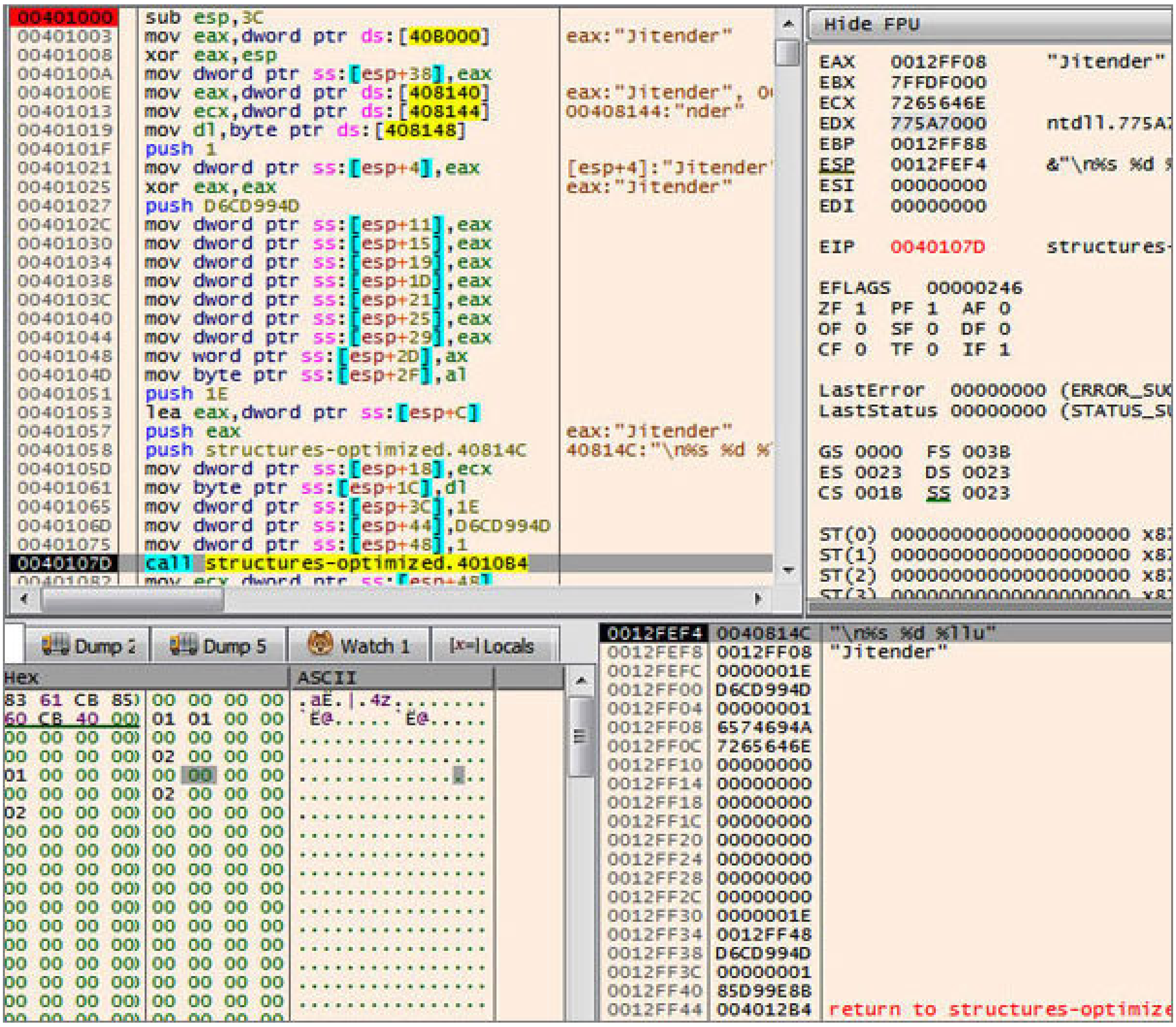
```
; Line 22
push 1


mov DWORD PTR _user$[esp+64], eax
xor eax, eax
push -691168947      ; d6cd994dH
mov DWORD PTR _user$[esp+77], eax
mov DWORD PTR _user$[esp+81], eax
mov DWORD PTR _user$[esp+85], eax
mov DWORD PTR _user$[esp+89], eax
mov DWORD PTR _user$[esp+93], eax
mov DWORD PTR _user$[esp+97], eax
mov DWORD PTR _user$[esp+101], eax
mov WORD PTR _user$[esp+105], ax
mov BYTE PTR _user$[esp+107], al
push 30        ; 0000001eH
lea eax, DWORD PTR _user$[esp+72]
push eax
push OFFSET $SG5662
```

```
mov DWORD PTR _user$[esp+84], ecx
mov BYTE PTR _user$[esp+88], dl
mov DWORD PTR _user$[esp+120], 30  ; 0000001eH
mov DWORD PTR _user$[esp+128], -691168947 ; d6cd994dH
mov DWORD PTR _user$[esp+132], 1
call _printf
```

In the non-optimized section, we saw how **EBP** is referred to access the structure elements on the stack. In the preceding ASM code, the stack is similarly filled with the elements of structure by referring to the stack memory with Also, for the first **printf** function call, the arguments are pushed onto the stack. The stack state before the call to **printf** will be:

[ESP] 0012FEF4 0040814C "\n%s %d %llu", argument to printf() is pushed here
[ESP+0x4] 0012FEF8 0012FF08 pointer to rgName array is pushed here
[ESP+0x8] 0012FEFC 0000001E iAge value is pushed here
[ESP+0xC] 0012FF00 D6CD994D uMobile value is pushed here
[ESP+0x10] 0012FF04 00000001 uMobile value is pushed here
[ESP+0x14] 0012FF08 6574694A 4 bytes of array, rgName in little-endian etiJ
[ESP+0x18] 0012FF0C 7265646E 4 bytes of array, rgName in little-endian redn
[ESP+0x1C] 0012FF10 00000000 String terminator with NULL values in array
[ESP+0x20] 0012FF14 00000000 NULL values in char array of size 40 bytes

```
[ESP+0x24] 0012FF18  00000000  NULL values in char array of size 40 bytes
[ESP+0x28] 0012FF1C  00000000  NULL values in char array of size 40 bytes
[ESP+0x2C] 0012FF20  00000000  NULL values in char array of size 40 bytes
[ESP+0x30] 0012FF24  00000000  NULL values in char array of size 40 bytes
[ESP+0x34] 0012FF28  00000000  NULL values in char array of size 40 bytes
[ESP+0x38] 0012FF2C  00000000  rgName from 0x0012FF08 to 0x0012FF30, total 40 bytes

[ESP+0x3C] 0012FF30  0000001E  Second element of Structure, iAge is stored here
[ESP+0x40] 0012FF34  0012FF48  Pointer to structure is stored here, puser
[ESP+0x44] 0012FF38  D6CD994D  4 bytes of uMobile stored here
[ESP+0x48] 0012FF3C  00000001  remaining 4 bytes of uMobile stored here
[ESP+0x4C] 0012FF40  85D99E8B  XOR of Stack Cookie and EAX is stored here
[ESP+0x50] 0012FF44  004012B4  return to structures-optimized.004012B4
```



```
00401000  sub esp,3C
00401003  mov eax,dword ptr ds:[408000]             eax:"Jitender"
00401008  xor eax,esp
0040100A  mov dword ptr ss:[esp+38],eax
0040100E  mov eax,dword ptr ds:[408140]             eax:"Jitender", 0
00401013  mov ecx,dword ptr ds:[408144]             00408144:"nder"
00401019  mov dl,byte ptr ds:[408148]
0040101F  push 1
00401021  mov dword ptr ss:[esp+4],eax              [esp+4]:"Jitender
00401025  xor eax,eax                               eax:"Jitender"
00401027  push D6CD994D
0040102C  mov dword ptr ss:[esp+11],eax
00401030  mov dword ptr ss:[esp+15],eax
00401034  mov dword ptr ss:[esp+19],eax
00401038  mov dword ptr ss:[esp+1D],eax
0040103C  mov dword ptr ss:[esp+21],eax
00401040  mov dword ptr ss:[esp+25],eax
00401044  mov dword ptr ss:[esp+29],eax
00401048  mov word ptr ss:[esp+2D],ax
0040104D  mov byte ptr ss:[esp+2F],al
00401051  push 1E
00401053  lea eax,dword ptr ss:[esp+C]
00401057  push eax                                  eax:"Jitender"
00401058  push structures-optimized.40814C          40814C:"\n%s %d %
0040105D  mov dword ptr ss:[esp+18],ecx
00401061  mov byte ptr ss:[esp+1C],dl
00401065  mov dword ptr ss:[esp+3C],1E
0040106D  mov dword ptr ss:[esp+44],D6CD994D
00401075  mov dword ptr ss:[esp+48],1
0040107D  call structures-optimized.401084
```

```
EAX   0012FF08     "Jitender"
EBX   7FFDF000
ECX   7265646E
EDX   775A7000     ntdll.775A
EBP   0012FF88
ESP   0012FEF4     &"\n%s %d %
ESI   00000000
EDI   00000000

EIP   0040107D     structures-

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SU
LastStatus  00000000 (STATUS_S

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x8
ST(1) 00000000000000000000 x8
ST(2) 00000000000000000000 x8
```

```
Hex                                      ASCII
83 61 CB 85  00 00 00 00                 .aÈ.|.4z........
60 CB 40 00  01 01 00 00                 È@......È@.....
00 00 00 00  00 00 00 00                 ................
00 00 00 00  02 00 00 00                 ................
01 00 00 00  00 00 00 00                 ................
00 00 00 00  02 00 00 00                 ................
02 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
00 00 00 00  00 00 00 00                 ................
```

```
0012FEF4  0040814C  "\n%s %d %llu"
0012FEF8  0012FF08  "Jitender"
0012FEFC  0000001E
0012FF00  D6CD994D
0012FF04  00000001
0012FF08  6574694A
0012FF0C  7265646E
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  00000000
0012FF2C  00000000
0012FF30  0000001E
0012FF34  0012FF48
0012FF38  D6CD994D
0012FF3C  00000001
0012FF40  85D99E8B
0012FF44  004012B4  return to structures-optimize
```

Figure 12.16: *The stack state before the call to printf*

▼Line 62-72

```
; Line 23
mov ecx, DWORD PTR _user$[esp+132]
mov edx, DWORD PTR _user$[esp+128]
mov eax, DWORD PTR _user$[esp+120]
push ecx
push edx
push eax
```

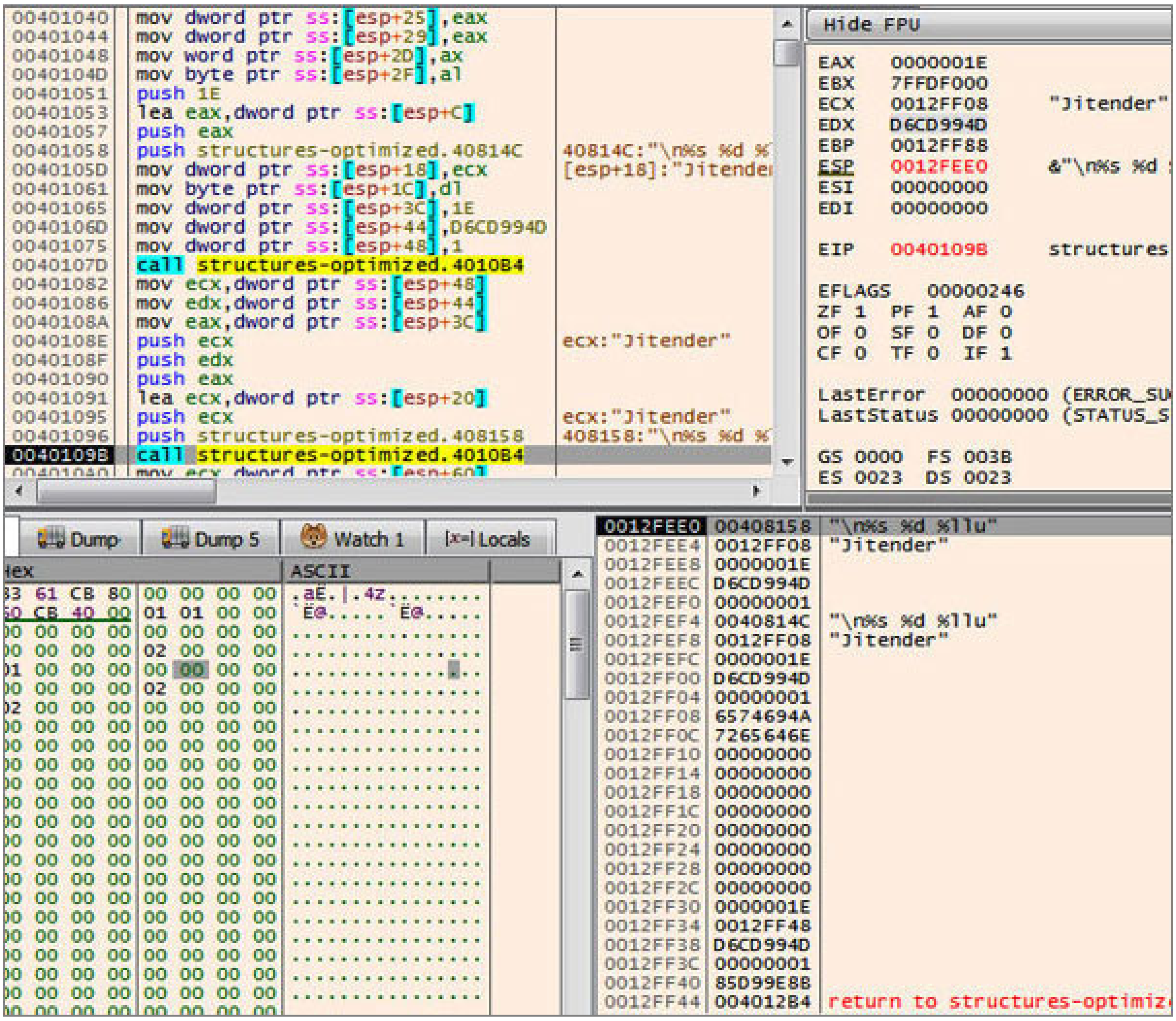
```
lea ecx, DWORD PTR _user$[esp+92]
push ecx
push OFFSET $SG5663
call _printf
```

The first three **MOV** and **PUSH** instructions push the **uMobile** and **iAge** values on the stack. **Load Effective Address** moves the pointer to structure on the **ECX** register and then to the stack. Now, we have all three arguments to **printf** on the stack. We can call the **printf** function. The stack state before the call to the second **printf** will be:

```
[ESP] 0012FEE0 00408158 "\n%s %d %llu", arg to 2nd printf() pushed here
[ESP+0x4] 0012FEE4 0012FF08 pointer to rgName array is pushed here
[ESP+0x8] 0012FEE8 0000001E iAge value is pushed here
[ESP+0x0C] 0012FEEC D6CD994D uMobile value is pushed here
[ESP+0x10] 0012FEF0 00000001 uMobile value is pushed here
[ESP+0x14] 0012FEF4 0040814C "\n%s %d %llu", arg to 1st printf() pushed here

[ESP+0x18] 0012FEF8 0012FF08 pointer to rgName array is pushed here
[ESP+0x1C] 0012FEFC 0000001E iAge value is pushed here
[ESP+0x20] 0012FF00 D6CD994D uMobile value is pushed here
[ESP+0x24] 0012FF04 00000001 uMobile value is pushed here
[ESP+0x28] 0012FF08 6574694A 4 bytes of array, rgName in little-endian etiJ
[ESP+0x2C] 0012FF0C 7265646E 4 bytes of array, rgName in little-endian redn
```

[ESP+0x30] 0012FF10 00000000 String terminator with NULL in remaining array
[ESP+0x34] 0012FF14 00000000 NULL values in char array of size 40 bytes
[ESP+0x38] 0012FF18 00000000 NULL values in char array of size 40 bytes
[ESP+0x3C] 0012FF1C 00000000 NULL values in char array of size 40 bytes
[ESP+0x40] 0012FF20 00000000 NULL values in char array of size 40 bytes
[ESP+0x44] 0012FF24 00000000 NULL values in char array of size 40 bytes
[ESP+0x48] 0012FF28 00000000 NULL values in char array of size 40 bytes
[ESP+0x4C] 0012FF2C 00000000 rgName from 0x0012FF08 to 0x0012FF30, total 40 bytes
[ESP+0x50] 0012FF30 0000001E Second element of Structure, iAge is stored here

[ESP+0x54] 0012FF34 0012FF48 Pointer to structure is stored here, puser
[ESP+0x58] 0012FF38 D6CD994D 4 bytes of uMobile stored here
[ESP+0x6C] 0012FF3C 00000001 remaining 4 bytes of uMobile stored here
[ESP+0x64] 0012FF40 85D99E8B XOR of Stack Cookie and EAX is stored here
[ESP+0x68] 0012FF44 004012B4 return to structures-optimized.004012B4



```
00401040   mov dword ptr ss:[esp+25],eax
00401044   mov dword ptr ss:[esp+29],eax
00401048   mov word ptr ss:[esp+2D],ax
0040104D   mov byte ptr ss:[esp+2F],al
00401051   push 1E
00401053   lea eax,dword ptr ss:[esp+C]
00401057   push eax
00401058   push structures-optimized.40814C      40814C:"\n%s %d %
0040105D   mov dword ptr ss:[esp+18],ecx         [esp+18]:"Jitende
00401061   mov byte ptr ss:[esp+1C],dl
00401065   mov dword ptr ss:[esp+3C],1E
0040106D   mov dword ptr ss:[esp+44],D6CD994D
00401075   mov dword ptr ss:[esp+48],1
0040107D   call structures-optimized.4010B4
00401082   mov ecx,dword ptr ss:[esp+48]
00401086   mov edx,dword ptr ss:[esp+44]
0040108A   mov eax,dword ptr ss:[esp+3C]
0040108E   push ecx                              ecx:"Jitender"
0040108F   push edx
00401090   push eax
00401091   lea ecx,dword ptr ss:[esp+20]
00401095   push ecx                              ecx:"Jitender"
00401096   push structures-optimized.408158      408158:"\n%s %d %
0040109B   call structures-optimized.4010B4
```

```
Hide FPU

EAX   0000001E
EBX   7FFDF000
ECX   0012FF08      "Jitender"
EDX   D6CD994D
EBP   0012FF88
ESP   0012FEE0      &"\n%s %d
ESI   00000000
EDI   00000000

EIP   0040109B      structures

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SU
LastStatus 00000000 (STATUS_S

GS 0000  FS 003B
ES 0023  DS 0023
```

```
Hex                                  ASCII
3 61 CB 80  00 00 00 00   .aE.|.4z........
0 CB 40 00  01 01 00 00    E@.....  E@.....
0 00 00 00  00 00 00 00   ................
0 00 00 00  02 00 00 00   ................
1 00 00 00  00 00 00 00   ................
0 00 00 00  02 00 00 00   ................
2 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
0 00 00 00  00 00 00 00   ................
```

```
0012FEE0  00408158  "\n%s %d %llu"
0012FEE4  0012FF08  "Jitender"
0012FEE8  0000001E
0012FEEC  D6CD994D
0012FEF0  00000001
0012FEF4  0040814C  "\n%s %d %llu"
0012FEF8  0012FF08  "Jitender"
0012FEFC  0000001E
0012FF00  D6CD994D
0012FF04  00000001
0012FF08  6574694A
0012FF0C  7265646E
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  00000000
0012FF2C  00000000
0012FF30  0000001E
0012FF34  0012FF48
0012FF38  D6CD994D
0012FF3C  00000001
0012FF40  85D99E8B
0012FF44  004012B4  return to structures-optimiz
```

Figure 12.17: *Stack state before call to second printf*

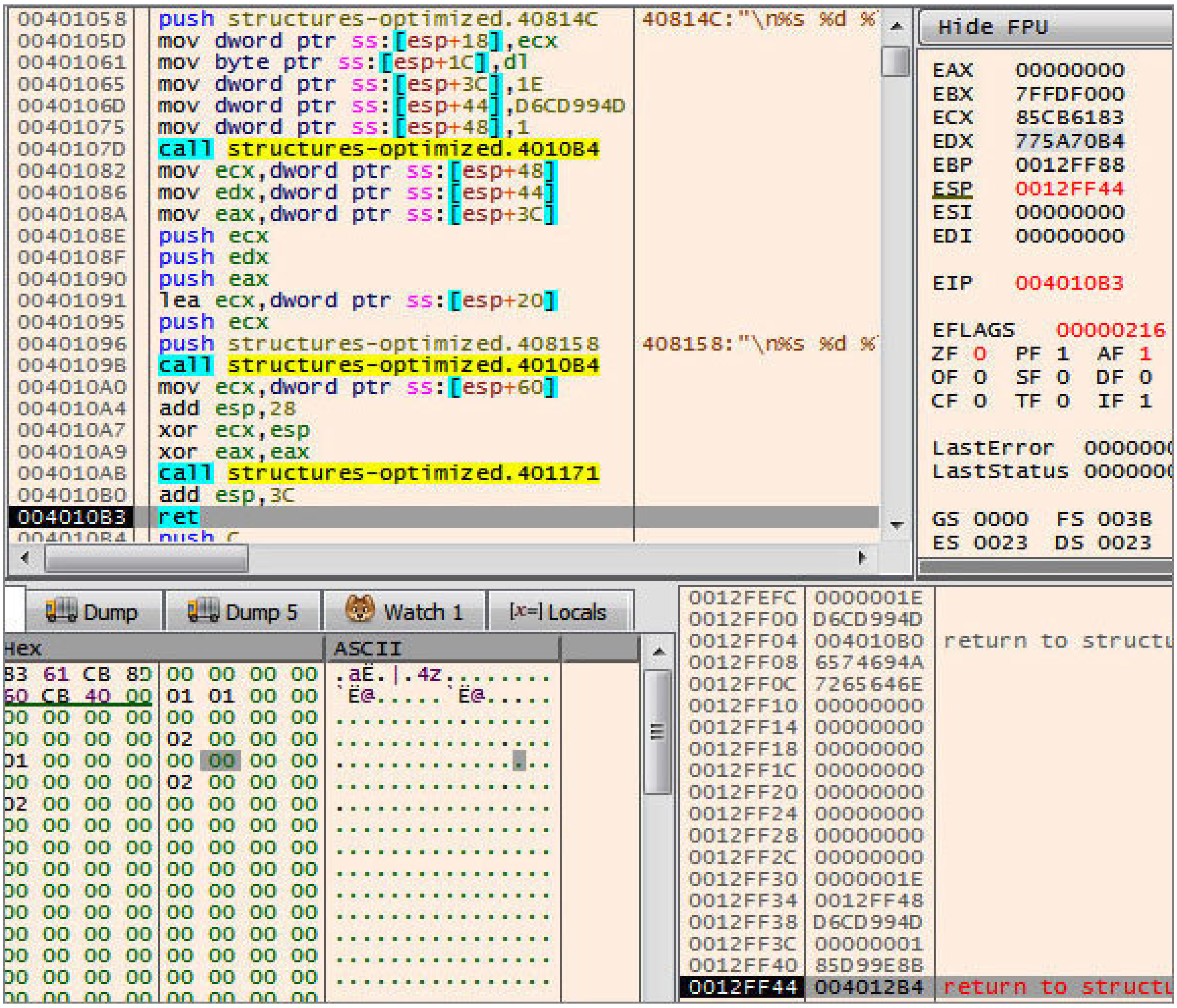
▼Line 73-83

```
; Line 26
mov ecx, DWORD PTR __$ArrayPad$[esp+100]


add esp, 40        ; 00000028H
xor ecx, esp
xor eax, eax
call @__security_check_cookie@4
add esp, 60        ; 0000003cH
```

```
ret 0
_main ENDP
_TEXT ENDS
END
```

The rest of the ASM code is the same, where the stack is cleaned using the ADD instruction to match the stack cookie for the buffer overflow. Finally, after a call to the security_check_cookie procedure, the stack is again cleaned to return 0 to end the main function, TEXT segment, and code. The stack state x32dbg will be as follows:

```
[ESP-0x40] 0012FF08  6574694A   Now JUNK
[ESP-0x38] 0012FF0C  7265646E   Now JUNK
[ESP-0x34] 0012FF10  00000000   Now JUNK
[ESP-0x30] 0012FF14  00000000   Now JUNK
[ESP-0x2C] 0012FF18  00000000   Now JUNK
[ESP-0x28] 0012FF1C  00000000   Now JUNK
[ESP-0x24] 0012FF20  00000000   Now JUNK
[ESP-0x20] 0012FF24  00000000   Now JUNK
[ESP-0x1C] 0012FF28  00000000   Now JUNK
[ESP-0x18] 0012FF2C  00000000   Now JUNK
[ESP-0x14] 0012FF30  0000001E   Now JUNK
[ESP-0x10] 0012FF34  0012FF48   Now JUNK
[ESP-0x0C] 0012FF38  D6CD994D   Now JUNK
[ESP-0x08] 0012FF3C  00000001   Now JUNK

[ESP-0x04] 0012FF40  85D99E8B   Now JUNK
[ESP]  0012FF44  004012B4   return to structures-
optimized.004012B4
```



```
00401058  push structures-optimized.40814C        40814C:"\n%s %d %
0040105D  mov dword ptr ss:[esp+18],ecx
00401061  mov byte ptr ss:[esp+1C],dl
00401065  mov dword ptr ss:[esp+3C],1E
0040106D  mov dword ptr ss:[esp+44],D6CD994D
00401075  mov dword ptr ss:[esp+48],1
0040107D  call structures-optimized.4010B4
00401082  mov ecx,dword ptr ss:[esp+48]
00401086  mov edx,dword ptr ss:[esp+44]
0040108A  mov eax,dword ptr ss:[esp+3C]
0040108E  push ecx
0040108F  push edx
00401090  push eax
00401091  lea ecx,dword ptr ss:[esp+20]
00401095  push ecx
00401096  push structures-optimized.408158        408158:"\n%s %d %
0040109B  call structures-optimized.4010B4
004010A0  mov ecx,dword ptr ss:[esp+60]
004010A4  add esp,28
004010A7  xor ecx,esp
004010A9  xor eax,eax
004010AB  call structures-optimized.401171
004010B0  add esp,3C
004010B3  ret
```

```
EAX     00000000
EBX     7FFDF000
ECX     85CB6183
EDX     775A70B4
EBP     0012FF88
ESP     0012FF44
ESI     00000000
EDI     00000000

EIP     004010B3

EFLAGS    00000216
ZF 0   PF 1   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   000000
LastStatus  000000

GS 0000   FS 003B
ES 0023   DS 0023
```

```
B3 61 CB 8D 00 00 00 00   .aË.|.4z........
60 CB 40 00 01 01 00 00   `Ë@.....`Ë@.....
00 00 00 00 00 00 00 00   ................
00 00 00 00 02 00 00 00   ................
01 00 00 00 00 00 00 00   ................
00 00 00 00 02 00 00 00   ................
02 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
00 00 00 00 00 00 00 00   ................
```

```
0012FEFC  0000001E
0012FF00  D6CD994D
0012FF04  004010B0  return to structu
0012FF08  6574694A
0012FF0C  7265646E
0012FF10  00000000
0012FF14  00000000
0012FF18  00000000
0012FF1C  00000000
0012FF20  00000000
0012FF24  00000000
0012FF28  00000000
0012FF2C  00000000
0012FF30  0000001E
0012FF34  0012FF48
0012FF38  D6CD994D
0012FF3C  00000001
0012FF40  85D99E8B
0012FF44  004012B4  return to structu
```

Figure 12.18: *The stack is cleaned*

Conclusion

In this chapter, we learned pointers to structures with respect to reverse engineering. We discussed about structures code pattern in disassembled code and how structures are stored in memory. The main point to understand is that, pointer to the structure points to the first element of structure as structures are always stored in contiguous memory locations. We also covered structures program with optimized and non-optimized code.

CHAPTER 13

Scanf Program Pattern in Reverse Engineering

Imagine you downloaded some hacking software from the internet. On running this software, it asks you to enter the password. The password is not mentioned anywhere on the site from where you downloaded this software. Using reverse engineering, we can break the password. Breaking a password is unethical and against laws until and unless we are permitted to do so. Now you must be thinking how one can break a password. Breaking a password is dependent on the quality of the software code. If the code is written with security in mind, chances are that a stronger mechanism must have been used.

In this chapter, we will discuss the reverse engineering part of the mechanism used by a software developer to capture user input. We will be talking about **scanf** function, which is used to capture the user input from the console.

Structure

In this chapter, we will cover the following topics:

Function **scanf** with Integers

Function scanf without Optimization

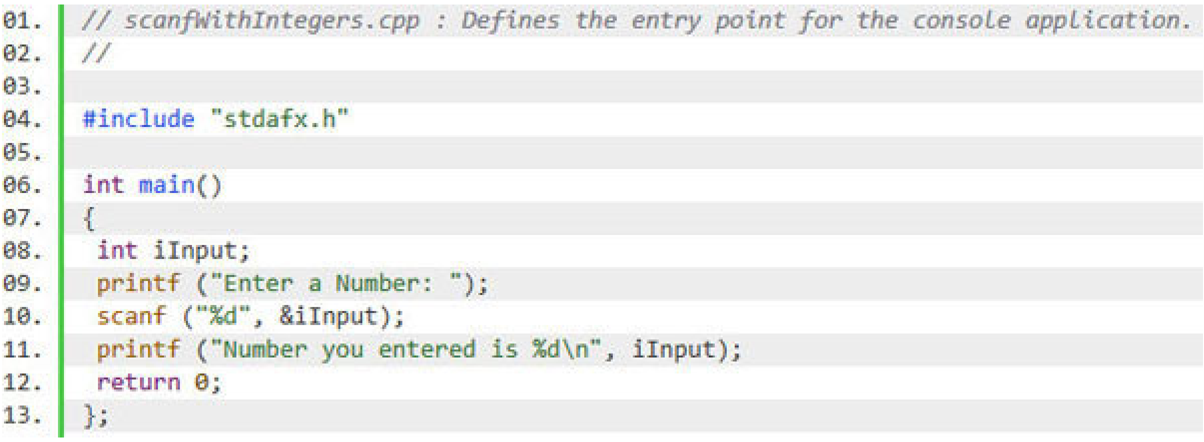
Function scanf with Optimization

Objective

In this chapter, we will understand the **scanf** function with respect to reverse engineering. We will talk about the **scanf** code pattern in the disassembled code and understand how **scanf** inputs are stored in memory. We will also cover the **scanf** program with both optimized and non-optimized code.

Function scanf with Integers

In this simple C/C++ code, we ask the user to input a number of type integer and print it on the console. We use **scanf** to capture user input and **printf** to print the number entered by the user in our C/C++ code:



```
// scanfWithIntegers.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"

int main()
{
 int iInput;
 printf ("Enter a Number: ");
 scanf ("%d", &iInput);
 printf ("Number you entered is %d\n", iInput);
 return 0;
};
```

Figure 13.1: *scanfWithIntegers.cpp*

Function scanf without Optimization

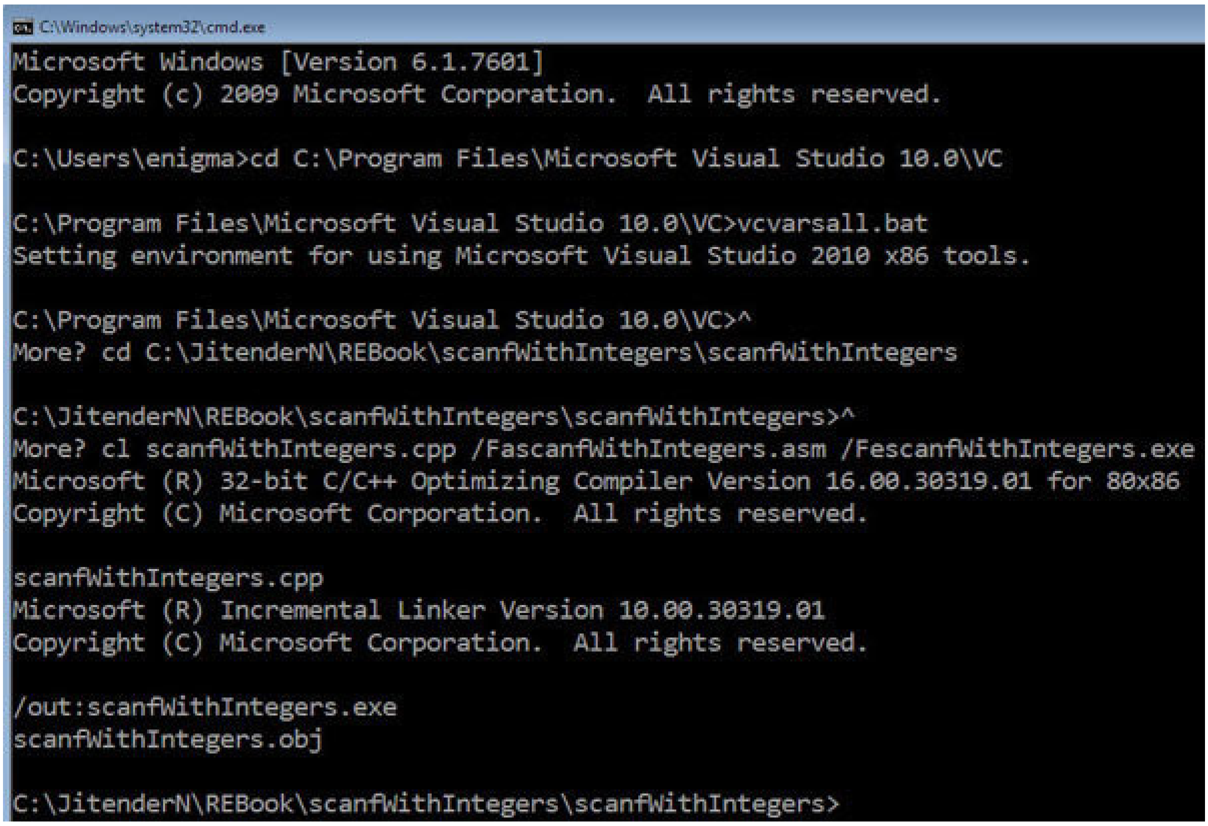
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:

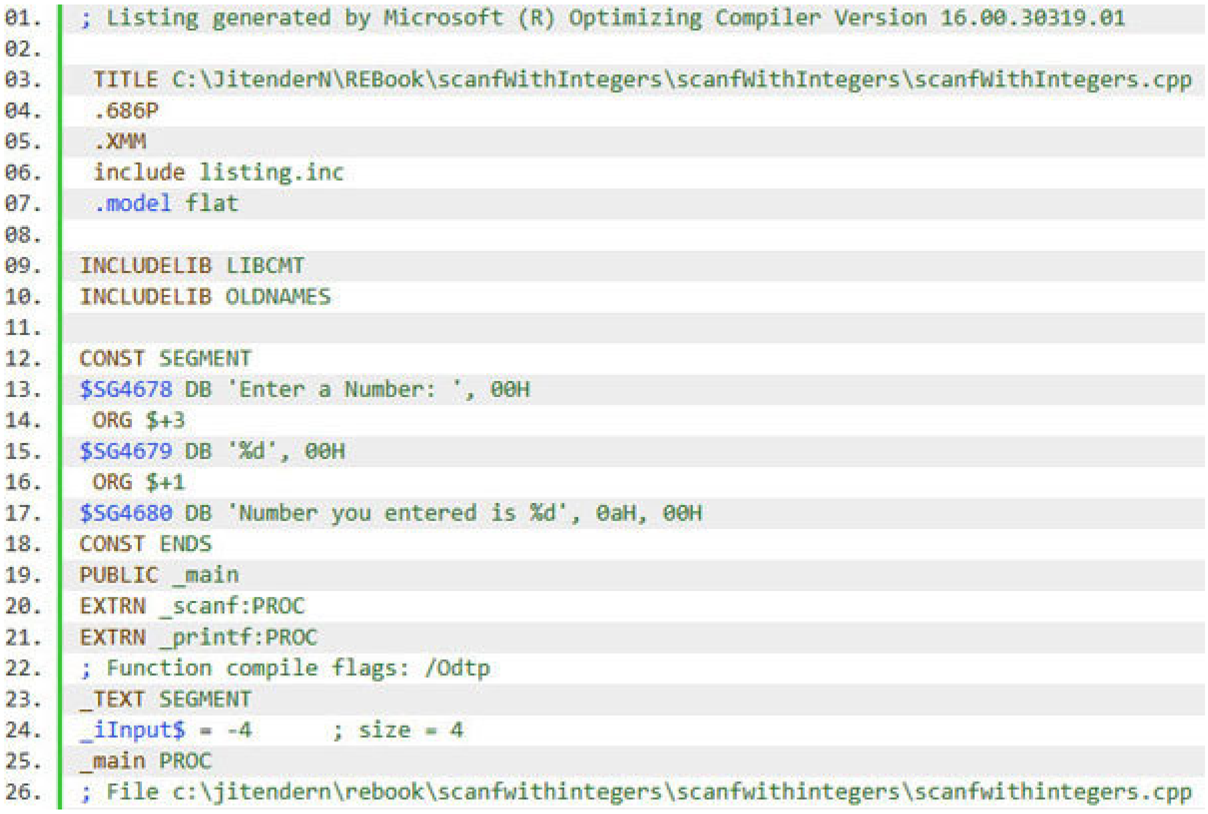


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers

C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers>^
More? cl scanfWithIntegers.cpp /FascanfWithIntegers.asm /FescanfWithIntegers.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

scanfWithIntegers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:scanfWithIntegers.exe
scanfWithIntegers.obj

C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers>
```

Figure 13.2: *Function scanf without Optimization*

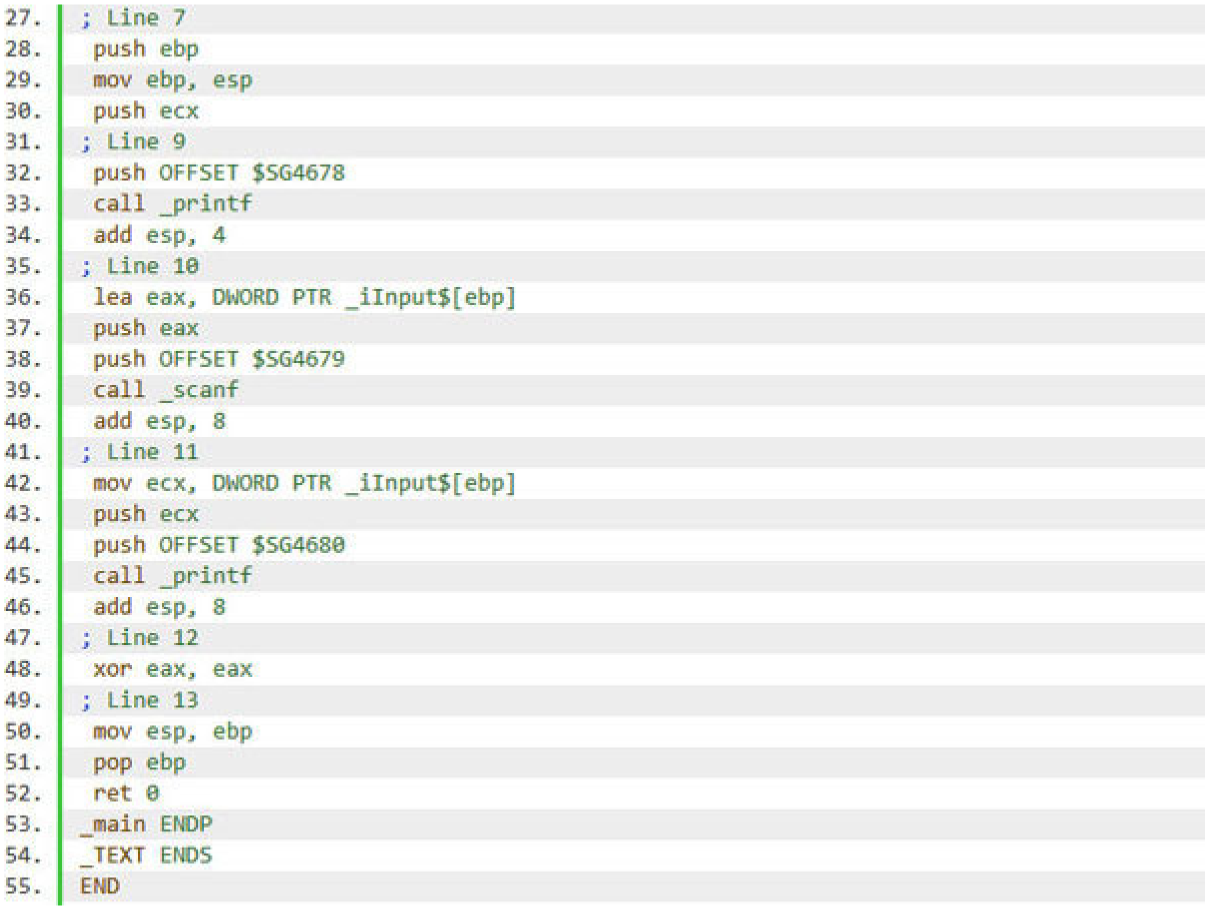
This will generate the assembly code and the EXE file. This time, before analyzing, we will disable the **Address Space Layout Randomization** It's a security mechanism by which the base address of the PE file is randomized on every load of the **Portable Executable** file generated with our MSVC compiler. This will help us reload the PE file without randomizing its base address. To disable ASLR, refer to the

We will now use the PE file with the disabled ASLR for further analysis using x32dbg. Let's move to the generated assembly listing:



```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers\scanfWithIntegers.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4678 DB 'Enter a Number: ', 00H
 ORG $+3
$SG4679 DB '%d', 00H
 ORG $+1
$SG4680 DB 'Number you entered is %d', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_iInput$ = -4      ; size = 4
_main PROC
; File c:\jitendern\rebook\scanfwithintegers\scanfwithintegers\scanfwithintegers.cpp
```

Figure 13.3: *scanfWithIntegers.asm-Part-1*



```
27. ; Line 7
28.  push ebp
29.  mov ebp, esp
30.  push ecx
31. ; Line 9
32.  push OFFSET $SG4678
33.  call _printf
34.  add esp, 4
35. ; Line 10
36.  lea eax, DWORD PTR _iInput$[ebp]
37.  push eax
38.  push OFFSET $SG4679
39.  call _scanf
40.  add esp, 8
41. ; Line 11
42.  mov ecx, DWORD PTR _iInput$[ebp]
43.  push ecx
44.  push OFFSET $SG4680
45.  call _printf
46.  add esp, 8
47. ; Line 12
48.  xor eax, eax
49. ; Line 13
50.  mov esp, ebp
51.  pop ebp
52.  ret 0
53. _main ENDP
54. _TEXT ENDS
55. END
```

Figure 13.4: *scanfWithIntegers.asm-Part-2*

Let's walk through the code:

▼Line 27-30

```
; Line 7
push ebp
mov ebp, esp
push ecx
```

The code starts with a simple **main** function prologue.

▼Line 31-34

```
; Line 9
push OFFSET $SG4678
call _printf
add esp, 4
```

The C/C++ code on line 9 prints the string constant on the console:

```
printf ("Enter a Number: ");
```

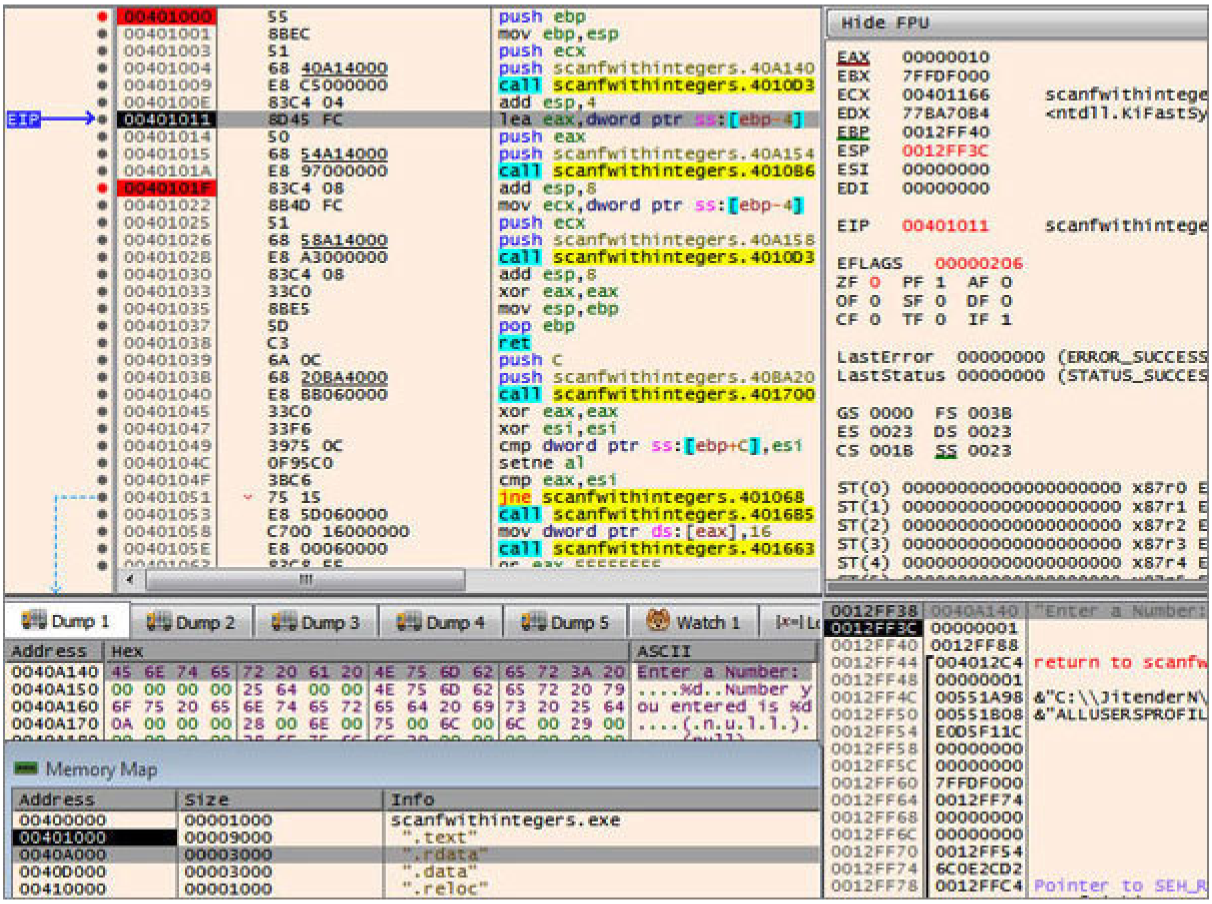
In our assembly code, before calling the **printf** function, the string constant **$SG4678** stored in the **.rdata** segment is pushed onto the stack. Once the arguments to the **printf** functions are pushed onto the stack, the call to the **printf** function is made. On return, 4 bytes are added to **ESP** for stack cleaning. For analyzing the assembly instruction, we will load the PE file in x32dbg, and then put two breakpoints. The first breakpoint is at the start of the .TEXT segment and the other breakpoint is just after the **scanf** function return. Now, after loading the PE file in x32dbg, run the code. The execution will stop at the first breakpoint. From the first breakpoint **step into** the code using x32dbg, until you hit the preceding **add esp,4** instruction. The **ADD** instruction cleans the stack and the following is how the stack looks after executing the **ADD** instruction:

```
[ESP-0x4] 0012FF38  0040A140  "Enter a Number: ", arg to 1st printf
```

[ESP] 0012FF3C 00000001 ECX pushed, later to store Integer input passed

[ESP+0x4] 0012FF40 0012FF88 [EBP]

[ESP+0x8] 0012FF44 004012C4 return to 0x004012C4 from 0x00401000



```
00401000   55               push ebp
00401001   8BEC             mov ebp,esp
00401003   51               push ecx
00401004   68 40A14000      push scanfwithintegers.40A140
00401009   E8 C5000000      call scanfwithintegers.4010D3
0040100E   83C4 04          add esp,4
00401011   8D45 FC          lea eax,dword ptr ss:[ebp-4]
00401014   50               push eax
00401015   68 54A14000      push scanfwithintegers.40A154
0040101A   E8 97000000      call scanfwithintegers.4010B6
0040101F   83C4 08          add esp,8
00401022   8B4D FC          mov ecx,dword ptr ss:[ebp-4]
00401025   51               push ecx
00401026   68 58A14000      push scanfwithintegers.40A158
0040102B   E8 A3000000      call scanfwithintegers.4010D3
00401030   83C4 08          add esp,8
00401033   33C0             xor eax,eax
00401035   8BE5             mov esp,ebp
00401037   5D               pop ebp
00401038   C3               ret
00401039   6A 0C            push C
0040103B   68 20BA4000      push scanfwithintegers.40BA20
00401040   E8 BB060000      call scanfwithintegers.401700
00401045   33C0             xor eax,eax
00401047   33F6             xor esi,esi
00401049   3975 0C          cmp dword ptr ss:[ebp+C],esi
0040104C   0F95C0           setne al
0040104F   3BC6             cmp eax,esi
00401051   75 15            jne scanfwithintegers.401068
00401053   E8 5D060000      call scanfwithintegers.401685
00401058   C700 16000000    mov dword ptr ds:[eax],16
0040105E   E8 00060000      call scanfwithintegers.401663
```

```
EAX     00000010
EBX     7FFDF000
ECX     00401166       scanfwithintege
EDX     77BA70B4       <ntdll.KiFastSy
EBP     0012FF40
ESP     0012FF3C
ESI     00000000
EDI     00000000

EIP     00401011       scanfwithintege

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS
LastStatus 00000000 (STATUS_SUCCES

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 E
ST(1) 00000000000000000000 x87r1 E
ST(2) 00000000000000000000 x87r2 E
ST(3) 00000000000000000000 x87r3 E
ST(4) 00000000000000000000 x87r4 E
```

```
Address   Hex                                               ASCII
0040A140  45 6E 74 65 72 20 61 20 4E 75 6D 62 65 72 3A 20   Enter a Number:
0040A150  00 00 00 00 25 64 00 00 4E 75 6D 62 65 72 20 79   ....%d..Number y
0040A160  6F 75 20 65 6E 74 65 72 65 64 20 69 73 20 25 64   ou entered is %d
0040A170  0A 00 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00   ....(.n.u.l.l.).
```

```
Address   Size      Info
00400000  00001000  scanfwithintegers.exe
00401000  00009000   ".text"
0040A000  00003000   ".rdata"
0040D000  00003000   ".data"
00410000  00001000   ".reloc"
```

```
0012FF38  0040A140  "Enter a Number:
0012FF3C  00000001
0012FF40  0012FF88
0012FF44  004012C4  return to scanfw
0012FF48  00000001
0012FF4C  00551A98  &"C:\\JitenderN\
0012FF50  00551808  &"ALLUSERSPROFIL
0012FF54  E005F11C
0012FF58  00000000
0012FF5C  00000000
0012FF60  7FFDF000
0012FF64  0012FF74
0012FF68  00000000
0012FF6C  00000000
0012FF70  0012FF54
0012FF74  6C0E2CD2
0012FF78  0012FFC4  Pointer to SEH_R
```

Figure 13.5: *Stack after ADD insruction*

▼Line 35-40

```
; Line 10
lea eax, DWORD PTR _iInput$[ebp]
push eax
push OFFSET $SG4679
```

```
call _scanf
add esp, 8
```

The C/C++ code on line 10 calls the **scanf** function, which accepts the following two arguments:

```
scanf ("%d", &iInput);
```

In the assembly code, the **LEA** instruction is evaluated to:

```
lea eax, ss:[ebp-0x4]
```

LEA will store the input placeholder memory location in By 'input placeholder memory location', we mean the memory location which will be used to store the integer input by the user. Once this input placeholder memory location is stored in then it is pushed onto the stack as a parameter to the **scanf** function. Then another push instruction pushes the string constant **$SG4679** on the stack.

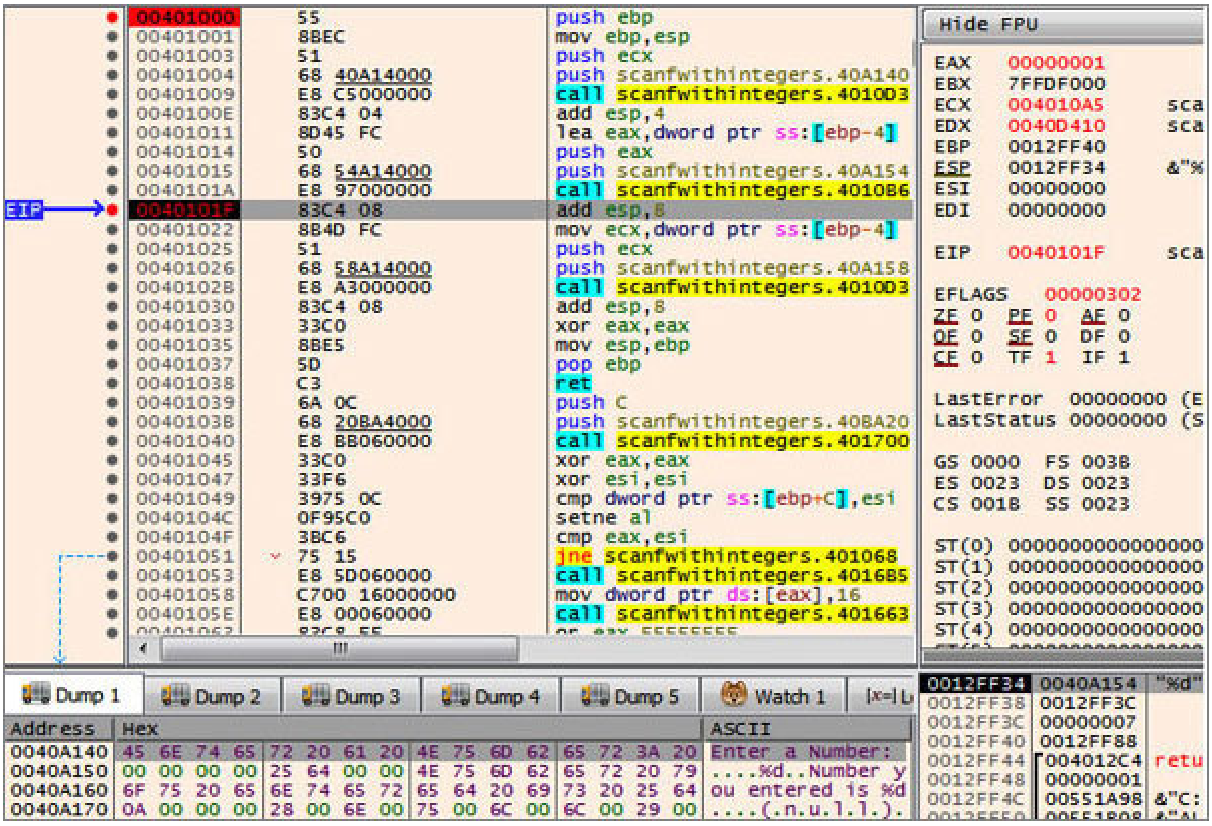
The call to **scanf** is made after pushing both the arguments. On return, 8 bytes are added to **ESP** for stack cleaning. The breakpoint is added at the **0x0040101F** memory location. The stack state at breakpoint is as follows:

```
[ESP]   0012FF34   0040A154   "%d", parameter to scanf ()
[ESP+0x4] 0012FF38   0012FF3C   Input placeholder memory location
```

[ESP+0x8] 0012FF3C 00000007 Input placeholder, Number 7 is entered by user

[ESP+0xC] 0012FF40 0012FF88 [EBP]

[ESP+0x10] 0012FF44 004012C4 return to 0x004012C4 from 0x00401000



```
00401000   55              push ebp
00401001   8BEC            mov ebp,esp
00401003   51              push ecx
00401004   68 40A14000     push scanfwithintegers.40A140
00401009   E8 C5000000     call scanfwithintegers.4010D3
0040100E   83C4 04         add esp,4
00401011   8D45 FC         lea eax,dword ptr ss:[ebp-4]
00401014   50              push eax
00401015   68 54A14000     push scanfwithintegers.40A154
0040101A   E8 97000000     call scanfwithintegers.4010B6
0040101F   83C4 08         add esp,8
00401022   8B4D FC         mov ecx,dword ptr ss:[ebp-4]
00401025   51              push ecx
00401026   68 58A14000     push scanfwithintegers.40A158
0040102B   E8 A3000000     call scanfwithintegers.4010D3
00401030   83C4 08         add esp,8
00401033   33C0            xor eax,eax
00401035   8BE5            mov esp,ebp
00401037   5D              pop ebp
00401038   C3              ret
00401039   6A 0C           push C
0040103B   68 20BA4000     push scanfwithintegers.40BA20
00401040   E8 BB060000     call scanfwithintegers.401700
00401045   33C0            xor eax,eax
00401047   33F6            xor esi,esi
00401049   3975 0C         cmp dword ptr ss:[ebp+C],esi
0040104C   0F95C0          setne al
0040104F   3BC6            cmp eax,esi
00401051   75 15           jne scanfwithintegers.401068
00401053   E8 5D060000     call scanfwithintegers.4016B5
00401058   C700 16000000   mov dword ptr ds:[eax],16
0040105E   E8 00060000     call scanfwithintegers.401663
```

```
EAX   00000001
EBX   7FFDF000
ECX   004010A5        sca
EDX   0040D410        sca
EBP   0012FF40
ESP   0012FF34        &"%
ESI   00000000
EDI   00000000

EIP   0040101F        sca

EFLAGS    00000302
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

LastError   00000000 (E
LastStatus  00000000 (S

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000
ST(1) 0000000000000000
ST(2) 0000000000000000
ST(3) 0000000000000000
ST(4) 0000000000000000
```

```
Address   Hex                                              ASCII
0040A140  45 6E 74 65 72 20 61 20 4E 75 6D 62 65 72 3A 20  Enter a Number: 
0040A150  00 00 00 00 25 64 00 00 4E 75 6D 62 65 72 20 79  ....%d..Number y
0040A160  6F 75 20 65 6E 74 65 72 65 64 20 69 73 20 25 64  ou entered is %d
0040A170  0A 00 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  ....(.n.u.l.l.).
```

```
0012FF34  0040A154  "%d"
0012FF38  0012FF3C
0012FF3C  00000007
0012FF40  0012FF88
0012FF44  004012C4  retu
0012FF48  00000001
0012FF4C  00551A98  &"C:
```

Figure 13.6: User entered number on stack

▼Line 41-46

```
; Line 11
mov ecx, DWORD PTR _iInput$[ebp]
push ecx
push OFFSET $SG4680
call _printf
```

```
add esp, 8
```

The C/C++ code on line 11 calls the **printf** function. This will print the number entered by the user:

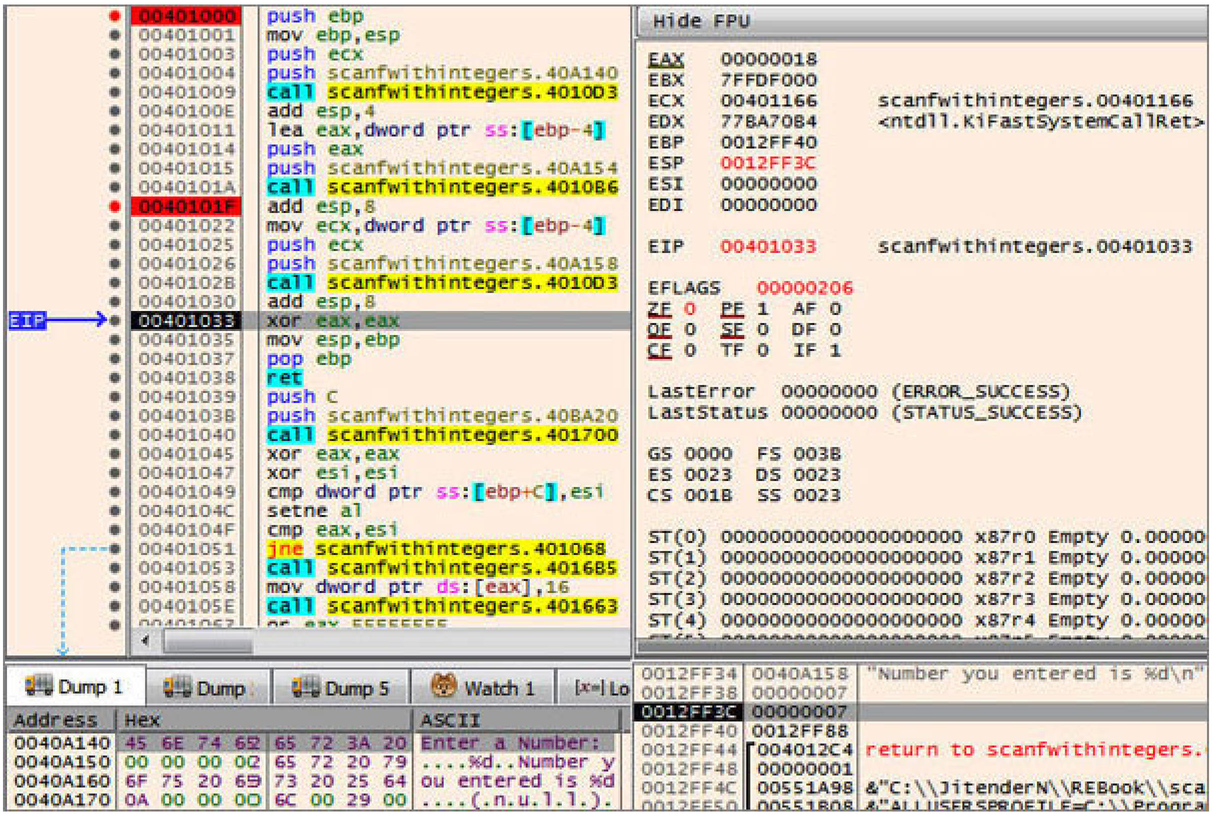
```
printf ("Number you entered is %d\n", iInput);
```

In the assembly code, the **MOV** instruction is evaluated to:

```
mov ecx, dword ptr ss:[ebp-0x4]
```

The **MOV** instruction will move the user input stored at **ss:[ebp-0x4]** to Now, both the arguments the first is the string constant and the second is the number that the user entered to the **printf** function are pushed on the stack:

```
[ESP-0x8] 0012FF34  0040A158  "Number you entered is %d\n", arg to 2nd printf()
[ESP-0x4] 0012FF38  00000007  Number 7 is entered by user, arg to 2nd printf()
[ESP]  0012FF3C  00000007  Input placeholder, Number 7 is entered by user
[ESP+0x4] 0012FF40  0012FF88  [EBP]
[ESP+0x8] 0012FF44  004012C4  return to 0x004012C4 from 0x00401000
```



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  push scanfwithintegers.40A140
00401009  call scanfwithintegers.4010D3
0040100E  add esp,4
00401011  lea eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push scanfwithintegers.40A154
0040101A  call scanfwithintegers.4010B6
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  push ecx
00401026  push scanfwithintegers.40A158
0040102B  call scanfwithintegers.4010D3
00401030  add esp,8
00401033  xor eax,eax
00401035  mov esp,ebp
00401037  pop ebp
00401038  ret
00401039  push C
0040103B  push scanfwithintegers.40BA20
00401040  call scanfwithintegers.401700
00401045  xor eax,eax
00401047  xor esi,esi
00401049  cmp dword ptr ss:[ebp+C],esi
0040104C  setne al
0040104F  cmp eax,esi
00401051  jne scanfwithintegers.401068
00401053  call scanfwithintegers.401685
00401058  mov dword ptr ds:[eax],16
0040105E  call scanfwithintegers.401663
```

```
EAX   00000018
EBX   7FFDF000
ECX   00401166     scanfwithintegers.00401166
EDX   77BA70B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF40
ESP   0012FF3C
ESI   00000000
EDI   00000000

EIP   00401033     scanfwithintegers.00401033

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.00000
ST(1) 00000000000000000000 x87r1 Empty 0.00000
ST(2) 00000000000000000000 x87r2 Empty 0.00000
ST(3) 00000000000000000000 x87r3 Empty 0.00000
ST(4) 00000000000000000000 x87r4 Empty 0.00000
```

```
Address   Hex                      ASCII
0040A140  45 6E 74 65 65 72 3A 20  Enter a Number:
0040A150  00 00 00 02 65 72 20 79  ....%d..Number y
0040A160  6F 75 20 65 73 20 25 64  ou entered is %d
0040A170  0A 00 00 00 6C 00 29 00  ....(.n.u.l.l.).
```

```
0012FF34  0040A158  "Number you entered is %d\n"
0012FF38  00000007
0012FF3C  00000007
0012FF40  0012FF88
0012FF44  004012C4  return to scanfwithintegers.
0012FF48  00000001
0012FF4C  00551A98  &"C:\\JitenderN\\REBook\\sca
```

Figure 13.7: After printf function

▼Line 47-55

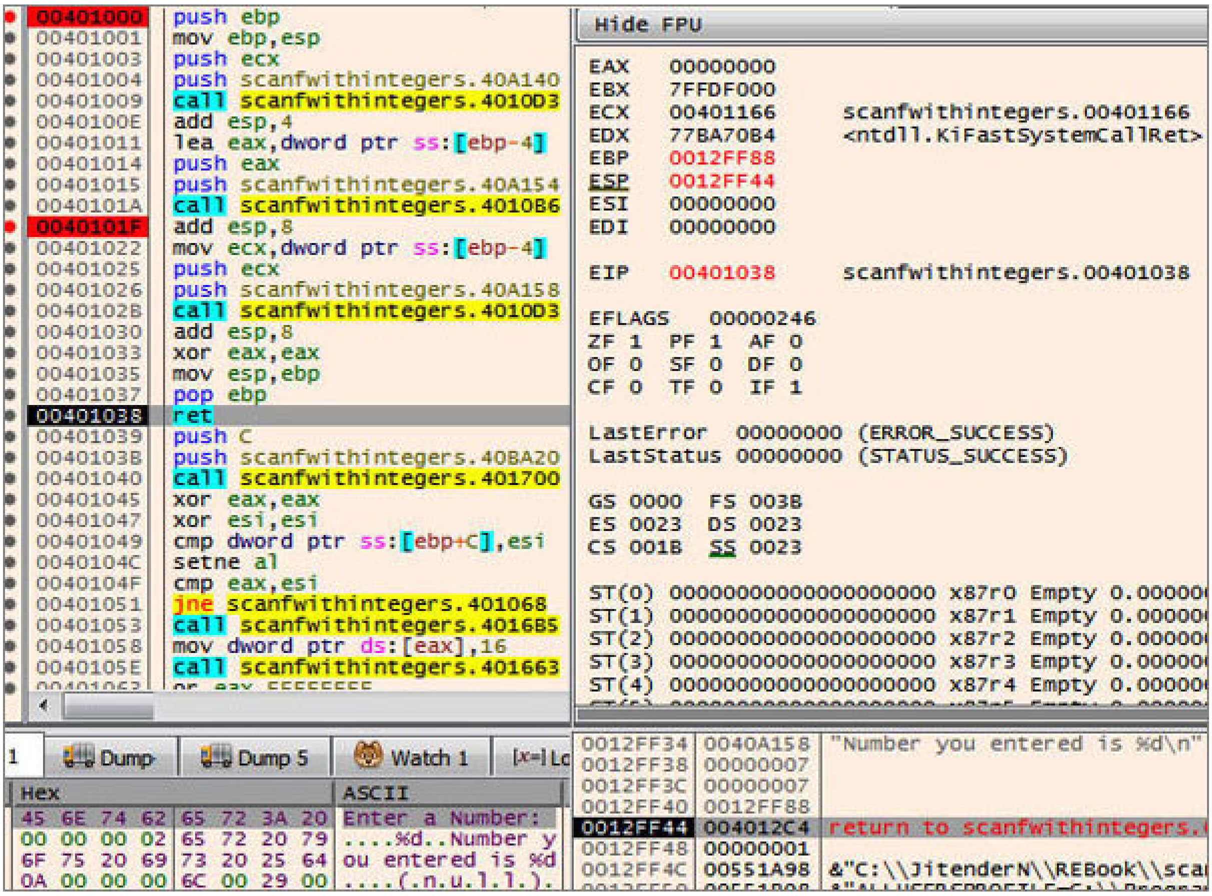
; Line 12
xor eax, eax
; Line 13
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END

XOR and function epilogue will clean up **EAX** and stack, respectively. EAX is XOR'ed to return 0. **ENDP** will end the **main**

procedure and **ENDS** will end the text segment. The stack will be as follows:

[ESP+0x10] 0012FF34 0040A158 JUNK
[ESP+0xC] 0012FF38 00000007 JUNK

[ESP+0x8] 0012FF3C 00000007 JUNK
[ESP+0x4] 0012FF40 0012FF88 [EBP] popped up
[ESP] 0012FF44 004012C4 return to 0x004012C4 from 0x00401000



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  push scanfwithintegers.40A140
00401009  call scanfwithintegers.4010D3
0040100E  add esp,4
00401011  lea eax,dword ptr ss:[ebp-4]
00401014  push eax
00401015  push scanfwithintegers.40A154
0040101A  call scanfwithintegers.4010B6
0040101F  add esp,8
00401022  mov ecx,dword ptr ss:[ebp-4]
00401025  push ecx
00401026  push scanfwithintegers.40A158
0040102B  call scanfwithintegers.4010D3
00401030  add esp,8
00401033  xor eax,eax
00401035  mov esp,ebp
00401037  pop ebp
00401038  ret
00401039  push C
0040103B  push scanfwithintegers.40BA20
00401040  call scanfwithintegers.401700
00401045  xor eax,eax
00401047  xor esi,esi
00401049  cmp dword ptr ss:[ebp+C],esi
0040104C  setne al
0040104F  cmp eax,esi
00401051  jne scanfwithintegers.401068
00401053  call scanfwithintegers.4016B5
00401058  mov dword ptr ds:[eax],16
0040105E  call scanfwithintegers.401663
```

```
EAX    00000000
EBX    7FFDF000
ECX    00401166     scanfwithintegers.00401166
EDX    77BA70B4     <ntdll.KiFastSystemCallRet>
EBP    0012FF88
ESP    0012FF44
ESI    00000000
EDI    00000000

EIP    00401038     scanfwithintegers.00401038

EFLAGS    00000246
ZF 1   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000 x87r0 Empty 0.00000
ST(1) 0000000000000000000 x87r1 Empty 0.00000
ST(2) 0000000000000000000 x87r2 Empty 0.00000
ST(3) 0000000000000000000 x87r3 Empty 0.00000
ST(4) 0000000000000000000 x87r4 Empty 0.00000
```

```
45 6E 74 62 65 72 3A 20  Enter a Number:
00 00 00 02 65 72 20 79  ....%d..Number y
6F 75 20 69 73 20 25 64  ou entered is %d
0A 00 00 00 6C 00 29 00  ....(.n.u.l.l.).
```

```
0012FF34  0040A158  "Number you entered is %d\n"
0012FF38  00000007
0012FF3C  00000007
0012FF40  0012FF88
0012FF44  004012C4  return to scanfwithintegers.
0012FF48  00000001
0012FF4C  00551A98  &"C:\\JitenderN\\REBook\\sca
```

Figure 13.8: Stack cleaned

Function scanf with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

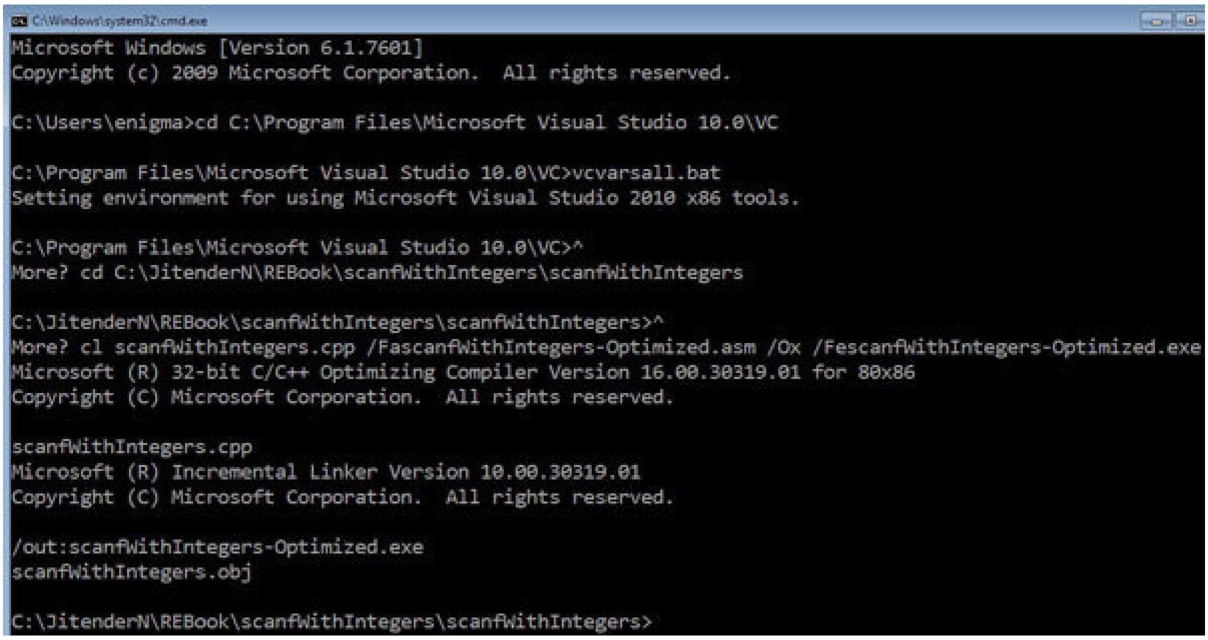
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



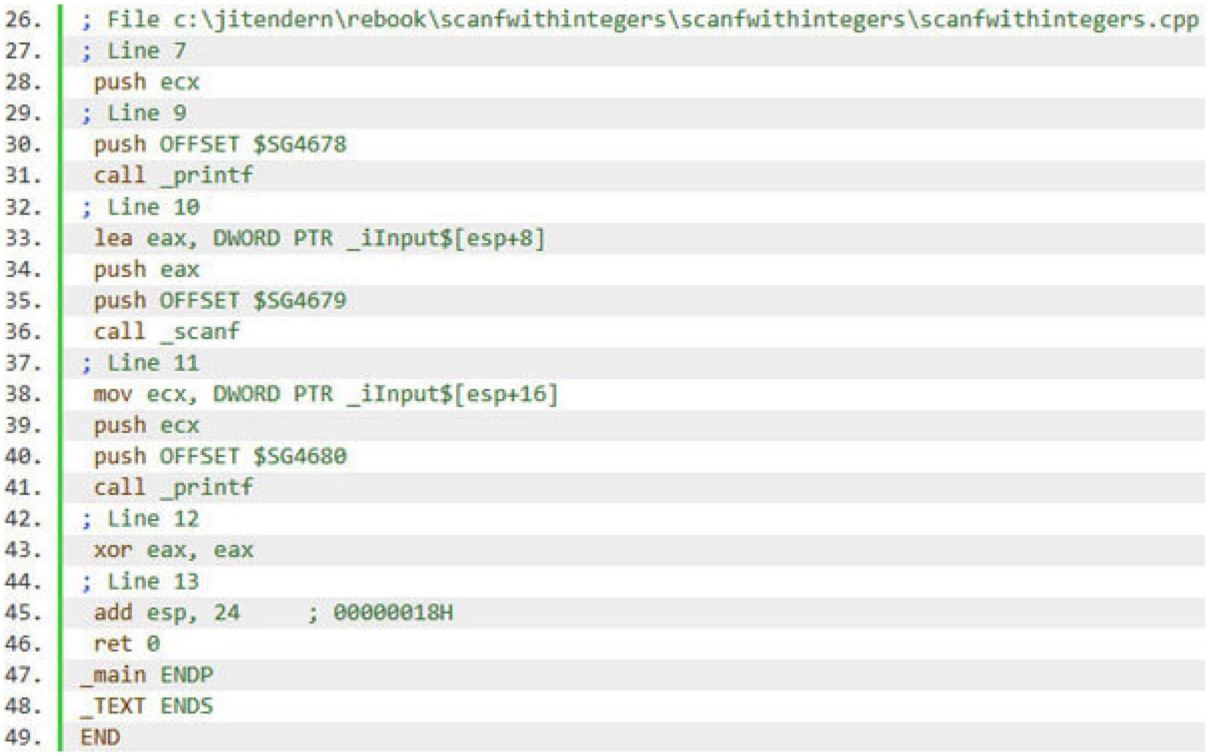
```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers

C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers>^
More? cl scanfWithIntegers.cpp /FascanfWithIntegers-Optimized.asm /Ox /FescanfWithIntegers-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

scanfWithIntegers.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:scanfWithIntegers-Optimized.exe
scanfWithIntegers.obj

C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers>
```

Figure 13.9: Function scanf with optimization

Let’s move to the generated assembly listing:

```
; Listing generated by Microsoft (R) Optimizing Compiler Version 16.00.30319.01

 TITLE C:\JitenderN\REBook\scanfWithIntegers\scanfWithIntegers\scanfWithIntegers.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4678 DB 'Enter a Number: ', 00H
 ORG $+3
$SG4679 DB '%d', 00H
 ORG $+1
$SG4680 DB 'Number you entered is %d', 0aH, 00H
CONST ENDS
PUBLIC _main
EXTRN _scanf:PROC
EXTRN _printf:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_iInput$ = -4       ; size = 4
_main PROC
```

Figure 13.10: *scanfWithIntegers-Optimized.asm-Part-1*



```
26. ; File c:\jitendern\rebook\scanfwithintegers\scanfwithintegers\scanfwithintegers.cpp
27. ; Line 7
28.  push ecx
29. ; Line 9
30.  push OFFSET $SG4678
31.  call _printf
32. ; Line 10
33.  lea eax, DWORD PTR _iInput$[esp+8]
34.  push eax
35.  push OFFSET $SG4679
36.  call _scanf
37. ; Line 11
38.  mov ecx, DWORD PTR _iInput$[esp+16]
39.  push ecx
40.  push OFFSET $SG4680
41.  call _printf
42. ; Line 12
43.  xor eax, eax
44. ; Line 13
45.  add esp, 24      ; 00000018H
46.  ret 0
47. _main ENDP
48. _TEXT ENDS
49. END
```

Figure 13.11: *scanfWithIntegers-Optimized.asm-Part-2*

The main difference between optimized and non-optimized code is that in optimized code, the function prologue and epilogue are eliminated. Secondly, the stack cleaning is not done after each and every function call but towards the end of the **main** function. We will walk through the assembly instruction in the same way we did in the non-optimization section by putting breakpoints in x32dbg:

▼Line 27-31

```
; Line 7
push ecx
```

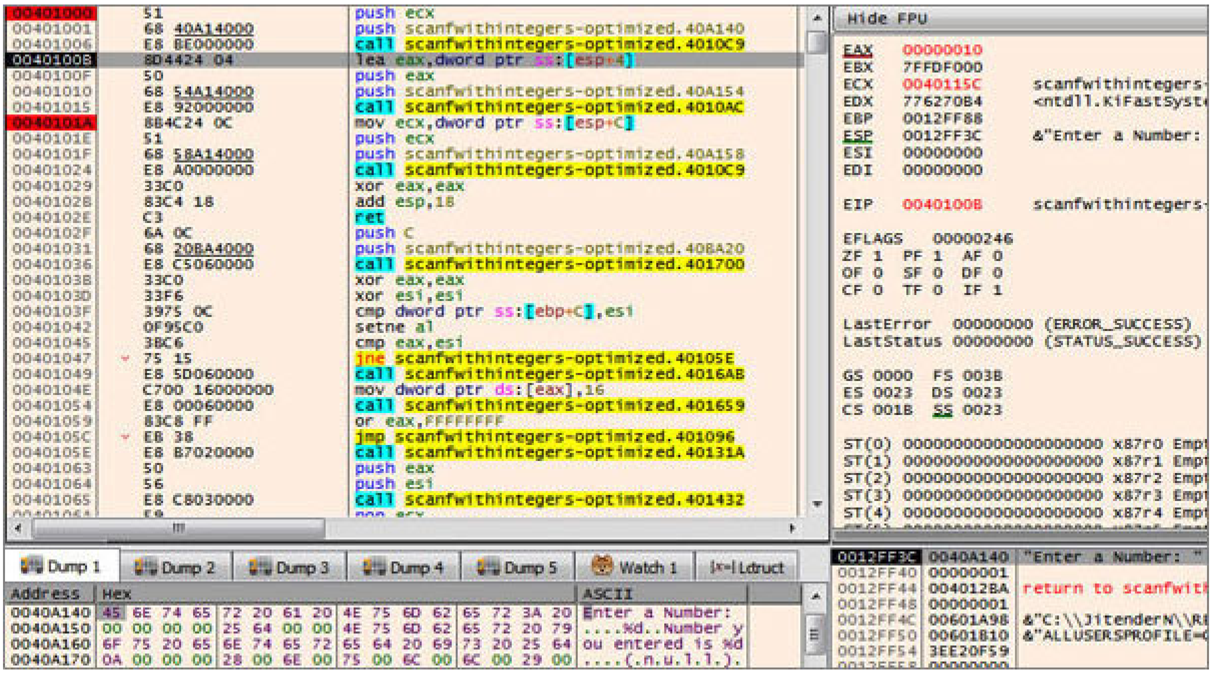
```
; Line 9
push OFFSET $SG4678
call _printf
```

The C/C++ code on line 9 prints the string constant on the console:

```
printf ("Enter a Number: ");
```

The string constant **$SG4678** stored in the **.rdata** segment is pushed onto the stack before calling the **printf** function. Note that the function prologue as well as the stack cleaning after the **printf** function call are eliminated. We will also see that the stack is cleaned towards the end. The following is the stack state after executing the preceding instruction:

```
[ESP]  0012FF3C  0040A140  "Enter a Number: ", parameter to 1st printf()
[ESP+0x4] 0012FF40  00000001  ECX is pushed, used later to store Integer input
[ESP+0x8] 0012FF44  004012BA  return to 0x004012BA from 0x00401000
```



```
00401000   51            push ecx
00401001   68 40A14000   push scanfwithintegers-optimized.40A140
00401006   E8 BE000000   call scanfwithintegers-optimized.4010C9
0040100B   8D4424 04     lea eax,dword ptr ss:[esp+4]
0040100F   50            push eax
00401010   68 54A14000   push scanfwithintegers-optimized.40A154
00401015   E8 92000000   call scanfwithintegers-optimized.4010AC
0040101A   8B4C24 0C     mov ecx,dword ptr ss:[esp+C]
0040101E   51            push ecx
0040101F   68 58A14000   push scanfwithintegers-optimized.40A158
00401024   E8 A0000000   call scanfwithintegers-optimized.4010C9
00401029   33C0          xor eax,eax
0040102B   83C4 18       add esp,18
0040102E   C3            ret
0040102F   6A 0C         push C
00401031   68 20BA4000   push scanfwithintegers-optimized.40BA20
00401036   E8 C5060000   call scanfwithintegers-optimized.401700
0040103B   33C0          xor eax,eax
0040103D   33F6          xor esi,esi
0040103F   3975 0C       cmp dword ptr ss:[ebp+C],esi
00401042   0F95C0        setne al
00401045   3BC6          cmp eax,esi
00401047   75 15         jne scanfwithintegers-optimized.40105E
00401049   E8 5D060000   call scanfwithintegers-optimized.4016AB
0040104E   C700 16000000 mov dword ptr ds:[eax],16
00401054   E8 00060000   call scanfwithintegers-optimized.401659
00401059   83C8 FF       or eax,FFFFFFFF
0040105C   EB 38         jmp scanfwithintegers-optimized.401096
0040105E   E8 B7020000   call scanfwithintegers-optimized.40131A
00401063   50            push eax
00401064   56            push esi
00401065   E8 C8030000   call scanfwithintegers-optimized.401432
```

```
EAX   00000010
EBX   7FFDF000
ECX   0040115C    scanfwithintegers-
EDX   776270B4    <ntdll.KiFastSyste
EBP   0012FF88
ESP   0012FF3C    &"Enter a Number:
ESI   00000000
EDI   00000000

EIP   0040100B    scanfwithintegers-

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empt
ST(1) 00000000000000000000 x87r1 Empt
ST(2) 00000000000000000000 x87r2 Empt
ST(3) 00000000000000000000 x87r3 Empt
ST(4) 00000000000000000000 x87r4 Empt
```

```
0012FF3C  0040A140  "Enter a Number: "
0012FF40  00000001
0012FF44  004012BA  return to scanfwit
0012FF48  00000001
0012FF4C  00601A98  &"C:\\JitenderN\\R
0012FF50  00601810  &"ALLUSERSPROFILE=
0012FF54  3EE20F59
```

