The elf in ELF

use 0-day(s) to cheat all disassemblers

david942j @ HITCON CMT 2018

Who Am I

- david942j
- 白帽駭客
  - 專精於 Linux 漏洞挖掘與逆向工程
- 國家米蟲
  - 專精於掃地拖地倒垃圾

This talk

- 3 tricks to cheat disassemblers
  - objdump, IDA Pro, etc.

取個名字

- 瞞天過海
  - IDA Pro's bug
- 天衣無縫
  - Linux kernel 0-day bug
- 偷天換日
  - Cheating ELF interpreter (ld.so)

這些漏洞

- What you see is NOT how it runs
- 反分析/反scanner
- anti-reverse-engineering

Introduction to ELF

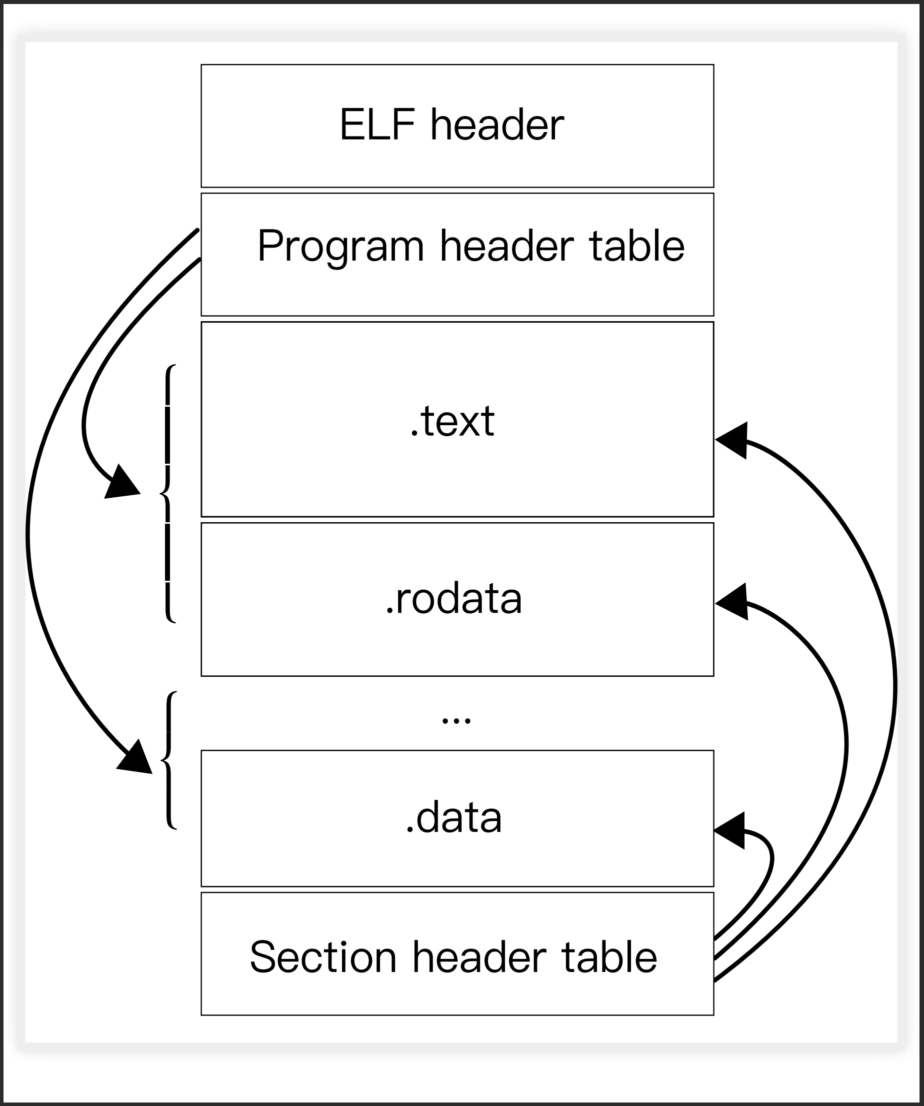
ELF

Executable and Linkable Format

- Linux 的執行檔格式

Header 有三種

- ELF header
- Program header
- Section header


ELF header
Program header table
.text
.rodata
...
.data
Section header table

ELF header

- ELF 的最前方
- 基本資訊
  - class: 32/64-bit
  - arch: x86/ARM/MIPS..
  - 標明 program/section header 的位置

Program header

- 執行時期 需要的資訊
- Needed Libraries, Segment Permissions, etc.

Section header

- Compile 時期需要的資訊 (static linker)
- 標記 ELF 中各區塊的用途
- `.text`, `.rodata`, etc.

In brief

- ELF header
  - mandatory
- Program header
  - Runtime 時要看的
- Section header
  - Compile time 時要看的

