

**课程报告**

**课程名称： 网络攻防基础**

**培养院系： 信息工程研究所**

**学 号： 202118018670043**

**姓 名： 张志宇**

**指导教师： 张玉清 龚晓锐 吴槟**

**报告日期： 2022.04.27**

**项目GIT：https://github.com/QGrain/learn-pwn**

**网络空间安全学院**

**目 录**

[实验2B ROP利用 1](#_Toc101991151)

[1.实验准备 1](#_Toc101991152)

[1.1实验要求 1](#_Toc101991153)

[1.2实验工具 1](#_Toc101991154)

[2.二进制防护手段 2](#_Toc101991155)

[2.1 ASLR 2](#_Toc101991156)

[2.2 NX 2](#_Toc101991157)

[2.3 PIE 2](#_Toc101991158)

[2.4 Canary 3](#_Toc101991159)

[2.5 RELRO 3](#_Toc101991160)

[3.实验level1 基本ROP 4](#_Toc101991161)

[3.1 ret2text 4](#_Toc101991162)

[3.2 ret2shellcode 7](#_Toc101991163)

[3.3 ret2syscall 9](#_Toc101991164)

[3.4 ret2libc 9](#_Toc101991165)

[4.实验level2 中高级ROP 9](#_Toc101991166)

[4.1 ret2csu 9](#_Toc101991167)

[5.实验总结 13](#_Toc101991168)

[6.附录 13](#_Toc101991169)

# 实验2B ROP利用

## 1.实验准备

### 1.1实验要求

（1）理解Linux系统进程控制的基本数据结构。（kernel里task\_struct结构的各个字段意义）

（2）深入理解二进制防护的手段及目的。

（3）完成给定的ROP攻击，写出Writeup。（level1：复现课堂讲授的4-5种ROP利用，level2：CTF-wiki的中高级ROP利用任选一题）

### 1.2实验工具

（1）Ubuntu 18.04（Kali）

（2）gdb + pwndbg

（3）python3 + pwntools

## 2.二进制防护手段

### 2.1 ASLR

ASLR的是操作系统的功能选项，作用于ELF装入内存运行时，因而只能随机化stack、heap、libraries的基址。开启后每次加载程序的 stack、libarys、heap等地址都会随机化，可用cat /proc/`pidof xxx`/maps查看。

1. 未开启：无作用

2. 半开启：随机化 stack 和 libarys

3. 全开启：随机化 stack、libarys 和 heap



图2-1 ASLR保护打开和关闭的方式

### 2.2 NX

No-Execute（不可执行），NX的原理是将数据所在内存页标识为不可执行，当程序执行流被劫持到栈上时，程序会尝试在数据页面上执行指令，因为数据页被标记为不可知性，此时CPU就会抛出异常，而不是去执行栈上数据。

在程序的某个位置有控制程序是否可以执行的标志位（若为6不可执行、7则可执行），也可以用execstack工具查询和设置该标志位，该标志位在51e5 7464后边。

1. NX disabled：栈可以执行，栈上的数据也可以被当作代码执行。
2. NX enabled：栈不可执行，栈上的数据程序只认为是数据，如果去执行的话会发生错误。即栈上的数据不可以被当作代码执行。

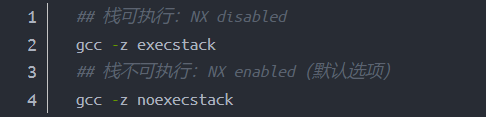


图2-2 gcc开启和关闭NX保护的方式

### 2.3 PIE

PIE（Position Independent Executables）是编译器（gcc，…）功能选项（-fPIE / -fpie），作用于编译过程，可将其理解为特殊的 PIC（so专用，Position Independent Code），加了 PIE 选项编译出来的 ELF 用 file 命令查看会显示其为 so，其随机化了 ELF 装载内存的基址（代码段、plt、got、data 等共同的基址）。其效果为用 objdump、IDA 反汇编之后的地址是用偏移表示的而不是绝对地址。

