13.037 - Introduction to Software Exploitation

Nathan Rittenhouse – nathan_@mit.edu

Why are you here?

- Obvious interest in making software go ‘boom’
- Reverse engineering software
  - Cracking some (non-copyrighted) software
- Exploitation on many different platforms
  - Windows
  - Linux
  - OSX
- Mostly basics for each OS covered

Why aren’t you here?

- LaTeX presentations
- Defeating the *latest* memory protection methods
  - Those will be covered, just few/no examples with them
- Walking out a Mark Dowd/Neel Mehta/Optyx/HD Moore/Matt Miller/Skape
- ‘Linux always has better security than Windows’
- ‘security models’ (see above)

Tools - Disassemblers

- Take in programs in a variety of binary file formats, convert the machine instructions into mnemonics that are quasi-human readable
- Objdump
  - Decent if on a random Unix based system
- Dumpbin
  - Equivalent to objdump for Windows

IDA Pro

- Hands down the best disassembler
- Runs on Linux and Windows
  - Linux version is curses…
- Free version and commercial version
  - Academic licensing is available
- Powerful plugin interface, many plugins written
  - Bindiff/Hex-rays/mIDA/x86emu/CollabREate
- Can be scripted in IDC/Python/Ruby/Perl
- Pretty graphing interface

IDA Pro

- Define data structures a program uses
- Code highlighting
- Code commenting
- Save to portable database
- Remotely debug on MANY different OSes and architectures

Tools - Debuggers

- Allow one to step through a program, analyze what happens at run time
  - What modules get loaded
  - Disassembly of executing code
  - Register states
  - Breakpoints
  - Symbol resolution (sometimes)
  - Limited tracking of operations to data

Tools - Debuggers

- GDB
  - Source oriented debugger for Unix based OSes
    - Yes, there is a Windows port, but.. No
  - Quite robust for source debugging, quite poor for binary analysis
- Ollydbg
  - Reverse engineering debugger for Windows
  - ‘Pretty’ but somewhat unstable
  - Many reverse engineering oriented plugins
  - Better data visualization, easier to use than…

Tools - Debuggers

- WinDbg
  - Binary and source debugger written by Microsoft
  - Incredibly well written, robust, has a steep learning curve
    - Great documentation
  - Supports remote/some local ring 0 debugging
  - Rich COM based plugin architecture
  - Mixture of command line and (poor) GUI interface

Tools – packet sniffer

- This is REALLY important when dealing with different protocols
- Read Wireshark's protocol dissectors
- Let Wireshark do hard work for you occasionally (with its protocol dissectors)

Tools - virtualization

- Do as MUCH work as you can in a VM as possible
- Bare metal machines are harder to recover
- Malware analysis can go horribly wrong
  - Even with a VM
- ‘Enterprise software’ – code words for ‘so bloated you’d NEVER want to put this on a real system’
- Allows for multiple snapshots
  - Can be for different service packs, different projects
  - Don’t have to revert to original state

Exploiting a Classic

Your first stack based buffer overflow on Linux

The stack

- A data structure that grows downward in address space in LIFO format
  - Think of a stack of plates
- Used for temporary variable storage
- Holds certain control flow information

```
int function(char * userInput){
  int blah;
  char array[400];
  strcpy(array, userInput);
```

Translation to x86

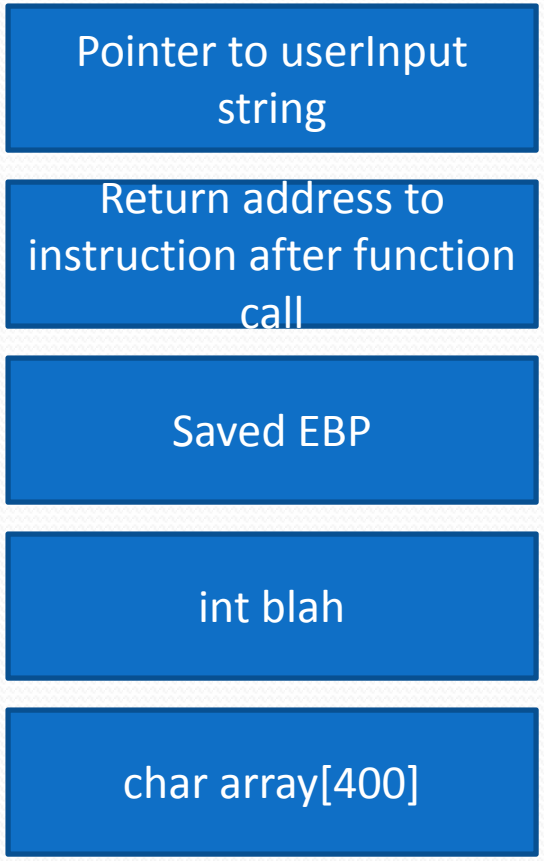
```
OtherFunction:
call function
test eax, eax ; <- return address points here

Function:
mov edi, edi
push ebp
mov ebp, esp
sub esp, 0d404
mov eax, esp
push [ebp+8]
push eax
call strcpy
```