瞞天過海

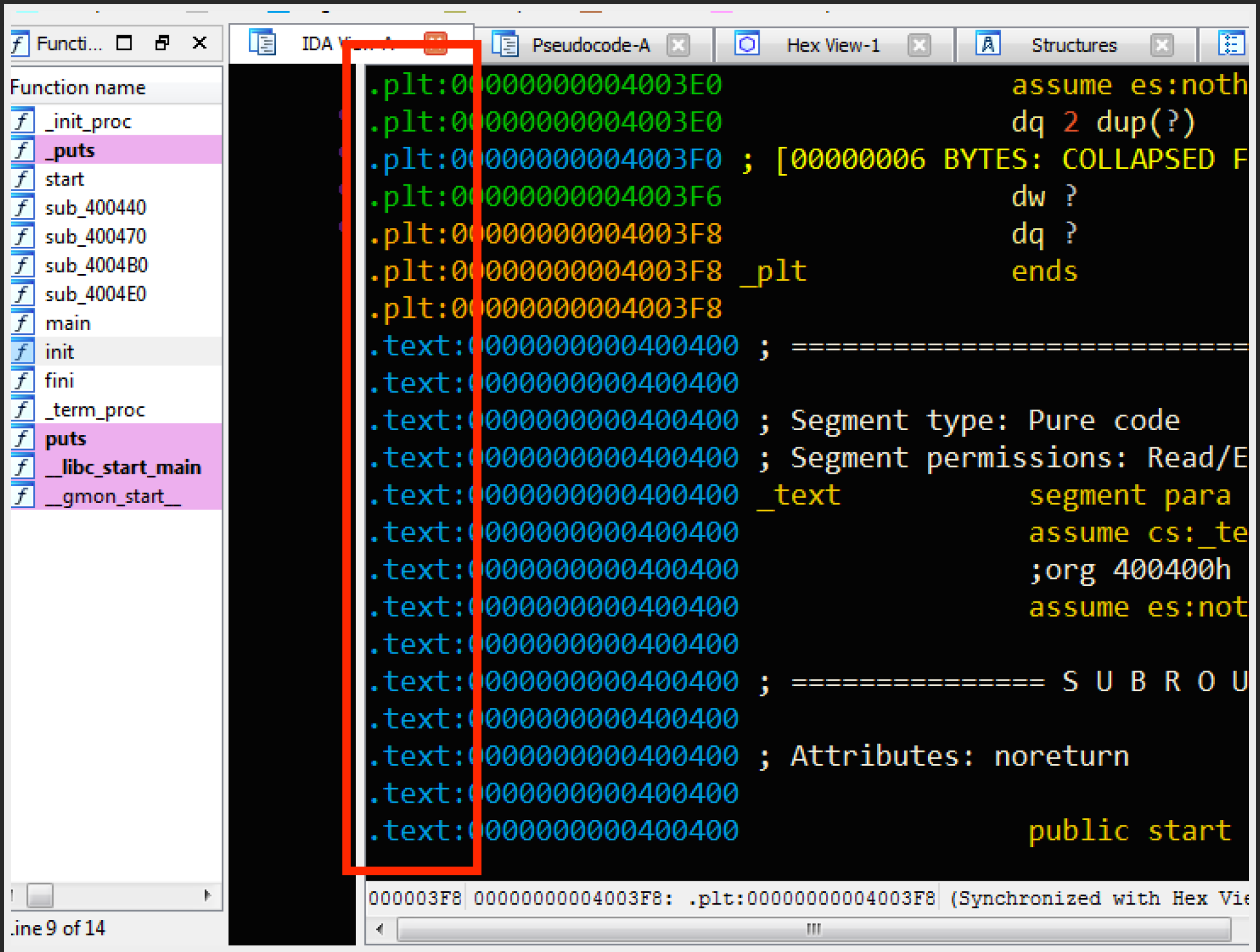
Idea

- Section header 在執行時期沒用
  ⇒ can be removed
  ⇒ can be forged

Forge section header

- Cheating objdump
- Cheating IDA Pro

IDA Pro considers sections



```
Function name
_init_proc
_puts
start
sub_400440
sub_400470
sub_4004B0
sub_4004E0
main
init
fini
_term_proc
puts
__libc_start_main
__gmon_start__
```

```
.plt:00000000004003E0                 assume es:noth
.plt:00000000004003E0                 dq 2 dup(?)
.plt:00000000004003F0 ; [00000006 BYTES: COLLAPSED F
.plt:00000000004003F6                 dw ?
.plt:00000000004003F8                 dq ?
.plt:00000000004003F8 _plt            ends
.plt:00000000004003F8
.text:0000000000400400 ; ===========================
.text:0000000000400400
.text:0000000000400400 ; Segment type: Pure code
.text:0000000000400400 ; Segment permissions: Read/E
.text:0000000000400400 _text           segment para
.text:0000000000400400                 assume cs:_te
.text:0000000000400400                 ;org 400400h
.text:0000000000400400                 assume es:not
.text:0000000000400400
.text:0000000000400400 ; =============== S U B R O U
.text:0000000000400400
.text:0000000000400400 ; Attributes: noreturn
.text:0000000000400400
.text:0000000000400400                 public start
```

```
000003F8 00000000004003F8: .plt:00000000004003F8 (Synchronized with Hex Vie
```