1. No PIE：无作用
2. PIE enabled：代码段、plt、got、data 等共同的基址会随机化。在编译后的程序中，只保留指令、数据等的偏移，而不是绝对地址的形式。

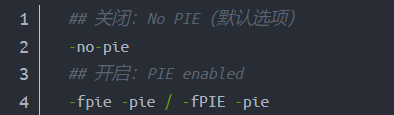


图2-3 gcc开启和关闭PIE保护的方式

### 2.4 Canary

金丝雀保护，是一种用来防护栈溢出的保护机制。其原理是在函数入口处，先从 fs/gs 寄存器中取出一个 4(eax)/8(rax) 字节的 cookie 信息存到栈上，当函数结束返回的时候会验证 cookie 信息是否合法(与开始存的是否一致)，如果不合法就停止程序运行。真正的 cookie 信息也会保存在程序的某个位置。插入栈中的 cookie 一般在 ebp / rbp 之上的一个内存单元保存。

1. 无Canary保护：无任何作用
2. 部分函数Canary保护：在一些容易受到攻击的函数返回地址前添加cookie。在函数返回时，检查该 cookie 与原本程序插入该位置的 cookie 是否一致，若一致则程序认为没有受到栈溢出攻击。
3. 全部函数Canary保护：所有的自定义函数在返回地址之前都会添加 cookie。在函数返回时，检查该 cookie 与原本程序插入该位置的 cookie 是否一致，若一致则程序认为没有受到栈溢出攻击。

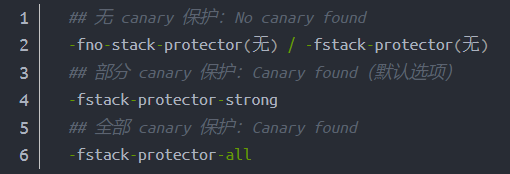


图2-4 gcc开启和关闭Canary栈溢出保护的方式

### 2.5 RELRO

设置符号重定位表格为只读或在程序启动时就解析并绑定所有动态符号，从而减少对 GOT 攻击。

1. No RELRO：此模式下关于重定位并不进行任何保护。
2. Partial RELRO：此模式下，一些段(包括.dynamic)初始化后会被标识为只读。
3. Full RELRO：此模式下，除了会开启部分保护外。惰性解析会被禁用（所有的导入符号将在开始时被解析，.got.plt段会被完全初始化为目标函数的终地址，并被标记为只读）。此外，既然惰性解析被禁用，GOT[1]与GOT[2]条目将不会被初始化为提到的值。



图2-5 gcc开启和关闭RELRO保护

## 3.实验level1 基本ROP

### 3.1 ret2text

#### 3.1.1 原理

ret2text 即控制程序执行程序本身已有的的代码 (.text)。其实，这种攻击方法是一种笼统的描述。我们控制执行程序已有的代码的时候也可以控制程序执行好几段不相邻的程序已有的代码 (也就是 gadgets)，这就是我们所要说的 ROP。