```
Address   Hex                                              ASCII
0040A140  45 6E 74 65 72 20 61 20 4E 75 6D 62 65 72 3A 20  Enter a Number:
0040A150  00 00 00 00 25 64 00 00 4E 75 6D 62 65 72 20 79  ....%d..Number y
0040A160  6F 75 20 65 6E 74 65 72 65 64 20 69 73 20 25 64  ou entered is %d
0040A170  0A 00 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  ....(.n.u.l.l.).
```

Figure 13.12: *After printf*

▼Line 32-36

; Line 10

lea eax, DWORD PTR _iInput$[esp+8]

push eax

push OFFSET $SG4679

call _scanf

The C/C++ code on line 10 calls the **scanf** function, which accepts the following two arguments:

scanf ("%d", &iInput);

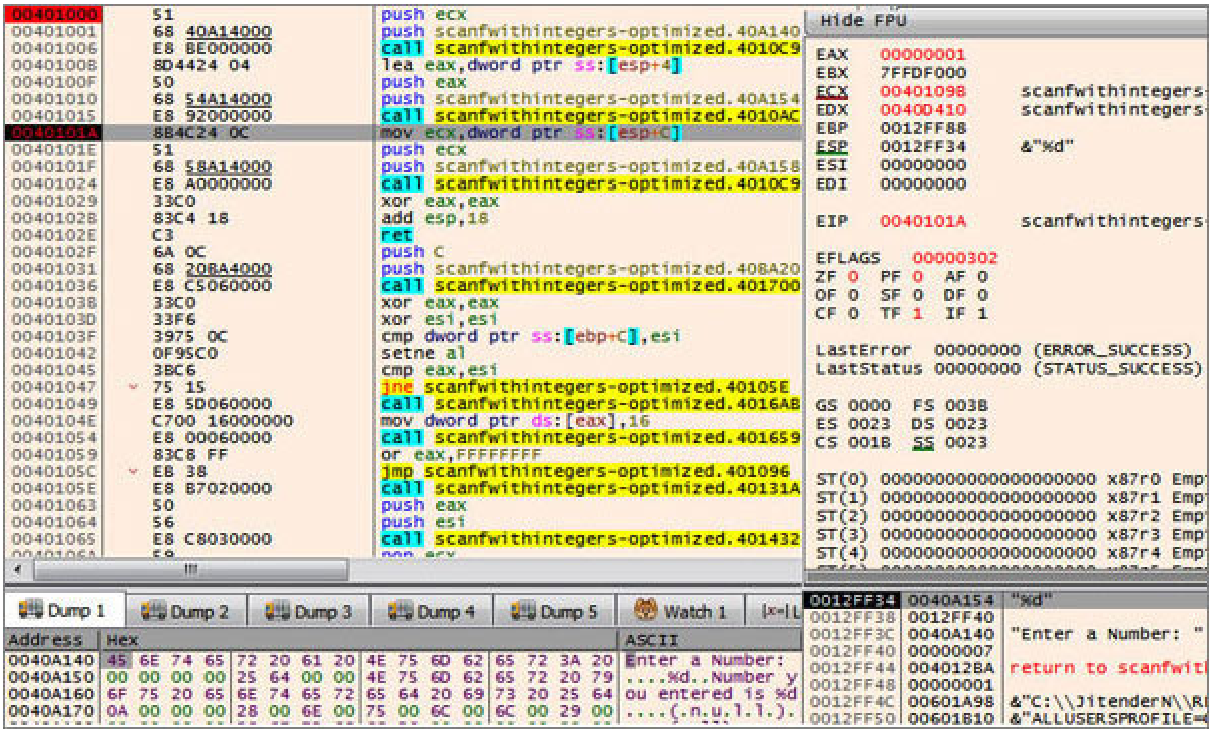
In the assembly code, the **LEA** instruction is evaluated to:

```
lea eax, ss:[esp+0x4]
```

LEA will load the effective address of the input placeholder memory location in EAX. As already stated in non-optimized section, input placeholder memory location means, the memory location which will be used to store the Integer input by user. The **Scanf** function takes two arguments, one is the memory location where the user's input will be stored and the other is the string constant

Once both the parameters are pushed onto the stack, the call to **scanf** is made. During the call, the user is asked to enter the number. The integer number supplied by the user will be stored at the input placeholder memory location. We can also notice that the stack cleaning is not done after the function call. The breakpoint is added at the **0x0040101A** memory location. The stack state at breakpoint is as follows:

```
[ESP]  0012FF34  0040A154  "%d", parameter to scanf ()
[ESP+0x4] 0012FF38  0012FF40  Input placeholder memory location
[ESP+0x8] 0012FF3C  0040A140  "Enter a Number: ", parameter to 1st printf()
[ESP+0xC] 0012FF40  00000007  Input placeholder, Number 7 is entered by user
[ESP+0x10] 0012FF44  004012BA  return to 0x004012BA from 0x00401000
```



```
00401000  51            push ecx
00401001  68 40A14000   push scanfwithintegers-optimized.40A140
00401006  E8 BE000000   call scanfwithintegers-optimized.4010C9
0040100B  8D4424 04     lea eax,dword ptr ss:[esp+4]
0040100F  50            push eax
00401010  68 54A14000   push scanfwithintegers-optimized.40A154
00401015  E8 92000000   call scanfwithintegers-optimized.4010AC
0040101A  8B4C24 0C     mov ecx,dword ptr ss:[esp+C]
0040101E  51            push ecx
0040101F  68 58A14000   push scanfwithintegers-optimized.40A158
00401024  E8 A0000000   call scanfwithintegers-optimized.4010C9
00401029  33C0          xor eax,eax
0040102B  83C4 18       add esp,18
0040102E  C3            ret
0040102F  6A 0C         push C
00401031  68 20BA4000   push scanfwithintegers-optimized.40BA20
00401036  E8 C5060000   call scanfwithintegers-optimized.401700
0040103B  33C0          xor eax,eax
0040103D  33F6          xor esi,esi
0040103F  3975 0C       cmp dword ptr ss:[ebp+C],esi
00401042  0F95C0        setne al
00401045  3BC6          cmp eax,esi
00401047  75 15         jne scanfwithintegers-optimized.40105E
00401049  E8 5D060000   call scanfwithintegers-optimized.4016AB
0040104E  C700 16000000 mov dword ptr ds:[eax],16
00401054  E8 00060000   call scanfwithintegers-optimized.401659
00401059  83C8 FF       or eax,FFFFFFFF
0040105C  EB 38         jmp scanfwithintegers-optimized.401096
0040105E  E8 B7020000   call scanfwithintegers-optimized.40131A
00401063  50            push eax
00401064  56            push esi
00401065  E8 C8030000   call scanfwithintegers-optimized.401432
```

```
EAX    00000001
EBX    7FFDF000
ECX    00401098     scanfwithintegers
EDX    00400410     scanfwithintegers
EBP    0012FF88
ESP    0012FF34     &"%d"
ESI    00000000
EDI    00000000

EIP    0040101A     scanfwithintegers

EFLAGS    00000302
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Emp
ST(1) 00000000000000000000 x87r1 Emp
ST(2) 00000000000000000000 x87r2 Emp
ST(3) 00000000000000000000 x87r3 Emp
ST(4) 00000000000000000000 x87r4 Emp
```

```
Address   Hex                                              ASCII
0040A140  45 6E 74 65 72 20 61 20 4E 75 6D 62 65 72 3A 20  Enter a Number: 
0040A150  00 00 00 00 25 64 00 00 4E 75 6D 62 65 72 20 79  ....%d..Number y
0040A160  6F 75 20 65 6E 74 65 72 65 64 20 69 73 20 25 64  ou entered is %d
0040A170  0A 00 00 00 28 00 6E 00 75 00 6C 00 6C 00 29 00  ....(.n.u.l.l.).
```

```
0012FF34  0040A154  "%d"
0012FF38  0012FF40
0012FF3C  0040A140  "Enter a Number: "
0012FF40  00000007
0012FF44  004012BA  return to scanfwit
0012FF48  00000001
0012FF4C  00601A98  &"C:\\JitenderN\\R
0012FF50  00601B10  &"ALLUSERSPROFILE=
```

***Figure 13.13:** After scanf*

▼Line 37-41

```
; Line 11
mov ecx, DWORD PTR _iInput$[esp+16]
push ecx
push OFFSET $SG4680
call _printf
```

The C/C++ code on line 11 calls the **printf** function. This will print the number that the user entered:

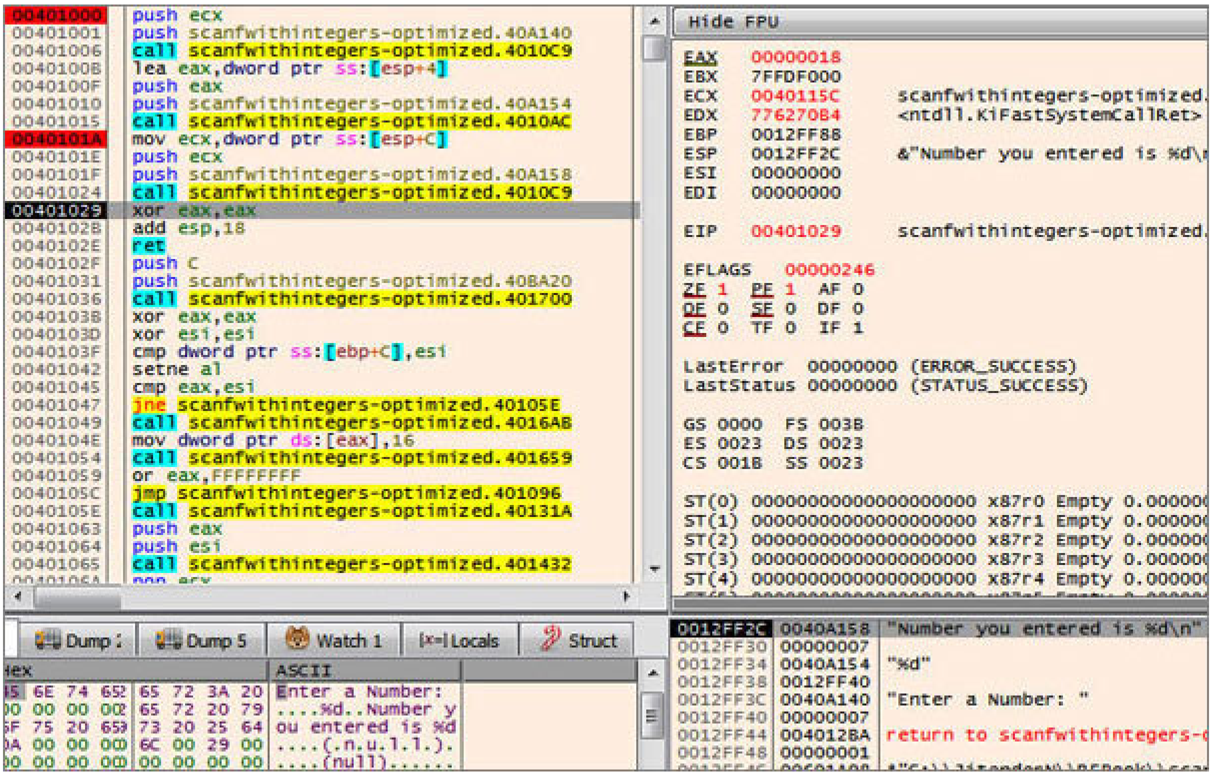
```
printf ("Number you entered is %d\n", iInput);
```

In the assembly code, the **MOV** instruction is evaluated to:

```
mov ecx, dword ptr ss:[esp+0xC]
```

The **MOV** instruction will move the user integer input stored at **ss:[esp+0xC]** to Now, both the arguments (first is the string constant and the second is the integer number that the user entered) of the **printf** function are pushed on the stack before the **printf** call. The stack state after the **CALL** instruction will be:

[ESP] 0012FF2C 0040A158 "Number you entered is %d\n", arg to 2nd printf()
[ESP+0x4] 0012FF30 00000007 Number 7 is entered by user, arg to 2nd printf()
[ESP+0x8] 0012FF34 0040A154 "%d", parameter to scanf ()
[ESP+0xC] 0012FF38 0012FF40 Input placeholder memory location
[ESP+0x10] 0012FF3C 0040A140 "Enter a Number: ", parameter to 1st printf()
[ESP+0x14] 0012FF40 00000007 Input placeholder, Number 7 is entered by user
[ESP+0x18] 0012FF44 004012BA return to 0x004012BA from 0x00401000



```
00401000  push ecx
00401001  push scanfwithintegers-optimized.40A140
00401006  call scanfwithintegers-optimized.4010C9
0040100B  lea eax,dword ptr ss:[esp+4]
0040100F  push eax
00401010  push scanfwithintegers-optimized.40A154
00401015  call scanfwithintegers-optimized.4010AC
0040101A  mov ecx,dword ptr ss:[esp+C]
0040101E  push ecx
0040101F  push scanfwithintegers-optimized.40A158
00401024  call scanfwithintegers-optimized.4010C9
00401029  xor eax,eax
0040102B  add esp,18
0040102E  ret
0040102F  push C
00401031  push scanfwithintegers-optimized.40BA20
00401036  call scanfwithintegers-optimized.401700
0040103B  xor eax,eax
0040103D  xor esi,esi
0040103F  cmp dword ptr ss:[ebp+C],esi
00401042  setne al
00401045  cmp eax,esi
00401047  jne scanfwithintegers-optimized.40105E
00401049  call scanfwithintegers-optimized.4016AB
0040104E  mov dword ptr ds:[eax],16
00401054  call scanfwithintegers-optimized.401659
00401059  or eax,FFFFFFFF
0040105C  jmp scanfwithintegers-optimized.401096
0040105E  call scanfwithintegers-optimized.40131A
00401063  push eax
00401064  push esi
00401065  call scanfwithintegers-optimized.401432
```

```
EAX   00000018
EBX   7FFDF000
ECX   0040115C     scanfwithintegers-optimized.
EDX   776270B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF2C     &"Number you entered is %d\
ESI   00000000
EDI   00000000

EIP   00401029     scanfwithintegers-optimized.

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.00000
ST(1) 00000000000000000000 x87r1 Empty 0.00000
ST(2) 00000000000000000000 x87r2 Empty 0.00000
ST(3) 00000000000000000000 x87r3 Empty 0.00000
ST(4) 00000000000000000000 x87r4 Empty 0.00000
```

```
5 6E 74 65 65 72 3A 20  Enter a Number:
0 00 00 00 65 72 20 79  ....%d..Number y
F 75 20 65 73 20 25 64  ou entered is %d
A 00 00 00 6C 00 29 00  ....(.n.u.l.l.).
0 00 00 00 00 00 00 00  ....(null)......
```

```
0012FF2C  0040A158  "Number you entered is %d\n"
0012FF30  00000007
0012FF34  0040A154  "%d"
0012FF38  0012FF40
0012FF3C  0040A140  "Enter a Number: "
0012FF40  00000007
0012FF44  004012BA  return to scanfwithintegers-
0012FF48  00000001
```

Figure 13.14: After second printf

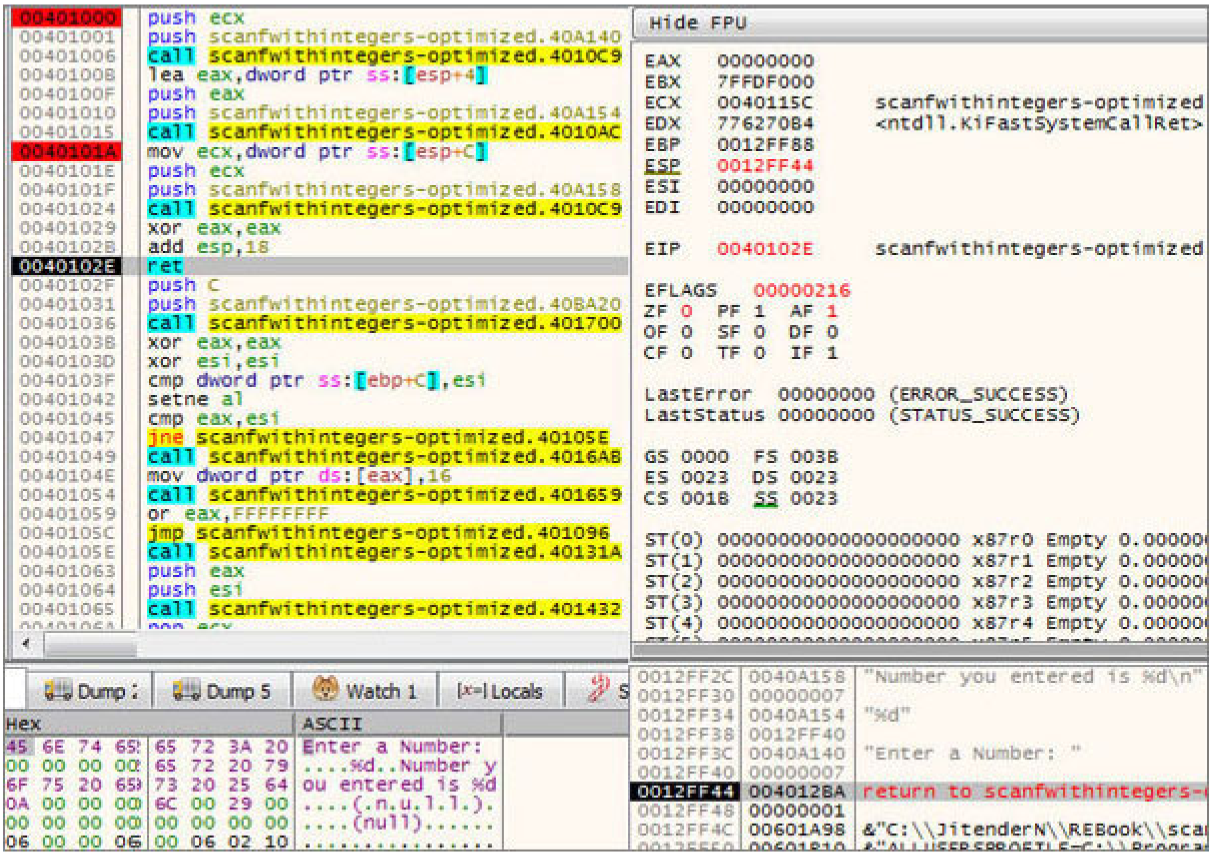
▼Line 42-49

```
; Line 12
xor eax, eax
; Line 13
add esp, 24        ; 00000018H
ret 0
_main ENDP
_TEXT ENDS
END
```

XOR will clean up **EAX** to return 0. You can notice that the stack is cleaned towards the end of the **main** function with the **ADD** instruction. The **ADD** instruction cleans the stack by adding 24 bytes to The **END** derivative ends the code. The stack state will be as follows:

[ESP-0x18] 0012FF2C 0040A158 JUNK
[ESP-0x14] 0012FF30 00000007 JUNK

[ESP-0x10] 0012FF34 0040A154 JUNK
[ESP-0xC] 0012FF38 0012FF40 JUNK
[ESP-0x8] 0012FF3C 0040A140 JUNK
[ESP-0x4] 0012FF40 00000007 JUNK
[ESP] 0012FF44 004012BA return to 0x004012BA from 0x00401000



```
00401000  push ecx
00401001  push scanfwithintegers-optimized.40A140
00401006  call scanfwithintegers-optimized.4010C9
0040100B  lea eax,dword ptr ss:[esp+4]
0040100F  push eax
00401010  push scanfwithintegers-optimized.40A154
00401015  call scanfwithintegers-optimized.4010AC
0040101A  mov ecx,dword ptr ss:[esp+C]
0040101E  push ecx
0040101F  push scanfwithintegers-optimized.40A158
00401024  call scanfwithintegers-optimized.4010C9
00401029  xor eax,eax
0040102B  add esp,18
0040102E  ret
0040102F  push C
00401031  push scanfwithintegers-optimized.40BA20
00401036  call scanfwithintegers-optimized.401700
0040103B  xor eax,eax
0040103D  xor esi,esi
0040103F  cmp dword ptr ss:[ebp+C],esi
00401042  setne al
00401045  cmp eax,esi
00401047  jne scanfwithintegers-optimized.40105E
00401049  call scanfwithintegers-optimized.4016AB
0040104E  mov dword ptr ds:[eax],16
00401054  call scanfwithintegers-optimized.401659
00401059  or eax,FFFFFFFF
0040105C  jmp scanfwithintegers-optimized.401096
0040105E  call scanfwithintegers-optimized.40131A
00401063  push eax
00401064  push esi
00401065  call scanfwithintegers-optimized.401432
```

```
EAX   00000000
EBX   7FFDF000
ECX   0040115C     scanfwithintegers-optimized
EDX   776270B4     <ntdll.KiFastSystemCallRet>
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   0040102E     scanfwithintegers-optimized

EFLAGS    00000216
ZF 0   PF 1   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.00000
ST(1) 00000000000000000000 x87r1 Empty 0.00000
ST(2) 00000000000000000000 x87r2 Empty 0.00000
ST(3) 00000000000000000000 x87r3 Empty 0.00000
ST(4) 00000000000000000000 x87r4 Empty 0.00000
```

```
45 6E 74 65 65 72 3A 20  Enter a Number:
00 00 00 00 65 72 20 79  ....%d..Number y
6F 75 20 65 73 20 25 64  ou entered is %d
0A 00 00 00 6C 00 29 00  ....(.n.u.l.l.).
00 00 00 00 00 00 00 00  ....(null)......
06 00 00 06 00 06 02 10  ................
```

```
0012FF2C  0040A158  "Number you entered is %d\n"
0012FF30  00000007
0012FF34  0040A154  "%d"
0012FF38  0012FF40
0012FF3C  0040A140  "Enter a Number: "
0012FF40  00000007
0012FF44  004012BA  return to scanfwithintegers-
0012FF48  00000001
0012FF4C  00601A98  &"C:\\JitenderN\\REBook\\sca
```

Figure 13.15: Stack cleaned

Conclusion

In this chapter, we understood how the input captured from a user is stored in memory using the **scanf** function. We saw the **scanf** code pattern in a disassembled code for both optimized and non-optimized code. In the next chapter, we will study the **strcpy** function and how its pattern looks while reverse engineering.

CHAPTER 14

Strcpy Program Pattern in Reverse Engineering

We have understood the patterns of arrays and pointers in the earlier chapters. Now we will talk about some real-world examples of code that use all these as a part of a single program. There are times when you want to copy data from one place to another place. To do so, we use some standard predefined functions. But in this chapter, we will talk about the implementation of strcpy, which is used to copy data from the source to the destination.

The implementation of strcpy will help you learn the combination of pointers and arrays in a single program and understand the code pattern in assembly. These patterns can be seen in many applications or software and you will find it interesting to know that this will allow you to find vulnerabilities in them.

Structure

In this chapter, we will cover the following topics:

Understand strcpy function

Strcpy without Optimization

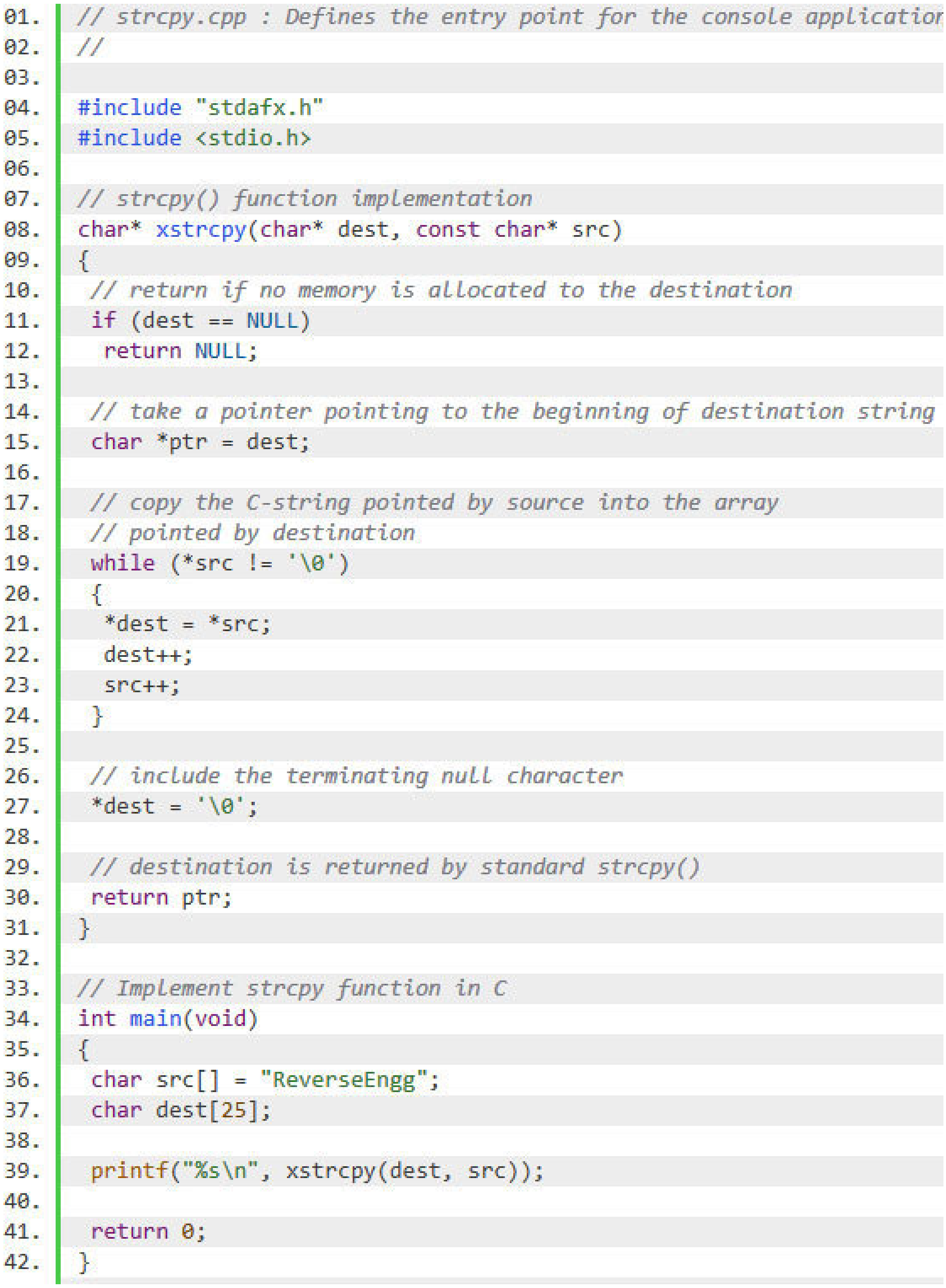
Strcpy with Optimization

Objective

In this chapter, we will talk about the **strcpy** function implementation with respect to reverse engineering. This function is very popular in software implementation to perform the string copy operation from the source location to the destination. We will also talk about byte-by-byte operations that happen during the **strcpy** execution. We will also cover **strcpy** program assembly pattern with optimized and non-optimized code.

Strcpy

In this example, we will take up the **strcpy** function with the name of **xstrcpy** to copy a string.



```
// strcpy.cpp : Defines the entry point for the console applicatio
//

#include "stdafx.h"
#include <stdio.h>

// strcpy() function implementation
char* xstrcpy(char* dest, const char* src)
{
 // return if no memory is allocated to the destination
 if (dest == NULL)
  return NULL;

 // take a pointer pointing to the beginning of destination string
 char *ptr = dest;

 // copy the C-string pointed by source into the array
 // pointed by destination
 while (*src != '\0')
 {
  *dest = *src;
  dest++;
  src++;
 }

 // include the terminating null character
 *dest = '\0';

 // destination is returned by standard strcpy()
 return ptr;
}

// Implement strcpy function in C
int main(void)
{
 char src[] = "ReverseEngg";
 char dest[25];

 printf("%s\n", xstrcpy(dest, src));

 return 0;
}
```

Figure 14.1: *strcpy.cpp*

Strcpy without Optimization

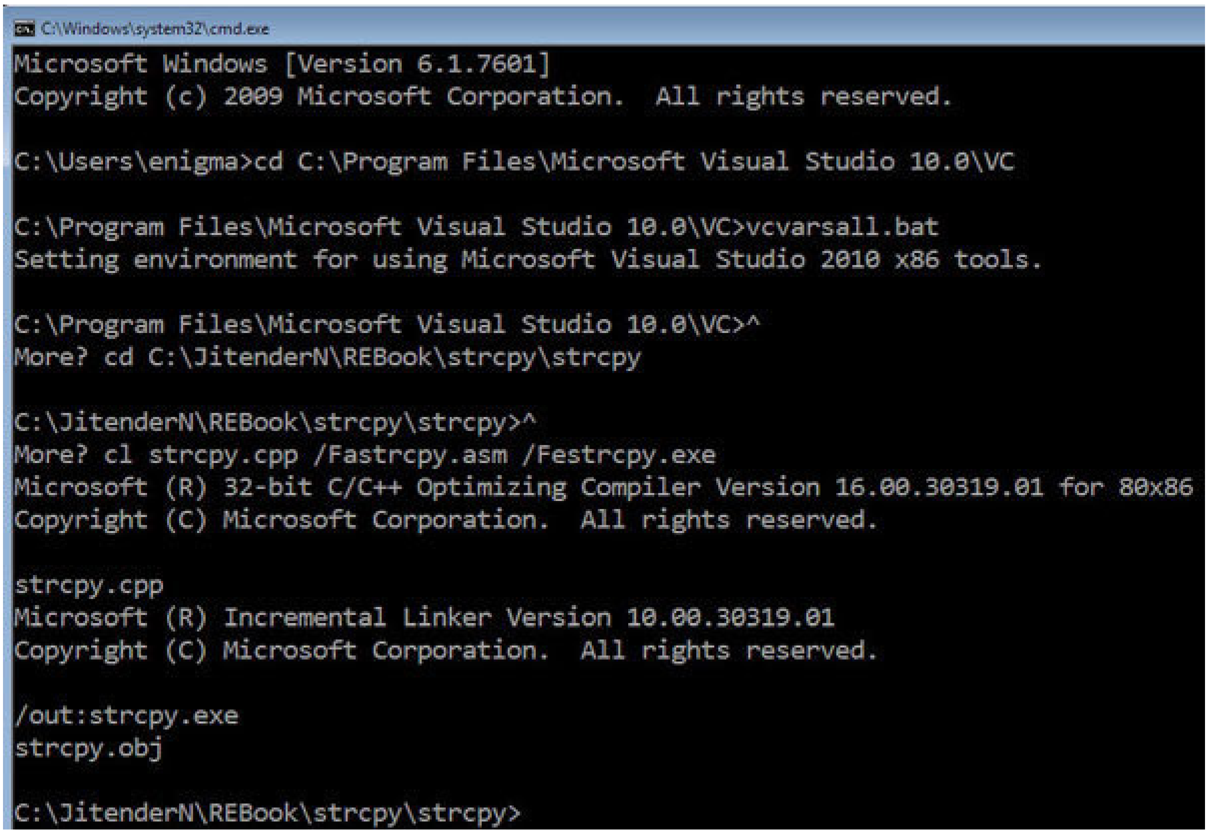
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:



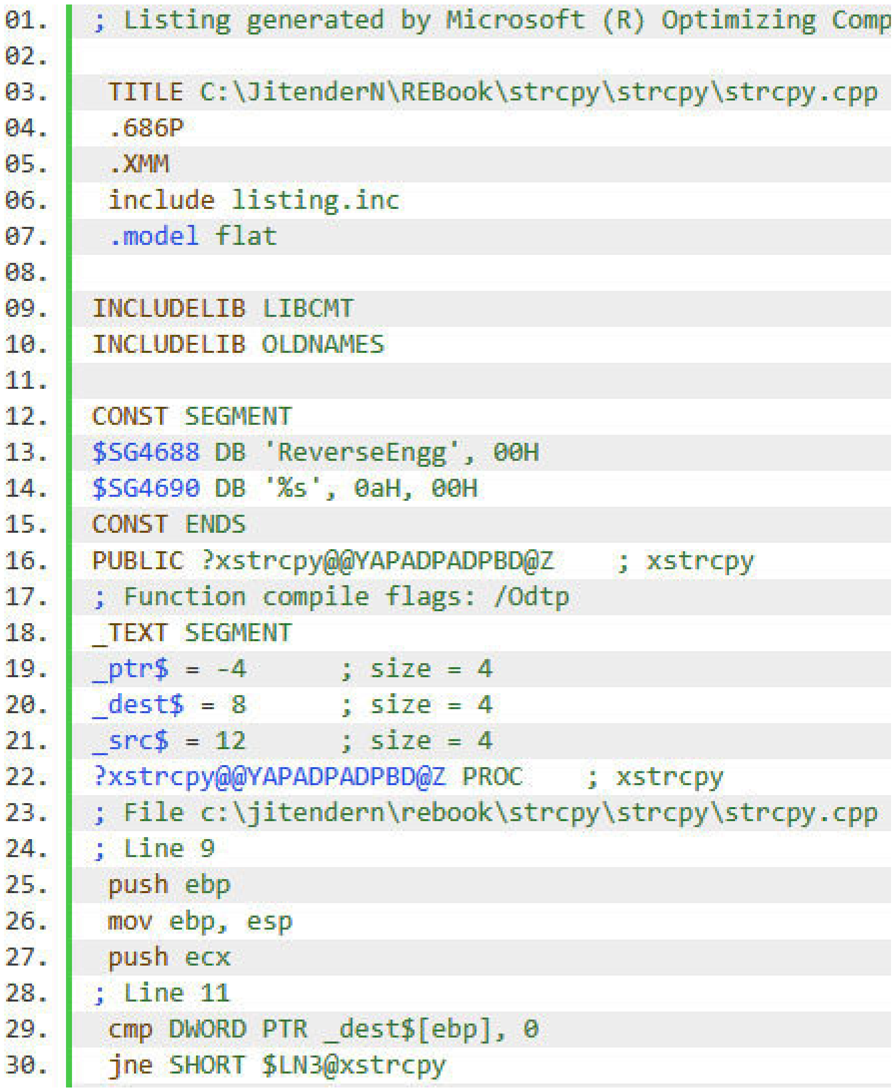
C:\Windows\system32\cmd.exe

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\strcpy\strcpy

C:\JitenderN\REBook\strcpy\strcpy>^
More? cl strcpy.cpp /Fastrcpy.asm /Festrcpy.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

strcpy.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:strcpy.exe
strcpy.obj

C:\JitenderN\REBook\strcpy\strcpy>
```

Figure 14.2: *Strcpy without Optimization*

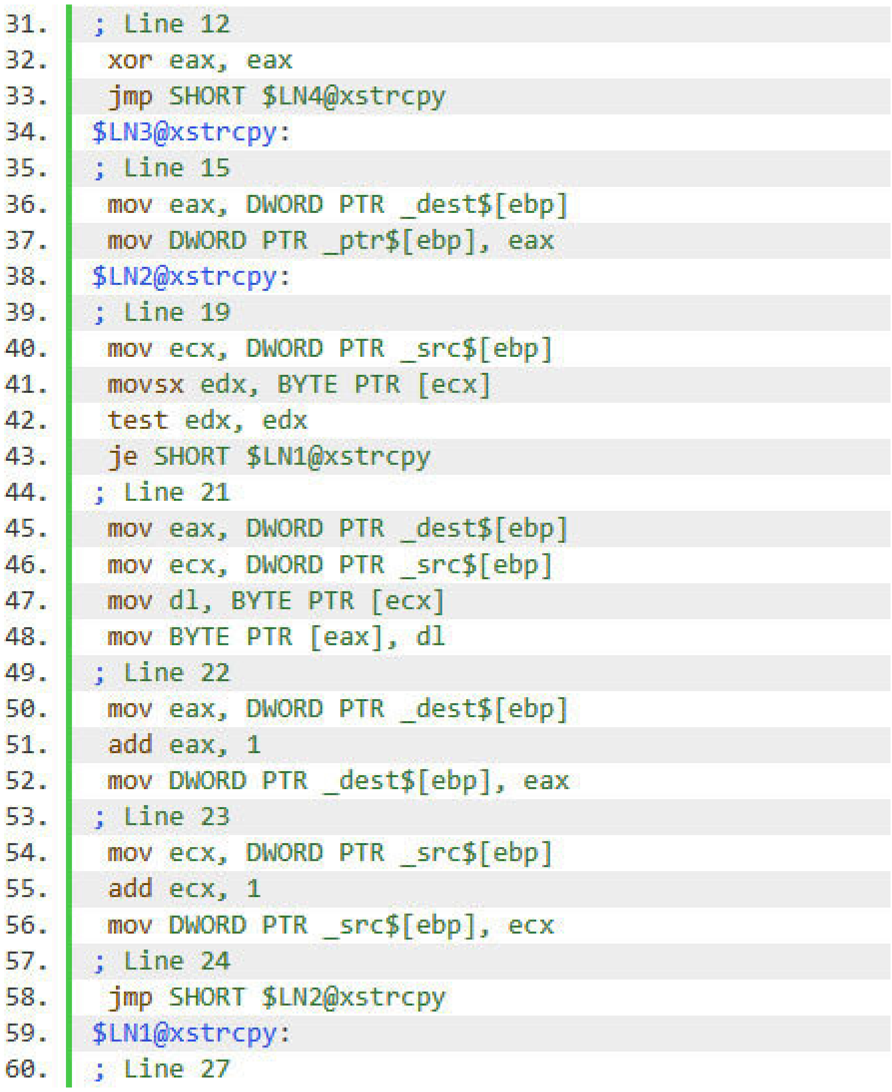
The compilation generates the EXE file and assembly code. Disable ASLR manually. To disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let's move on to the generated assembly listing:



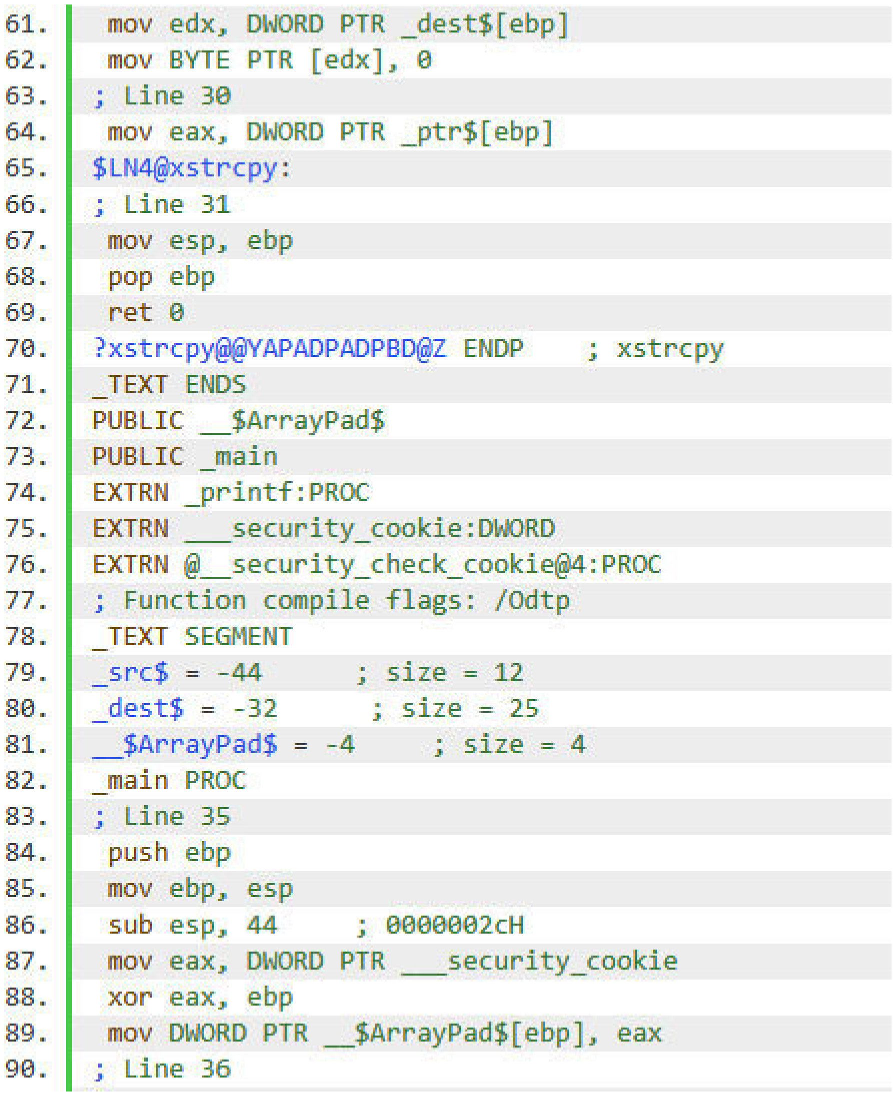
```
; Listing generated by Microsoft (R) Optimizing Comp

 TITLE C:\JitenderN\REBook\strcpy\strcpy\strcpy.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4688 DB 'ReverseEngg', 00H
$SG4690 DB '%s', 0aH, 00H
CONST ENDS
PUBLIC ?xstrcpy@@YAPADPADPBD@Z    ; xstrcpy
; Function compile flags: /Odtp
_TEXT SEGMENT
_ptr$ = -4        ; size = 4
_dest$ = 8        ; size = 4
_src$ = 12        ; size = 4
?xstrcpy@@YAPADPADPBD@Z PROC    ; xstrcpy
; File c:\jitendern\rebook\strcpy\strcpy\strcpy.cpp
; Line 9
 push ebp
 mov ebp, esp
 push ecx
; Line 11
 cmp DWORD PTR _dest$[ebp], 0
 jne SHORT $LN3@xstrcpy
```

Figure 14.3: *strcpy.asm-Part-1*



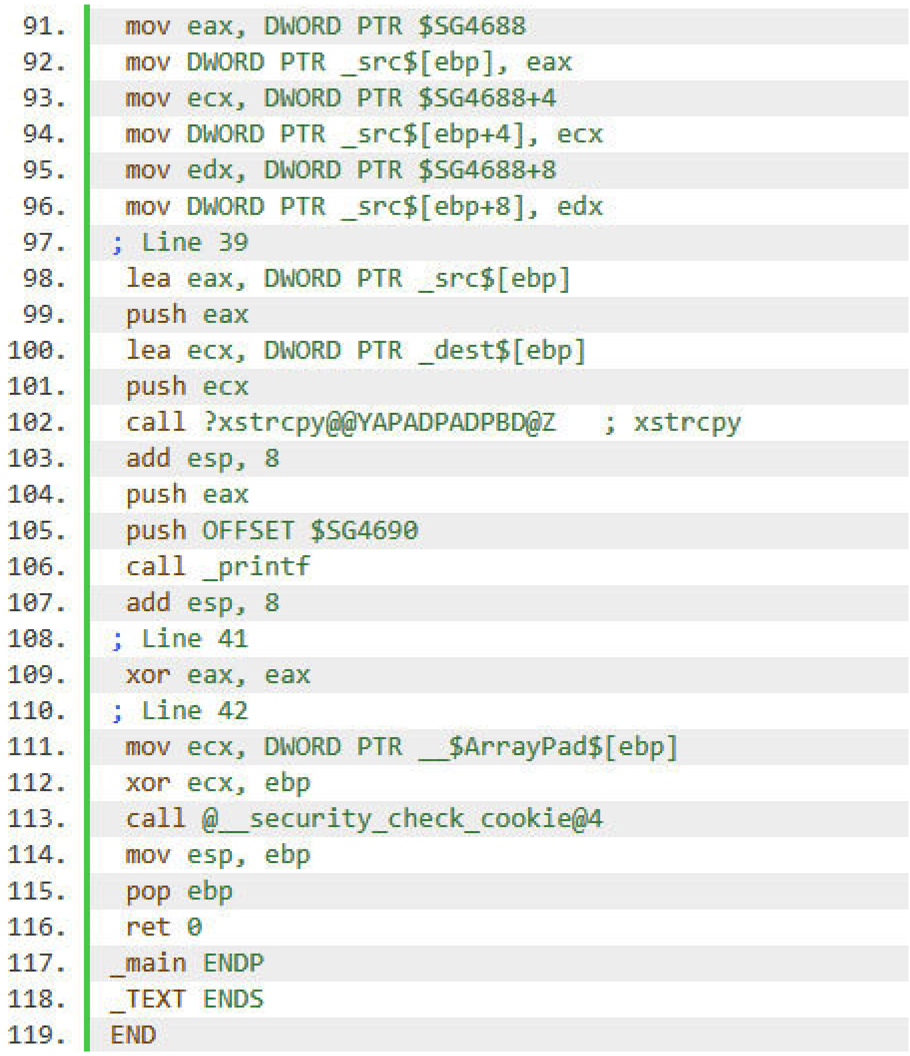
```
; Line 12
 xor eax, eax
 jmp SHORT $LN4@xstrcpy
$LN3@xstrcpy:
; Line 15
 mov eax, DWORD PTR _dest$[ebp]
 mov DWORD PTR _ptr$[ebp], eax
$LN2@xstrcpy:
; Line 19
 mov ecx, DWORD PTR _src$[ebp]
 movsx edx, BYTE PTR [ecx]
 test edx, edx
 je SHORT $LN1@xstrcpy
; Line 21
 mov eax, DWORD PTR _dest$[ebp]
 mov ecx, DWORD PTR _src$[ebp]
 mov dl, BYTE PTR [ecx]
 mov BYTE PTR [eax], dl
; Line 22
 mov eax, DWORD PTR _dest$[ebp]
 add eax, 1
 mov DWORD PTR _dest$[ebp], eax
; Line 23
 mov ecx, DWORD PTR _src$[ebp]
 add ecx, 1
 mov DWORD PTR _src$[ebp], ecx
; Line 24
 jmp SHORT $LN2@xstrcpy
$LN1@xstrcpy:
; Line 27
```

Figure 14.4: *strcpy.asm-Part-2*



```
 mov edx, DWORD PTR _dest$[ebp]
 mov BYTE PTR [edx], 0
; Line 30
 mov eax, DWORD PTR _ptr$[ebp]
$LN4@xstrcpy:
; Line 31
 mov esp, ebp
 pop ebp
 ret 0
?xstrcpy@@YAPADPADPBD@Z ENDP    ; xstrcpy
_TEXT ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Odtp
_TEXT SEGMENT
_src$ = -44       ; size = 12
_dest$ = -32       ; size = 25
__$ArrayPad$ = -4     ; size = 4
_main PROC
; Line 35
 push ebp
 mov ebp, esp
 sub esp, 44     ; 0000002cH
 mov eax, DWORD PTR ___security_cookie
 xor eax, ebp
 mov DWORD PTR __$ArrayPad$[ebp], eax
; Line 36
```

Figure 14.5: *strcpy.asm-Part-3*



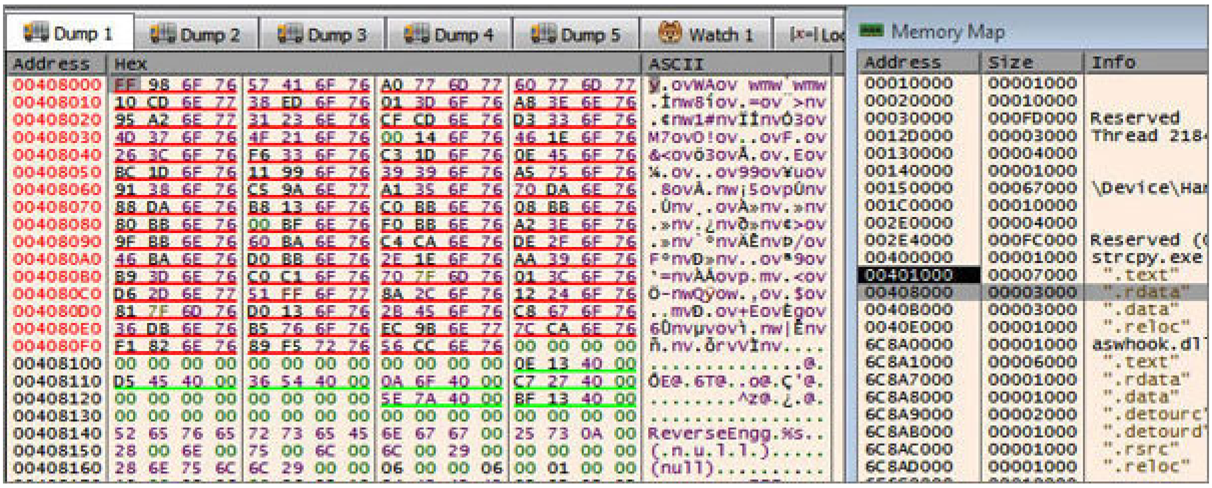
```
 91.    mov eax, DWORD PTR $SG4688
 92.    mov DWORD PTR _src$[ebp], eax
 93.    mov ecx, DWORD PTR $SG4688+4
 94.    mov DWORD PTR _src$[ebp+4], ecx
 95.    mov edx, DWORD PTR $SG4688+8
 96.    mov DWORD PTR _src$[ebp+8], edx
 97.  ; Line 39
 98.    lea eax, DWORD PTR _src$[ebp]
 99.    push eax
100.    lea ecx, DWORD PTR _dest$[ebp]
101.    push ecx
102.    call ?xstrcpy@@YAPADPADPBD@Z   ; xstrcpy
103.    add esp, 8
104.    push eax
105.    push OFFSET $SG4690
106.    call _printf
107.    add esp, 8
108.  ; Line 41
109.    xor eax, eax
110.  ; Line 42
111.    mov ecx, DWORD PTR __$ArrayPad$[ebp]
112.    xor ecx, ebp
113.    call @__security_check_cookie@4
114.    mov esp, ebp
115.    pop ebp
116.    ret 0
117.  _main ENDP
118.  _TEXT ENDS
119.  END
```

Figure 14.6: *strcpy.asm-Part-4*

▼Line 12-15

CONST SEGMENT
$SG4688 DB 'ReverseEngg', 00H
$SG4690 DB '%s', 0aH, 00H
CONST ENDS

Constant Segment defines two string constants, **$SG4688** and The linker renames **CONST SEGMENT** to **.rdata** (the code is placed in the **.code** segment, the constant strings are placed in **CONST** segment and if not, the constant is placed in the **.data** segment), which can be viewed in the memory dump using x32dbg, shown as follows:



```
Address  Hex                                              ASCII
00408000 FF 98 6F 76 57 41 6F 76 A0 77 6D 77 60 77 6D 77  ÿ.ovWAov wmw`wmw
00408010 10 CD 6E 77 38 ED 6F 76 01 3D 6F 76 A8 3E 6E 76  .Ínw8íov.=ov¨>nv
00408020 95 A2 6E 77 31 23 6E 76 CF CD 6E 76 D3 33 6F 76  .¢nw1#nvÏÍnvÓ3ov
00408030 4D 37 6F 76 4F 21 6F 76 00 14 6F 76 46 1E 6F 76  M7ovO!ov..ovF.ov
00408040 26 3C 6F 76 F6 33 6F 76 C3 1D 6F 76 0E 45 6F 76  &<ovö3ovÃ.ov.Eov
00408050 BC 1D 6F 76 11 99 6F 76 39 39 6F 76 A5 75 6F 76  ¼.ov..ov99ov¥uov
00408060 91 38 6F 76 C5 9A 6E 77 A1 35 6F 76 70 DA 6E 76  .8ovÅ.nw¡5ovpÚnv
00408070 88 DA 6E 76 B8 13 6F 76 C0 BB 6E 76 08 BB 6E 76  .Únv¸.ovÀ»nv.»nv
00408080 80 BB 6E 76 00 BF 6E 76 F0 BB 6E 76 A2 3E 6F 76  .»nv.¿nvð»nv¢>ov
00408090 9F BB 6E 76 60 BA 6E 76 C4 CA 6E 76 DE 2F 6F 76  .»nv`ºnvÄÊnvÞ/ov
004080A0 46 BA 6E 76 D0 BB 6E 76 2E 1E 6F 76 AA 39 6F 76  FºnvÐ»nv..ovª9ov
004080B0 B9 3D 6E 76 C0 C1 6F 76 70 7F 6D 76 01 3C 6F 76  ¹=nvÀÁovp.mv.<ov
004080C0 D6 2D 6E 77 51 FF 6F 77 8A 2C 6F 76 12 24 6F 76  Ö-nwQÿow.,ov.$ov
004080D0 81 7F 6D 76 D0 13 6F 76 2B 45 6F 76 C8 67 6F 76  ..mvÐ.ov+EovÈgov
004080E0 36 DB 6E 76 B5 76 6F 76 EC 9B 6E 77 7C CA 6E 76  6Ûnvµvovì.nw|Ênv
004080F0 F1 82 6E 76 89 F5 72 76 56 CC 6E 76 00 00 00 00  ñ.nv.õrvVÌnv....
00408100 00 00 00 00 00 00 00 00 00 00 00 00 0E 13 40 00  ..............@.
00408110 D5 45 40 00 36 54 40 00 0A 6F 40 00 C7 27 40 00  ÕE@.6T@..o@.Ç'@.
00408120 00 00 00 00 00 00 00 00 5E 7A 40 00 BF 13 40 00  ........^z@.¿.@.
00408130 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00408140 52 65 76 65 72 73 65 45 6E 67 67 00 25 73 0A 00  ReverseEngg.%s..
00408150 28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
00408160 28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
```

```
Address  Size     Info
00010000 00001000
00020000 00010000
00030000 000FD000 Reserved
0012D000 00003000 Thread 218
00130000 00004000
00140000 00001000
00150000 00067000 \Device\Ha
001C0000 00010000
002E0000 00004000
002E4000 000FC000 Reserved (
00400000 00001000 strcpy.exe
00401000 00007000  ".text"
00408000 00003000  ".rdata"
0040B000 00003000  ".data"
0040E000 00001000  ".reloc"
6C8A0000 00001000 aswhook.dl
6C8A1000 00006000  ".text"
6C8A7000 00001000  ".rdata"
6C8A8000 00001000  ".data"
6C8A9000 00002000  ".detourc
6C8AB000 00001000  ".detourd
6C8AC000 00001000  ".rsrc"
6C8AD000 00001000  ".reloc"
```

Figure 14.7: *.rdata*

$SG4688 and **$SG4690** are internal names given by the compiler to handle the string constant. DB defines the byte, which is the data type.

'ReverseEngg', 00H is the string data, which is null terminated ASCII string.

'%s', 0aH, 00H is also a string data.

By **CONST** the constant segment is ended.

First, we will take the main code starting with:

▼Line 72-73

```
PUBLIC __$ArrayPad$
PUBLIC _main
```

PUBLIC is the derivative which makes the **_main** procedure and the **__$ArrayPad$** macro public, which can be accessed by other modules:

▼Line 74-76

```
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD

EXTRN @__security_check_cookie@4:PROC
```

The **EXTRN** derivative declares the extern function, which is **printf** in our case. All functions begin with an underscore.

▼Line 78-81

```
_TEXT SEGMENT
_src$ = -44         ; size = 12
_dest$ = -32         ; size = 25
__$ArrayPad$ = -4       ; size = 4
```

This is the start of the **_TEXT** segment, where our **main** function code resides. After we have the different variable macro defined. The local variable on the stack frame can be accessed by adding **_$** to the **EBP** address.

▼Line 82

```
_main PROC
```

This is the start of the **main** procedure.

▼Line 83-89

```
; Line 35
push ebp
mov ebp, esp
sub esp, 44          ; 0000002cH
mov eax, DWORD PTR ___security_cookie
xor eax, ebp
mov DWORD PTR __$ArrayPad$[ebp], eax
```

Line 35 of the C/C++ code starts the **main** function:

```
int main() {
```

The ASM code starts with the **main** function prologue of **PUSH** and **MOV** instruction. The **SUB** instruction creates room for variables on the stack by subtracting 44 (0x2C) bytes from 44 bytes can be visualized as 4 bytes for the 12 bytes for the **src** array, and 28 bytes for the **dest** array. The **src** and **dest** array sizes have been round off to the multiple of 4 bytes.

▼Line 87-89

```
mov eax, DWORD PTR ___security_cookie
xor eax, ebp
mov DWORD PTR __$ArrayPad$[ebp], eax
```

We have already discussed the concept of a stack cookie. In the preceding instruction, the stack cookie is moved from **EAX** to XOR with **EBP** and the resultant XOR value is placed on the stack. This value is stored on the stack right above the old This value will be validated before the **main** function epilogue to check the buffer overflow condition. Now, let's move on to the next instructions:

▼Line 90-96

```
; Line 36
mov eax, DWORD PTR $SG4688
mov DWORD PTR _src$[ebp], eax
mov ecx, DWORD PTR $SG4688+4
mov DWORD PTR _src$[ebp+4], ecx
mov edx, DWORD PTR $SG4688+8
mov DWORD PTR _src$[ebp+8], edx
```

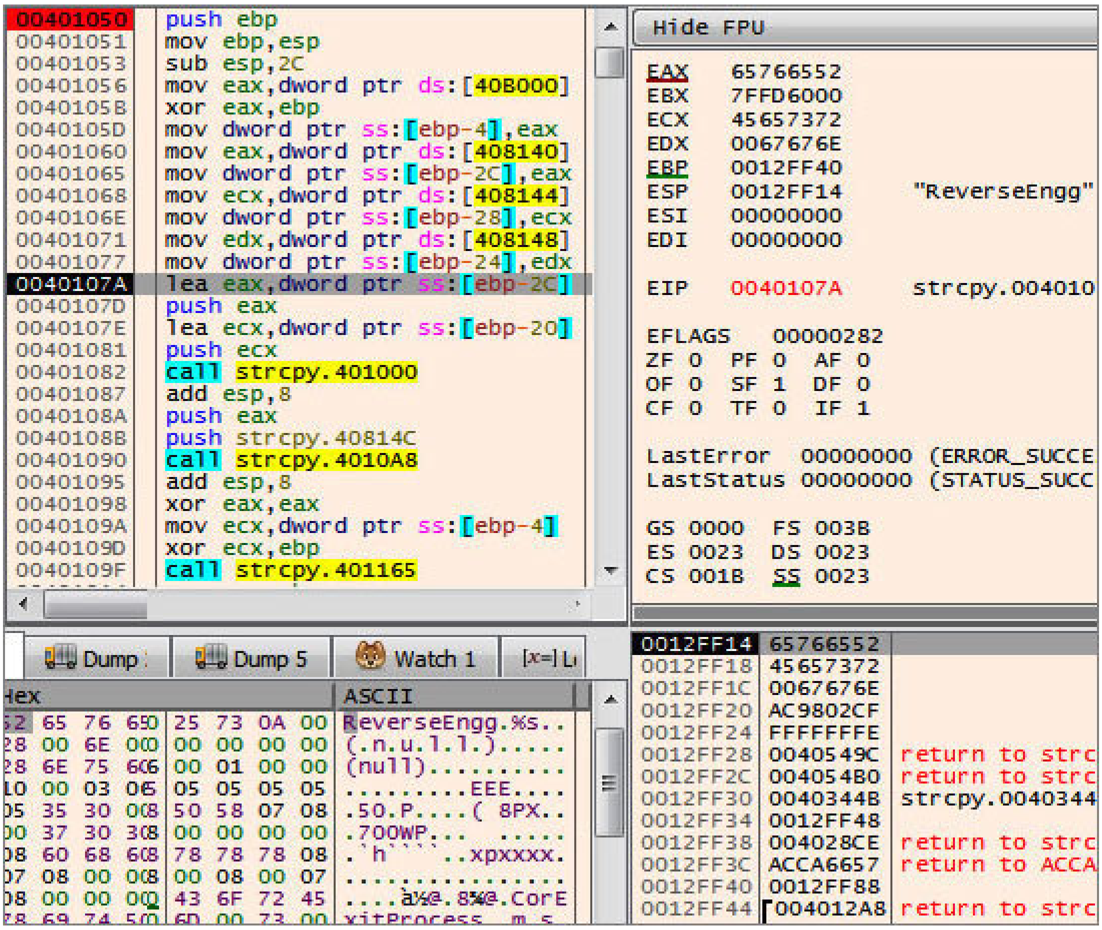
Line 36 of the C/C++ code is:

```
char src[] = "ReverseEngg";
```

In the ASM code, we can see that the string **$SG4688** is moved to the stack in three steps and with six **MOV** instructions. The first two **MOV** instructions move the first 4 bytes of string ("Reve") to **EAX** and then from it is pushed to the stack at the **[EBP - 0x2C]** location. Similarly, the next 4 bytes of string ("rseE") are moved to **ECX** and then from it is pushed to the stack at the **[EBP - 0x28]** location. Then in the remaining bytes, padded with NULL ("ngg" + "0x00") are moved to **EDX** and then from **EDX** to the stack at the **[EBP - 0x24]** location. The stack state after the execution of the preceding instruction in x32dbg will be as follows:

[ESP] 0012FF14 65766552 [EBP-0x2C] "eveR" is pushed here, in little endian
[ESP+0x04] 0012FF18 45657372 [EBP-0x28] "Eesr" is pushed here, little endian
[ESP+0x08] 0012FF1C 0067676E [EBP-0x24] " ggn" is pushed here, little endian
[ESP+0x0C] 0012FF20 AC9802CF [EBP-0x20] JUNK HERE
[ESP+0x10] 0012FF24 FFFFFFFE [EBP-0x1C] JUNK HERE
[ESP+0x14] 0012FF28 0040549C [EBP-0x18] JUNK HERE
[ESP+0x18] 0012FF2C 004054B0 [EBP-0x14] JUNK HERE
[ESP+0x1C] 0012FF30 0040344B [EBP-0x10] JUNK HERE
[ESP+0x20] 0012FF34 0012FF48 [EBP-0x0C] JUNK HERE
[ESP+0x24] 0012FF38 004028CE [EBP-0x08] JUNK HERE
[ESP+0x28] 0012FF3C ACCA6657 [EBP-0x04] XOR of stack cookie and EBP stored here
[ESP+0x2C] 0012FF40 0012FF88 [EBP] EBP of earlier stack frame stored here

[ESP+0x30] 0012FF44 004012A8 [EBP+0x04] return to 0x004012A8 from 0x00401050



```
00401050  push ebp
00401051  mov ebp,esp
00401053  sub esp,2C
00401056  mov eax,dword ptr ds:[40B000]
0040105B  xor eax,ebp
0040105D  mov dword ptr ss:[ebp-4],eax
00401060  mov eax,dword ptr ds:[408140]
00401065  mov dword ptr ss:[ebp-2C],eax
00401068  mov ecx,dword ptr ds:[408144]
0040106E  mov dword ptr ss:[ebp-28],ecx
00401071  mov edx,dword ptr ds:[408148]
00401077  mov dword ptr ss:[ebp-24],edx
0040107A  lea eax,dword ptr ss:[ebp-2C]
0040107D  push eax
0040107E  lea ecx,dword ptr ss:[ebp-20]
00401081  push ecx
00401082  call strcpy.401000
00401087  add esp,8
0040108A  push eax
0040108B  push strcpy.40814C
00401090  call strcpy.4010A8
00401095  add esp,8
00401098  xor eax,eax
0040109A  mov ecx,dword ptr ss:[ebp-4]
0040109D  xor ecx,ebp
0040109F  call strcpy.401165
```

```
Hide FPU

EAX   65766552
EBX   7FFD6000
ECX   45657372
EDX   0067676E
EBP   0012FF40
ESP   0012FF14      "ReverseEngg"
ESI   00000000
EDI   00000000

EIP   0040107A      strcpy.004010

EFLAGS    00000282
ZF 0  PF 0  AF 0
OF 0  SF 1  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC9802CF
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40  0012FF88
0012FF44  004012A8  return to strc
```

```
Hex                                ASCII
52 65 76 65 | 25 73 0A 00 | ReverseEngg.%s..
28 00 6E 00 | 00 00 00 00 | (.n.u.l.l.).....
28 6E 75 6C | 00 01 00 00 | (null)..........
10 00 03 06 | 05 05 05 05 | .........EEE....
05 35 30 08 | 50 58 07 08 | .50.P....( 8PX..
00 37 30 38 | 00 00 00 00 | .700WP...  .....
08 60 68 68 | 78 78 78 08 | .`h```  ..xpxxxx.
07 08 00 08 | 00 08 00 07 | ................
08 00 00 00 | 43 6F 72 45 | ....à¼@.8¼@.CorE
```

Figure 14.8: $SG4688 is moved to the stack

▼Line 97-102

```
; Line 39
lea eax, DWORD PTR _src$[ebp]
push eax
lea ecx, DWORD PTR _dest$[ebp]
push ecx
call ?xstrcpy@@YAPADPADPBD@Z    ; xstrcpy
```

Line 39 of the C/C++ code is:
printf("%s\n", xstrcpy(dest, src));

In the ASM code, before making the **printf** function call, we need to push the return value of the **xstrcpy** function and the string **$SG4690** onto the stack. To evaluate the return value of the **xstrcpy** function, arguments to the function are pushed onto the stack. So, the first LEA (Load Effective Address) loads the address on the stack (which is **[EBP - 0x2C],** pointing to the source array in the **EAX** register. From it is further pushed onto the stack. Similarly, the second LEA instruction loads the address on the stack (which is **[EBP -** pointing to the destination array in the **ECX** register. From it is further pushed onto the stack. As the destination array is not initialized in the C/C++ code, it is pointing to the uninitialized memory location (which is **[EBP -** the on stack. Once both the arguments to the **xstrcpy** function are pushed onto the stack, a call to the **xstrcpy** function is made. The stack state before the call to the **xstrcpy** function is as follows:

[ESP] 0012FF0C 0012FF20 [EBP-0x34] 2nd arg to xstrcpy, ptr to dest array
[ESP+0x04] 0012FF10 0012FF14 [EBP-0x30] 1st arg to xstrcpy(),ptr to src array
[ESP+0x08] 0012FF14 65766552 [EBP-0x2C] “eveR” is pushed here, in little endian
[ESP+0x0C] 0012FF18 45657372 [EBP-0x28] “Eesr” is pushed here, little endian
[ESP+0x10] 0012FF1C 0067676E [EBP-0x24] “ ggn” is pushed here, little endian

[ESP+0x14] 0012FF20 AC9802CF [EBP-0x20] dest array start, uninitialized JUNK

[ESP+0x18] 0012FF24 FFFFFFFE [EBP-0x1C] JUNK HERE

[ESP+0x1C] 0012FF28 0040549C [EBP-0x18] JUNK HERE

[ESP+0x20] 0012FF2C 004054B0 [EBP-0x14] JUNK HERE

[ESP+0x24] 0012FF30 0040344B [EBP-0x10] JUNK HERE

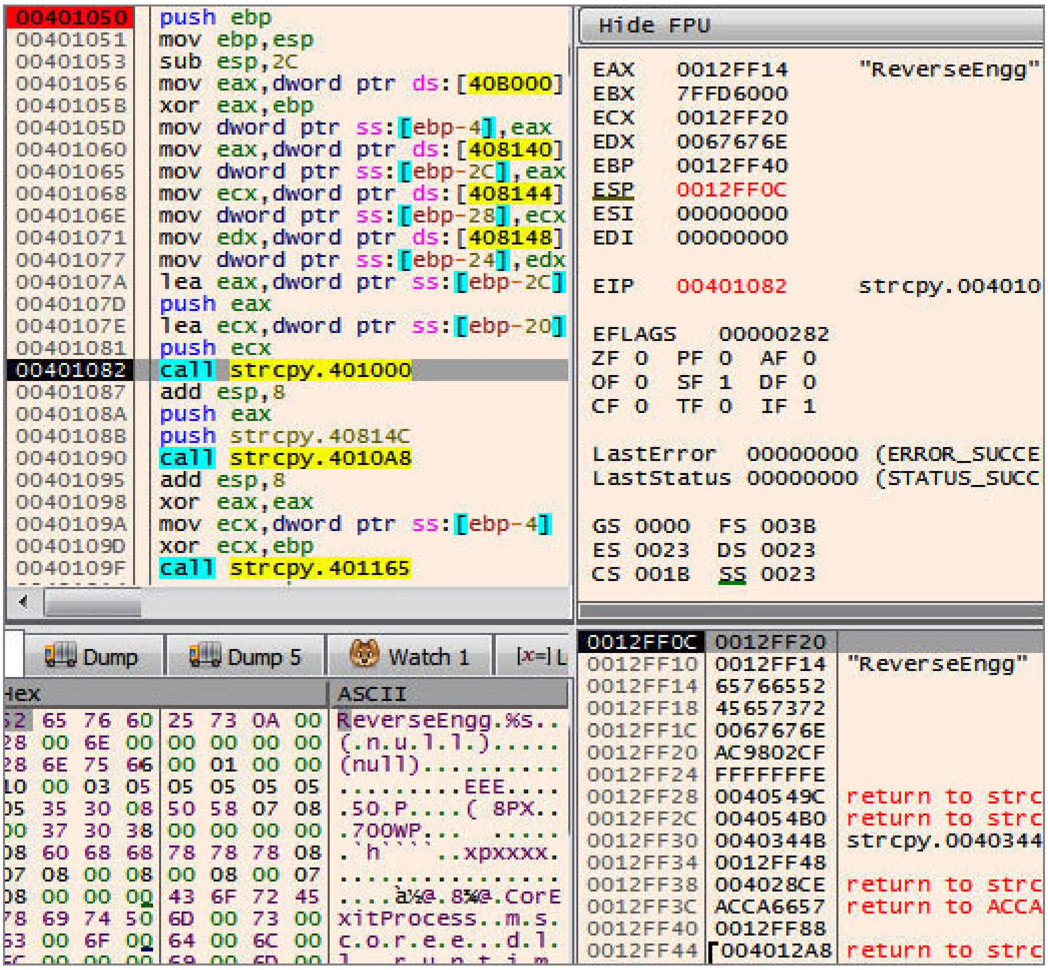
[ESP+0x28] 0012FF34 0012FF48 [EBP-0x0C] JUNK HERE

[ESP+0x2C] 0012FF38 004028CE [EBP-0x08] JUNK HERE

[ESP+0x30] 0012FF3C ACCA6657 [EBP-0x04] XOR of stack cookie and EBP stored here

[ESP+0x34] 0012FF40 0012FF88 [EBP] EBP of earlier stack frame stored here

[ESP+0x38] 0012FF44 004012A8 [EBP+0x04] return to 0x004012A8 from 0x00401050



```
00401050  push ebp
00401051  mov ebp,esp
00401053  sub esp,2C
00401056  mov eax,dword ptr ds:[408000]
0040105B  xor eax,ebp
0040105D  mov dword ptr ss:[ebp-4],eax
00401060  mov eax,dword ptr ds:[408140]
00401065  mov dword ptr ss:[ebp-2C],eax
00401068  mov ecx,dword ptr ds:[408144]
0040106E  mov dword ptr ss:[ebp-28],ecx
00401071  mov edx,dword ptr ds:[408148]
00401077  mov dword ptr ss:[ebp-24],edx
0040107A  lea eax,dword ptr ss:[ebp-2C]
0040107D  push eax
0040107E  lea ecx,dword ptr ss:[ebp-20]
00401081  push ecx
00401082  call strcpy.401000
00401087  add esp,8
0040108A  push eax
0040108B  push strcpy.40814C
00401090  call strcpy.4010A8
00401095  add esp,8
00401098  xor eax,eax
0040109A  mov ecx,dword ptr ss:[ebp-4]
0040109D  xor ecx,ebp
0040109F  call strcpy.401165
```

```
EAX   0012FF14     "ReverseEngg"
EBX   7FFD6000
ECX   0012FF20
EDX   0067676E
EBP   0012FF40
ESP   0012FF0C
ESI   00000000
EDI   00000000

EIP   00401082     strcpy.004010

EFLAGS   00000282
ZF 0  PF 0  AF 0
OF 0  SF 1  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
2 65 76 60 | 25 73 0A 00 | ReverseEngg.%s..
8 00 6E 00 | 00 00 00 00 | (.n.u.l.l.).....
8 6E 75 66 | 00 01 00 00 | (null)..........
0 00 03 05 | 05 05 05 05 | ...........EEE....
5 35 30 08 | 50 58 07 08 | .50.P....( 8PX..
0 37 30 38 | 00 00 00 00 | .700WP...  .....
8 60 68 68 | 78 78 78 08 | .`h````..xpxxxx.
7 08 00 08 | 00 08 00 07 | ................
8 00 00 00 | 43 6F 72 45 | ....à½@.8½@.CorE
8 69 74 50 | 6D 00 73 00 | xitProcess..m.s.
3 00 6F 00 | 64 00 6C 00 | c.o.r.e.e...d.l.
```

```
0012FF0C  0012FF20
0012FF10  0012FF14  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC9802CF
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40  0012FF88
0012FF44  004012A8  return to strc
```

Figure 14.9: *Before call to xstrcpy*

The following is the start of the **xstrcpy** function:

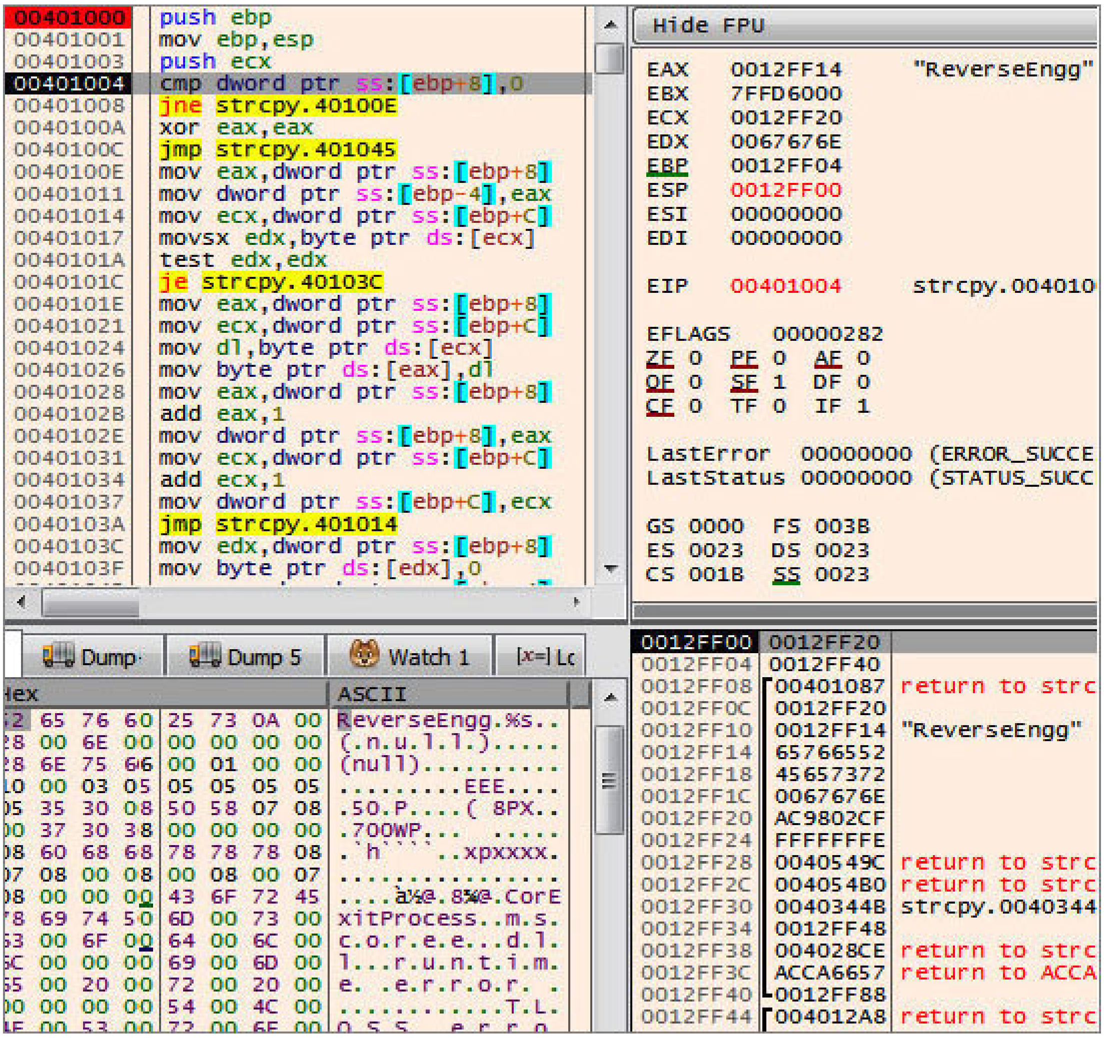
▼Line 24-27

```
; Line 9
push ebp


mov ebp, esp
push ecx
```

The **xstrcpy** function prologue is called. The **ECX** register has the pointer to the **dest** array, so it is saved onto the stack by pushing the **ECX** register. The stack state after pushing the **ECX** register is as follows. From this point onwards, the two stack frames have been differentiated with **EBP** marked with superscript, that is, **EBP** for **main** and **xstrcpy.**

to dest
[ESP+0x04] EBP of main() stack frame
[ESP+0x08] return to main()
[ESP+0x0C] arg to xstrcpy
[ESP+0x10] arg to xstrcpy
[ESP+0x14] “eveR” is pushed here
[ESP+0x18] “Eesr” is pushed here
[ESP+0x1C] “ ggn” is pushed here
[ESP+0x20] uninitialized dest array
[ESP+0x24] JUNK HERE

[ESP+0x28] JUNK HERE
[ESP+0x2C] JUNK HERE
[ESP+0x30] JUNK HERE
[ESP+0x34] JUNK HERE
[ESP+0x38] JUNK HERE
[ESP+0x3C] XOR of stack cookie and EBP
[ESP+0x40] of earlier stack frame
[ESP+0x44] return to 0x004012A8 from 0x00401050



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
Hide FPU

EAX  0012FF14     "ReverseEngg"
EBX  7FFD6000
ECX  0012FF20
EDX  0067676E
EBP  0012FF04
ESP  0012FF00
ESI  00000000
EDI  00000000

EIP  00401004     strcpy.004010

EFLAGS    00000282
ZF 0  PE 0  AE 0
OF 0  SF 1  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCE
LastStatus 00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                  ASCII
2 65 76 60 | 25 73 0A 00   ReverseEngg.%s..
8 00 6E 00 | 00 00 00 00   (.n.u.l.l.).....
8 6E 75 66 | 00 01 00 00   (null)..........
0 00 03 05 | 05 05 05 05   ..........EEE....
5 35 30 08 | 50 58 07 08   .50.P....( 8PX..
0 37 30 38 | 00 00 00 00   .700WP...  .....
8 60 68 68 | 78 78 78 08   . h`  ..xpxxxx.
7 08 00 08 | 00 08 00 07   ................
8 00 00 00 | 43 6F 72 45   ....à%@.8%@.CorE
8 69 74 50 | 6D 00 73 00   xitProcess..m.s.
3 00 6F 00 | 64 00 6C 00   c.o.r.e.e...d.l.
C 00 00 00 | 69 00 6D 00   l...r.u.n.t.i.m.
5 00 20 00 | 72 00 20 00   e. .e.r.r.o.r. .
0 00 00 00 | 54 00 4C 00   ............T.L.
```

```
0012FF00  0012FF20
0012FF04  0012FF40
0012FF08  00401087  return to strc
0012FF0C  0012FF20
0012FF10  0012FF14  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC9802CF
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40  0012FF88
0012FF44  004012A8  return to strc
```

Figure 14.10: Stack state after pushing ECX

▼Line 28-30

```
; Line 11
cmp DWORD PTR _dest$[ebp], 0
jne SHORT $LN3@xstrcpy
```

Line 11 of the C/C++ code is:

```
if (dest == NULL)
```

The ASM code compares the value at with NULL. A jump will take place when the value at is not equal to NULL. As memory is already allocated to the **dest** array at which is not equal to NULL, so the jump will take place to the **$LN3@xstrcpy** label. The stack state after the jump instruction will be the same as earlier.

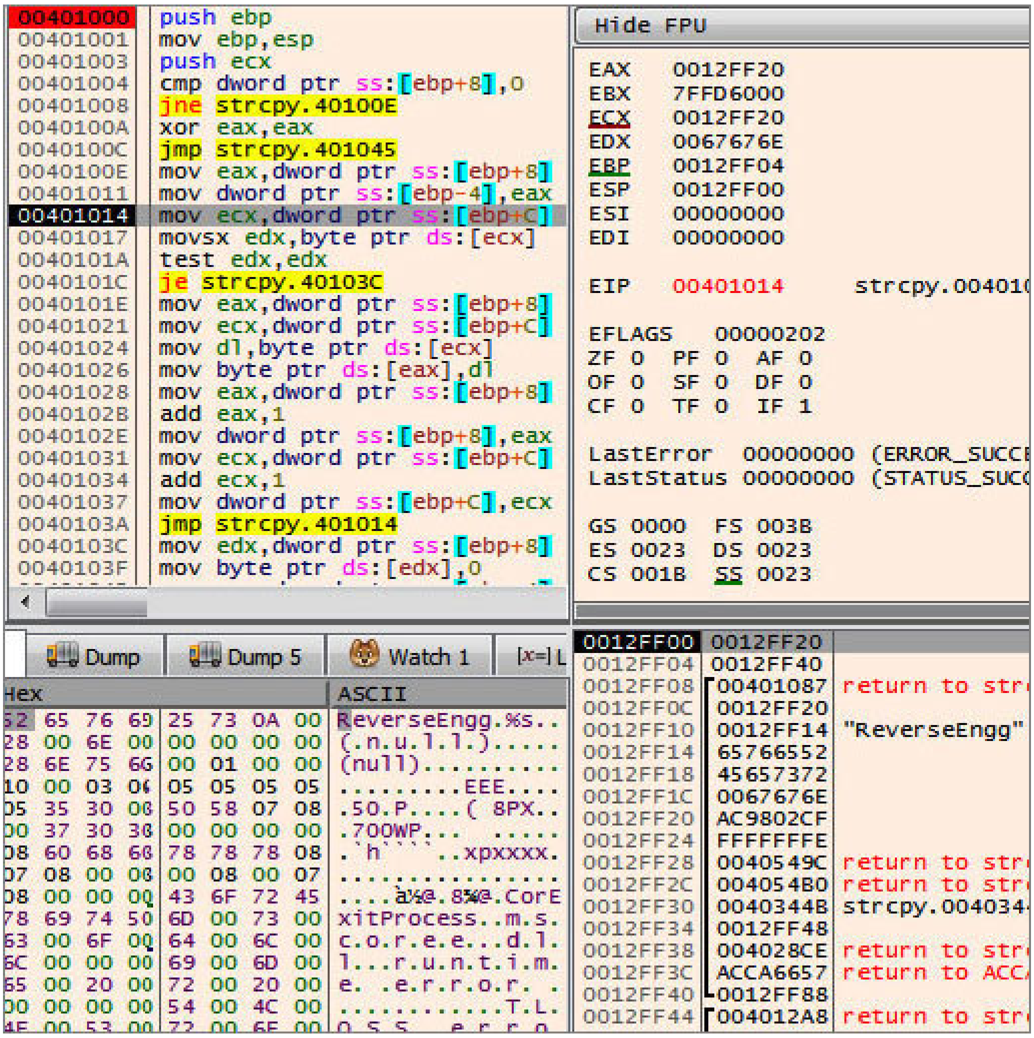
▼Line 34-37

```
$LN3@xstrcpy:
; Line 15
mov eax, DWORD PTR _dest$[ebp]
mov DWORD PTR _ptr$[ebp], eax
```

Line 11 of the C/C++ code is:

```
char *ptr = dest;
```

In the ASM code it is just taking the pointer to the **dest** array into EAX and then pushing it on to the stack at If you remember, we saved ECX at the start of the **xstrcpy** function on-to the stack at the same location ECX was having the same pointer to the **dest** array. So basically in these two instructions we are overwriting with the same value. So the stack state will be the same as earlier:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
EAX   0012FF20
EBX   7FFD6000
ECX   0012FF20
EDX   0067676E
EBP   0012FF04
ESP   0012FF00
ESI   00000000
EDI   00000000

EIP   00401014      strcpy.004010

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCC
LastStatus  00000000 (STATUS_SUC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                 ASCII
52 65 76 69 25 73 0A 00 ReverseEngg.%s..
28 00 6E 00 00 00 00 00 (.n.u.l.l.).....
28 6E 75 6G 00 01 00 00 (null)..........
10 00 03 06 05 05 05 05 ..........EEE....
05 35 30 08 50 58 07 08 .50.P....( 8PX..
00 37 30 38 00 00 00 00 .700WP...  .....
08 60 68 68 78 78 78 08 .`h````..xpxxxx.
07 08 00 08 00 08 00 07 ................
08 00 00 00 43 6F 72 45 ....à½@.8½@.CorE
78 69 74 50 6D 00 73 00 xitProcess..m.s.
63 00 6F 00 64 00 6C 00 c.o.r.e.e...d.l.
6C 00 00 00 69 00 6D 00 l...r.u.n.t.i.m.
65 00 20 00 72 00 20 00 e. .e.r.r.o.r. .
00 00 00 00 54 00 4C 00 ............T.L.
```

```
0012FF00  0012FF20
0012FF04  0012FF40
0012FF08  00401087  return to str
0012FF0C  0012FF20
0012FF10  0012FF14  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC9802CF
0012FF24  FFFFFFFE
0012FF28  0040549C  return to str
0012FF2C  004054B0  return to str
0012FF30  0040344B  strcpy.004034
0012FF34  0012FF48
0012FF38  004028CE  return to str
0012FF3C  ACCA6657  return to ACC
0012FF40  0012FF88
0012FF44  004012A8  return to str
```

Figure 14.11: *Stack state after line 37*

▼Line 38-43

$LN2@xstrcpy:

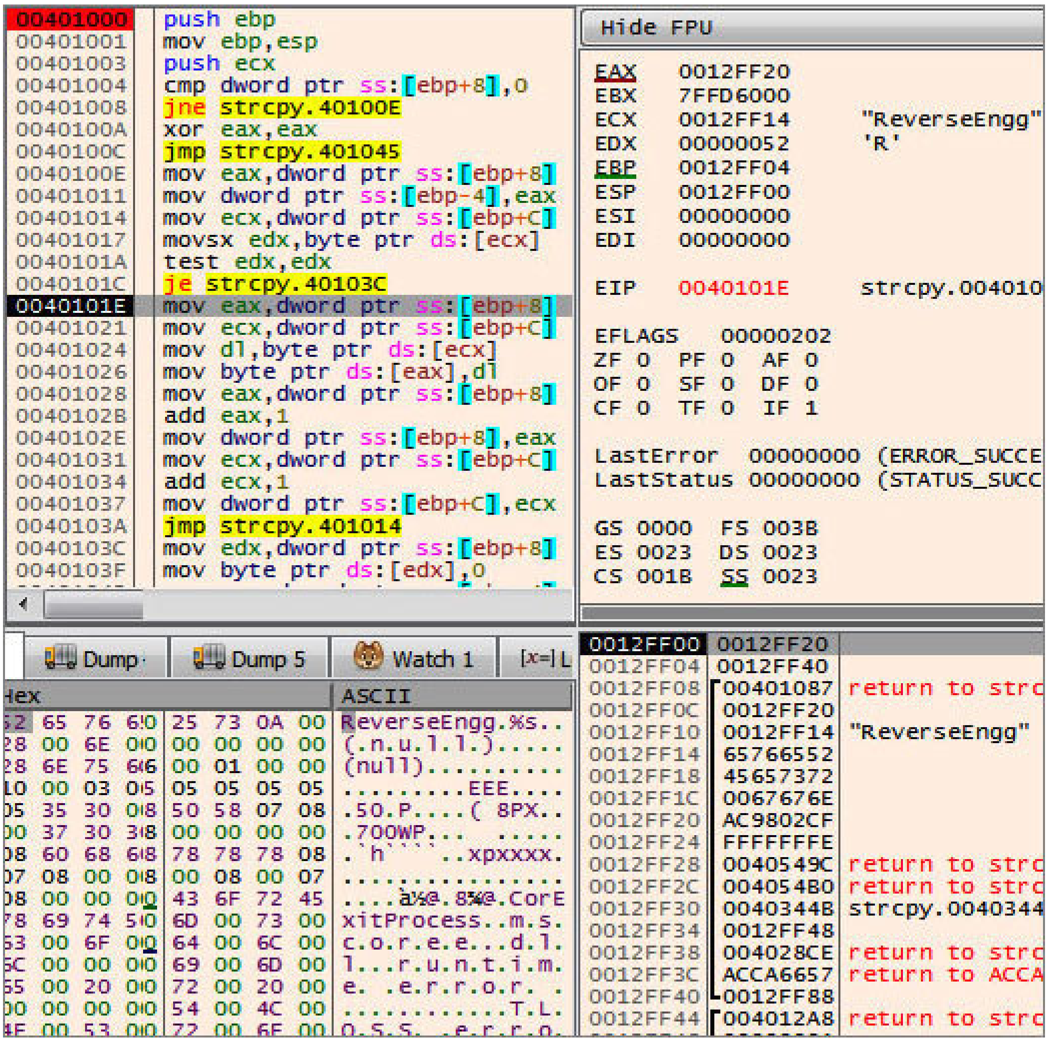
```
; Line 19
mov ecx, DWORD PTR _src$[ebp]
movsx edx, BYTE PTR [ecx]
test edx, edx
je SHORT $LN1@xstrcpy
```

Line 19 of the C/C++ code is:

```
while (*src != '\0')
```

In the ASM code, the **MOV** instruction is moving the pointer to the **src** array in the **ECX** register. The next **MOV** instruction moves the first byte of the **src** array into making (ascii “R”).