ine 9 of 14

.text

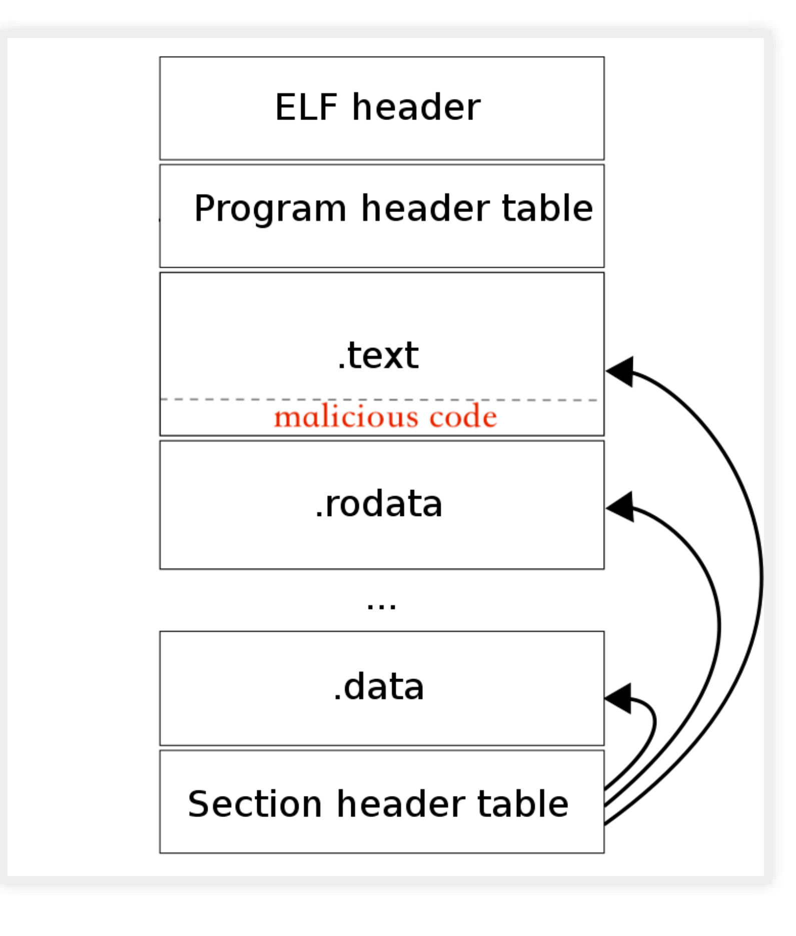
`.text`

- user 寫的 code 都在這
- IDA Pro 反組譯 `.text`

想法

- *縮小* `.text` 的範圍

Shrink `.text`



ELF header

Program header table

.text

malicious code

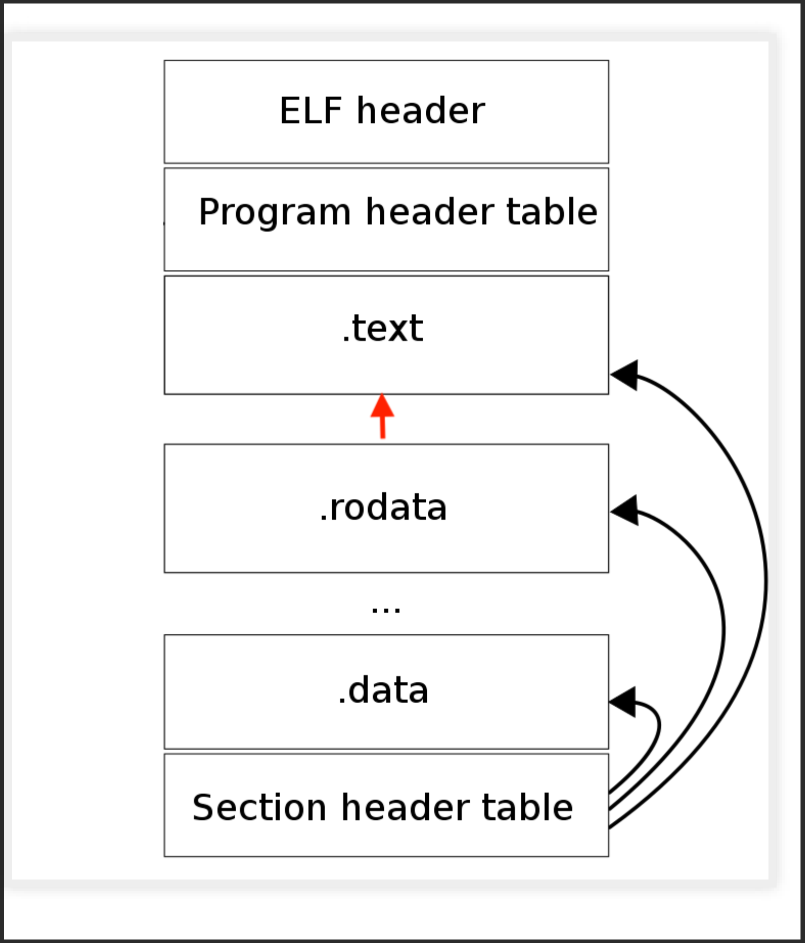
.rodata

...

.data

Section header table

→



ELF header

Program header table

.text

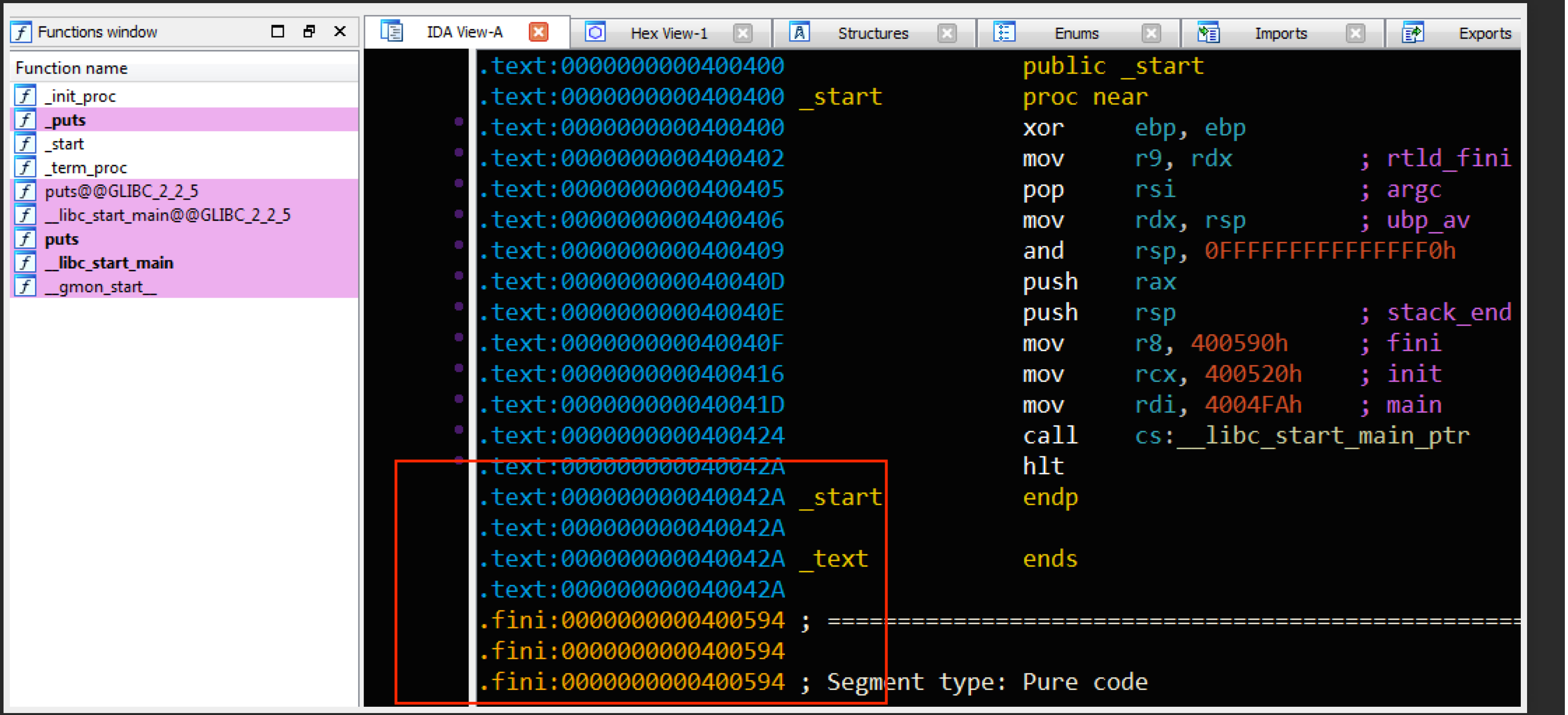
.rodata

...

.data

Section header table

σﾟ∀ﾟ)ﾟ∀ﾟ)σ



Function name
- _init_proc
- _puts
- _start
- _term_proc
- puts@@GLIBC_2_2_5
- __libc_start_main@@GLIBC_2_2_5
- puts
- __libc_start_main
- __gmon_start__

```
.text:0000000000400400                          public _start
.text:0000000000400400 _start                   proc near
.text:0000000000400400                          xor     ebp, ebp
.text:0000000000400402                          mov     r9, rdx         ; rtld_fini
.text:0000000000400405                          pop     rsi             ; argc
.text:0000000000400406                          mov     rdx, rsp        ; ubp_av
.text:0000000000400409                          and     rsp, 0FFFFFFFFFFFFFFF0h
.text:000000000040040D                          push    rax
.text:000000000040040E                          push    rsp             ; stack_end
.text:000000000040040F                          mov     r8, 400590h     ; fini
.text:0000000000400416                          mov     rcx, 400520h    ; init
.text:000000000040041D                          mov     rdi, 4004FAh    ; main
.text:0000000000400424                          call    cs:__libc_start_main_ptr
.text:000000000040042A                          hlt
.text:000000000040042A _start                   endp
.text:000000000040042A
.text:000000000040042A _text                    ends
.text:000000000040042A
.fini:0000000000400594 ; ==========================================================
.fini:0000000000400594
.fini:0000000000400594 ; Segment type: Pure code
```

But...