这时，我们需要知道对应返回的代码的位置。当然程序也可能会开启某些保护（参见上一章节），我们需要想办法去绕过这些保护。

#### 3.1.2 exploit

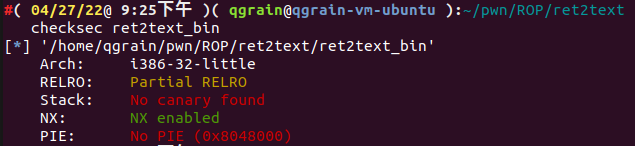


图3-1 ret2text\_bin的checksec信息

可以看出程序是 32 位程序，其仅仅开启了栈不可执行保护。然后，我们使用IDA pro来查看源代码。

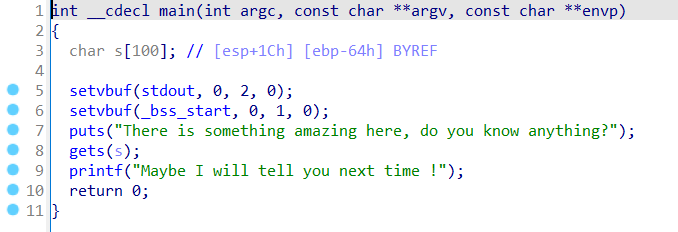


图3-2 ret2text\_bin反编译得到的main函数源码

又发现在secure函数中调用了system(“/bin/sh”)，但是执行到该语句需要满足随机数与输入的数相等，通过正常执行流程很难走到system这一句。

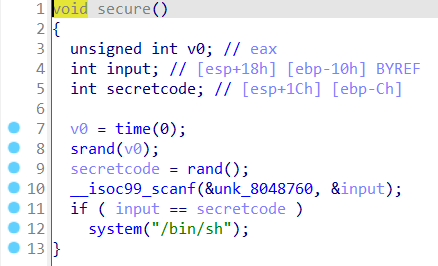


图3-3 secure函数

然而在secure的汇编代码可以看到，如果我们直接跳转到0x804863A，则可以实现直接执行/bin/sh。

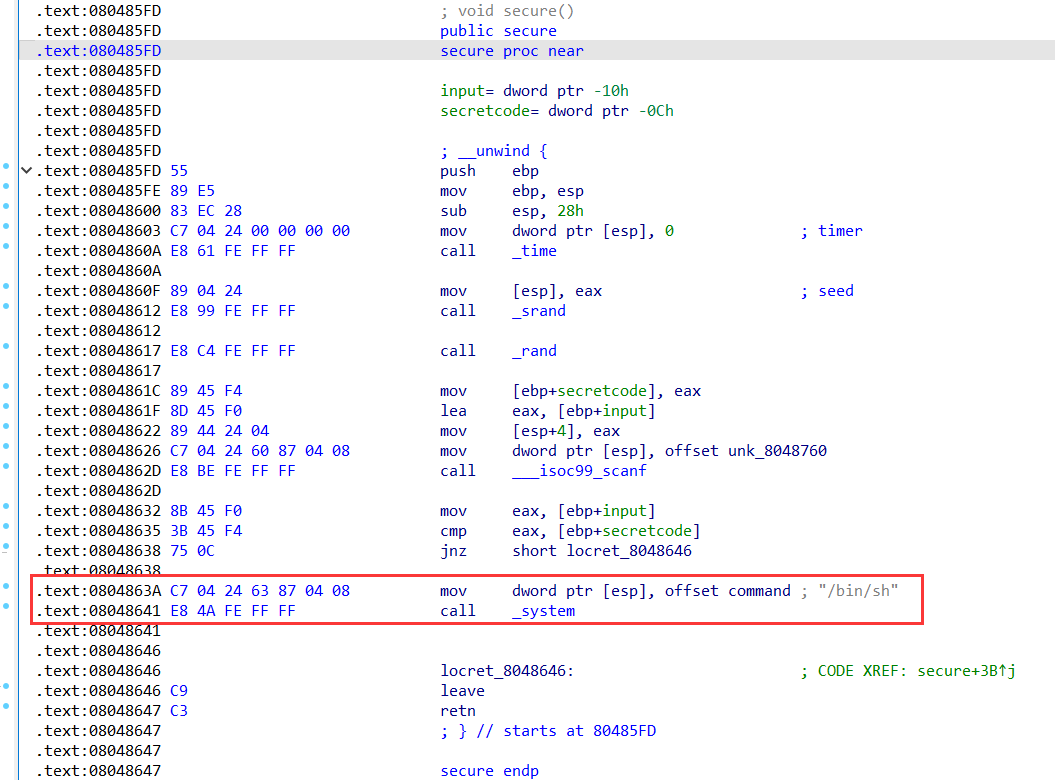
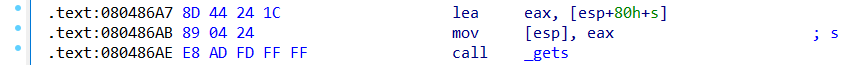


图3-4 Assembly of secure()

下面就是我们如何构造payload了，首先需要确定的是我们能够控制的内存的起始地址距离main函数的返回地址的字节数。在main函数的call \_gets处下断点（0x080486AE）：