The **TEST** instruction will perform an **AND** operation of **EDX** with itself, resulting in a non-zero value in **EDX** and ZF=0. So, a jump to the label **$LN1@xstrcpy** will not take place. The instruction pointer will move to the next instruction. The stack state will be the same as earlier:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
Hide FPU

EAX   0012FF20
EBX   7FFD6000
ECX   0012FF14      "ReverseEngg"
EDX   00000052      'R'
EBP   0012FF04
ESP   0012FF00
ESI   00000000
EDI   00000000

EIP   0040101E      strcpy.004010

EFLAGS   00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCE
LastStatus 00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                   ASCII
2  65 76 60  25 73 0A 00  ReverseEngg.%s..
8  00 6E 00  00 00 00 00  (.n.u.l.l.).....
8  6E 75 66  00 01 00 00  (null)..........
0  00 03 05  05 05 05 05  .........EEE....
5  35 30 08  50 58 07 08  .50.P....( 8PX..
0  37 30 38  00 00 00 00  .700WP...  .....
8  60 68 68  78 78 78 08  .`h```` ..xpxxxx.
7  08 00 08  00 08 00 07  ................
8  00 00 00  43 6F 72 45  ....à¾@.8¾@.CorE
8  69 74 50  6D 00 73 00  xitProcess..m.s.
3  00 6F 00  64 00 6C 00  c.o.r.e.e...d.l.
C  00 00 00  69 00 6D 00  l...r.u.n.t.i.m.
5  00 20 00  72 00 20 00  e. .e.r.r.o.r. .
0  00 00 00  54 00 4C 00  ............T.L.
```

```
0012FF00  0012FF20
0012FF04  0012FF40
0012FF08  00401087  return to strc
0012FF0C  0012FF20
0012FF10  0012FF14  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC9802CF
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40  0012FF88
0012FF44  004012A8  return to strc
```

Figure 14.12: *Stack state after je*

▼Line 44-48

; Line 21

mov eax, DWORD PTR _dest$[ebp]
mov ecx, DWORD PTR _src$[ebp]
mov dl, BYTE PTR [ecx]
mov BYTE PTR [eax], dl

Line 21 of the C/C++ code is:

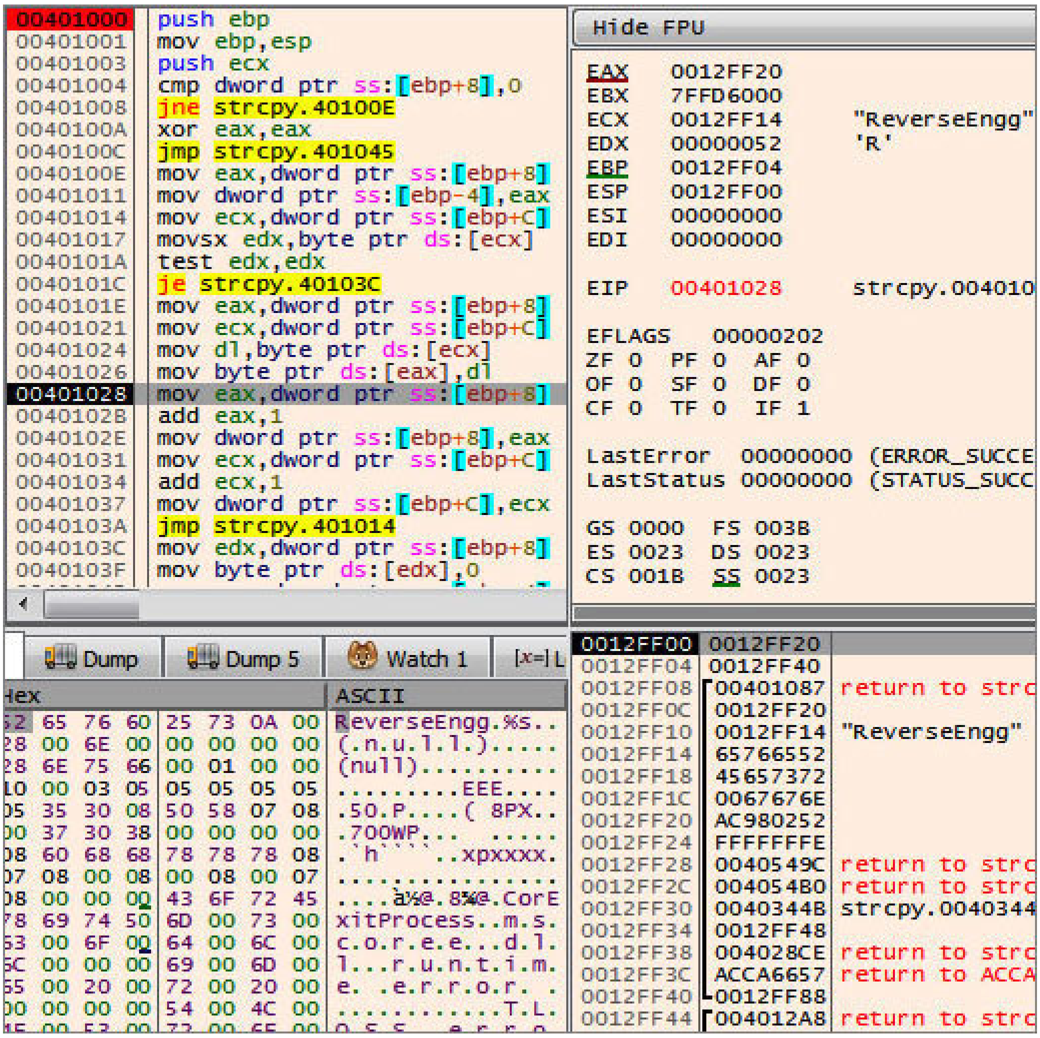
*dest = *src;

As this ASM code is non-optimized, we will come across many instructions which are duplicating operations. We can see that the **MOV** instruction is again moving the pointer from the **dest** array to **EAX** register and pointer to **src** array is moved to the **ECX** register. Now, in the remaining two **MOV** instructions, we are copying the first byte from the **src** array to the **dest** array using the **DL** register. This is done by copying a byte from the memory pointed by the **ECX** register (which is pointing to the src array) to the **DL** register and then from the **DL** register to the memory pointed by the **EAX** register (which is pointing to the **dest** array). The stack state after these instructions will be:

to dest
[ESP+0x04] EBP of main() stack frame
[ESP+0x08] return to main()
[ESP+0x0C] arg to xstrcpy
[ESP+0x10] arg to xstrcpy

[ESP+0x14] “eveR” is pushed here
[ESP+0x18] “Eesr” is pushed here
[ESP+0x1C] “ ggn” is pushed here
[ESP+0x20] only “R” is copied here at dest array
[ESP+0x24] JUNK HERE
[ESP+0x28] JUNK HERE
[ESP+0x2C] JUNK HERE
[ESP+0x30] JUNK HERE
[ESP+0x34] JUNK HERE
[ESP+0x38] JUNK HERE

[ESP+0x3C] XOR of stack cookie and EBP
[ESP+0x40] of earlier stack frame stored here
[ESP+0x44] return to 0x004012A8 from 0x00401050



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
EAX   0012FF20
EBX   7FFD6000
ECX   0012FF14      "ReverseEngg"
EDX   00000052      'R'
EBP   0012FF04
ESP   0012FF00
ESI   00000000
EDI   00000000

EIP   00401028      strcpy.004010

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                        ASCII
2 65 76 60 25 73 0A 00     ReverseEngg.%s..
8 00 6E 00 00 00 00 00     (.n.u.l.l.).....
8 6E 75 66 00 01 00 00     (null)..........
0 00 03 05 05 05 05 05     ..........EEE....
5 35 30 08 50 58 07 08     .50.P....( 8PX..
0 37 30 38 00 00 00 00     .700WP...  .....
8 60 68 68 78 78 78 08     .`h````..xpxxxx.
7 08 00 08 00 08 00 07     ................
8 00 00 00 43 6F 72 45     ....à¾@.8¾@.CorE
8 69 74 50 6D 00 73 00     xitProcess..m.s.
3 00 6F 00 64 00 6C 00     c.o.r.e.e...d.l.
C 00 00 00 69 00 6D 00     l...r.u.n.t.i.m.
5 00 20 00 72 00 20 00     e. .e.r.r.o.r. .
0 00 00 00 54 00 4C 00     ............T.L.
```

```
0012FF00  0012FF20
0012FF04  0012FF40
0012FF08 [00401087  return to strc
0012FF0C  0012FF20
0012FF10  0012FF14  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  AC980252
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40 [0012FF88
0012FF44 [004012A8  return to strc
```

Figure 14.13: *Stack state after line 48*

▼Line 49-52

```
; Line 22
mov eax, DWORD PTR _dest$[ebp]
add eax, 1
```

```
mov DWORD PTR _dest$[ebp], eax
```

Line 22 of the C/C++ code is:

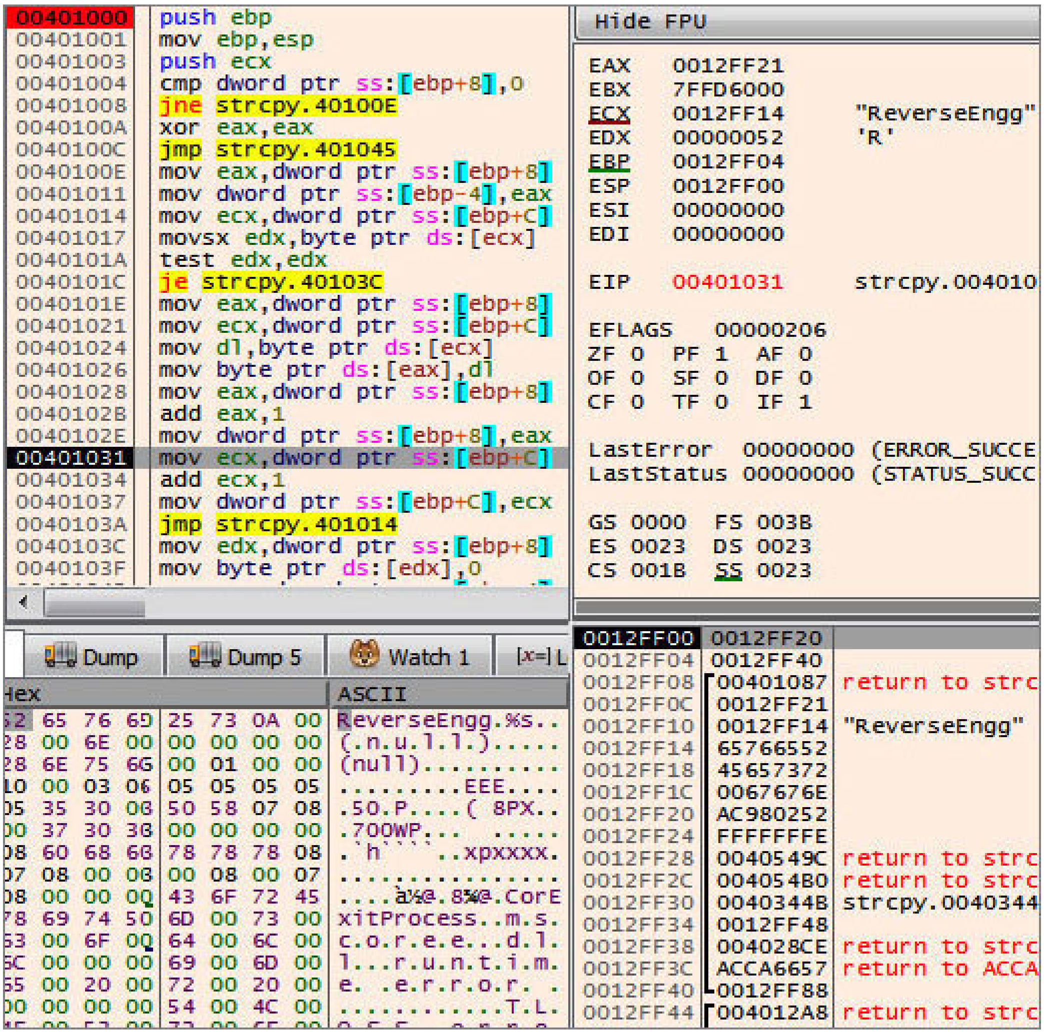
```
dest++;
```

In this ASM section, the pointer to the **dest** array is again moved to the **EAX** register so that it can be incremented to 1 by the **ADD** instruction. This incremented pointer value to the **dest** array is moved back on the stack at the location. The stack state after these instructions will be as follows:

to dest
[ESP+0x04] EBP of main() stack frame
[ESP+0x08] return to main()
[ESP+0x0C] dest array incremented
[ESP+0x10] First arg to xstrcpy
[ESP+0x14] “eveR” is pushed here
[ESP+0x18] “Eesr” is pushed here
[ESP+0x1C] “ ggn” is pushed here
[ESP+0x20] only “R” is copied here at dest array
[ESP+0x24] JUNK HERE
[ESP+0x28] JUNK HERE
[ESP+0x2C] JUNK HERE

[ESP+0x30] JUNK HERE
[ESP+0x34] JUNK HERE
[ESP+0x38] JUNK HERE
[ESP+0x3C] XOR of stack cookie and EBP

[ESP+0x40] of earlier stack frame stored here
[ESP+0x44] return to 0x004012A8 from 0x00401050



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
EAX   0012FF21
EBX   7FFD6000
ECX   0012FF14      "ReverseEngg"
EDX   00000052      'R'
EBP   0012FF04
ESP   0012FF00
ESI   00000000
EDI   00000000

EIP   00401031      strcpy.004010

EFLAGS    00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                  ASCII
2  65 76 65 25 73 0A 00   ReverseEngg.%s..
8  00 6E 00 00 00 00 00   (.n.u.l.l.).....
8  6E 75 6C 00 01 00 00   (null)..........
0  00 03 06 05 05 05 05   ..........EEE....
5  35 30 0B 50 58 07 08   .50.P....( 8PX..
0  37 30 3B 00 00 00 00   .700WP...  .....
8  60 68 6B 78 78 78 08   .`h````..xpxxxx.
7  08 00 0B 00 08 00 07   ................
8  00 00 00 43 6F 72 45   ....à%@.8%@.CorE
8  69 74 50 6D 00 73 00   xitProcess..m.s.
3  00 6F 00 64 00 6C 00   c.o.r.e.e...d.l.
C  00 00 00 69 00 6D 00   l...r.u.n.t.i.m.
5  00 20 00 72 00 20 00   e. .e.r.r.o.r. .
0  00 00 00 54 00 4C 00   ............T.L.
```

```
0012FF00   0012FF20
0012FF04   0012FF40
0012FF08  [00401087   return to strc
0012FF0C   0012FF21
0012FF10   0012FF14   "ReverseEngg"
0012FF14   65766552
0012FF18   45657372
0012FF1C   0067676E
0012FF20   AC980252
0012FF24   FFFFFFFE
0012FF28   0040549C   return to strc
0012FF2C   004054B0   return to strc
0012FF30   0040344B   strcpy.0040344
0012FF34   0012FF48
0012FF38   004028CE   return to strc
0012FF3C   ACCA6657   return to ACCA
0012FF40  L0012FF88
0012FF44  [004012A8   return to strc
```

Figure 14.14: Dest array incremented

▼Line 53-56

; Line 23
mov ecx, DWORD PTR _src$[ebp]
add ecx, 1
mov DWORD PTR _src$[ebp], ecx

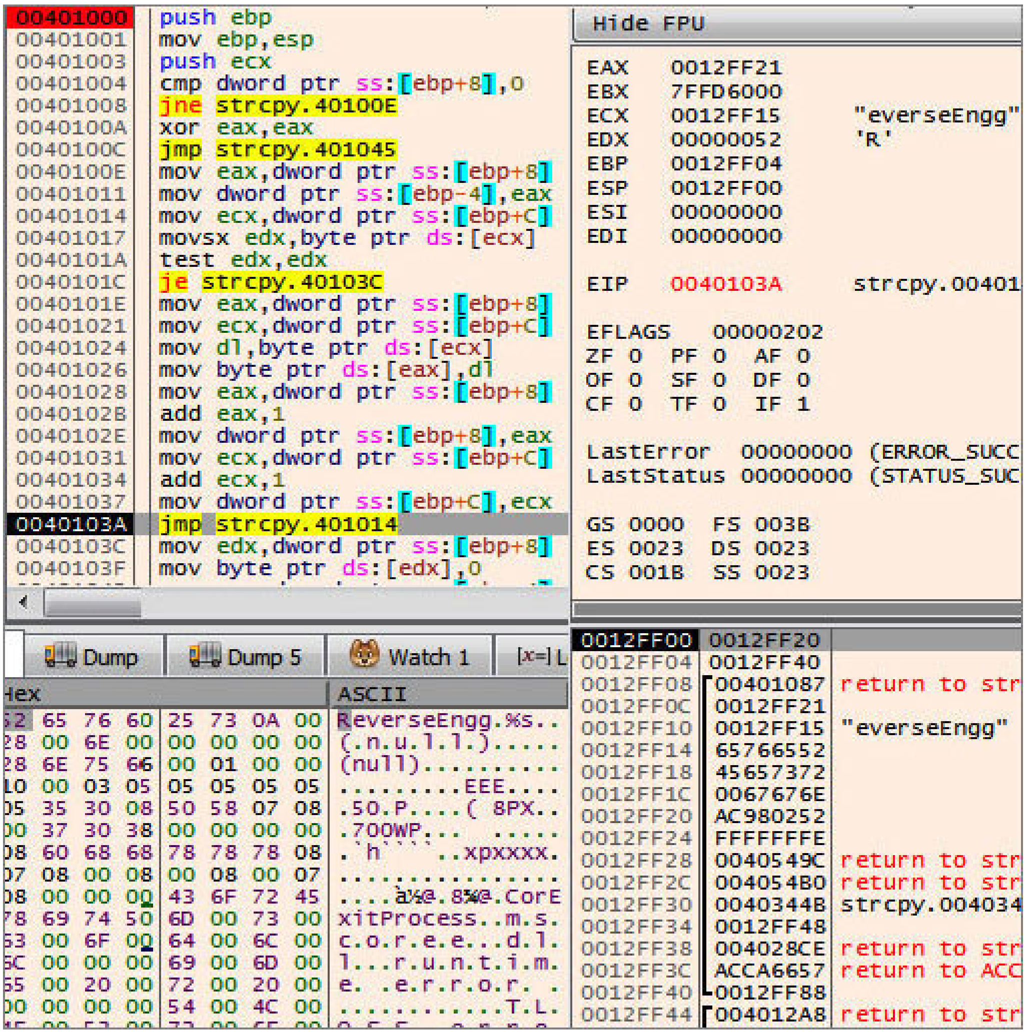
Line 23 of the C/C++ code is:

src++;

In this ASM section, the same steps as the preceding ones are done with the **src** array. The **src** array is moved to the **ECX** register so that it can be incremented to 1 by the **ADD** instruction. This incremented pointer value to the **src** array is moved back on the stack at the location. The stack state after these instructions will be as follows:

to dest array
[ESP+0x04] EBP of main() stack frame
[ESP+0x08] return to main()
[ESP+0x0C] dest array incremented
[ESP+0x10] src array incremented
[ESP+0x14] "eveR" is pushed here
[ESP+0x18] "Eesr" is pushed here
[ESP+0x1C] " ggn" is pushed here

[ESP+0x20] only "R" is copied here at dest array
[ESP+0x24] JUNK HERE
[ESP+0x28] JUNK HERE
[ESP+0x2C] JUNK HERE
[ESP+0x30] JUNK HERE
[ESP+0x34] JUNK HERE
[ESP+0x38] JUNK HERE
[ESP+0x3C] XOR of stack cookie and EBP
[ESP+0x40] of earlier stack frame stored here
[ESP+0x44] return to 0x004012A8 from 0x00401050



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
Hide FPU

EAX   0012FF21
EBX   7FFD6000
ECX   0012FF15     "everseEngg"
EDX   00000052     'R'
EBP   0012FF04
ESP   0012FF00
ESI   00000000
EDI   00000000

EIP   0040103A     strcpy.00401

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCC
LastStatus 00000000 (STATUS_SUC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                  ASCII
2  65 76 60 | 25 73 0A 00   ReverseEngg.%s..
8  00 6E 00 | 00 00 00 00   (.n.u.l.l.).....
8  6E 75 66 | 00 01 00 00   (null)..........
0  00 03 05 | 05 05 05 05   .........EEE....
5  35 30 08 | 50 58 07 08   .50.P....( 8PX..
0  37 30 38 | 00 00 00 00   .700WP...  .....
8  60 68 68 | 78 78 78 08   .`h````..xpxxxx.
7  08 00 08 | 00 08 00 07   ................
8  00 00 00 | 43 6F 72 45   ....à½@.8½@.CorE
8  69 74 50 | 6D 00 73 00   xitProcess..m.s.
3  00 6F 00 | 64 00 6C 00   c.o.r.e.e...d.l.
C  00 00 00 | 69 00 6D 00   l...r.u.n.t.i.m.
5  00 20 00 | 72 00 20 00   e. .e.r.r.o.r. .
0  00 00 00 | 54 00 4C 00   ............T.L.
```

```
0012FF00   0012FF20
0012FF04   0012FF40
0012FF08  [00401087  return to str
0012FF0C   0012FF21
0012FF10   0012FF15  "everseEngg"
0012FF14   65766552
0012FF18   45657372
0012FF1C   0067676E
0012FF20   AC980252
0012FF24   FFFFFFFE
0012FF28   0040549C  return to str
0012FF2C   004054B0  return to str
0012FF30   0040344B  strcpy.004034
0012FF34   0012FF48
0012FF38   004028CE  return to str
0012FF3C   ACCA6657  return to ACC
0012FF40  L0012FF88
0012FF44  [004012A8  return to str
```

Figure 14.15: *Src array incremented*

▼Line 57-58

; Line 24
jmp SHORT $LN2@xstrcpy

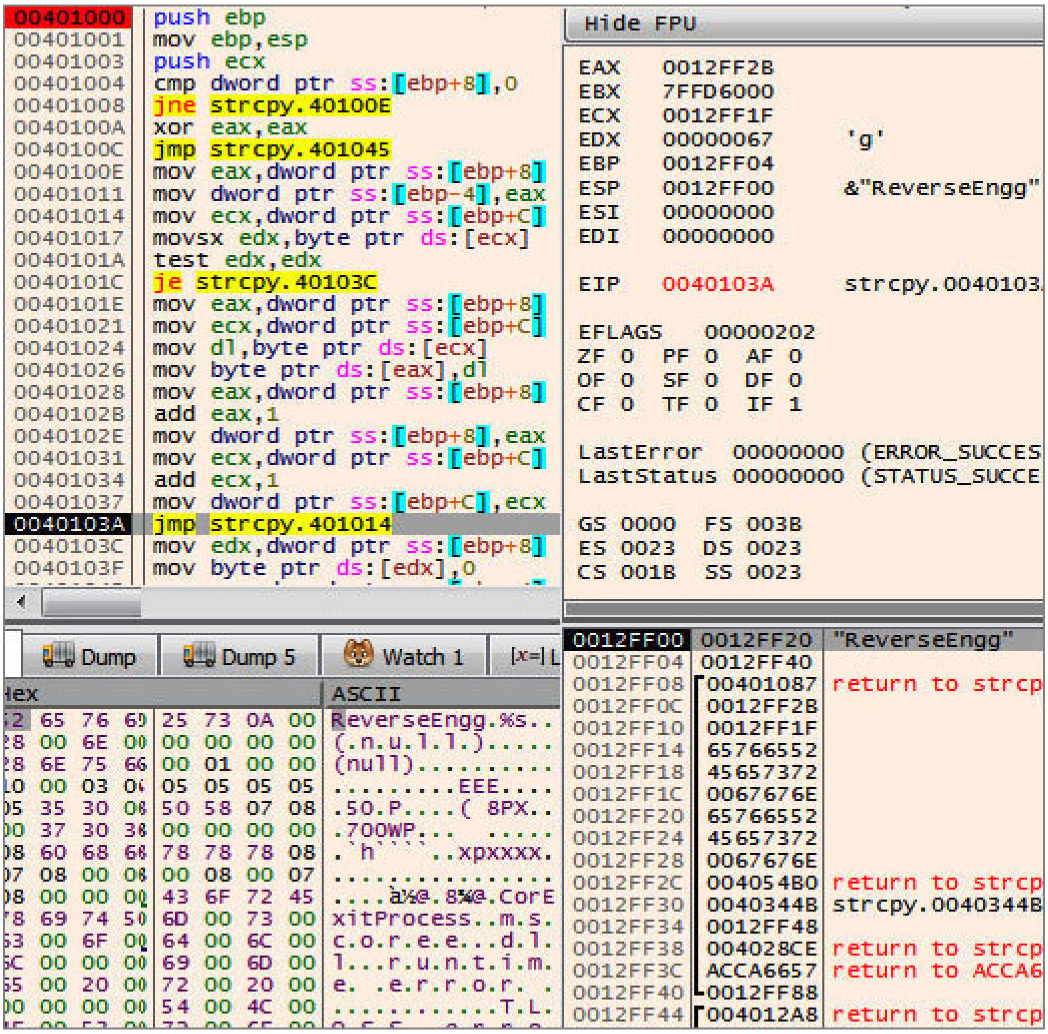
Line 24 of the C/C++ code is:

} //while loop closing

This ASM instruction will perform an unconditional jump to the label This unconditional jump will copy the remaining bytes from the **src** array (stored at to the **dest** array (stored at Now we will consider the iteration where all the bytes of the **src** array are copied to the **dest** array. The stack state after copying all the bytes will be as follows:

to dest array
[ESP+0x04] EBP of main() stack frame
[ESP+0x08] return to main()
[ESP+0x0C] array incremented
[ESP+0x10] array incremented
[ESP+0x14] “eveR” is pushed here
[ESP+0x18] “Eesr” is pushed here
[ESP+0x1C] “ ggn” is pushed here
[ESP+0x20] “eveR” is copied here at dest array
[ESP+0x24] “Eesr” is copied here at dest array
[ESP+0x28] 0012FF28 “ggn” is copied here at dest array

[ESP+0x2C] JUNK HERE
[ESP+0x30] JUNK HERE
[ESP+0x34] JUNK HERE
[ESP+0x38] JUNK HERE
[ESP+0x3C] XOR of stack cookie and EBP
[ESP+0x40] of earlier stack frame stored here
[ESP+0x44] to 0x004012A8 from 0x00401050



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
EAX   0012FF2B
EBX   7FFD6000
ECX   0012FF1F
EDX   00000067     'g'
EBP   0012FF04
ESP   0012FF00     &"ReverseEngg"
ESI   00000000
EDI   00000000

EIP   0040103A     strcpy.0040103

EFLAGS   00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCES
LastStatus 00000000 (STATUS_SUCCE

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
25 73 0A 00  ReverseEngg.%s..
00 00 00 00  (.n.u.l.l.).....
00 01 00 00  (null)..........
05 05 05 05  .........EEE....
50 58 07 08  .50.P....( 8PX..
00 00 00 00  .700WP...  .....
78 78 78 08  .`h````..xpxxxx.
00 08 00 07  ................
43 6F 72 45  ....à%@.8%@.CorE
6D 00 73 00  xitProcess..m.s.
64 00 6C 00  c.o.r.e.e...d.l.
69 00 6D 00  l...r.u.n.t.i.m.
72 00 20 00  e. .e.r.r.o.r. .
54 00 4C 00  ............T.L.
```

```
0012FF00  0012FF20  "ReverseEngg"
0012FF04  0012FF40
0012FF08  00401087  return to strcp
0012FF0C  0012FF2B
0012FF10  0012FF1F
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  65766552
0012FF24  45657372
0012FF28  0067676E
0012FF2C  004054B0  return to strcp
0012FF30  0040344B  strcpy.0040344B
0012FF34  0012FF48
0012FF38  004028CE  return to strcp
0012FF3C  ACCA6657  return to ACCA6
0012FF40  0012FF88
0012FF44  004012A8  return to strcp
```

Figure 14.16: *Src array are copied to dest array*

Now, we will take the iteration (where an unconditional jump to the label **$LN2@xstrcpy** was after all the bytes are copied from the **src** array to the **dest** array.

▼Line 38-43

$LN2@xstrcpy:
; Line 19
mov ecx, DWORD PTR _src$[ebp]

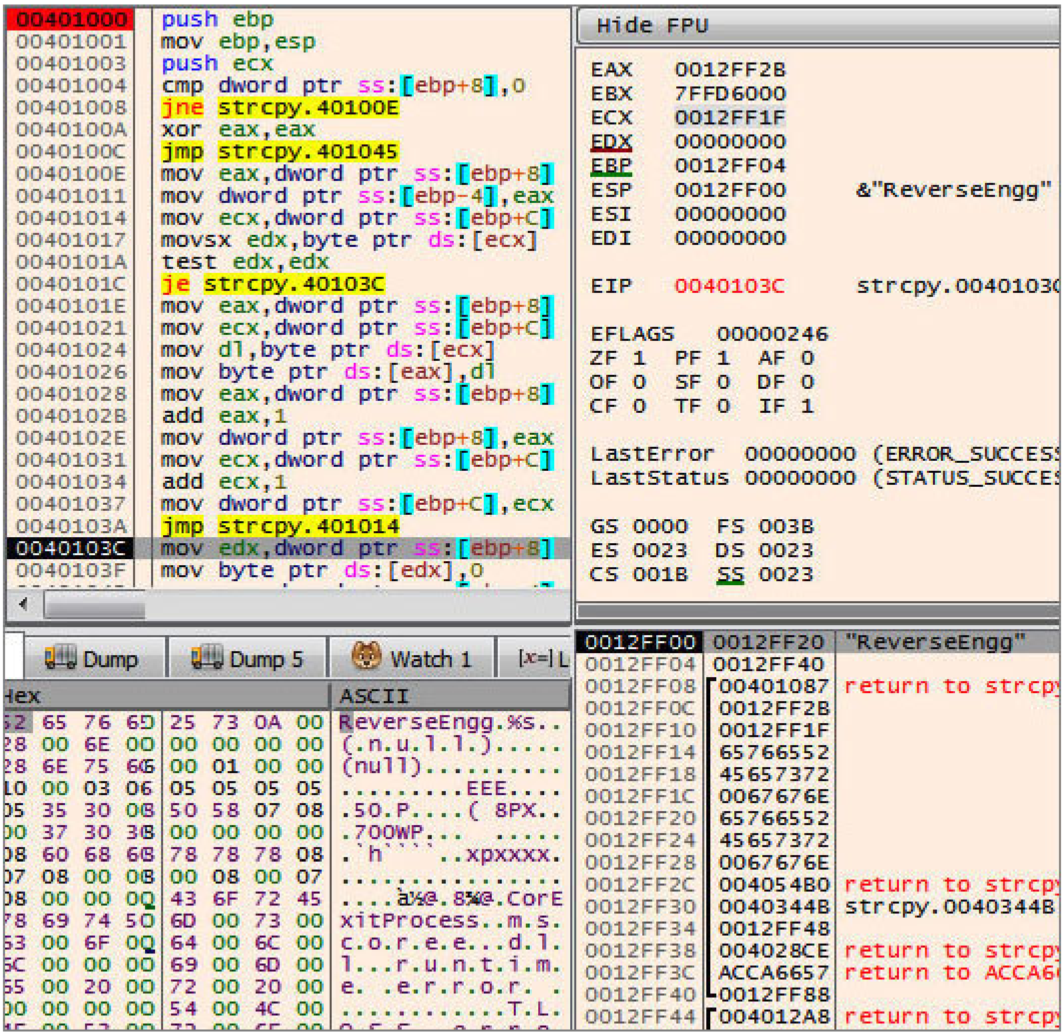
```
movsx edx, BYTE PTR [ecx]
test edx, edx
je SHORT $LN1@xstrcpy
```

Line 19 of the C/C++ code is:

```
while (*src != '\0')
```

In the ASM code, the **MOV** instruction is moving the pointer from the **src** array byte (0x0012FF1F at to the **ECX** register. The next **MOV** instruction moves byte stored at 0x0012FF1F to making

The **TEST** instruction will perform an **AND** operation of **EDX** with itself, resulting in a zero value in **EDX** and ZF=1. So, a jump to the label **$LN1@xstrcpy** will take place. The stack state will be the same as earlier:



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
```

```
EAX   0012FF2B
EBX   7FFD6000
ECX   0012FF1F
EDX   00000000
EBP   0012FF04
ESP   0012FF00      &"ReverseEngg"
ESI   00000000
EDI   00000000

EIP   0040103C      strcpy.0040103

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCES
LastStatus  00000000 (STATUS_SUCCE

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023
```

```
Hex                                ASCII
2  65 76 6D  25 73 0A 00   ReverseEngg.%s..
8  00 6E 00  00 00 00 00   (.n.u.l.l.).....
8  6E 75 66  00 01 00 00   (null)..........
0  00 03 06  05 05 05 05   ..........EEE....
5  35 30 08  50 58 07 08   .50.P....( 8PX..
0  37 30 38  00 00 00 00   .700WP...  .....
8  60 68 68  78 78 78 08   .`h```  ..xpxxxx.
7  08 00 08  00 08 00 07   ................
8  00 00 00  43 6F 72 45   ....à%@.8%@.CorE
8  69 74 50  6D 00 73 00   xitProcess..m.s.
3  00 6F 00  64 00 6C 00   c.o.r.e.e...d.l.
C  00 00 00  69 00 6D 00   l...r.u.n.t.i.m.
5  00 20 00  72 00 20 00   e. .e.r.r.o.r. .
0  00 00 00  54 00 4C 00   ............T.L.
```

```
0012FF00  0012FF20  "ReverseEngg"
0012FF04  0012FF40
0012FF08  00401087  return to strcpy
0012FF0C  0012FF2B
0012FF10  0012FF1F
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  65766552
0012FF24  45657372
0012FF28  0067676E
0012FF2C  004054B0  return to strcpy
0012FF30  0040344B  strcpy.0040344B
0012FF34  0012FF48
0012FF38  004028CE  return to strcpy
0012FF3C  ACCA6657  return to ACCA6
0012FF40  0012FF88
0012FF44  004012A8  return to strcpy
```

__Figure 14.17:__ EDX is 0x00

▼Line 59-62

```
$LN1@xstrcpy:
; Line 27
mov edx, DWORD PTR _dest$[ebp]
mov BYTE PTR [edx], 0
```

Line 27 of the C/C++ code is:

```
*dest = '\0';
```

In ASM code is pushing pointer to **dest** array (0x0012FF2B at to **EDX** register, where it is filled with the null character byte. The stack state will be the same as earlier:

▼Line 63-69

```
; Line 30
mov eax, DWORD PTR _ptr$[ebp]
$LN4@xstrcpy:
; Line 31
mov esp, ebp
pop ebp
ret 0
```

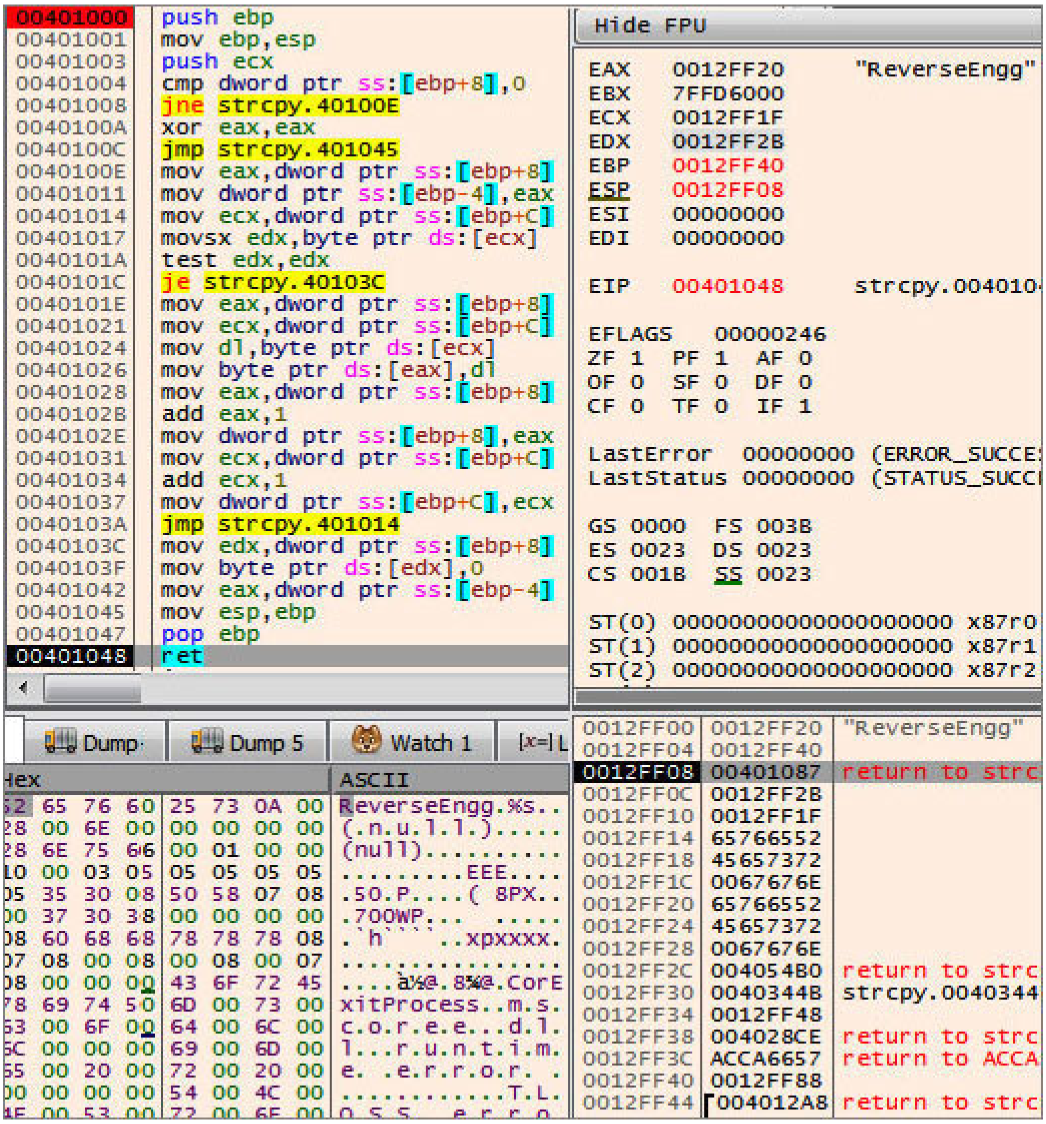
Line 30 of the C/C++ code is:

```
return ptr;
```

In the ASM code, the pointer to the **dest** array stored at is pushed to the **EAX** register, as the function return value will be stored in the **EAX** register. Once this value is moved to the **xstrcpy** function epilogue is called with the RET instruction. The RET instruction will move the instruction pointer back to the **main** function. The stack state before the RET instruction is as follows:

```
[ESP-0x08] 0012FF00   0012FF20    Now JUNK
[ESP-0x04] 0012FF04   0012FF40    Now JUNK
[ESP]   0012FF08   00401087    return address to main()
```

```
[ESP+0x04] 0012FF0C   0012FF2B    Pointer to dest array
incremented
[ESP+0x08] 0012FF10   0012FF1F    Pointer to src array
incremented
[ESP+0x0C] 0012FF14   65766552    “eveR” is pushed here

[ESP+0x10] 0012FF18   45657372    “Eesr” is pushed here
[ESP+0x14] 0012FF1C   0067676E    “ ggn” is pushed here
[ESP+0x18] 0012FF20   65766552    “eveR” is copied here at dest
array
[ESP+0x1C] 0012FF24   45657372    “Eesr” is copied here at dest
array
[ESP+0x20] 0012FF28   0067676E    “ggn” is copied here at dest
array
[ESP+0x24] 0012FF2C   004054B0    JUNK HERE
[ESP+0x28] 0012FF30   0040344B    JUNK HERE
[ESP+0x2C] 0012FF34   0012FF48    JUNK HERE
[ESP+0x30] 0012FF38   004028CE    JUNK HERE
[ESP+0x34] 0012FF3C   ACCA6657    XOR of stack cookie and EBP
[ESP+0x38] 0012FF40   0012FF88    of earlier stack frame
[ESP+0x3C] 0012FF44   004012A8    return to 0x004012A8
```



```
00401000  push ebp
00401001  mov ebp,esp
00401003  push ecx
00401004  cmp dword ptr ss:[ebp+8],0
00401008  jne strcpy.40100E
0040100A  xor eax,eax
0040100C  jmp strcpy.401045
0040100E  mov eax,dword ptr ss:[ebp+8]
00401011  mov dword ptr ss:[ebp-4],eax
00401014  mov ecx,dword ptr ss:[ebp+C]
00401017  movsx edx,byte ptr ds:[ecx]
0040101A  test edx,edx
0040101C  je strcpy.40103C
0040101E  mov eax,dword ptr ss:[ebp+8]
00401021  mov ecx,dword ptr ss:[ebp+C]
00401024  mov dl,byte ptr ds:[ecx]
00401026  mov byte ptr ds:[eax],dl
00401028  mov eax,dword ptr ss:[ebp+8]
0040102B  add eax,1
0040102E  mov dword ptr ss:[ebp+8],eax
00401031  mov ecx,dword ptr ss:[ebp+C]
00401034  add ecx,1
00401037  mov dword ptr ss:[ebp+C],ecx
0040103A  jmp strcpy.401014
0040103C  mov edx,dword ptr ss:[ebp+8]
0040103F  mov byte ptr ds:[edx],0
00401042  mov eax,dword ptr ss:[ebp-4]
00401045  mov esp,ebp
00401047  pop ebp
00401048  ret
```

```
EAX   0012FF20     "ReverseEngg"
EBX   7FFD6000
ECX   0012FF1F
EDX   0012FF2B
EBP   0012FF40
ESP   0012FF08
ESI   00000000
EDI   00000000

EIP   00401048     strcpy.004010

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCE
LastStatus 00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0
ST(1) 00000000000000000000 x87r1
ST(2) 00000000000000000000 x87r2
```

```
2 65 76 60 | 25 73 0A 00  ReverseEngg.%s..
8 00 6E 00 | 00 00 00 00  (.n.u.l.l.).....
8 6E 75 66 | 00 01 00 00  (null)..........
0 00 03 05 | 05 05 05 05  ..........EEE....
5 35 30 08 | 50 58 07 08  .50.P....( 8PX..
0 37 30 38 | 00 00 00 00  .700WP...  .....
8 60 68 68 | 78 78 78 08  .`h```  ..xpxxxx.
7 08 00 08 | 00 08 00 07  ................
8 00 00 00 | 43 6F 72 45  ....à%@.8%@.CorE
8 69 74 50 | 6D 00 73 00  xitProcess..m.s.
3 00 6F 00 | 64 00 6C 00  c.o.r.e.e...d.l.
C 00 00 00 | 69 00 6D 00  l...r.u.n.t.i.m.
5 00 20 00 | 72 00 20 00  e. .e.r.r.o.r. .
0 00 00 00 | 54 00 4C 00  ..............T.L.
```

```
0012FF00  0012FF20  "ReverseEngg"
0012FF04  0012FF40
0012FF08  00401087  return to strc
0012FF0C  0012FF2B
0012FF10  0012FF1F
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  65766552
0012FF24  45657372
0012FF28  0067676E
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy.0040344
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  ACCA6657  return to ACCA
0012FF40  0012FF88
0012FF44  004012A8  return to strc
```

Figure 14.18: *Stack state before RET*

▼Line 103-107

add esp, 8
push eax
push OFFSET $SG4690
call _printf
add esp, 8

Line 30 of the C/C++ code is:

```
printf("%s\n", xstrcpy(dest, src));
```

On return, the **ADD** instruction will clean up the stack by 8 bytes.

The **printf** function is called by pushing the return value of **xstrcpy** onto the stack and the string constant **$SG4690** (which is **$SG4690 DB '%s', 0aH,** to call the **printf** function.

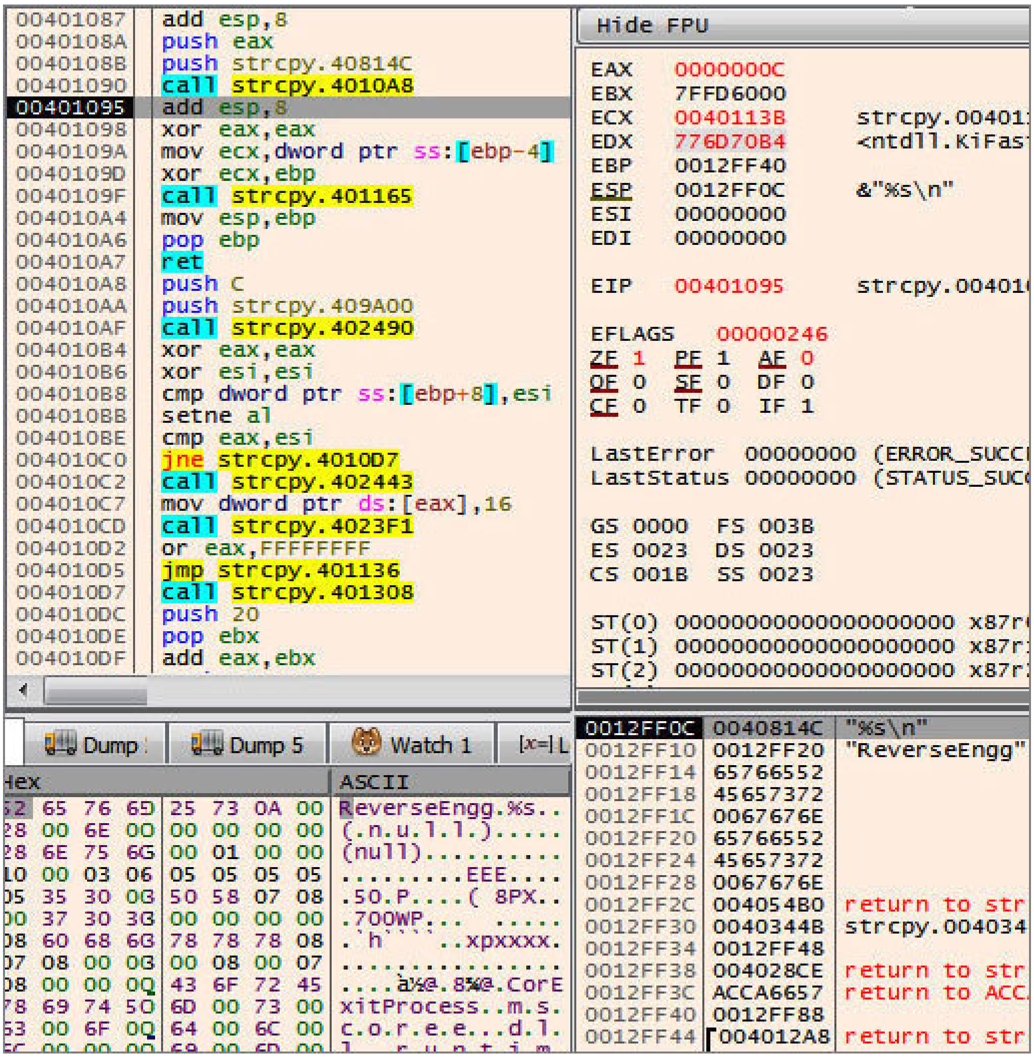
This will print the copied string on the screen.

The **ADD** instruction after **printf** is called to clean up the stack by 8 bytes. The stack state before the **ADD** instruction is as follows:

"%s\n", argument to printf()
[ESP+0x04] "ReverseEngg", argument to printf()
[ESP+0x08] “eveR” is pushed here
[ESP+0x0C] “Eesr” is pushed here
[ESP+0x10] “ ggn” is pushed here
[ESP+0x14] “eveR” is copied here at dest array
[ESP+0x18] “Eesr” is copied here at dest array
[ESP+0x1C] “ggn” is copied here at dest array
[ESP+0x20] JUNK HERE
[ESP+0x24] JUNK HERE

[ESP+0x28] JUNK HERE
[ESP+0x2C] JUNK HERE
[ESP+0x30] XOR of stack cookie and EBP
[ESP+0x34] of earlier stack frame stored here

[ESP+0x38] return to 0x004012A8



```
00401087  add esp,8
0040108A  push eax
0040108B  push strcpy.40814C
00401090  call strcpy.4010A8
00401095  add esp,8
00401098  xor eax,eax
0040109A  mov ecx,dword ptr ss:[ebp-4]
0040109D  xor ecx,ebp
0040109F  call strcpy.401165
004010A4  mov esp,ebp
004010A6  pop ebp
004010A7  ret
004010A8  push C
004010AA  push strcpy.409A00
004010AF  call strcpy.402490
004010B4  xor eax,eax
004010B6  xor esi,esi
004010B8  cmp dword ptr ss:[ebp+8],esi
004010BB  setne al
004010BE  cmp eax,esi
004010C0  jne strcpy.4010D7
004010C2  call strcpy.402443
004010C7  mov dword ptr ds:[eax],16
004010CD  call strcpy.4023F1
004010D2  or eax,FFFFFFFF
004010D5  jmp strcpy.401136
004010D7  call strcpy.401308
004010DC  push 20
004010DE  pop ebx
004010DF  add eax,ebx
```

```
EAX   0000000C
EBX   7FFD6000
ECX   0040113B     strcpy.00401
EDX   776D70B4     <ntdll.KiFas
EBP   0012FF40
ESP   0012FF0C     &"%s\n"
ESI   00000000
EDI   00000000

EIP   00401095     strcpy.00401

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCC
LastStatus  00000000 (STATUS_SUC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000 x87r
ST(1) 0000000000000000000 x87r
ST(2) 0000000000000000000 x87r
```

```
 65 76 65 25 73 0A 00  ReverseEngg.%s..
 00 6E 00 00 00 00 00  (.n.u.l.l.).....
 6E 75 6C 00 01 00 00  (null)..........
 00 03 06 05 05 05 05  ..........EEE....
 35 30 0C 50 58 07 08  .50.P....( 8PX..
 37 30 3C 00 00 00 00  .700WP...  .....
 60 68 6C 78 78 78 08  .`h````..xpxxxx.
 08 00 0C 00 08 00 07  ................
 00 00 00 43 6F 72 45  ....à½@.8½@.CorE
 69 74 50 6D 00 73 00  xitProcess..m.s.
 00 6F 00 64 00 6C 00  c.o.r.e.e...d.l.
```

```
0012FF0C  0040814C  "%s\n"
0012FF10  0012FF20  "ReverseEngg"
0012FF14  65766552
0012FF18  45657372
0012FF1C  0067676E
0012FF20  65766552
0012FF24  45657372
0012FF28  0067676E
0012FF2C  004054B0  return to str
0012FF30  0040344B  strcpy.004034
0012FF34  0012FF48
0012FF38  004028CE  return to str
0012FF3C  ACCA6657  return to ACC
0012FF40  0012FF88
0012FF44  004012A8  return to str
```

Figure 14.19: *Stack state before ADD instruction*

▼Line 108-119

; Line 41

xor eax, eax

; Line 42

mov ecx, DWORD PTR __$ArrayPad$[ebp]

xor ecx, ebp

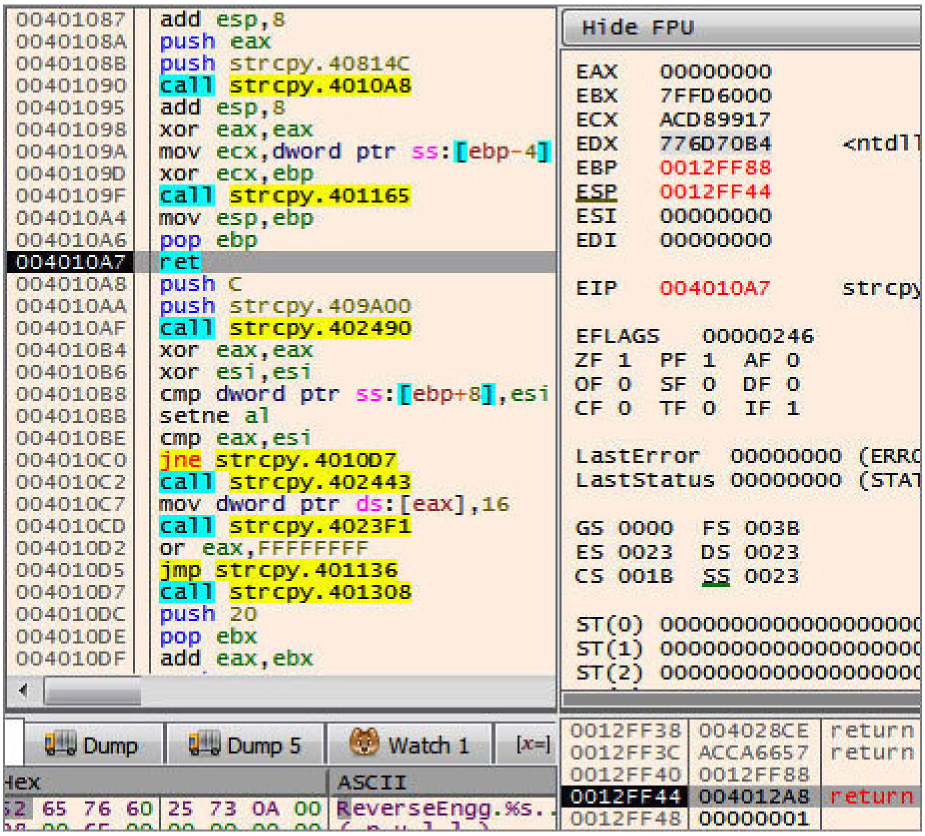
```
call @__security_check_cookie@4
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

Line 30 of the C/C++ code is:

```
return 0;
}
```

In the ASM code, **EAX** is cleared by XOR and then the stack cookie is checked for any buffer overflow attack by calling the **security_check_cookie** procedure. On return, the **main** function epilogue is called to return 0 and end the **main** function, TEXT segment, and code. The stack state before the RET instruction is as follows:

```
[ESP-0x0C] 0012FF38   004028CE    Now JUNK HERE
[ESP-0x08] 0012FF3C   ACCA6657    Now JUNK HERE
[ESP-0x04] 0012FF40   0012FF88    Now JUNK HERE
[ESP]   0012FF44   004012A8    return to strcpy.004012A8 from
strcpy.00401050
```



```
00401087  add esp,8
0040108A  push eax
0040108B  push strcpy.40814C
00401090  call strcpy.4010A8
00401095  add esp,8
00401098  xor eax,eax
0040109A  mov ecx,dword ptr ss:[ebp-4]
0040109D  xor ecx,ebp
0040109F  call strcpy.401165
004010A4  mov esp,ebp
004010A6  pop ebp
004010A7  ret
004010A8  push C
004010AA  push strcpy.409A00
004010AF  call strcpy.402490
004010B4  xor eax,eax
004010B6  xor esi,esi
004010B8  cmp dword ptr ss:[ebp+8],esi
004010BB  setne al
004010BE  cmp eax,esi
004010C0  jne strcpy.4010D7
004010C2  call strcpy.402443
004010C7  mov dword ptr ds:[eax],16
004010CD  call strcpy.4023F1
004010D2  or eax,FFFFFFFF
004010D5  jmp strcpy.401136
004010D7  call strcpy.401308
004010DC  push 20
004010DE  pop ebx
004010DF  add eax,ebx
```

```
EAX   00000000
EBX   7FFD6000
ECX   ACD89917
EDX   776D70B4      <ntdll
EBP   0012FF88
ESP   0012FF44
ESI   00000000
EDI   00000000

EIP   004010A7      strcpy

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STAT

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
```

```
0012FF38  004028CE  return
0012FF3C  ACCA6657  return
0012FF40  0012FF88
0012FF44  004012A8  return
0012FF48  00000001
```

```
65 76 60 25 73 0A 00  ReverseEngg.%s..
```

Figure 14.20: *Stack state before RET instruction*

Strcpy with Optimization

Compile the code with the optimization flag, **/Ox** flag. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

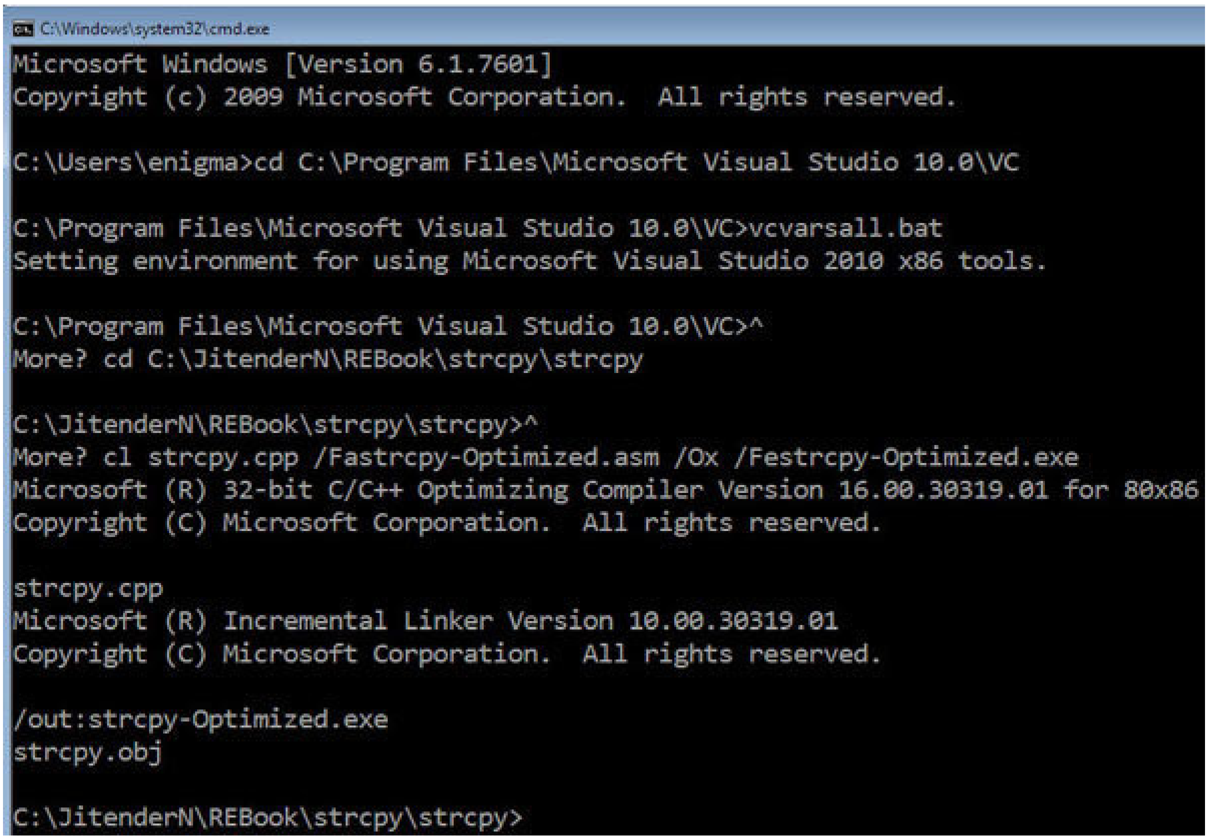
Enable maximum optimization

Name of the output assembly listing file

Name of the output executable file

```
file file file file file file file file file file file file file file file file
```

The following is the output of running the preceding commands:

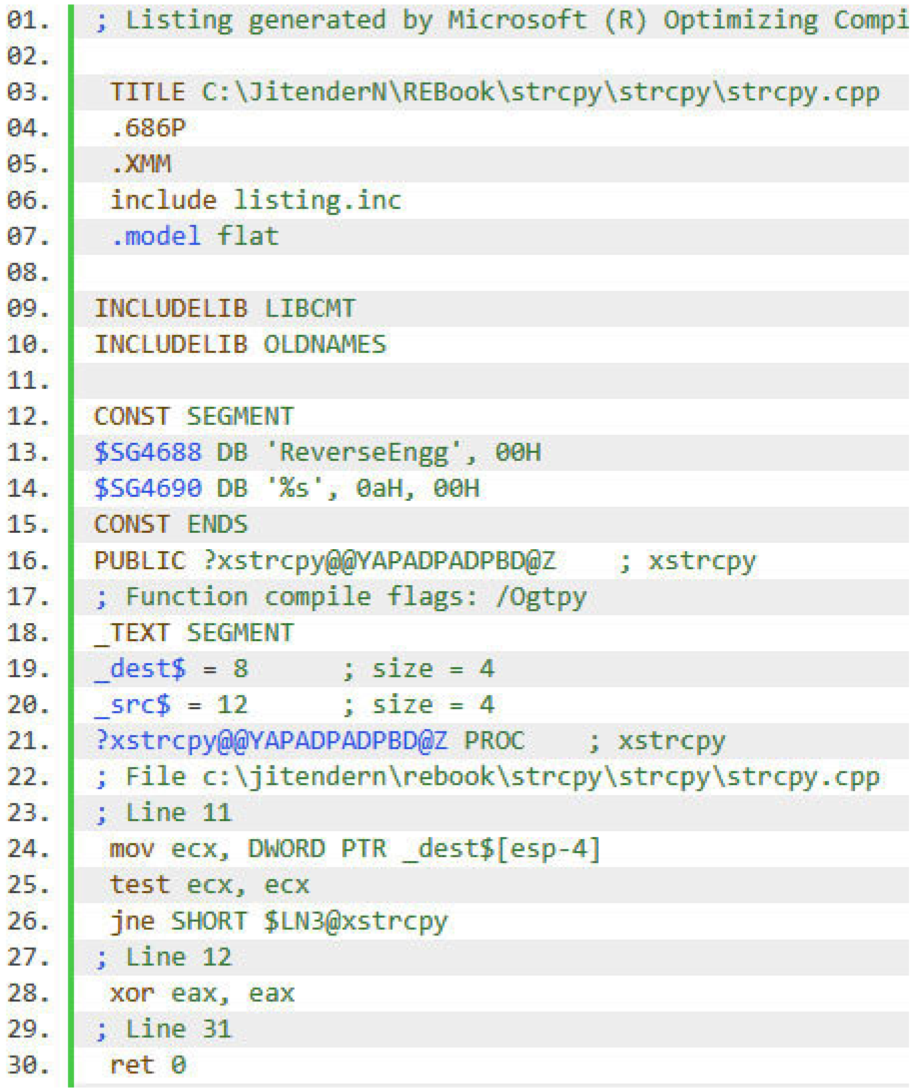


```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\strcpy\strcpy

C:\JitenderN\REBook\strcpy\strcpy>^
More? cl strcpy.cpp /Fastrcpy-Optimized.asm /Ox /Festrcpy-Optimized.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

strcpy.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:strcpy-Optimized.exe
strcpy.obj

C:\JitenderN\REBook\strcpy\strcpy>
```

Figure 14.21: *Strcpy with Optimization*

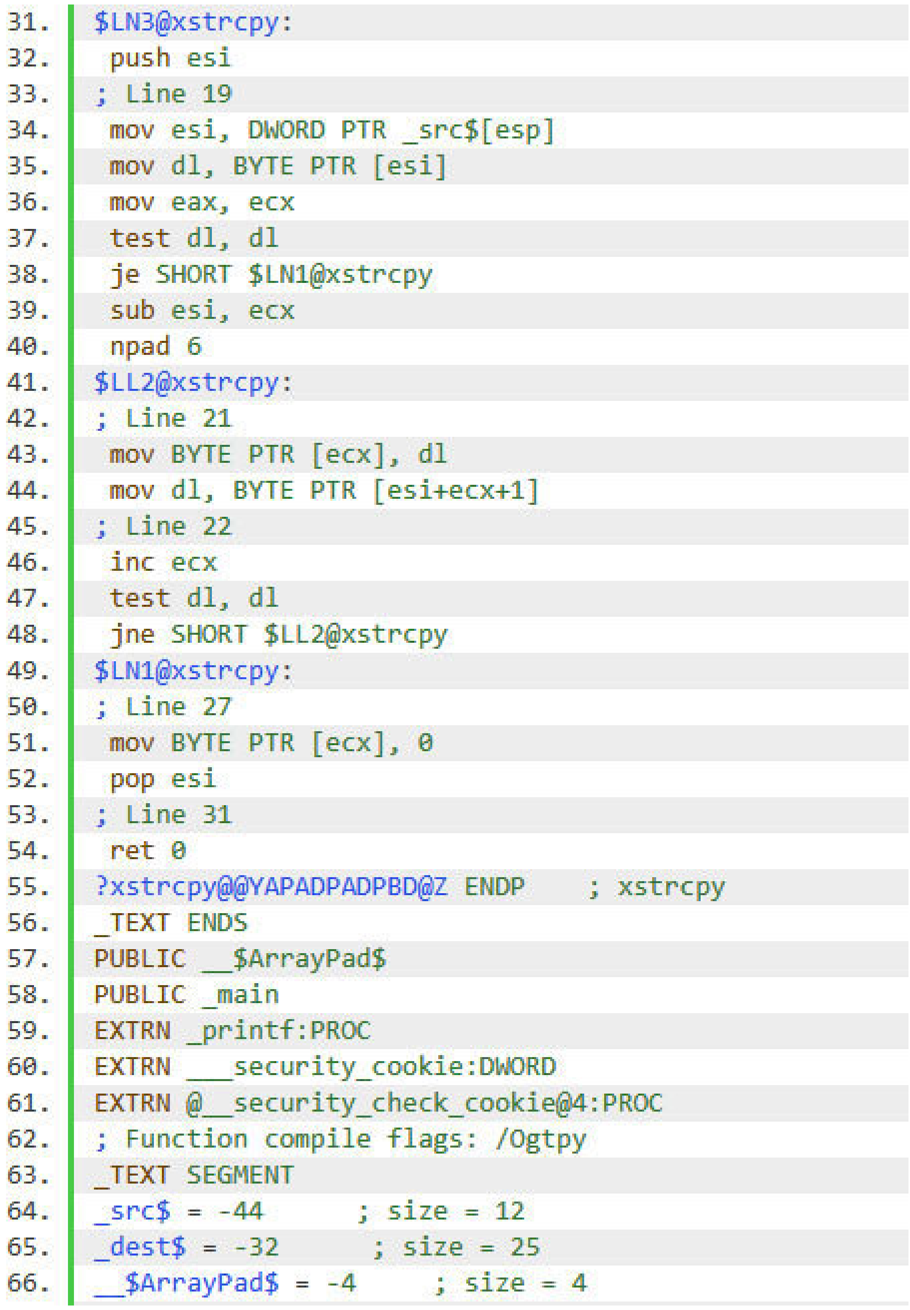
The compilation generates the EXE file and the assembly code. We will again manually disable ASLR. To disable ASLR, use the CFF explorer and change the **DllCharacteristics** parameter to uncheck **DLL can**

Now, let’s move on to the generated assembly listing:



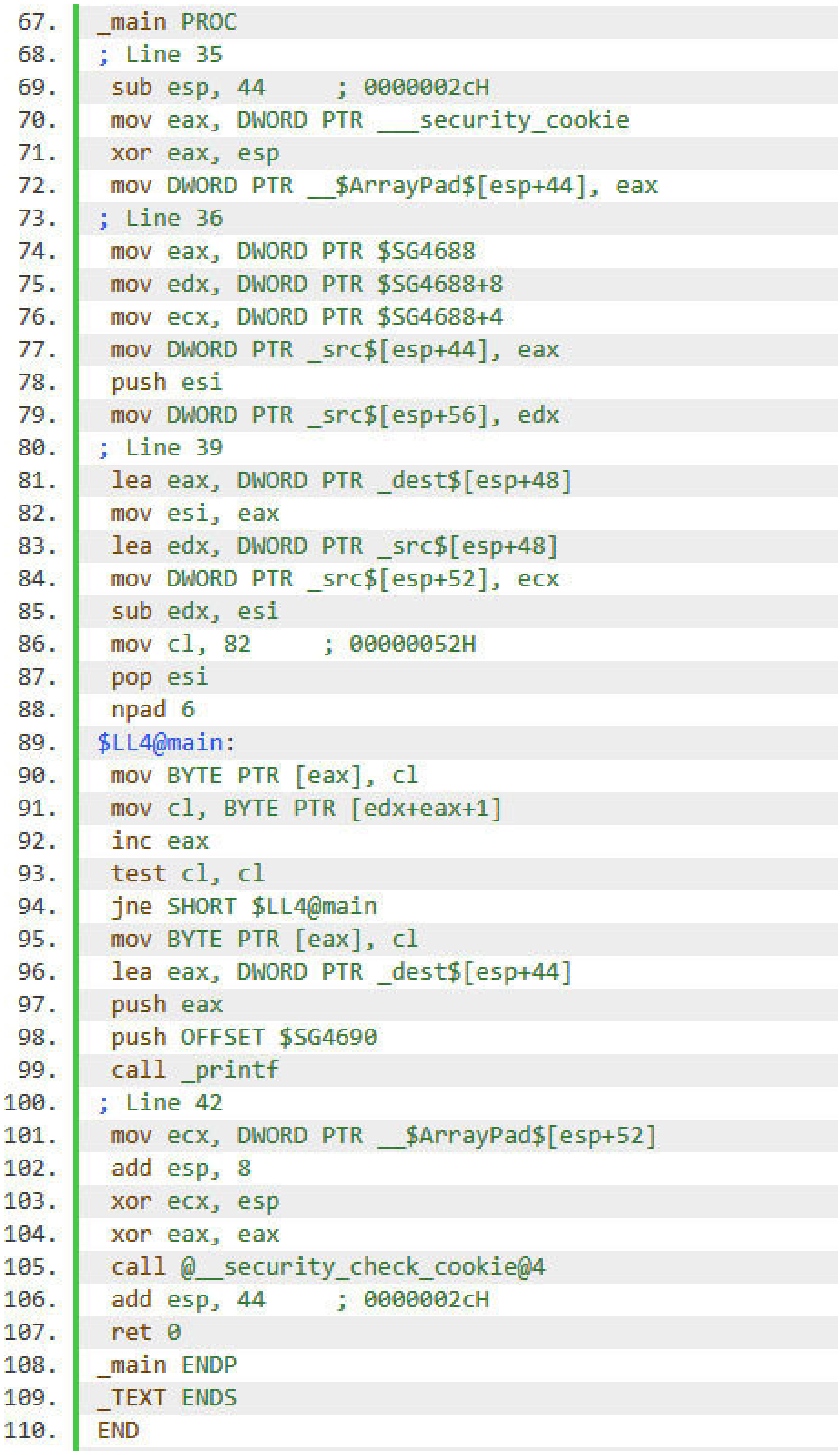
```
; Listing generated by Microsoft (R) Optimizing Compi

 TITLE C:\JitenderN\REBook\strcpy\strcpy\strcpy.cpp
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4688 DB 'ReverseEngg', 00H
$SG4690 DB '%s', 0aH, 00H
CONST ENDS
PUBLIC ?xstrcpy@@YAPADPADPBD@Z    ; xstrcpy
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_dest$ = 8        ; size = 4
_src$ = 12        ; size = 4
?xstrcpy@@YAPADPADPBD@Z PROC    ; xstrcpy
; File c:\jitendern\rebook\strcpy\strcpy\strcpy.cpp
; Line 11
 mov ecx, DWORD PTR _dest$[esp-4]
 test ecx, ecx
 jne SHORT $LN3@xstrcpy
; Line 12
 xor eax, eax
; Line 31
 ret 0
```

Figure 14.22: strcpy-Optimized.asm-Part-1



```
$LN3@xstrcpy:
 push esi
; Line 19
 mov esi, DWORD PTR _src$[esp]
 mov dl, BYTE PTR [esi]
 mov eax, ecx
 test dl, dl
 je SHORT $LN1@xstrcpy
 sub esi, ecx
 npad 6
$LL2@xstrcpy:
; Line 21
 mov BYTE PTR [ecx], dl
 mov dl, BYTE PTR [esi+ecx+1]
; Line 22
 inc ecx
 test dl, dl
 jne SHORT $LL2@xstrcpy
$LN1@xstrcpy:
; Line 27
 mov BYTE PTR [ecx], 0
 pop esi
; Line 31
 ret 0
?xstrcpy@@YAPADPADPBD@Z ENDP    ; xstrcpy
_TEXT ENDS
PUBLIC __$ArrayPad$
PUBLIC _main
EXTRN _printf:PROC
EXTRN ___security_cookie:DWORD
EXTRN @__security_check_cookie@4:PROC
; Function compile flags: /Ogtpy
_TEXT SEGMENT
_src$ = -44       ; size = 12
_dest$ = -32       ; size = 25
__$ArrayPad$ = -4      ; size = 4
```

Figure 14.23: *strcpy-Optimized.asm-Part-2*



```
_main PROC
; Line 35
 sub esp, 44     ; 0000002cH
 mov eax, DWORD PTR ___security_cookie
 xor eax, esp
 mov DWORD PTR __$ArrayPad$[esp+44], eax
; Line 36
 mov eax, DWORD PTR $SG4688
 mov edx, DWORD PTR $SG4688+8
 mov ecx, DWORD PTR $SG4688+4
 mov DWORD PTR _src$[esp+44], eax
 push esi
 mov DWORD PTR _src$[esp+56], edx
; Line 39
 lea eax, DWORD PTR _dest$[esp+48]
 mov esi, eax
 lea edx, DWORD PTR _src$[esp+48]
 mov DWORD PTR _src$[esp+52], ecx
 sub edx, esi
 mov cl, 82     ; 00000052H
 pop esi
 npad 6
$LL4@main:
 mov BYTE PTR [eax], cl
 mov cl, BYTE PTR [edx+eax+1]
 inc eax
 test cl, cl
 jne SHORT $LL4@main
 mov BYTE PTR [eax], cl
 lea eax, DWORD PTR _dest$[esp+44]
 push eax
 push OFFSET $SG4690
 call _printf
; Line 42
 mov ecx, DWORD PTR __$ArrayPad$[esp+52]
 add esp, 8
 xor ecx, esp
 xor eax, eax
 call @__security_check_cookie@4
 add esp, 44     ; 0000002cH
 ret 0
_main ENDP
_TEXT ENDS
END
```

Figure 14.24: *strcpy-Optimized.asm-Part-3*

Let's directly jump to the start of the TEXT segment of the **_main** function.

▼Line 63-67

```
_TEXT SEGMENT
_src$ = -44          ; size = 12
_dest$ = -32          ; size = 25
__$ArrayPad$ = -4        ; size = 4
_main PROC
```

The local variable on the stack frame can be accessed by adding **_$** to the EBP address.

▼Line 67

```
_main PROC
```

Start of the **main** procedure.

▼Line 68-72

```
; Line 35
sub esp, 44        ; 0000002cH
mov eax, DWORD PTR ___security_cookie
xor eax, esp
mov DWORD PTR __$ArrayPad$[esp+44], eax
```

Line 35 of the C/C++ code starts the **main** function.

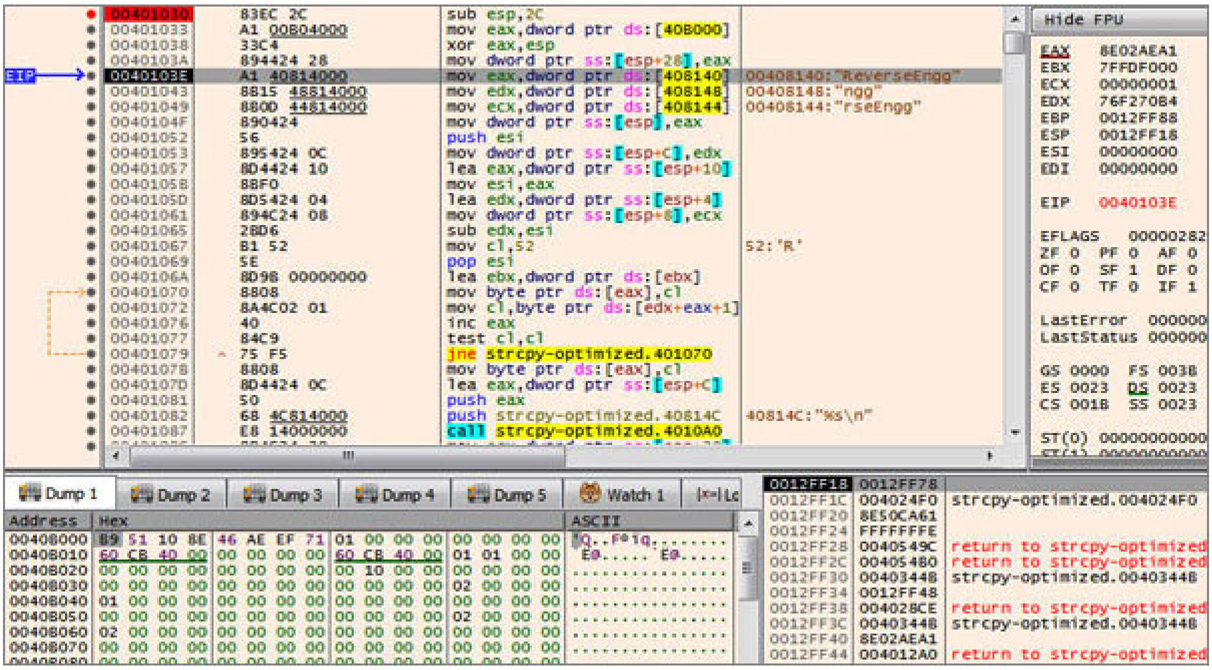
```
int main() {
```

This ASM code is optimized and starts by subtracting **ESP** by 44(0x2C) bytes to create room for the local variables. 44 bytes can be visualized as 4 bytes for **security_cookie** plus 12 bytes for the **src** array, and 28 bytes for the **dest** array. The sizes of **src** and **dest** arrays have been round off to a multiple of 4 bytes.

The next **MOV** instruction moves the stack cookie from the **EAX** register to XOR with **ESP** and stores the resultant on the stack. In a non-optimized code, this XOR procedure happens with EBP.

Let's see the stack state after moving the stack cookie on the stack:

```
[ESP]   0012FF18   0012FF78   JUNK HERE
[ESP+0x04] 0012FF1C   004024F0   JUNK HERE
[ESP+0x08] 0012FF20   8E50CA61   JUNK HERE
[ESP+0x0C] 0012FF24   FFFFFFFE   JUNK HERE
[ESP+0x10] 0012FF28   0040549C   JUNK HERE
[ESP+0x14] 0012FF2C   004054B0   JUNK HERE
[ESP+0x18] 0012FF30   0040344B   JUNK HERE
[ESP+0x1C] 0012FF34   0012FF48   JUNK HERE
[ESP+0x20] 0012FF38   004028CE   JUNK HERE
[ESP+0x24] 0012FF3C   0040344B   JUNK HERE
[ESP+0x28] 0012FF40   8E02AEA1   XOR of stack cookie and EBP stored here
[ESP+0x2C] 0012FF44   004012A0   ESP at the start of main procedure
```



```
00401030   83EC 2C            sub esp,2C
00401033   A1 00B04000        mov eax,dword ptr ds:[40B000]
00401038   33C4               xor eax,esp
0040103A   894424 28          mov dword ptr ss:[esp+28],eax
0040103E   A1 40814000        mov eax,dword ptr ds:[408140]      00408140:"ReverseEngg"
00401043   8B15 48814000      mov edx,dword ptr ds:[408148]      00408148:"ngg"
00401049   8B0D 44814000      mov ecx,dword ptr ds:[408144]      00408144:"rseEngg"
0040104F   890424             mov dword ptr ss:[esp],eax
00401052   56                 push esi
00401053   895424 0C          mov dword ptr ss:[esp+C],edx
00401057   8D4424 10          lea eax,dword ptr ss:[esp+10]
0040105B   8BF0               mov esi,eax
0040105D   8D5424 04          lea edx,dword ptr ss:[esp+4]
00401061   894C24 08          mov dword ptr ss:[esp+8],ecx
00401065   2BD6               sub edx,esi
00401067   B1 52              mov cl,52                          52:'R'
00401069   5E                 pop esi
0040106A   8D9B 00000000      lea ebx,dword ptr ds:[ebx]
00401070   8808               mov byte ptr ds:[eax],cl
00401072   8A4C02 01          mov cl,byte ptr ds:[edx+eax+1]
00401076   40                 inc eax
00401077   84C9               test cl,cl
00401079   75 F5              jne strcpy-optimized.401070
0040107B   8808               mov byte ptr ds:[eax],cl
0040107D   8D4424 0C          lea eax,dword ptr ss:[esp+C]
00401081   50                 push eax
00401082   68 4C814000        push strcpy-optimized.40814C       40814C:"%s\n"
00401087   E8 14000000        call strcpy-optimized.4010A0

EAX   8E02AEA1
EBX   7FFDF000
ECX   00000001
EDX   76F27084
EBP   0012FF88
ESP   0012FF18
ESI   00000000
EDI   00000000

EIP   0040103E

EFLAGS    00000282
ZF 0   PF 0   AF 0
OF 0   SF 1   DF 0
CF 0   TF 0   IF 1

LastError   000000
LastStatus  000000

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000

Address   Hex                                                ASCII
00408000  89 51 10 8E 46 AE EF 71 01 00 00 00 00 00 00 00   .Q..F®1q........
00408010  60 CB 40 00 00 00 00 00 60 CB 40 00 01 01 00 00   `Ë@.....`Ë@.....
00408020  00 00 00 00 00 00 00 00 00 10 00 00 00 00 00 00   ................
00408030  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00   ................
00408040  01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408050  00 00 00 00 00 00 00 00 00 00 00 00 02 00 00 00   ................
00408060  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00408070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................

