github链接https://github.com/isBigChen/ucas_ret2csu_lab

1.背景知识

1.1 rop

Return Oriented Programming返回导向编程
在栈溢出的基础上，利用程序中可执行的一些代码片段组合起来修改相应的寄存器，并控制程序的执行流程，最终执行shell命令。

1.2 ret2csu

在64位程序中，函数调用约定规定前6个参数通过寄存器传递，参数顺序依次为RDI、RSI、RDX、RCX、R8、R9。但是很多情况下很难找到每个寄存器对应的gadgets，这样就无法控制调用函数时传递的参数。但是64位程序下有__libc_csu_init这个函数，用于对libc进行初始化操作，而一般的程序都会调用libc，所以这个libc的初始化函数会存在。不同版本中__libc_csu_init函数实现不一样，但是效果相似。

```
.text:0000000000400600                          loc_400600:                             ; CODE XREF: __libc_csu_init+
.text:0000000000400600 4C 89 EA                 mov     rdx, r13
.text:0000000000400603 4C 89 F6                 mov     rsi, r14
.text:0000000000400606 44 89 FF                 mov     edi, r15d
.text:0000000000400609 41 FF 14 DC              call    ds:(__frame_dummy_init_array_entry - 600E10h)[r12+rbx*8]
.text:0000000000400609
.text:000000000040060D 48 83 C3 01              add     rbx, 1
.text:0000000000400611 48 39 EB                 cmp     rbx, rbp
.text:0000000000400614 75 EA                    jnz     short loc_400600
.text:0000000000400614
.text:0000000000400616
.text:0000000000400616                          loc_400616:                             ; CODE XREF: __libc_csu_init+
.text:0000000000400616 48 83 C4 08              add     rsp, 8
.text:000000000040061A 5B                       pop     rbx
.text:000000000040061B 5D                       pop     rbp
.text:000000000040061C 41 5C                    pop     r12
.text:000000000040061E 41 5D                    pop     r13
.text:0000000000400620 41 5E                    pop     r14
.text:0000000000400622 41 5F                    pop     r15
.text:0000000000400624 C3                       retn
```

可以利用这些gadgets对函数调用相关的寄存器进行赋值，并结合rop来达到攻击效果。

1.3 plt和got

对于动态链接，如何获取函数地址，主要用到两个表
一个是PLT表(Procedure Link Table)程序链接表，一个是GOT表(Global Offset Table)全局偏移表

PLT表中的每一个表项的内容都是对应的GOT表中这一项的地址，PLT表中的数据不是函数的真实地址，而是GOT表项的地址。
GOT表项中的数据才是函数真实的地址，所以可以通过PLT表跳转到GOT表来得到函数真正的地址。

2.解题思路

题目开启了栈不可执行保护，因此需要构造rop链

```
[*] '/home/jgc/桌面/level5'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

main函数中调用了vulnerable_func()函数

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  write(1, "Hello, World\n", 0xDuLL);
  vulnerable_function(1LL);
  return 0;
}
```

在函数中存在栈溢出

```
ssize_t vulnerable_function()
{
  char buf[128]; // [rsp+0h] [rbp-80h] BYREF

  return read(0, buf, 0x200uLL);
}
```

但是程序中没有 `execve` 和 `/bin/sh`，没办法直接利用。

execve可以利用libc中的函数地址，`/bin/sh` 可以通过调用read函数写入；即把execve函数地址及其参数写入到程序的bss段

第1步，利用libc的gadgets获取write函数的真实地址，并让程序重新执行main函数

第2步，获取libc中的execve函数的地址

第3步，利用libc的gadgets执行read函数，将execve函数地址和 `/bin/sh` 写入到程序的bss段，并让程序重新执行main函数

第4步，利用libc的gadgets执行execve函数，得到shell

3.解题细节

3.1 loc_400600

`libc_csu_init` 函数中的部分gadgets如下

```
loc_400600:                             ; CODE XREF: __libc_csu_i
mov     rdx, r13
mov     rsi, r14
mov     edi, r15d
call    ds:(__frame_dummy_init_array_entry - 600E10h)[r12+rbx*8]

add     rbx, 1
cmp     rbx, rbp
jnz     short loc_400600
```

- `mov edi, r15d` 的效果是将r15低32位赋值给edi，但是此时rdi高32位会被赋值为0，这样也就控制了edi寄存器
- `call` 语句中，地址采用 `[r12+rbx*8]`，这里可以将rbx总设置为0，就可以将要调用的函数的地址保存到r12寄存器中
- `cmp rbx, rbp` 中比较了两个寄存器的值，由于上面设置rbx为0，所以这里需要让rbp为1，才能继续执行 `jnz` 下面的指令

3.2 loc_400616

```
                                loc_400616:
48 83 C4 08                     add     rsp, 8
5B                              pop     rbx
5D                              pop     rbp
41 5C                           pop     r12
41 5D                           pop     r13
41 5E                           pop     r14
41 5F                           pop     r15
C3                              retn
```

- 这段gadgets从`pop rbx`开始可以简化构造的payload
- 在返回到loc_400600处开始执行后，需要用到loc_400616处的retn指令返回到main函数起始地址重新开始执行，所以这里需要有6个地址分别被pop给相应的寄存器，这些地址数据不需要有实际的意义，只作为占位使用。
- 另外ret指令会将rsp值加8，即旧的ebp地址被弹出，因此在编写payload时需要留出的占位字节数为6*8+8

3.3 execve函数地址获取

首先获取got表中write函数的地址，减去libc静态文件中write函数的地址再加上libc静态文件中execve函数的地址，就是程序运行时libc在内存中execve函数的地址。

4.payload栈结构