暗黑 code 被藏起來

總會有呼叫暗黑 code 的地方

如何藏呼叫的地方

爛招: 藏木於林

編一個有夠大的 binary 就找不到呼叫的地方

好招

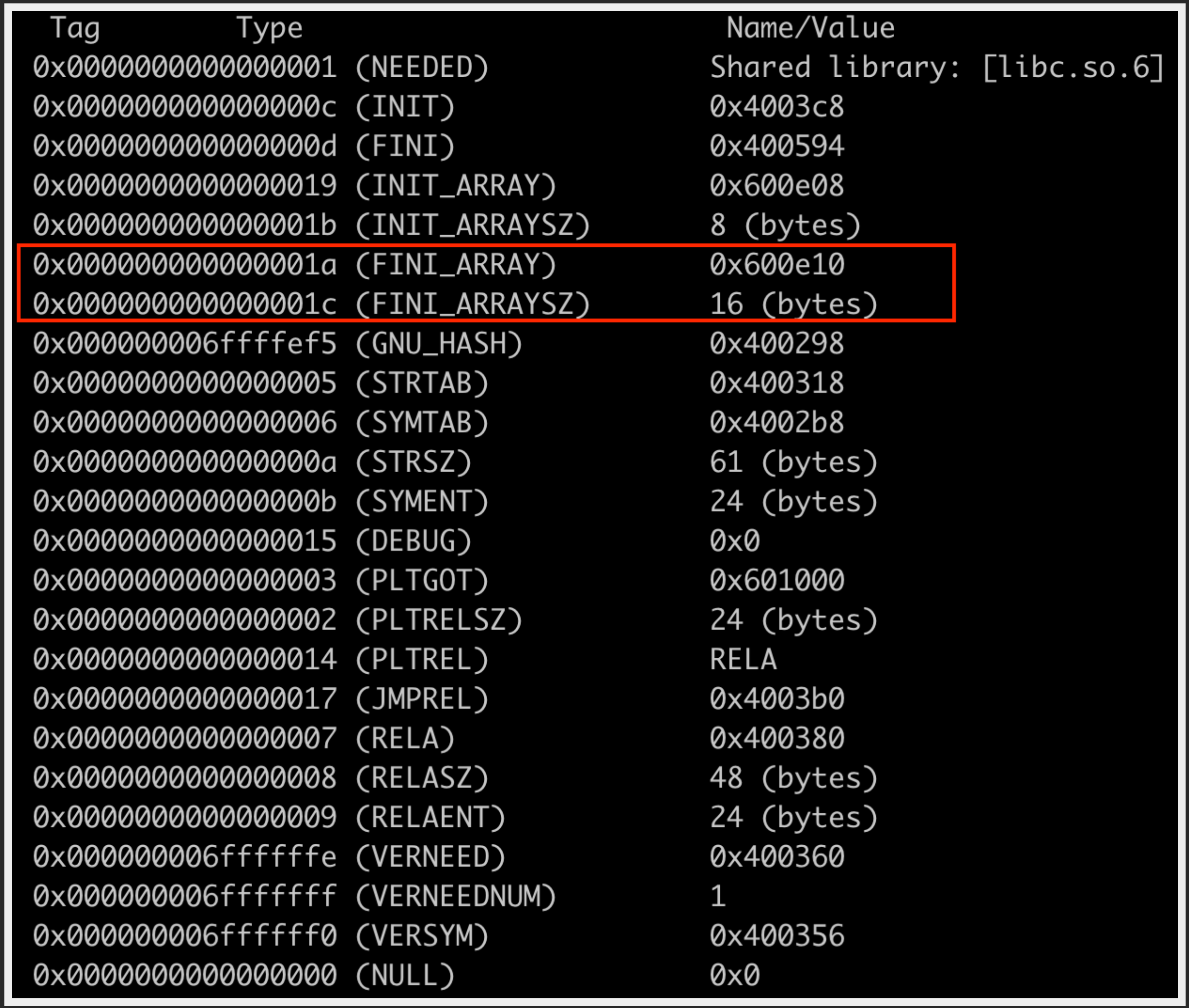
利用 `.init_array/.fini_array`

INIT / FINI_ARRAY

- Array of function pointers
- before / after `main` 會呼叫

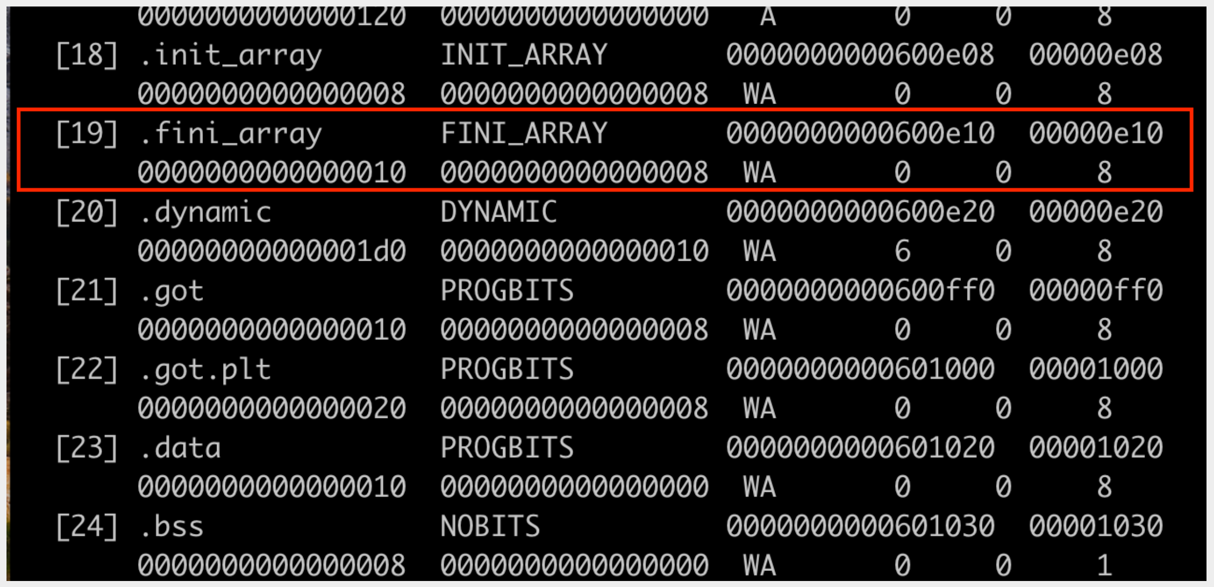
```c
#include <stdio.h>
__attribute__((constructor)) void before() {
  puts("Before main");
}
__attribute__((destructor)) void after() {
  puts("After main");
}
int main() {
  puts("Hi");
  return 0;
}
```

In program header→dynamic_tag



```
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4003c8
 0x000000000000000d (FINI)               0x400594
 0x0000000000000019 (INIT_ARRAY)         0x600e08
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x600e10
 0x000000000000001c (FINI_ARRAYSZ)       16 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x400318
 0x0000000000000006 (SYMTAB)             0x4002b8
 0x000000000000000a (STRSZ)              61 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x601000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4003b0
 0x0000000000000007 (RELA)               0x400380
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400360
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x400356
 0x0000000000000000 (NULL)               0x0
```