0012FF18  0012FF78
0012FF1C  004024F0  strcpy-optimized.004024F0
0012FF20  8E50CA61
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strcpy-optimized
0012FF2C  004054B0  return to strcpy-optimized
0012FF30  0040344B  strcpy-optimized.0040344B
0012FF34  0012FF48
0012FF38  004028CE  return to strcpy-optimized
0012FF3C  0040344B  strcpy-optimized.0040344B
0012FF40  8E02AEA1
0012FF44  004012A0  return to strcpy-optimized
```

Figure 14.25: *Stack cookie on the stack*

▼Line 73-79

```
; Line 36
mov eax, DWORD PTR $SG4688


mov edx, DWORD PTR $SG4688+8
mov ecx, DWORD PTR $SG4688+4
mov DWORD PTR _src$[esp+44], eax
push esi
mov DWORD PTR _src$[esp+56], edx
```

Line 36 of the C/C++ code starts the **src** array initialization:

```
char src[] = "ReverseEngg";
```

In the ASM code, we see three **MOV** instructions to move the string **$SG4688** in three parts (4 bytes + 4 bytes + 4 bytes) to different registers. The first **MOV** instruction moves the first 4 bytes of the string (“Reve”) to another 4 bytes of the string (“ngg” + “0x00”) are moved to and then the remaining bytes (“rseE”) are moved to the **ECX** register.

In the next **MOV** instruction, the first part of the string **$SG4688** is moved to the top of the stack.

The **PUSH** instruction stores the **ESI** value on the stack, so that it can be restored later.

The last **MOV** instruction moves the third part of the string **$SG4688** stored in **EDX** to the stack at **[ESP+0x0C].**

The stack state after the execution of the preceding instruction in x32dbg will be:

[ESP] 0012FF14 00000000 ESI is saved here
[ESP+0x04] 0012FF18 65766552 “eveR” is pushed here, in little endian
[ESP+0x08] 0012FF1C 004024F0 “Eesr” will be pushed here, in little endian
[ESP+0x0C] 0012FF20 0067676E “ ggn” is pushed here, little endian
[ESP+0x10] 0012FF24 FFFFFFFE JUNK HERE

[ESP+0x14] 0012FF28 0040549C JUNK HERE
[ESP+0x18] 0012FF2C 004054B0 JUNK HERE

[ESP+0x1C] 0012FF30 0040344B JUNK HERE

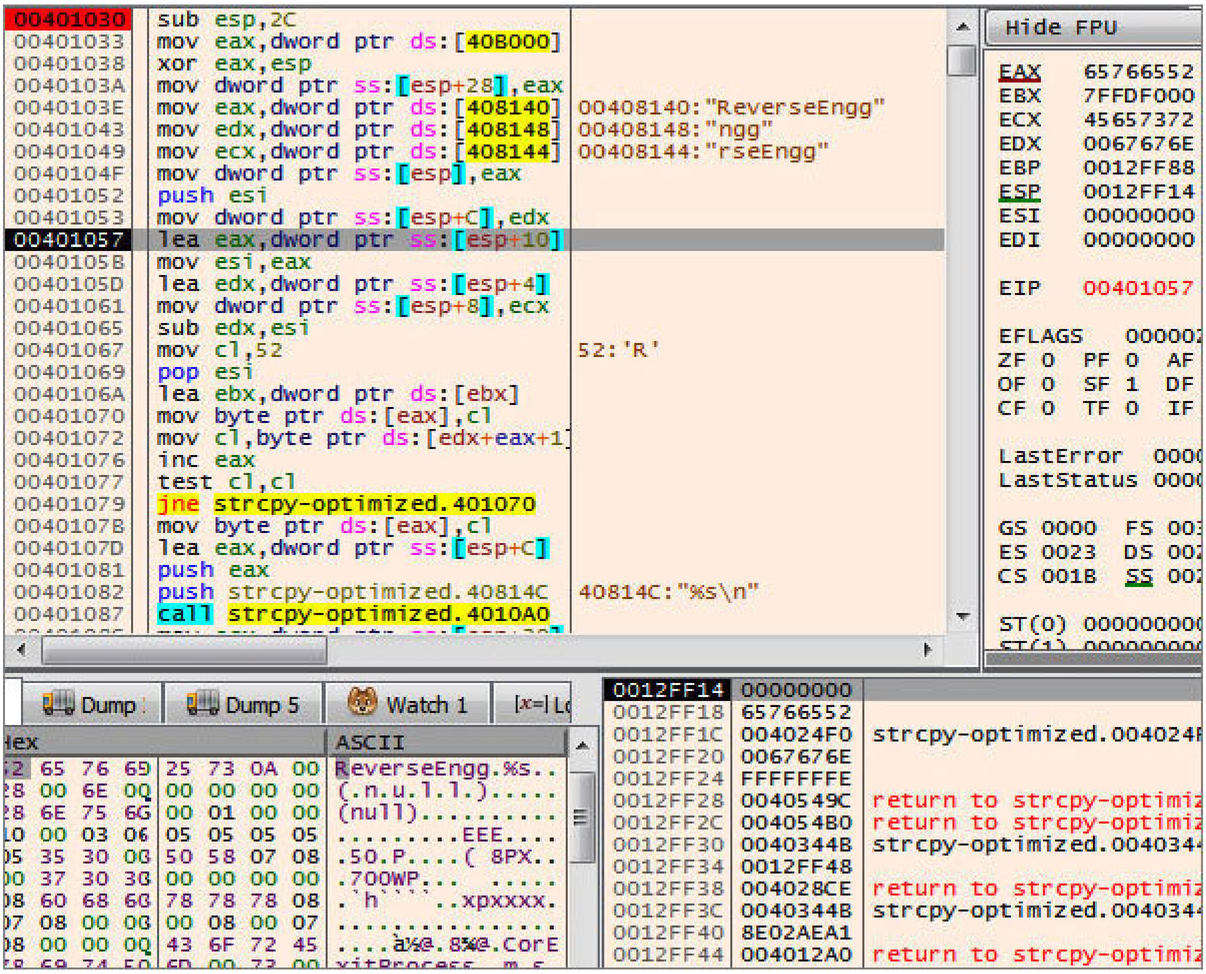
[ESP+0x20] 0012FF34 0012FF48 JUNK HERE

[ESP+0x24] 0012FF38 004028CE JUNK HERE

[ESP+0x28] 0012FF3C 0040344B JUNK HERE

[ESP+0x2C] 0012FF40 8E02AEA1 XOR of stack cookie and EBP stored here

[ESP+0x30] 0012FF44 004012A0 ESP at the start of main procedure



```
00401030  sub esp,2C
00401033  mov eax,dword ptr ds:[408000]
00401038  xor eax,esp
0040103A  mov dword ptr ss:[esp+28],eax
0040103E  mov eax,dword ptr ds:[408140]   00408140:"ReverseEngg"
00401043  mov edx,dword ptr ds:[408148]   00408148:"ngg"
00401049  mov ecx,dword ptr ds:[408144]   00408144:"rseEngg"
0040104F  mov dword ptr ss:[esp],eax
00401052  push esi
00401053  mov dword ptr ss:[esp+C],edx
00401057  lea eax,dword ptr ss:[esp+10]
0040105B  mov esi,eax
0040105D  lea edx,dword ptr ss:[esp+4]
00401061  mov dword ptr ss:[esp+8],ecx
00401065  sub edx,esi
00401067  mov cl,52                        52:'R'
00401069  pop esi
0040106A  lea ebx,dword ptr ds:[ebx]
00401070  mov byte ptr ds:[eax],cl
00401072  mov cl,byte ptr ds:[edx+eax+1]
00401076  inc eax
00401077  test cl,cl
00401079  jne strcpy-optimized.401070
0040107B  mov byte ptr ds:[eax],cl
0040107D  lea eax,dword ptr ss:[esp+C]
00401081  push eax
00401082  push strcpy-optimized.40814C    40814C:"%s\n"
00401087  call strcpy-optimized.4010A0
```

```
Hide FPU

EAX  65766552
EBX  7FFDF000
ECX  45657372
EDX  0067676E
EBP  0012FF88
ESP  0012FF14
ESI  00000000
EDI  00000000

EIP  00401057

EFLAGS   00000
ZF 0  PF 0  AF
OF 0  SF 1  DF
CF 0  TF 0  IF

LastError   000
LastStatus  000

GS 0000  FS 00
ES 0023  DS 00
CS 001B  SS 00

ST(0) 00000000
```

```
Hex                       ASCII
2 65 76 69 25 73 0A 00    ReverseEngg.%s..
8 00 6E 00 00 00 00 00    (.n.u.l.l.).....
8 6E 75 6C 00 01 00 00    (null)..........
0 00 03 06 05 05 05 05    ..........EEE....
5 35 30 00 50 58 07 08    .50.P....( 8PX..
0 37 30 30 00 00 00 00    .700WP...  .....
8 60 68 60 78 78 78 08    .`h````..xpxxxx.
7 08 00 00 00 08 00 07    ................
8 00 00 00 43 6F 72 45    ....à½@.8½@.CorE
```

```
0012FF14  00000000
0012FF18  65766552
0012FF1C  004024F0  strcpy-optimized.004024
0012FF20  0067676E
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strcpy-optimiz
0012FF2C  004054B0  return to strcpy-optimiz
0012FF30  0040344B  strcpy-optimized.004034
0012FF34  0012FF48
0012FF38  004028CE  return to strcpy-optimiz
0012FF3C  0040344B  strcpy-optimized.004034
0012FF40  8E02AEA1
0012FF44  004012A0  return to strcpy-optimiz
```

Figure 14.26: *String $SG4688 on stack*

▼Line 80-88

; Line 39

```
lea eax, DWORD PTR _dest$[esp+48]

mov esi, eax
lea edx, DWORD PTR _src$[esp+48]
mov DWORD PTR _src$[esp+52], ecx
sub edx, esi
mov cl, 82        ; 00000052H
pop esi
npad 6
```

Line 39 of the C/C++ code starts the **printf** function.

```
printf("%s\n", xstrcpy(dest, src));
```

In the ASM section, the **Load Effective Address** instruction loads the address **[ESP+0x10]** form the **dest** array to the **EAX** register. This allocates space for the **dest** array on the stack. **EAX** is then moved to the **ESI** register using the **MOV** instruction. This is to store the address of the **dest** array in the ESI register and at this address is where the **src** array will be copied. **ESI** will become

The next **LEA** instruction does the same for the **src** array. It loads the address of the **src** array **[ESP+0x04]** to the **EDX** register. **EDX** will become

The **MOV** instruction moves the second part of the string **$SG4688** stored in **ECX** to the stack at **[ESP+0x08].**

The **SUB** instruction will calculate the offset between the **dest** array and the **src** array by subtracting **EDX** from The resultant offset will be saved in **EDX** for later calculations.

ESI=0x0012FF24
SUB= EDX-ESI = 0xFFFFFFF4

The **CL** register is loaded with the first byte 52(“R”) of the **src** array using the **MOV** instruction.

Using **POP ESI** is restored back to the old value.

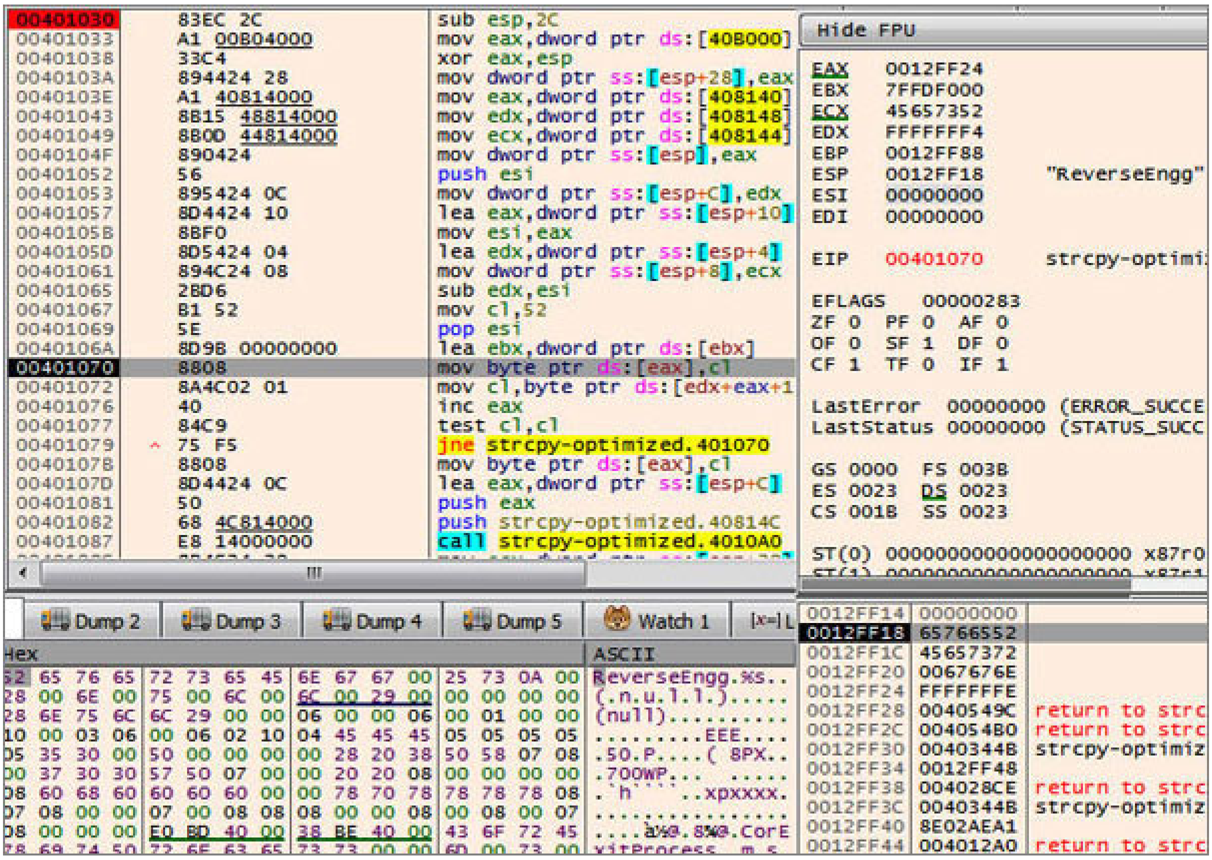
Next is the **npad** macro, which inserts the non-destructive, non-operational instructions. For information on please refer to the Appendix section. Our assembly listing uses the **npad** which is defined in **LISTING.INC** as **lea ebx, [ebx+00000000].** The stack state after the npad macro is as follows:

[ESP-0x04] 0012FF14 00000000 JUNK HERE
[ESP] 0012FF18 65766552 “eveR” is pushed here, in little endian
[ESP+0x04] 0012FF1C 45657372 “Eesr” will be pushed here, in little endian
[ESP+0x08] 0012FF20 0067676E “ ggn” is pushed here, in little endian
[ESP+0x0C] 0012FF24 FFFFFFFE Space for dest array
[ESP+0x10] 0012FF28 0040549C JUNK HERE
[ESP+0x14] 0012FF2C 004054B0 JUNK HERE
[ESP+0x18] 0012FF30 0040344B JUNK HERE
[ESP+0x1C] 0012FF34 0012FF48 JUNK HERE
[ESP+0x20] 0012FF38 004028CE JUNK HERE

[ESP+0x24] 0012FF3C 0040344B JUNK HERE

[ESP+0x28] 0012FF40 8E02AEA1 XOR of stack cookie and EBP stored here

[ESP+0x2C] 0012FF44 004012A0 ESP at the start of main procedure



```
00401030    83EC 2C          sub esp,2C
00401033    A1 00B04000      mov eax,dword ptr ds:[40B000]
00401038    33C4             xor eax,esp
0040103A    894424 28        mov dword ptr ss:[esp+28],eax
0040103E    A1 40814000      mov eax,dword ptr ds:[408140]
00401043    8B15 48814000    mov edx,dword ptr ds:[408148]
00401049    8B0D 44814000    mov ecx,dword ptr ds:[408144]
0040104F    890424           mov dword ptr ss:[esp],eax
00401052    56               push esi
00401053    895424 0C        mov dword ptr ss:[esp+C],edx
00401057    8D4424 10        lea eax,dword ptr ss:[esp+10]
0040105B    8BF0             mov esi,eax
0040105D    8D5424 04        lea edx,dword ptr ss:[esp+4]
00401061    894C24 08        mov dword ptr ss:[esp+8],ecx
00401065    2BD6             sub edx,esi
00401067    B1 52            mov cl,52
00401069    5E               pop esi
0040106A    8D9B 00000000    lea ebx,dword ptr ds:[ebx]
00401070    8808             mov byte ptr ds:[eax],cl
00401072    8A4C02 01        mov cl,byte ptr ds:[edx+eax+1
00401076    40               inc eax
00401077    84C9             test cl,cl
00401079  ^ 75 F5            jne strcpy-optimized.401070
0040107B    8808             mov byte ptr ds:[eax],cl
0040107D    8D4424 0C        lea eax,dword ptr ss:[esp+C]
00401081    50               push eax
00401082    68 4C814000      push strcpy-optimized.40814C
00401087    E8 14000000      call strcpy-optimized.4010A0
```

```
Hide FPU

EAX    0012FF24
EBX    7FFDF000
ECX    45657352
EDX    FFFFFFF4
EBP    0012FF88
ESP    0012FF18        "ReverseEngg"
ESI    00000000
EDI    00000000

EIP    00401070        strcpy-optimi

EFLAGS    00000283
ZF 0   PF 0   AF 0
OF 0   SF 1   DF 0
CF 1   TF 0   IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000 x87r0
```

```
Hex                                                  ASCII
52 65 76 65 72 73 65 45 6E 67 67 00 25 73 0A 00  ReverseEngg.%s..
28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
05 35 30 00 50 00 00 00 00 28 20 38 50 58 07 08  .50.P....( 8PX..
00 37 30 30 57 50 07 00 00 20 20 08 00 00 00 00  .700WP...  .....
08 60 68 60 60 60 60 00 00 78 70 78 78 78 78 08  .`h````..xpxxxx.
07 08 00 00 07 00 08 08 08 00 00 08 00 08 00 07  ................
08 00 00 00 E0 BD 40 00 38 BE 40 00 43 6F 72 45  ....à½@.8¾@.CorE
```

```
0012FF14  00000000
0012FF18  65766552
0012FF1C  45657372
0012FF20  0067676E
0012FF24  FFFFFFFE
0012FF28  0040549C  return to strc
0012FF2C  004054B0  return to strc
0012FF30  0040344B  strcpy-optimiz
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  0040344B  strcpy-optimiz
0012FF40  8E02AEA1
0012FF44  004012A0  return to strc
```

Figure 14.27: Stack state after npad

▼Line 89-94

$LL4@main:
mov BYTE PTR [eax], cl
mov cl, BYTE PTR [edx+eax+1]
inc eax

```
test cl, cl
jne SHORT $LL4@main
```

This piece of ASM code is where the **src** array is copied to the **dest** array memory location byte-by-byte. The first **MOV** instruction copies the "R" (first char or byte) in the **src** array to the start of the **dest** array memory location, The stack with the **src** and **dest** array can be visualized as follows:

follows: follows: follows:

follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows:
follows:

Table 14.1: *Stack with src and dest array - Part-1*

Once the first byte is copied from the **src** array to the **dest** array, the CL register is filled with the next byte of the **src** array using the second **MOV** instruction with calculations explained as follows:

```
mov cl, BYTE PTR [edx+eax+1]
```

EDX = 0XFFFFFFF4

EAX = 0x0012FF24 (this is the memory address of the start of dest array)
EDX+EAX+1 = 0x0012FF19

Now, the byte will be copied from **0x0012FF19** (which is “e” or 0x65) to the **CL** register.

As the byte copied to the **CL** register needs to be moved to the next memory location, **EAX** is incremented by 1 using the **INC** instruction. **EAX** will become

The **TEST** instruction performs the **AND** of CL with itself, resulting in a non-zero output in CL and ZF=0. So, a jump to the label **$LL4@main** will take place. The next iteration of this same ASM code results in copying the next byte to the **dest** array. The stack state will become as follows:

follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows:
follows:

Table 14.2: *Stack with src and dest array - Part-2*

Now, imagine the iteration where we reach the following stack state where we copy all bytes from the **src** array to the **dest** array till 0x67 or “g”.

“g”. “g”. “g”.
“g”. “g”. “g”.
“g”. “g”. “g”.
“g”. “g”. “g”.
“g”. “g”. “g”.
“g”. “g”. “g”.

Table 14.3: *Stack with src and dest array - Part-3*

In this iteration, 0x67 or “g” is copied at the **0x0012FF2E** memory location with the first **MOV** instruction. At this point:

EAX=0x0012FF2E
EDX=0xFFFFFFF4

So, the second **MOV** instruction will be evaluated to:

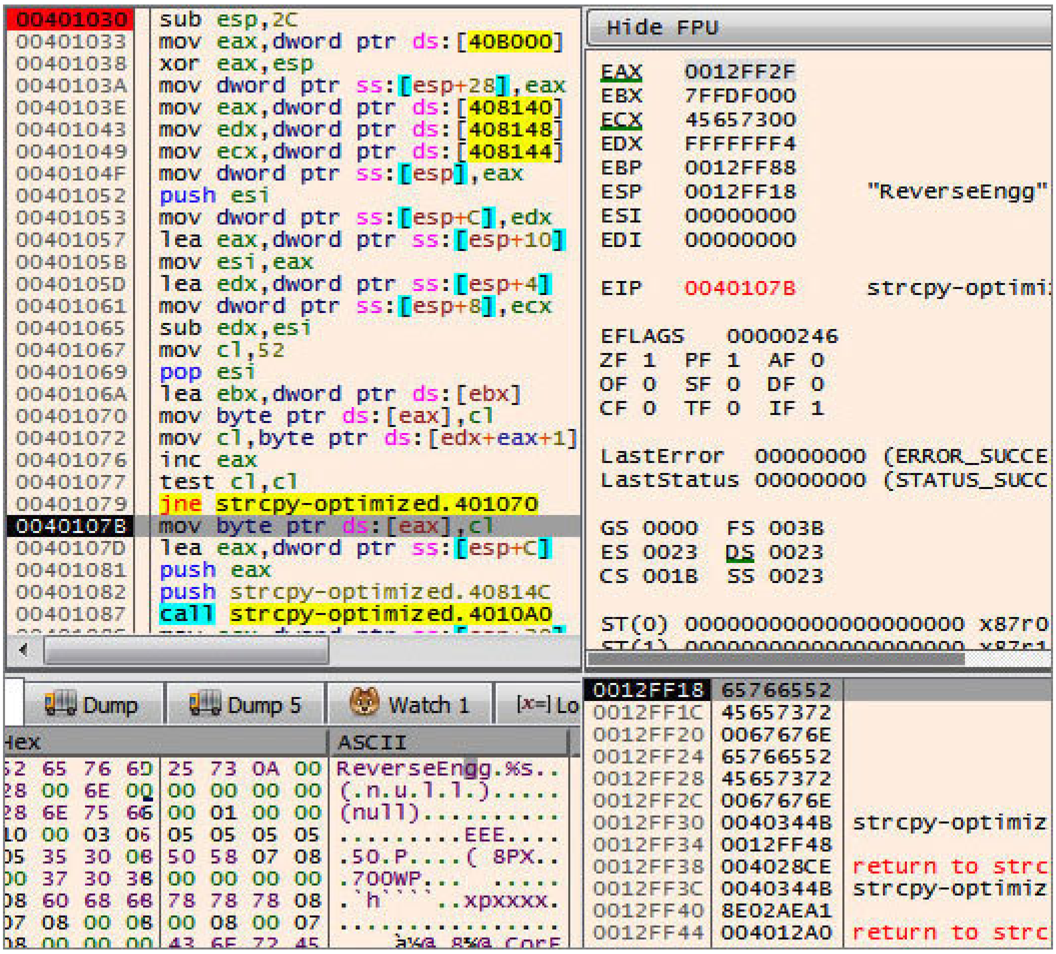
mov cl, byte ptr ds:[edx+eax*1+0x1]
mov cl, byte ptr ds:[0x0012FF23]

This will copy 0x00 to the **CL** register. The next **INC** instruction will increment **EAX** to

The **TEST** instruction this time will set ZF=1 as CL=0x00, stopping the jump instruction and breaking the loop to move on to the next instruction. The stack state after this part of the ASM code will be:

[ESP-0x04] 0012FF14 00000000 JUNK HERE
[ESP] 0012FF18 65766552 “eveR” is pushed here, in little endian
[ESP+0x04] 0012FF1C 45657372 “Eesr” will be pushed here, in little endian
[ESP+0x08] 0012FF20 0067676E “ ggn” is pushed here, in little endian
[ESP+0x0C] 0012FF24 65766552 “eveR” is pushed here in dest array
[ESP+0x10] 0012FF28 45657372 “Eesr” will be pushed here in dest array
[ESP+0x14] 0012FF2C 0067676E “ggn” is pushed here in dest array
[ESP+0x18] 0012FF30 0040344B JUNK HERE
[ESP+0x1C] 0012FF34 0012FF48 JUNK HERE

[ESP+0x20] 0012FF38 004028CE JUNK HERE
[ESP+0x24] 0012FF3C 0040344B JUNK HERE
[ESP+0x28] 0012FF40 8E02AEA1 XOR of stack cookie and EBP stored here
[ESP+0x2C] 0012FF44 004012A0 ESP at the start of main procedure



```
00401030  sub esp,2C
00401033  mov eax,dword ptr ds:[408000]
00401038  xor eax,esp
0040103A  mov dword ptr ss:[esp+28],eax
0040103E  mov eax,dword ptr ds:[408140]
00401043  mov edx,dword ptr ds:[408148]
00401049  mov ecx,dword ptr ds:[408144]
0040104F  mov dword ptr ss:[esp],eax
00401052  push esi
00401053  mov dword ptr ss:[esp+C],edx
00401057  lea eax,dword ptr ss:[esp+10]
0040105B  mov esi,eax
0040105D  lea edx,dword ptr ss:[esp+4]
00401061  mov dword ptr ss:[esp+8],ecx
00401065  sub edx,esi
00401067  mov cl,52
00401069  pop esi
0040106A  lea ebx,dword ptr ds:[ebx]
00401070  mov byte ptr ds:[eax],cl
00401072  mov cl,byte ptr ds:[edx+eax+1]
00401076  inc eax
00401077  test cl,cl
00401079  jne strcpy-optimized.401070
0040107B  mov byte ptr ds:[eax],cl
0040107D  lea eax,dword ptr ss:[esp+C]
00401081  push eax
00401082  push strcpy-optimized.40814C
00401087  call strcpy-optimized.4010A0
```

```
EAX   0012FF2F
EBX   7FFDF000
ECX   45657300
EDX   FFFFFFF4
EBP   0012FF88
ESP   0012FF18      "ReverseEngg"
ESI   00000000
EDI   00000000

EIP   0040107B      strcpy-optimi

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError   00000000 (ERROR_SUCCE
LastStatus  00000000 (STATUS_SUCC

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000 x87r0
```

```
2 65 76 65 | 25 73 0A 00 | ReverseEngg.%s..
8 00 6E 00 | 00 00 00 00 | (.n.u.l.l.).....
8 6E 75 66 | 00 01 00 00 | (null)..........
0 00 03 06 | 05 05 05 05 | .........EEE....
5 35 30 08 | 50 58 07 08 | .50.P....( 8PX..
0 37 30 38 | 00 00 00 00 | .700WP...  .....
8 60 68 68 | 78 78 78 08 | .`h````..xpxxxx.
7 08 00 08 | 00 08 00 07 | ................
```

```
0012FF18  65766552
0012FF1C  45657372
0012FF20  0067676E
0012FF24  65766552
0012FF28  45657372
0012FF2C  0067676E
0012FF30  0040344B  strcpy-optimiz
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  0040344B  strcpy-optimiz
0012FF40  8E02AEA1
0012FF44  004012A0  return to strc
```

Figure 14.28: Stack with src and dest array

As we saw in the preceding instructions, the **dest** array is not NULL terminated as we move out of the loop before NULL is copied to The answer lies in the next instruction. Let's move on to the next instructions:

▼Line 95-99

mov BYTE PTR [eax], cl
lea eax, DWORD PTR _dest$[esp+44]

push eax
push OFFSET $SG4690

call _printf

In the first preceding **MOV** instruction, the ASM code moves oxoo byte from the **CL** register to This will NULL terminate the **dest** array shown as follows:

follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows: follows: follows:
follows: follows: follows:

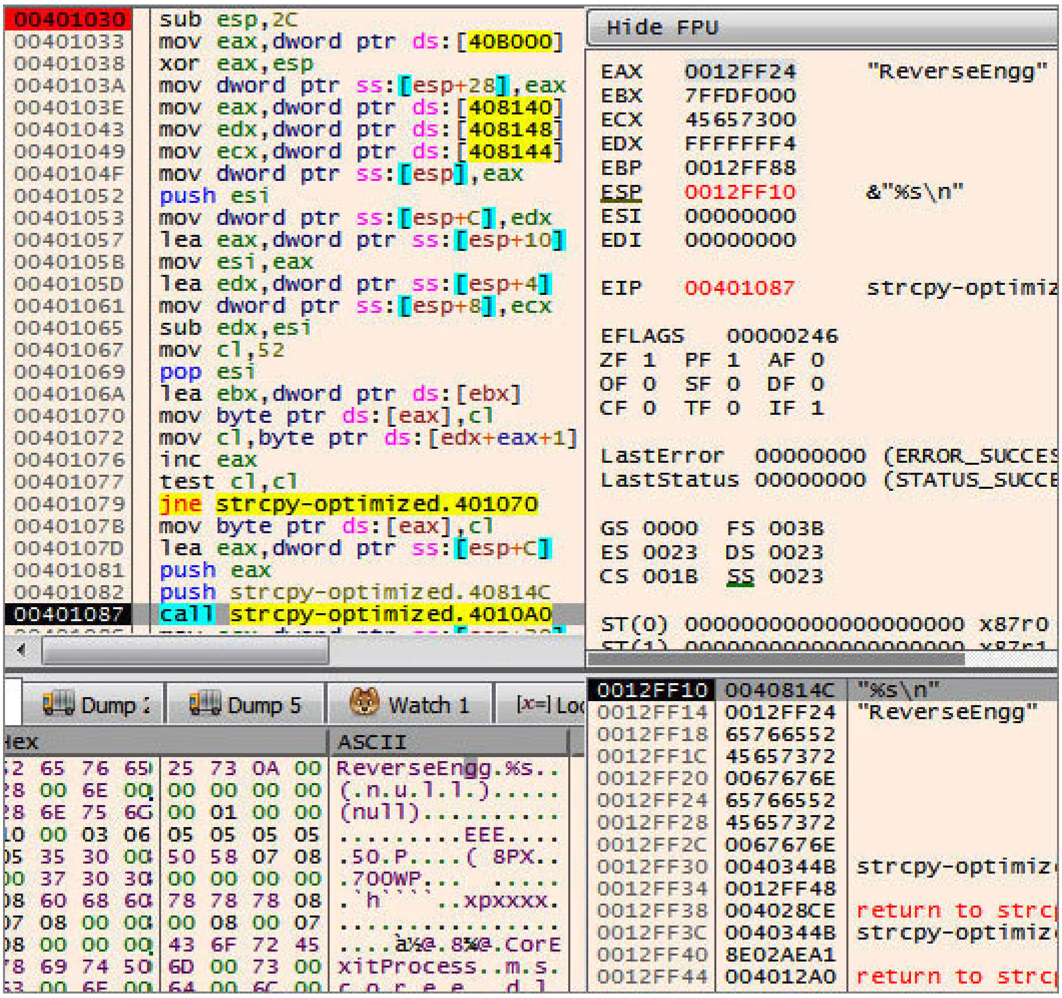
Table 14.4: *Stack with the src and dest array - Part-4*

Now, we have a pointer to the **src** array and a pointer to the **dest** array on the stack. Next, we have to call the **printf** function for which the arguments to **printf** need to be pushed onto the stack.

The **LEA** instruction moves the pointer from the **dest** array to The **PUSH** instruction pushes the first argument onto the stack by **PUSH**

The second **PUSH** instruction pushes the string constant onto the stack, which is the second argument to Once both arguments are pushed, a call to **printf** is made. The stack state before the execution of the **CALL** instruction is as follows:

```
[ESP]  0012FF10  0040814C  "%s\n", second argument to printf is pushed here
[ESP+0x04] 0012FF14  0012FF24  "ReverseEngg", second argument to printf()is pushed
[ESP+0x08] 0012FF18  65766552  “eveR” is pushed here, in little endian
[ESP+0x0C] 0012FF1C  45657372  “Eesr” will be pushed here, in little endian
[ESP+0x10] 0012FF20  0067676E  “ ggn” is pushed here, in little endian
[ESP+0x14] 0012FF24  65766552  “eveR” is pushed here in dest array
0012FF28  45657372  “Eesr” will be pushed here in dest array
[ESP+0x1C] 0012FF2C  0067676E  “ggn” is pushed here in dest array
[ESP+0x20] 0012FF30  0040344B  JUNK HERE
[ESP+0x24] 0012FF34  0012FF48  JUNK HERE
[ESP+0x28] 0012FF38  004028CE  JUNK HERE
[ESP+0x2C] 0012FF3C  0040344B  JUNK HERE
[ESP+0x30] 0012FF40  8E02AEA1  XOR of stack cookie and EBP stored here
[ESP+0x34] 0012FF44  004012A0  ESP at the start of main procedure
```



```
00401030  sub esp,2C
00401033  mov eax,dword ptr ds:[40B000]
00401038  xor eax,esp
0040103A  mov dword ptr ss:[esp+28],eax
0040103E  mov eax,dword ptr ds:[408140]
00401043  mov edx,dword ptr ds:[408148]
00401049  mov ecx,dword ptr ds:[408144]
0040104F  mov dword ptr ss:[esp],eax
00401052  push esi
00401053  mov dword ptr ss:[esp+C],edx
00401057  lea eax,dword ptr ss:[esp+10]
0040105B  mov esi,eax
0040105D  lea edx,dword ptr ss:[esp+4]
00401061  mov dword ptr ss:[esp+8],ecx
00401065  sub edx,esi
00401067  mov cl,52
00401069  pop esi
0040106A  lea ebx,dword ptr ds:[ebx]
00401070  mov byte ptr ds:[eax],cl
00401072  mov cl,byte ptr ds:[edx+eax+1]
00401076  inc eax
00401077  test cl,cl
00401079  jne strcpy-optimized.401070
0040107B  mov byte ptr ds:[eax],cl
0040107D  lea eax,dword ptr ss:[esp+C]
00401081  push eax
00401082  push strcpy-optimized.40814C
00401087  call strcpy-optimized.4010A0
```

```
EAX   0012FF24     "ReverseEngg"
EBX   7FFDF000
ECX   45657300
EDX   FFFFFFF4
EBP   0012FF88
ESP   0012FF10     &"%s\n"
ESI   00000000
EDI   00000000

EIP   00401087     strcpy-optimiz

EFLAGS    00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCES
LastStatus 00000000 (STATUS_SUCCE

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0
```

```
2 65 76 65 | 25 73 0A 00 | ReverseEngg.%s..
8 00 6E 00 | 00 00 00 00 | (.n.u.l.l.).....
8 6E 75 6C | 00 01 00 00 | (null)..........
0 00 03 06 | 05 05 05 05 | ...........EEE....
5 35 30 00 | 50 58 07 08 | .50.P....( 8PX..
0 37 30 30 | 00 00 00 00 | .700WP...  .....
8 60 68 60 | 78 78 78 08 | .`h````..xpxxxx.
7 08 00 00 | 00 08 00 07 | ................
8 00 00 00 | 43 6F 72 45 | ....àx@.8x@.CorE
8 69 74 50 | 6D 00 73 00 | xitProcess..m.s.
```

```
0012FF10  0040814C  "%s\n"
0012FF14  0012FF24  "ReverseEngg"
0012FF18  65766552
0012FF1C  45657372
0012FF20  0067676E
0012FF24  65766552
0012FF28  45657372
0012FF2C  0067676E
0012FF30  0040344B  strcpy-optimiz
0012FF34  0012FF48
0012FF38  004028CE  return to strc
0012FF3C  0040344B  strcpy-optimiz
0012FF40  8E02AEA1
0012FF44  004012A0  return to strc
```

Figure 14.29: *Stack before CALL instruction*

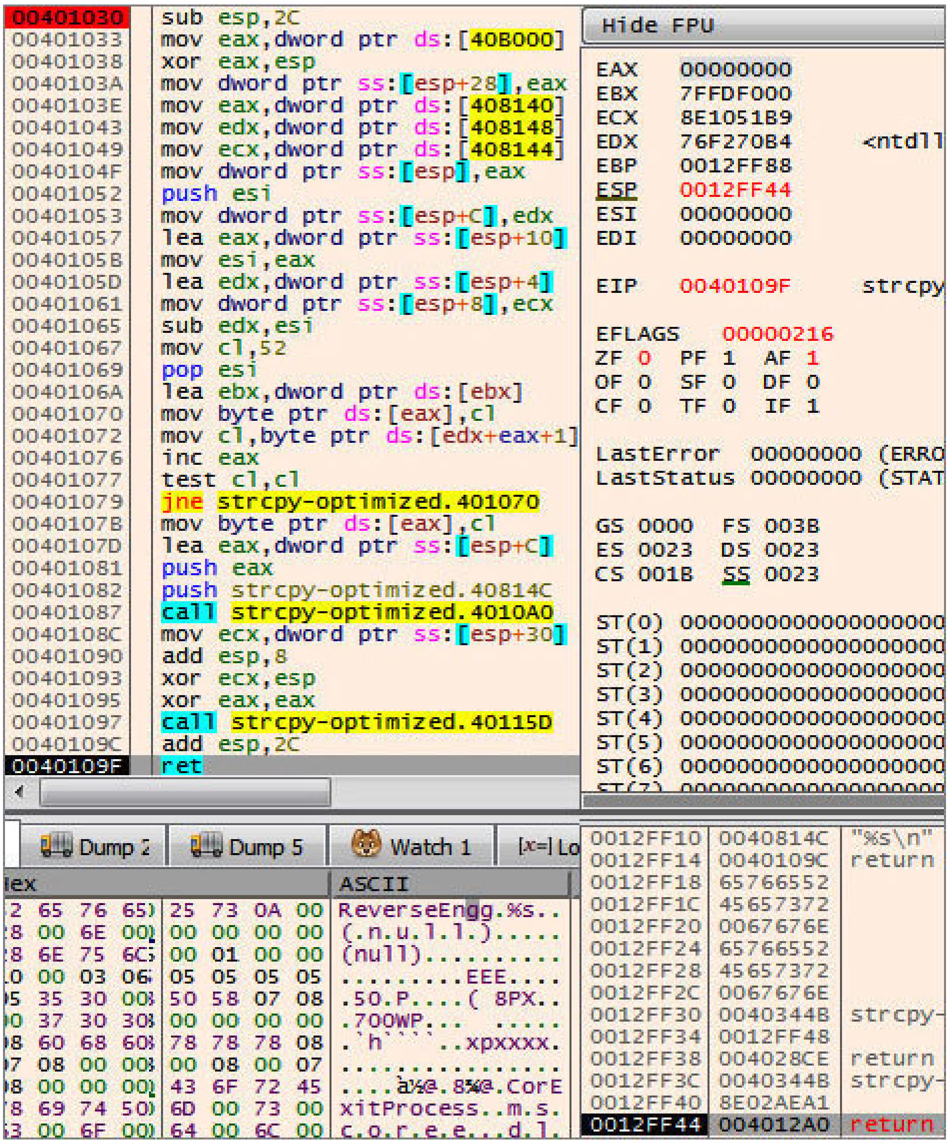
▼Line 100-110

```
; Line 42
mov ecx, DWORD PTR __$ArrayPad$[esp+52]
add esp, 8
xor ecx, esp
xor eax, eax
call @__security_check_cookie@4
add esp, 44          ; 0000002cH
ret 0
_main ENDP
```

```
_TEXT ENDS

END
```

In the preceding ASM code, the **MOV** instruction moves the stack cookie stored at **[ESP+0x30]** to where it is XOR'ed with **ECX** to check the buffer overflow condition by calling the **security_check_cookie** procedure. The rest of the **ADD** instructions clean up the stack to end the **main** procedure, TEXT segment, and code. The stack state before the **RET** instruction is as follows:

```
[ESP-0x34] 0012FF10 0040814C Now JUNK HERE
[ESP-0x30] 0012FF14 0012FF24 Now JUNK HERE
[ESP-0x2C] 0012FF18 65766552 Now JUNK HERE
[ESP-0x28] 0012FF1C 45657372 Now JUNK HERE
[ESP-0x24] 0012FF20 0067676E Now JUNK HERE
[ESP-0x20] 0012FF24 65766552 Now JUNK HERE
[ESP-0x1C] 0012FF28 45657372 Now JUNK HERE
[ESP-0x18] 0012FF2C 0067676E Now JUNK HERE
[ESP-0x14] 0012FF30 0040344B Now JUNK HERE
[ESP-0x10] 0012FF34 0012FF48 Now JUNK HERE
[ESP-0x0C] 0012FF38 004028CE Now JUNK HERE
[ESP-0x08] 0012FF3C 0040344B Now JUNK HERE
[ESP-0x04] 0012FF40 8E02AEA1 Now JUNK HERE
[ESP] 0012FF44 004012A0 ESP at the start of main procedure
```



```
00401030  sub esp,2C
00401033  mov eax,dword ptr ds:[40B000]
00401038  xor eax,esp
0040103A  mov dword ptr ss:[esp+28],eax
0040103E  mov eax,dword ptr ds:[408140]
00401043  mov edx,dword ptr ds:[408148]
00401049  mov ecx,dword ptr ds:[408144]
0040104F  mov dword ptr ss:[esp],eax
00401052  push esi
00401053  mov dword ptr ss:[esp+C],edx
00401057  lea eax,dword ptr ss:[esp+10]
0040105B  mov esi,eax
0040105D  lea edx,dword ptr ss:[esp+4]
00401061  mov dword ptr ss:[esp+8],ecx
00401065  sub edx,esi
00401067  mov cl,52
00401069  pop esi
0040106A  lea ebx,dword ptr ds:[ebx]
00401070  mov byte ptr ds:[eax],cl
00401072  mov cl,byte ptr ds:[edx+eax+1]
00401076  inc eax
00401077  test cl,cl
00401079  jne strcpy-optimized.401070
0040107B  mov byte ptr ds:[eax],cl
0040107D  lea eax,dword ptr ss:[esp+C]
00401081  push eax
00401082  push strcpy-optimized.40814C
00401087  call strcpy-optimized.4010A0
0040108C  mov ecx,dword ptr ss:[esp+30]
00401090  add esp,8
00401093  xor ecx,esp
00401095  xor eax,eax
00401097  call strcpy-optimized.40115D
0040109C  add esp,2C
0040109F  ret
```

```
EAX  00000000
EBX  7FFDF000
ECX  8E1051B9
EDX  76F270B4     <ntdll
EBP  0012FF88
ESP  0012FF44
ESI  00000000
EDI  00000000

EIP  0040109F     strcpy

EFLAGS    00000216
ZF 0   PF 1   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERRO
LastStatus  00000000 (STAT

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
```

```
lex                 ASCII
2 65 76 65  25 73 0A 00  ReverseEngg.%s..
8 00 6E 00  00 00 00 00  (.n.u.l.l.).....
8 6E 75 6C  00 01 00 00  (null)..........
0 00 03 06  05 05 05 05  ..........EEE....
5 35 30 00  50 58 07 08  .50.P....( 8PX..
0 37 30 30  00 00 00 00  .700WP...  .....
8 60 68 60  78 78 78 08  .`h```..xpxxxx.
7 08 00 00  00 08 00 07  ................
8 00 00 00  43 6F 72 45  ....à½@.8½@.CorE
8 69 74 50  6D 00 73 00  xitProcess..m.s.
3 00 6F 00  64 00 6C 00  c.o.r.e.e...d.l.
```

```
0012FF10  0040814C  "%s\n"
0012FF14  0040109C  return
0012FF18  65766552
0012FF1C  45657372
0012FF20  0067676E
0012FF24  65766552
0012FF28  45657372
0012FF2C  0067676E
0012FF30  0040344B  strcpy-
0012FF34  0012FF48
0012FF38  004028CE  return
0012FF3C  0040344B  strcpy-
0012FF40  8E02AEA1
0012FF44  004012A0  return
```

Figure 14.30: *Stack cleaned*

Conclusion

In this chapter, we covered the strcpy function implementation by copying data from one memory location to another with respect to reverse engineering. We also learned about byte-by-byte operations that happen during the strcpy execution. In the optimized assembly listing, during the stack cookie operation, the XOR procedure happens with ESP and in a non-optimized assembly listing, it happens with EBP. We also covered the difference between optimized and non-optimized codes of strcpy program assembly pattern. In the next chapter, we will talk about another real-world example, wherein we will write a program to calculate simple interest code.

CHAPTER 15

Simple Interest Code Pattern in Reverse Engineering

We all dream about buying a beautiful house and a nice car. Now to buy any of these, we need money which can either be earned with our skills or it can be borrowed from a bank. This is where terms like interest come into picture. Imaging a person named Atul Narula who works for International Institute of Cyber Security and wants to buy a car. Now, Atul has savings of 10,000 dollars and decides to take the remaining 10,000 dollars from a bank. A bank offers him a loan on an annual simple interest rate of 2 percent for a term of 5 years. Atul goes to the bank and asks the bank official about the interest that he has to pay over the term of the loan. The bank official uses a software which uses a mathematical formula to calculate interest. The mathematical formula that goes inside the software will be $10,000 times 2 percent times 5 years, or 10,000 x .02 x 5. This is the interest amount Atul has to pay over the entire term of the loan.

Now imagine, as a reverse engineer, you have to extract the mathematical formula of this banking software without any access to the software code. In this chapter, we will use this real-life software example to understand the assembly pattern of such kinds of software. This type of reverse engineering will help you understand some patterns used by malware writers.

Structure

In this chapter, we will cover the following topics:

Program to calculate simple interest

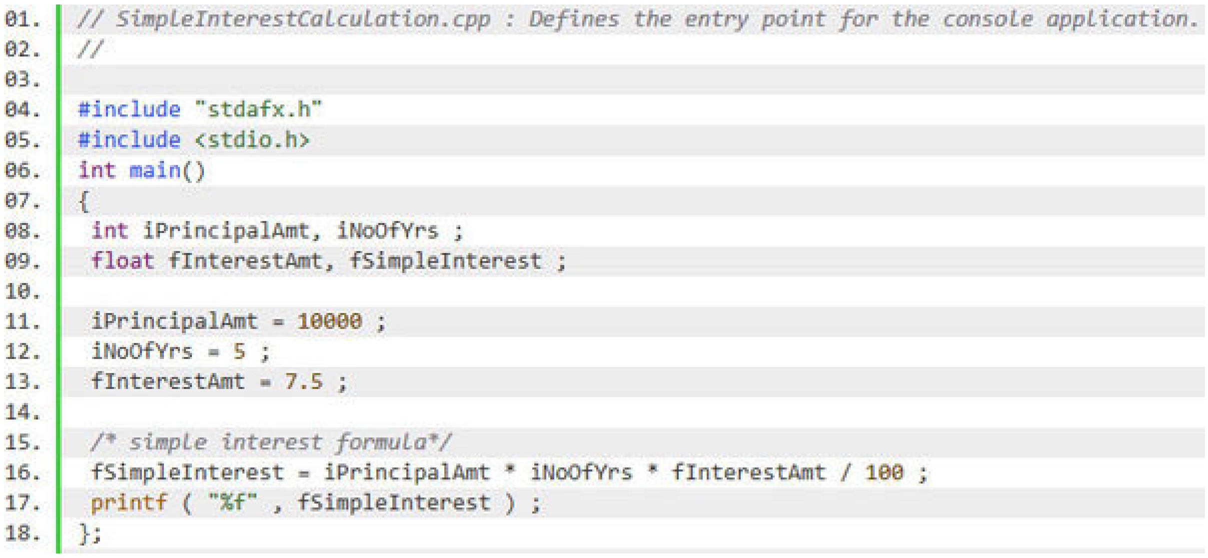
Calculate simple interest without Optimization

Objective

In this chapter, we will talk about a real-life software that uses an internal mathematical formula to calculate simple interest. We will find out how an integer and float are together handled in memory. Reverse engineering a software with mathematical calculation will help us understand the assembly pattern of the software along with the mathematical calculations.

Program to Calculate Simple Interest

Let’s write down a C/C++ program to calculate simple interest from a set of values represented by Principal Amount, Interest Amount, and Number of Years.



```
// SimpleInterestCalculation.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include <stdio.h>
int main()
{
 int iPrincipalAmt, iNoOfYrs ;
 float fInterestAmt, fSimpleInterest ;

 iPrincipalAmt = 10000 ;
 iNoOfYrs = 5 ;
 fInterestAmt = 7.5 ;

 /* simple interest formula*/
 fSimpleInterest = iPrincipalAmt * iNoOfYrs * fInterestAmt / 100 ;
 printf ( "%f" , fSimpleInterest ) ;
};
```

Figure 15.1: *SimpleInterestCalculation.cpp*

Calculate Simple Interest Without Optimization

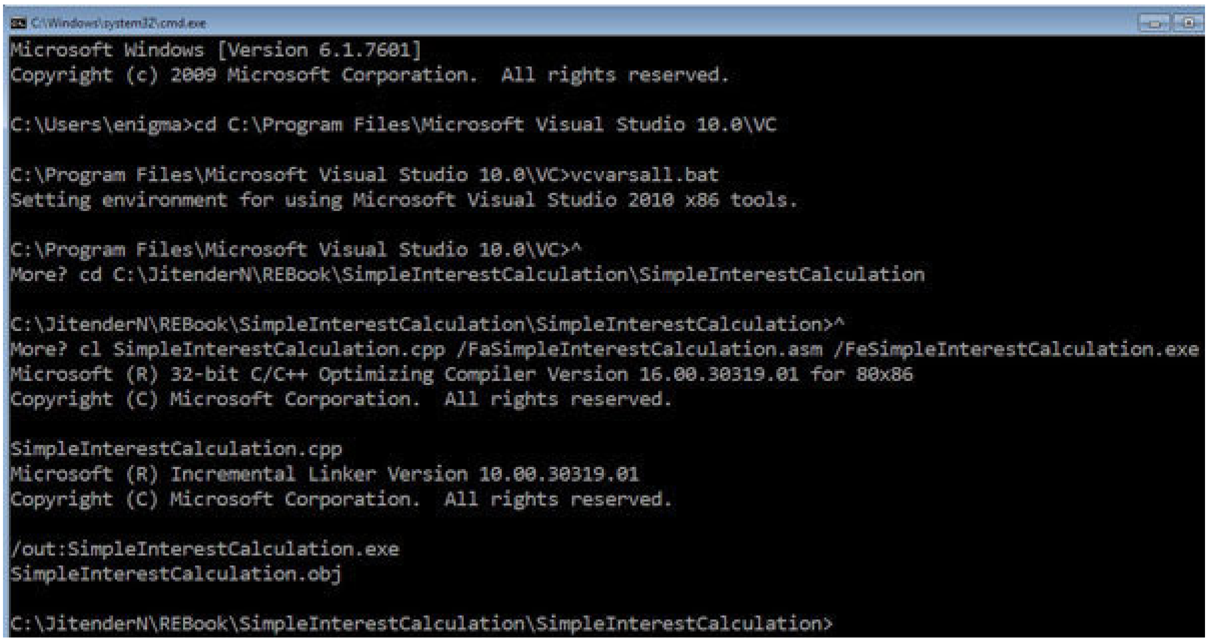
Compile the code without optimization in the MSVC compiler on the same x86 Windows machine. Run the following commands on the Windows command prompt to set the environment for **cl.exe** (VS compiler), and then compile the code with the following switches:

Name of the output assembly listing file

Name of the output executable file

file file file file file file file file file file file file file file file

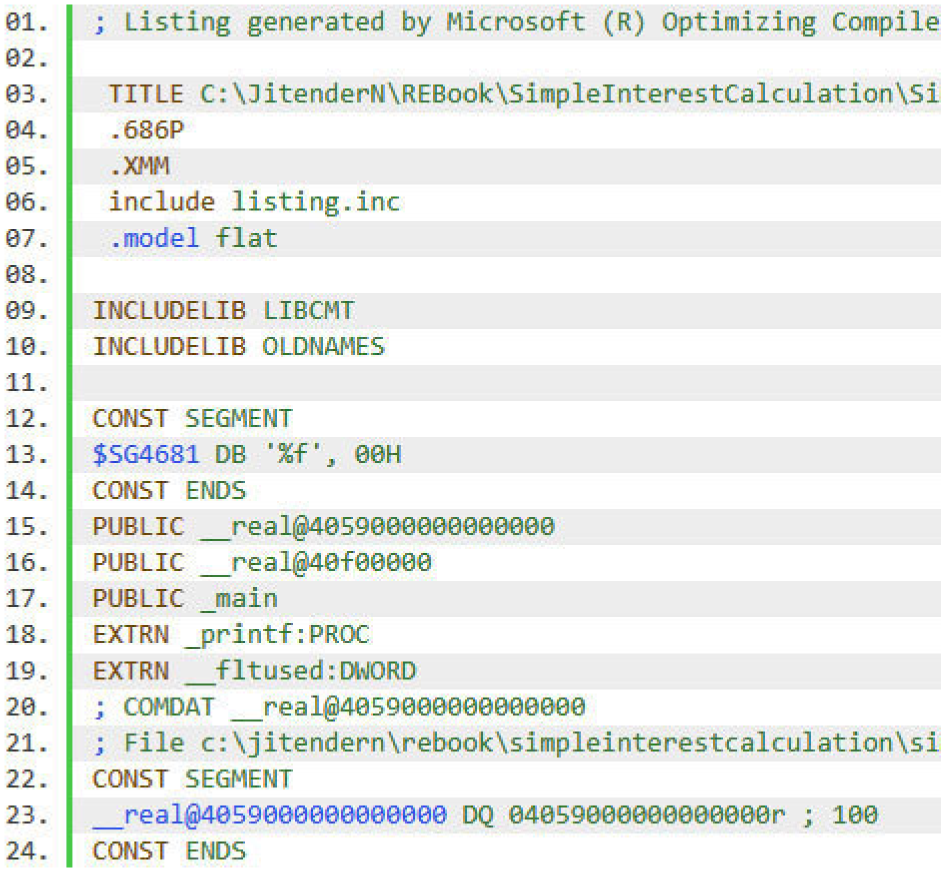
The following is the output of running the preceding commands:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\enigma>cd C:\Program Files\Microsoft Visual Studio 10.0\VC

C:\Program Files\Microsoft Visual Studio 10.0\VC>vcvarsall.bat
Setting environment for using Microsoft Visual Studio 2010 x86 tools.

C:\Program Files\Microsoft Visual Studio 10.0\VC>^
More? cd C:\JitenderN\REBook\SimpleInterestCalculation\SimpleInterestCalculation

C:\JitenderN\REBook\SimpleInterestCalculation\SimpleInterestCalculation>^
More? cl SimpleInterestCalculation.cpp /FaSimpleInterestCalculation.asm /FeSimpleInterestCalculation.exe
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 16.00.30319.01 for 80x86
Copyright (C) Microsoft Corporation.  All rights reserved.

SimpleInterestCalculation.cpp
Microsoft (R) Incremental Linker Version 10.00.30319.01
Copyright (C) Microsoft Corporation.  All rights reserved.

/out:SimpleInterestCalculation.exe
SimpleInterestCalculation.obj

C:\JitenderN\REBook\SimpleInterestCalculation\SimpleInterestCalculation>
```

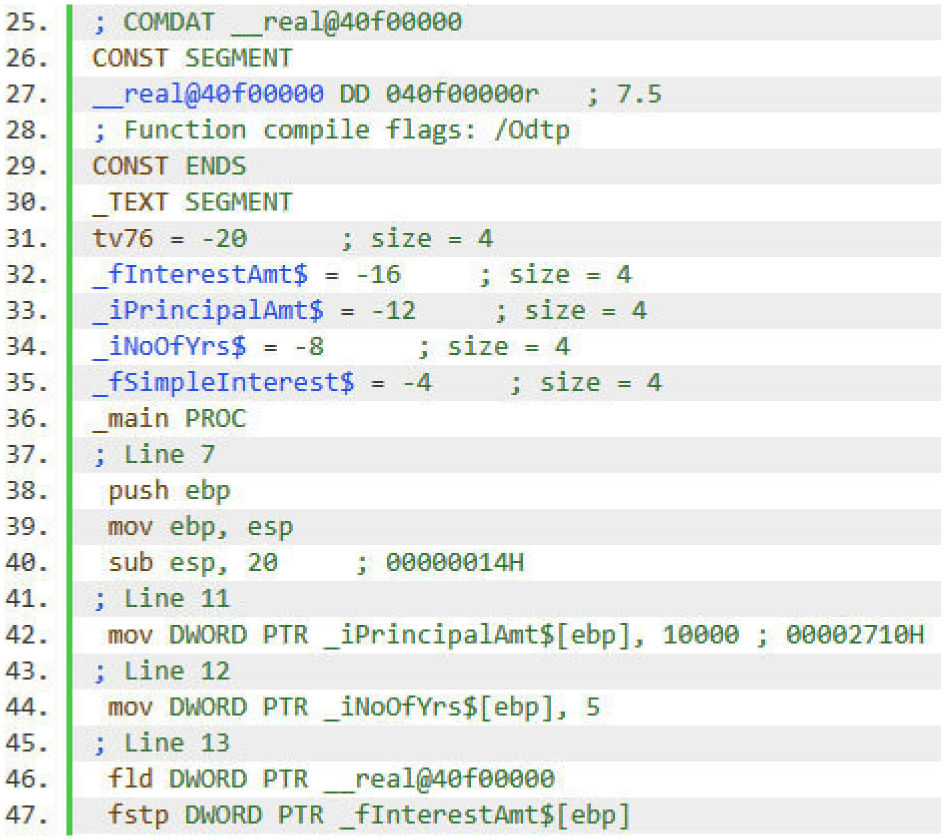
Figure 15.2: Simple Interest without Optimization

The generated assembly code will be as follows:



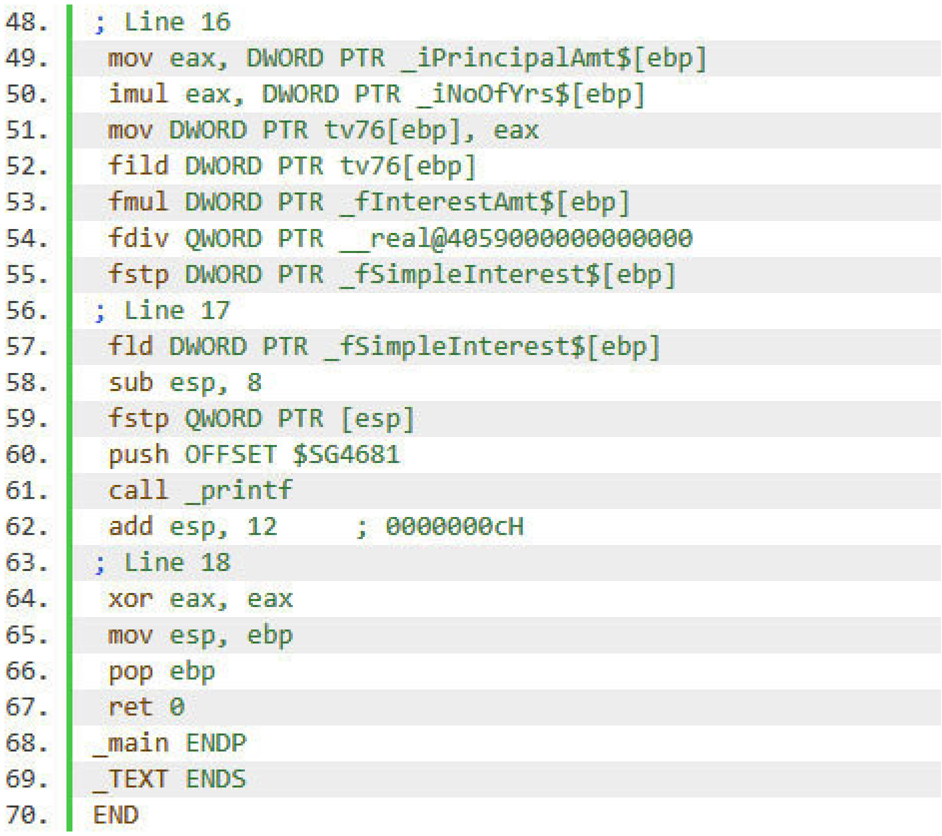
```
; Listing generated by Microsoft (R) Optimizing Compile

 TITLE C:\JitenderN\REBook\SimpleInterestCalculation\Si
 .686P
 .XMM
 include listing.inc
 .model flat

INCLUDELIB LIBCMT
INCLUDELIB OLDNAMES

CONST SEGMENT
$SG4681 DB '%f', 00H
CONST ENDS
PUBLIC __real@4059000000000000
PUBLIC __real@40f00000
PUBLIC _main
EXTRN _printf:PROC
EXTRN __fltused:DWORD
; COMDAT __real@4059000000000000
; File c:\jitendern\rebook\simpleinterestcalculation\si
CONST SEGMENT
__real@4059000000000000 DQ 04059000000000000r ; 100
CONST ENDS
```

Figure 15.3: SimpleInterestCalculation.asm-Part-1



```
; COMDAT __real@40f00000
CONST SEGMENT
__real@40f00000 DD 040f00000r   ; 7.5
; Function compile flags: /Odtp
CONST ENDS
_TEXT SEGMENT
tv76 = -20      ; size = 4
_fInterestAmt$ = -16      ; size = 4
_iPrincipalAmt$ = -12      ; size = 4
_iNoOfYrs$ = -8      ; size = 4
_fSimpleInterest$ = -4      ; size = 4
_main PROC
; Line 7
 push ebp
 mov ebp, esp
 sub esp, 20     ; 00000014H
; Line 11
 mov DWORD PTR _iPrincipalAmt$[ebp], 10000 ; 00002710H
; Line 12
 mov DWORD PTR _iNoOfYrs$[ebp], 5
; Line 13
 fld DWORD PTR __real@40f00000
 fstp DWORD PTR _fInterestAmt$[ebp]
```

Figure 15.4: *SimpleInterestCalculation.asm-Part-2*



```
48. ; Line 16
49.  mov eax, DWORD PTR _iPrincipalAmt$[ebp]
50.  imul eax, DWORD PTR _iNoOfYrs$[ebp]
51.  mov DWORD PTR tv76[ebp], eax
52.  fild DWORD PTR tv76[ebp]
53.  fmul DWORD PTR _fInterestAmt$[ebp]
54.  fdiv QWORD PTR __real@4059000000000000
55.  fstp DWORD PTR _fSimpleInterest$[ebp]
56. ; Line 17
57.  fld DWORD PTR _fSimpleInterest$[ebp]
58.  sub esp, 8
59.  fstp QWORD PTR [esp]
60.  push OFFSET $SG4681
61.  call _printf
62.  add esp, 12      ; 0000000cH
63. ; Line 18
64.  xor eax, eax
65.  mov esp, ebp
66.  pop ebp
67.  ret 0
68. _main ENDP
69. _TEXT ENDS
70. END
```

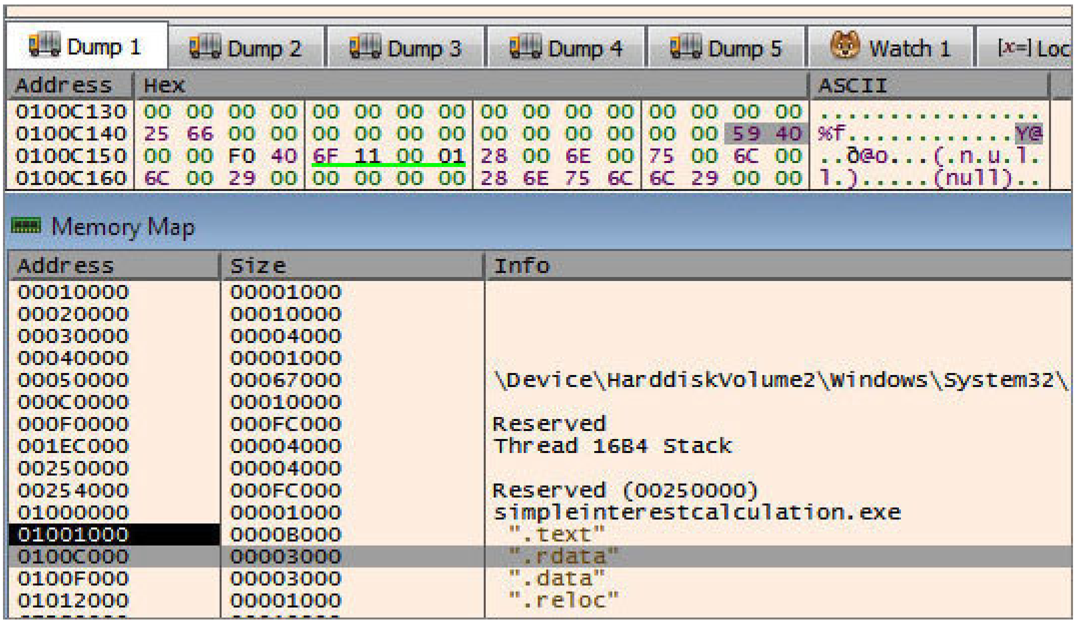
Figure 15.5: *SimpleInterestCalculation.asm-Part-3*

We have already discussed most of the parts in the listing in the preceding chapters. Let's move on to line 12.

▼Line 12-14

```
CONST SEGMENT
$SG4681 DB '%f', 00H
CONST ENDS
```

The string constant is allocated in the constant segment. In our case, the linker is renamed from **CONST SEGMENT** to **.rdata** (the code is placed in the **.code** segment, the constant strings are placed in the **CONST (.rdata)** segment, and if not a constant, it is placed in the **.data** segment), which can be dumped using any debugger. Following screenshot demonstrates string **$SG4681** in the memory dump:



```
Address   Hex                                                 ASCII
0100C130  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0100C140  25 66 00 00 00 00 00 00 00 00 00 00 00 00 59 40  %f............Y@
0100C150  00 00 F0 40 6F 11 00 01 28 00 6E 00 75 00 6C 00  ..ð@o...(.n.u.l.
0100C160  6C 00 29 00 00 00 00 00 28 6E 75 6C 6C 29 00 00  l.).....(null)..
```

```
Address    Size       Info
00010000   00001000
00020000   00010000
00030000   00004000
00040000   00001000
00050000   00067000   \Device\HarddiskVolume2\Windows\System32\
000C0000   00010000
000F0000   000FC000   Reserved
001EC000   00004000   Thread 16B4 Stack
00250000   00004000
00254000   000FC000   Reserved (00250000)
01000000   00001000   simpleinterestcalculation.exe
01001000   0000B000    ".text"
0100C000   00003000    ".rdata"
0100F000   00003000    ".data"
01012000   00001000    ".reloc"
```

Figure 15.6: .rdata

▼Line 15-21

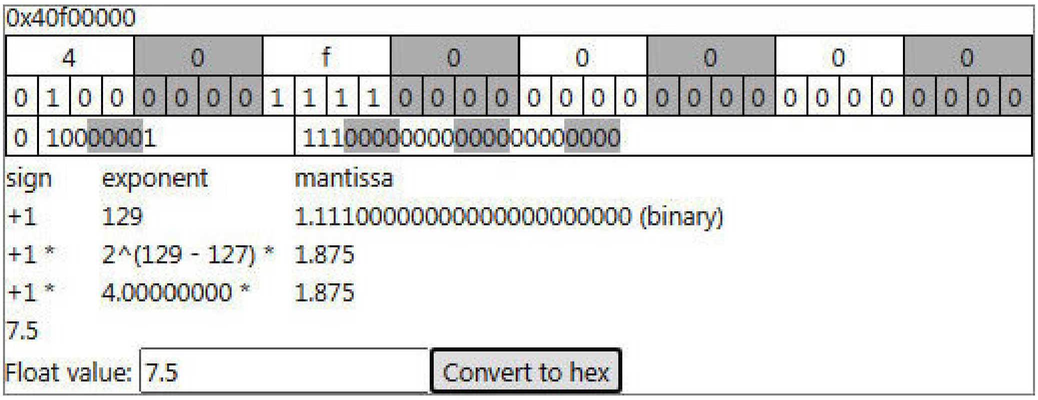
```
PUBLIC __real@4059000000000000
PUBLIC __real@40f00000
PUBLIC _main
EXTRN _printf:PROC
EXTRN __fltused:DWORD
; COMDAT __real@4059000000000000
; File c:\jitendern\rebook\simpleinterestcalculation\simpleinterestcalculation\simpleinterestcalculation.cpp
```

The public derivative is to make the real variable public to make it available across modules. The **EXTRN** derivative declares the extern function which is **printf** and The rest are all comments.

▼Line 22-29

```
CONST SEGMENT
__real@4059000000000000 DQ 04059000000000000r ; 100

CONST ENDS
; COMDAT __real@40f00000
CONST SEGMENT
__real@40f00000 DD 040f00000r    ; 7.5
; Function compile flags: /Odtp
CONST ENDS
```

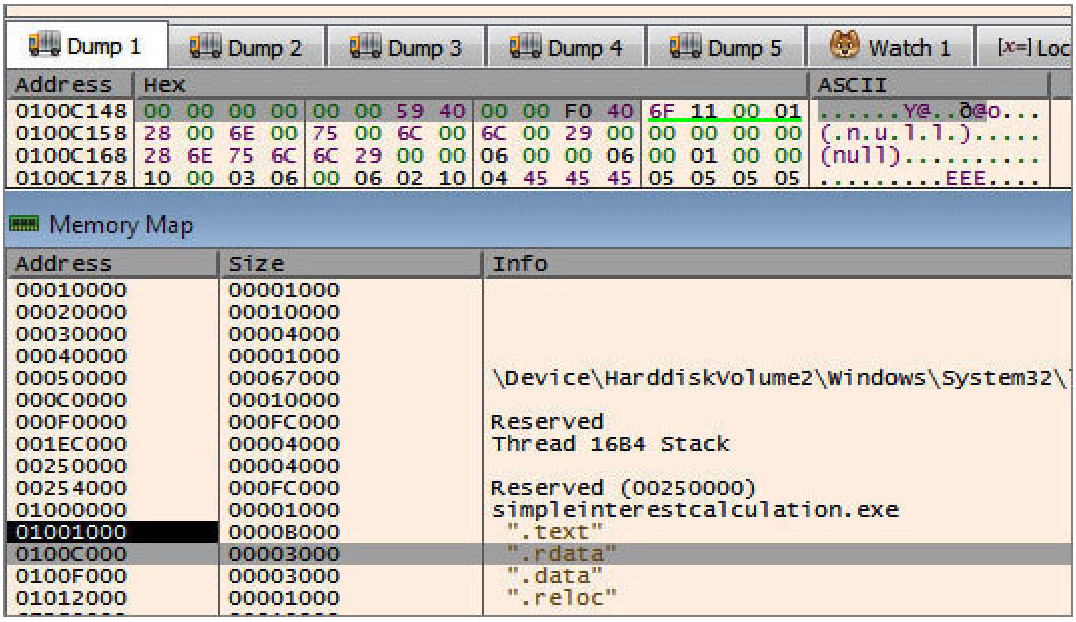
Here, the real variables/numbers are allocated in the **.rdata** segment. Both can be viewed by dumping the **.rdata** segment:



```
0x40f00000
4 0 f 0 0 0 0 0
0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 10000001 11100000000000000000000
sign   exponent           mantissa
+1     129                1.11100000000000000000000 (binary)
+1 *   2^(129 - 127) *    1.875
+1 *   4.00000000 *       1.875
7.5
```

Figure 15.7: *Float 7.5 to hex*

The floating point argument's hexadecimal representation will be stored in a reverse order, as we are dealing with the little-endian.



```
Address   Hex                                              ASCII
0100C148  00 00 00 00 00 00 59 40 00 00 F0 40 6F 11 00 01  ......Y@..ð@o...
0100C158  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
0100C168  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
0100C178  10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
```

```
Address    Size       Info
00010000   00001000
00020000   00010000
00030000   00004000
00040000   00001000
00050000   00067000   \Device\HarddiskVolume2\Windows\System32\
000C0000   00010000
000F0000   000FC000   Reserved
001EC000   00004000   Thread 16B4 Stack
00250000   00004000
00254000   000FC000   Reserved (00250000)
01000000   00001000   simpleinterestcalculation.exe
01001000   0000B000    ".text"
0100C000   00003000    ".rdata"
0100F000   00003000    ".data"
01012000   00001000    ".reloc"
```

Figure 15.8: .rdata with float hex

▼Line 30-35

```
_TEXT SEGMENT
tv76 = -20              ; size = 4
_fInterestAmt$ = -16         ; size = 4
_iPrincipalAmt$ = -12         ; size = 4
_iNoOfYrs$ = -8           ; size = 4
_fSimpleInterest$ = -4        ; size = 4
```

From here, our text segment starts. To access the local variable on the stack frame, we have to add **_$** to the **EBP** address. The following figure will help us understand the concept behind accessing arguments on the stack:

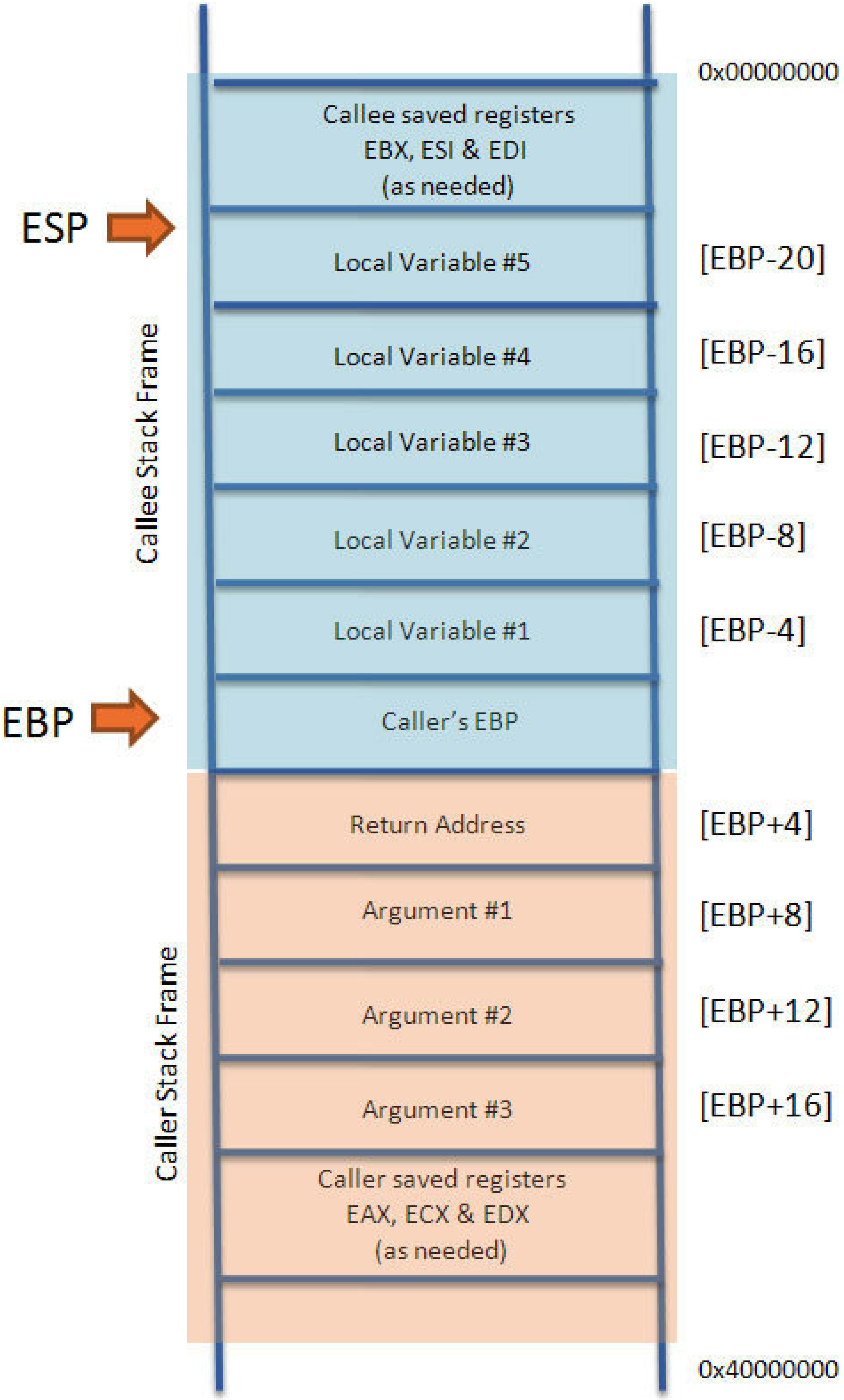


Figure 15.9: *Stack view*

To access the local variable **fInterestAmt** on the stack frame, we have to add **_fInterestAmt$** to the **EBP** address. So, to access the **fInterestAmt** variable on the stack, we have to add -16 to the **EBP** address. The same applies to other variables, each of 4 bytes in size. The same preceding stack can be visualized as follows:

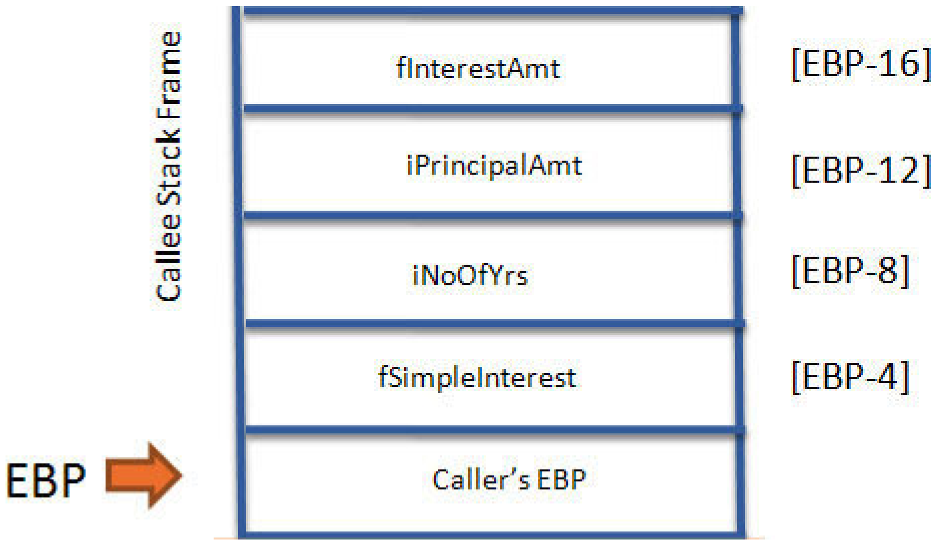
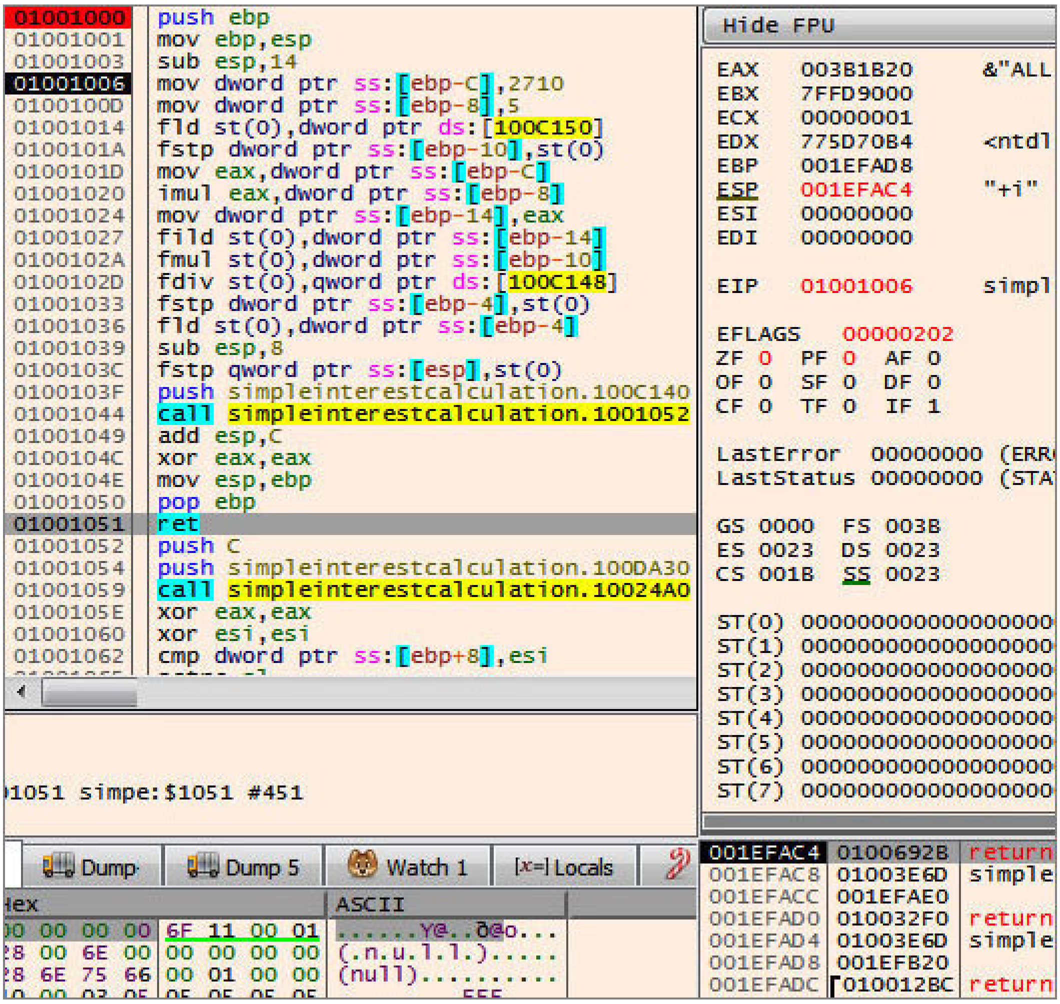


Figure 15.10: *Variables on stack*

▼Line 36-40

```
_main PROC
; Line 7
push ebp
mov ebp, esp
sub esp, 20        ; 00000014H
```

The **main** procedure starts here and the real code begins with the function prologue. The **SUB** instruction is creating room for the local variables by subtracting 20 bytes (14H in Hex), equivalent to the space for 5 local variables as shown in the following screenshot.



```
01001000  push ebp
01001001  mov ebp,esp
01001003  sub esp,14
01001006  mov dword ptr ss:[ebp-C],2710
0100100D  mov dword ptr ss:[ebp-8],5
01001014  fld st(0),dword ptr ds:[100C150]
0100101A  fstp dword ptr ss:[ebp-10],st(0)
0100101D  mov eax,dword ptr ss:[ebp-C]
01001020  imul eax,dword ptr ss:[ebp-8]
01001024  mov dword ptr ss:[ebp-14],eax
01001027  fild st(0),dword ptr ss:[ebp-14]
0100102A  fmul st(0),dword ptr ss:[ebp-10]
0100102D  fdiv st(0),qword ptr ds:[100C148]
01001033  fstp dword ptr ss:[ebp-4],st(0)
01001036  fld st(0),dword ptr ss:[ebp-4]
01001039  sub esp,8
0100103C  fstp qword ptr ss:[esp],st(0)
0100103F  push simpleinterestcalculation.100C140
01001044  call simpleinterestcalculation.1001052
01001049  add esp,C
0100104C  xor eax,eax
0100104E  mov esp,ebp
01001050  pop ebp
01001051  ret
01001052  push C
01001054  push simpleinterestcalculation.100DA30
01001059  call simpleinterestcalculation.10024A0
0100105E  xor eax,eax
01001060  xor esi,esi
01001062  cmp dword ptr ss:[ebp+8],esi
```

```
1051 simpe:$1051 #451
```

```
EAX   003B1B20     &"ALL
EBX   7FFD9000
ECX   00000001
EDX   775D70B4     <ntdl
EBP   001EFAD8
ESP   001EFAC4     "+i"
ESI   00000000
EDI   00000000

EIP   01001006     simpl

EFLAGS    00000202
ZF 0   PF 0   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERR
LastStatus  00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
ST(7) 0000000000000000000
```

```
ex                          ASCII
0 00 00 00 6F 11 00 01      ......Y@..ð@o...
8 00 6E 00 00 00 00 00      (.n.u.l.l.).....
8 6E 75 66 00 01 00 00      (null)..........
```

```
001EFAC4  0100692B  return
001EFAC8  01003E6D  simple
001EFACC  001EFAE0
001EFAD0  010032F0  return
001EFAD4  01003E6D  simple
001EFAD8  001EFB20
001EFADC  010012BC  return
```

