Return-to-libc Attack Lab

1 Background

The learning objective of this lab is to gain the first-hand experience on an interesting variant of buffer-overflow attack. This attack can bypass a protection scheme currently implemented in 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, 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, you are given a program with a buffer-overflow vulnerability. Your task is to develop a `return-to-libc` attack to exploit the vulnerability and finally to get the shell. In addition to the attack, you will experience several protection schemes that have been implemented in Ubuntu to counter against the buffer-overflow attacks. You need to evaluate whether the schemes work or not and explain why.

2 Lab Tasks

Non-Executable Stack. In `Ubuntu` the binary images of programs (and shared libraries) declare whether it requires 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
```

Because the objective of this lab is to realize that non-executable stack protection can be exploited, you should always compile your program using the `"-z noexecstack"` option in this lab.

2.1 The Vulnerable Program

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

int vuln(FILE *input){
     char buffer[12];
     fread(buffer, sizeof(char), 40, input);

     return 1;
}

int main(int argc, char **argv){
     FILE *input;

     input = fopen("input", "r");
     vuln(input);

     printf("YOU REALLY GOT IT !!\n");

     fclose(input);
     return 1;
}
```

The above program has a buffer overflow vulnerability. It first reads an input of size 40 bytes from a file called ”input” into a buffer of size 12, causing the overflow. The function fread() does not check boundaries, so buffer overflow will occur. It should be noted that the program gets its input from a file called ”input”. This file is under users control. Now, your objective is to create the contents for ”input”, such that when the vulnerable program copies the contents into its buffer, a shell can be spawned.

2.2 Task 1: Exploiting the Vulnerability

You need to create the **input**. You may use the sample code exploit.c to create one.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char **argv){

  char buf[40];
  FILE *badfile;

  input = fopen("./input", "w");

  *(long *) &buf[X] = some address ;   //  "/bin/sh"
  *(long *) &buf[Y] = some address ;   //  system()
  *(long *) &buf[Z] = some address ;   //  exit()

  fwrite(buf, sizeof(buf), 1, input);
  fclose(input);
}
```

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. You may use the environment variable method to find the address of ”/bin/sh”.A sample program to get the address of the environment variable is given as `getEnvironmentVariable.c`.

After you finish the above program, compile and run it. This will generate the contents for ”input”. Run the vulnerable program `retlib`. If your exploit is implemented correctly, when the function `vuln` returns, it will return to the `system()` libc function, and execute `system("/bin/sh")`.

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.

```
$ gcc -o exploit exploit.c
$./exploit         // create the badfile
$./retlib          // launch the attack by running the vulnerable program
$ <---- You've got a shell!
```

2.3 Task 2: Address Randomization

In this task, you will turn on the Ubuntu's address randomization protection. 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 be able to answer these questions. You can use the following instructions to turn on the address randomization:

```
$ su root
  Password: (enter root password)
# /sbin/sysctl -w kernel.randomize_va_space=2
```

2.4 Task 3: Stack Guard Protection

In this task, let us turn on the Ubuntu's Stack Guard protection. Please remember to turn off the 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 Stack Guard protection make your return-to-libc attack difficult? You can use the following instructions to compile your program with the Stack Guard protection turned on.

```
$ gcc -z noexecstack  -o retlib retlib.c
```

References

[1] c0ntext Bypassing non-executable-stack during exploitation using return-to-libc http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[2] Phrack by Nergal Advanced return-to-libc exploit(s) *Phrack 49*, Volume 0xb, Issue 0x3a. Available at http://www.phrack.org/archives/58/p58-0x04