Stack layout

```
int function(char * userInput){
  int blah;
  char array[400];

  strcpy(array, userInput);
…
}
```



The vulnerability

- strcpy will not stop copying data until there is a NULL byte reached
  - If this input is taken from the user, this means that we can input arbitrarily long strings to the program
  - If this is the case, we can overwrite EBP and the saved return address
- We can then point the return address at any arbitrary code that we want

Where IS the return address?

- One of two options
  - Reverse engineer the program and look at stack math
  - Be lazy and use pattern_create from Metasploit

Shellcode

- Shellcode does stuff
  - It's the exploit's payload
  - Shellcode is NOT the vulnerability
- Shellcode can
  - Send a shell to a remote attacker
  - Load more code from a remote system
  - Create another backdoor (depends on permissions of user)
  - Anything the attacker wants

Shellcode

- ..also must evade certain ‘bad chars’
  - A ‘bad char’ causes the shellcode not to be copied fully or get modified in a way not desired by the attacker
- Examples
  - Must avoid NULL bytes in the previous example
  - Must avoid uppercase characters if there is a tolower() conversion
  - Same if there is a toupper() conversion
  - Usually any control characters for what protocol you’re exploiting under

Standard Linux shellcode

- Control flow
  - Listen for connection
  - Accept connection
  - Set stdin/stdout/stderr to be ‘hooked up’ to the new connection instead
  - execve /bin/sh

Where am I?

- 32bit x86 doesn’t allow [EIP+x]
  - EIP can’t be directly read, either
- Needed for finding the address of a string
- Also for creating temporary variable storage
  - Using raw `push`/`pop` instructions can overwrite shellcode – remember we are EXECUTING on the stack

GetEIP – call, pop

```
call end
startShellcode:
…
end:
    pop esi
    jmp startShellcode
```

GetEIP – jmp/call

- Avoids having a null byte

```
jmp end
trampoline:
    pop esi
    jmp startShellcode
end:
    call trampoline
startShellcode:
…
```

GetEIP – FPU

- Short, no null bytes
- `fstenv` dumps context of last FPU instruction

```
fldz
fstenv [esp-0x0c]
pop esi
```

- From the Intel manual

```
DEST[FPUControlWord) FPUControlWord;
DEST[FPUStatusWord) FPUStatusWord;
DEST[FPUTagWord) FPUTagWord;
DEST[FPUDataPointer) FPUDataPointer;
DEST[FPUInstructionPointer) FPUInstructionPointer;
DEST[FPULastInstructionOpcode) FPULastInstructionOpcode;
```

GetEIP – Gera's trick

- As long as FPU method, but this is interesting
- At start

0: [E8 FF FF FF FF] call 0x4

5: [C3] ret

6: [58] pop eax

At Start

- Becomes

4: [FF C3] inc ebx

5: [58] pop eax ;eax contains EIP

The return address

- Set return right to stack address
  - Unreliable – standard 2.6.x kernels randomize stack base
- Set to a location inside a binary loaded at a fixed address
  - Mainline kernel does not have ASLR turned on by default.. most binaries aren't compiled –fPIC
- Analyze the crash, see what registers you 'control'
  - `core-file core` in GDB

The return address

- Msfelfscan finds byte sequences that translate to x86 that will transfer control to a register you specify

References

- http://www.metasploit.com/data/confs/recon2005/recent_shellcode_developments-recon05.pdf
- http://milw0rm.org/shellcode/1649
- http://insecure.org/stf/smashstack.html <- note this is intended to be run on 2.4.x kernels with zero protection mechanisms
  - Also available on phrack.org