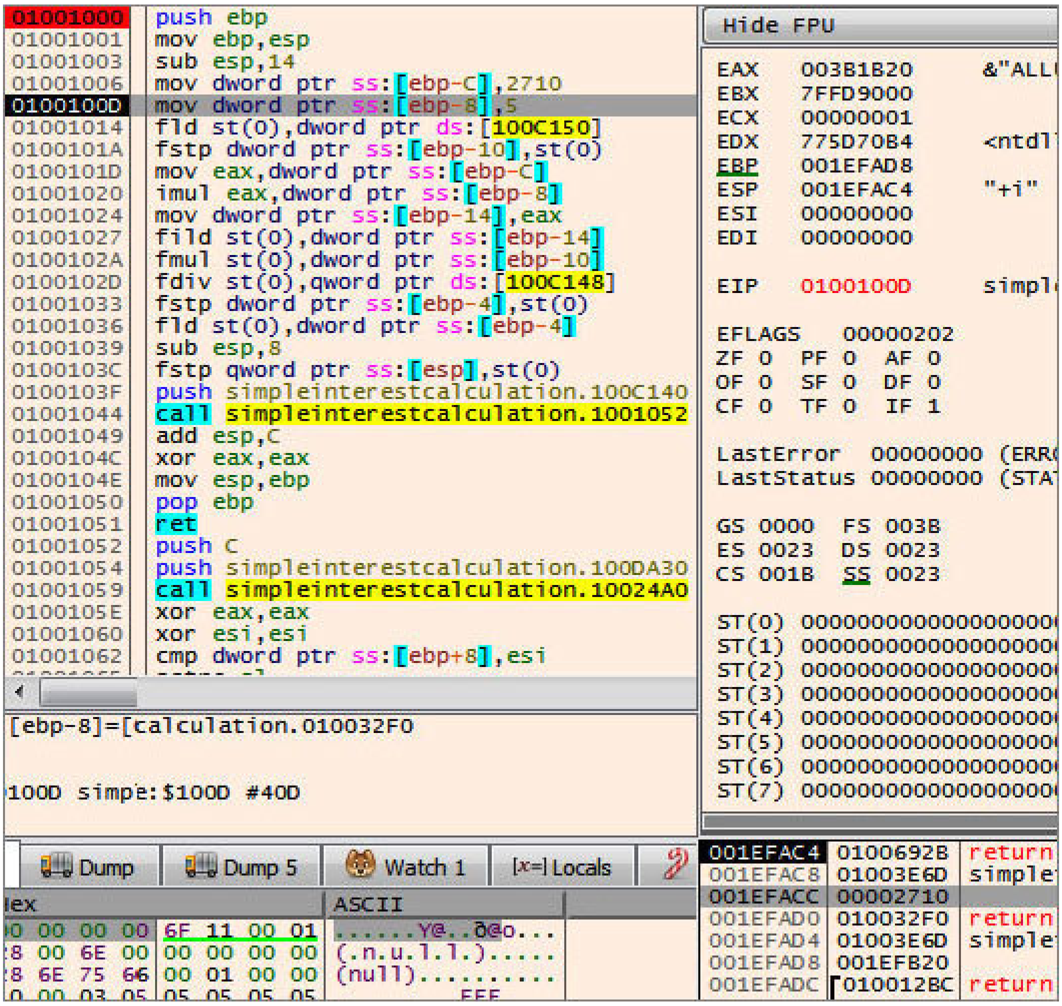
Figure 15.11: *Creating room for variables on the stack*

As we have only 4 local variables, you may be wondering why a space for 5 local variables is required. To understand this point, we will walk through the code instruction-by-instruction by putting a breakpoint point on the **main** procedure call. We will use x32dbg to step into the next instruction:

▼Line 41-42

```
; Line 11
mov DWORD PTR _iPrincipalAmt$[ebp], 10000 ; 00002710H
```

This instruction will move the first variable, that is on the stack at **[EBP-12]** as shown in the following screenshot.



```
01001000  push ebp
01001001  mov ebp,esp
01001003  sub esp,14
01001006  mov dword ptr ss:[ebp-C],2710
0100100D  mov dword ptr ss:[ebp-8],5
01001014  fld st(0),dword ptr ds:[100C150]
0100101A  fstp dword ptr ss:[ebp-10],st(0)
0100101D  mov eax,dword ptr ss:[ebp-C]
01001020  imul eax,dword ptr ss:[ebp-8]
01001024  mov dword ptr ss:[ebp-14],eax
01001027  fild st(0),dword ptr ss:[ebp-14]
0100102A  fmul st(0),dword ptr ss:[ebp-10]
0100102D  fdiv st(0),qword ptr ds:[100C148]
01001033  fstp dword ptr ss:[ebp-4],st(0)
01001036  fld st(0),dword ptr ss:[ebp-4]
01001039  sub esp,8
0100103C  fstp qword ptr ss:[esp],st(0)
0100103F  push simpleinterestcalculation.100C140
01001044  call simpleinterestcalculation.1001052
01001049  add esp,C
0100104C  xor eax,eax
0100104E  mov esp,ebp
01001050  pop ebp
01001051  ret
01001052  push C
01001054  push simpleinterestcalculation.100DA30
01001059  call simpleinterestcalculation.10024A0
0100105E  xor eax,eax
01001060  xor esi,esi
01001062  cmp dword ptr ss:[ebp+8],esi
```

```
[ebp-8]=[calculation.010032F0

100D simpe:$100D #40D
```

```
EAX   003B1B20     &"ALL
EBX   7FFD9000
ECX   00000001
EDX   775D70B4     <ntdl
EBP   001EFAD8
ESP   001EFAC4     "+i"
ESI   00000000
EDI   00000000

EIP   0100100D     simpl

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
ST(7) 0000000000000000000
```

```
lex                           ASCII
0 00 00 00  6F 11 00 01  ......Y@..ð@o...
8 00 6E 00  00 00 00 00  (.n.u.l.l.).....
8 6E 75 66  00 01 00 00  (null)..........
```

```
001EFAC4  0100692B  return
001EFAC8  01003E6D  simple
001EFACC  00002710
001EFAD0  010032F0  return
001EFAD4  01003E6D  simple
001EFAD8  001EFB20
001EFADC  010012BC  return
```

Figure 15.12: *iPrincipalAmt on the stack*

▼Line 43-44

```
; Line 12
mov DWORD PTR _iNoOfYrs$[ebp], 5
```

This instruction will move the second variable, that is on the stack at **[EBP-8].**

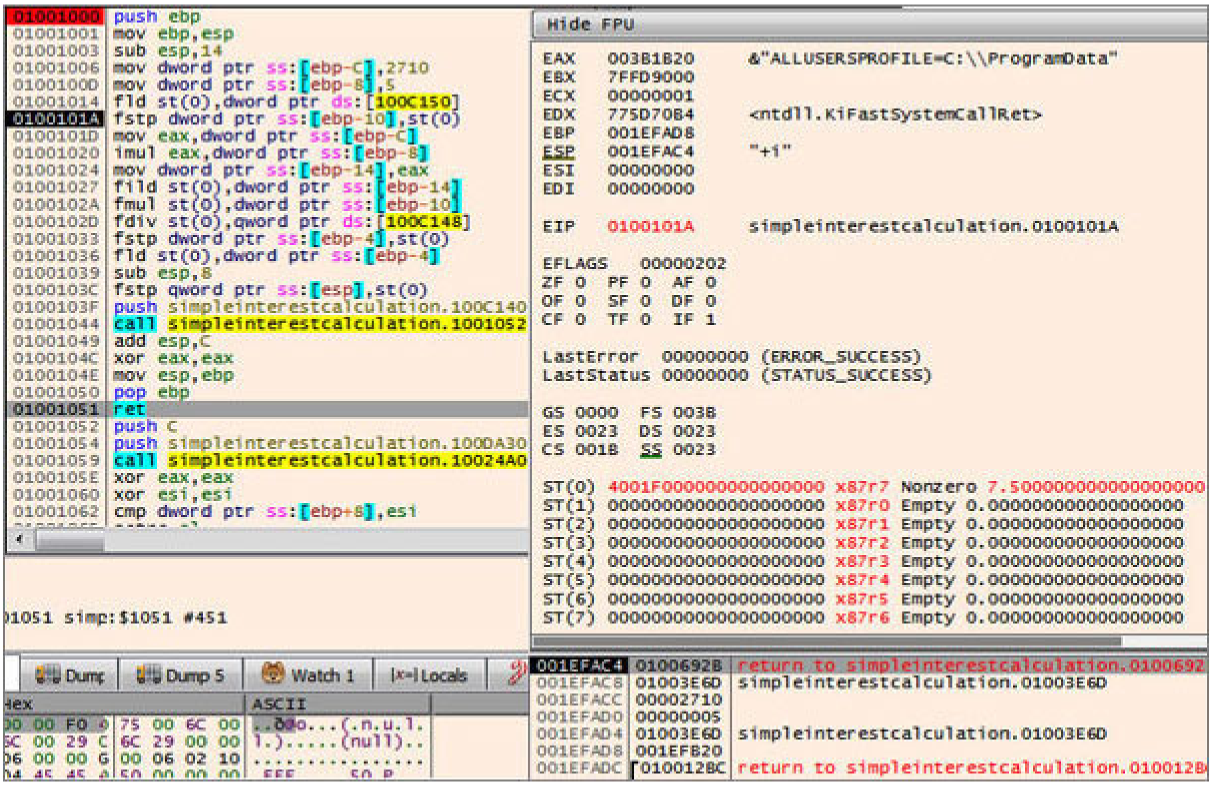
▼Line 45-46

```
; Line 13
fld DWORD PTR __real@40f00000.
```

The **Floating Point Load** will push the floating point value on the FPU stack. In x32dbg, we can see the same instruction as follows:

```
fld st0, dword ptr ds:[0x0100C150]
```

This means pushing the floating point value stored at **ds:[0x0100C150]** to the **ST(0)** register as shown below.



```
01001000 push ebp
01001001 mov ebp,esp
01001003 sub esp,14
01001006 mov dword ptr ss:[ebp-C],2710
0100100D mov dword ptr ss:[ebp-8],5
01001014 fld st(0),dword ptr ds:[100C150]
0100101A fstp dword ptr ss:[ebp-10],st(0)
0100101D mov eax,dword ptr ss:[ebp-C]
01001020 imul eax,dword ptr ss:[ebp-8]
01001024 mov dword ptr ss:[ebp-14],eax
01001027 fild st(0),dword ptr ss:[ebp-14]
0100102A fmul st(0),dword ptr ss:[ebp-10]
0100102D fdiv st(0),qword ptr ds:[100C148]
01001033 fstp dword ptr ss:[ebp-4],st(0)
01001036 fld st(0),dword ptr ss:[ebp-4]
01001039 sub esp,8
0100103C fstp qword ptr ss:[esp],st(0)
0100103F push simpleinterestcalculation.100C140
01001044 call simpleinterestcalculation.1001052
01001049 add esp,C
0100104C xor eax,eax
0100104E mov esp,ebp
01001050 pop ebp
01001051 ret
01001052 push C
01001054 push simpleinterestcalculation.100DA30
01001059 call simpleinterestcalculation.10024A0
0100105E xor eax,eax
01001060 xor esi,esi
01001062 cmp dword ptr ss:[ebp+8],esi

)1051 simp:$1051 #451
```

```
EAX   003B1B20     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX   7FFD9000
ECX   00000001
EDX   775D70B4     <ntdll.KiFastSystemCallRet>
EBP   001EFAD8
ESP   001EFAC4     "+i"
ESI   00000000
EDI   00000000

EIP   0100101A     simpleinterestcalculation.0100101A

EFLAGS    00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 4001F000000000000000 x87r7 Nonzero 7.500000000000000000
ST(1) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r6 Empty 0.000000000000000000
```

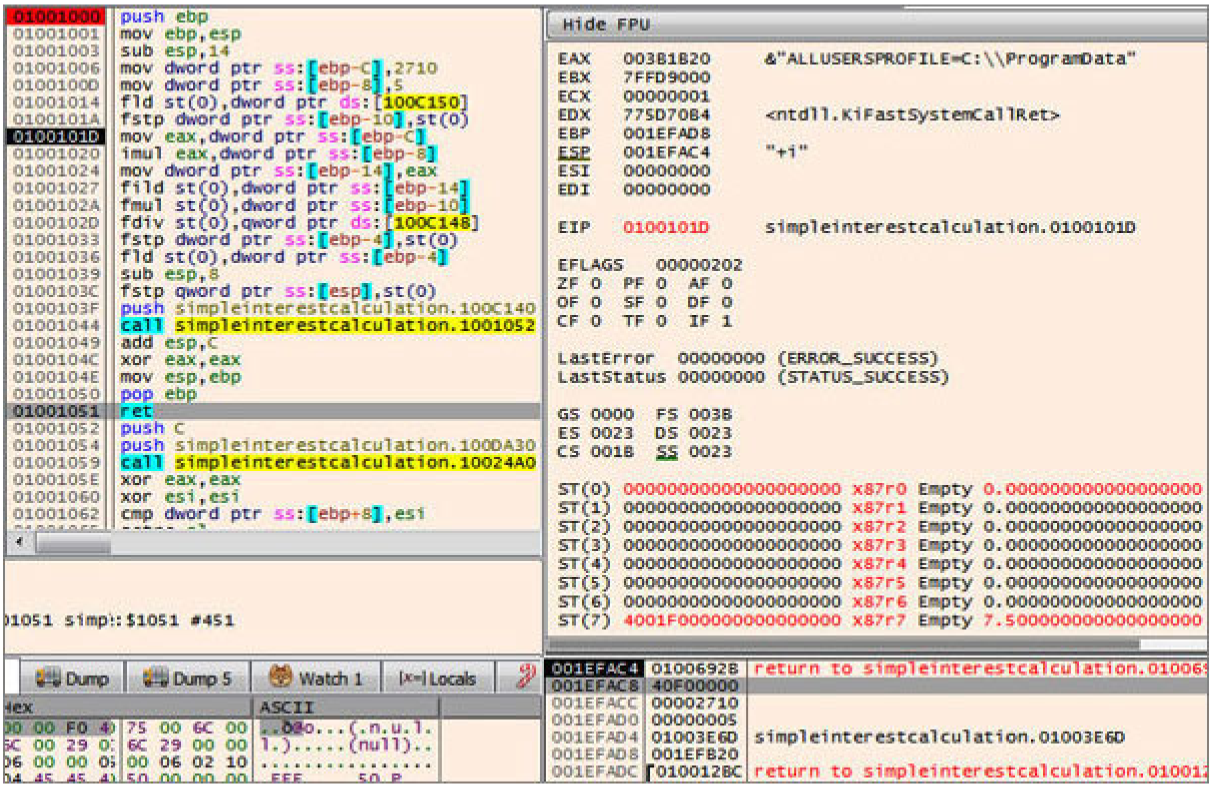
```
001EFAC4 0100692B return to simpleinterestcalculation.0100692
001EFAC8 01003E6D simpleinterestcalculation.01003E6D
001EFACC 00002710
001EFAD0 00000005
001EFAD4 01003E6D simpleinterestcalculation.01003E6D
001EFAD8 001EFB20
001EFADC 0100128C return to simpleinterestcalculation.010012B
```

Figure 15.13: *Floating point value on the FPU stack*

▼Line 47

fstp DWORD PTR _flnterestAmt$[ebp]

This will move the **flnterestAmt** floating point value stored at **ST(0)** to the stack **[EBP-0x10]** and POPs the variable from the FPU stack. The same instruction is viewed as follows in x32dbg: fstp dword ptr ss:[ebp-0x10], st0



```
01001000  push ebp
01001001  mov ebp,esp
01001003  sub esp,14
01001006  mov dword ptr ss:[ebp-C],2710
0100100D  mov dword ptr ss:[ebp-8],5
01001014  fld st(0),dword ptr ds:[100C150]
0100101A  fstp dword ptr ss:[ebp-10],st(0)
0100101D  mov eax,dword ptr ss:[ebp-C]
01001020  imul eax,dword ptr ss:[ebp-8]
01001024  mov dword ptr ss:[ebp-14],eax
01001027  fild st(0),dword ptr ss:[ebp-14]
0100102A  fmul st(0),dword ptr ss:[ebp-10]
0100102D  fdiv st(0),qword ptr ds:[100C148]
01001033  fstp dword ptr ss:[ebp-4],st(0)
01001036  fld st(0),dword ptr ss:[ebp-4]
01001039  sub esp,8
0100103C  fstp qword ptr ss:[esp],st(0)
0100103F  push simpleinterestcalculation.100C140
01001044  call simpleinterestcalculation.1001052
01001049  add esp,C
0100104C  xor eax,eax
0100104E  mov esp,ebp
01001050  pop ebp
01001051  ret
01001052  push C
01001054  push simpleinterestcalculation.100DA30
01001059  call simpleinterestcalculation.10024A0
0100105E  xor eax,eax
01001060  xor esi,esi
01001062  cmp dword ptr ss:[ebp+8],esi

)1051 simp::$1051 #451
```

```
EAX    003B1B20     &"ALLUSERSPROFILE=C:\\ProgramData"
EBX    7FFD9000
ECX    00000001
EDX    775D70B4     <ntdll.KiFastSystemCallRet>
EBP    001EFAD8
ESP    001EFAC4     "+i"
ESI    00000000
EDI    00000000

EIP    0100101D     simpleinterestcalculation.0100101D

EFLAGS   00000202
ZF 0  PF 0  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 4001F000000000000000 x87r7 Empty 7.500000000000000000
```

```
001EFAC4  01006928  return to simpleinterestcalculation.01006
001EFAC8  40F00000
001EFACC  00002710
001EFAD0  00000005
001EFAD4  01003E6D  simpleinterestcalculation.01003E6D
001EFAD8  001EFB20
001EFADC  010012BC  return to simpleinterestcalculation.01001
```

Figure 15.14: fInterestAmt on stack

▼Line 49

mov eax, DWORD PTR _iPrincipalAmt$[ebp]

It will move the **iPrincipalAmt** variable value to **EAX** for further calculation.

▼Line 50

imul eax, DWORD PTR _iNoOfYrs$[ebp]

This multiplies the **iNoOfYrs** variable stored on the stack **[EBP-8]** with the **iPrincipalAmt** variable stored in The result of the multiplication is stored in the **EAX** register.

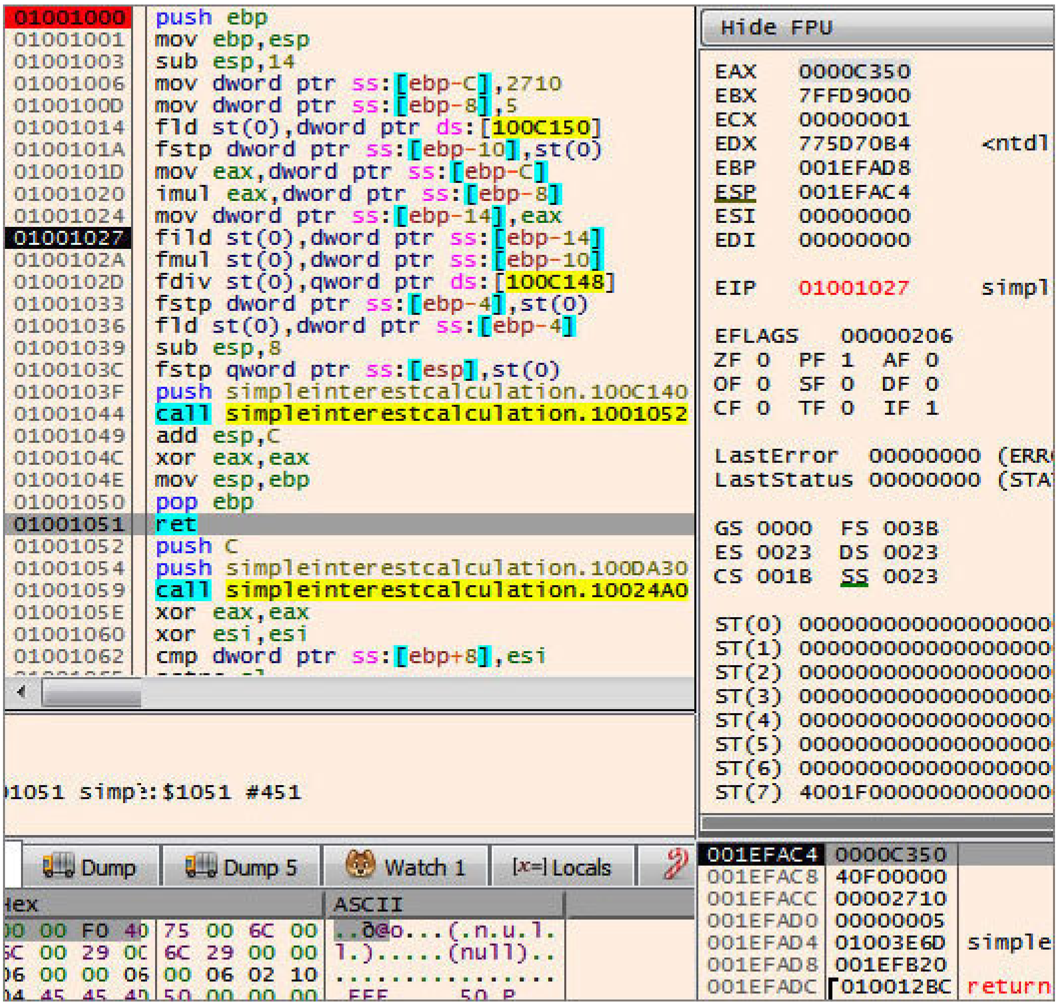
▼Line 51

```
mov DWORD PTR tv76[ebp], eax
```

The same can be viewed as follows in x32dbg:

```
mov dword ptr ss:[ebp-0x14], eax
```

MOV will move the multiplication value of **iPrincipalAmt** and **iNoOfYrs** onto the stack at **[EBP-0x14]** as a temporary location. We can imagine this multiplication result as a variable local to the **main** procedure. This is where we our variable comes in picture as shown below in the screenshot.



```
01001000  push ebp
01001001  mov ebp,esp
01001003  sub esp,14
01001006  mov dword ptr ss:[ebp-C],2710
0100100D  mov dword ptr ss:[ebp-8],5
01001014  fld st(0),dword ptr ds:[100C150]
0100101A  fstp dword ptr ss:[ebp-10],st(0)
0100101D  mov eax,dword ptr ss:[ebp-C]
01001020  imul eax,dword ptr ss:[ebp-8]
01001024  mov dword ptr ss:[ebp-14],eax
01001027  fild st(0),dword ptr ss:[ebp-14]
0100102A  fmul st(0),dword ptr ss:[ebp-10]
0100102D  fdiv st(0),qword ptr ds:[100C148]
01001033  fstp dword ptr ss:[ebp-4],st(0)
01001036  fld st(0),dword ptr ss:[ebp-4]
01001039  sub esp,8
0100103C  fstp qword ptr ss:[esp],st(0)
0100103F  push simpleinterestcalculation.100C140
01001044  call simpleinterestcalculation.1001052
01001049  add esp,C
0100104C  xor eax,eax
0100104E  mov esp,ebp
01001050  pop ebp
01001051  ret
01001052  push C
01001054  push simpleinterestcalculation.100DA30
01001059  call simpleinterestcalculation.10024A0
0100105E  xor eax,eax
01001060  xor esi,esi
01001062  cmp dword ptr ss:[ebp+8],esi
```

```
1051 simp:$1051 #451
```

```
EAX     0000C350
EBX     7FFD9000
ECX     00000001
EDX     775D70B4      <ntdl
EBP     001EFAD8
ESP     001EFAC4
ESI     00000000
EDI     00000000

EIP     01001027      simpl

EFLAGS     00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERR
LastStatus 00000000 (STA

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 0000000000000000000
ST(1) 0000000000000000000
ST(2) 0000000000000000000
ST(3) 0000000000000000000
ST(4) 0000000000000000000
ST(5) 0000000000000000000
ST(6) 0000000000000000000
ST(7) 4001F000000000000000
```

```
001EFAC4  0000C350
001EFAC8  40F00000
001EFACC  00002710
001EFAD0  00000005
001EFAD4  01003E6D  simple
001EFAD8  001EFB20
001EFADC  010012BC  return
```

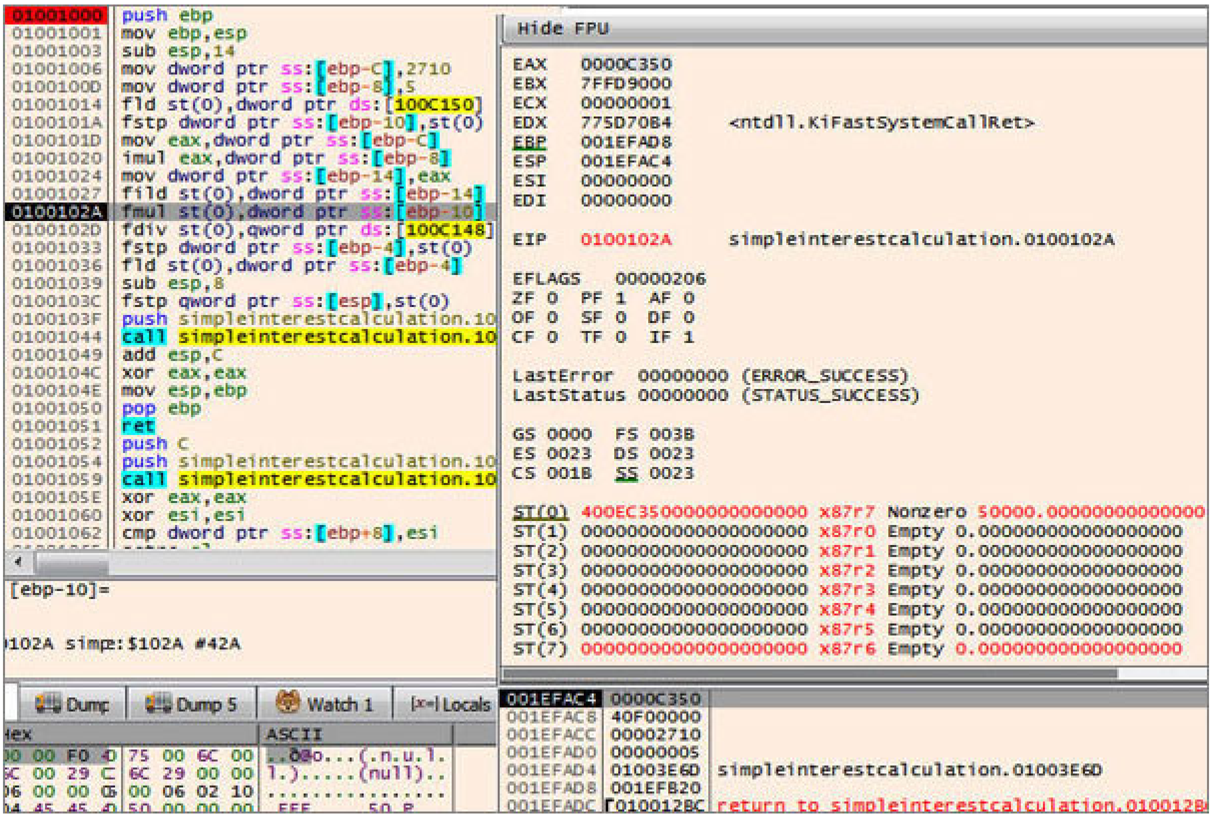
```
Hex                                  ASCII
00 00 F0 40  75 00 6C 00   ..ð@o...(.n.u.l.
6C 00 29 0C  6C 29 00 00   l.).....(null)..
06 00 00 06  00 06 02 10   ................
```

Figure 15.15: Multiplication value on the stack

▼Line 52

fild DWORD PTR tv76[ebp] ;In x32dbg, fild st0, dword ptr ss:[ebp-0x14]

The multiplication result of **iPrincipalAmt** and **iNoOfYrs** stored at **[EBP-0x14]** is signed-integer. If it has to be moved to the FPU stack, it has to be converted to the floating point format. The **FILD** instruction converts the signed-integer into a floating point format and then pushes the value onto the FPU register stack.



```
01001000  push ebp
01001001  mov ebp,esp
01001003  sub esp,14
01001006  mov dword ptr ss:[ebp-C],2710
0100100D  mov dword ptr ss:[ebp-8],5
01001014  fld st(0),dword ptr ds:[100C150]
0100101A  fstp dword ptr ss:[ebp-10],st(0)
0100101D  mov eax,dword ptr ss:[ebp-C]
01001020  imul eax,dword ptr ss:[ebp-8]
01001024  mov dword ptr ss:[ebp-14],eax
01001027  fild st(0),dword ptr ss:[ebp-14]
0100102A  fmul st(0),dword ptr ss:[ebp-10]
0100102D  fdiv st(0),qword ptr ds:[100C148]
01001033  fstp dword ptr ss:[ebp-4],st(0)
01001036  fld st(0),dword ptr ss:[ebp-4]
01001039  sub esp,8
0100103C  fstp qword ptr ss:[esp],st(0)
0100103F  push simpleinterestcalculation.10
01001044  call simpleinterestcalculation.10
01001049  add esp,C
0100104C  xor eax,eax
0100104E  mov esp,ebp
01001050  pop ebp
01001051  ret
01001052  push C
01001054  push simpleinterestcalculation.10
01001059  call simpleinterestcalculation.10
0100105E  xor eax,eax
01001060  xor esi,esi
01001062  cmp dword ptr ss:[ebp+8],esi

[ebp-10]=

102A simpe:$102A #42A
```

```
Hide FPU

EAX     0000C350
EBX     7FFD9000
ECX     00000001
EDX     775D70B4      <ntdll.KiFastSystemCallRet>
EBP     001EFAD8
ESP     001EFAC4
ESI     00000000
EDI     00000000

EIP     0100102A      simpleinterestcalculation.0100102A

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 400EC350000000000000 x87r7 Nonzero 50000.0000000000000
ST(1) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r6 Empty 0.000000000000000000
```

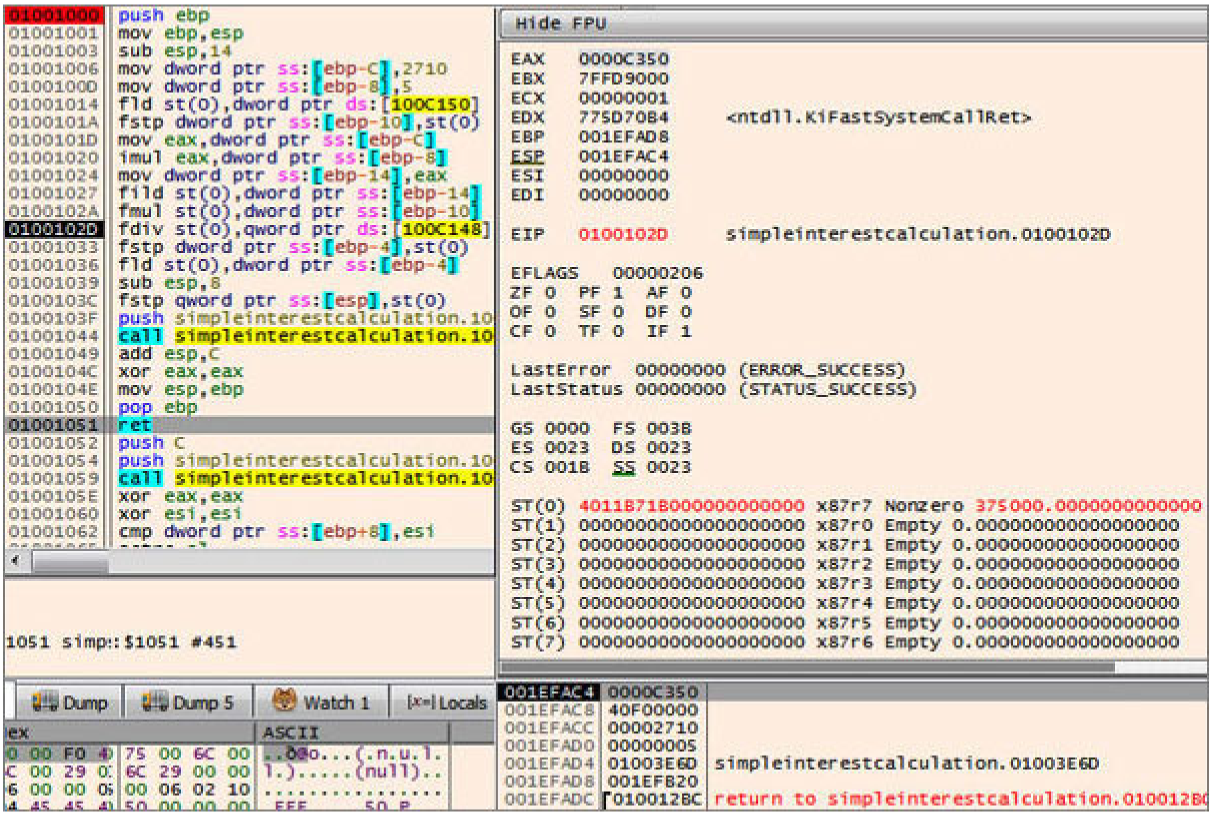
```
001EFAC4  0000C350
001EFAC8  40F00000
001EFACC  00002710
001EFAD0  00000005
001EFAD4  01003E6D  simpleinterestcalculation.01003E6D
001EFAD8  001EFB20
001EFADC  010012BC  return to simpleinterestcalculation.010012B
```

Figure 15.16: *FILD instruction output*

▼Line 53

fmul DWORD PTR _fInterestAmt$[ebp] ;in x32dbg, fmul sto, dword ptr ss:[ebp-0x10]

It will multiple the **fInterestAmt** variable stored on the stack with the value stored at **ST(0).** So, this basically multiples the multiplication result of **iPrincipalAmt** and **iNoOfYrs** (stored at with **fInterestAmt** (stored on the stack



```
01001000 push ebp
01001001 mov ebp,esp
01001003 sub esp,14
01001006 mov dword ptr ss:[ebp-C],2710
0100100D mov dword ptr ss:[ebp-8],5
01001014 fld st(0),dword ptr ds:[100C150]
0100101A fstp dword ptr ss:[ebp-10],st(0)
0100101D mov eax,dword ptr ss:[ebp-C]
01001020 imul eax,dword ptr ss:[ebp-8]
01001024 mov dword ptr ss:[ebp-14],eax
01001027 fild st(0),dword ptr ss:[ebp-14]
0100102A fmul st(0),dword ptr ss:[ebp-10]
0100102D fdiv st(0),qword ptr ds:[100C148]
01001033 fstp dword ptr ss:[ebp-4],st(0)
01001036 fld st(0),dword ptr ss:[ebp-4]
01001039 sub esp,8
0100103C fstp qword ptr ss:[esp],st(0)
0100103F push simpleinterestcalculation.10
01001044 call simpleinterestcalculation.10
01001049 add esp,C
0100104C xor eax,eax
0100104E mov esp,ebp
01001050 pop ebp
01001051 ret
01001052 push C
01001054 push simpleinterestcalculation.10
01001059 call simpleinterestcalculation.10
0100105E xor eax,eax
01001060 xor esi,esi
01001062 cmp dword ptr ss:[ebp+8],esi

1051 simp::$1051 #451
```

```
EAX    0000C350
EBX    7FFD9000
ECX    00000001
EDX    775D70B4      <ntdll.KiFastSystemCallRet>
EBP    001EFAD8
ESP    001EFAC4
ESI    00000000
EDI    00000000

EIP    0100102D      simpleinterestcalculation.0100102D

EFLAGS    00000206
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000  FS 003B
ES 0023  DS 0023
CS 001B  SS 0023

ST(0) 4011B71B000000000000 x87r7 Nonzero 375000.000000000000
ST(1) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(7) 00000000000000000000 x87r6 Empty 0.000000000000000000
```

```
001EFAC4 0000C350
001EFAC8 40F00000
001EFACC 00002710
001EFAD0 00000005
001EFAD4 01003E6D simpleinterestcalculation.01003E6D
001EFAD8 001EFB20
001EFADC 010012BC return to simpleinterestcalculation.010012BC
```

__Figure 15.17:__ FMUL instruction output

▼Line 54

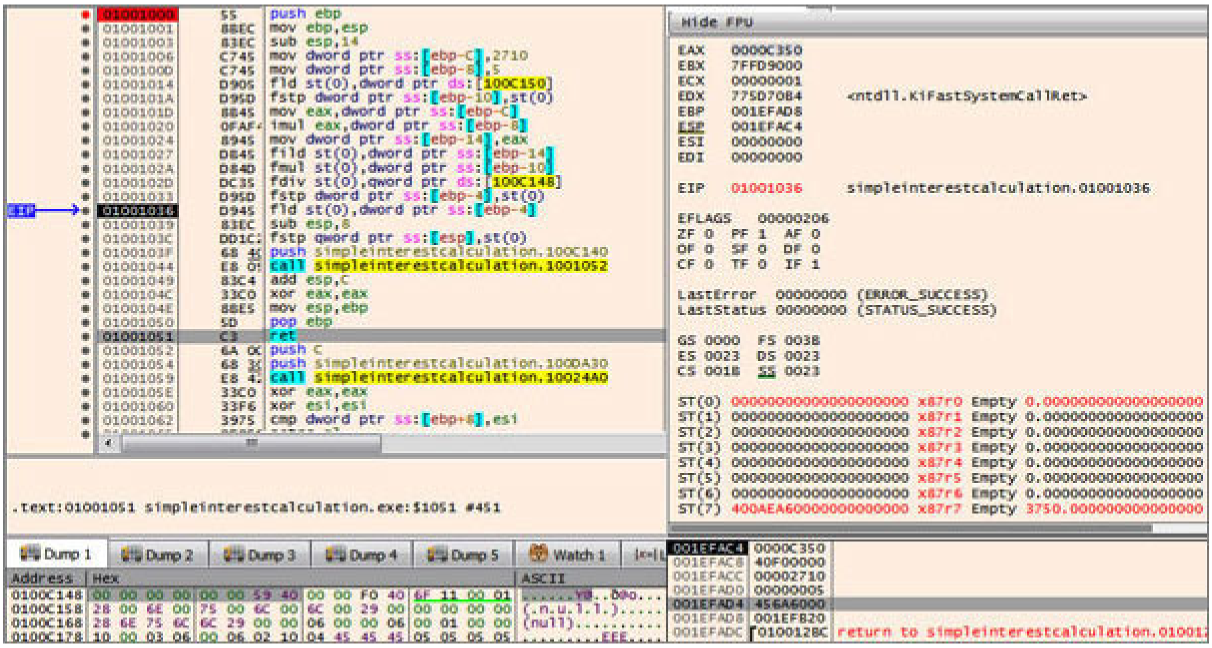
```
fdiv QWORD PTR __real@4059000000000000
; in x32dbg, fdiv st0, qword ptr ds:[0x0100C148]
```

Up until now, we have multiplied all three, **iPrincipalAmt** * **iNoOfYrs** * and the result of this is stored in the **FPU** stack at Now, this will divide the resultant with 100 as per our C/C++ code. **FDIV** will divide QWORD (1 QWORD = 8 bytes) stored at **ds:[0x0100C148]** with the value on the FPU stack.

▼Line 55

fstp DWORD PTR _fSimpleInterest$[ebp]
;In x32dbg, fstp dword ptr ss:[ebp-0x4], st0

This will move the **fSimpleInterest** that we got from **ST(0)** to the stack **[EBP-0x4]** and POP the variable from FPU stack as shown in the following screenshot.



```
01001000  55        push ebp
01001001  8BEC      mov ebp,esp
01001003  83EC      sub esp,14
01001006  C745      mov dword ptr ss:[ebp-C],2710
0100100D  C745      mov dword ptr ss:[ebp-8],5
01001014  D905      fld st(0),dword ptr ds:[100C150]
0100101A  D95D      fstp dword ptr ss:[ebp-10],st(0)
0100101D  8B45      mov eax,dword ptr ss:[ebp-C]
01001020  0FAF4     imul eax,dword ptr ss:[ebp-8]
01001024  8945      mov dword ptr ss:[ebp-14],eax
01001027  DB45      fild st(0),dword ptr ss:[ebp-14]
0100102A  D84D      fmul st(0),dword ptr ss:[ebp-10]
0100102D  DC35      fdiv st(0),qword ptr ds:[100C148]
01001033  D95D      fstp dword ptr ss:[ebp-4],st(0)
01001036  D945      fld st(0),dword ptr ss:[ebp-4]
01001039  83EC      sub esp,8
0100103C  DD1C2     fstp qword ptr ss:[esp],st(0)
0100103F  68 4      push simpleinterestcalculation.100C140
01001044  E8 0      call simpleinterestcalculation.1001052
01001049  83C4      add esp,C
0100104C  33C0      xor eax,eax
0100104E  8BE5      mov esp,ebp
01001050  5D        pop ebp
01001051  C3        ret
01001052  6A 0C     push C
01001054  68 3      push simpleinterestcalculation.100DA30
01001059  E8 4      call simpleinterestcalculation.10024A0
0100105E  33C0      xor eax,eax
01001060  33F6      xor esi,esi
01001062  3975      cmp dword ptr ss:[ebp+8],esi

.text:01001051 simpleinterestcalculation.exe:$1051 #451
```

```
EAX     0000C350
EBX     7FFD9000
ECX     00000001
EDX     775D70B4      <ntdll.KiFastSystemCallRet>
EBP     001EFAD8
ESP     001EFAC4
ESI     00000000
EDI     00000000

EIP     01001036      simpleinterestcalculation.01001036

EFLAGS    00000206
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError  00000000 (ERROR_SUCCESS)
LastStatus 00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 400AEA60000000000000 x87r7 Empty 3750.000000000000000
```

```
Address   Hex                                              ASCII
0100C148  00 00 00 00 00 00 59 40 00 00 F0 40 6F 11 00 01  ......Y@..ð@o...
0100C158  28 00 6E 00 75 00 6C 00 6C 00 29 00 00 00 00 00  (.n.u.l.l.).....
0100C168  28 6E 75 6C 6C 29 00 00 06 00 00 06 00 01 00 00  (null)..........
0100C178  10 00 03 06 00 06 02 10 04 45 45 45 05 05 05 05  .........EEE....
```

```
001EFAC4  0000C350
001EFAC8  40F00000
001EFACC  00002710
001EFAD0  00000005
001EFAD4  456A6000
001EFAD8  001EFB20
001EFADC  0100128C  return to simpleinterestcalculation.01001
```

Figure 15.18: FSTP instruction output

▼Line 57

fld DWORD PTR _fSimpleInterest$[ebp]
;In x32dbg, fld st0, dword ptr ss:[ebp-0x4]

This pushes the **fSimpleInterest** floating point value onto the FPU stack.

▼Line 58

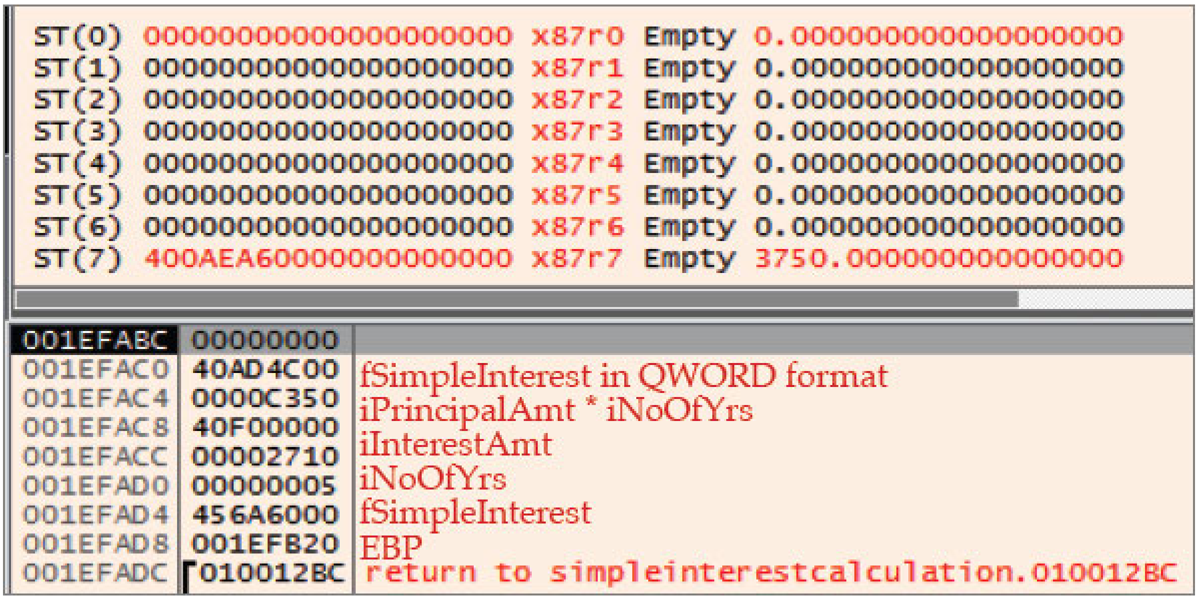
```
sub esp, 8
```

Before calling the **printf** function, it will create room for arguments on the stack by 8 bytes.

▼Line 59

```
fstp QWORD PTR [esp]
;In x32dbg, fstp qword ptr ss:[esp], sto
```

This will move **fSimpleInterest** from **ST(0)** to the stack **[ESP]** in the **QWORD** format and then POP the variable from FPU stack. In the following screenshot, variables are marked on stack for proper understanding.



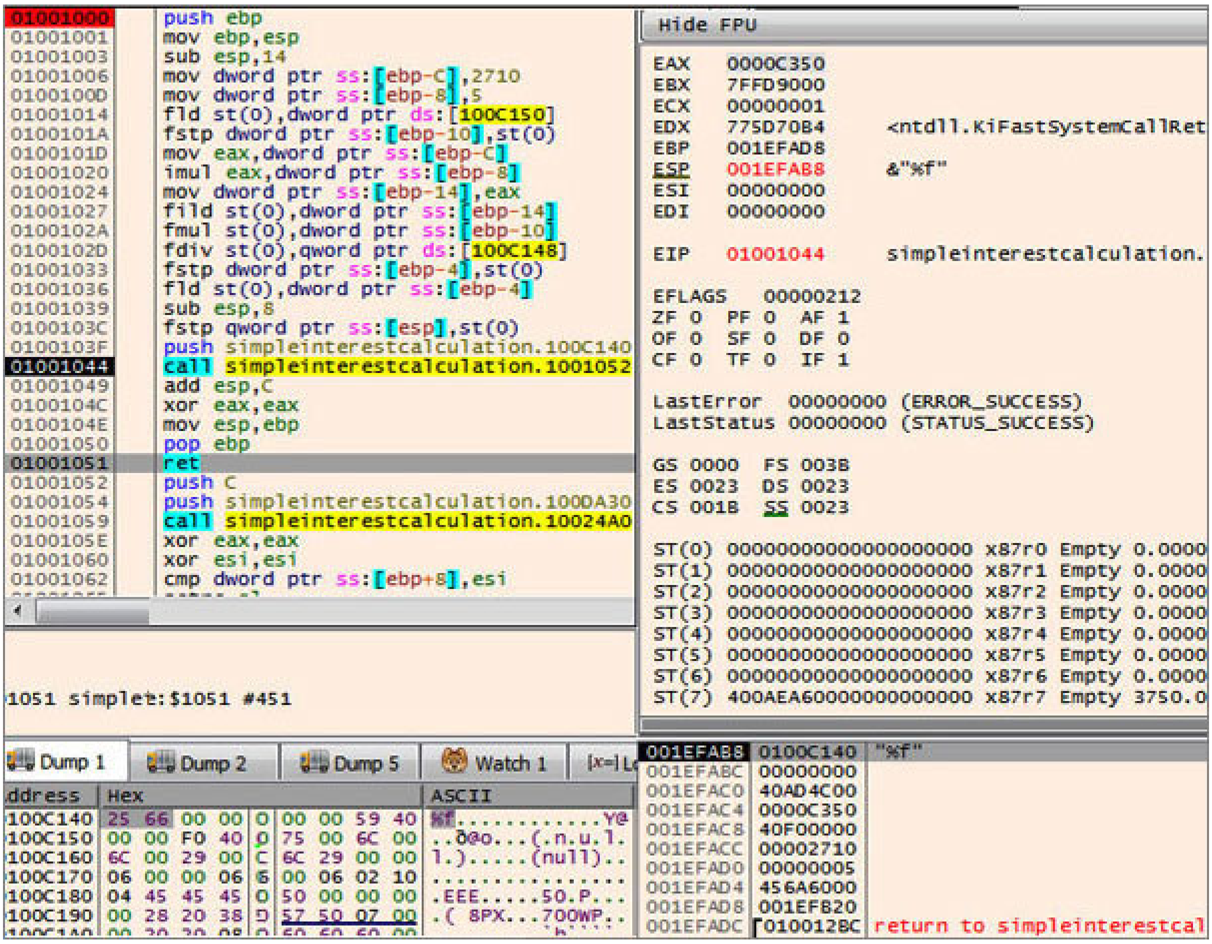
```
ST(0) 00000000000000000000 x87r0 Empty 0.000000000000000000
ST(1) 00000000000000000000 x87r1 Empty 0.000000000000000000
ST(2) 00000000000000000000 x87r2 Empty 0.000000000000000000
ST(3) 00000000000000000000 x87r3 Empty 0.000000000000000000
ST(4) 00000000000000000000 x87r4 Empty 0.000000000000000000
ST(5) 00000000000000000000 x87r5 Empty 0.000000000000000000
ST(6) 00000000000000000000 x87r6 Empty 0.000000000000000000
ST(7) 400AEA60000000000000 x87r7 Empty 3750.000000000000000

001EFABC 00000000
001EFAC0 40AD4C00 fSimpleInterest in QWORD format
001EFAC4 0000C350 iPrincipalAmt * iNoOfYrs
001EFAC8 40F00000 iInterestAmt
001EFACC 00002710 iNoOfYrs
001EFAD0 00000005 fSimpleInterest
001EFAD4 456A6000 EBP
001EFAD8 001EFB20
001EFADC 010012BC return to simpleinterestcalculation.010012BC
```

Figure 15.19: fSimpleInterest from ST(0) to stack

▼Line 60

```
push OFFSET $SG4681
; in x32dbg, push 0x100C140
```

This will push the string constant on the stack.



```
01001000    push ebp
01001001    mov ebp,esp
01001003    sub esp,14
01001006    mov dword ptr ss:[ebp-C],2710
0100100D    mov dword ptr ss:[ebp-8],5
01001014    fld st(0),dword ptr ds:[100C150]
0100101A    fstp dword ptr ss:[ebp-10],st(0)
0100101D    mov eax,dword ptr ss:[ebp-C]
01001020    imul eax,dword ptr ss:[ebp-8]
01001024    mov dword ptr ss:[ebp-14],eax
01001027    fild st(0),dword ptr ss:[ebp-14]
0100102A    fmul st(0),dword ptr ss:[ebp-10]
0100102D    fdiv st(0),qword ptr ds:[100C148]
01001033    fstp dword ptr ss:[ebp-4],st(0)
01001036    fld st(0),dword ptr ss:[ebp-4]
01001039    sub esp,8
0100103C    fstp qword ptr ss:[esp],st(0)
0100103F    push simpleinterestcalculation.100C140
01001044    call simpleinterestcalculation.1001052
01001049    add esp,C
0100104C    xor eax,eax
0100104E    mov esp,ebp
01001050    pop ebp
01001051    ret
01001052    push C
01001054    push simpleinterestcalculation.100DA30
01001059    call simpleinterestcalculation.10024A0
0100105E    xor eax,eax
01001060    xor esi,esi
01001062    cmp dword ptr ss:[ebp+8],esi

1051 simplet:$1051 #451
```

```
Hide FPU

EAX    0000C350
EBX    7FFD9000
ECX    00000001
EDX    775D70B4      <ntdll.KiFastSystemCallRet
EBP    001EFAD8
ESP    001EFAB8      &"%f"
ESI    00000000
EDI    00000000

EIP    01001044      simpleinterestcalculation.

EFLAGS    00000212
ZF 0   PF 0   AF 1
OF 0   SF 0   DF 0
CF 0   TF 0   IF 1

LastError   00000000 (ERROR_SUCCESS)
LastStatus  00000000 (STATUS_SUCCESS)

GS 0000   FS 003B
ES 0023   DS 0023
CS 001B   SS 0023

ST(0) 00000000000000000000 x87r0 Empty 0.0000
ST(1) 00000000000000000000 x87r1 Empty 0.0000
ST(2) 00000000000000000000 x87r2 Empty 0.0000
ST(3) 00000000000000000000 x87r3 Empty 0.0000
ST(4) 00000000000000000000 x87r4 Empty 0.0000
ST(5) 00000000000000000000 x87r5 Empty 0.0000
ST(6) 00000000000000000000 x87r6 Empty 0.0000
ST(7) 400AEA60000000000000 x87r7 Empty 3750.0
```

```
ddress  | Hex                                  | ASCII
100C140 | 25 66 00 00 0| 00 00 59 40 | %f............Y@
100C150 | 00 00 F0 40 0| 75 00 6C 00 | ..ð@o...(.n.u.l.
100C160 | 6C 00 29 00 C| 6C 29 00 00 | l.).....(null)..
100C170 | 06 00 00 06 6| 00 06 02 10 | ................
100C180 | 04 45 45 45 0| 50 00 00 00 | .EEE.....50.P...
100C190 | 00 28 20 38 D| 57 50 07 00 | .( 8PX...700WP..
```

```
001EFAB8  0100C140  "%f"
001EFABC  00000000
001EFAC0  40AD4C00
001EFAC4  0000C350
001EFAC8  40F00000
001EFACC  00002710
001EFAD0  00000005
001EFAD4  456A6000
001EFAD8  001EFB20
001EFADC  0100128C  return to simpleinterestcal
```

Figure 15.20: Before call to printf

▼Line 61-62

```
call _printf
add esp, 12         ; 0000000cH
xor eax, eax
mov esp, ebp
pop ebp
ret 0
_main ENDP
_TEXT ENDS
END
```

Now, the **printf** function arguments are pushed onto the stack; CALL instruction will call the **printf** function. On return as per the CDECL calling convention, the caller cleans the stack. With this, the listing ends by calling the function epilogue and returning 0.

Conclusion

In this chapter, we learned about a real-life software that uses internal mathematical formula to calculate simple interest. We saw how an integer and float are together handled in memory. We studied the assembly pattern of a software with mathematical calculations. In the next chapter, we will reverse engineer a popular ransomware to crack.

CHAPTER 16

Breaking Wannacry Ransomware With Reverse Engineering

Ransomware is a kind of malware that encrypts the victim's file and demands a ransom to decrypt those files. If the ransom is not given within time, the victim's computer data is deleted or left encrypted forever or, sometimes, is sold in the black market. Wannacry was such ransomware which targeted the Windows computers by encrypting data and then demanding a ransom to decrypt the data. Ransom was demanded in the form of Bitcoin cryptocurrency. The impact of Wannacry was so big that it infected millions of computers worldwide and, moreover, it also infected Apple & other servers' OS. Information Security Newspaper reported that many big companies' manufacturing plants, like Honda's, was shut down after some of their computers got infected with Wannacry. Check the following link for reference:

https://www.securitynewspaper.com/2017/06/21/one-month-later-wannacry-ransomware-still-shutting-factories/

As a reverse engineer, we will analyze and break the Wannacry ransomware. When we say 'break it', it means that we will try to dig into the code flow of Wannacry and find something that can change the operation to make it ineffective. We will use the reverse engineering framework called Ghidra to analyze and break the Wannacry ransomware.

Structure

In this chapter, we will cover the following topics:

Installation of reverse engineering framework called Ghidra

How to analyze malware using reverse engineering

How to kill Wannacry malware

Objective

The objective of this chapter is to understand the steps involved in installing the reverse engineering framework called Ghidra. After installing it, we will analyze the Wannacry malware and find the kill switch of Wannacry. The attack of Wannacry was stopped after a few days of the kill switch discovery. It affected thousands of computers in 150 countries with a loss of billions of dollars.

Installation

As we have covered the basic introduction of Ghidra in *Chapter 3, Up and Running with Reverse Engineering* in this chapter we will walk over the installation of the reverse engineering framework. Ghidra installation is very simple and we will use the Windows 10 64-bit version to carry out our Ghidra installation. Follow the given procedure step by step to install Ghidra:

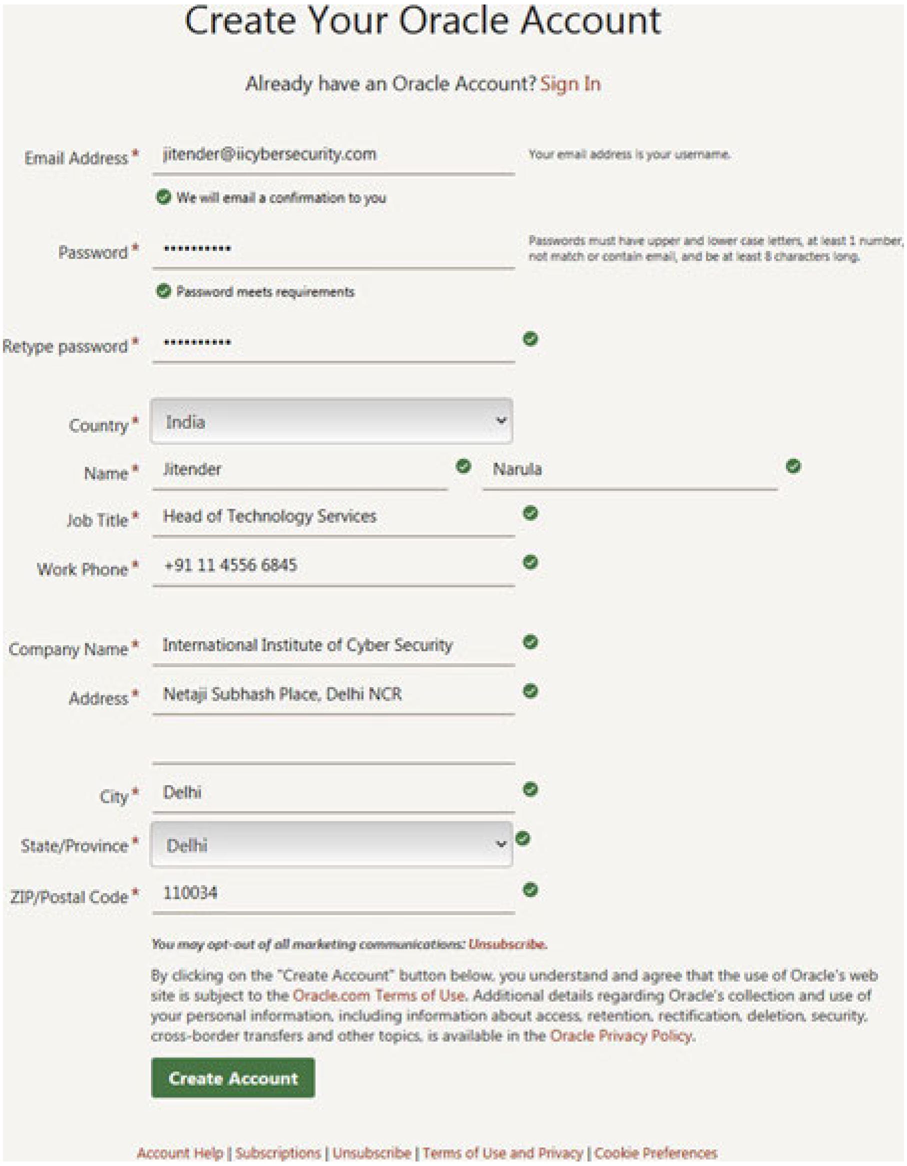
Download and install JDK 11 from the official website of Oracle –

https://www.oracle.com/java/technologies/javase-jdk11-downloads.html

Select Windows x64 Installer for download.

While downloading JDK, you will be redirected to create an account on the Oracle website as follows:

https://profile.oracle.com/myprofile/account/create-account.jspx



Create Your Oracle Account

Already have an Oracle Account? Sign In

Email Address * jitender@iicybersecurity.com

Your email address is your username.

We will email a confirmation to you

Password * ••••••••••

Passwords must have upper and lower case letters, at least 1 number, not match or contain email, and be at least 8 characters long.

Password meets requirements

Retype password * ••••••••••

Country * India

Name * Jitender Narula

Job Title * Head of Technology Services

Work Phone * +91 11 4556 6845

Company Name * International Institute of Cyber Security

Address * Netaji Subhash Place, Delhi NCR

City * Delhi

State/Province * Delhi

ZIP/Postal Code * 110034

You may opt-out of all marketing communications: Unsubscribe.

By clicking on the "Create Account" button below, you understand and agree that the use of Oracle's web site is subject to the Oracle.com Terms of Use. Additional details regarding Oracle's collection and use of your personal information, including information about access, retention, rectification, deletion, security, cross-border transfers and other topics, is available in the Oracle Privacy Policy.

Figure 16.1: *Registration for JDK download*

After creating an account, login to download and install the JDK on your machine.

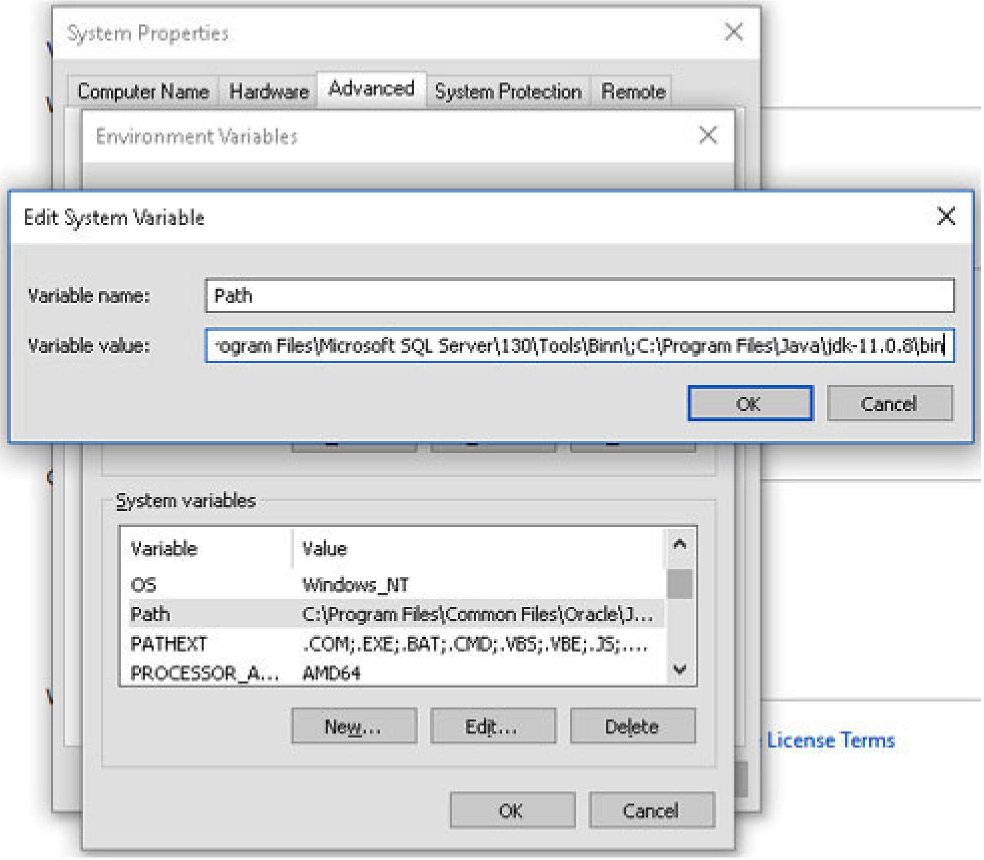
Add the JDK bin directory to your windows machine PATH variable. To do that, follow:

Right-click on the Windows **Start** button and click on

Click on **Advanced system**

Under **Advance** TAB in the **System Properties** window, click on **Environment**

Under **System** edit the **PATH** variable. At the end of the variable, add a semicolon followed by path of JDK dir>\bin as follows:



Variable name: Path

Variable value: ·ogram Files\Microsoft SQL Server\130\Tools\Binn\;C:\Program Files\Java\jdk-11.0.8\bin

System variables

Variable	Value
OS	Windows_NT
Path	C:\Program Files\Common Files\Oracle\J...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;....
PROCESSOR_A...	AMD64

Figure 16.2: *Add JDK to PATH*

Now the JDK path is set. It is time to download Ghidra. Do so from the following link:

https://ghidra-sre.org/

Extract the downloaded folder and run **ghidraRun.bat** as follows:

This PC > Desktop > ghidra_9.2_PUBLIC_20201113 > ghidra_9.2_PUBLIC

Name	Date modified	Type
docs	12/10/2020 1:04 AM	File folder
Extensions	12/10/2020 1:05 AM	File folder
Ghidra	12/10/2020 1:16 AM	File folder
GPL	12/10/2020 1:05 AM	File folder
licenses	12/10/2020 1:04 AM	File folder
server	12/10/2020 1:04 AM	File folder
support	12/10/2020 1:05 AM	File folder
ghidraRun	12/10/2020 1:04 AM	File
ghidraRun.bat	12/10/2020 1:39 AM	Windows Batch File
LICENSE	12/10/2020 1:04 AM	File

Figure 16.3: *Start Ghidra*

It will prompt you to **Agree the Ghidra User** On accepting, you will get the **NO ACTIVE PROJECT** screen as we have no active projects on Ghidra.

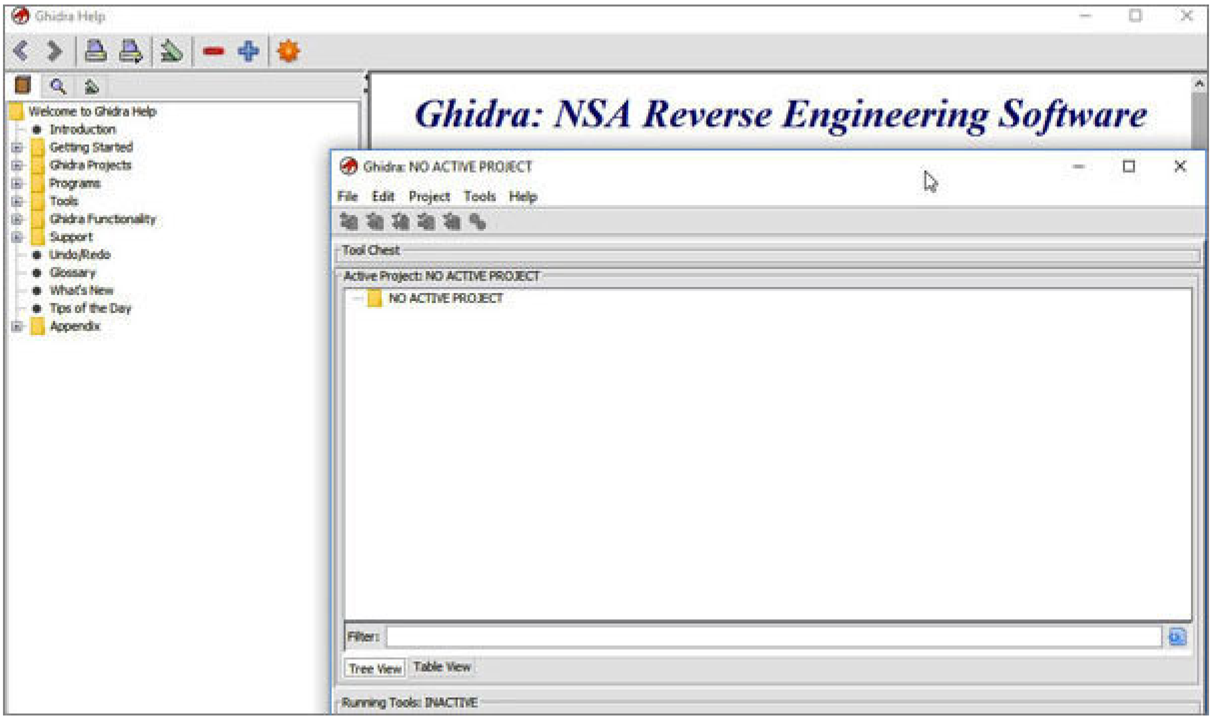


Figure 16.4: *Ghidra first screen*

With this, we have successfully installed and started Ghidra.

Analyzing and Breaking Wannacry

Before analyzing the Wannacry ransomware, we need to get the copy of the ransomware from the following website. This is 32-bit version of the ransomware binary.

https://www.ghidra.ninja/samples/wannacry.zip

Note: Do not run executable file compressed in the Wannacry zip file on your machine. If you want to run and check how Wannacry malware works, then do it on a virtual machine with no important data on it. Once this binary is executed on machine, you will not be able to access files on that virtual machine.

If you run the Wannacry executable or binary on your virtual machine, disable **Defender** to get the following screen:

Ooops, your files have been encrypted!

What Happened to My Computer?

Your important files are encrypted.

Many of your documents, photos, videos, databases and other files are no longer accessible because they have been encrypted. Maybe you are busy looking for a way to recover your files, but do not waste your time. Nobody can recover your files without our decryption service.

Can I Recover My Files?

Sure. We guarantee that you can recover all your files safely and easily. But you have not so enough time.

You can decrypt some of your files for free. Try now by clicking <Decrypt>.

But if you want to decrypt all your files, you need to pay.

You only have 3 days to submit the payment. After that the price will be doubled.

Also, if you don't pay in 7 days, you won't be able to recover your files forever.

We will have free events for users who are so poor that they couldn't pay in 6 months.

How Do I Pay?

Payment is accepted in Bitcoin only. For more information, click <About bitcoin>.

Please check the current price of Bitcoin and buy some bitcoins. For more information, click <How to buy bitcoins>.

And send the correct amount to the address specified in this window.

After your payment, click <Check Payment>. Best time to check: 9:00am - 11:00am

Payment will be raised on

3/28/2019 13:44:14

Time Left

00:00:00:00

Your files will be lost on

4/1/2019 14:44:14

Time Left

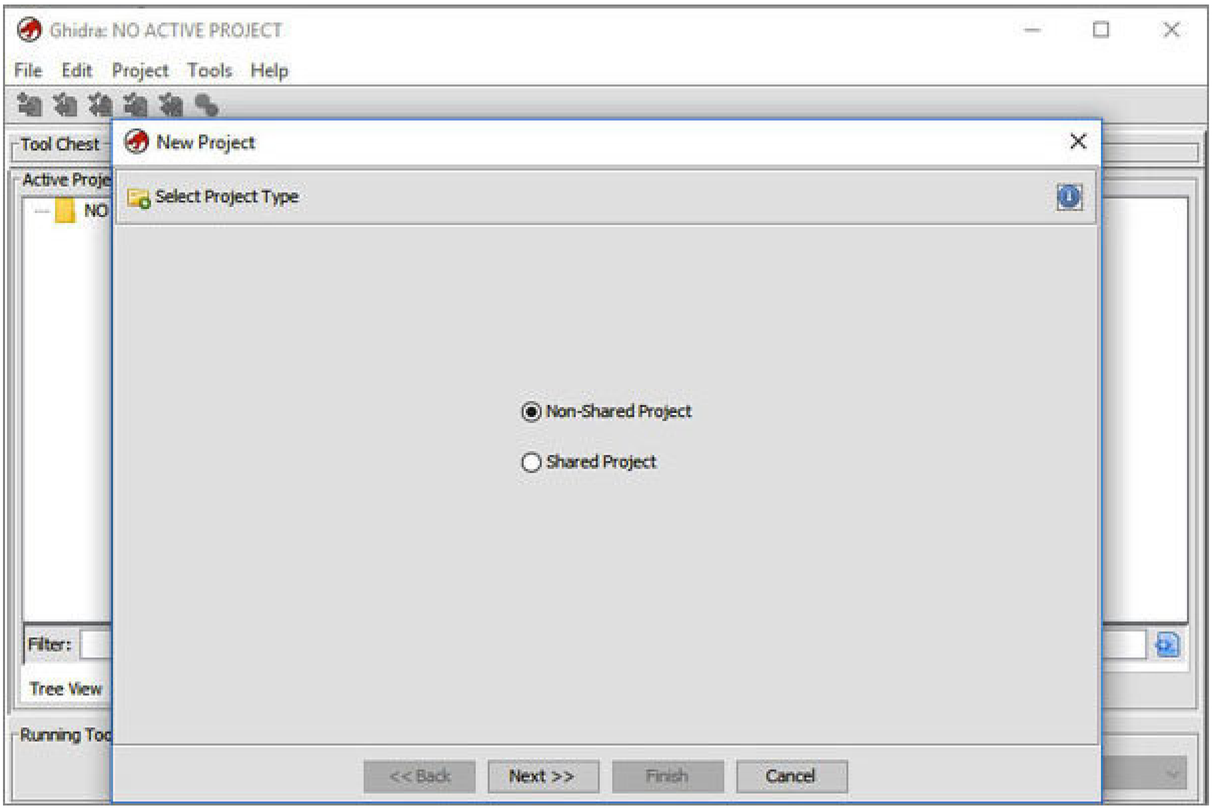
00:00:00:00

Send $300 worth of bitcoin to this address:

12t9YDPgwueZ9NyMgw519p7AA8isjr6SMw

Figure 16.5: Wannacry

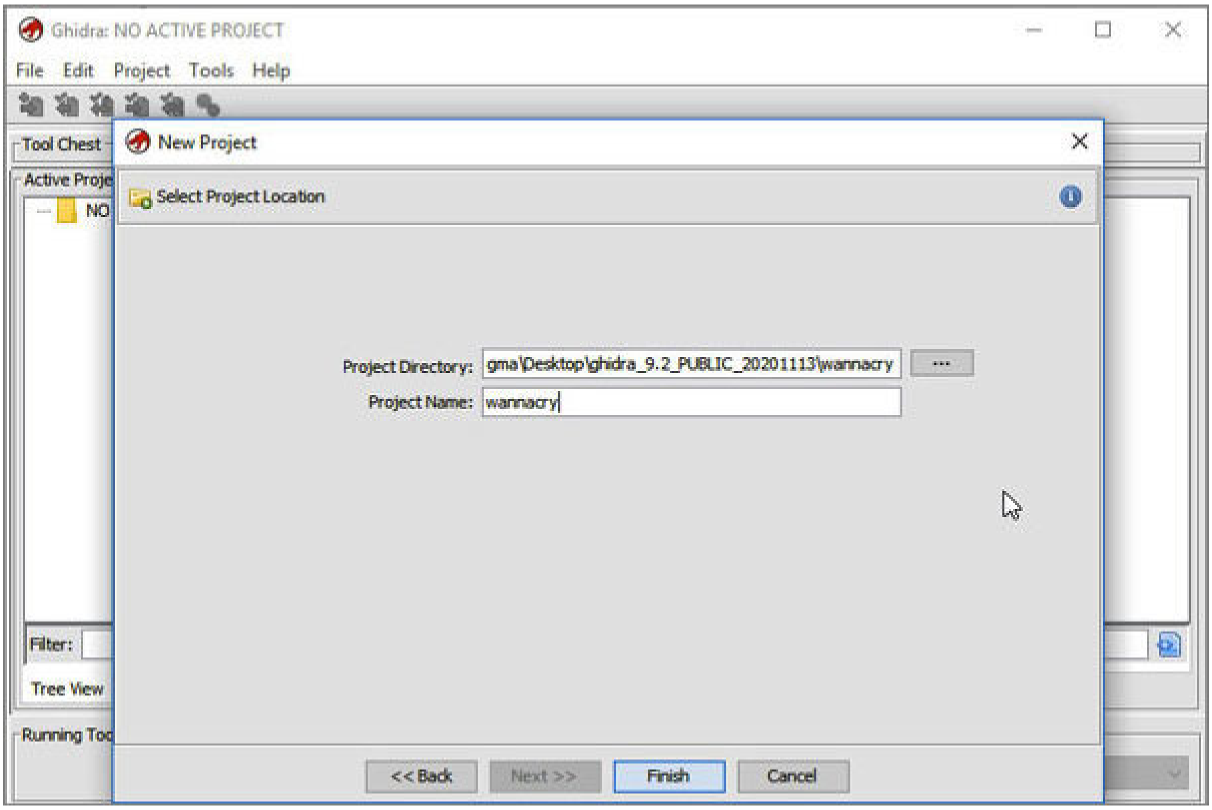
Let's load Wannacry into Ghidra by downloading the Wannacry zip (password: from the preceding link and extract it. Open Ghidra to create a new project from **File** > **New** Name it



● Non-Shared Project

○ Shared Project

Figure 16.6: Ghidra new project screen

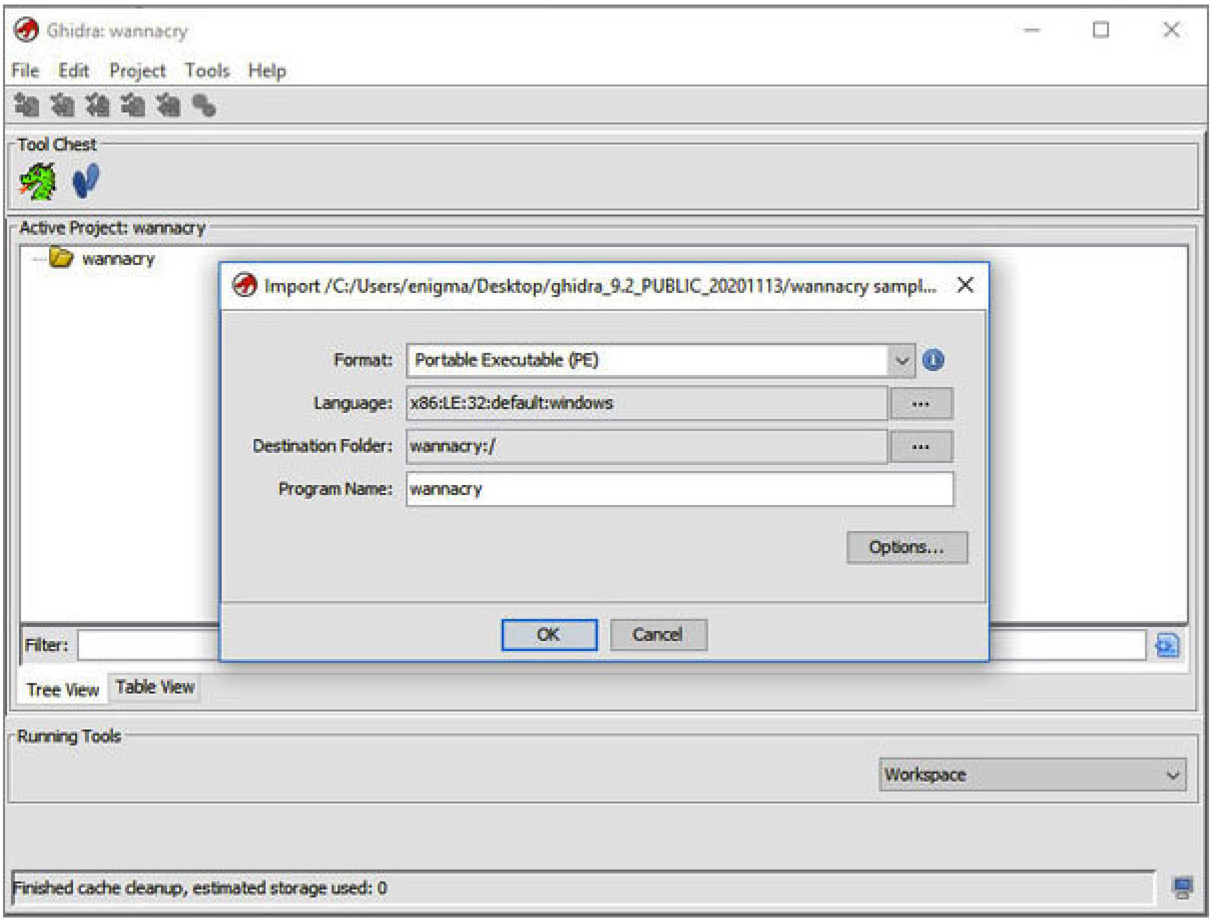


Project Directory: gma\Desktop\ghidra_9.2_PUBLIC_20201113\wannacry

Project Name: wannacry

Figure 16.7: Project name

On clicking your Wannacry project will come under **Active** Drag and drop the Wannacry executable under **Active Project** and it will import the binary and flash the following screen:



Import /C:/Users/enigma/Desktop/ghidra_9.2_PUBLIC_20201113/wannacry sampl...

Format: Portable Executable (PE)

Language: x86:LE:32:default:windows

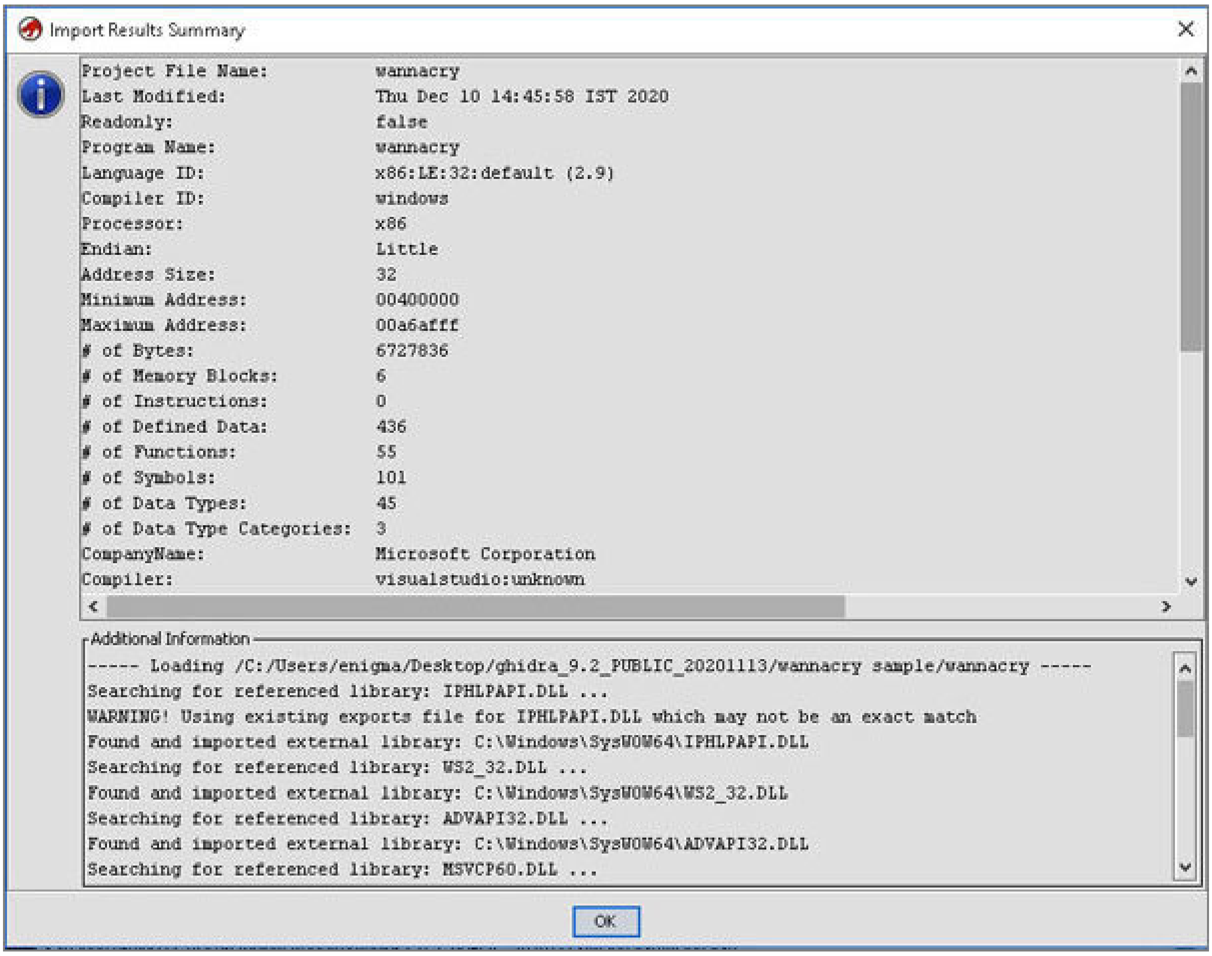
Destination Folder: wannacry:/

Program Name: wannacry

Finished cache cleanup, estimated storage used: 0

Figure 16.8: *Drag drop wannacry binary*

On clicking the next screen will show the **Import Result Summary** for the Wannacry binary. This will list the Executable format, Compiler ID, Processor, Endianess, and many other components as follows:

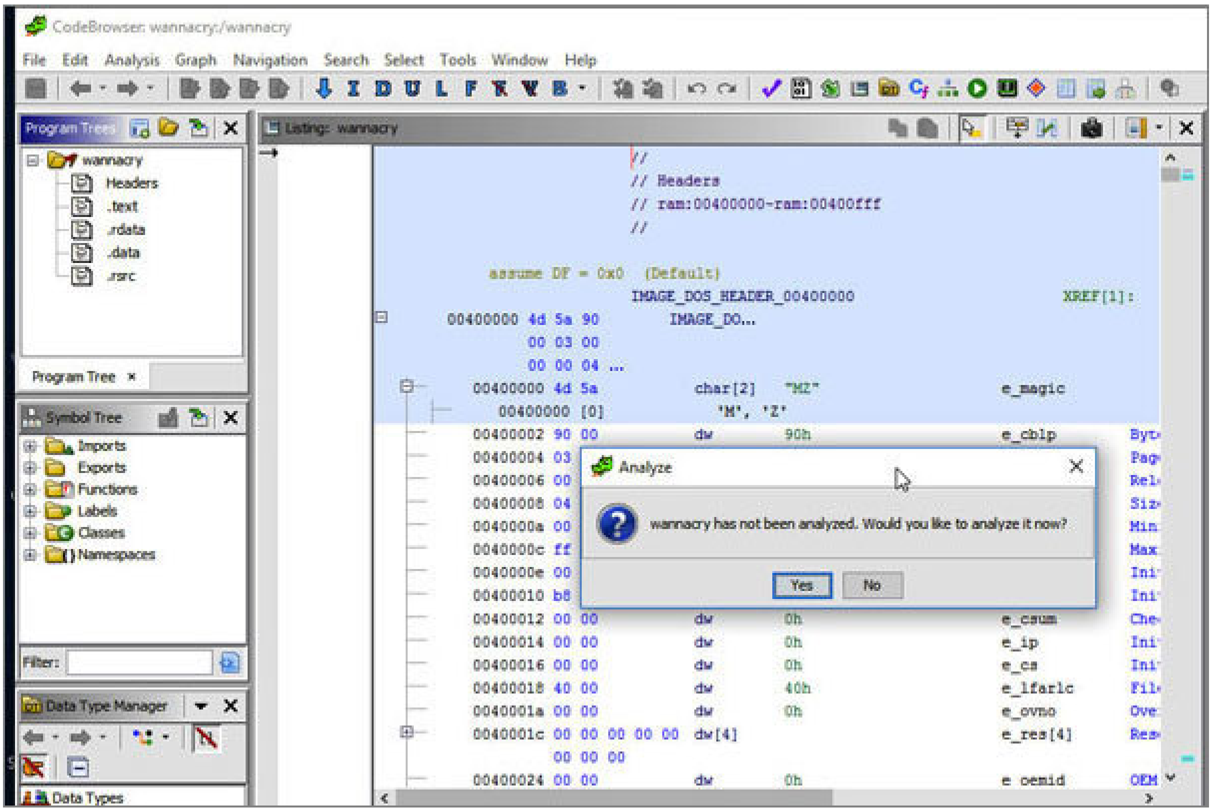


```
Project File Name:           wannacry
Last Modified:               Thu Dec 10 14:45:58 IST 2020
Readonly:                    false
Program Name:                wannacry
Language ID:                 x86:LE:32:default (2.9)
Compiler ID:                 windows
Processor:                   x86
Endian:                      Little
Address Size:                32
Minimum Address:             00400000
Maximum Address:             00a6afff
# of Bytes:                  6727836
# of Memory Blocks:          6
# of Instructions:           0
# of Defined Data:           436
# of Functions:              55
# of Symbols:                101
# of Data Types:             45
# of Data Type Categories:   3
CompanyName:                 Microsoft Corporation
Compiler:                    visualstudio:unknown
```

```
----- Loading /C:/Users/enigma/Desktop/ghidra_9.2_PUBLIC_20201113/wannacry sample/wannacry -----
Searching for referenced library: IPHLPAPI.DLL ...
WARNING! Using existing exports file for IPHLPAPI.DLL which may not be an exact match
Found and imported external library: C:\Windows\SysWOW64\IPHLPAPI.DLL
Searching for referenced library: WS2_32.DLL ...
Found and imported external library: C:\Windows\SysWOW64\WS2_32.DLL
Searching for referenced library: ADVAPI32.DLL ...
Found and imported external library: C:\Windows\SysWOW64\ADVAPI32.DLL
Searching for referenced library: MSVCP60.DLL ...
```

Figure 16.9: *Import result summary*

Press **OK** and double click on the **wannacry** project to open **CodeBrowser** as follows. In the it will ask you to analyze the executable.