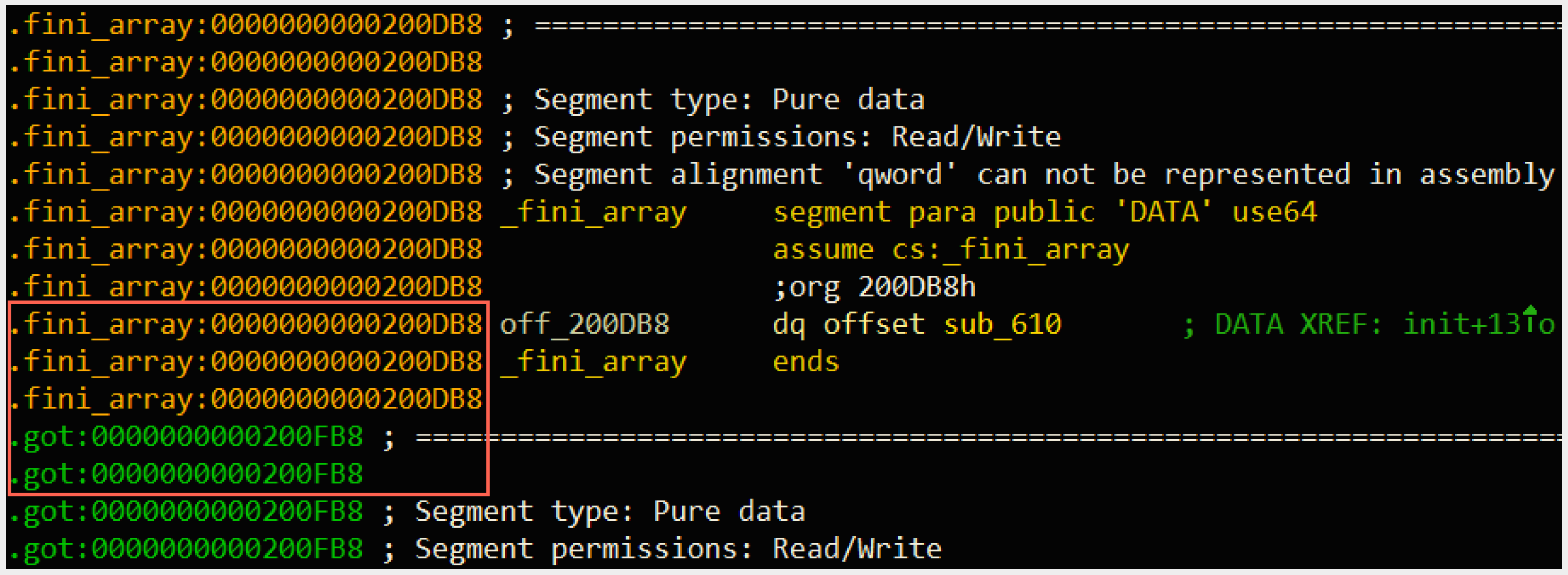
In section header



```
       0000000000000120  0000000000000000   A       0     0     8
  [18] .init_array       INIT_ARRAY       0000000000600e08  00000e08
       0000000000000008  0000000000000008  WA       0     0     8
  [19] .fini_array       FINI_ARRAY       0000000000600e10  00000e10
       0000000000000010  0000000000000008  WA       0     0     8
  [20] .dynamic          DYNAMIC          0000000000600e20  00000e20
       00000000000001d0  0000000000000010  WA       6     0     8
  [21] .got              PROGBITS         0000000000600ff0  00000ff0
       0000000000000010  0000000000000008  WA       0     0     8
  [22] .got.plt          PROGBITS         0000000000601000  00001000
       0000000000000020  0000000000000008  WA       0     0     8
  [23] .data             PROGBITS         0000000000601020  00001020
       0000000000000010  0000000000000000  WA       0     0     8
  [24] .bss              NOBITS           0000000000601030  00001030
       0000000000000008  0000000000000000  WA       0     0     1
```

與 `.text` 一樣可以縮短

Shrink `.fini_array`'s size



```
.fini_array:0000000000200DB8 ; ===============================================================
.fini_array:0000000000200DB8
.fini_array:0000000000200DB8 ; Segment type: Pure data
.fini_array:0000000000200DB8 ; Segment permissions: Read/Write
.fini_array:0000000000200DB8 ; Segment alignment 'qword' can not be represented in assembly
.fini_array:0000000000200DB8 _fini_array     segment para public 'DATA' use64
.fini_array:0000000000200DB8                 assume cs:_fini_array
.fini_array:0000000000200DB8                 ;org 200DB8h
.fini_array:0000000000200DB8 off_200DB8      dq offset sub_610       ; DATA XREF: init+13↑o
.fini_array:0000000000200DB8 _fini_array     ends
.fini_array:0000000000200DB8
.got:0000000000200FB8 ; ======================================================================
.got:0000000000200FB8
.got:0000000000200FB8 ; Segment type: Pure data
.got:0000000000200FB8 ; Segment permissions: Read/Write
```