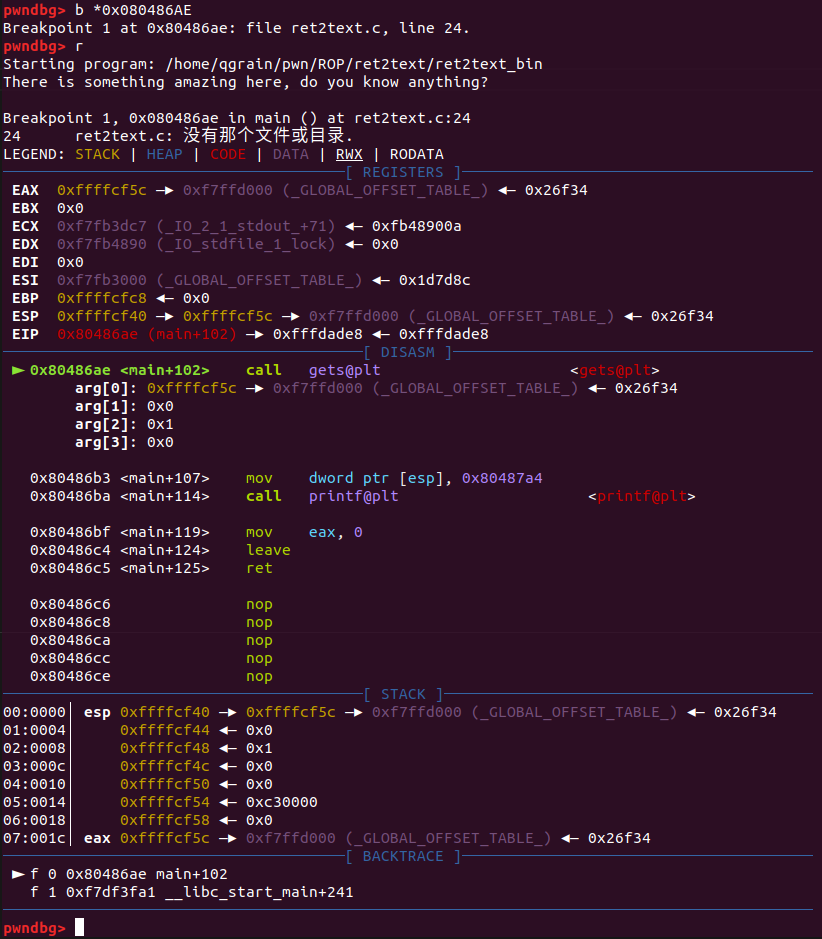


图3-5 在0x080486AE call \_gets处下断点

可以看到esp为0xffffcf40，ebp为0xffffcfc8，而字符串char s[100]是通过eax传参给gets，而eax=esp+0x80-0x64=esp+0x1c。因此可以得出s的地址为0xffffcf5c，所以s相对于返回地址的偏移为0x6c+4。

完整的exp.py如下：

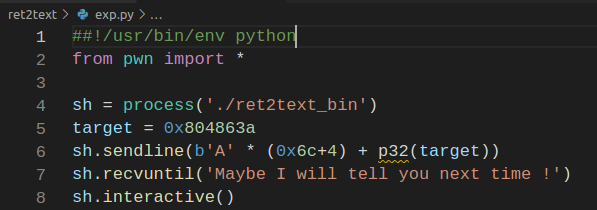


图3-6 ret2text的exp代码

运行结果如下：

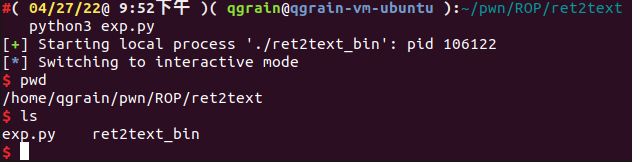


图3-7 ret2text结果

### 3.2 ret2shellcode

#### 3.2.1 原理

ret2shellcode，即控制程序执行 shellcode 代码。shellcode 指的是用于完成某个功能的汇编代码，常见的功能主要是获取目标系统的 shell。一般来说，shellcode 需要我们自己填充。这其实是另外一种典型的利用方法，即此时我们需要自己去填充一些可执行的代码。

在栈溢出的基础上，要想执行 shellcode，需要对应的 binary 在运行时，shellcode 所在的区域具有可执行权限。

#### 3.2.2 exploit

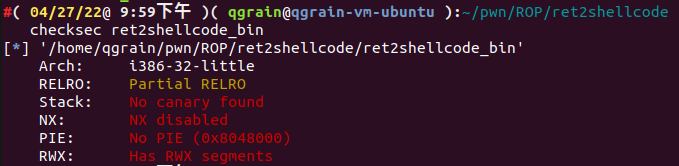


图3-8 checksec ret2shellcode\_bin

可以看出源程序几乎没有开启任何保护，并且有可读，可写，可执行段。我们再使用IDA pro看一下代码。



图3-9 ret2shellcode代码

可知程序仍然是基本的栈溢出漏洞，不过这次还同时将对应的字符串复制到buf2处。简单查看可知buf2在bss段。（.bss:0804A080; char buf2[100]）