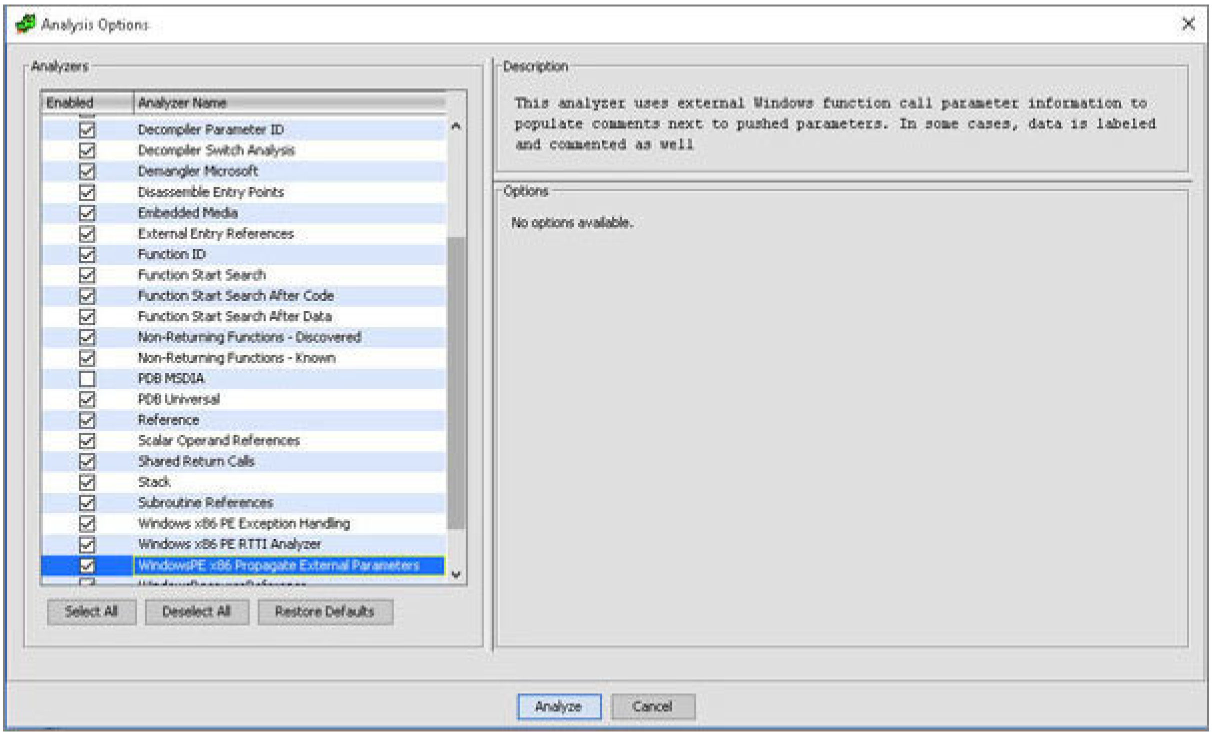


```
                                //
                                // Headers
                                // ram:00400000-ram:00400fff
                                //

             assume DF = 0x0  (Default)
                                IMAGE_DOS_HEADER_00400000                    XREF[1]:
        00400000 4d 5a 90          IMAGE_DO...
                 00 03 00
                 00 00 04 ...
          00400000 4d 5a          char[2]    "MZ"              e_magic
            00400000 [0]            'M', 'Z'
          00400002 90 00          dw         90h               e_cblp      Byt
          00400004 03                                                      Pag
          00400006 00                                                      Rel
          00400008 04                                                      Siz
          0040000a 00                                                      Min
          0040000c ff                                                      Max
          0040000e 00                                                      Ini
          00400010 b8                                                      Ini
          00400012 00 00          dw         0h                e_csum      Che
          00400014 00 00          dw         0h                e_ip        Ini
          00400016 00 00          dw         0h                e_cs        Ini
          00400018 40 00          dw         40h               e_lfarlc    Fil
          0040001a 00 00          dw         0h                e_ovno      Ove
          0040001c 00 00 00 00 00 dw[4]                        e_res[4]    Res
                   00 00 00
          00400024 00 00          dw         0h                e_oemid     OEM
```

wannacry has not been analyzed. Would you like to analyze it now?

Figure 16.10: *Wannacry analyze now screen*

Click **Yes** to continue. You will have to enable a few analysis options like **WindowsPE x86 Propagate External Parameter** and **Decompiler Parameter**



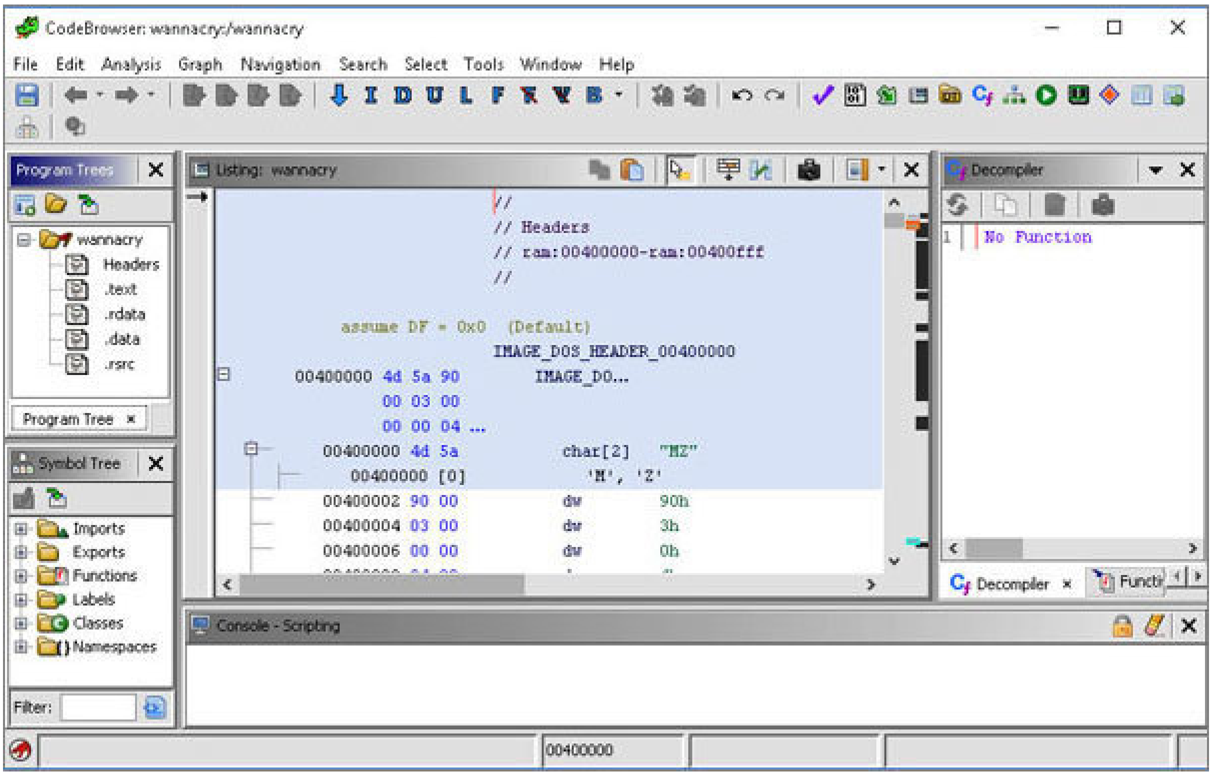
Enabled	Analyzer Name
☑	Decompiler Parameter ID
☑	Decompiler Switch Analysis
☑	Demangler Microsoft
☑	Disassemble Entry Points
☑	Embedded Media
☑	External Entry References
☑	Function ID
☑	Function Start Search
☑	Function Start Search After Code
☑	Function Start Search After Data
☑	Non-Returning Functions - Discovered
☑	Non-Returning Functions - Known
☐	PDB MSDIA
☑	PDB Universal
☑	Reference
☑	Scalar Operand References
☑	Shared Return Calls
☑	Stack
☑	Subroutine References
☑	Windows x86 PE Exception Handling
☑	Windows x86 PE RTTI Analyzer
☑	WindowsPE x86 Propagate External Parameters

This analyzer uses external Windows function call parameter information to populate comments next to pushed parameters. In some cases, data is labeled and commented as well

No options available.

Figure 16.11: Enable analysis options

Once enabled, click on **Analyze** and ignore the warnings if any.



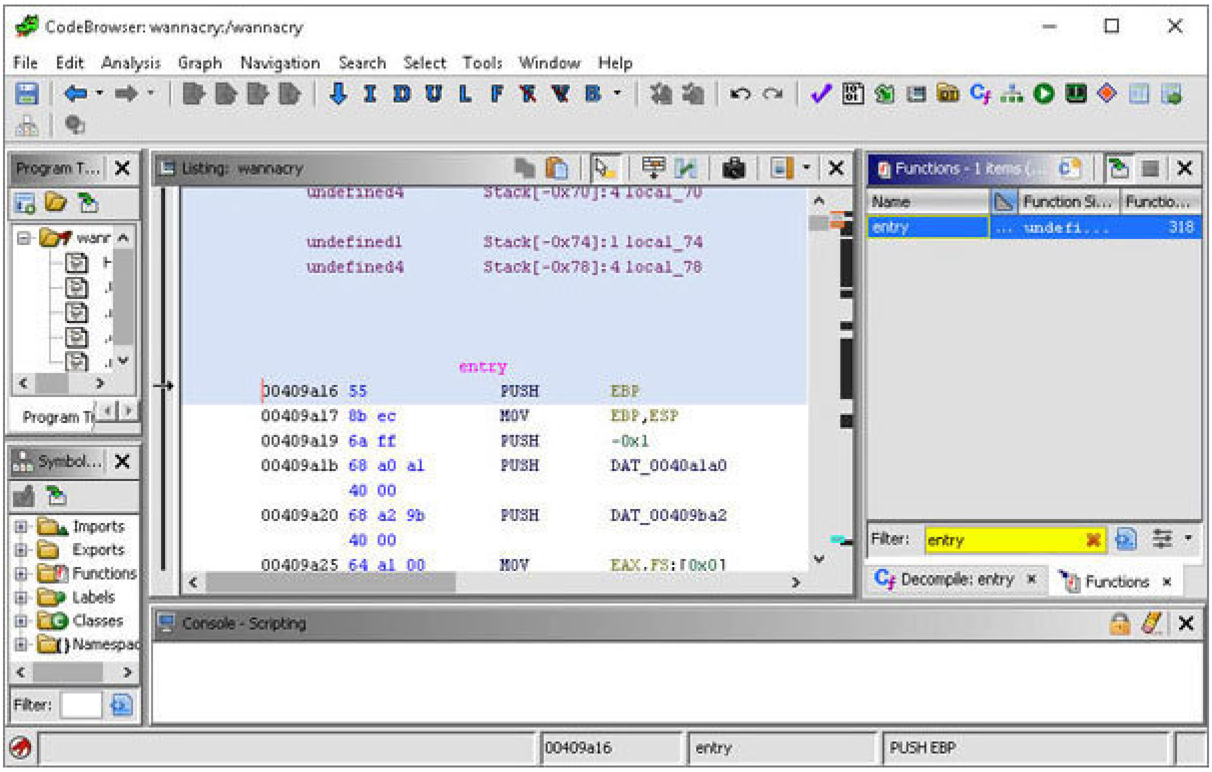
```
                     //
                     // Headers
                     // ram:00400000-ram:00400fff
                     //

         assume DF = 0x0  (Default)
                     IMAGE_DOS_HEADER_00400000
    00400000 4d 5a 90       IMAGE_DO...
             00 03 00
             00 00 04 ...
      00400000 4d 5a          char[2]    "MZ"
        00400000 [0]                'M', 'Z'
      00400002 90 00          dw         90h
      00400004 03 00          dw         3h
      00400006 00 00          dw         0h
```

```
1  No Function
```

00400000

***Figure 16.12:** Disassembly view*

On the right, we see **Program Trees** which displays the PE section details. In the **Symbol** it displays the symbols currently defined in the program. The middle screen is the disassembled view of the binary and on the left, we see It shows **No Function** as we have not selected any function from the **Functions TAB** on the bottom of the **Decompiler** window. To view the entry point of this executable, type in **entry** in the functions tab and double click on the entry function to view its disassembled and decompiled code.



```
undefined4      Stack[-0x70]:4 local_70

undefined1      Stack[-0x74]:1 local_74
undefined4      Stack[-0x78]:4 local_78



                     entry
00409a16 55              PUSH    EBP
00409a17 8b ec           MOV     EBP,ESP
00409a19 6a ff           PUSH    -0x1
00409a1b 68 a0 a1        PUSH    DAT_0040a1a0
         40 00
00409a20 68 a2 9b        PUSH    DAT_00409ba2
         40 00
00409a25 64 a1 00        MOV     EAX,FS:[0x0]
```

Name	Function Si...	Functio...
entry	... undefi...	318

Filter: entry

00409a16 | entry | PUSH EBP

***Figure 16.13:** Entry function*

The assembly code we see in the listing section of the preceding screenshot is the entry code of the binary. This entry function is a **main()** or **WinMain()** function. For breaking Wannacry, we will have to analyze and understand the code flow. In this process of understanding the code flow, we will first see all the function calls made from the entry disassembled function. This is done by going to the menu **Windows** > **Function Call** A window with all the calls made from the entry function will be shown in a graphical way:

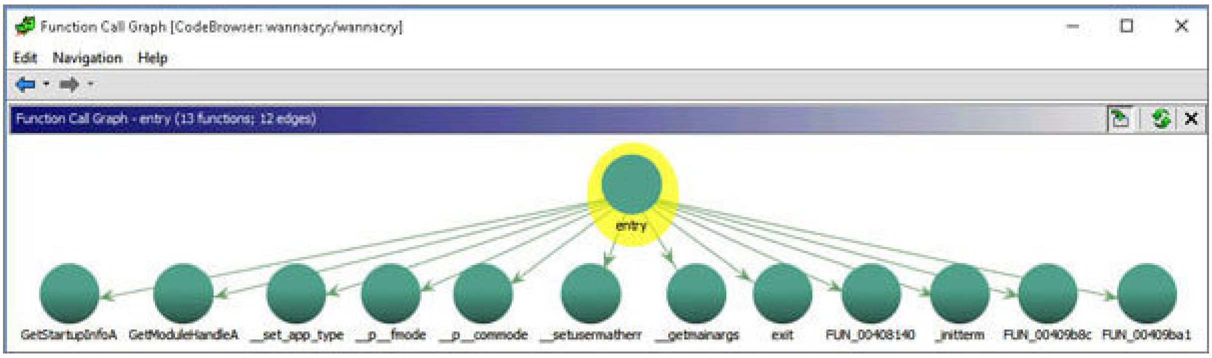
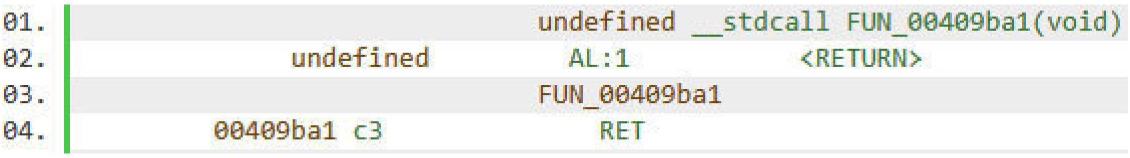


Figure 16.14: *Entry function call graph*

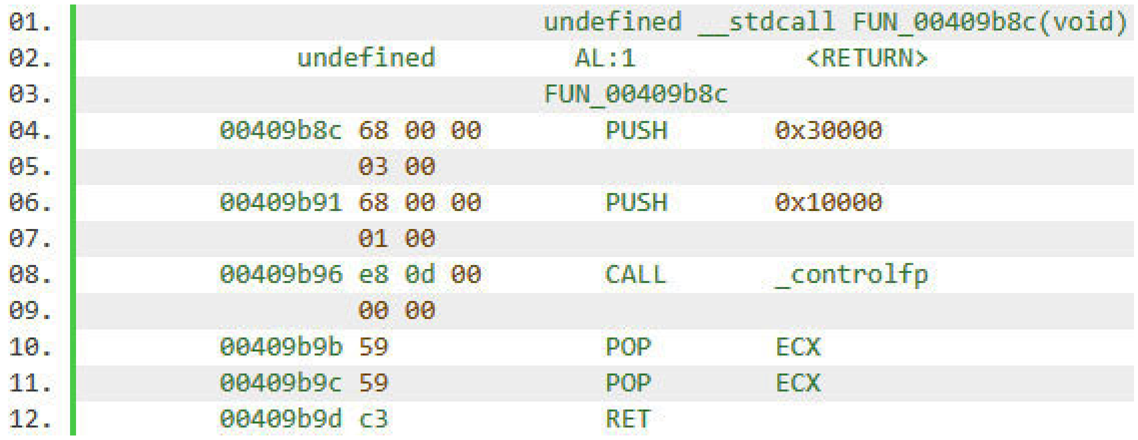
Most of the calls are made to the internal libraries functions. Let's analyze the functions of our interest.



```
                                undefined __stdcall FUN_00409ba1(void)
               undefined          AL:1           <RETURN>
                                FUN_00409ba1
            00409ba1 c3             RET
```

Figure 16.15: *FUN_00409ba1*

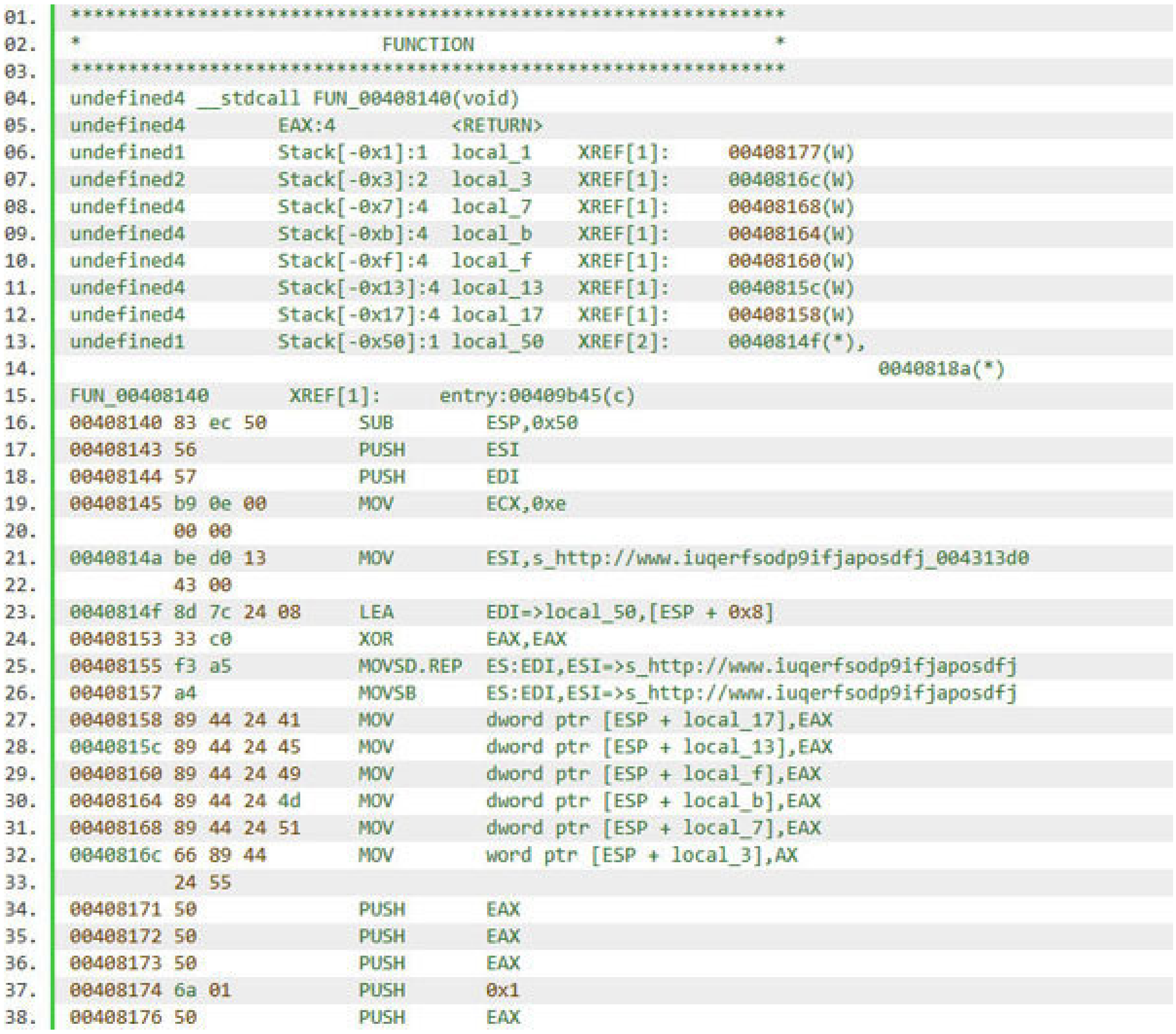
FUN_00409ba1 is just a so it does not revealing anything. Move to the next function as follows:



```
                              undefined __stdcall FUN_00409b8c(void)
            undefined           AL:1             <RETURN>
                              FUN_00409b8c
        00409b8c 68 00 00          PUSH          0x30000
                 03 00
        00409b91 68 00 00          PUSH          0x10000
                 01 00
        00409b96 e8 0d 00          CALL          _controlfp
                 00 00
        00409b9b 59                POP           ECX
        00409b9c 59                POP           ECX
        00409b9d c3                RET
```

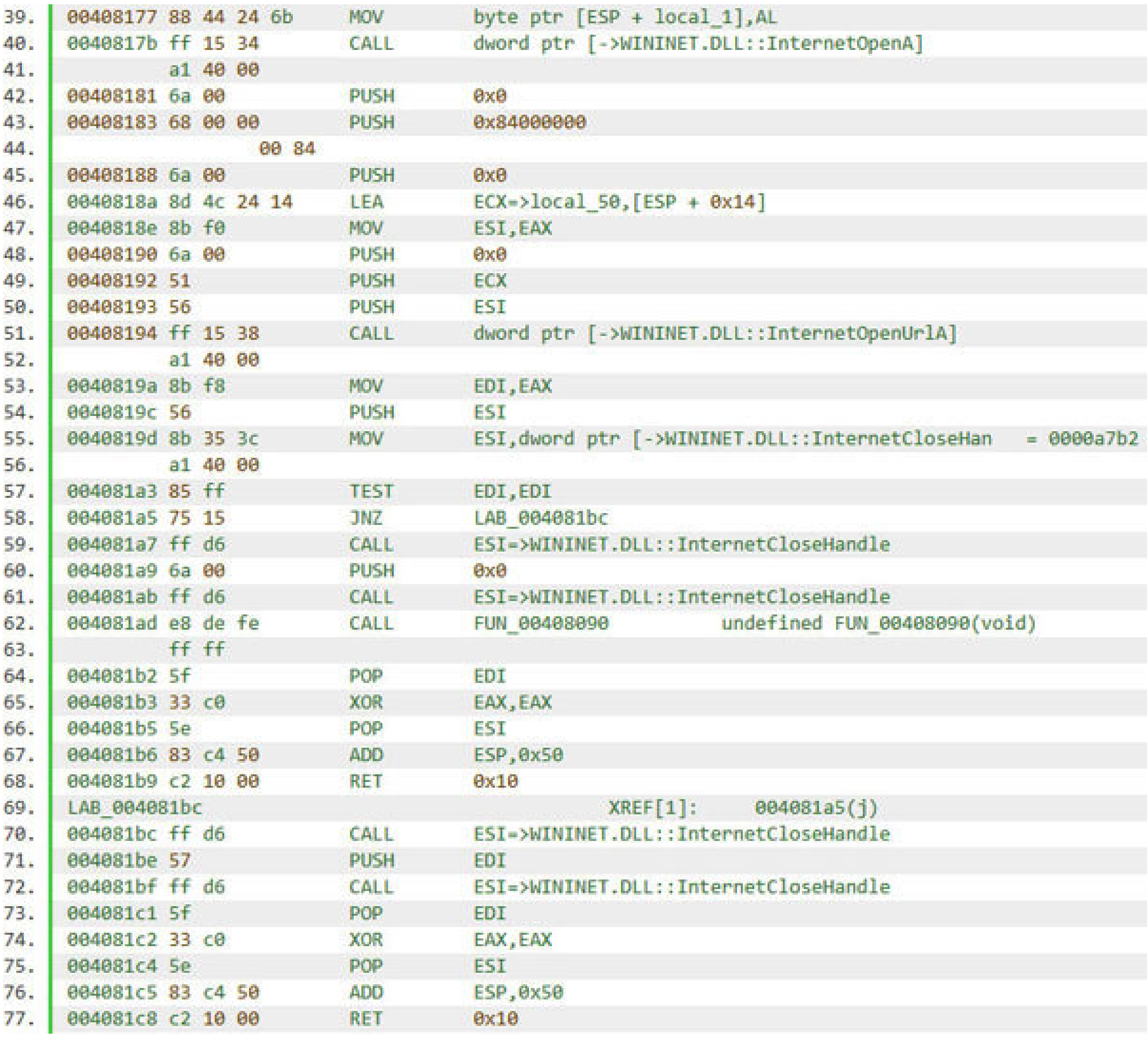
Figure 16.16: *FUN_00409b8c*

FUN_00409b8c also does not revealing anything. Let's move to the next function, as follows:



```
******************************************************************
*                           FUNCTION                             *
******************************************************************
undefined4 __stdcall FUN_00408140(void)
undefined4         EAX:4          <RETURN>
undefined1         Stack[-0x1]:1  local_1    XREF[1]:     00408177(W)
undefined2         Stack[-0x3]:2  local_3    XREF[1]:     0040816c(W)
undefined4         Stack[-0x7]:4  local_7    XREF[1]:     00408168(W)
undefined4         Stack[-0xb]:4  local_b    XREF[1]:     00408164(W)
undefined4         Stack[-0xf]:4  local_f    XREF[1]:     00408160(W)
undefined4         Stack[-0x13]:4 local_13   XREF[1]:     0040815c(W)
undefined4         Stack[-0x17]:4 local_17   XREF[1]:     00408158(W)
undefined1         Stack[-0x50]:1 local_50   XREF[2]:     0040814f(*),
                                                                       0040818a(*)
FUN_00408140       XREF[1]:     entry:00409b45(c)
00408140 83 ec 50         SUB        ESP,0x50
00408143 56               PUSH       ESI
00408144 57               PUSH       EDI
00408145 b9 0e 00         MOV        ECX,0xe
         00 00
0040814a be d0 13         MOV        ESI,s_http://www.iuqerfsodp9ifjaposdfj_004313d0
         43 00
0040814f 8d 7c 24 08      LEA        EDI=>local_50,[ESP + 0x8]
00408153 33 c0            XOR        EAX,EAX
00408155 f3 a5            MOVSD.REP  ES:EDI,ESI=>s_http://www.iuqerfsodp9ifjaposdfj
00408157 a4               MOVSB      ES:EDI,ESI=>s_http://www.iuqerfsodp9ifjaposdfj
00408158 89 44 24 41      MOV        dword ptr [ESP + local_17],EAX
0040815c 89 44 24 45      MOV        dword ptr [ESP + local_13],EAX
00408160 89 44 24 49      MOV        dword ptr [ESP + local_f],EAX
00408164 89 44 24 4d      MOV        dword ptr [ESP + local_b],EAX
00408168 89 44 24 51      MOV        dword ptr [ESP + local_7],EAX
0040816c 66 89 44         MOV        word ptr [ESP + local_3],AX
         24 55
00408171 50               PUSH       EAX
00408172 50               PUSH       EAX
00408173 50               PUSH       EAX
00408174 6a 01            PUSH       0x1
00408176 50               PUSH       EAX
```

Figure 16.17: *FUN_00408140-Part-1*



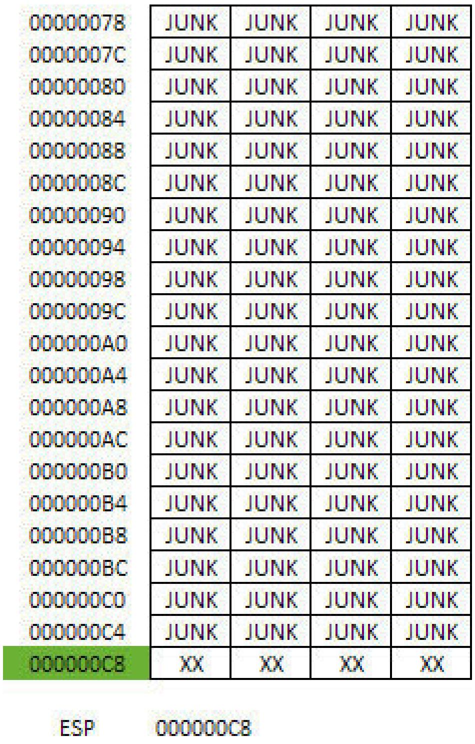
```
00408177 88 44 24 6b     MOV        byte ptr [ESP + local_1],AL
0040817b ff 15 34        CALL       dword ptr [->WININET.DLL::InternetOpenA]
         a1 40 00
00408181 6a 00           PUSH       0x0
00408183 68 00 00        PUSH       0x84000000
                 00 84
00408188 6a 00           PUSH       0x0
0040818a 8d 4c 24 14     LEA        ECX=>local_50,[ESP + 0x14]
0040818e 8b f0           MOV        ESI,EAX
00408190 6a 00           PUSH       0x0
00408192 51              PUSH       ECX
00408193 56              PUSH       ESI
00408194 ff 15 38        CALL       dword ptr [->WININET.DLL::InternetOpenUrlA]
         a1 40 00
0040819a 8b f8           MOV        EDI,EAX
0040819c 56              PUSH       ESI
0040819d 8b 35 3c        MOV        ESI,dword ptr [->WININET.DLL::InternetCloseHan   = 0000a7b2
         a1 40 00
004081a3 85 ff           TEST       EDI,EDI
004081a5 75 15           JNZ        LAB_004081bc
004081a7 ff d6           CALL       ESI=>WININET.DLL::InternetCloseHandle
004081a9 6a 00           PUSH       0x0
004081ab ff d6           CALL       ESI=>WININET.DLL::InternetCloseHandle
004081ad e8 de fe        CALL       FUN_00408090          undefined FUN_00408090(void)
         ff ff
004081b2 5f              POP        EDI
004081b3 33 c0           XOR        EAX,EAX
004081b5 5e              POP        ESI
004081b6 83 c4 50        ADD        ESP,0x50
004081b9 c2 10 00        RET        0x10
LAB_004081bc                                    XREF[1]:     004081a5(j)
004081bc ff d6           CALL       ESI=>WININET.DLL::InternetCloseHandle
004081be 57              PUSH       EDI
004081bf ff d6           CALL       ESI=>WININET.DLL::InternetCloseHandle
004081c1 5f              POP        EDI
004081c2 33 c0           XOR        EAX,EAX
004081c4 5e              POP        ESI
004081c5 83 c4 50        ADD        ESP,0x50
004081c8 c2 10 00        RET        0x10
```

Figure 16.18: *FUN_00408140-Part-2*

This function is something of our interest. We will analyze the disassembled code from whatever we have learned from the earlier chapters. Ghidra also decompiles the binary, but we will first go over the disassembled code step by step to review our learning. While analyzing the disassembled code line by line, we will visualize the stack state to get an overview of the stack state during code flow.

Note: We will be doing a static analysis of the disassembled code. We will not execute this binary during our analysis. Static analysis will help us understand the code flow as well as the working of the ransomware.

Before we start the analysis, let’s visualize a stack where **ESP** is pointing on the top of the stack. In the following screenshot, you can see the visualization of the stack in a sequence of 4 cells, one above the other. Each cell denotes a byte and the memory addressing is done on the left side from a higher memory location to a lower memory location as we move up. The memory location marked with green is the location of ESP, pointing to the top of the stack having some data marked with XX bytes. The initial stack state before starting the **FUN_00408140** function will be as follows. Right now, all the cells are filled with some data which is marked as JUNK. With the instructions flow, bytes will be pushed and popped of the stack.



00000078	JUNK	JUNK	JUNK	JUNK
0000007C	JUNK	JUNK	JUNK	JUNK
00000080	JUNK	JUNK	JUNK	JUNK
00000084	JUNK	JUNK	JUNK	JUNK
00000088	JUNK	JUNK	JUNK	JUNK
0000008C	JUNK	JUNK	JUNK	JUNK
00000090	JUNK	JUNK	JUNK	JUNK
00000094	JUNK	JUNK	JUNK	JUNK
00000098	JUNK	JUNK	JUNK	JUNK
0000009C	JUNK	JUNK	JUNK	JUNK
000000A0	JUNK	JUNK	JUNK	JUNK
000000A4	JUNK	JUNK	JUNK	JUNK
000000A8	JUNK	JUNK	JUNK	JUNK
000000AC	JUNK	JUNK	JUNK	JUNK
000000B0	JUNK	JUNK	JUNK	JUNK
000000B4	JUNK	JUNK	JUNK	JUNK
000000B8	JUNK	JUNK	JUNK	JUNK
000000BC	JUNK	JUNK	JUNK	JUNK
000000C0	JUNK	JUNK	JUNK	JUNK
000000C4	JUNK	JUNK	JUNK	JUNK
000000C8	XX	XX	XX	XX

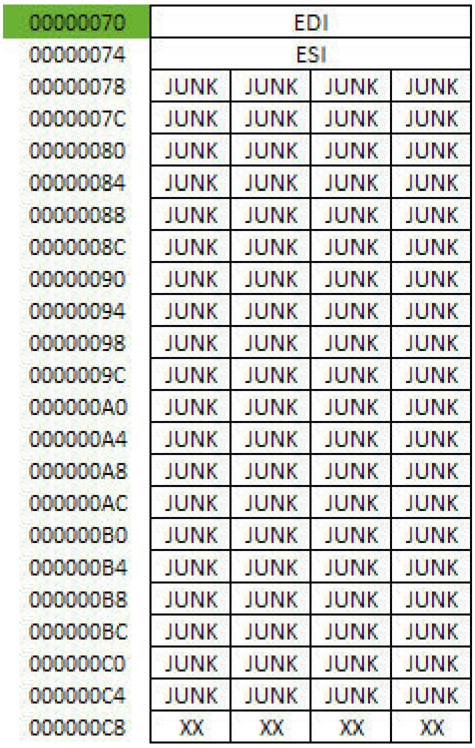
ESP 000000C8

Figure 16.19: Initial stack state

▼Line 01-18

```
undefined4 __stdcall FUN_00408140(void)
undefined4           EAX:4
undefined1           Stack[-0x1]:1   local_1      XREF[1]:
00408177(W)
undefined2           Stack[-0x3]:2   local_3      XREF[1]:
0040816c(W)

undefined4           Stack[-0x7]:4   local_7      XREF[1]:
00408168(W)
undefined4           Stack[-0xb]:4   local_b      XREF[1]:
00408164(W)
undefined4           Stack[-0xf]:4   local_f      XREF[1]:
00408160(W)
undefined4           Stack[-0x13]:4  local_13     XREF[1]:
0040815c(W)
undefined4           Stack[-0x17]:4  local_17     XREF[1]:
00408158(W)
undefined1           Stack[-0x50]:1  local_50     XREF[2]:
0040814f(*),
SUB ESP,0x50
PUSH ESI
PUSH EDI
```

At the start of we see that different variable macros are defined. **SUB** is creating room for variables on the stack by subtracting 0x50 from After creating some room, the **ESI** and **EDI** values are preserved on the stack by pushing them on the stack. The stack state after these instructions will be as follows:



00000070	EDI			
00000074	ESI			
00000078	JUNK	JUNK	JUNK	JUNK
0000007C	JUNK	JUNK	JUNK	JUNK
00000080	JUNK	JUNK	JUNK	JUNK
00000084	JUNK	JUNK	JUNK	JUNK
00000088	JUNK	JUNK	JUNK	JUNK
0000008C	JUNK	JUNK	JUNK	JUNK
00000090	JUNK	JUNK	JUNK	JUNK
00000094	JUNK	JUNK	JUNK	JUNK
00000098	JUNK	JUNK	JUNK	JUNK
0000009C	JUNK	JUNK	JUNK	JUNK
000000A0	JUNK	JUNK	JUNK	JUNK
000000A4	JUNK	JUNK	JUNK	JUNK
000000A8	JUNK	JUNK	JUNK	JUNK
000000AC	JUNK	JUNK	JUNK	JUNK
000000B0	JUNK	JUNK	JUNK	JUNK
000000B4	JUNK	JUNK	JUNK	JUNK
000000B8	JUNK	JUNK	JUNK	JUNK
000000BC	JUNK	JUNK	JUNK	JUNK
000000C0	JUNK	JUNK	JUNK	JUNK
000000C4	JUNK	JUNK	JUNK	JUNK
000000C8	XX	XX	XX	XX

Figure 16.20: *After creating room for the local variables*

▼Line 19-32

```
MOV  ECX,0xe
MOV  ESI,s_http://www.iuqerfsodp9ifjaposdfj_004313d0
LEA  EDI=>local_50,[ESP + 0x8]
XOR  EAX,EAX
MOVSD.REP  ES:EDI,ESI=>s_http://www.iuqerfsodp9ifjaposdfj
MOVSB  ES:EDI,ESI=>s_http://www.iuqerfsodp9ifjaposdfj

MOV  dword ptr [ESP + local_17],EAX
MOV  dword ptr [ESP + local_13],EAX
```

```
MOV dword ptr [ESP + local_f],EAX
MOV dword ptr [ESP + local_b],EAX
MOV dword ptr [ESP + local_7],EAX
MOV word ptr [ESP + local_3],AX
```

With the **MOV** instruction, **ECX** is filled with 0x0E which will act as a counter to the loop instruction afterwards.

ESI is pointing to offset

The length of this URL string is 57 bytes, where one byte is for null termination.

The **Load effective address** instruction is loading the address of **ESP+0x08** in EDI.

In these instructions, you will see the **REP** and **MOVS** instruction. The **REP** instruction repeats the string operation **ECX** times, where **ECX** is initialized to 0x0E, which is 14 in decimal. As the **MOVSD** instruction is copying the bytes in the chunk of 4 bytes that is, DWORD from **ESI** to So, the total number of bytes that are copied using the **MOVSD.REP** operation is 14 multiplied by 4 bytes (1 DWORD has 4 bytes) = 56 bytes. An additional null byte for termination is copied using the **MOVSB** operation, resulting in a total of 57 bytes that are copied from **ESI** to This is the same as the length of this URL string, which is 57 bytes.

During the start of the function, we created room for the local variables by subtracting 0x50 (80 bytes in decimal) from After copying the URL string in this 80-bytes room, the remaining 23 bytes (80 bytes minus 57 bytes) of the memory location on the stack will be cleared using **XOR** and several **MOV** instructions. **EAX** is cleared using the XOR operation and with the remaining **MOV** instructions, the 22 bytes of the memory location will be cleared as shown in the following stack state screenshot. The yellow-marked cells are the URL string data copied from **ESI** to **EDI** on the stack. The blank cells marked with 0x00 show the result of the XOR and **MOV XOR** operations. The remaining 1 byte will be cleared in the subsequent **MOV** instruction.

00000070	EDI			
00000074	ESI			
00000078	http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com			
0000007C				
00000080				
00000084				
00000088				
0000008C				
00000090				
00000094				
00000098				
0000009C				
000000A0				
000000A4				
000000A8				
000000AC				
000000B0	00	00	00	00
000000B4	00	00	00	00
000000B8	00	00	00	00
000000BC	00	00	00	00
000000C0	00	00	00	00
000000C4		00	00	00
000000C8	XX	XX	XX	XX

EAX 00000000
EDI 00000078

Figure 16.21: *URL copied from ESI to EDI*

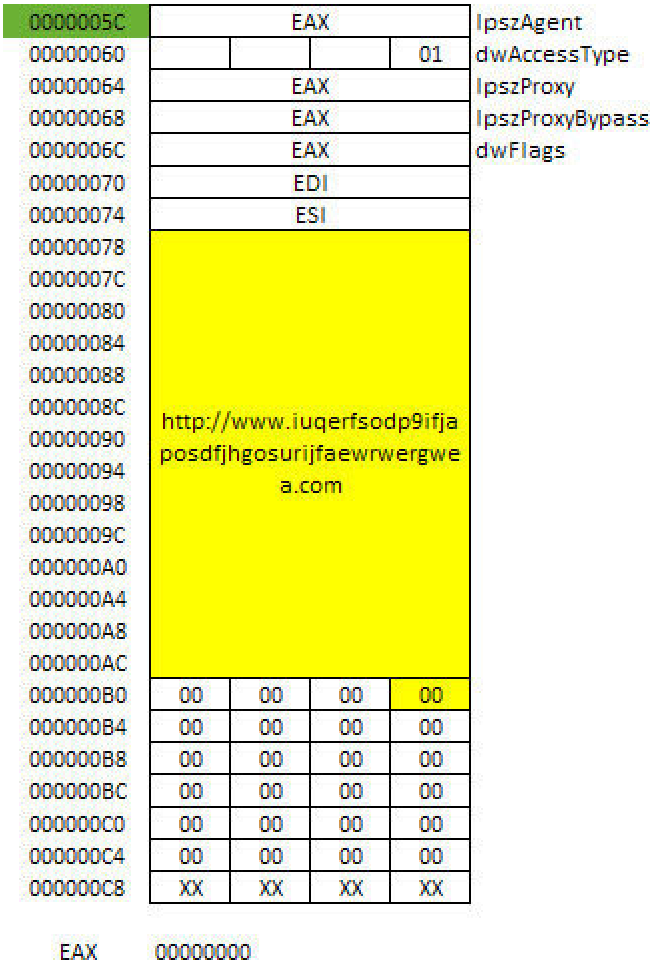
▼Line 34-40

```
PUSH      EAX   ; dwFlags
PUSH      EAX   ; lpszProxyBypass
PUSH      EAX   ; lpszProxy
PUSH      0x1   ; dwAccessType
PUSH      EAX   ; lpszAgent
MOV       byte ptr [ESP + local_1],AL
CALL      dword ptr [->WININET.DLL::InternetOpenA]
```

The **InternetOpenA** function is called, as we can see in this set of assembly listing. As per the Microsoft documentation, the **InternetOpenA** syntax is defined as follows:

```
void InternetOpenA(
LPCSTR lpszAgent,
DWORD dwAccessType,
LPCSTR lpszProxy,
LPCSTR lpszProxyBypass,
DWORD dwFlags
);
```

All the parameters to **InternetOpenA** are pushed on the stack using the **PUSH** instructions one by one. This is marked in the following stack state for a better understanding. The MOV instruction is clearing the remaining 1 byte location on the stack as shown in the following stack state:



0000005C	EAX				lpszAgent
00000060				01	dwAccessType
00000064	EAX				lpszProxy
00000068	EAX				lpszProxyBypass
0000006C	EAX				dwFlags
00000070	EDI				
00000074	ESI				
00000078	http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com				
0000007C					
00000080					
00000084					
00000088					
0000008C					
00000090					
00000094					
00000098					
0000009C					
000000A0					
000000A4					
000000A8					
000000AC					
000000B0	00	00	00	00	
000000B4	00	00	00	00	
000000B8	00	00	00	00	
000000BC	00	00	00	00	
000000C0	00	00	00	00	
000000C4	00	00	00	00	
000000C8	XX	XX	XX	XX	

EAX 00000000

Figure 16.22: *InternetOpenA call*

▼Line 42-51

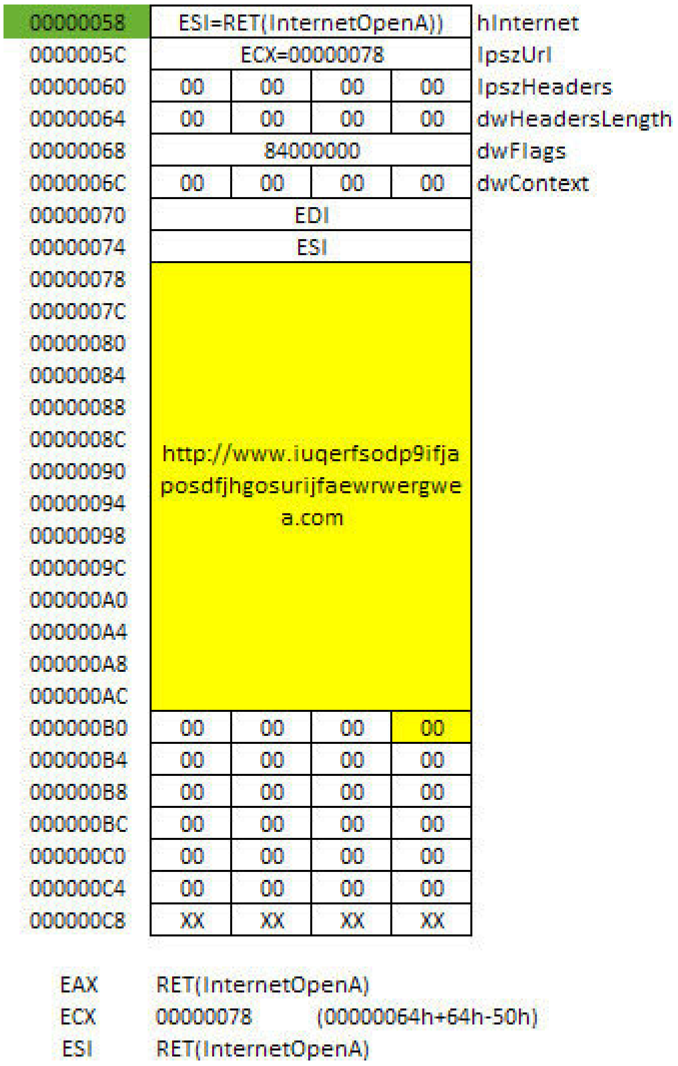
```
PUSH  0x0            ;  dwContext


PUSH  0x84000000          ;  dwFlags
PUSH  0x0            ;  dwHeadersLength
LEA  ECX=>local_50,[ESP + 0x14]
MOV  ESI,EAX
PUSH  0x0            ;  lpszHeaders
PUSH  ECX             ;  lpszUrl
PUSH  ESI            ;  hInternet
```

```
CALL dword ptr [->WININET.DLL::InternetOpenUrlA]
```

The **InternetOpenA** function in **wininet.dll** (32-bit) returns with the **RETN 0x14** instructions, which clears off the pushed parameters of the **InternetOpenA** function on the stack upon Now, with the preceding instructions, the stack is again populated with parameters to the **InternetOpenUrlA** function. According to the Microsoft documentation, the syntax of **InternetOpenUrlA** function is:

```
void InternetOpenUrlA(
HINTERNET hInternet,
LPCSTR      lpszUrl,
LPCSTR      lpszHeaders,
DWORD       dwHeadersLength,
DWORD       dwFlags,
DWORD_PTR dwContext
);
```

The return value of the **InternetOpenA** function is stored in the **EAX** register and is pushed onto the stack as a parameter to the **InternetOpenUrlA** function with **MOV ESI,** and **PUSH ESI** instructions. The **LEA** instruction loads the address of the URL string at **[ESP + 0x14]** into which is later pushed onto the stack as a parameter to the **InternetOpenUrlA** function. The stack state before the call to the **InternetOpenUrlA** function will be as follows:



00000058	ESI=RET(InternetOpenA))				hInternet
0000005C	ECX=00000078				lpszUrl
00000060	00	00	00	00	lpszHeaders
00000064	00	00	00	00	dwHeadersLength
00000068	84000000				dwFlags
0000006C	00	00	00	00	dwContext
00000070	EDI				
00000074	ESI				
00000078	http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com				
0000007C					
00000080					
00000084					
00000088					
0000008C					
00000090					
00000094					
00000098					
0000009C					
000000A0					
000000A4					
000000A8					
000000AC					
000000B0	00	00	00	00	
000000B4	00	00	00	00	
000000B8	00	00	00	00	
000000BC	00	00	00	00	
000000C0	00	00	00	00	
000000C4	00	00	00	00	
000000C8	XX	XX	XX	XX	

EAX	RET(InternetOpenA)	
ECX	00000078	(00000064h+64h-50h)
ESI	RET(InternetOpenA)	

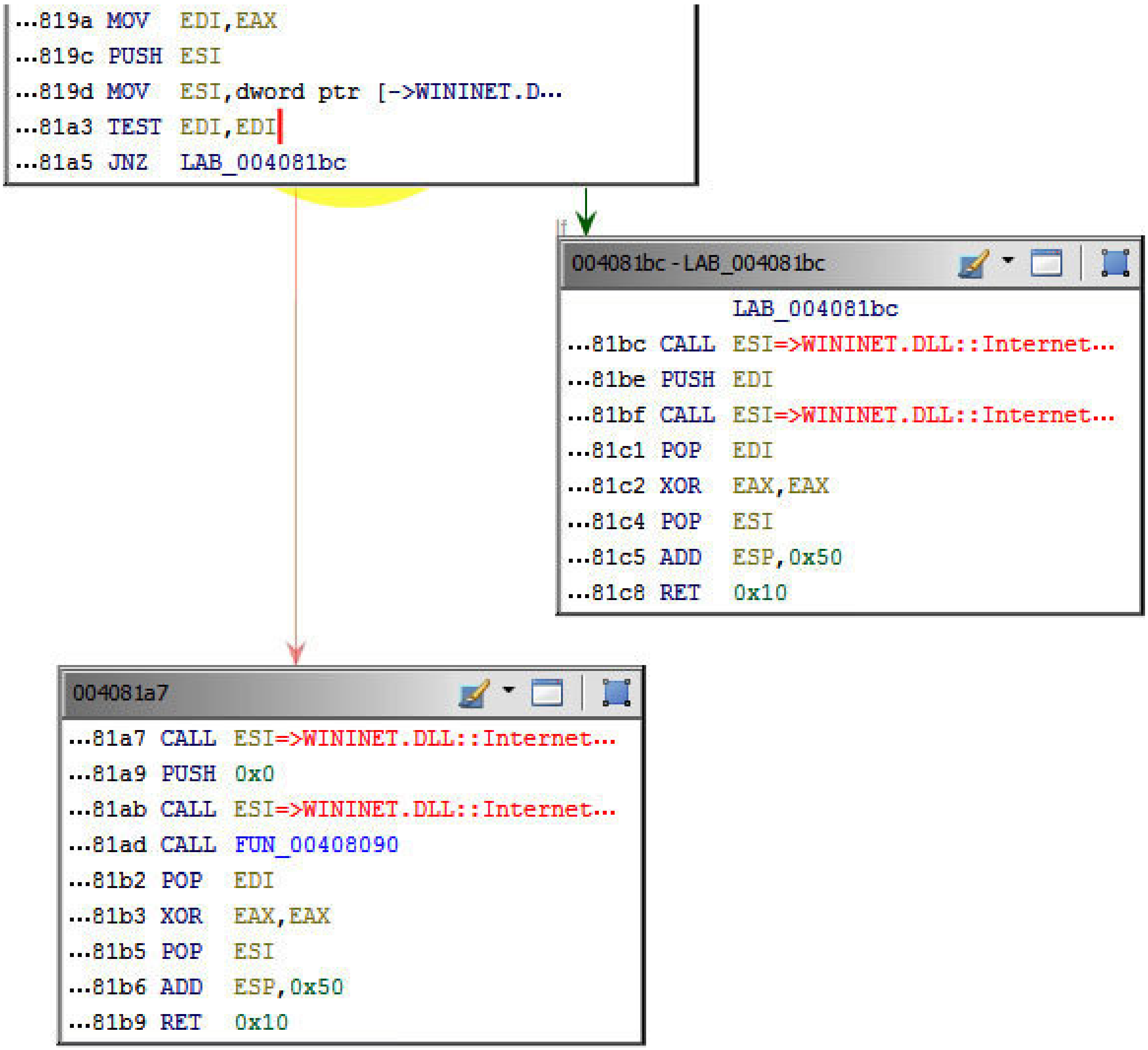
Figure 16.23: Before call to InternetOpenUrlA

▼Line 53-77

```
MOV     EDI,EAX
PUSH    ESI
MOV     ESI,dword ptr [->WININET.DLL::InternetCloseHan = 0000a7b2
TEST    EDI,EDI
JNZ     LAB_004081bc
CALL    ESI=>WININET.DLL::InternetCloseHandle
```

```
PUSH     0x0
CALL     ESI=>WININET.DLL::InternetCloseHandle
CALL     FUN_00408090
POP      EDI
XOR      EAX,EAX
POP      ESI
ADD      ESP,0x50
RET      0x10
LAB_004081bc     XREF[1]:       004081a5(j)
CALL     ESI=>WININET.DLL::InternetCloseHandle
PUSH     EDI
CALL     ESI=>WININET.DLL::InternetCloseHandle
POP      EDI
XOR      EAX,EAX
POP      ESI
ADD      ESP,0x50
RET      0x10
```

The **InternetOpenUrlA** function in **wininet.dll** (32-bit) returns with the **RETN 0x18** instruction, which clears off the pushed parameters of the **InternetOpenUrlA** function on the stack. In this set of assembly instructions, we see something really interesting. The return value of the function in **EAX** is moved to As per the documentation of Microsoft for **InternetOpenUrlA** function, if the return value is NULL, it means that the connection to the URL failed. If the return value is a valid handle to the URL, it means that the connection to the URL is successfully established. This condition is checked with the **TEST** instruction in the preceding assembly listing. This brings us to some interesting conclusions from the **Function**



```
...819a MOV   EDI,EAX
...819c PUSH  ESI
...819d MOV   ESI,dword ptr [->WININET.D...
...81a3 TEST  EDI,EDI
...81a5 JNZ   LAB_004081bc
```

```
            LAB_004081bc
...81bc CALL  ESI=>WININET.DLL::Internet...
...81be PUSH  EDI
...81bf CALL  ESI=>WININET.DLL::Internet...
...81c1 POP   EDI
...81c2 XOR   EAX,EAX
...81c4 POP   ESI
...81c5 ADD   ESP,0x50
...81c8 RET   0x10
```

```
...81a7 CALL  ESI=>WININET.DLL::Internet...
...81a9 PUSH  0x0
...81ab CALL  ESI=>WININET.DLL::Internet...
...81ad CALL  FUN_00408090
...81b2 POP   EDI
...81b3 XOR   EAX,EAX
...81b5 POP   ESI
...81b6 ADD   ESP,0x50
...81b9 RET   0x10
```

Figure 16.24: *TEST condition*

```
TEST    EDI,EDI
JNZ     LAB_004081bc
```

The **TEST** instruction performs the logical AND between **EDI** and This instruction is used to check the registers for zero without altering its value. If **EDI** is equal to 0, set ZF to 1. If the ZF is set to 1, then no action will be taken and the next instruction following it will be executed. This is a conditional jump instruction which jumps to the **LAB_004081bc** location if the zero flag (ZF) is set to 0.

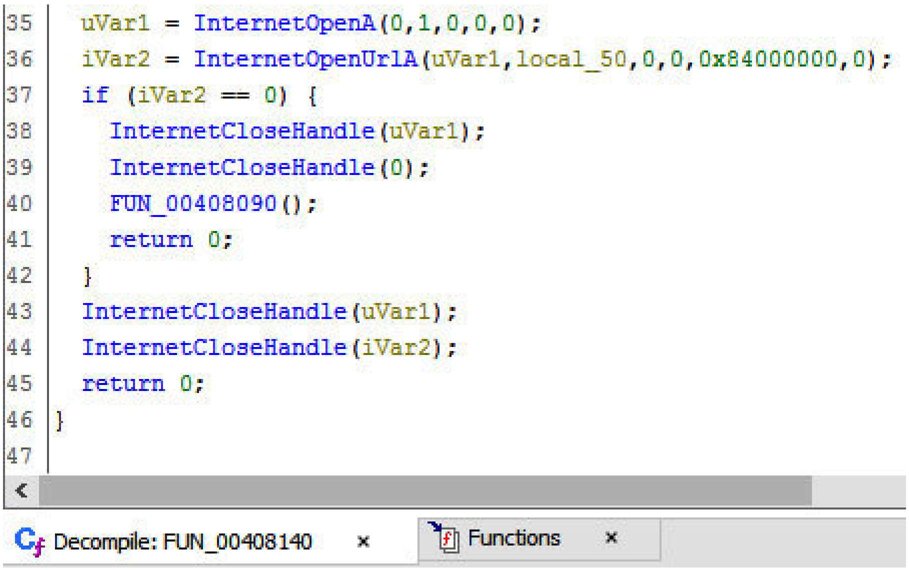
The **TEST** instruction checks the return value of the **InternetOpenUrlA** function. If the request to http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com fails, the **InternetOpenUrlA** function returns the null handle which then closes the handle to calls **FUN_00408090** function. This is where the ransomware does all its working.

But if the request to http://www.iuqerfsodp9ifjaposdfjhgosurijfaewrwergwea.com passes, it simply closes the handle and then quits the Wannacry program to make the ransomware ineffective.

According to Information Security Newspaper, link as mentioned below:

Marcus Hutchins was the hero behind the switch of Wannacry. So once he registered this domain, Wannacry was shut down.

So we saw by understanding the assembly listing, we are able to decode the functioning of the malware and find a way to break the Wannacry ransomware. Now, we can validate our understanding of the code flow with the decompiled code generated by Ghidra as follows:



```
  uVar1 = InternetOpenA(0,1,0,0,0);
  iVar2 = InternetOpenUrlA(uVar1,local_50,0,0,0x84000000,0);
  if (iVar2 == 0) {
    InternetCloseHandle(uVar1);
    InternetCloseHandle(0);
    FUN_00408090();
    return 0;
  }
  InternetCloseHandle(uVar1);
  InternetCloseHandle(iVar2);
  return 0;
}
```

Figure 16.25: *Decompiled code*

Whatever we have analyzed from the step-by-step assembly instructions is in line with the decompiled code generated from Ghidra. If the **InternetOpenUrlA** function returns the null handle, the ransomware does all its working with the **FUN_00408090** function. Or else, it simply closes the handle and then quits the Wannacry program.

Conclusion

In this chapter, we covered the steps to install a reverse engineering framework called Using Ghidra, we analyzed the Wannacry malware to disassemble the malware. With what we learned from the earlier chapters, we analyzed the code flow with the help of visualizing the stack state during the instruction flow. This helped us understood the code flow and find the kill switch of Wannacry.

CHAPTER 17

Generate Pseudo Code From Binary File

In the earlier chapter, we covered the different patterns of assembly code for various C/C++ applications and some real-life examples. The job of a reverse engineer is to get the copy of the binary for reverse engineering and understanding the code flow. This binary can be of any software or application or it can be malware. All the modern malwares are coded with some URL to communicate with the server to upload data or to send feeds of the malware activities. So, these URL are not hard coded in the plain text format but are encrypted or encoded inside the code.

In this chapter, we will generate the pseudo code from the binary file to crack such encrypted URL. The encryption logic used in such cases can be standard ones or custom made, all depending on the malware writer. We will use which is an open-source interface to the Radare2 reverse engineering framework. Radare2 is used for static and dynamic analysis of binary formats on different platforms and architectures. Radare2 is for those who love to work on command line interface. Cutter is the graphical user interface of Radare2.

Structure

In this chapter, we will cover the following topics:

Installation of the reverse engineering framework called Cutter

Binary analysis using Cutter

Decrypting a hidden URL

Objective

The objective of this chapter is to understand the steps involved in installing the reverse engineering framework called Cutter. After installing, we will analyze the binary to generate the pseudo code to extract the encryption key. This encryption key is used to encrypt the URL in the binary. We will also see how this encryption of URL is used to escape from reverse engineering.

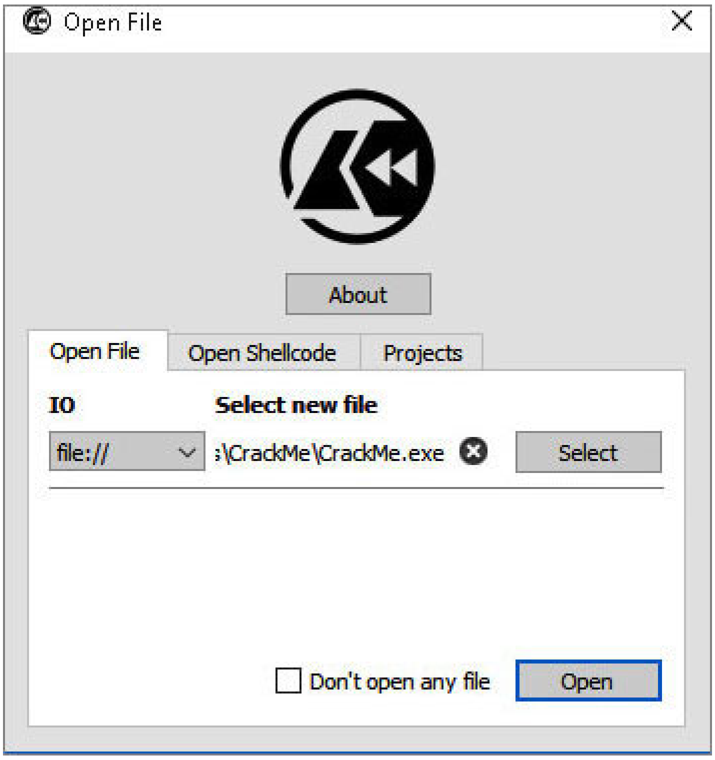
Cutter Installation

Cutter can be downloaded from the GitHub repository. It can be compiled from the source code or can be downloaded as a binary. It is available for different platforms (Windows, Linux, macOS). We will use the Windows 10 64-bit version to carry out our Cutter installation. Following is the step-by-step procedure to install Cutter:

Download and extract the zip file from the GitHub repository:

https://github.com/rizinorg/cutter/releases/download/v1.12.0/Cutter-v1.12.0-x64.Windows.zip

In the extracted folder, run **cutter.exe** to open the first screen of Cutter. This is the screen where we have to select and open the binary file to analyze.



IO: file://

Select new file: :\CrackMe\CrackMe.exe

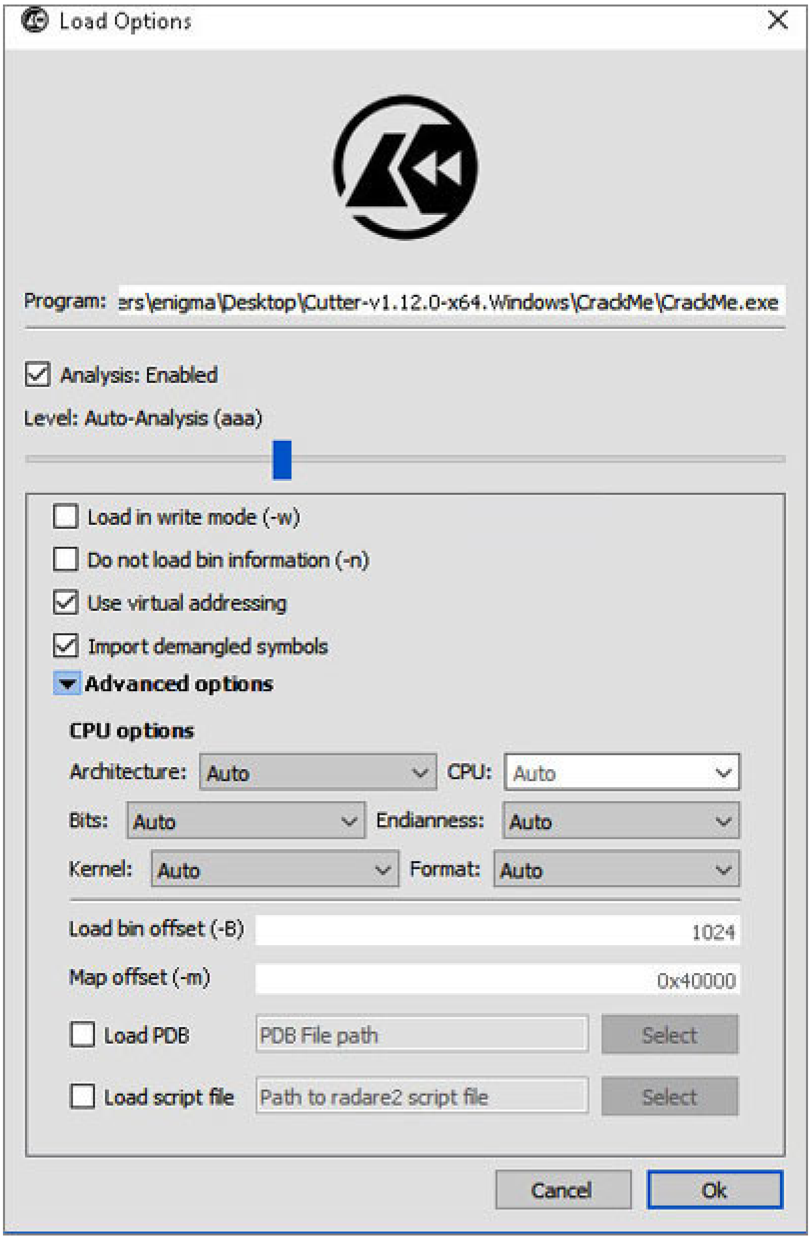
☐ Don't open any file

Figure 17.1: Cutter first screen

Download this **CrackMe.exe** from the following link:

https://github.com/bpbpublications/Implementing-Reverse-Engineering

Once the binary file to be analyzed is selected, click **Open** to get the cutter **Load** We are keeping everything default. If you have a symbol file, we can select the **Load PDB** option.



Program: ers\enigma\Desktop\Cutter-v1.12.0-x64.Windows\CrackMe\CrackMe.exe

☑ Analysis: Enabled

Level: Auto-Analysis (aaa)

☐ Load in write mode (-w)

☐ Do not load bin information (-n)

☑ Use virtual addressing

☑ Import demangled symbols

Advanced options

CPU options

Architecture: Auto

CPU: Auto

Bits: Auto

Endianness: Auto

Kernel: Auto

Format: Auto

Load bin offset (-B) 1024

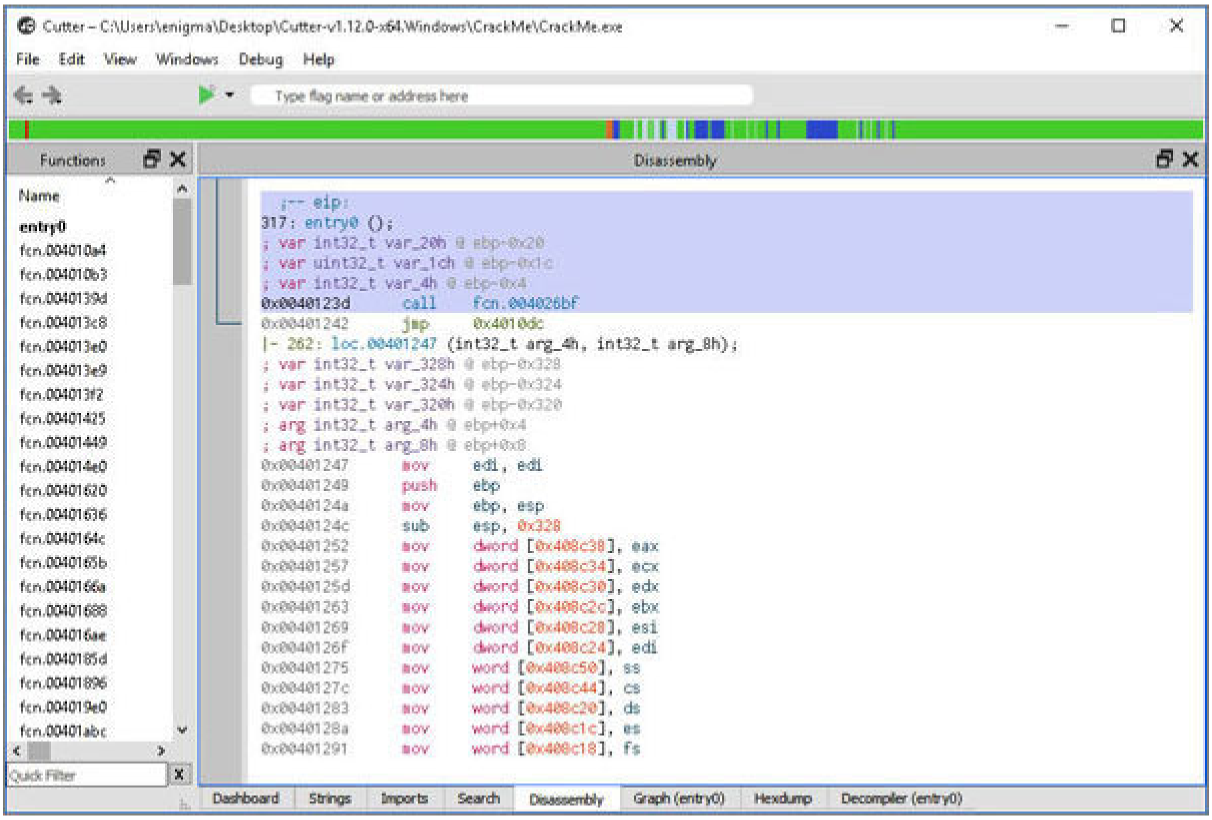
Map offset (-m) 0x40000

☐ Load PDB — PDB File path — Select

☐ Load script file — Path to radare2 script file — Select

Figure 17.2: *Cutter Load Options*

On clicking you will be presented with the disassembly view of the binary. Along with this, we get to see several tabs at the bottom of window.



```
  ;-- eip:
317: entry0 ();
; var int32_t var_20h @ ebp-0x20
; var uint32_t var_1ch @ ebp-0x1c
; var int32_t var_4h @ ebp-0x4
0x0040123d      call    fcn.004026bf
0x00401242      jmp     0x4010dc
|- 262: loc.00401247 (int32_t arg_4h, int32_t arg_8h);
; var int32_t var_328h @ ebp-0x328
; var int32_t var_324h @ ebp-0x324
; var int32_t var_320h @ ebp-0x320
; arg int32_t arg_4h @ ebp+0x4
; arg int32_t arg_8h @ ebp+0x8
0x00401247      mov     edi, edi
0x00401249      push    ebp
0x0040124a      mov     ebp, esp
0x0040124c      sub     esp, 0x328
0x00401252      mov     dword [0x408c38], eax
0x00401257      mov     dword [0x408c34], ecx
0x0040125d      mov     dword [0x408c30], edx
0x00401263      mov     dword [0x408c2c], ebx
0x00401269      mov     dword [0x408c28], esi
0x0040126f      mov     dword [0x408c24], edi
0x00401275      mov     word [0x408c50], ss
0x0040127c      mov     word [0x408c44], cs
0x00401283      mov     word [0x408c20], ds
0x0040128a      mov     word [0x408c1c], es
0x00401291      mov     word [0x408c18], fs
```

Figure 17.3: *Binary disassemble view*

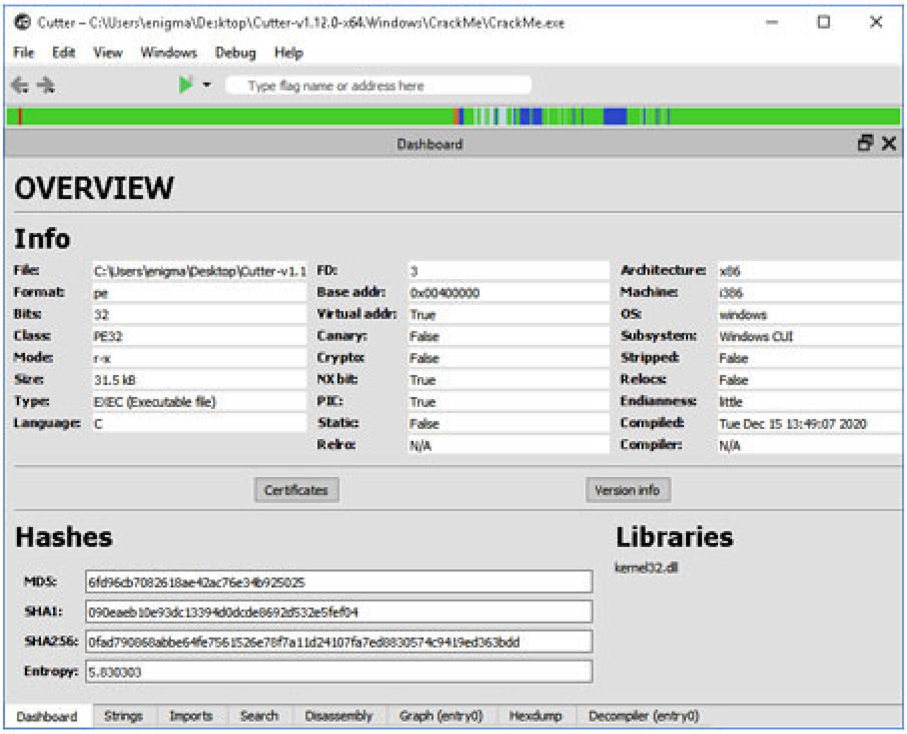
With this, we have successfully started Cutter for further analysis. But before we begin our analysis, we will walk through the different functionalities and terminologies in the Cutter graphical user interface.

Binary Analysis Using Cutter

The Cutter user interface has many tabs. We will walk through each tab and explain its relevance in analyzing any binary file.

Dashboard

The **Dashboard** tab shows the binary path, architecture, endianness, and many other details. Our binary file is a **portable executable** format file and compiled for 32-bit architecture as follows. The values present in the **Hashes** section is used to ensure that the file is not corrupted or altered by unauthorized users. Some antivirus companies use these values to determine if a file is malicious or not. They maintain the hash database of known malwares and upon scanning, they evaluate the file hash to compare with that in the database. If the hash values match the values in the database, the file infection is triggered.



File: C:\Users\enigma\Desktop\Cutter-v1.1
Format: pe
Bits: 32
Class: PE32
Mode: r-x
Size: 31.5 kB
Type: EXEC (Executable file)
Language: C
FD: 3
Base addr: 0x00400000
Virtual addr: True
Canary: False
Crypto: False
NX bit: True
PIC: True
Static: False
Relro: N/A
Architecture: x86
Machine: i386
OS: windows
Subsystem: Windows CUI
Stripped: False
Relocs: False
Endianness: little
Compiled: Tue Dec 15 13:49:07 2020
Compiler: N/A

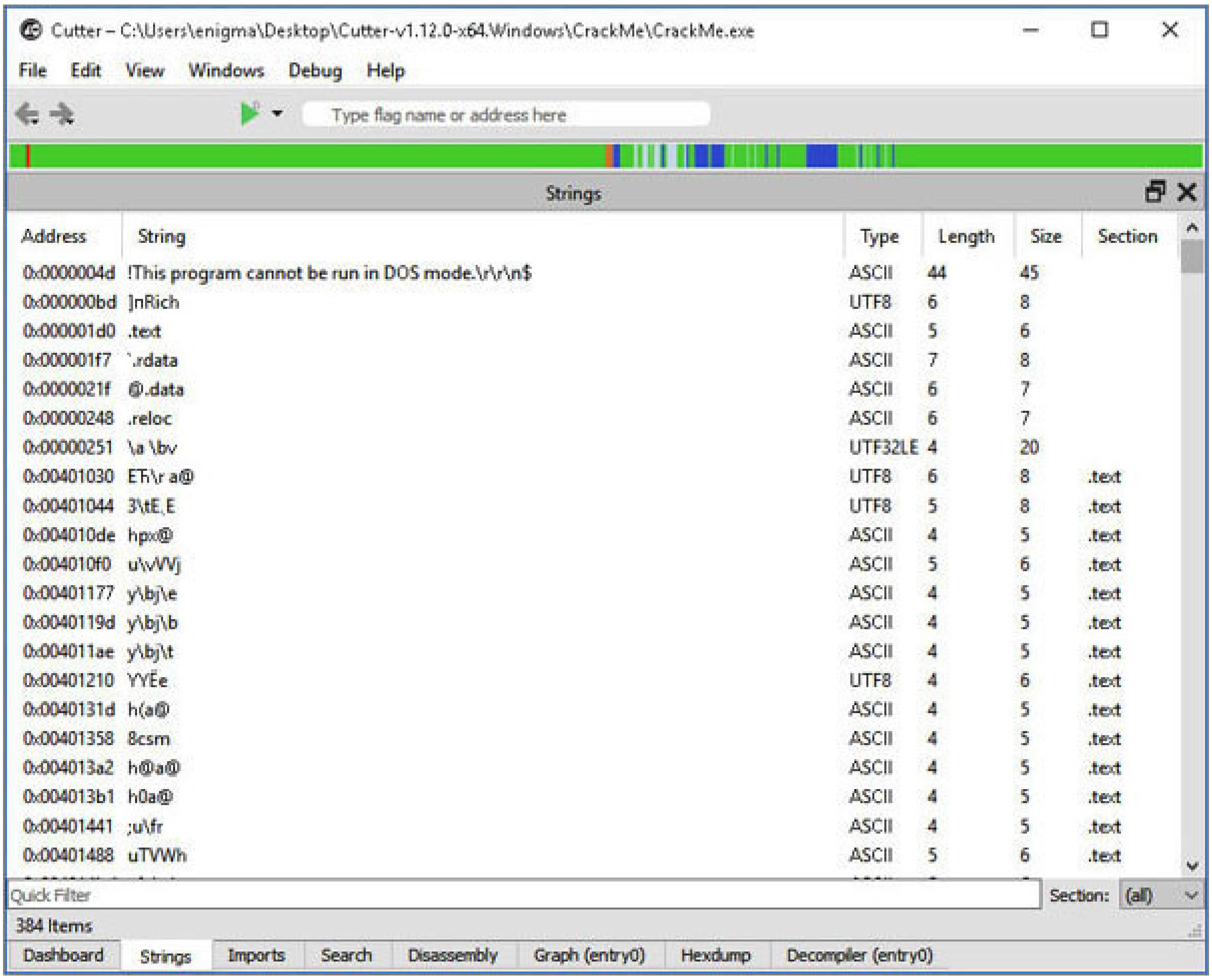
MD5: 6fd96cb7082618ae42ac76e34b925025
SHA1: 090eaeb10e93dc13394d0dcde8692d532e5fef04
SHA256: 0fad790868abbe64fe7561526e78f7a11d24107fa7ed8830574c9419ed363bdd
Entropy: 5.830303

kernel32.dll

Figure 17.4: Dashboard view

Strings

The **Strings** tab shows the text string found in the binary. It is the first level of analysis for any binary as sometimes it gives a lot of clues about the binary internals. Imagine if some IP address or URL is used in a plain text format in a malware code, then that IP or URL will be reflected in the **Strings** tab.

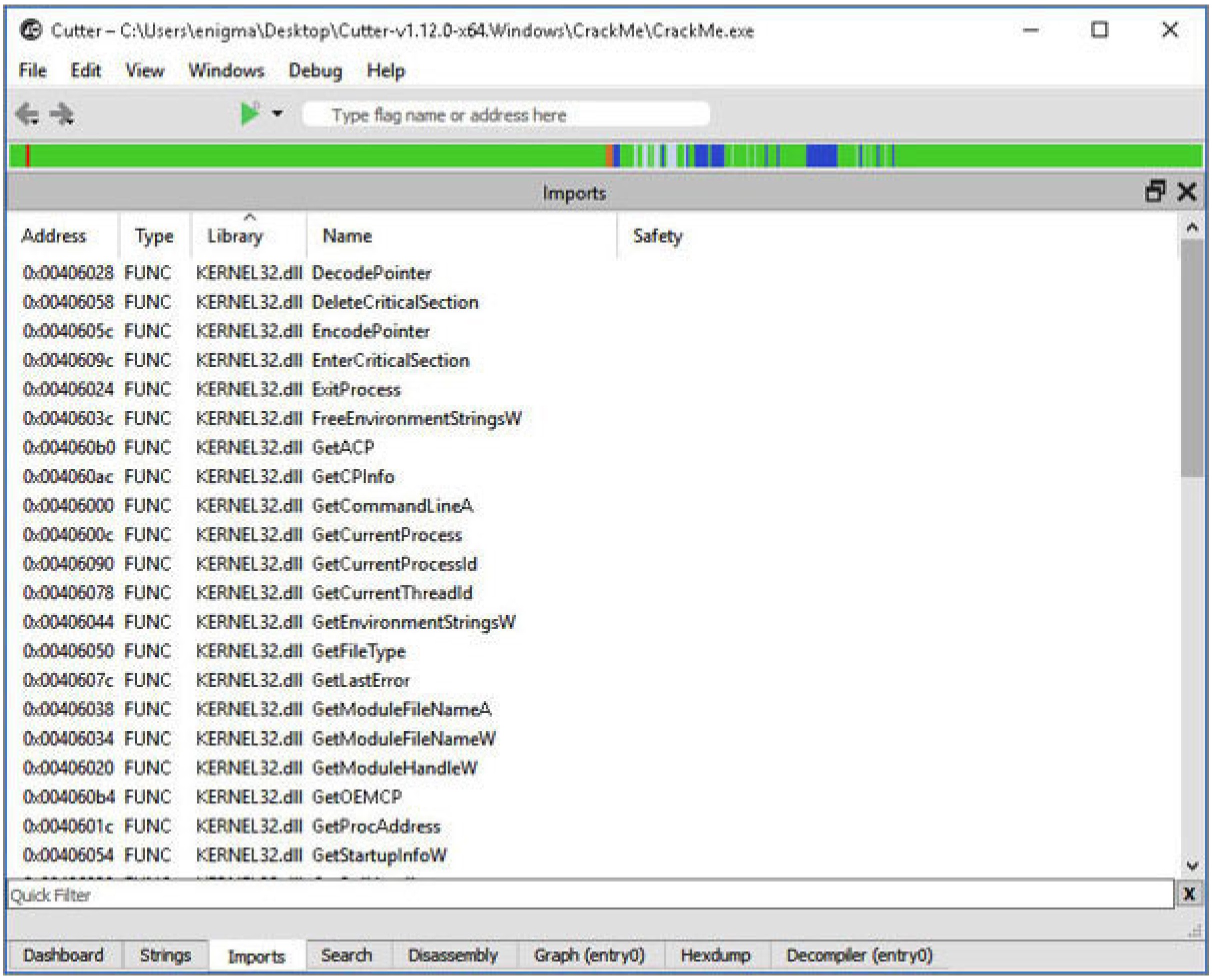


Address	String	Type	Length	Size	Section
0x0000004d	!This program cannot be run in DOS mode.\r\r\n$	ASCII	44	45	
0x000000bd	]nRich	UTF8	6	8	
0x000001d0	.text	ASCII	5	6	
0x000001f7	`.rdata	ASCII	7	8	
0x0000021f	@.data	ASCII	6	7	
0x00000248	.reloc	ASCII	6	7	
0x00000251	\a \bv	UTF32LE	4	20	
0x00401030	Eћ\r a@	UTF8	6	8	.text
0x00401044	3\tE,E	UTF8	5	8	.text
0x004010de	hpx@	ASCII	4	5	.text
0x004010f0	u\vVVj	ASCII	5	6	.text
0x00401177	y\bj\e	ASCII	4	5	.text
0x0040119d	y\bj\b	ASCII	4	5	.text
0x004011ae	y\bj\t	ASCII	4	5	.text
0x00401210	YYËe	UTF8	4	6	.text
0x0040131d	h(a@	ASCII	4	5	.text
0x00401358	8csm	ASCII	4	5	.text
0x004013a2	h@a@	ASCII	4	5	.text
0x004013b1	h0a@	ASCII	4	5	.text
0x00401441	;u\fr	ASCII	4	5	.text
0x00401488	uTVWh	ASCII	5	6	.text

Figure 17.5: Strings view

Imports

This tab shows the libraries imported by the binary. If the binary is using the internet to connect to some service, then it can be figured out with the use of the relevant functions used to connect to the internet.