瞞天過海

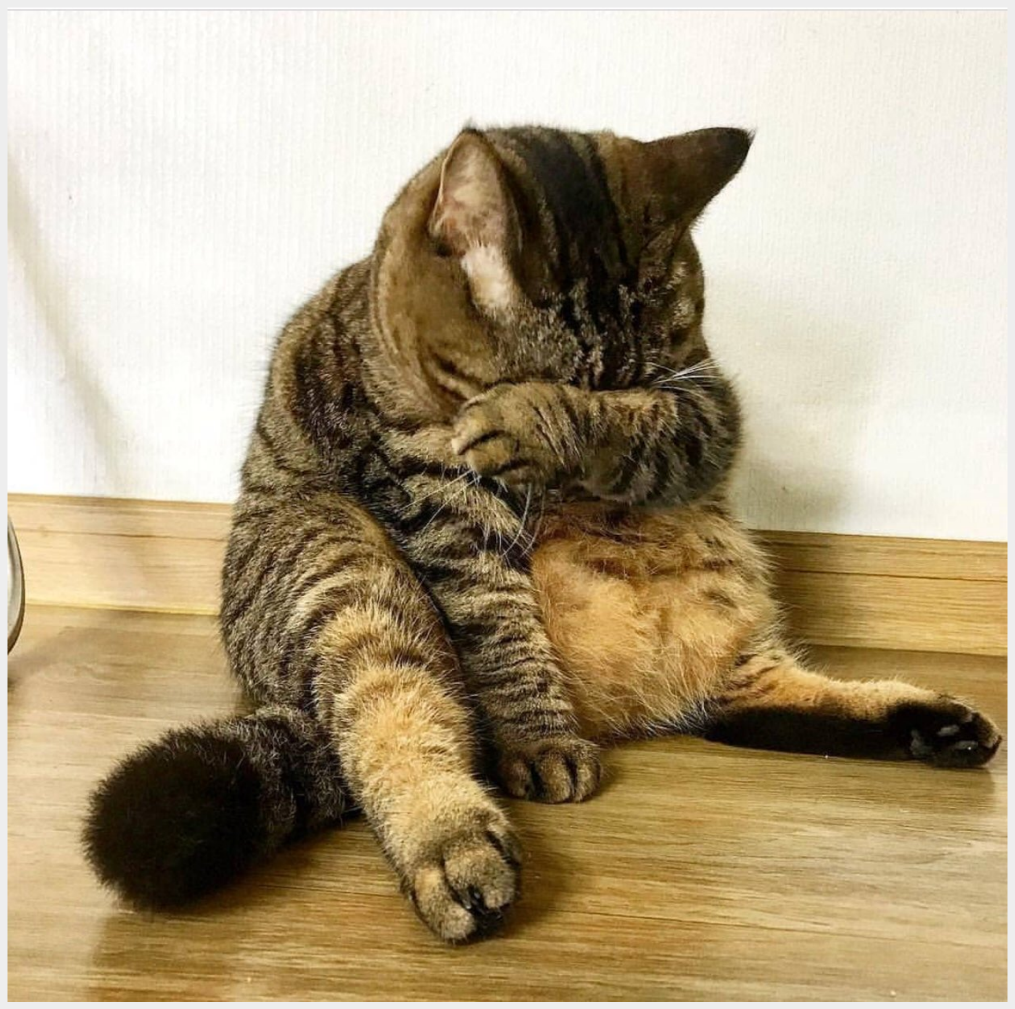
1. 放暗黑 code 在 `.text` 的底部
2. 讓 `FINI_ARRAY` 的 entry 指向暗黑 code
3. 縮短 `.text` & `.fini_array`
4. 在 `main` 結束後自動呼叫

Demo?

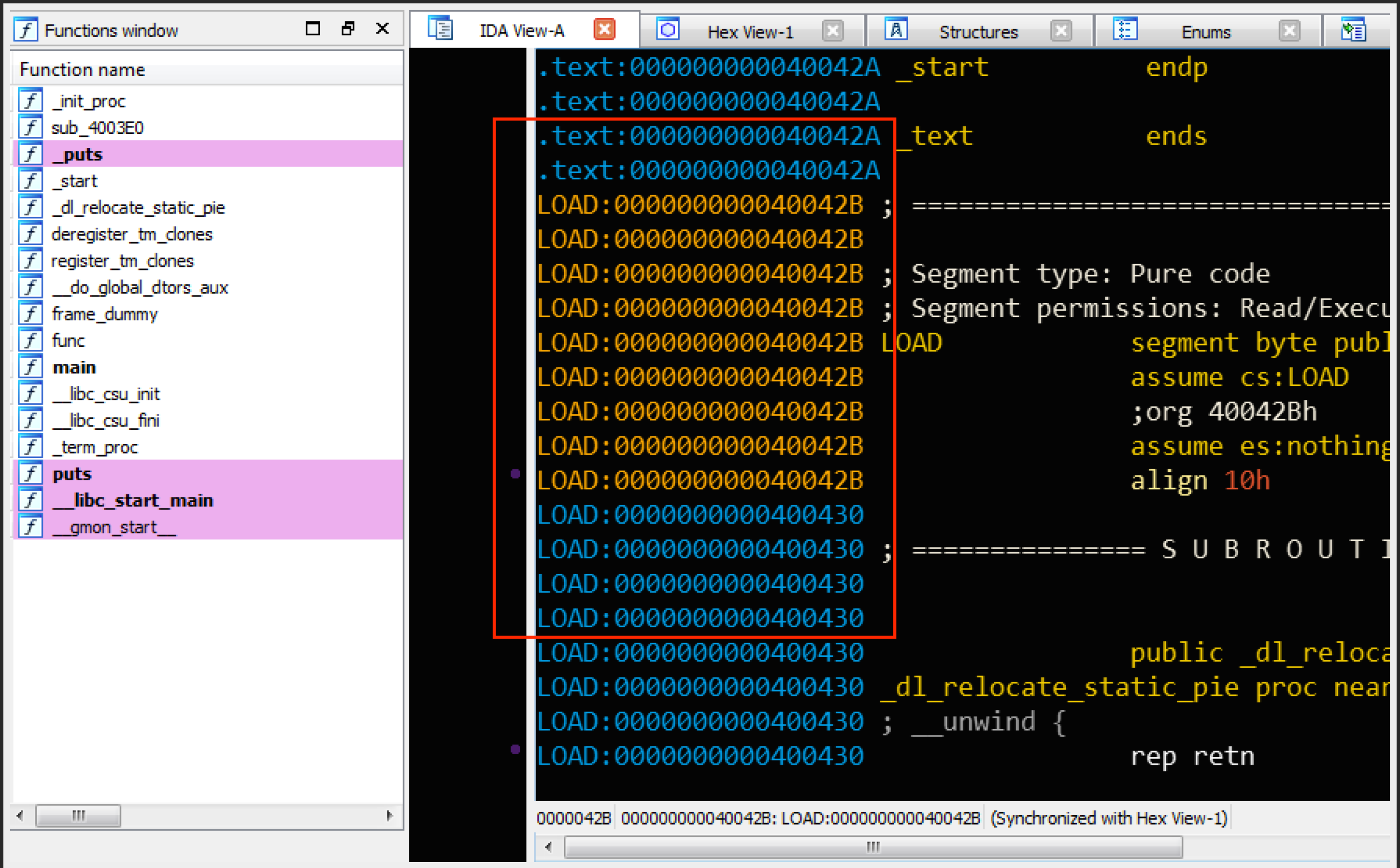
剛好 IDA Pro 出新版

Try newer version of IDA Pro

(TдT)



IDA Pro 7.0



Function name

- _init_proc
- sub_4003E0
- _puts
- _start
- _dl_relocate_static_pie
- deregister_tm_clones
- register_tm_clones
- __do_global_dtors_aux
- frame_dummy
- func
- main
- __libc_csu_init
- __libc_csu_fini
- _term_proc
- puts
- __libc_start_main
- __gmon_start__

```
.text:000000000040042A _start          endp
.text:000000000040042A
.text:000000000040042A _text           ends
.text:000000000040042A
LOAD:000000000040042B ; ===============================
LOAD:000000000040042B
LOAD:000000000040042B ; Segment type: Pure code
LOAD:000000000040042B ; Segment permissions: Read/Execu
LOAD:000000000040042B LOAD             segment byte publ
LOAD:000000000040042B                  assume cs:LOAD
LOAD:000000000040042B                  ;org 40042Bh
LOAD:000000000040042B                  assume es:nothing
LOAD:000000000040042B                  align 10h
LOAD:0000000000400430
LOAD:0000000000400430 ; =============== S U B R O U T I
LOAD:0000000000400430
LOAD:0000000000400430
LOAD:0000000000400430                  public _dl_reloca
LOAD:0000000000400430 _dl_relocate_static_pie proc near
LOAD:0000000000400430 ; __unwind {
LOAD:0000000000400430                  rep retn
```

0000042B 000000000040042B: LOAD:000000000040042B (Synchronized with Hex View-1)

IDA Pro 7.0

- uses `LOAD` instead of `.text`
- Bug fixed QQ

瞞天過海 is dead

IDA Pro 6.x	IDA Pro 7.0
✅	❌

瞞天過海 2

瞞天過海 2

- IDA Pro 未解析 relocation 在 `.init_array`/`.fini_array` 的資訊

Relocation?