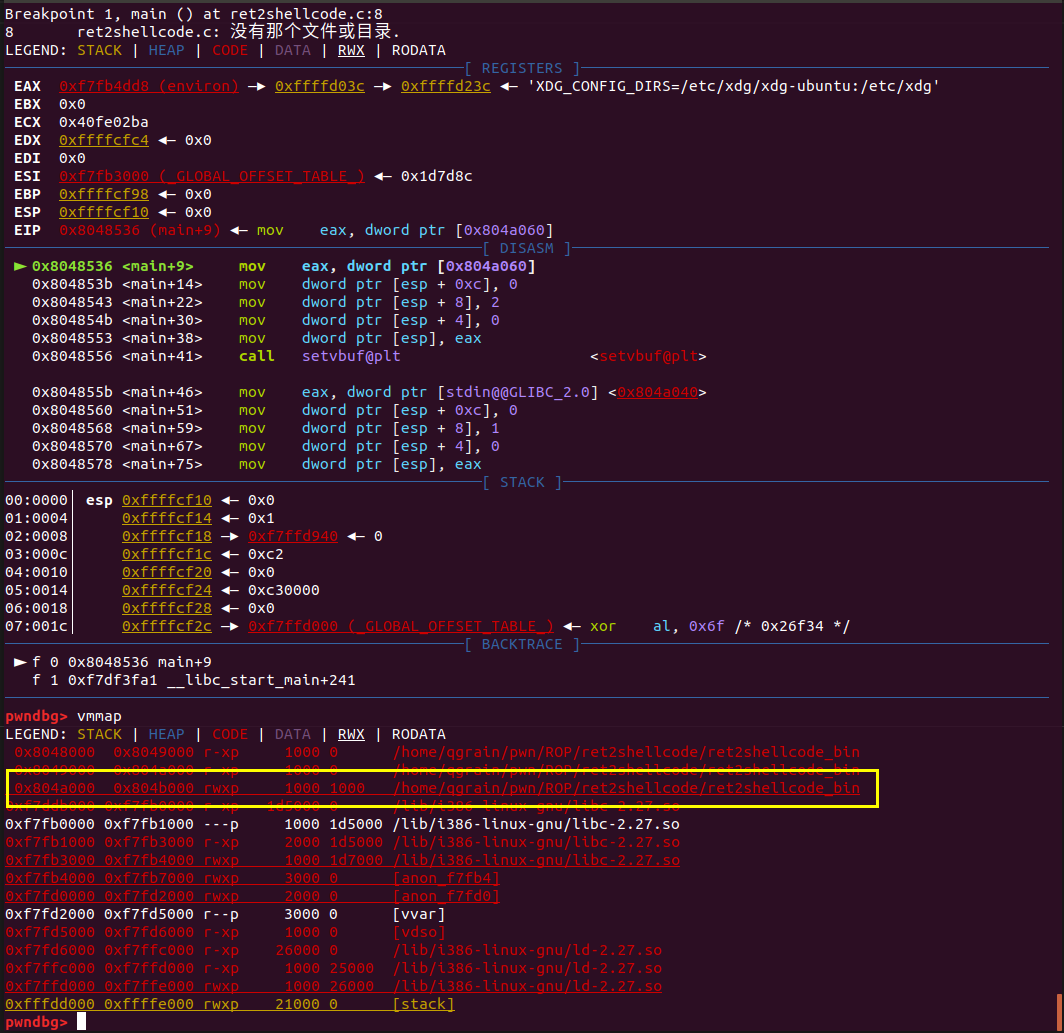


图3-10 gdb vmmap查看bss段可执行权限

通过vmmap，我们可以看到 bss 段对应的段具有可执行权限，那么这次我们就控制程序执行 shellcode，也就是读入 shellcode，然后控制程序执行 bss 段处的 shellcode。其中，相应的偏移计算类似于 ret2text。完整exp.py代码如下：