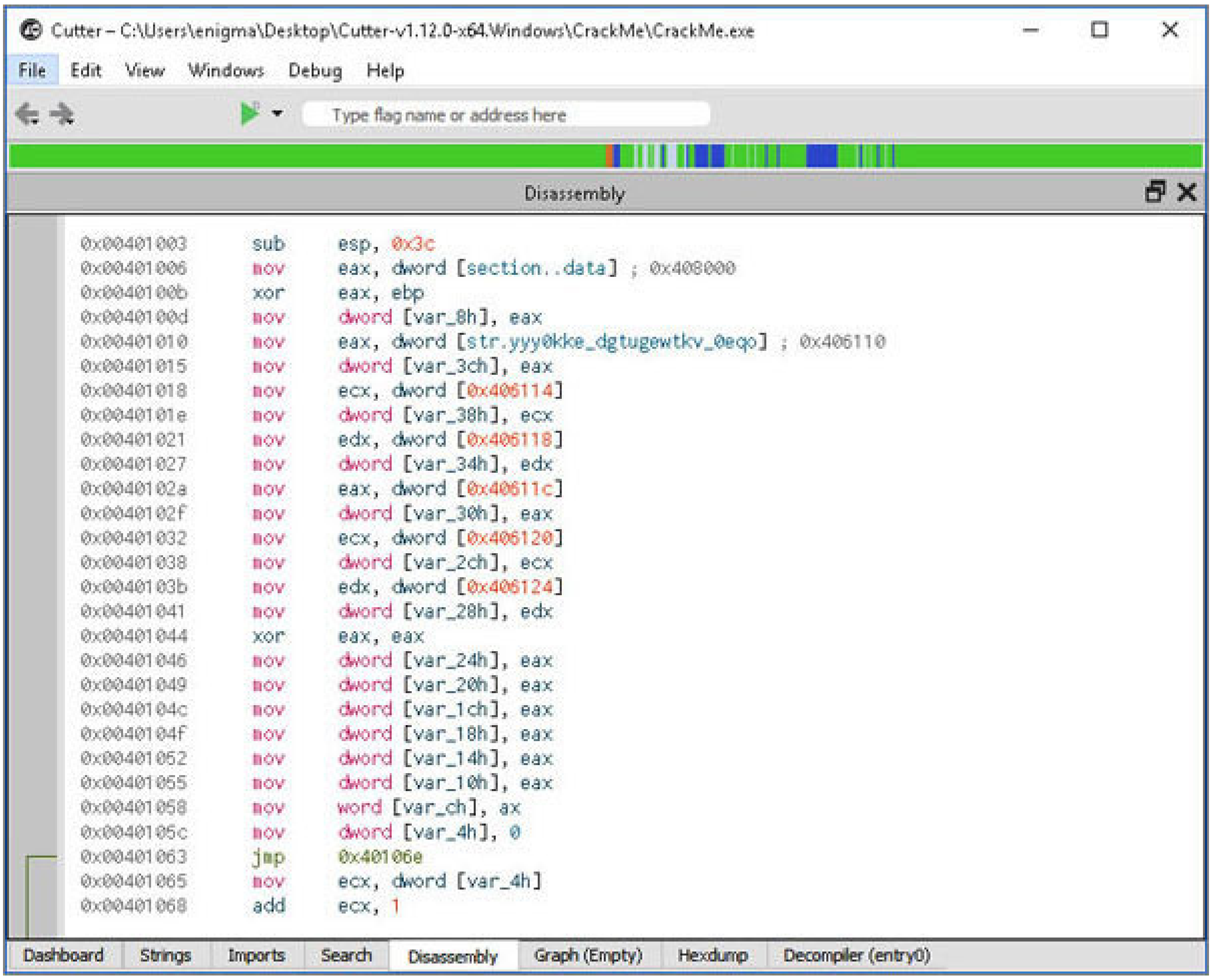


Address	Type	Library	Name	Safety
0x00406028	FUNC	KERNEL32.dll	DecodePointer	
0x00406058	FUNC	KERNEL32.dll	DeleteCriticalSection	
0x0040605c	FUNC	KERNEL32.dll	EncodePointer	
0x0040609c	FUNC	KERNEL32.dll	EnterCriticalSection	
0x00406024	FUNC	KERNEL32.dll	ExitProcess	
0x0040603c	FUNC	KERNEL32.dll	FreeEnvironmentStringsW	
0x004060b0	FUNC	KERNEL32.dll	GetACP	
0x004060ac	FUNC	KERNEL32.dll	GetCPInfo	
0x00406000	FUNC	KERNEL32.dll	GetCommandLineA	
0x0040600c	FUNC	KERNEL32.dll	GetCurrentProcess	
0x00406090	FUNC	KERNEL32.dll	GetCurrentProcessId	
0x00406078	FUNC	KERNEL32.dll	GetCurrentThreadId	
0x00406044	FUNC	KERNEL32.dll	GetEnvironmentStringsW	
0x00406050	FUNC	KERNEL32.dll	GetFileType	
0x0040607c	FUNC	KERNEL32.dll	GetLastError	
0x00406038	FUNC	KERNEL32.dll	GetModuleFileNameA	
0x00406034	FUNC	KERNEL32.dll	GetModuleFileNameW	
0x00406020	FUNC	KERNEL32.dll	GetModuleHandleW	
0x004060b4	FUNC	KERNEL32.dll	GetOEMCP	
0x0040601c	FUNC	KERNEL32.dll	GetProcAddress	
0x00406054	FUNC	KERNEL32.dll	GetStartupInfoW	

Figure 17.6: *Imports view*

Disassembly

This tab shows the disassembled view of the binary. Whatever concepts and instructions we have learned in the earlier chapters will help us understand this flow. It might be overwhelming for you in the first view but don't worry, we will understand this in a step-by-step manner.



```
0x00401003      sub     esp, 0x3c
0x00401006      mov     eax, dword [section..data] ; 0x408000
0x0040100b      xor     eax, ebp
0x0040100d      mov     dword [var_8h], eax
0x00401010      mov     eax, dword [str.yyy0kke_dgtugewtkv_0eqo] ; 0x406110
0x00401015      mov     dword [var_3ch], eax
0x00401018      mov     ecx, dword [0x406114]
0x0040101e      mov     dword [var_38h], ecx
0x00401021      mov     edx, dword [0x406118]
0x00401027      mov     dword [var_34h], edx
0x0040102a      mov     eax, dword [0x40611c]
0x0040102f      mov     dword [var_30h], eax
0x00401032      mov     ecx, dword [0x406120]
0x00401038      mov     dword [var_2ch], ecx
0x0040103b      mov     edx, dword [0x406124]
0x00401041      mov     dword [var_28h], edx
0x00401044      xor     eax, eax
0x00401046      mov     dword [var_24h], eax
0x00401049      mov     dword [var_20h], eax
0x0040104c      mov     dword [var_1ch], eax
0x0040104f      mov     dword [var_18h], eax
0x00401052      mov     dword [var_14h], eax
0x00401055      mov     dword [var_10h], eax
0x00401058      mov     word [var_ch], ax
0x0040105c      mov     dword [var_4h], 0
0x00401063      jmp     0x40106e
0x00401065      mov     ecx, dword [var_4h]
0x00401068      add     ecx, 1
```

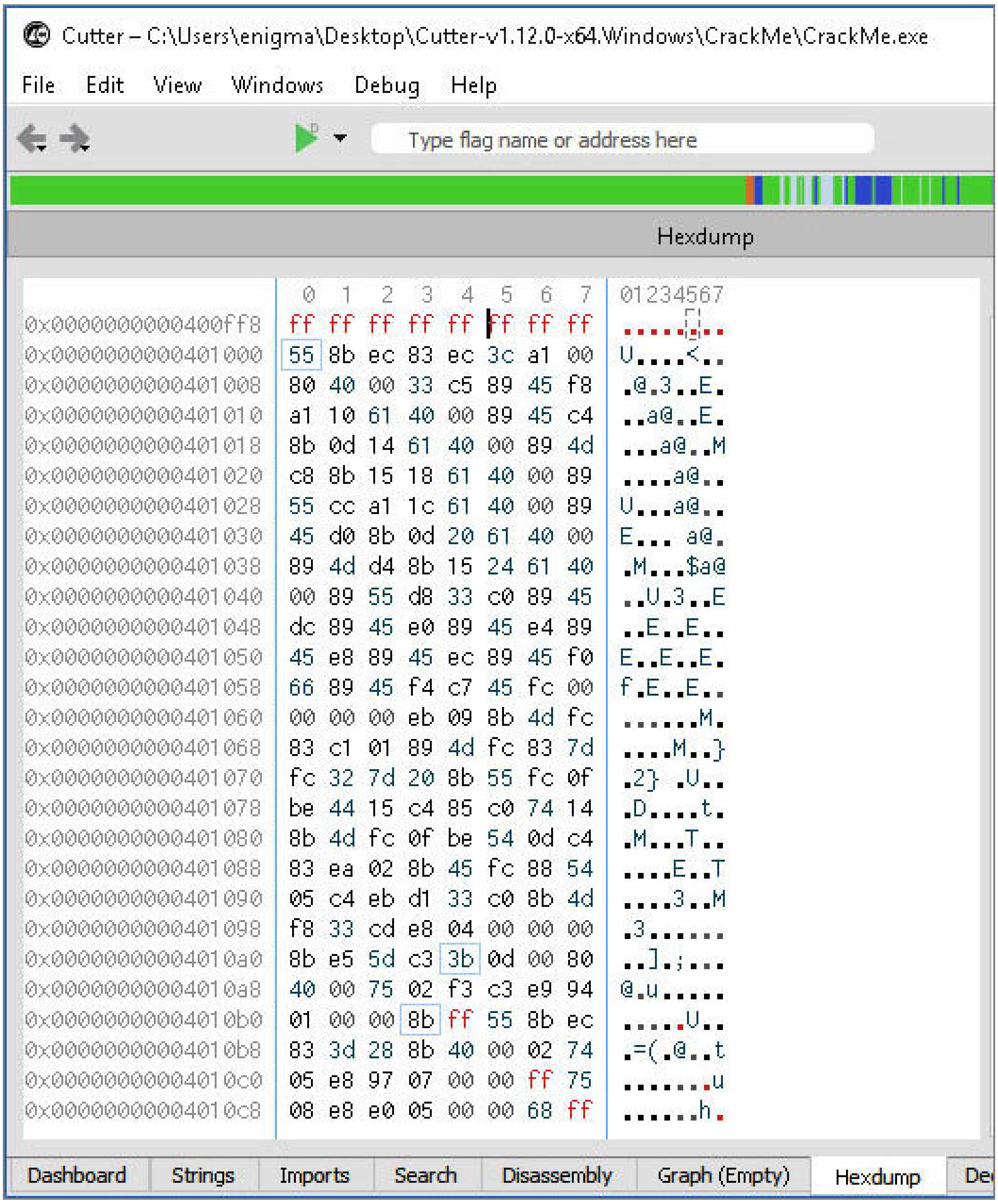
__Figure 17.7:__ Disassemble view

Graph

As seen in the earlier figure, this tab is named as Graph (Empty). Empty is suffixed as we have not selected any function. On this tab selection, we get an empty screen with the same message to select a function.

Hexdump

This shows the hex dump of the binary file and is represented by 3 columns. The first column shows the offset, the second column shows the hexadecimal output, and the third column is the representation of data in the ASCII format.

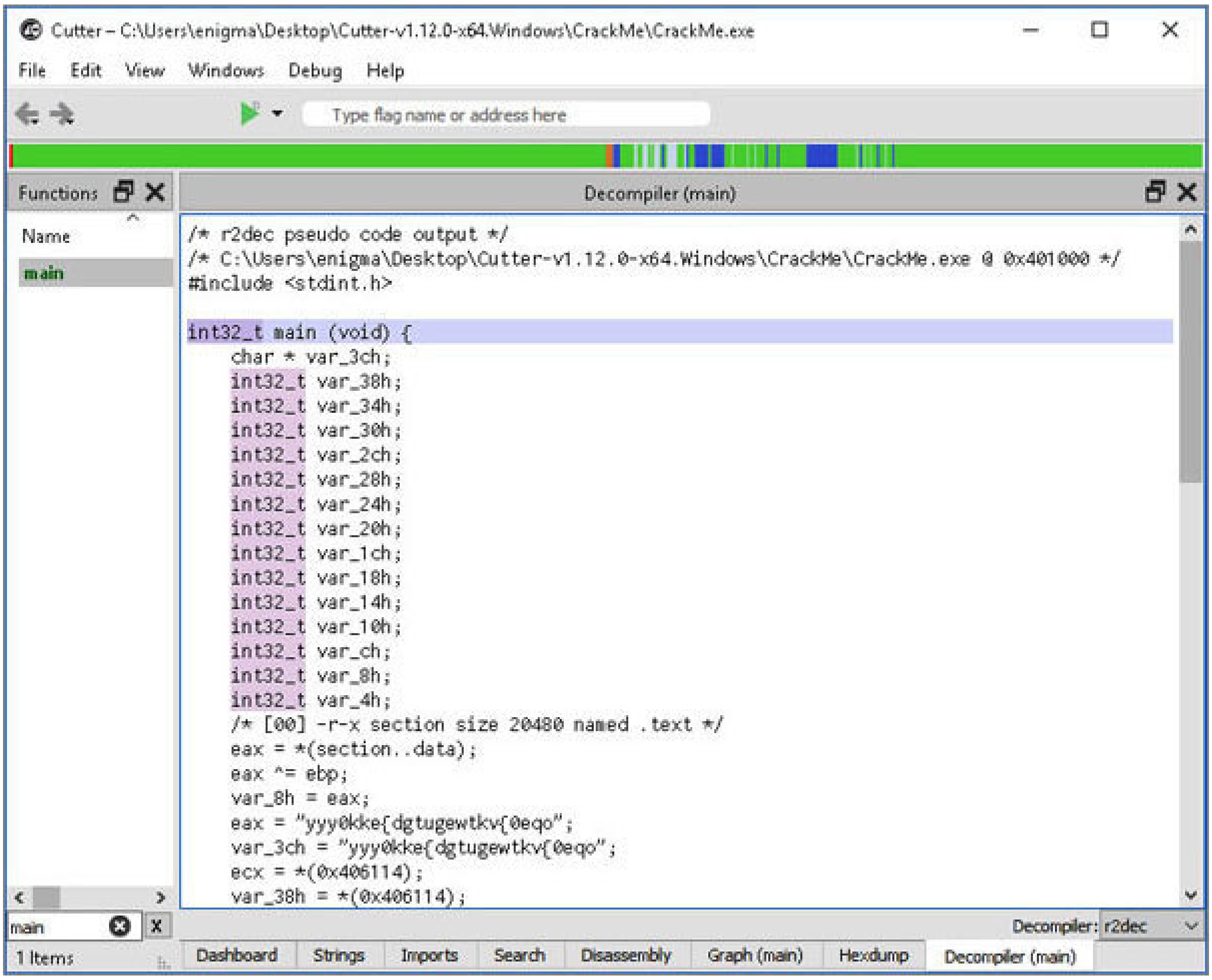


```
                      0  1  2  3  4  5  6  7   01234567
0x0000000000400ff8   ff ff ff ff ff ff ff ff   ........
0x0000000000401000   55 8b ec 83 ec 3c a1 00   U....<..
0x0000000000401008   80 40 00 33 c5 89 45 f8   .@.3..E.
0x0000000000401010   a1 10 61 40 00 89 45 c4   ..a@..E.
0x0000000000401018   8b 0d 14 61 40 00 89 4d   ...a@..M
0x0000000000401020   c8 8b 15 18 61 40 00 89   ....a@..
0x0000000000401028   55 cc a1 1c 61 40 00 89   U...a@..
0x0000000000401030   45 d0 8b 0d 20 61 40 00   E... a@.
0x0000000000401038   89 4d d4 8b 15 24 61 40   .M...$a@
0x0000000000401040   00 89 55 d8 33 c0 89 45   ..U.3..E
0x0000000000401048   dc 89 45 e0 89 45 e4 89   ..E..E..
0x0000000000401050   45 e8 89 45 ec 89 45 f0   E..E..E.
0x0000000000401058   66 89 45 f4 c7 45 fc 00   f.E..E..
0x0000000000401060   00 00 00 eb 09 8b 4d fc   ......M.
0x0000000000401068   83 c1 01 89 4d fc 83 7d   ....M..}
0x0000000000401070   fc 32 7d 20 8b 55 fc 0f   .2} .U..
0x0000000000401078   be 44 15 c4 85 c0 74 14   .D....t.
0x0000000000401080   8b 4d fc 0f be 54 0d c4   .M...T..
0x0000000000401088   83 ea 02 8b 45 fc 88 54   ....E..T
0x0000000000401090   05 c4 eb d1 33 c0 8b 4d   ....3..M
0x0000000000401098   f8 33 cd e8 04 00 00 00   .3......
0x00000000004010a0   8b e5 5d c3 3b 0d 00 80   ..].;...
0x00000000004010a8   40 00 75 02 f3 c3 e9 94   @.u.....
0x00000000004010b0   01 00 00 8b ff 55 8b ec   .....U..
0x00000000004010b8   83 3d 28 8b 40 00 02 74   .=(.@..t
0x00000000004010c0   05 e8 97 07 00 00 ff 75   .......u
0x00000000004010c8   08 e8 e0 05 00 00 68 ff   ......h.
```

Figure 17.8: *Hexdump view*

Decompiler

To populate this tab, we will first go to the Windows menu and select **Functions** to get the list of functions in the binary. In the filter, search for the **main** function as it is the starting point of the binary. Once the **main** function is double-clicked, the decompiler will analyze the binary to display the high-level representation of the assembly code in the **Decompiler** tab. Cutter supports plugin for multiple decompilers such as RetDec and Ghidra.



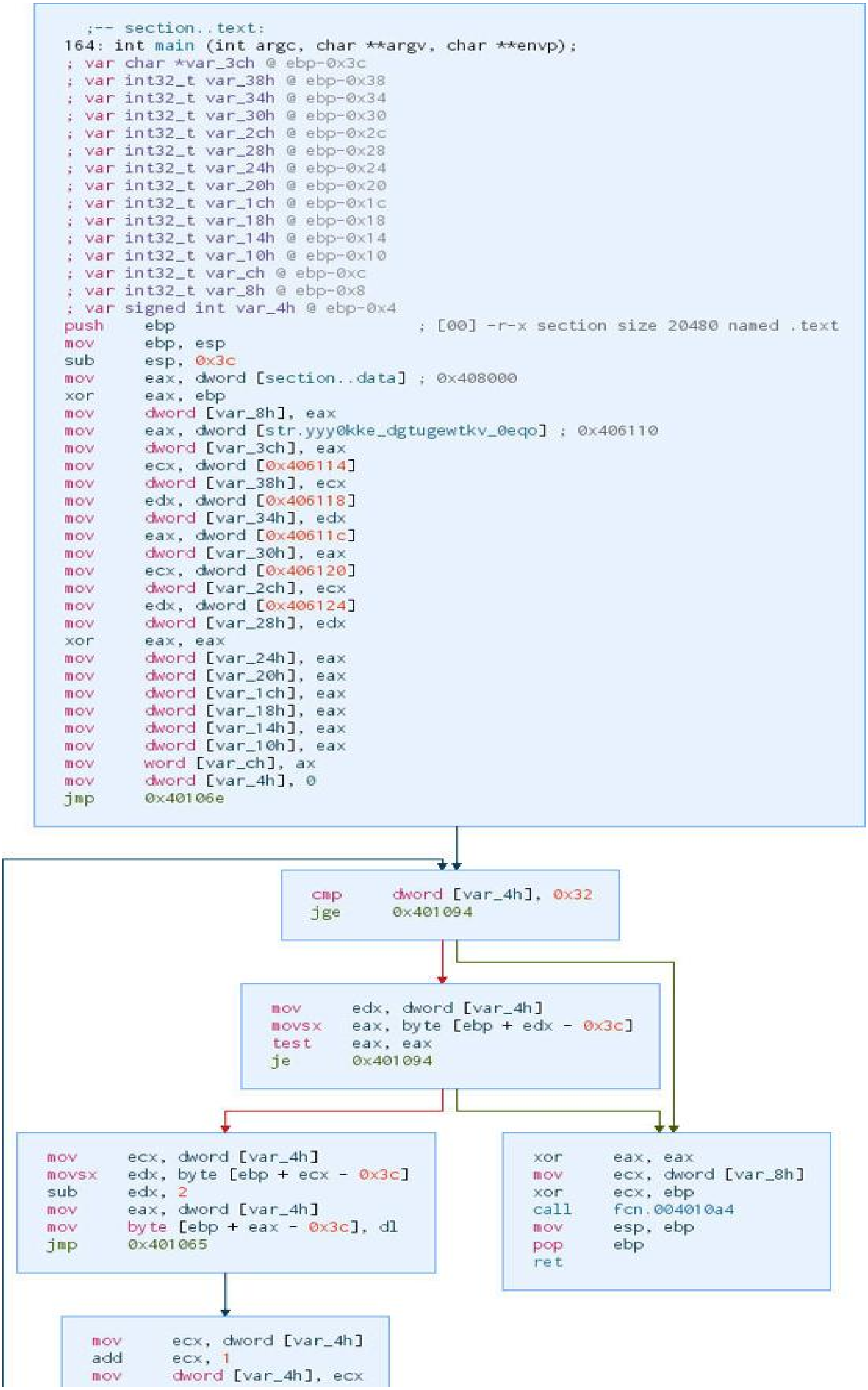
Decompiler: r2dec

```
/* r2dec pseudo code output */
/* C:\Users\enigma\Desktop\Cutter-v1.12.0-x64.Windows\CrackMe\CrackMe.exe @ 0x401000 */
#include <stdint.h>

int32_t main (void) {
    char * var_3ch;
    int32_t var_38h;
    int32_t var_34h;
    int32_t var_30h;
    int32_t var_2ch;
    int32_t var_28h;
    int32_t var_24h;
    int32_t var_20h;
    int32_t var_1ch;
    int32_t var_18h;
    int32_t var_14h;
    int32_t var_10h;
    int32_t var_ch;
    int32_t var_8h;
    int32_t var_4h;
    /* [00] -r-x section size 20480 named .text */
    eax = *(section..data);
    eax ^= ebp;
    var_8h = eax;
    eax = "yyy0kke{dgtugewtkv{0eqo";
    var_3ch = "yyy0kke{dgtugewtkv{0eqo";
    ecx = *(0x406114);
    var_38h = *(0x406114);
```

Figure 17.9: Decompiler view

Decrypting the Hidden URL

Now we will move back to the Graph section which displays the visual process flow of the **main** function. This graph view can be zoomed in or out using the *Ctrl*++ or *Ctrl*— shortcuts. We will understand the execution path in a step-by-step manner to understand what the binary is actually doing. The following is the screenshot of what we got in the graph view for the **main** function. In the graph view, we see blocks connected with arrows. Arrows are representations of different jumps such as The green arrow shows what happens when a jump takes place. The red arrow shows if a jump does not take place. The blue arrow shows the loop.



```
  ;-- section..text:
164: int main (int argc, char **argv, char **envp);
; var char *var_3ch @ ebp-0x3c
; var int32_t var_38h @ ebp-0x38
; var int32_t var_34h @ ebp-0x34
; var int32_t var_30h @ ebp-0x30
; var int32_t var_2ch @ ebp-0x2c
; var int32_t var_28h @ ebp-0x28
; var int32_t var_24h @ ebp-0x24
; var int32_t var_20h @ ebp-0x20
; var int32_t var_1ch @ ebp-0x1c
; var int32_t var_18h @ ebp-0x18
; var int32_t var_14h @ ebp-0x14
; var int32_t var_10h @ ebp-0x10
; var int32_t var_ch @ ebp-0xc
; var int32_t var_8h @ ebp-0x8
; var signed int var_4h @ ebp-0x4
push    ebp                       ; [00] -r-x section size 20480 named .text
mov     ebp, esp
sub     esp, 0x3c
mov     eax, dword [section..data] ; 0x408000
xor     eax, ebp
mov     dword [var_8h], eax
mov     eax, dword [str.yyy0kke_dgtugewtkv_0eqo] ; 0x406110
mov     dword [var_3ch], eax
mov     ecx, dword [0x406114]
mov     dword [var_38h], ecx
mov     edx, dword [0x406118]
mov     dword [var_34h], edx
mov     eax, dword [0x40611c]
mov     dword [var_30h], eax
mov     ecx, dword [0x406120]
mov     dword [var_2ch], ecx
mov     edx, dword [0x406124]
mov     dword [var_28h], edx
xor     eax, eax
mov     dword [var_24h], eax
mov     dword [var_20h], eax
mov     dword [var_1ch], eax
mov     dword [var_18h], eax
mov     dword [var_14h], eax
mov     dword [var_10h], eax
mov     word [var_ch], ax
mov     dword [var_4h], 0
jmp     0x40106e
```

```
cmp     dword [var_4h], 0x32
jge     0x401094
```

```
mov     edx, dword [var_4h]
movsx   eax, byte [ebp + edx - 0x3c]
test    eax, eax
je      0x401094
```

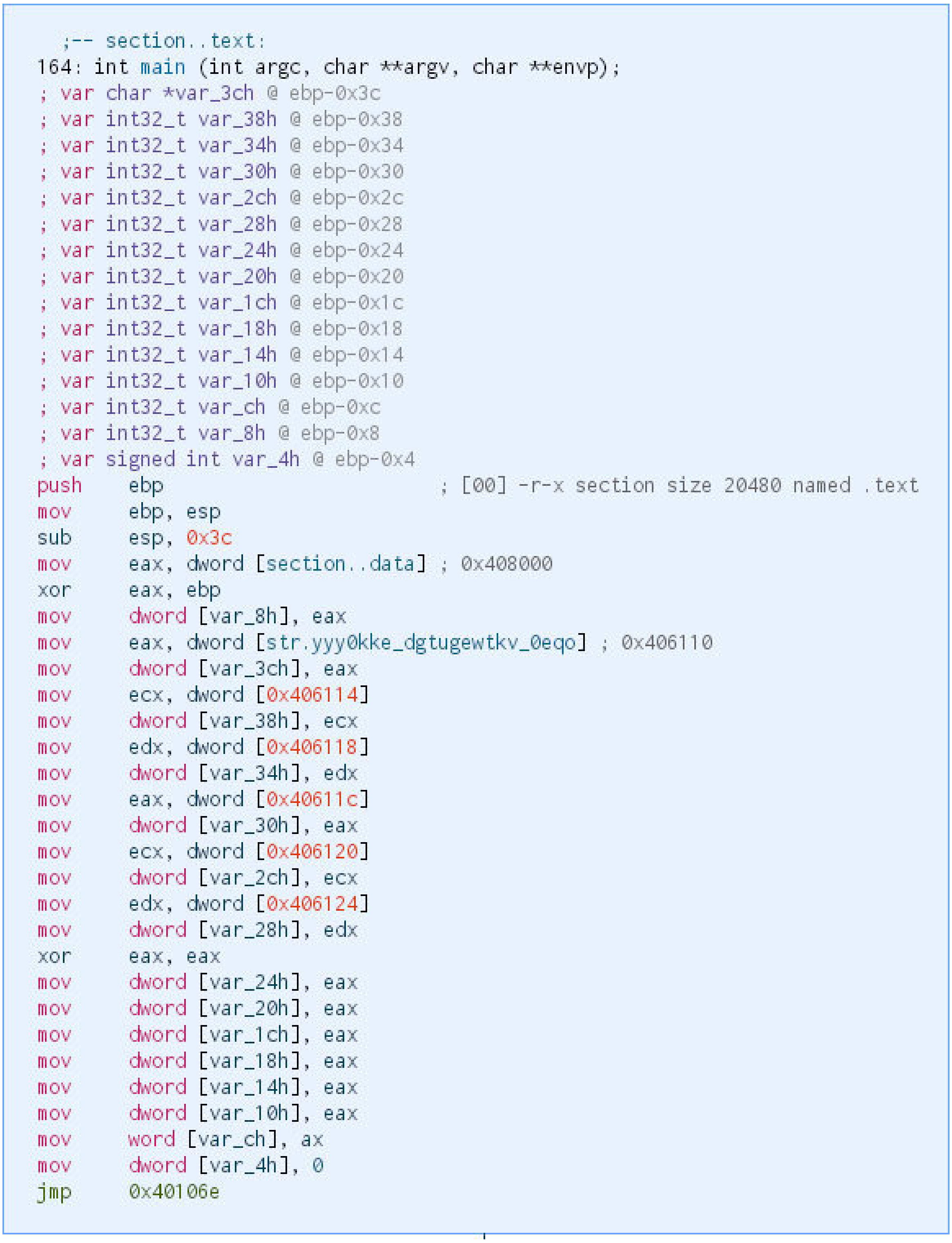
```
mov     ecx, dword [var_4h]
movsx   edx, byte [ebp + ecx - 0x3c]
sub     edx, 2
mov     eax, dword [var_4h]
mov     byte [ebp + eax - 0x3c], dl
jmp     0x401065
```

```
xor     eax, eax
mov     ecx, dword [var_8h]
xor     ecx, ebp
call    fcn.004010a4
mov     esp, ebp
pop     ebp
ret
```

```
mov     ecx, dword [var_4h]
add     ecx, 1
mov     dword [var_4h], ecx
```

Figure 17.10: Exported graph of main

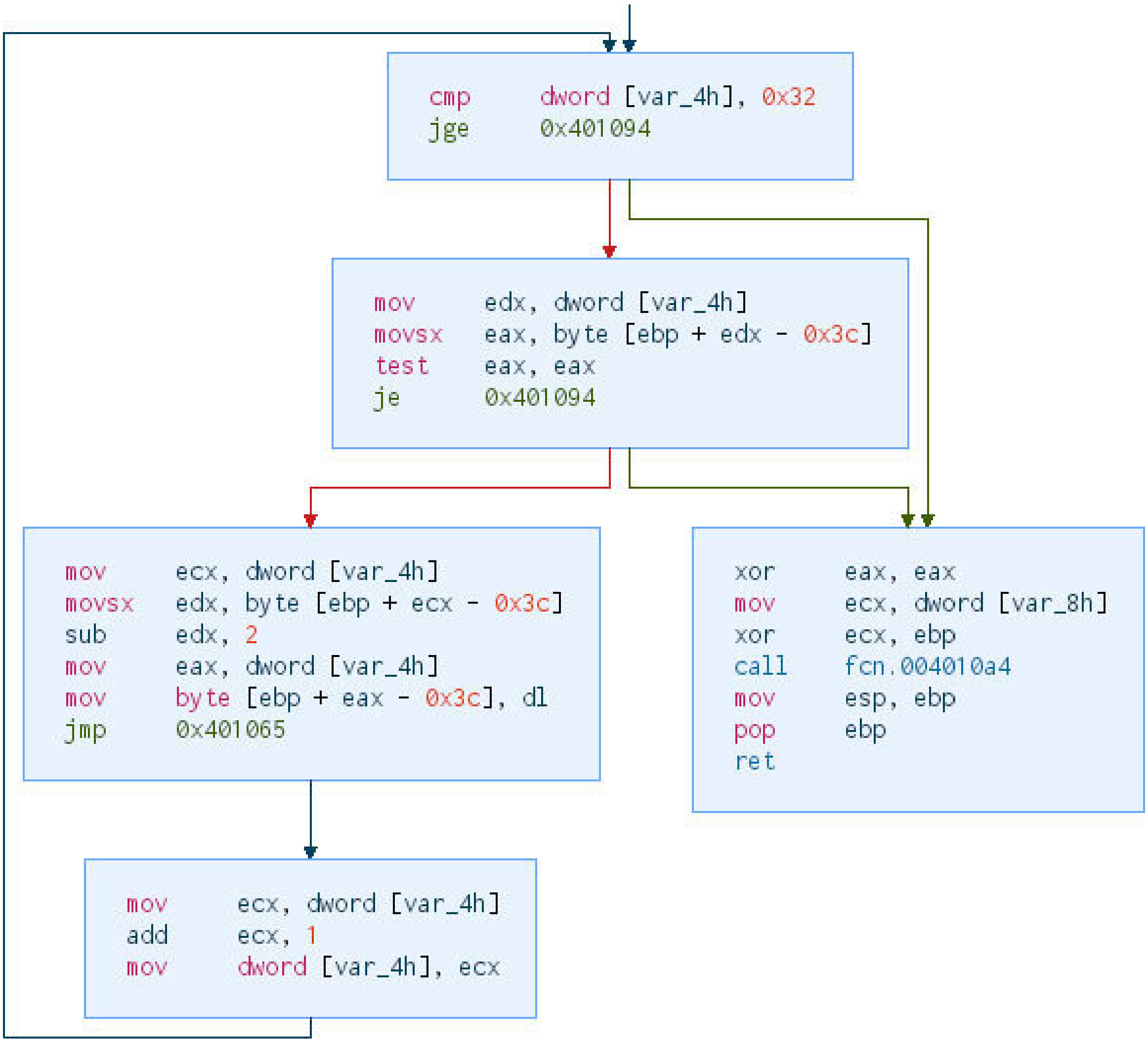
Let’s walk over the disassembled code block one by one to understand the code flow:



```
  ;-- section..text:
164: int main (int argc, char **argv, char **envp);
; var char *var_3ch @ ebp-0x3c
; var int32_t var_38h @ ebp-0x38
; var int32_t var_34h @ ebp-0x34
; var int32_t var_30h @ ebp-0x30
; var int32_t var_2ch @ ebp-0x2c
; var int32_t var_28h @ ebp-0x28
; var int32_t var_24h @ ebp-0x24
; var int32_t var_20h @ ebp-0x20
; var int32_t var_1ch @ ebp-0x1c
; var int32_t var_18h @ ebp-0x18
; var int32_t var_14h @ ebp-0x14
; var int32_t var_10h @ ebp-0x10
; var int32_t var_ch @ ebp-0xc
; var int32_t var_8h @ ebp-0x8
; var signed int var_4h @ ebp-0x4
push    ebp                        ; [00] -r-x section size 20480 named .text
mov     ebp, esp
sub     esp, 0x3c
mov     eax, dword [section..data] ; 0x408000
xor     eax, ebp
mov     dword [var_8h], eax
mov     eax, dword [str.yyy0kke_dgtugewtkv_0eqo] ; 0x406110
mov     dword [var_3ch], eax
mov     ecx, dword [0x406114]
mov     dword [var_38h], ecx
mov     edx, dword [0x406118]
mov     dword [var_34h], edx
mov     eax, dword [0x40611c]
mov     dword [var_30h], eax
mov     ecx, dword [0x406120]
mov     dword [var_2ch], ecx
mov     edx, dword [0x406124]
mov     dword [var_28h], edx
xor     eax, eax
mov     dword [var_24h], eax
mov     dword [var_20h], eax
mov     dword [var_1ch], eax
mov     dword [var_18h], eax
mov     dword [var_14h], eax
mov     dword [var_10h], eax
mov     word [var_ch], ax
mov     dword [var_4h], 0
jmp     0x40106e
```

Figure 17.11: *Exported graph of main-Block-1*

After the function prologue, the **SUB** instruction is creating room for the local variable by subtracting **ESP** by 0x3C (60 bytes in decimal). At the stack cookie is stored by XOR'ing **EAX** and In the subsequent **MOV** instructions, the text string with string length 24 (in decimal) is copied on the stack. This text string seems to be the encrypted version of something. We will call it the encrypted text string. Out of 60 bytes (in decimal) on the stack, we have consumed 24 bytes (in decimal) for encrypted text string and 4 bytes for the stack cookie. The remaining memory locations on the stack are cleared using the **MOV** instructions after XOR'ing the **EAX** instruction. This ends the blocks with a JMP instruction. Let's move to the remaining instructions in the blocks.



```
cmp     dword [var_4h], 0x32
jge     0x401094
```

```
mov     edx, dword [var_4h]
movsx   eax, byte [ebp + edx - 0x3c]
test    eax, eax
je      0x401094
```

```
mov     ecx, dword [var_4h]
movsx   edx, byte [ebp + ecx - 0x3c]
sub     edx, 2
mov     eax, dword [var_4h]
mov     byte [ebp + eax - 0x3c], dl
jmp     0x401065
```

```
xor     eax, eax
mov     ecx, dword [var_8h]
xor     ecx, ebp
call    fcn.004010a4
mov     esp, ebp
pop     ebp
ret
```

```
mov     ecx, dword [var_4h]
add     ecx, 1
mov     dword [var_4h], ecx
```

Figure 17.12: *Exported Graph of main remaining blocks*

We see the looping arrow represented by the blue arrow. It indicates that **ECX** is initialized to 0x32 (50 in decimal) to loop over with the **CMP** and **JGE** instructions. The green arrow on the right indicates that **ECX** reaches a count of 50 in decimal. This green arrow moves to the end of the code block. The red arrow indicates that we are inside the loop.

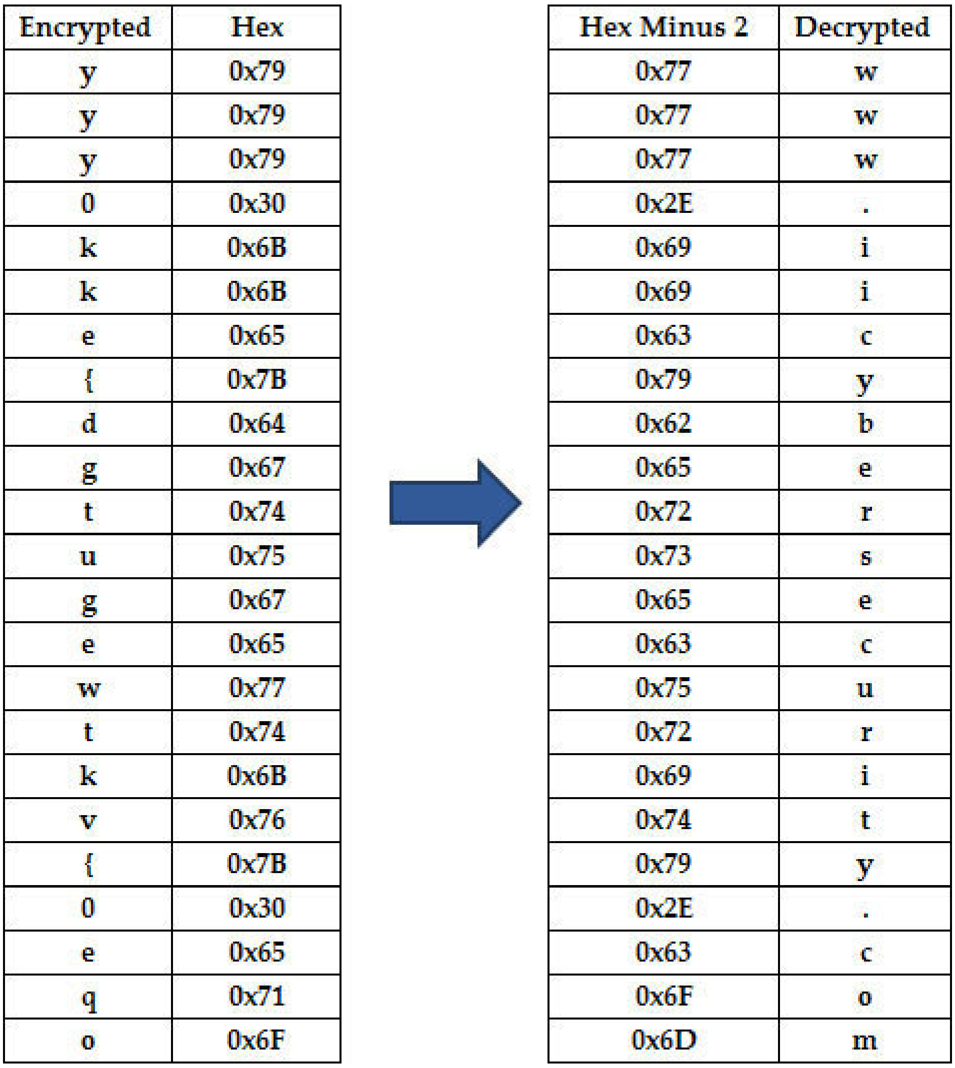
The **MOVSX** instruction is copying every **char** from the encrypted text string on the stack to Check if it is NULL or not. If it is NULL, then move to the end of the code block at the bottom right. Or else, again copy the first byte (char) from the encrypted text string on the stack to **EDX** using the **MOVSX** instruction. The **SUB** instruction subtracts 2 from the HEX value of the char copied in From it is moved to overwrite the first char of the encrypted text string on the stack in the first loop. This loop is carried out until all the HEX values of the characters of the encrypted text string are subtracted by 2. Thus, 2 is the encrypting key used to encrypt some text string.

With this, we can generate a high-level pseudo code for the binary as follows:

```
EncryptedText = “yyyokke{dgtugewtkv{oeqo”
for(iteration=50, EncryptedText !=’\0’, iteration++)
{
EncryptedText[iteration] = EncryptedText[iteration] -2;
}
return 0;
```

So, we saw how to generate the pseudo code by stepping over the assembly instructions and extracting the meaning out of it. Your pseudo code might not be the same as the original binary code, but it's a glimpse of what's happening inside the binary working. With this, we are now clear on how to extract the hidden text behind the encrypted text string.

To extract the original text string, we can either follow a manual or automated process. Manually, it can be done as follows. Refer to the ASCII table in the *Appendix* for the char to hex conversion.



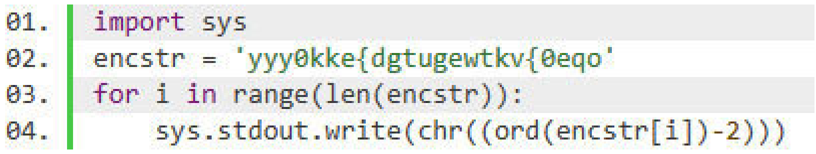
Encrypted	Hex
y	0x79
y	0x79
y	0x79
0	0x30
k	0x6B
k	0x6B
e	0x65
{	0x7B
d	0x64
g	0x67
t	0x74
u	0x75
g	0x67
e	0x65
w	0x77
t	0x74
k	0x6B
v	0x76
{	0x7B
0	0x30
e	0x65
q	0x71
o	0x6F

Hex Minus 2	Decrypted
0x77	w
0x77	w
0x77	w
0x2E	.
0x69	i
0x69	i
0x63	c
0x79	y
0x62	b
0x65	e
0x72	r
0x73	s
0x65	e
0x63	c
0x75	u
0x72	r
0x69	i
0x74	t
0x79	y
0x2E	.
0x63	c
0x6F	o
0x6D	m

***Figure 17.13:** Decrypted text*

So, the decrypted text is a URL, which is This was a simple custom encryption to hide the URL in the code. If a plain text URL is used in the code, then it would be visible in the Strings tab of the Cutter graphical interface.

The automated approach to extract the URL from an encrypted URL is to write the Python code as follows:



```
import sys
encstr = 'yyy0kke{dgtugewtkv{0eqo'
for i in range(len(encstr)):
    sys.stdout.write(chr((ord(encstr[i])-2)))
```

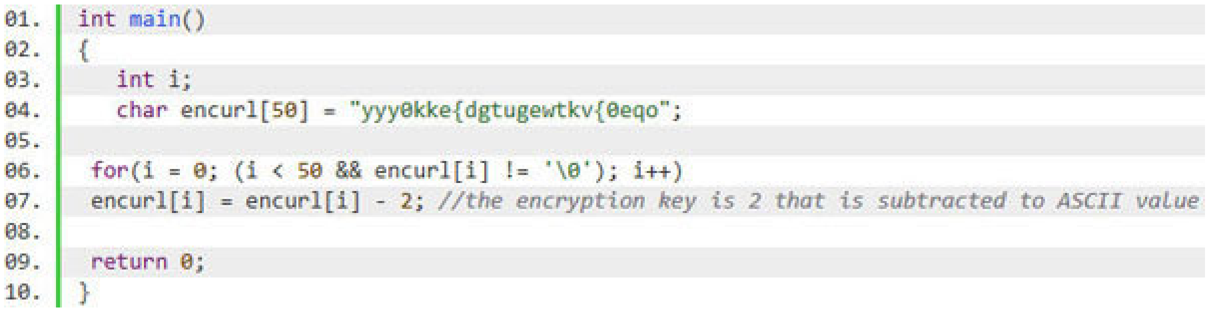
***Figure 17.14:** Python code to get the decrypted text*

The output of the preceding Python program is as follows:

www.iicybersecurity.com

In the preceding Python code, we are iterating over every char of **encstr** and subtracting the encryption key 2 from the ASCII value to get the decrypted char using the **chr** function.

It is time to check the original C++ code from which the **CrackMe** binary is generated.



```
int main()
{
   int i;
   char encurl[50] = "yyy0kke{dgtugewtkv{0eqo";

 for(i = 0; (i < 50 && encurl[i] != '\0'); i++)
 encurl[i] = encurl[i] - 2; //the encryption key is 2 that is subtracted to ASCII value

 return 0;
}
```

Figure 17.15: *Binary CPP code*

With this, we are able to crack the simple encryption used in the binary.

Conclusion

We covered the steps involved in the installation of the reverse engineering framework called With what we have learned in the previous chapters, we were able to analyze the binary to generate a pseudo code to extract the encryption key. We also covered the manual as well as the automated way to extract a hidden URL from the encrypted text. In the next chapter, we will learn some new things about the well-known Windows application.

CHAPTER 18

Fun With Windows Calculator Using Reverse Engineering

In this chapter, we will take up an example to understand how we can use reverse engineering to modify applications or software behaviour without having access to the source code. We will take a well-known Windows application used by everyone. Even those who know the basics of computer use it. We are talking about the Windows Calculator. It is used by computer learners, intermediates, and experts. Everyone uses it for basic and advance calculations. So, what are we going to do with this calculator? This will be an interesting real-life example where, as a reverse engineer, we will change the working of an application with having its source code.

If you are reverse engineering a malware, then this type of real-life scenario will help you change the execution flow of any malware. We will also talk in detail about many concepts involved in this process. This will be a fun exercise to understand.

Structure

In this chapter, we will cover the following topics:

Reverse engineering a calculator

Understanding the code flow with breakpoints

Finding a placeholder to call our code

Writing our code in the Code Cave

Patching the binary

Objective

In this chapter, using reverse engineering, we will change the working of a calculator by modifying its behaviour to output our defined string for any calculation that we perform. It means that rather than getting 8 as an output to 2+6 or any other calculation, the calculator will display our defined message on the press of **equal to** button. For this, we will use the Win32 Calculator available in the old Windows XP.

Reverse Engineering Calculator

If we want to change the behaviour of any application, we can do so in the application source code and recompile it to get the desired result on execution. In this case, we only have the calculator binary or **Portable Executable** file. We will use engineering to modify the calculator binary by writing our code to work as desired. To get the desired result, we will follow a 4-step process to modify the calculator binary:

Understanding the code flow with breakpoints

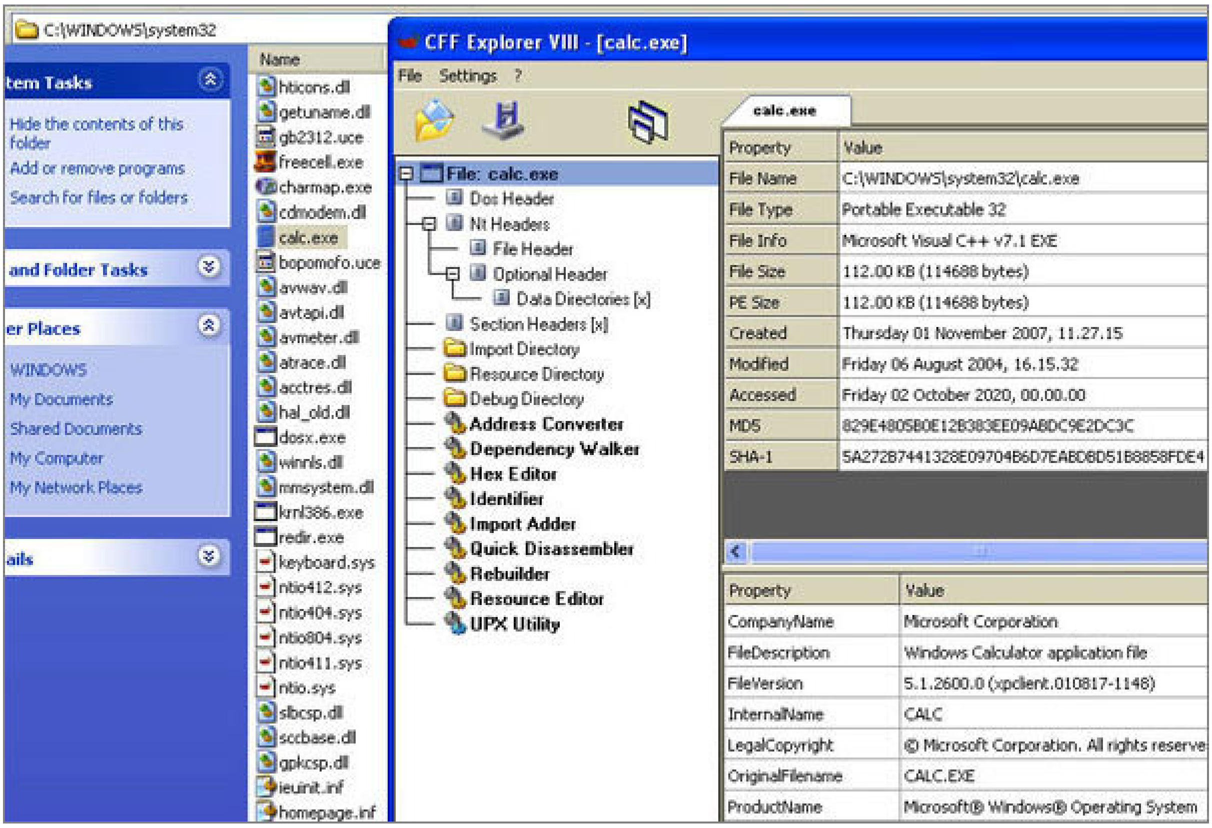
Finding a placeholder to call our code

Writing our code in the Code Cave

Patching the binary

Understanding the code flow with breakpoints

This step involves understanding the code flow of an application, the calculator in our case, using x32dbg or Ollydbg. Our objective is to modify the calculator is such a way that if somebody presses the **equal to** button, the user will be presented with our desired string rather than the calculated output. To start, take a copy of **calc.exe** from the **WinXP system32** folder.

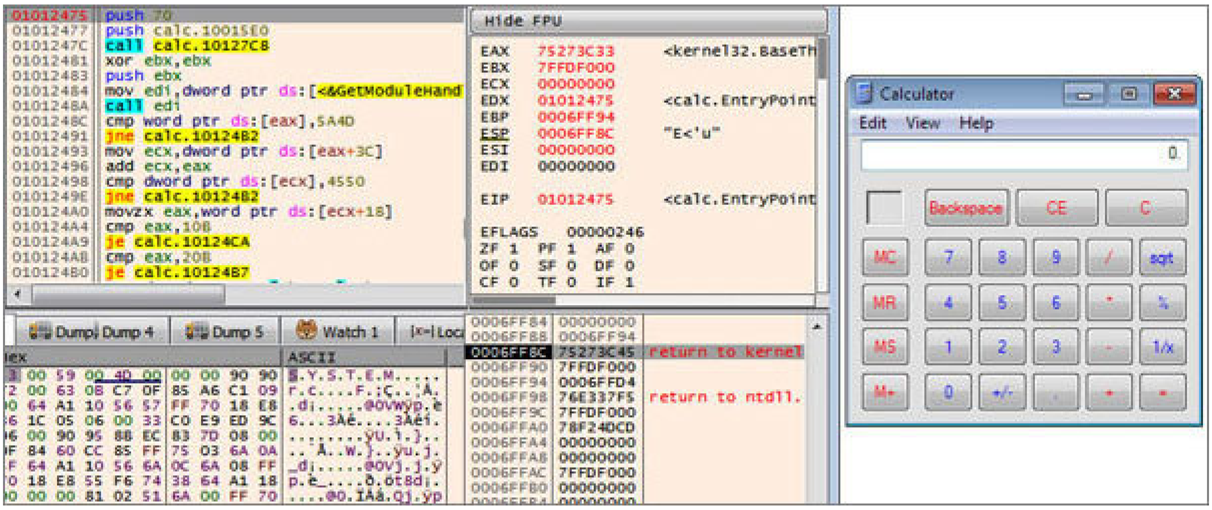


Property	Value
File Name	C:\WINDOWS\system32\calc.exe
File Type	Portable Executable 32
File Info	Microsoft Visual C++ v7.1 EXE
File Size	112.00 KB (114688 bytes)
PE Size	112.00 KB (114688 bytes)
Created	Thursday 01 November 2007, 11.27.15
Modified	Friday 06 August 2004, 16.15.32
Accessed	Friday 02 October 2020, 00.00.00
MD5	829E4805B0E12B383EE09ABDC9E2DC3C
SHA-1	5A272B7441328E09704B6D7EABD8D51B8858FDE4

Property	Value
CompanyName	Microsoft Corporation
FileDescription	Windows Calculator application file
FileVersion	5.1.2600.0 (xpclient.010817-1148)
InternalName	CALC
LegalCopyright	© Microsoft Corporation. All rights reserve
OriginalFilename	CALC.EXE
ProductName	Microsoft® Windows® Operating System

Figure 18.1: *Calculator binary path*

CFF Explorer shows that the binary is Portable Executable 32. Open this binary in x32dbg and go to **Debug | Run** to start the calculator. Or you can also press *F9* to run.



```
01012475  push 70
01012477  push calc.1001SE0
0101247C  call calc.10127C8
01012481  xor ebx,ebx
01012483  push ebx
01012484  mov edi,dword ptr ds:[<&GetModuleHand
0101248A  call edi
0101248C  cmp word ptr ds:[eax],5A4D
01012491  jne calc.10124B2
01012493  mov ecx,dword ptr ds:[eax+3C]
01012496  add ecx,eax
01012498  cmp dword ptr ds:[ecx],4550
0101249E  jne calc.10124B2
010124A0  movzx eax,word ptr ds:[ecx+18]
010124A4  cmp eax,10B
010124A9  je calc.10124CA
010124AB  cmp eax,20B
010124B0  je calc.10124B7
```

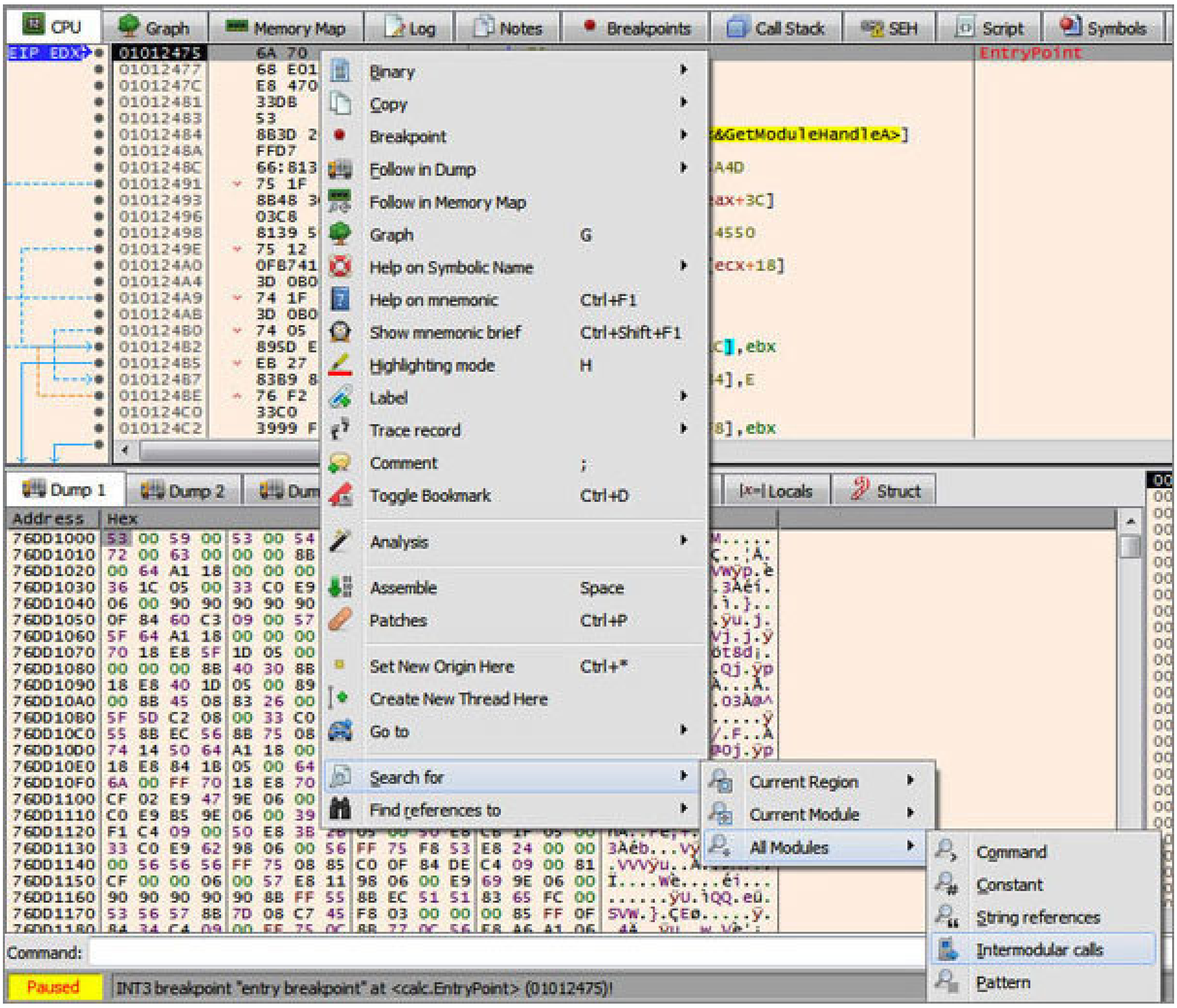
```
Hide FPU

EAX   75273C33     <kernel32.BaseTh
EBX   7FFDF000
ECX   00000000
EDX   01012475     <calc.EntryPoint
EBP   0006FF94
ESP   0006FF8C     "E<'u"
ESI   00000000
EDI   00000000

EIP   01012475     <calc.EntryPoint

EFLAGS   00000246
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 0  IF 1
```

```
0006FF84  00000000
0006FF88  0006FF94
0006FF8C  75273C45  return to kernel
0006FF90  7FFDF000
0006FF94  0006FFD4
0006FF98  76E337F5  return to ntdll.
0006FF9C  7FFDF000
0006FFA0  78F24DCD
0006FFA4  00000000
0006FFA8  00000000
0006FFAC  7FFDF000
0006FFB0  00000000
```

```
ex                                   ASCII
3 00 59 00 40 00  00 00 90 90
2 00 63 0B C7 0F  85 A6 C1 09
0 64 A1 10 56 57  FF 70 18 E8
6 1C 05 06 00 33  C0 E9 ED 9C
6 00 90 95 8B EC  83 7D 08 00
F 84 60 CC 85 FF  75 03 6A 0A
F 64 A1 10 56 6A  0C 6A 08 FF
0 18 E8 55 F6 74  38 64 A1 18
0 00 00 81 02 51  6A 00 FF 70
```

Figure 18.2: *Open the calculator binary in x32dbg*

First, we will check the list of Win32 function calls in our binary by going to the **CPU** tab, right-clicking to get the context menu, and then going to **Search for -> All Modules -> Intermodular**



INT3 breakpoint "entry breakpoint" at <calc.EntryPoint> (01012475)!

Figure 18.3: Intermodular calls

This will bring up the **All Modules (Calls)** tab, where we can see a bunch of function calls and in the bottom, we have a filter option to find any specific function call reference. Now, Win32 offers the **SetWindowText** function to change the text of control.

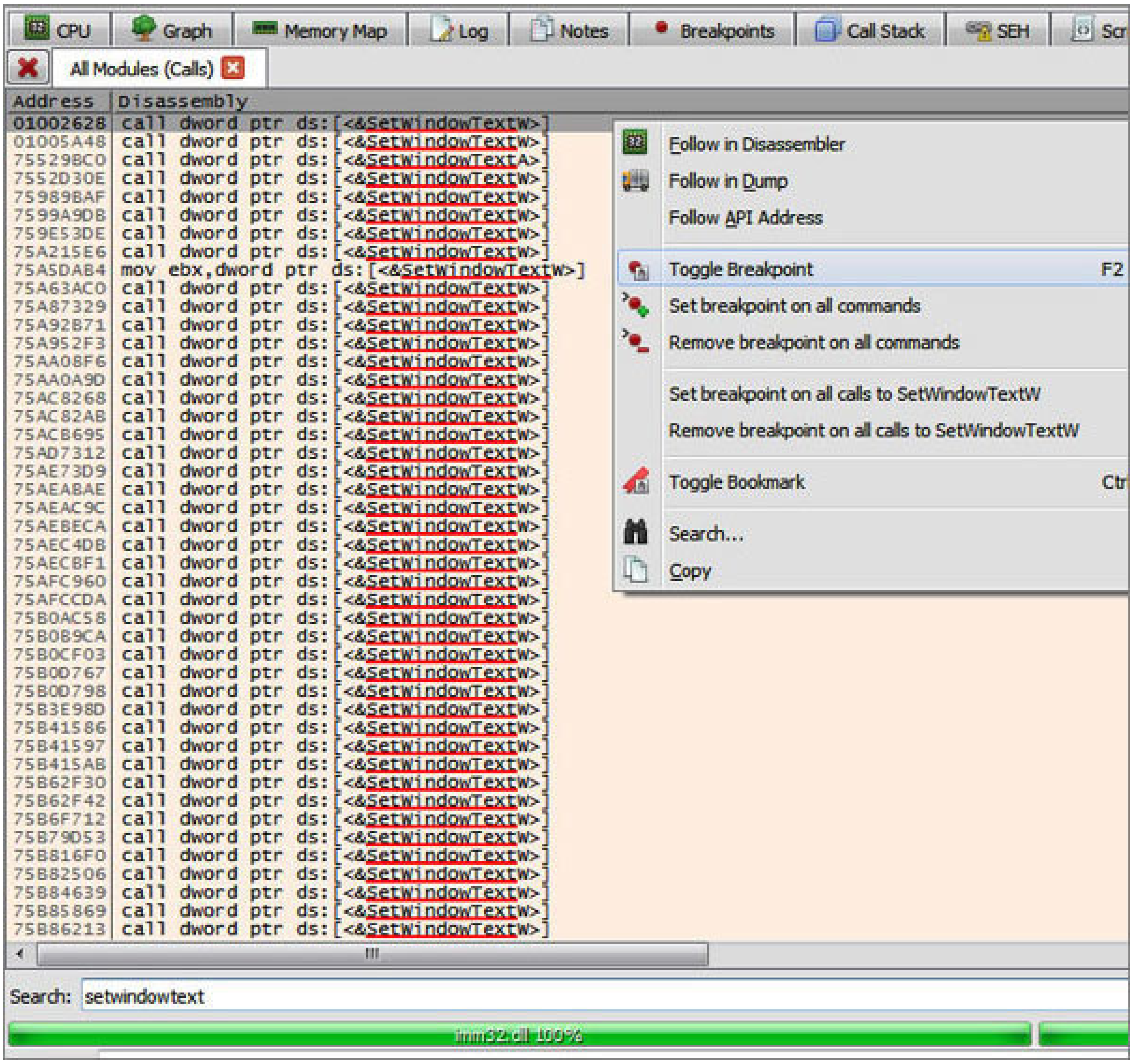
```
BOOL SetWindowTextW(
HWND      hWnd,
LPCWSTR lpString
);
```

It takes 2 parameters:

hWnd is the handle to the window or control whose text is to be changed.

lpString is the control text

Let's find this function in the **Search** filter to find all the references to the function call in our code and set the breakpoint.

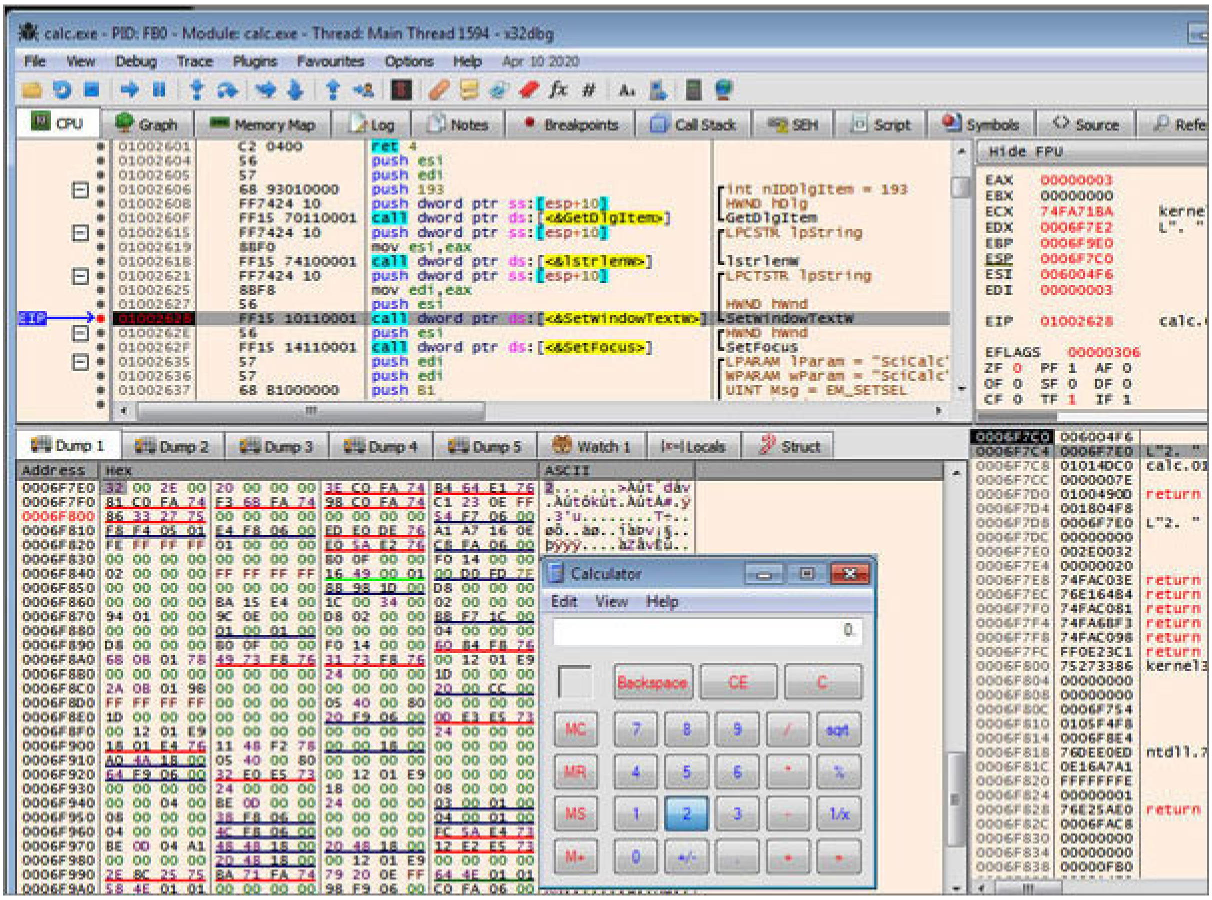


```
Address  Disassembly
01002628 call dword ptr ds:[<&SetWindowTextW>]
01005A48 call dword ptr ds:[<&SetWindowTextW>]
755298C0 call dword ptr ds:[<&SetWindowTextA>]
7552D30E call dword ptr ds:[<&SetWindowTextW>]
759898AF call dword ptr ds:[<&SetWindowTextW>]
7599A9D8 call dword ptr ds:[<&SetWindowTextW>]
759E53DE call dword ptr ds:[<&SetWindowTextW>]
75A215E6 call dword ptr ds:[<&SetWindowTextW>]
75A5DAB4 mov ebx,dword ptr ds:[<&SetWindowTextW>]
75A63AC0 call dword ptr ds:[<&SetWindowTextW>]
75A87329 call dword ptr ds:[<&SetWindowTextW>]
75A92B71 call dword ptr ds:[<&SetWindowTextW>]
75A952F3 call dword ptr ds:[<&SetWindowTextW>]
75AA08F6 call dword ptr ds:[<&SetWindowTextW>]
75AA0A9D call dword ptr ds:[<&SetWindowTextW>]
75AC8268 call dword ptr ds:[<&SetWindowTextW>]
75AC82AB call dword ptr ds:[<&SetWindowTextW>]
75ACB695 call dword ptr ds:[<&SetWindowTextW>]
75AD7312 call dword ptr ds:[<&SetWindowTextW>]
75AE73D9 call dword ptr ds:[<&SetWindowTextW>]
75AEABAE call dword ptr ds:[<&SetWindowTextW>]
75AEAC9C call dword ptr ds:[<&SetWindowTextW>]
75AEBECA call dword ptr ds:[<&SetWindowTextW>]
75AEC4DB call dword ptr ds:[<&SetWindowTextW>]
75AECBF1 call dword ptr ds:[<&SetWindowTextW>]
75AFC960 call dword ptr ds:[<&SetWindowTextW>]
75AFCCDA call dword ptr ds:[<&SetWindowTextW>]
75B0AC58 call dword ptr ds:[<&SetWindowTextW>]
75B0B9CA call dword ptr ds:[<&SetWindowTextW>]
75B0CF03 call dword ptr ds:[<&SetWindowTextW>]
75B0D767 call dword ptr ds:[<&SetWindowTextW>]
75B0D798 call dword ptr ds:[<&SetWindowTextW>]
75B3E98D call dword ptr ds:[<&SetWindowTextW>]
75B41586 call dword ptr ds:[<&SetWindowTextW>]
75B41597 call dword ptr ds:[<&SetWindowTextW>]
75B415AB call dword ptr ds:[<&SetWindowTextW>]
75B62F30 call dword ptr ds:[<&SetWindowTextW>]
75B62F42 call dword ptr ds:[<&SetWindowTextW>]
75B6F712 call dword ptr ds:[<&SetWindowTextW>]
75B79D53 call dword ptr ds:[<&SetWindowTextW>]
75B816F0 call dword ptr ds:[<&SetWindowTextW>]
75B82506 call dword ptr ds:[<&SetWindowTextW>]
75B84639 call dword ptr ds:[<&SetWindowTextW>]
75B85869 call dword ptr ds:[<&SetWindowTextW>]
75B86213 call dword ptr ds:[<&SetWindowTextW>]
```

Search: setwindowtext

Figure 18.4: Search SetWindowTextW and set the breakpoint

Now, we are all set to understand the code flow of the application. We will perform a calculation of 2+6 to see if our breakpoint is hit or not. Let's begin by first pressing the button for the digit As soon as **2** is pressed on the calculator, our breakpoint is hit.



```
01002601    C2 0400            ret 4
01002604    56                 push esi
01002605    57                 push edi
01002606    68 93010000        push 193                              int nIDDlgItem = 193
01002608    FF7424 10          push dword ptr ss:[esp+10]            HWND hDlg
0100260F    FF15 70110001      call dword ptr ds:[<&GetDlgItem>]     GetDlgItem
01002615    FF7424 10          push dword ptr ss:[esp+10]            LPCSTR lpString
01002619    8BF0               mov esi,eax
0100261B    FF15 74100001      call dword ptr ds:[<&lstrlenW>]       lstrlenW
01002621    FF7424 10          push dword ptr ss:[esp+10]            LPCTSTR lpString
01002625    8BF8               mov edi,eax
01002627    56                 push esi                              HWND hwnd
01002628    FF15 10110001      call dword ptr ds:[<&SetWindowTextW>] SetWindowTextW
0100262E    56                 push esi                              HWND hwnd
0100262F    FF15 14110001      call dword ptr ds:[<&SetFocus>]       SetFocus
01002635    57                 push edi                              LPARAM lParam = "SciCalc'
01002636    57                 push edi                              WPARAM wParam = "SciCalc'
01002637    68 B1000000        push B1                               UINT Msg = EM_SETSEL
```

```
EAX   00000003
EBX   00000000
ECX   74FA718A
EDX   0006F7E2
EBP   0006F9E0
ESP   0006F7C0
ESI   006004F6
EDI   00000003

EIP   01002628

EFLAGS  00000306
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1
```

Figure 18.5: Breakpoint hit as 2 is pressed

The breakpoint is set at the **SetWindowTextW** function call and we can see that before the **CALL** to the arguments to the function are pushed onto the stack.

stack. stack. stack. stack.

stack. stack. stack. stack.

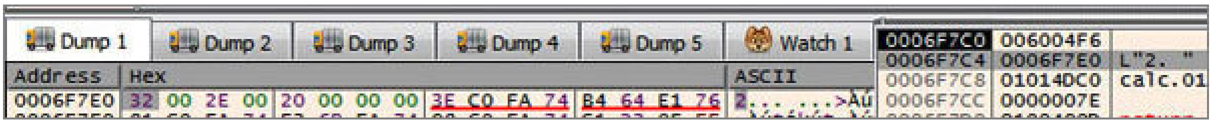
stack.

The second argument of **SetWindowTextW** is **LPCWSTR** which is pushed first on the stack by the instruction:

push dword ptr ss:[esp+0x10]

This pushes **[ESP+0x10]** on the stack. While running this push instruction, **[ESP+0x10]** was pointing to which is the memory location of the digit we pressed on the calculator, So that means the memory location of the digit **2** (which we pressed on the calculator) was pushed on the stack.

Let's see how the digit 2 is stored in the memory by dumping **0x0006F7E0** in the memory dump. We can observe that the digit is stored in the Unicode format. For understanding ASCII and Unicode, refer to the



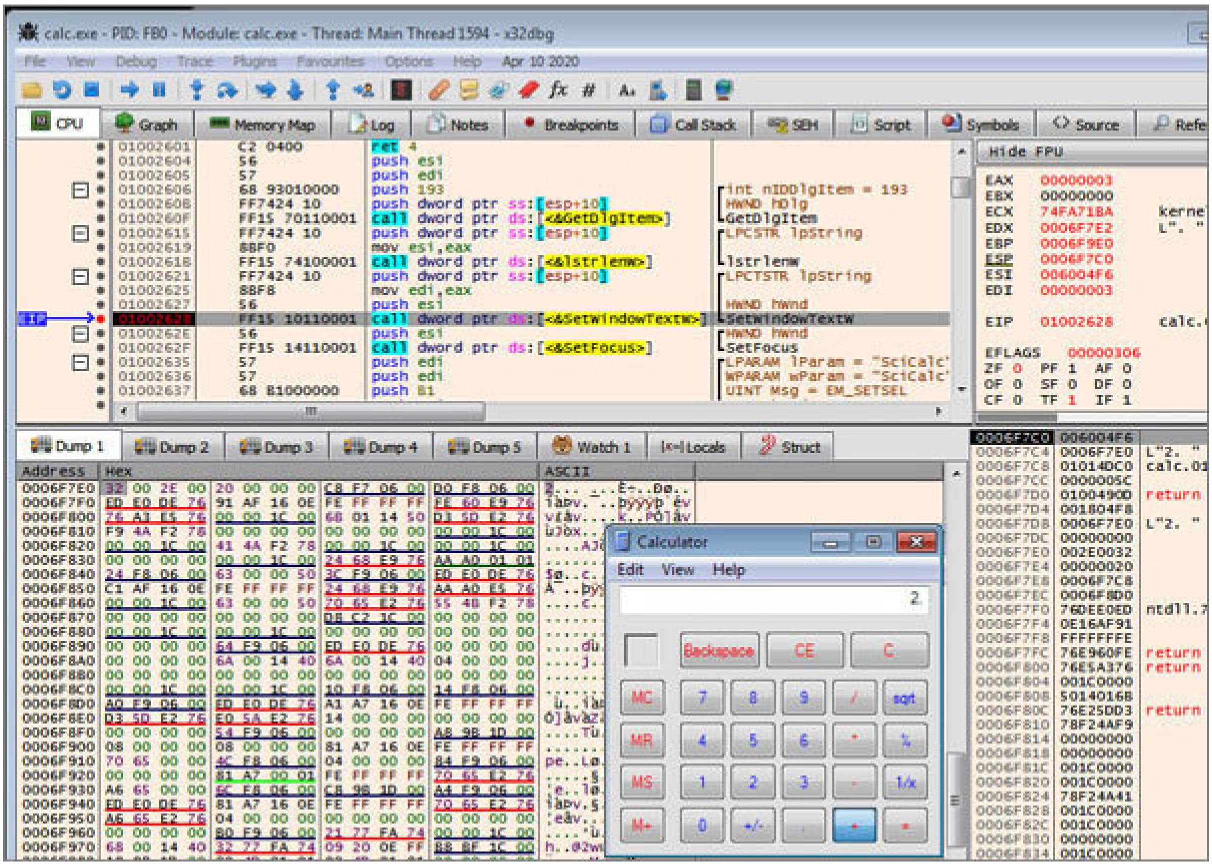
```
Address   Hex                                                ASCII
0006F7E0  32 00 2E 00 20 00 00 00 3E C0 FA 74 B4 64 E1 76  2.... ...>Àú
```

```
0006F7C0  006004F6
0006F7C4  0006F7E0  L"2. "
0006F7C8  01014DC0  calc.01
0006F7CC  0000007E
```

__Figure 18.6:__ Memory location of the digit 2

The first argument of **SetWindowTextW** is **HWND** which is the handle to the control whose text is to be changed. This is pushed on the stack by the **PUSH ESI** instruction.

We now understand how a digit that we press on the calculator is stored in the memory. Now we will press **Run** in x32dbg to return the control back to the calculator. Once the calculator has the control, press **plus (+)** on the calculator. After pressing **plus** the breakpoint is hit again.



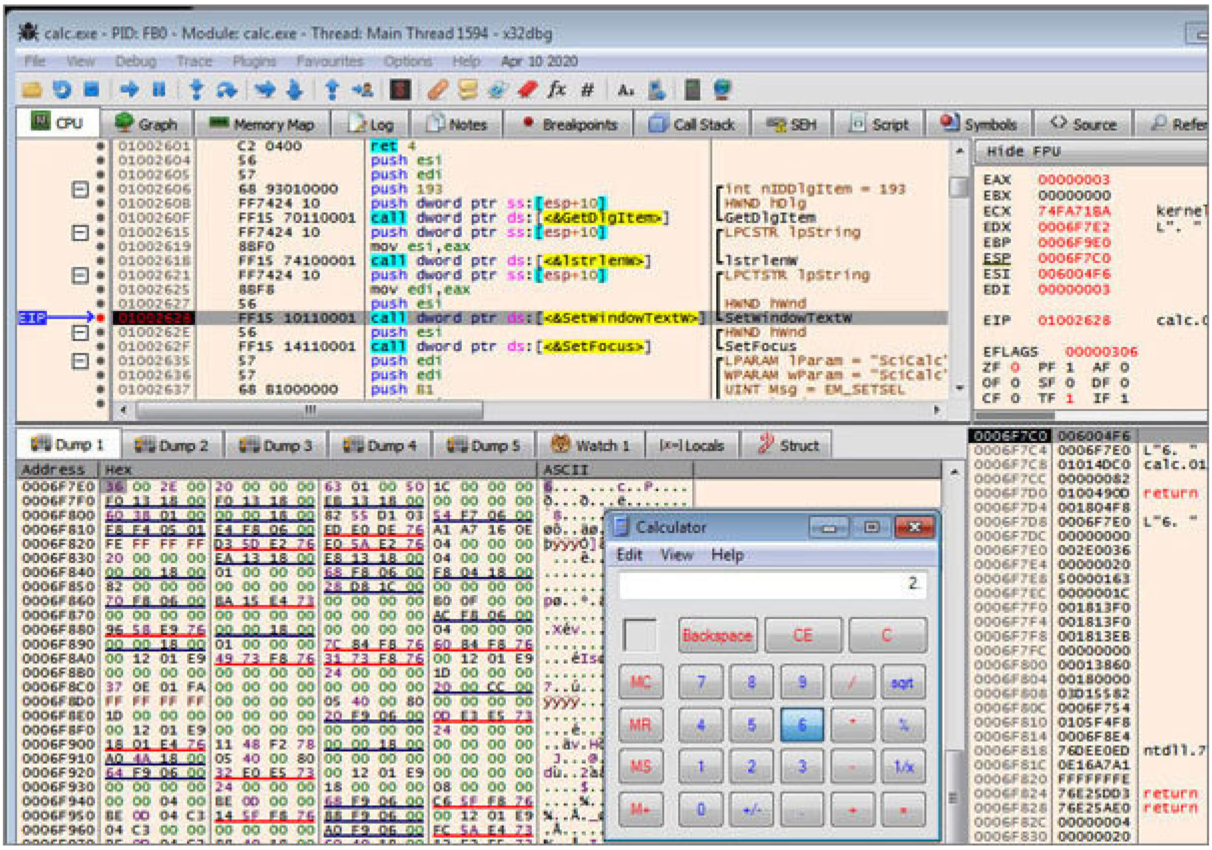
```
01002601   C2 0400           ret 4
01002604   56                push esi
01002605   57                push edi
01002606   68 93010000       push 193                              int nIDDlgItem = 193
01002608   FF7424 10         push dword ptr ss:[esp+10]            HWND hDlg
0100260F   FF15 70110001     call dword ptr ds:[<&GetDlgItem>]     GetDlgItem
01002615   FF7424 10         push dword ptr ss:[esp+10]            LPCSTR lpString
01002619   8BF0              mov esi,eax
0100261B   FF15 74100001     call dword ptr ds:[<&lstrlenW>]       lstrlenW
01002621   FF7424 10         push dword ptr ss:[esp+10]            LPCTSTR lpString
01002625   8BF8              mov edi,eax
01002627   56                push esi                              HWND hwnd
01002628   FF15 10110001     call dword ptr ds:[<&SetWindowTextW>] SetWindowTextW
0100262E   56                push esi                              HWND hwnd
0100262F   FF15 14110001     call dword ptr ds:[<&SetFocus>]       SetFocus
01002635   57                push edi                              LPARAM lParam = "SciCalc"
01002636   57                push edi                              WPARAM wParam = "SciCalc"
01002637   68 B1000000       push B1                               UINT Msg = EM_SETSEL
```

```
EAX   00000003
EBX   00000000
ECX   74FA71BA     kerne
EDX   0006F7E2     L". "
EBP   0006F9E0
ESP   0006F7C0
ESI   006004F6
EDI   00000003

EIP   01002628     calc.

EFLAGS   00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

__Figure 18.7:__ Breakpoint hit when + is pressed

All the registers are unchanged at this point and we see that there is no identifier to differentiate when the **plus (+)** is pressed on the calculator. Press **Run** in x32dbg again to return the control back to the calculator.

Now we will press the next digit, which is on the calculator. We can again see that our breakpoint is hit and the stack is pushed with the memory location of the digit The same behaviour was observed earlier, wherein the parameters to the **SetWindowTextW** function were pushed onto the stack.



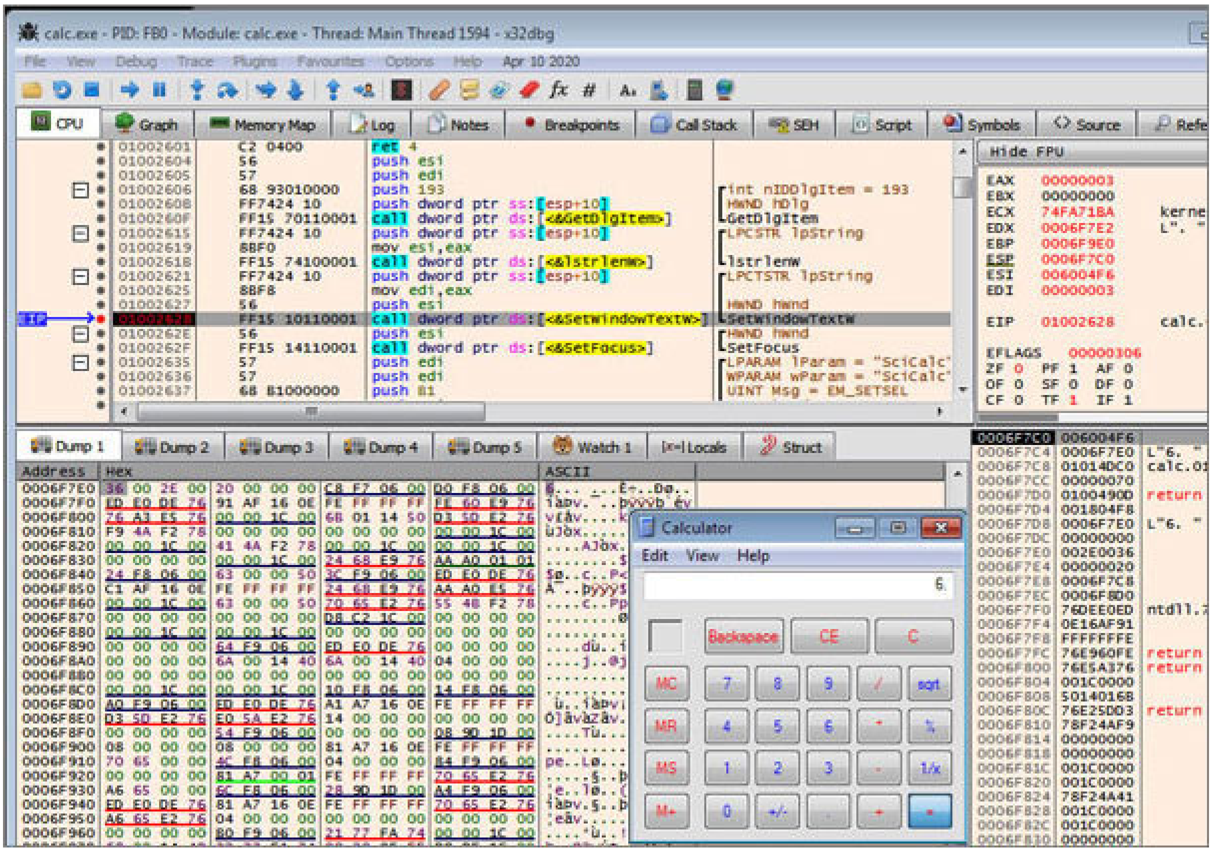
calc.exe - PID: FB0 - Module: calc.exe - Thread: Main Thread 1594 - x32dbg

```
01002601   C2 0400          ret 4
01002604   56               push esi
01002605   57               push edi
01002606   68 93010000      push 193                                   int nIDDlgItem = 193
01002608   FF7424 10        push dword ptr ss:[esp+10]                 HWND hDlg
0100260F   FF15 70110001    call dword ptr ds:[<&GetDlgItem>]          GetDlgItem
01002615   FF7424 10        push dword ptr ss:[esp+10]                 LPCSTR lpString
01002619   8BF0             mov esi,eax
0100261B   FF15 74100001    call dword ptr ds:[<&lstrlenW>]            lstrlenW
01002621   FF7424 10        push dword ptr ss:[esp+10]                 LPCTSTR lpString
01002625   8BF8             mov edi,eax
01002627   56               push esi                                   HWND hwnd
01002628   FF15 10110001    call dword ptr ds:[<&SetWindowTextW>]      SetWindowTextW
0100262E   56               push esi                                   HWND hwnd
0100262F   FF15 14110001    call dword ptr ds:[<&SetFocus>]            SetFocus
01002635   57               push edi                                   LPARAM lParam = "SciCalc"
01002636   57               push edi                                   WPARAM wParam = "SciCalc"
01002637   68 81000000      push 81                                    UINT Msg = EM_SETSEL
```

```
EAX     00000003
EBX     00000000
ECX     74FA718A     kernel
EDX     0006F7E2     L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628     calc.0

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

Figure 18.8: Breakpoint hit on pressing 6

Next, to evaluate 2+6, we will press the **equal to** button on the calculator. After pressing **equal** our breakpoint is hit but nothing special is observed on the stack or in the register value to help us differentiate when the **equal to** button is pressed on the calculator.