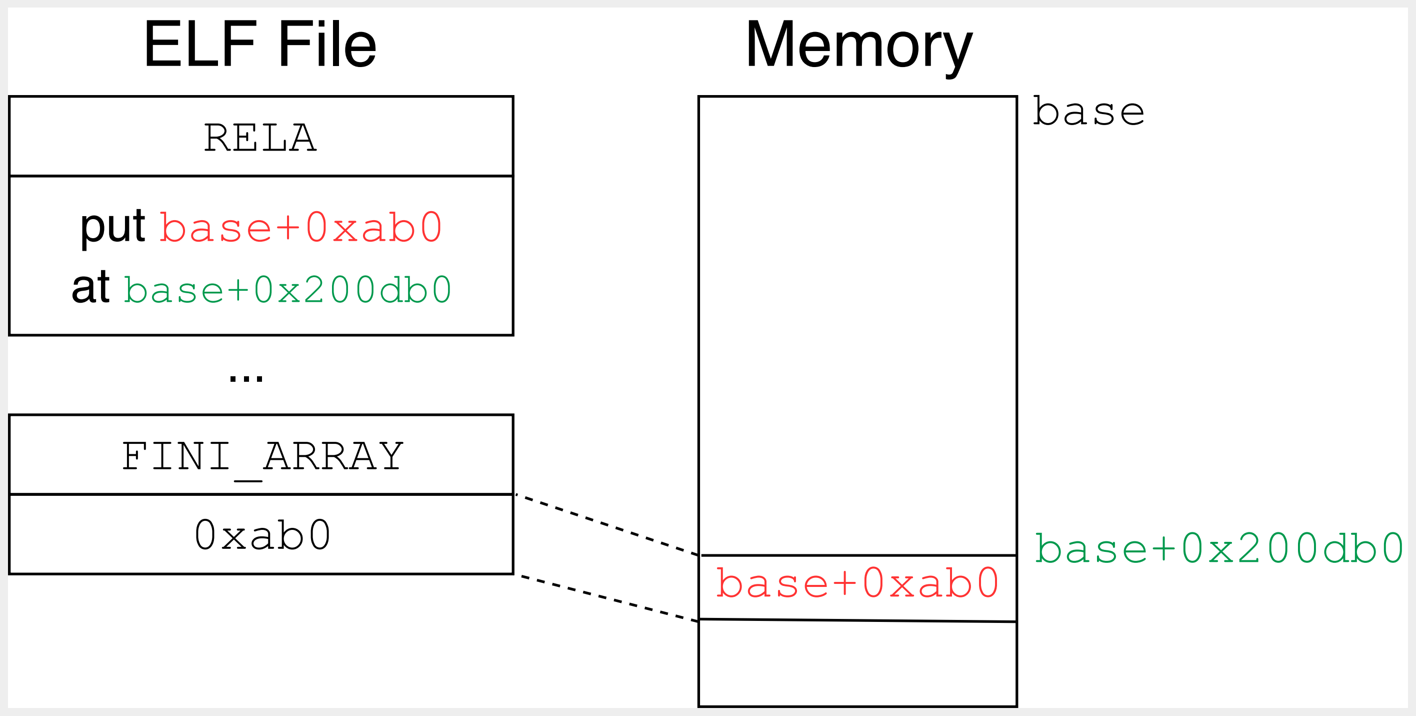
Relocation

- `phdr→DYNAMIC` 裡的表
- 種類很多種
- 處理執行時期才知道的記憶體位址

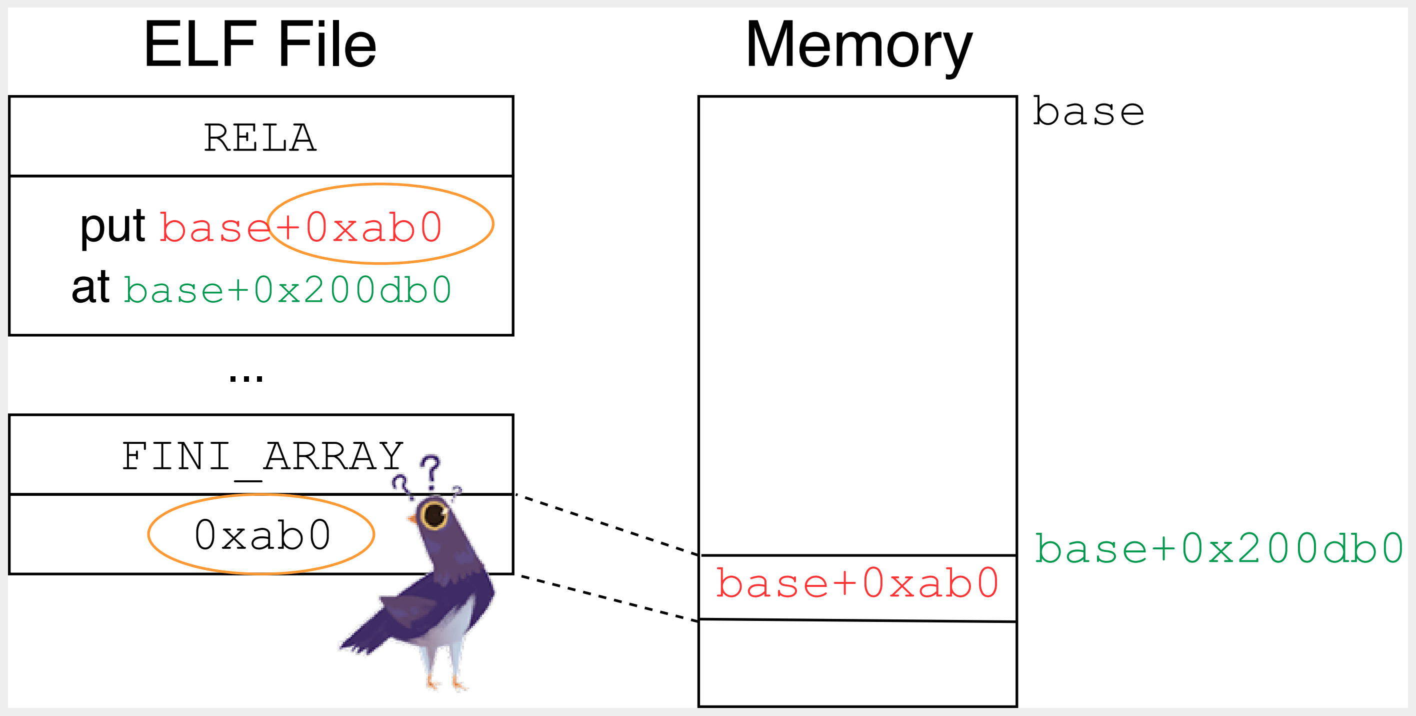
以 `FINI_ARRAY` 為例

- 有開 PIE (position-independent executable)
    - 執行檔本身的基底位址隨機
- `FINI_ARRAY` 上的值要執行時期才知道
- ld.so 根據 relocation table 將正確的 function 位址放上 `FINI_ARRAY`