图3-11 exp of ret2shellcode

执行exp.py结果如下图所示，

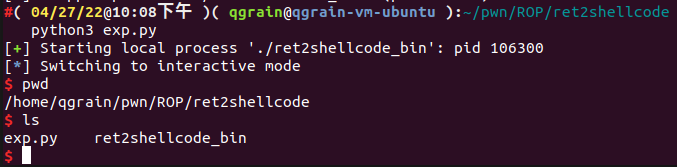


图3-12 ret2shellcode结果

### 3.3 ret2syscall

#### 3.3.1 原理

#### 3.3.2 exploit

时间关系，没能补充报告部分，后续Git仓库会更新此报告和相应exp代码

### 3.4 ret2libc

#### 3.4.1 原理

#### 3.4.2 exploit

时间关系，没能补充报告部分，后续Git仓库会更新此报告和相应exp代码

## 4.实验level2 中高级ROP

### 4.1 ret2csu

#### 4.1.1 原理

在 64 位程序中，函数的前 6 个参数是通过寄存器传递的，但是大多数时候，我们很难找到每一个寄存器对应的 gadgets。

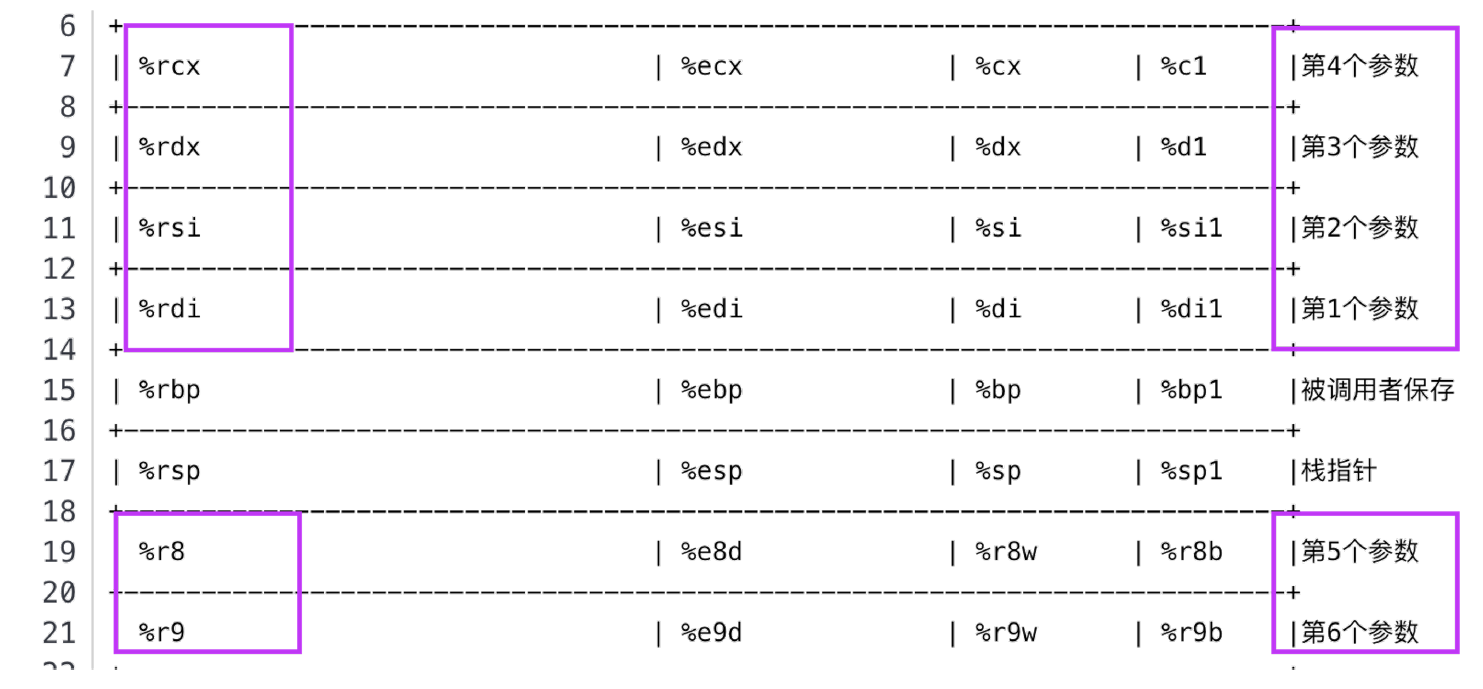


图4-1 64位程序函数六个传递参数的寄存器

然而对64位程序的溢出不能再如同对32位那样的单纯的参数压栈，我们需要对寄存器进行操作，因此需要一段代码(gadget)帮助我们将目的参数放进寄存器并传入函数。这时候，我们可以利用x64下的\_\_libc\_csu\_init中的gadgets。这个函数是用来对libc进行初始化操作的，而一般的程序都会调用libc函数，所以这个函数一定会存在。