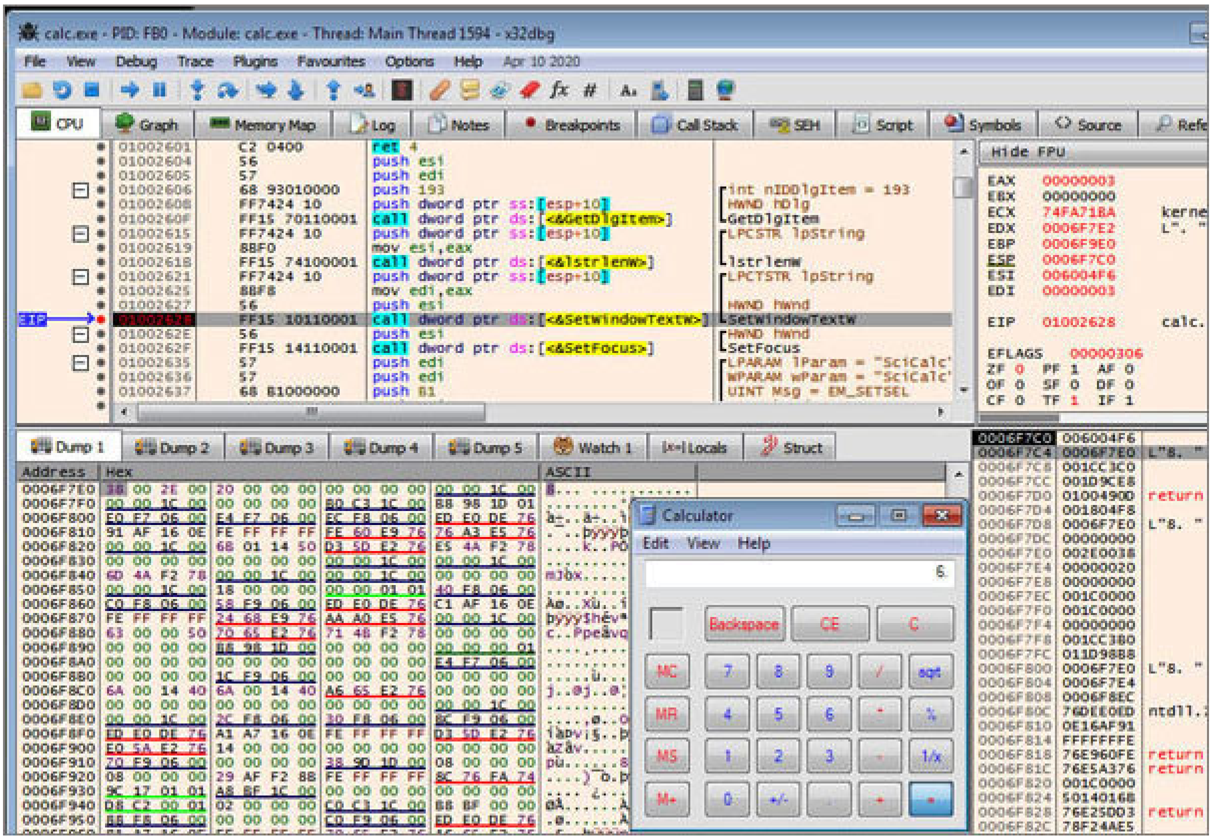


```
01002601  C2 0400            ret 4
01002604  56                 push esi
01002605  57                 push edi
01002606  68 93010000        push 193                                   int nIDDlgItem = 193
0100260B  FF7424 10          push dword ptr ss:[esp+10]                 HWND hDlg
0100260F  FF15 70110001      call dword ptr ds:[<&GetDlgItem>]          GetDlgItem
01002615  FF7424 10          push dword ptr ss:[esp+10]                 LPCSTR lpString
01002619  8BF0               mov esi,eax
0100261B  FF15 74100001      call dword ptr ds:[<&lstrlenW>]            lstrlenW
01002621  FF7424 10          push dword ptr ss:[esp+10]                 LPCTSTR lpString
01002625  8BF8               mov edi,eax
01002627  56                 push esi                                   HWND hwnd
01002628  FF15 10110001      call dword ptr ds:[<&SetWindowTextW>]      SetWindowTextW
0100262E  56                 push esi                                   HWND hwnd
0100262F  FF15 14110001      call dword ptr ds:[<&SetFocus>]            SetFocus
01002635  57                 push edi                                   LPARAM lParam = "SciCalc"
01002636  57                 push edi                                   WPARAM wParam = "SciCalc"
01002637  68 B1000000        push B1                                    UINT Msg = EM_SETSEL
```

```
EAX  00000003
EBX  00000000
ECX  74FA71BA
EDX  0006F7E2
EBP  0006F9E0
ESP  0006F7C0
ESI  006004F6
EDI  00000003

EIP  01002628

EFLAGS  00000306
ZF 0  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1
```

Figure 18.9: *Breakpoint hit when = is pressed*

So, we will pass the execution by pressing **Run** again. This time when the breakpoint is hit, we can see that the result of 2+6, which is is pushed on the stack.



```
01002601   C2 0400         ret 4
01002604   56              push esi
01002605   57              push edi
01002606   68 93010000     push 193                              int nIDDlgItem = 193
01002608   FF7424 10       push dword ptr ss:[esp+10]            HWND hDlg
0100260F   FF15 70110001   call dword ptr ds:[<&GetDlgItem>]     GetDlgItem
01002615   FF7424 10       push dword ptr ss:[esp+10]            LPCSTR lpString
01002619   8BF0            mov esi,eax
0100261B   FF15 74100001   call dword ptr ds:[<&lstrlenW>]       lstrlenW
01002621   FF7424 10       push dword ptr ss:[esp+10]            LPCTSTR lpString
01002625   8BF8            mov edi,eax
01002627   56              push esi                              HWND hwnd
EIP 01002628   FF15 10110001   call dword ptr ds:[<&SetWindowTextW>] SetWindowTextW
0100262E   56              push esi                              HWND hwnd
0100262F   FF15 14110001   call dword ptr ds:[<&SetFocus>]       SetFocus
01002635   57              push edi                              LPARAM lParam = "SciCalc'
01002636   57              push edi                              WPARAM wParam = "SciCalc'
01002637   68 81000000     push 81                               UINT Msg = EM_SETSEL
```

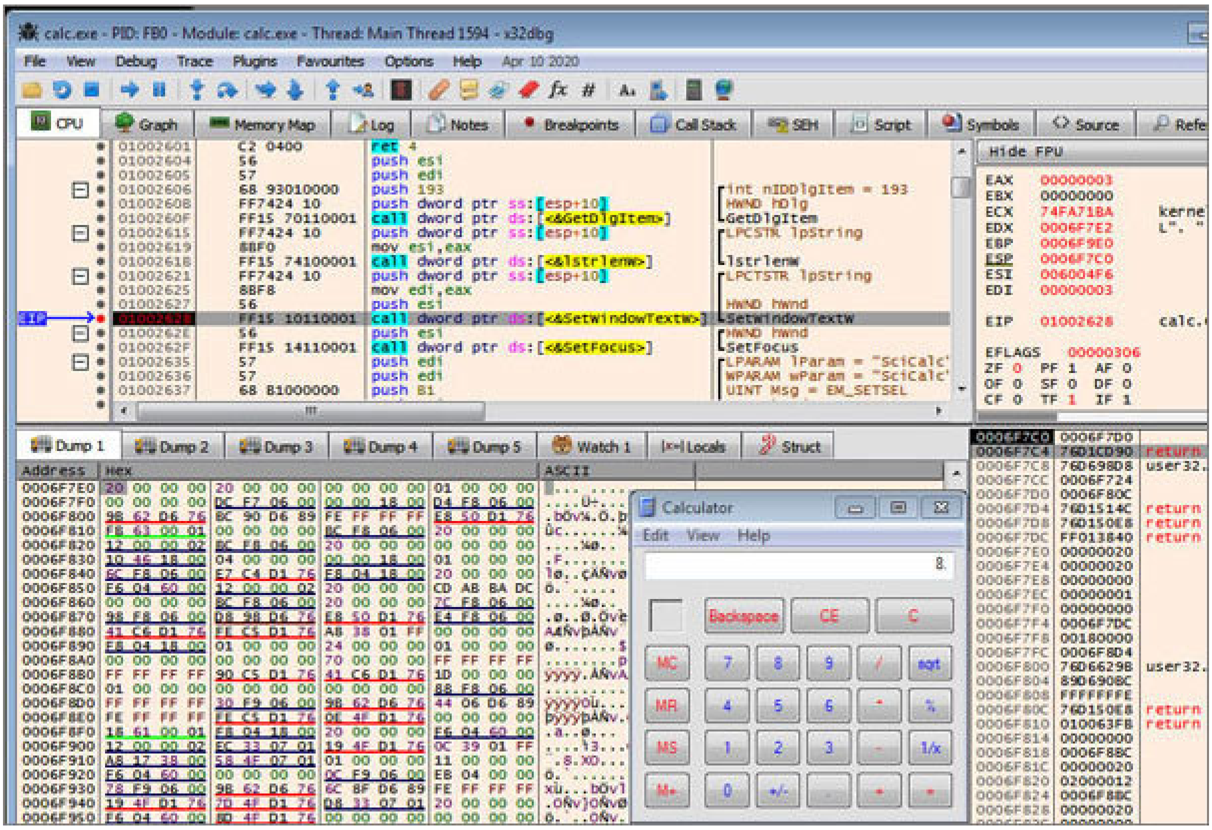
```
Hide FPU
EAX     00000003
EBX     00000000
ECX     74FA718A      kerne
EDX     0006F7E2      L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628      calc.

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

```
0006F7C0  006004F6
0006F7C4  0006F7E0  L"8. "
0006F7C8  001CC3C0
0006F7CC  001D9CE8
0006F7D0  01004900  return
0006F7D4  001804F8
0006F7D8  0006F7E0  L"8. "
0006F7DC  00000000
0006F7E0  002E0038
0006F7E4  00000020
0006F7E8  00000000
0006F7EC  001C0000
0006F7F0  001C0000
0006F7F4  00000000
0006F7F8  001CC3B0
0006F7FC  011D9888
0006F800  0006F7E0  L"8. "
0006F804  0006F7E4
0006F808  0006F8EC
0006F80C  76DEE0ED  ntdll.
0006F810  0E16AF91
0006F814  FFFFFFFE
0006F818  76E960FE  return
0006F81C  76E5A376  return
0006F820  001C0000
0006F824  50140168
0006F828  76E25DD3  return
0006F82C  78F24AE5
```

Figure 18.10: *Sum 8 is pushed on the stack*

Again, **0x0006F7E0** is the memory location of the evaluated result, which is By dumping **0x0006F7E0** in the memory dump, we can observe that the result is stored in the Unicode format.

On the next **Run** in x32dbg, our evaluated value **8** is displayed on the calculator.



```
01002601   C2 0400          ret 4
01002604   56               push esi
01002605   57               push edi
01002606   68 93010000      push 193                                      int nIDDlgItem = 193
01002608   FF7424 10        push dword ptr ss:[esp+10]                    HWND hDlg
0100260F   FF15 70110001    call dword ptr ds:[<&GetDlgItem>]             GetDlgItem
01002615   FF7424 10        push dword ptr ss:[esp+10]                    LPCSTR lpString
01002619   8BF0             mov esi,eax
0100261B   FF15 74100001    call dword ptr ds:[<&lstrlenW>]               lstrlenW
01002621   FF7424 10        push dword ptr ss:[esp+10]                    LPCTSTR lpString
01002625   8BF8             mov edi,eax
01002627   56               push esi                                      HWND hwnd
01002628   FF15 10110001    call dword ptr ds:[<&SetWindowTextW>]         SetWindowTextW
0100262E   56               push esi                                      HWND hwnd
0100262F   FF15 14110001    call dword ptr ds:[<&SetFocus>]               SetFocus
01002635   57               push edi                                      LPARAM lParam = "SciCalc'
01002636   57               push edi                                      WPARAM wParam = "SciCalc'
01002637   68 B1000000      push B1                                       UINT Msg = EM_SETSEL
```

```
EAX   00000003
EBX   00000000
ECX   74FA71BA      kerne
EDX   0006F7E2      L". "
EBP   0006F9E0
ESP   0006F7C0
ESI   006004F6
EDI   00000003

EIP   01002628      calc.

EFLAGS   00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1
```

```
Address   Hex
0006F7E0  20 00 00 00  20 00 00 00  00 00 00 00  01 00 00 00
0006F7F0  00 00 00 00  DC F7 06 00  00 00 18 00  D4 F8 06 00
0006F800  98 62 D6 76  BC 90 D6 89  FE FF FF FF  E8 50 D1 76
0006F810  F8 63 00 01  00 00 00 00  BC F8 06 00  20 00 00 00
0006F820  12 00 00 02  BC F8 06 00  20 00 00 00  00 00 00 00
0006F830  10 46 18 00  04 00 00 00  00 00 18 00  01 00 00 00
0006F840  6C F8 06 00  E7 C4 D1 76  E8 04 18 00  20 00 00 00
0006F850  F6 04 60 00  12 00 00 02  20 00 00 00  CD AB BA DC
0006F860  00 00 00 00  BC F8 06 00  20 00 00 00  7C F8 06 00
0006F870  98 F8 06 00  D8 98 D6 76  E8 50 D1 76  E4 F8 06 00
0006F880  41 C6 D1 76  FE C5 D1 76  A8 38 01 FF  00 00 00 00
0006F890  E8 04 18 00  01 00 00 00  24 00 00 00  01 00 00 00
0006F8A0  00 00 00 00  00 00 00 00  70 00 00 00  FF FF FF FF
0006F8B0  FF FF FF FF  90 C5 D1 76  41 C6 D1 76  1D 00 00 00
0006F8C0  01 00 00 00  00 00 00 00  00 00 00 00  88 F8 06 00
0006F8D0  FF FF FF FF  30 F9 06 00  98 62 D6 76  44 06 D6 89
0006F8E0  FE FF FF FF  FE C5 D1 76  0E 4F D1 76  00 00 00 00
0006F8F0  18 61 00 01  E8 04 18 00  20 00 00 00  F6 04 60 00
0006F900  12 00 00 02  EC 33 07 01  19 4F D1 76  0C 39 01 FF
0006F910  A8 17 38 00  58 4F 07 01  01 00 00 00  11 00 00 00
0006F920  F6 04 60 00  00 00 00 00  0C F9 06 00  EB 04 00 00
0006F930  78 F9 06 00  98 62 D6 76  6C 8F D6 89  FE FF FF FF
0006F940  19 4F D1 76  7D 4F D1 76  D8 33 07 01  20 00 00 00
0006F950  F6 04 60 00  8D 4F D1 76  00 00 00 00  00 00 00 00
```

```
0006F7C0  0006F7D0
0006F7C4  76D1CD90  return
0006F7C8  76D698D8  user32.
0006F7CC  0006F724
0006F7D0  0006F80C
0006F7D4  76D1514C  return
0006F7D8  76D150E8  return
0006F7DC  FF013840  return
0006F7E0  00000020
0006F7E4  00000020
0006F7E8  00000000
0006F7EC  00000001
0006F7F0  00000000
0006F7F4  0006F7DC
0006F7F8  00180000
0006F7FC  0006F8D4
0006F800  76D6629B  user32.
0006F804  89D6908C
0006F808  FFFFFFFE
0006F80C  76D150E8  return
0006F810  010063FB  return
0006F814  00000000
0006F818  0006F88C
0006F81C  00000020
0006F820  02000012
0006F824  0006F88C
0006F828  00000020
```

Figure 18.11: *Sum 8 is displayed on the calculator*

We can observe that every time we press any digit on the calculator, the **SetWindowTextW** function is called, which pushes the memory location of the digit on the stack. But we were not able to find any identifier or differential flow till that point to help us identify when the user pressed the **equal to** button on the calculator.

Finding a placeholder to call our code

The objective of this step is to identify a condition where we can differentiate between the press of the **equal to** button on the calculator and the press of another digit or a button on the calculator. We have to be able to find that differential flow or some register value to help us identify the press of the **equal to** button. Using that as a condition or trigger, we can jump to our written code to flash our defined string on the calculator when the **equal to** button is pressed.

To find that differentiating parameter, we will follow a simple process, wherein we will note down the value of all the registers and the stack at the press of any button on the calculator. As we have the x32dbg debugger attached to our calculator, we will follow the given steps:

First press **2** on the calculator to hit the breakpoint. When the execution is paused at the breakpoint, note the register value and the stack.

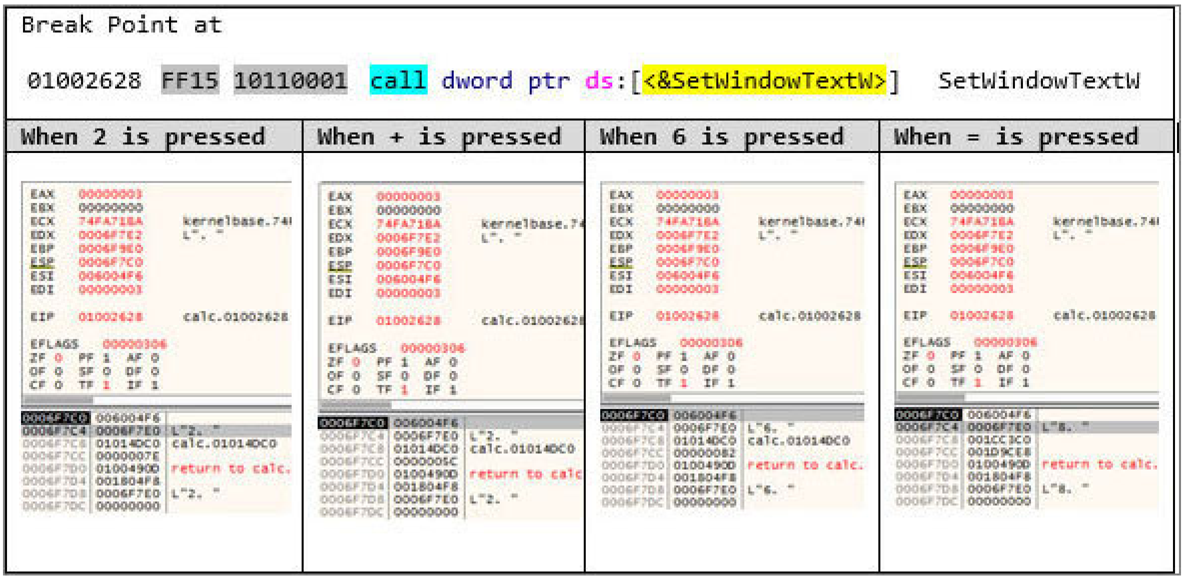
Now press **Run** in x32dbg to return the control to the calculator. Press the **plus (+)** button to hit breakpoint again and note the register value and the stack.

Press **Run** in x32dbg again and then press **6** to hit breakpoint to note the register value and the stack.

Press **Run** again in x32dbg to return the control to the calculator. Then press **equal to (=)** to hit the breakpoint. At this point, you will see that the calculated value of **8** is not pushed on the stack. Press **Run** in x32dbg again and note the registers' values and the stack.

Now if you press **Run** in x32dbg again, it will update the calculator with the calculated value, which is 2+6 = 8.

For your better understanding, we have placed the output of the preceding steps in a table format. This will also help us analyze all the registers and the stack to find any differentiating parameter to help us identify the press of the **equal to** button on the calculator.



```
Break Point at

01002628 FF15 10110001 call dword ptr ds:[<&SetWindowTextW>]   SetWindowTextW
```

When 2 is pressed

```
EAX     00000003
EBX     00000000
ECX     74FA718A        kernelbase.74
EDX     0006F7E2        L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628        calc.01002628

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

0006F7C0 006004F6
0006F7C4 0006F7E0 L"2. "
0006F7C8 01014DC0 calc.01014DC0
0006F7CC 0000007E
0006F7D0 0100490D return to calc.
0006F7D4 001804F8
0006F7D8 0006F7E0 L"2. "
0006F7DC 00000000
```

When + is pressed

```
EAX     00000003
EBX     00000000
ECX     74FA718A        kernelbase.7
EDX     0006F7E2        L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628        calc.0100262

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

0006F7C0 006004F6
0006F7C4 0006F7E0 L"2. "
0006F7C8 01014DC0 calc.01014DC0
0006F7CC 0000005C
0006F7D0 0100490D return to calc
0006F7D4 001804F8
0006F7D8 0006F7E0 L"2. "
0006F7DC 00000000
```

When 6 is pressed

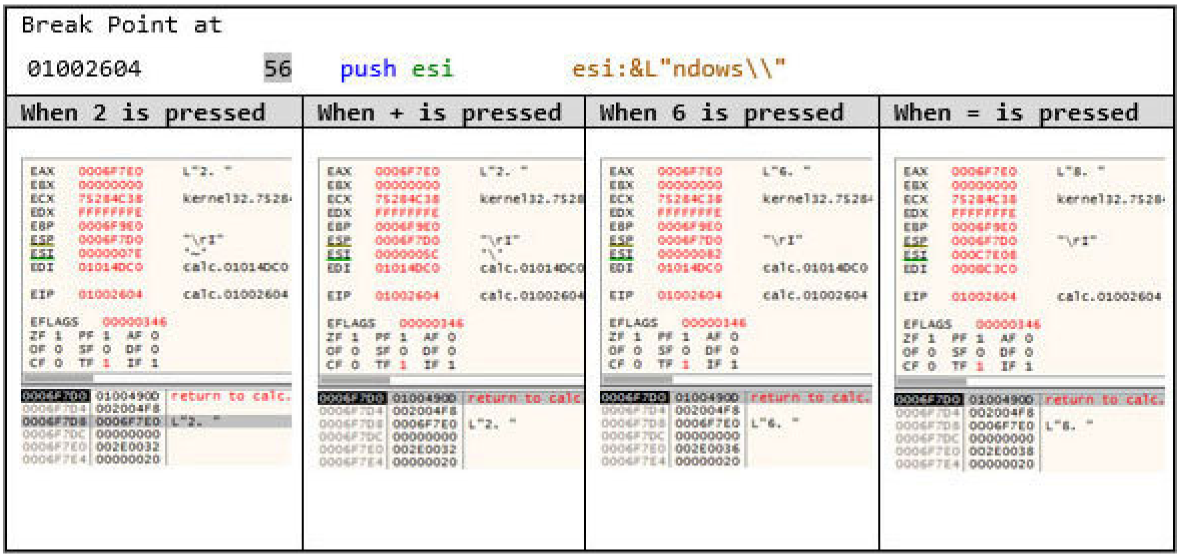
```
EAX     00000003
EBX     00000000
ECX     74FA718A        kernelbase.74
EDX     0006F7E2        L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628        calc.01002628

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

0006F7C0 006004F6
0006F7C4 0006F7E0 L"6. "
0006F7C8 01014DC0 calc.01014DC0
0006F7CC 00000082
0006F7D0 0100490D return to calc.
0006F7D4 001804F8
0006F7D8 0006F7E0 L"6. "
0006F7DC 00000000
```

When = is pressed

```
EAX     00000003
EBX     00000000
ECX     74FA718A        kernelbase.74
EDX     0006F7E2        L". "
EBP     0006F9E0
ESP     0006F7C0
ESI     006004F6
EDI     00000003

EIP     01002628        calc.01002628

EFLAGS    00000306
ZF 0   PF 1   AF 0
OF 0   SF 0   DF 0
CF 0   TF 1   IF 1

0006F7C0 006004F6
0006F7C4 0006F7E0 L"8. "
0006F7C8 001CC3C0
0006F7CC 001D9CE8
0006F7D0 0100490D return to calc.
0006F7D4 001804F8
0006F7D8 0006F7E0 L"8. "
0006F7DC 00000000
```

***Table 18.1:** Comparison of registers and the stack when BP at CALL*

As we can see from the table, the register values are the same when the different keys are pressed on the calculator. This is not leading us to find any differentiating parameter to determine whether the user pressed the **equal to** button or some other digit. Let's move our breakpoint position to the start of the procedure and see if we can find something there to differentiate between the keys pressed on the calculator. Set the breakpoint at the start of procedure and follow the same debugging process to calculate 2+6 on the calculator.



Break Point at

```
01002604        56      push esi          esi:&L"ndows\\"
```

When 2 is pressed

```
EAX   0006F7E0      L"2. "
EBX   00000000
ECX   75284C38      kernel32.7528
EDX   FFFFFFFE
EBP   0006F9E0
ESP   0006F7D0      "\rI"
ESI   0000007E      "~"
EDI   01014DC0      calc.01014DC0

EIP   01002604      calc.01002604

EFLAGS   00000346
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

0006F7D0  01004900  return to calc.
0006F7D4  002004F8
0006F7D8  0006F7E0  L"2. "
0006F7DC  00000000
0006F7E0  002E0032
0006F7E4  00000020
```

When + is pressed

```
EAX   0006F7E0      L"2. "
EBX   00000000
ECX   75284C38      kernel32.7528
EDX   FFFFFFFE
EBP   0006F9E0
ESP   0006F7D0      "\rI"
ESI   0000005C      "\"
EDI   01014DC0      calc.01014DC0

EIP   01002604      calc.01002604

EFLAGS   00000346
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

0006F7D0  01004900  return to calc
0006F7D4  002004F8
0006F7D8  0006F7E0  L"2. "
0006F7DC  00000000
0006F7E0  002E0032
0006F7E4  00000020
```

When 6 is pressed

```
EAX   0006F7E0      L"6. "
EBX   00000000
ECX   75284C38      kernel32.7528
EDX   FFFFFFFE
EBP   0006F9E0
ESP   0006F7D0      "\rI"
ESI   00000082
EDI   01014DC0      calc.01014DC0

EIP   01002604      calc.01002604

EFLAGS   00000346
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

0006F7D0  01004900  return to calc.
0006F7D4  002004F8
0006F7D8  0006F7E0  L"6. "
0006F7DC  00000000
0006F7E0  002E0036
0006F7E4  00000020
```

When = is pressed

```
EAX   0006F7E0      L"8. "
EBX   00000000
ECX   75284C38      kernel32.7528
EDX   FFFFFFFE
EBP   0006F9E0
ESP   0006F7D0      "\rI"
ESI   000C7E08
EDI   000BC3C0

EIP   01002604      calc.01002604

EFLAGS   00000346
ZF 1  PF 1  AF 0
OF 0  SF 0  DF 0
CF 0  TF 1  IF 1

0006F7D0  01004900  return to calc.
0006F7D4  002004F8
0006F7D8  0006F7E0  L"6. "
0006F7DC  00000000
0006F7E0  002E0038
0006F7E4  00000020
```

***Table 18.2:** Comparison of the registers and the stack when BP at the PROC start*

In this exercise, we can observe two points:

First, **EAX** is holding the memory location of the evaluated result (that is 2 + 6 = 8) and we know that **EAX** holds the return value of the caller function.

Second, the **ESI** register can be of our interest. When **2** or **plus (+)** or **6** is pressed on the calculator, the value of **ESI** is always lower than let's say about 1000 (in decimal) (0x3E8 in hex). But when the **equal to** button is pressed on the calculator, the **ESI** value is greater than 1000 (in decimal) (0x3E8 in hex). This condition will help us differentiate between the other buttons that are pressed and when the **equal to** button is pressed on the calculator.

So, we can use this as a triggering condition to jump to our code and to automate this whole process of checking when the **equal to** button is pressed on the calculator. The pseudo code of this can be as follows:

```
If (ESI ≤ 0x3E8)   // 2 or plus (+) or 6 button is pressed on the calculator

{
    Continue execution normally
} else      // equal to button is pressed on the calculator
{
    Run our code to print text on calculator screen
}
```

Now, we will move on to write our code. But the big question is, **where are we going to write our code?**

The answer to the question is that we will write our code in the Code Cave. But first, we will understand what code cave is in the following section.

Writing our code in the Code Cave

To modify any application or add some functionality to any application, we need to have the source code. Having a source code and modifying it is the simplest solution to add some functionality to the application. But what if we don't have the source code?

In that case, we will have to modify the application binary and add our code to it. For adding our code to the compiled application or binary, we will use the code cave.

For adding the code, we will have to find an unused area in the compiled application or exe file. This unused space will be our We will insert our custom code in this cave. Finally, somewhere in the original binary, we will add jump to our code cave so that it is executed along with the original binary execution. At the end, to return the execution from our code cave to the original binary, we will add return to our code cave. This process is used by many malware writers to add custom code to the complied application.

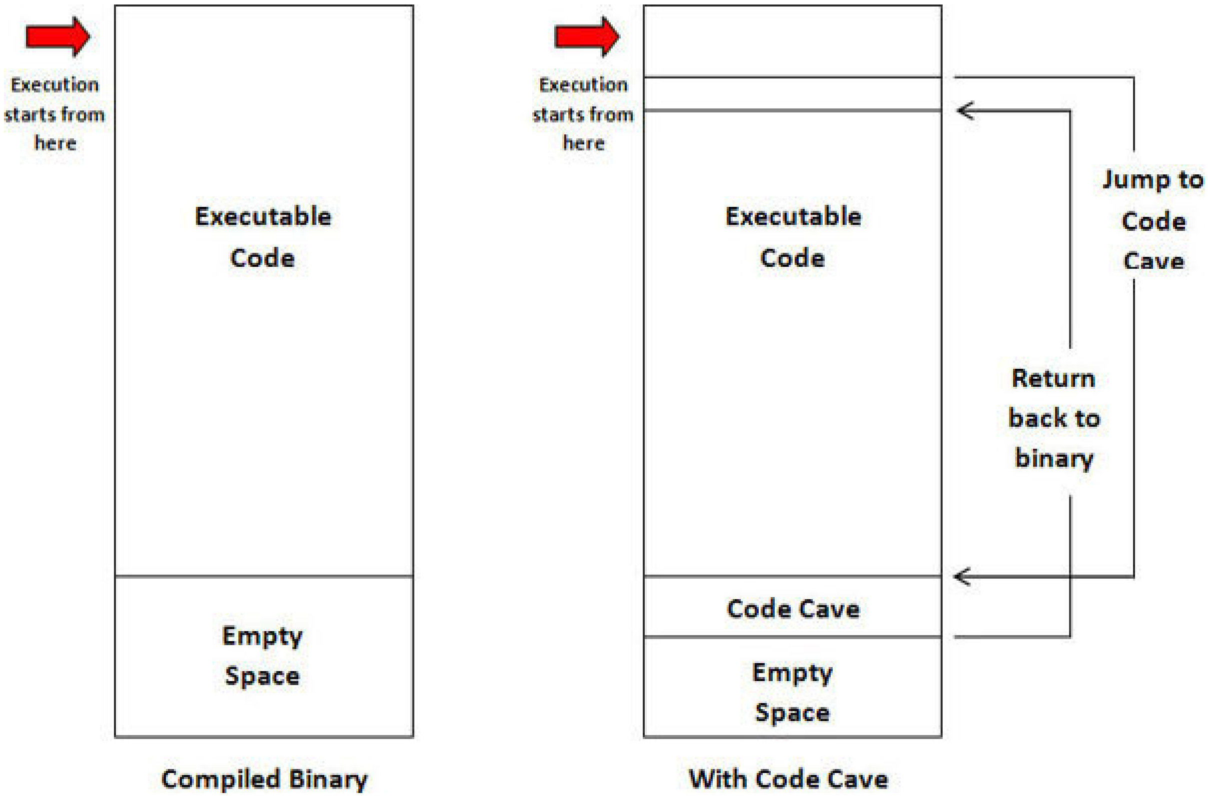


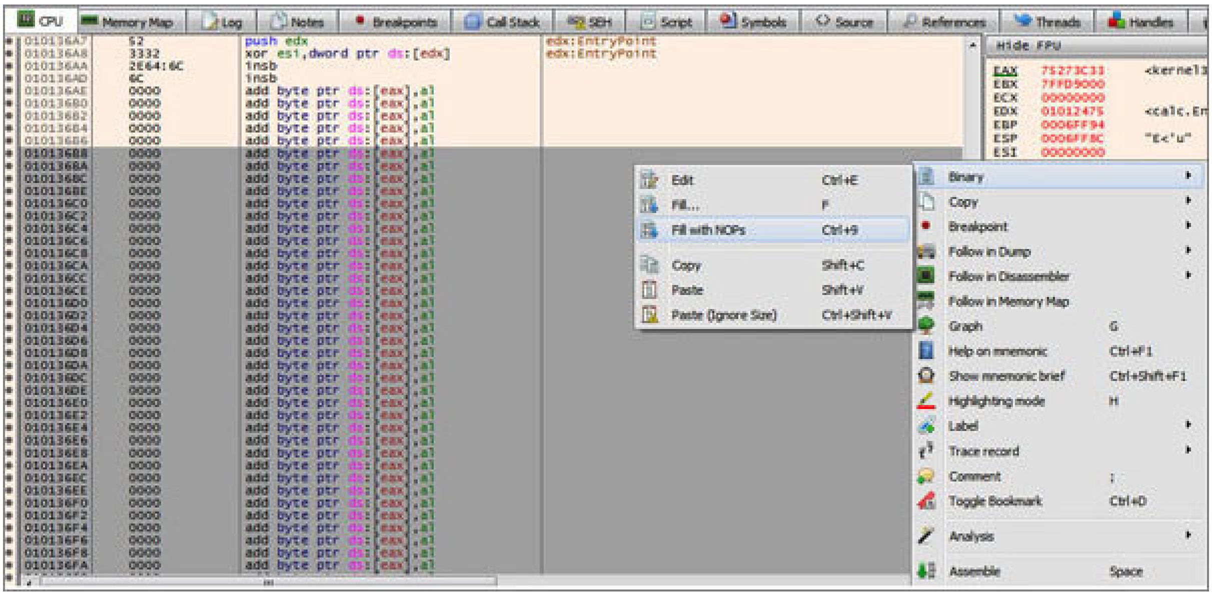
Figure 18.12: *Code cave concept*

Sometimes, an empty space is not enough to write a big code. In such cases, a new section can be added to the binary with executable privileges. Then we can jump to that section and return back. This technique is not covered here, so let's move on to code cave.

While writing a code in the cave, we always have to remember that every byte counts. The size of the code should be as small as possible. To find the space available for us, let's walk through the compiled binary of the calculator and based on that, we will take a decision.

Compiled application, exe, portable exe, binary – All these mean the same.

As we move towards the end of the calculator executable, we see enough space to write our code (shown in the following screenshot). We will select some sections of the space and fill it with the **NOP** instruction. This section can be referred to as the cave.

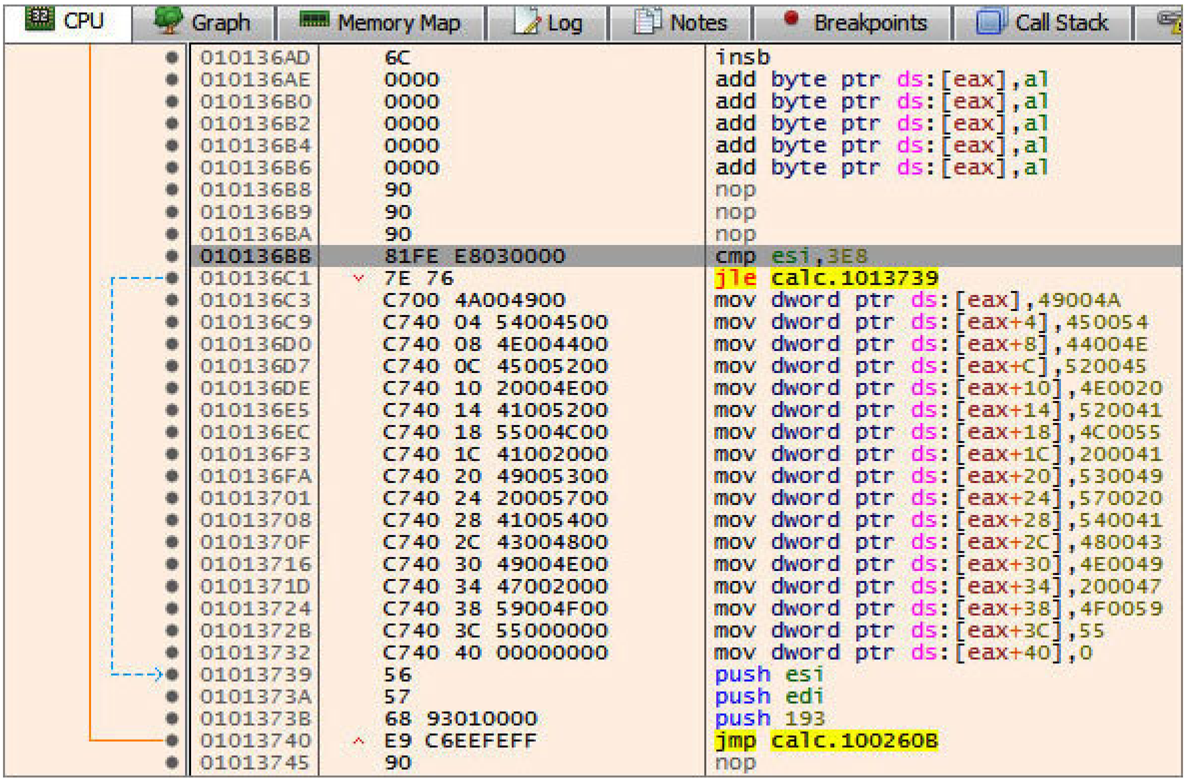


```
010136A7   52        push edx                          edx:EntryPoint
010136A8   3332      xor esi,dword ptr ds:[edx]        edx:EntryPoint
010136AA   2E64:6C   insb
010136AD   6C        insb
010136AE   0000      add byte ptr ds:[eax],al
010136B0   0000      add byte ptr ds:[eax],al
010136B2   0000      add byte ptr ds:[eax],al
010136B4   0000      add byte ptr ds:[eax],al
010136B6   0000      add byte ptr ds:[eax],al
010136B8   0000      add byte ptr ds:[eax],al
010136BA   0000      add byte ptr ds:[eax],al
010136BC   0000      add byte ptr ds:[eax],al
010136BE   0000      add byte ptr ds:[eax],al
010136C0   0000      add byte ptr ds:[eax],al
010136C2   0000      add byte ptr ds:[eax],al
010136C4   0000      add byte ptr ds:[eax],al
010136C6   0000      add byte ptr ds:[eax],al
010136C8   0000      add byte ptr ds:[eax],al
010136CA   0000      add byte ptr ds:[eax],al
010136CC   0000      add byte ptr ds:[eax],al
010136CE   0000      add byte ptr ds:[eax],al
010136D0   0000      add byte ptr ds:[eax],al
010136D2   0000      add byte ptr ds:[eax],al
010136D4   0000      add byte ptr ds:[eax],al
010136D6   0000      add byte ptr ds:[eax],al
010136D8   0000      add byte ptr ds:[eax],al
010136DA   0000      add byte ptr ds:[eax],al
010136DC   0000      add byte ptr ds:[eax],al
010136DE   0000      add byte ptr ds:[eax],al
010136E0   0000      add byte ptr ds:[eax],al
010136E2   0000      add byte ptr ds:[eax],al
010136E4   0000      add byte ptr ds:[eax],al
010136E6   0000      add byte ptr ds:[eax],al
010136E8   0000      add byte ptr ds:[eax],al
010136EA   0000      add byte ptr ds:[eax],al
010136EC   0000      add byte ptr ds:[eax],al
010136EE   0000      add byte ptr ds:[eax],al
010136F0   0000      add byte ptr ds:[eax],al
010136F2   0000      add byte ptr ds:[eax],al
010136F4   0000      add byte ptr ds:[eax],al
010136F6   0000      add byte ptr ds:[eax],al
010136F8   0000      add byte ptr ds:[eax],al
010136FA   0000      add byte ptr ds:[eax],al
```

```
EAX   75273C33     <kernel3
EBX   7FFD9000
ECX   00000000
EDX   01012475     <calc.En
EBP   0006FF94
ESP   0006FF8C     "E<'u"
ESI   00000000
```

__Figure 18.13:__ Code cave with NOPs

After filling the cave with the we can start writing our code. The objective of this code is to print **JITENDER NARULA IS WATCHING YOU** on the calculator when somebody performs some calculation and presses the **equal to** button for the result. This output text can be customized to anything of your choice. The code that goes into the cave is:



```
010136AD    6C                     insb
010136AE    0000                   add byte ptr ds:[eax],al
010136B0    0000                   add byte ptr ds:[eax],al
010136B2    0000                   add byte ptr ds:[eax],al
010136B4    0000                   add byte ptr ds:[eax],al
010136B6    0000                   add byte ptr ds:[eax],al
010136B8    90                     nop
010136B9    90                     nop
010136BA    90                     nop
010136BB    81FE E8030000          cmp esi,3E8
010136C1  v 7E 76                  jle calc.1013739
010136C3    C700 4A004900          mov dword ptr ds:[eax],49004A
010136C9    C740 04 54004500       mov dword ptr ds:[eax+4],450054
010136D0    C740 08 4E004400       mov dword ptr ds:[eax+8],44004E
010136D7    C740 0C 45005200       mov dword ptr ds:[eax+C],520045
010136DE    C740 10 20004E00       mov dword ptr ds:[eax+10],4E0020
010136E5    C740 14 41005200       mov dword ptr ds:[eax+14],520041
010136EC    C740 18 55004C00       mov dword ptr ds:[eax+18],4C0055
010136F3    C740 1C 41002000       mov dword ptr ds:[eax+1C],200041
010136FA    C740 20 49005300       mov dword ptr ds:[eax+20],530049
01013701    C740 24 20005700       mov dword ptr ds:[eax+24],570020
01013708    C740 28 41005400       mov dword ptr ds:[eax+28],540041
0101370F    C740 2C 43004800       mov dword ptr ds:[eax+2C],480043
01013716    C740 30 49004E00       mov dword ptr ds:[eax+30],4E0049
0101371D    C740 34 47002000       mov dword ptr ds:[eax+34],200047
01013724    C740 38 59004F00       mov dword ptr ds:[eax+38],4F0059
0101372B    C740 3C 55000000       mov dword ptr ds:[eax+3C],55
01013732    C740 40 00000000       mov dword ptr ds:[eax+40],0
01013739    56                     push esi
0101373A    57                     push edi
0101373B    68 93010000            push 193
01013740  ^ E9 C6EEFEFF            jmp calc.100260B
01013745    90                     nop
```

Figure 18.14: Code cave

The code has been explained as follows:

follows: follows:

follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

follows: follows: follows: follows: follows: follows: follows: follows:

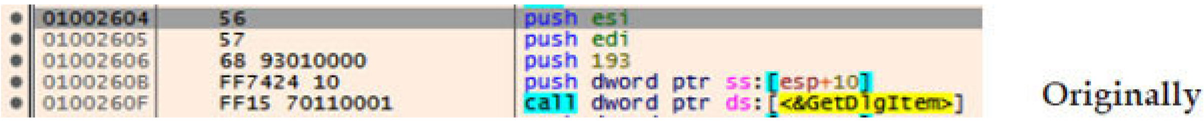
follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:

follows: follows: follows: follows:

follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows: follows: follows:
follows: follows:
follows: follows:
follows: follows:
follows: follows: follows: follows: follows: follows: follows: follows:

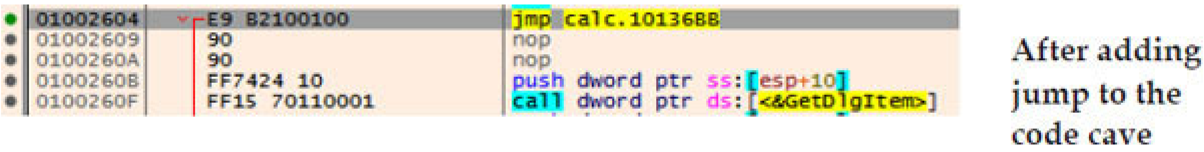
Table 18.3: *Code cave explained instruction by instruction*

Now we have written our code in the cave. To call this code during the calculator execution, we have to modify the execution flow of the executable as explained in *figure* To do this, we can either use the **CALL** instruction or the **JMP** instruction. We can use the **CALL** instruction to call our code but since we don't have any space constraints, we will use the basic **JMP** instruction for easy understanding. Follow below screens to modify executable code flow at the start of the procedure where we inserted our breakpoint for the comparison of registers and the stack to get the triggering condition in *table*



```
01002604  56              push esi
01002605  57              push edi
01002606  68 93010000     push 193
0100260B  FF7424 10       push dword ptr ss:[esp+10]
0100260F  FF15 70110001   call dword ptr ds:[<&GetDlgItem>]
```

Figure 18.15: Original executable



```
01002604  E9 B2100100     jmp calc.10136BB
01002609  90              nop
0100260A  90              nop
0100260B  FF7424 10       push dword ptr ss:[esp+10]
0100260F  FF15 70110001   call dword ptr ds:[<&GetDlgItem>]
```

Figure 18.16: After adding jump to Code Cave

The **JMP** instruction is pointing to the code cave and the instructions occupied by **NOP** are called in the code cave. Now we are done with our code insertion in the compiled binary. We will now patch this binary to save the changes in the calculator exe.

Patching the binary

Patching will help us in writing our code in the compiled calculator application on the disk. To patch, go to **File -> Patch Select All to Patch** Specify the path & filename to save the file on the disk.

Now, by performing any calculation on our patched calculator binary, the following result will be produced on pressing the **equal to** button. For example, if we press 2+6 and then the **equal to** button, we will get the following result:

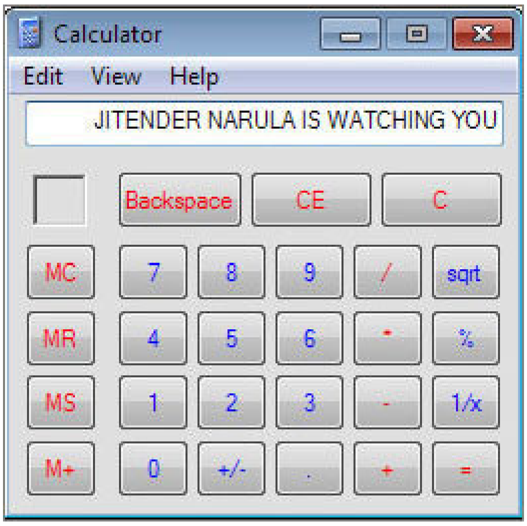


Figure 18.17: *Patched calculator*

Placing the breakpoint in our code cave shows that we have overwritten the calculator result in the memory with our custom text **JITENDER NARULA IS WATCHING**

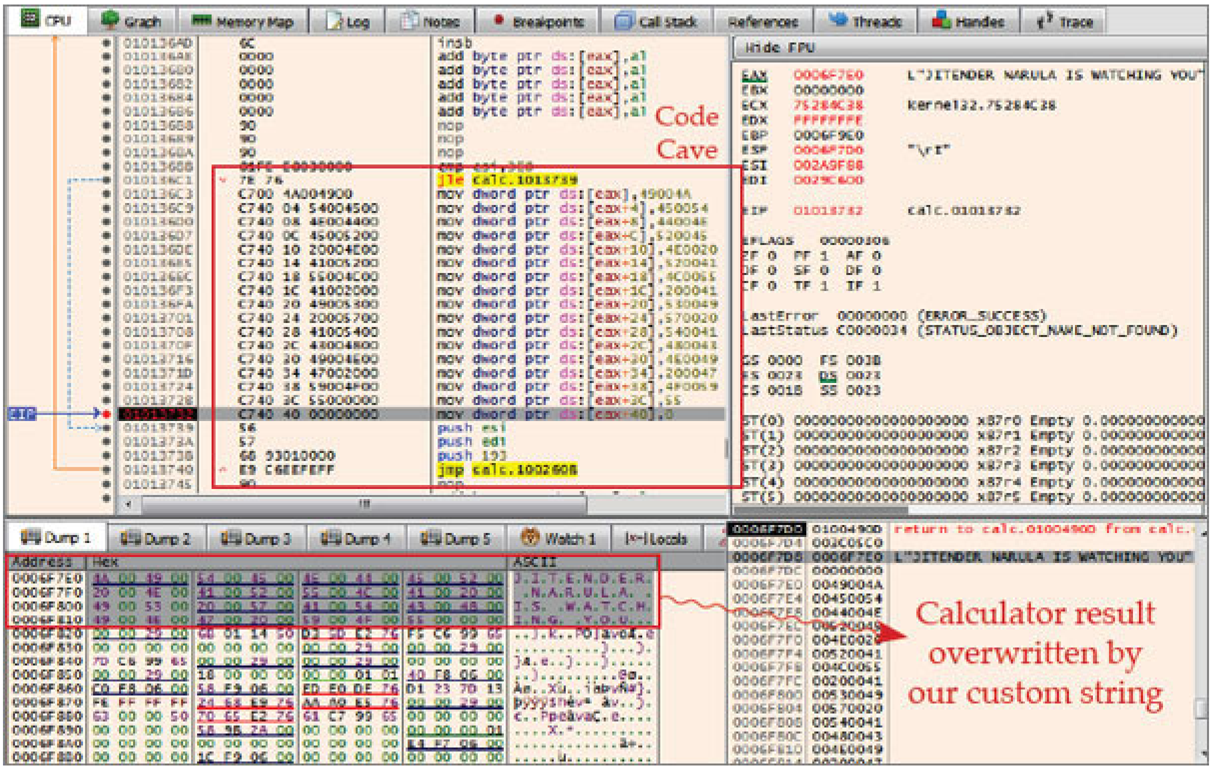


Figure 18.18: Patched calculator in x32dbg

Conclusion

In this chapter, we learned to reverse engineer the calculator by modifying its behaviour to output our defined string for any calculation. This means that rather than getting **8** as an output to 2+6 or any other calculation, the calculator will display our defined message on pressing the **equal to** button. We also learned the concept of finding a cave in the closed binary and writing a code cave to change the execution path of the binary. Further, we learned how to patch the binary to permanently write the changes on the disk.

Appendix

Macro

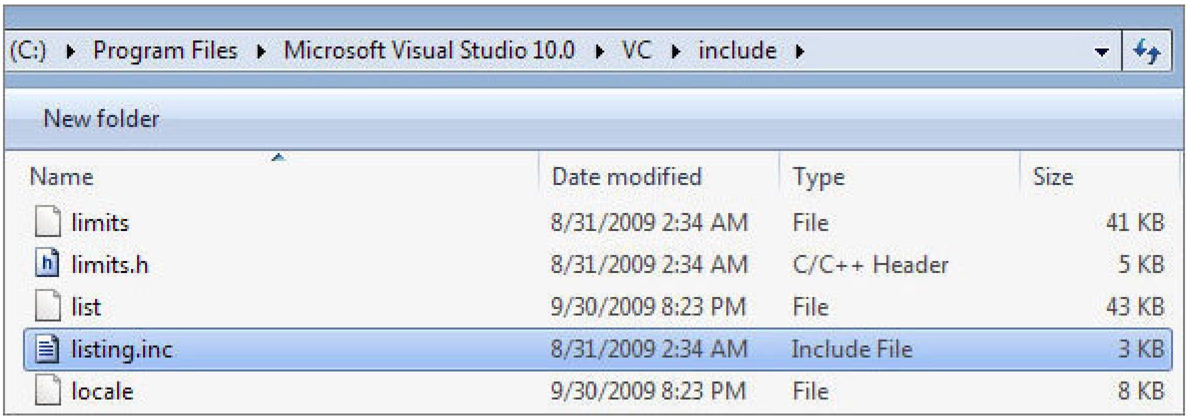
Macros in the assemble language are used to write modular programs. Macros are the sequence of instruction, assigned by name. They can be used anywhere in the code. The macros make programs shorter and readable.

Procedure

A procedure or subroutine is a sequence of instructions that perform certain tasks. They are very important in assembly language. They are identified by name and the end of the procedure is identified by RET statement.

npad

The npad is a macro defined in the **listing.inc** which resides in **Root_Folder\VC\include** folder of MSVC. npad macro inserts non-destructive and non-operational instructions, rather than a series of NOP instruction. While doing optimization the compiler insert non-operational instruction for enforcing alignment of the data. A series of NOP's can also be used, but single instructions are always good for better CPU performance.



Name	Date modified	Type	Size
limits	8/31/2009 2:34 AM	File	41 KB
limits.h	8/31/2009 2:34 AM	C/C++ Header	5 KB
list	9/30/2009 8:23 PM	File	43 KB
listing.inc	8/31/2009 2:34 AM	Include File	3 KB
locale	9/30/2009 8:23 PM	File	8 KB

Figure A.1: *Listing.inc file path*

npad macro is accompanied by a number (X). Where number X can be from 1 to 15, this number defines the amount of memory alignment or padding required by the compiler during optimization. Compiler pads the extra space between the previous instruction/data and the one after npad macro.

If we look into

npad 1 defines padding of 1 byte **NOP**

npad 2 defines padding of 2 bytes with **mov edi, edi** instruction.

npad 2 defines padding of 3 bytes with **lea ecx, [ecx+00]** instruction, so on

All these are basically different variations of NOP, which have no impact on the code flow.

LISTING.INC for reference:

```
; LISTING.INC
;
; This file contains assembler macros and is included by the files created
; with the -FA compiler switch to be assembled by MASM (Microsoft Macro
; Assembler).
;
; Copyright (c) 1993-2003, Microsoft Corporation. All rights reserved.

; non-destructive nops
npad macro size
if size eq 1
nop
else
```

```
if size eq 2
mov edi, edi
else
if size eq 3
; lea ecx, [ecx+00]
DB 8DH, 49H, 00H
else
if size eq 4
; lea esp, [esp+00]
DB 8DH, 64H, 24H, 00H
else
if size eq 5
add eax, DWORD PTR 0
else

if size eq 6
; lea ebx, [ebx+00000000]
DB 8DH, 9BH, 00H, 00H, 00H, 00H
else
if size eq 7
; lea esp, [esp+00000000]
DB 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
else
if size eq 8
; jmp .+8; .npad 6
DB 0EBH, 06H, 8DH, 9BH, 00H, 00H, 00H, 00H
else
if size eq 9
; jmp .+9; .npad 7
DB 0EBH, 07H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H
else
if size eq 10
```

```
; jmp .+A; .npad 7; .npad 1
DB 0EBH, 08H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 90H
else
if size eq 11
; jmp .+B; .npad 7; .npad 2
DB 0EBH, 09H, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8BH,
0FFH
else
if size eq 12
; jmp .+C; .npad 7; .npad 3

DB 0EBH, 0AH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH,
49H, 00H
else
if size eq 13
; jmp .+D; .npad 7; .npad 4
DB 0EBH, 0BH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH,
64H, 24H, 00H
else
if size eq 14
; jmp .+E; .npad 7; .npad 5
DB 0EBH, 0CH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 05H,
00H, 00H, 00H, 00H
else
if size eq 15
; jmp .+F; .npad 7; .npad 6
DB 0EBH, 0DH, 8DH, 0A4H, 24H, 00H, 00H, 00H, 00H, 8DH,
9BH, 00H, 00H, 00H, 00H
else
%out error: unsupported npad size
.err
endif
```

```
endif
endif
endif
endif
endif
endif
endif

endif
endif
endif
endif
endif
endif
endif
endm

; destructive nops
dpad macro size, reg
if size eq 1
inc reg
else
%out error: unsupported dpad size
.err
endif
endm
```

LSB and MSB

To understand **least significant bit** and **most significant bit** we will take an example of byte:

0000 0001

In the above byte example, bit in the right end most is set to 1. This is what is referred as the LSB. Now take another example:

1000 0000

In the preceding byte example, bit in the left end most is set to 1. This is what is referred as the MSB.

This same concept goes with the WORD, DWORD and so on.

WORD

(MSB) 1000 0000 0000 0001 (LSB)

DWORD

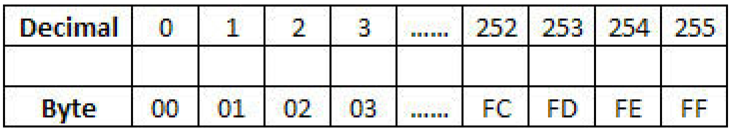
(MSB) 1000 0000 0000 0000 0000 0000 0000 0001 (LSB)

Signed and Unsigned

In mathematics we have positive (1, 2, 3, 4, so on) and negative numbers (-1, -2, -3, -4, so on). Similarly to represent positive and negative number concept in programming, signed and unsigned terms are used.

Unsigned

Data is represented by byte of data, where 00 is the lowest number in byte and FF is the highest number in byte. The positive decimal numbers are represented in bytes as shown below and similarly numbers can also be represented in WORD, DWORD:



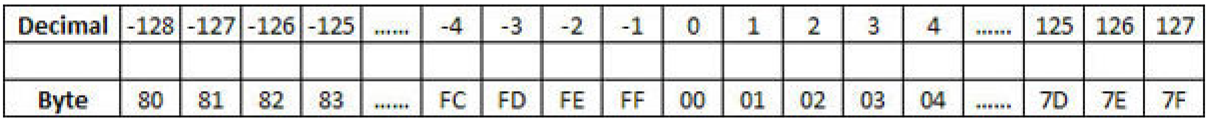
Decimal	0	1	2	3	......	252	253	254	255
Byte	00	01	02	03	......	FC	FD	FE	FF

Figure A.2: *Positive decimal numbers in bytes*

For WORD this range is from 0000 to FFFF and for DWORD it is from 00000000 to FFFFFFFF. These all are referred to as unsigned numbers.

Signed

When data is represented as a signed, then 0x00 to 0x7F is treated as positive and 0x80 to 0xFF is treated as negative numbers:



Decimal	-128	-127	-126	-125	......	-4	-3	-2	-1	0	1	2	3	4	......	125	126	127
Byte	80	81	82	83	......	FC	FD	FE	FF	00	01	02	03	04	......	7D	7E	7F

Figure A.3: *Signed numbers in bytes*

As we can observe MSB of all negative number is set to 1 and for positive numbers MSB is set to 0. Consider -4 and + 4 in decimal, its byte and binary representation is:

+4 = 0x04 = 0000 0100, MSB = 0

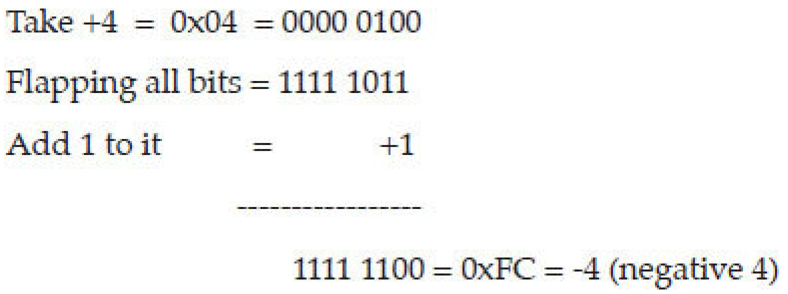
-4 = 0xFC = 1111 1100, MSB = 1

But when the same decimal is represented in WORD, -4 (decimal) will not become 0x00FC. As 0x00FC is not negative, it represents positive number that falls between 0000 and 7FFF. To represent -4 in WORD, we need to extend it as 0xFFFC and in DWORD it is 0xFFFFFFFC.

Now we understood that simply changing the MSB bit will not change the polarity of number. To convert positive number to negative and vice versa, can be done by performing 2’s complement of the number. To calculate 2’s complement of number following process is followed:

Invert all bits in byte or WORD or DWORD

After flapping all bits, add 1 to the number



Take +4 = 0x04 = 0000 0100

Flapping all bits = 1111 1011

Add 1 to it = +1

1111 1100 = 0xFC = -4 (negative 4)

Bit Shifting

To understand bit shifting concept, take 0x44 which in binary is:

0100 0100

In bit shifting, bits are shifted either right or left. When shifted to right 0x44 will become:

?0100 010

When shifted to left, 0x44 will become:

100 0100?

When we shifted to right or left a bit placeholder position is created, denoted by question mark (?) above. Now this question mark can be 0 or 1, which is decided by the type of shifting done .i.e. Logical and Arithmetic shifting.

Logical bit shifting

In logical bit shifting, when bits are shifted to the right, bit placeholder which is question mark is always set to 0. Example:

0100 0100

Logical shift right of 0x44 becomes,

00100 010

When bit shifting is done to left, LSB is always set to 0:

0100 0100

Logical shift left of 0x44 becomes,

100 01000

Arithmetic bit shifting

In Arithmetic bit shifting, when bits are shifted to the right, bit placeholder which is question mark is decided by most significant bit (MSB). Example:

0100 0100

Arithmetic shift right of 0x44 becomes,

00100 010

Take another example:

100 01000

Arithmetic shift right of 0x44 becomes,

1100 0100

When bit shifting is done to left, LSB is always set to 0:

0100 0100

Arithmetic Shift left of 0x44 becomes,

100 01000

ASCII

We all are familiar with the ASCII, where 7 bits are used to represent 128 characters and storing them in 8 bits. Every character occupies 1 byte. Below is the ASCII table for reference:

reference: reference:
reference: reference: reference:
reference: reference: reference:
reference: reference: reference:
reference: reference: reference:
reference:
reference:
reference:
reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference: reference:
reference: reference: reference:
reference: reference: reference:

reference: reference: reference:
reference: reference: reference:
reference: reference:
reference: reference:
reference: reference: reference: reference:
reference:
reference: reference: reference:
reference:
reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference:
reference: reference:
reference: reference: reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference:
reference:
reference: reference: reference: reference:
reference: reference: reference: reference:

reference:
reference: reference:
reference:
reference:
reference: reference: reference:
reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference:
reference:
reference: reference:
reference: reference:
reference: reference: reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference: reference: reference: reference:
reference:
reference: reference: reference: reference: reference:
reference: reference: reference: reference:
reference: reference: reference: reference: reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:
reference: reference:

reference: reference:
reference: reference:
reference: reference:
reference: reference: reference: reference:
reference: reference: reference: reference: reference: reference: reference:
reference: reference: reference: reference:
reference: reference: reference: reference:
reference:

Table A.1: *ASCII table*

Unicode

There are many versions of Unicode, UTF-16 is the most popular one. It is represented by 16 bits, which is needed to satisfy any language efficiently. It ranges from 0x0000 to 0xFFFF.

In ASCII it was not possible to store characters of different languages, as it just 7 bits. But Unicode versions are expanded to 16, 32 bits

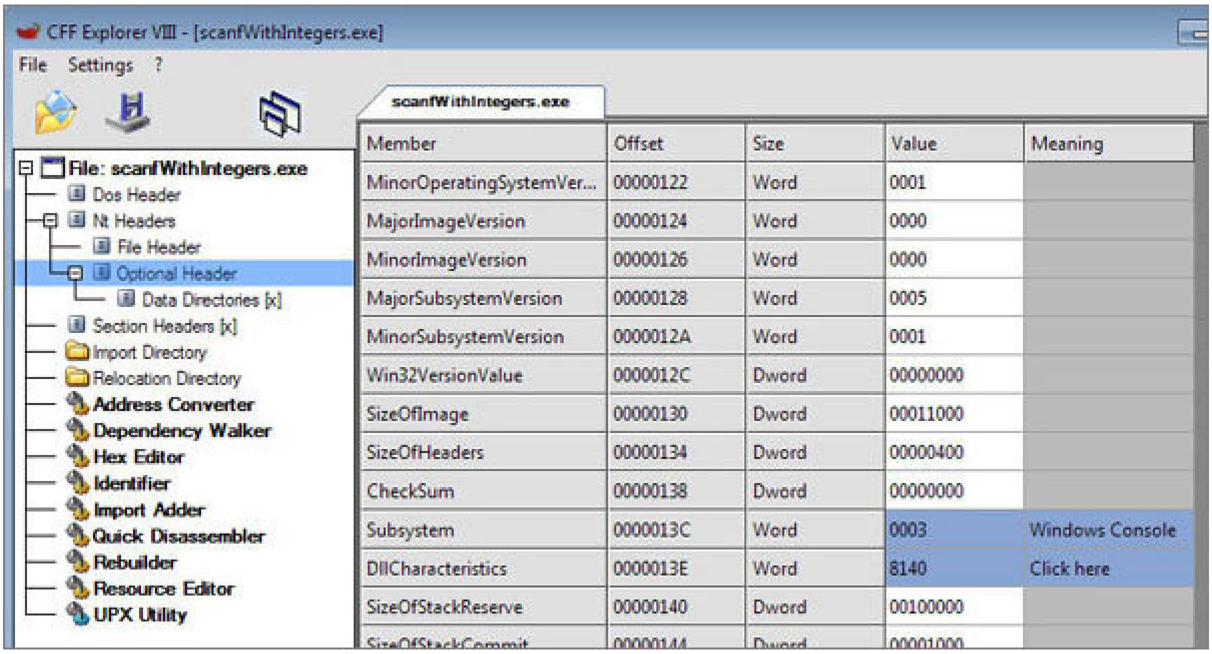
Example: ASCII of ‘A’ = 0x41 and Unicode (UTF=16) representation is 0041.

Disable Address Space Layout Randomization

Address Space Layout Randomization is a security mechanism by which base address of PE file is randomized on every load. To disable the ASLR on the **Portable Executable** file generated with our MSVC compiler we will following steps. This will help us reload PE file without randomizing base address of PE file. To disable ASLR we will use a CFF Explorer (it can be download from Open the PE file generated in CFF Explorer and follow steps:

Select from the left panel, **Optional Header**

In the **Optional** find

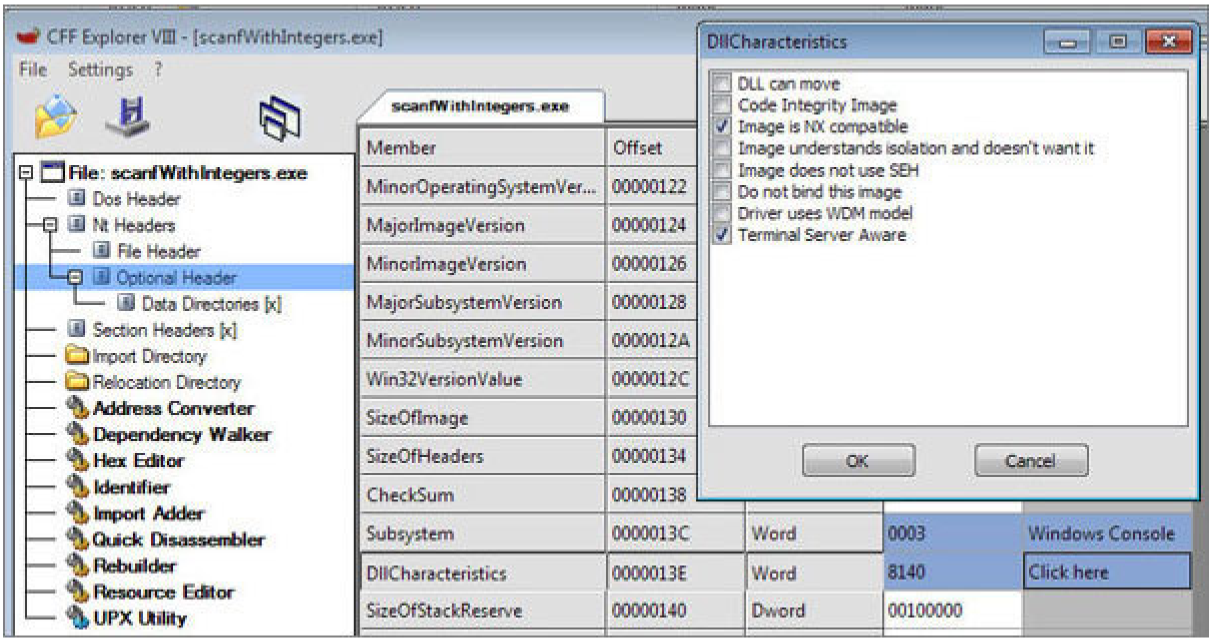


Member	Offset	Size	Value	Meaning
MinorOperatingSystemVer...	00000122	Word	0001	
MajorImageVersion	00000124	Word	0000	
MinorImageVersion	00000126	Word	0000	
MajorSubsystemVersion	00000128	Word	0005	
MinorSubsystemVersion	0000012A	Word	0001	
Win32VersionValue	0000012C	Dword	00000000	
SizeOfImage	00000130	Dword	00011000	
SizeOfHeaders	00000134	Dword	00000400	
CheckSum	00000138	Dword	00000000	
Subsystem	0000013C	Word	0003	Windows Console
DllCharacteristics	0000013E	Word	8140	Click here
SizeOfStackReserve	00000140	Dword	00100000	
SizeOfStackCommit	00000144	Dword	00001000	

Figure A.4: *PE file optional header using CFF Explorer*

Click on **Click here** on the right bottom.

Uncheck the **DLL can move** and click



Member	Offset			
MinorOperatingSystemVer...	00000122			
MajorImageVersion	00000124			
MinorImageVersion	00000126			
MajorSubsystemVersion	00000128			
MinorSubsystemVersion	0000012A			
Win32VersionValue	0000012C			
SizeOfImage	00000130			
SizeOfHeaders	00000134			
CheckSum	00000138			
Subsystem	0000013C	Word	0003	Windows Console
DllCharacteristics	0000013E	Word	8140	Click here
SizeOfStackReserve	00000140	Dword	00100000	

☐ DLL can move
☐ Code Integrity Image
☑ Image is NX compatible
☐ Image understands isolation and doesn't want it
☐ Image does not use SEH
☐ Do not bind this image
☐ Driver uses WDM model
☑ Terminal Server Aware

Figure A.5: *Disable ASLR on PE file*

Now go to **File menu** and **Save** changes.

Now using x32dbg, we can load PE file to do analysis without randomizing base address on every load.

Index

A

AddNumber.cpp code 86
Address Space Layout Randomization (ASLR)
about 460
disabling 461
Advanced Micro Devices 3
architecture, computing system
about 9
CPU 9
input/output devices 10
memory 10
System Bus 10
arithmetic bit shifting 456
arithmetic instructions
AAA 41
AAD 42
AAM 43
AAS 42
ADC 43
ADD 43
CMP 44
DAA 44
DAS 44
DEC 45
DIV 45
IDIV 46
IMUL 47

INC 47
MUL 47
NEG 48
SBB 48
SUB 48
XADD 49
array
about 247
defining 248
array code
reverse engineering pattern 247
array loop
with optimization
without optimization
ASCII 460
assembly instructions
about 23
arithmetic instructions 41
bit manipulation instructions 61
branching instructions 54
categories 35
data transfer instructions 37
processor control instructions 70
program execution instructions 49
stack instructions 35
string instructions 73
assembly instruction, with register operands
example 35
assembly language instructions 34

B

basic code
reverse engineering pattern 97
binary analysis, with Cutter
dashboard tab 419
decompiler tab 423
disassembly tab 421
graph tab 421
hexdump tab 422
imports tab 420
strings tab 420
bit manipulation instructions
about 61
AND 62
BSWAP 61
NOT 62
OR 62
RCL 63
XOR 63
bit shifting 455
branching instructions
about 54
JA 57
JAE 58
JB 59
JBE 59
JE 55
JECXZ 61
JG 56

JGE 57
JL 58
JLE 58
JMP 54
JNE 56
JNZ 55
JO 60
JS 60
JZ 55
RCR 64
ROL 64
ROR 65
SAL 68
SAR 68
SHL 67
SHLD 69
SHR 66
SHRD 69
building blocks, of computing system
Bit 11
Byte 11
DWORD 11
Nibble 11
Word 11

C

calling conventions
about 84
CDECL 85
FASTCALL 85

STDCALL 85
types 84
CDECL calling convention 85
CFF Explorer, PE 28
Clear the Carry Flag (CLC) instruction 70
Clear the Direction Flag (CLD) instruction 70
Clear the Interrupt Flag (CLI) instruction 71
code cave concept 442
code optimization
about 98
empty function 98
empty function with optimization 107
empty function without optimization
common sections, PE
.bss 27
.debug 28
.edata 28
.idata 28
.rdata 28
.rsrc 28
.text 27
Complement Carry (CMC) instruction 71
Component Object Model (COM) 27
computing system
about 8
architecture 9
building blocks 11
fixed program computing system 8
stored program computing system 8

concept, behind different calling conventions

about 86
CDECL
FASTCALL
STDCALL
conditional jump 54
CPU, computing system
control unit 9
execution unit 9
flags 10
registers 9
cryptocurrency 4
Cutter
about 416
downloading 416
installing
using, for binary analysis 418

D

data transfer instructions
CMPXCHG 38
LAHF 39
LAR 39
LEA 38
MOV 37
MOVS 41
MOVSX 40
MOVZX 40
SAHF 39
XCHG 38

XLAT 40
debuggers
about 30
x32dbg 31
decision control structure
reverse engineering pattern 186
disassemblers
about 29
Cutter 30
Ghidra 29
Disk Operating System (DOS) 2
DLL can move 187
DllCharacteristics parameter 187
Dynamic Link Library (DLL) 27

E

EEFLAGS register 17
EIP. *See* Instruction Pointer register
Escape to Floating Point Coprocessor (ESC) instruction 71
Exchange and Add (XADD) instruction 48

F

FASTCALL 85
first generation microprocessors
Intel 8086 12
Intel 80186 12
Intel 80286 13
Intel 80386 13

Intel 80486 13
fixed program computing system 8
Floating Point Unit (FPU) 131

Foo function 19
for loop
about 230
with optimization
without optimization
function printf with char
about 147
function printf printing char with optimization
function printf printing char without optimization
function printf with float
about 131
function printf printing float with optimization
function printf printing float without optimization
function printf with integers
about 122
function printf printing integers with optimization 130
function printf printing integers without optimization
function scanf
with integers 312
with optimization
without optimization

G

general purpose registers, x86 architecture
EAX 16
EBP 16

EBX 16
ECX 16
EDI 16
EDX 16

ESI 16
ESP 16
Ghidra
about 29
installing

H

Hard Disk (HDD) 10
Hashes section 418
“hello, world” program
about 113
with optimization
without optimization
hidden URL
decrypting
Hungarian Notation 154

I

iArray 248
if-else statement
about 186
with optimization
without optimization
installation

Ghidra 394
Instruction Pointer register 18
Integer Multiple (IMUL) instruction 47
Intel 8086 12
Intel 80186 12
Intel 80286 13
Intel 80386 13

Intel 80486 13
Intel processor
x86-16 14
x86-32 (aka IA32) 14
x86-64 14
Intel x86-32 processor
memory 14
registers 15
Interrupt Overflow (INTO) 51
Interrupt Return (IRET) instruction 52
Interrupt Service Routine (ISR) 52
iNumber 154
I/O (input/output) devices, computing device 10

J

JDK 11
downloading 394
installing 394
Jump if above (JA) instruction 57
Jump if above or equal (JAE) instruction 57
Jump if below (JB) instruction 59
Jump if below or equal (JBE) instruction 59

Jump if Equal (JE) instruction 55
Jump if Greater (JG) instruction 56
Jump if Greater or Equal (JGE) instruction 57
Jump if Less (JL) instruction 58
Jump if Less or equal (JLE) instruction 58
Jump if Not Equal (JNE) instruction 56
jumps, in x86
conditional jump 54

unconditional jump 54

L

Last In First Out (LIFO) 18
least significant bit (LSB) 453
Load Effective Address (LEA) 162
Load String Byte (LODSB) 78
Load String DWORD (LODSD) 78
Load String (LODS) 78
Load String Word (LODSW) 78
logical bit shifting 455
loop control structure
reverse engineering pattern 211

M

macro 449
main function 19
memory, computing system
Random Access Memory (RAM) 10
Read Only Memory (ROM) 10

memory, Intel x86-32 processor
data 15
heap 15
kernel space 15
libraries 15
stack 15
text 15
microprocessor
about 12
first generation microprocessor models 12
most significant bit (MSB) 453

Move String Byte (MOVSB) 80
Move String DWORD (MOVSD) 80
Move String (MOVS) 80
Move String Word (MOVSW) 80

N

National Security Agency (NSA) 29
No Operation (NOP) instruction 72
npad 449
npad macro 450

O

operands
intermediate operands 34
memory address operands 34
register operands 34
operation code 34

optimization 98
OUTSB instruction 77
OUTSD instruction 77
OUTSW instruction 77
Overflow flag (OF) 51

P

patched calculator 446
pointer program
reverse engineering pattern 153
pointers
about
with optimization
without optimization
POP 18

Portable Executable Editors
about 27
basic structure 27
CFF Explorer 28
common sections 27
printf program
reverse engineering pattern 121
procedure 449
processor control instructions
CLC 70
CLD 70
CLI 71
CMC 71
ESC 71

LOCK 72
NOP 72
STC 73
STD 73
STI 73
program code 34
program execution instructions
about 49
CALL 49
ENTER 50
INT 51
INTO 51
IRET 52
LEAVE 50
LOOP 52

LOOPE 52
LOOPNE 53
TEST 53
pseudo code
about 18
Foo function 19
main function 19
PUSH 18

R

Radare2 30
Random Access Memory (RAM) 10
ransomwares 3
Read Only Memory (ROM) 10

registers, Intel x86-32 processor
about 17
general purpose registers 16
Instruction Pointer register 18
segment registers 17
Repeat if Equal (REPE) 81
Repeat if Not Equal (REPNE) 82
Repeat if Not Zero (REPNZ) 82
Repeat if Zero (REPZ) 81
Repeat (REP) 81
returning value
about 108
returning value with optimization
returning value without optimization
Reverse Engineering (RE)
about 2

bounty for cyber enthusiasts 5
example 3
existing design, studying 4
importance 4
military espionage 4
outdated or lost product, redeveloping 4
product vulnerabilities, finding 5
role 6
security audit 4
sensitive data, finding 4
reverse engineering tools
categories 27
debuggers 30
disassemblers 29
importance 26

Portable Executable Editors 27
Rotate Left (ROL) 64
Rotate Right (ROR) 65
Rotate through Carry Left (RCL) instruction 63
Rotate through carry right (RCR) 64

S

scanf program pattern
in reverse engineering 311
Scan String Byte (SCASB) 79
Scan String DWORD (SCASD) 79
Scan String (SCAS) 79
Scan String Word (SCASW) 79
segment registers, x86 architecture
Code Segment (CS) 17

Data Segment (DS) 17
ES 17
FS 17
GS 17
Stack Segment (SS) 17
Set the Carry Flag (STC) instruction 73
Set the Direction Flag (STD) instruction 73
Set the Interrupt Flag (STI) instruction 73
Shift Arithmetic Left (SAL) 68
Shift Arithmetic Right (SAR) 68
Shift Left Double (SHLD) 69
Shift Logical Left (SHL) 67
Shift Logical Right (SHR) 66
Shift Right Double (SHRD) 69

signed terms 455
simple interest calculation
program, writing for 376
without optimization
simple interest code pattern
in reverse engineering 375
stack
about 19
pseudo code 18
stack frame
about 19
Callee After Function Call 22
Callee Before Returning 23
Caller After Returning 24
Caller Before Callee Call 21

stack instructions
about 35
POP 36
POPAD 37
POPFD 37
PUSH 36
PUSHAD 36
PUSHFD 36
RET 37
status register, x86 architecture
Carry Flag (CF) 17
Overflow Flag (OF) 18
Parity Flag (PF) 18
Sign Flag (SF) 17
Trap Flag (TF) 18
Zero Flag (ZF) 17

STDCALL 85
stored program computing system 8
Store String Byte (STOSB) 78
Store String DWORD (STOSD) 79
Store String (STOS) 78
Store String Word (STOSW) 79
strcpy
about 328
with optimization
without optimization
strcpy program pattern
in reverse engineering 327
string instructions

CMPS/CMPSB/CMPSW 74
IN/INSB/INSW/INSD 76
LODS/LODSB/LODSW/LODSD 78
MOVS/MOVSB/MOVSW 80
OUT/OUTSB/OUTSW/OUTSD 77
REP 81
REPE/REPZ 81
REPNE/REPNZ 82
SCAS/SCASB/SCASW 79
STOS/STOSB/STOSW 78
structure
about 282
with optimization
without optimization
structure code
reverse engineering pattern 281
subroutine 449
System Bus, computing system

Address Bus 10
Control Bus 10
Data Bus 10

T

tools, reverse engineering. *See* reverse engineering tools

U

unconditional jump 54
Unicode 460
unsigned terms 454
UTF-16 460

V

von Neumann architecture 9

W

Wannacry ransomware
about 3
analyzing
breaking 393
disassembling
while condition
about 212

with optimization
without optimization
Windows Calculator
reverse engineering 429
Windows Calculator, reverse engineering
about 430
binary, patching 446
code flow, with breakpoints
code, writing in code cave
placeholder, finding to call code

X

x32dbg
about 31
disassembly or CPU instructions 31
memory dump 31
registers and flags 31
stack 31
x86 Intel family 14