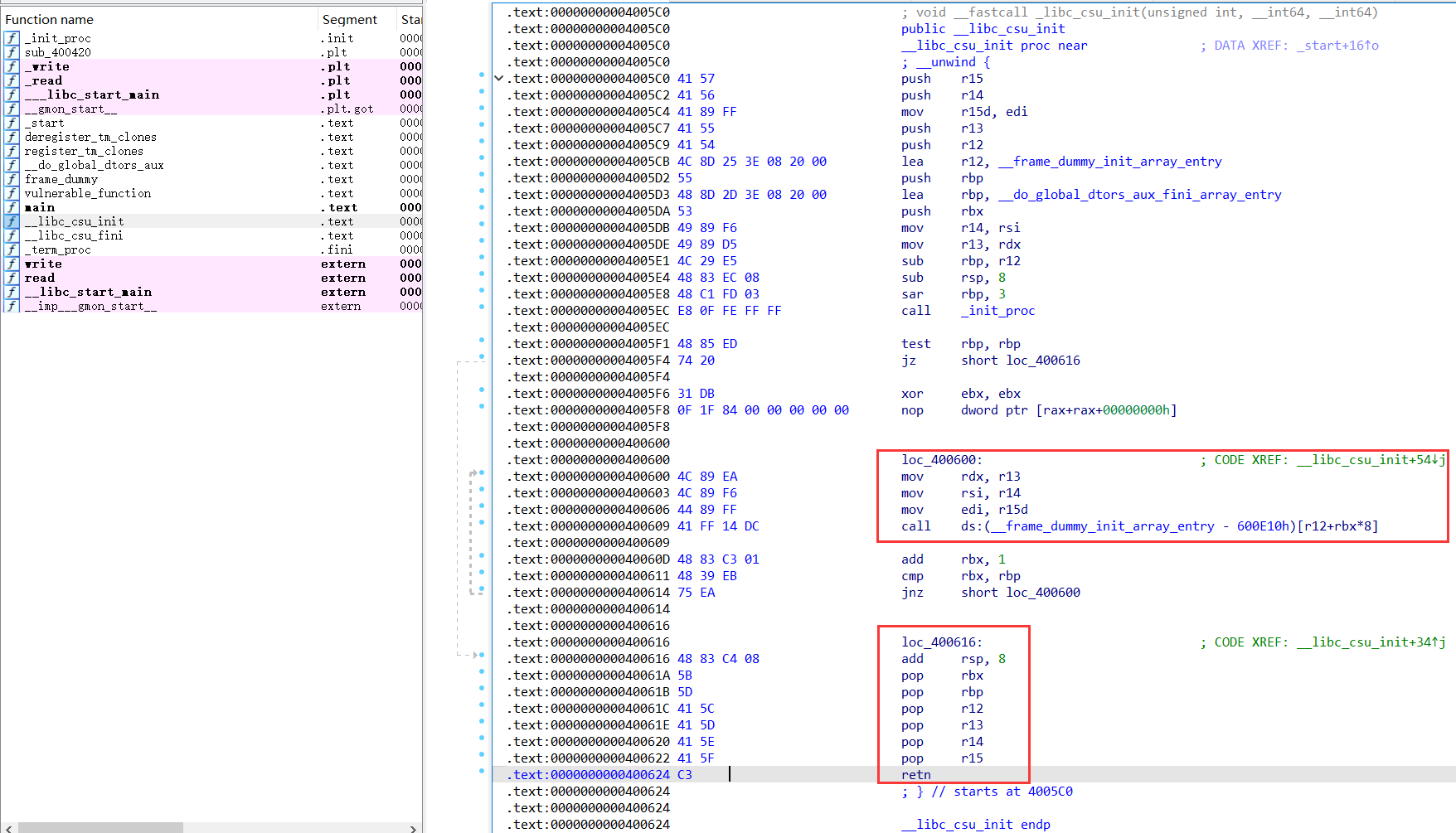


图4-2 \_\_libc\_csu\_init汇编代码

这里我们可以利用以下几个地方：

1. 从0x000000000040061A一直到结尾，我们可以利用栈溢出构造栈上数据来控制rbx,rbp,r12,r13,r14,r15寄存器的数据。
2. 从0x0000000000400600到0x0000000000400609，我们可以将r13赋给rdx，将r14赋给rsi，将r15d赋给edi（rdi的低32位），而这三个寄存器，也是x64函数调用中传递的前三个寄存器。此外，如果我们可以合理地控制r12与rbx，那么我们就可以调用我们想要调用的函数。比如说我们可以控制rbx为0，r12为存储我们想要调用的函数的地址。
3. 从0x000000000040060D到0x0000000000400614，我们可以控制rbx与rbp的之间的关系为rbx+1 = rbp，这样我们就不会执行loc\_400600，进而可以继续执行下面的汇编程序。这里我们可以简单的设置rbx=0，rbp=1。

#### 4.1.2 exploit

首先检查程序二进制保护的开启状况：

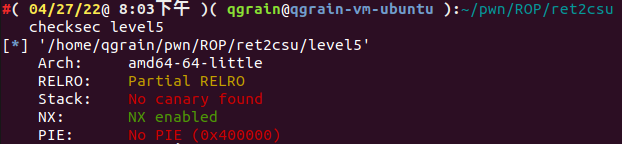


图4-3 level5二进制保护开启状况

由上图4-3可以发现是一个x64小端序的ELF文件，开启了NX执行保护，部分开启了RELRO，没有开启Canary栈保护和PIE保护。

IDA pro反编译level5，看到有一个命名意味很明显的函数：

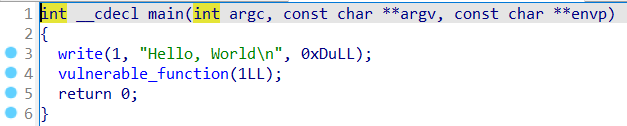


图4-4 level5 main函数

点进vulnerable\_function可以看到，有一个read函数。简单浏览完毕，发现程序中既没有system函数地址，也没有/bin/sh字符串，所以两者都需要我们自己去构造了。

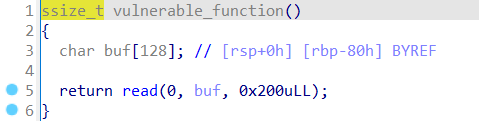


图4-5 level5 vulnerable\_function函数

由于read函数不会检查输入的字符串长度，因此可以利用其进行溢出。

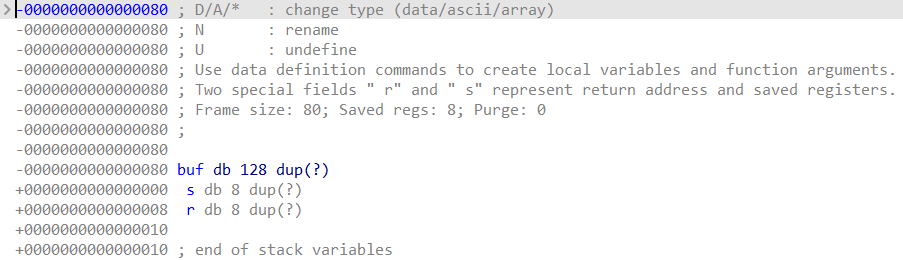


图4-6 stack of vulnerable\_function

可以看到该函数的栈空间大小为0x80h，再加上return处保存的rbp还有8字节，因此一共是0x88字节。

构造exp的思路如下：

1. 查找write，read函数的GOT表地址和bss段地址，以及main函数的地址。
2. 利用栈溢出执行libc\_csu\_init中的gadgets，获取 write 函数地址，并使得程序重新执行main函数
3. 通过write函数相对地址偏移计算libc基址，然后获取execve函数地址。(system据提示不可用)
4. 再次利用栈溢出执行libc\_csu\_init的gadgets，向bss段写入execve地址以及'/bin/sh’地址，并使得程序重新执行main函数。
5. 再次利用栈溢出执行libc\_csu\_init的gadgets执行execve('/bin/sh')获取shell。

完整的exp.py代码如下：



图4-7 ret2csu例题level5完整exp.py

运行exp.py得到结果如下，成功拿到/bin/sh：

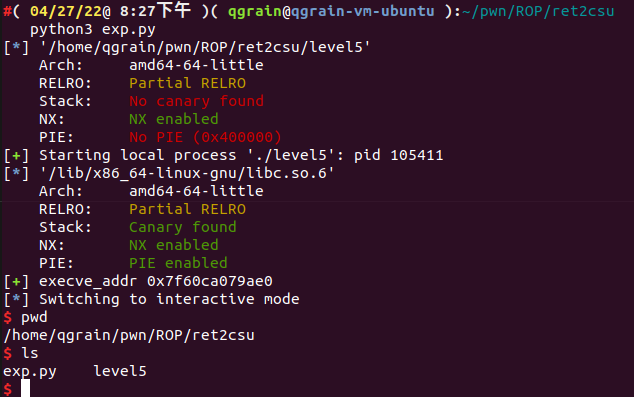


图4-8 ret2csu例题level5实验结果

## 5.实验总结

学习到了ROP的基本姿势。

## 6.附录

代码参见ROP目录