Relocation of `FINI_ARRAY`



Relocation of `FINI_ARRAY`



Value of `FINI_ARRAY` means nothing

relocation is the boss

瞞天過海 2

IDA Pro only uses value on `FINI_ARRAY`!

於是

- 實際呼叫的函式跟看起來的不同(!)
- 感謝 IDA Pro 的努力

But..

IDA Pro 7.0 後 LOAD 段都被解析

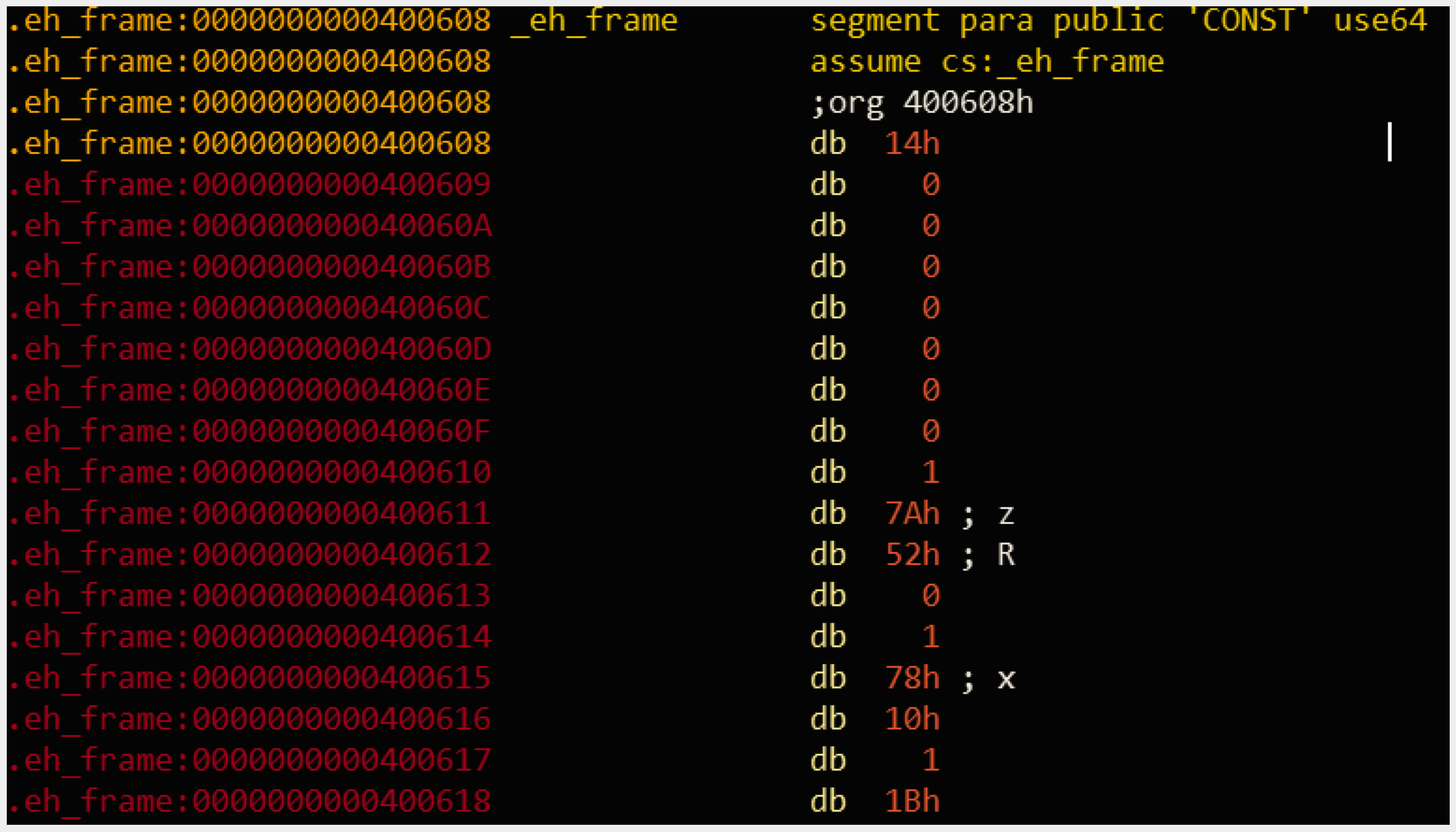
We have arbitrary function call

Where to put malicious code?

在沒用的(?) section 藏 code

- `.eh_frame`
    - Error Handling
- Who care error handling
- 至少 0x100 byte
    - 長度正相關於 #func
- Nice to hide code

Normal `.eh_frame` looks like



```
.eh_frame:0000000000400608 _eh_frame       segment para public 'CONST' use64
.eh_frame:0000000000400608                 assume cs:_eh_frame
.eh_frame:0000000000400608                 ;org 400608h
.eh_frame:0000000000400608                 db  14h
.eh_frame:0000000000400609                 db    0
.eh_frame:000000000040060A                 db    0
.eh_frame:000000000040060B                 db    0
.eh_frame:000000000040060C                 db    0
.eh_frame:000000000040060D                 db    0
.eh_frame:000000000040060E                 db    0
.eh_frame:000000000040060F                 db    0
.eh_frame:0000000000400610                 db    1
.eh_frame:0000000000400611                 db  7Ah ; z
.eh_frame:0000000000400612                 db  52h ; R
.eh_frame:0000000000400613                 db    0
.eh_frame:0000000000400614                 db    1
.eh_frame:0000000000400615                 db  78h ; x
.eh_frame:0000000000400616                 db  10h
.eh_frame:0000000000400617                 db    1
.eh_frame:0000000000400618                 db  1Bh
```

瞞天過海2

1. 放後門在`.eh_frame`
2. 竄改 relocation table 使 `FINI_ARRAY` 指在後門
3. `main` 結束後呼叫後門

HITCON CTF Quals 2017

`void`

天衣無縫

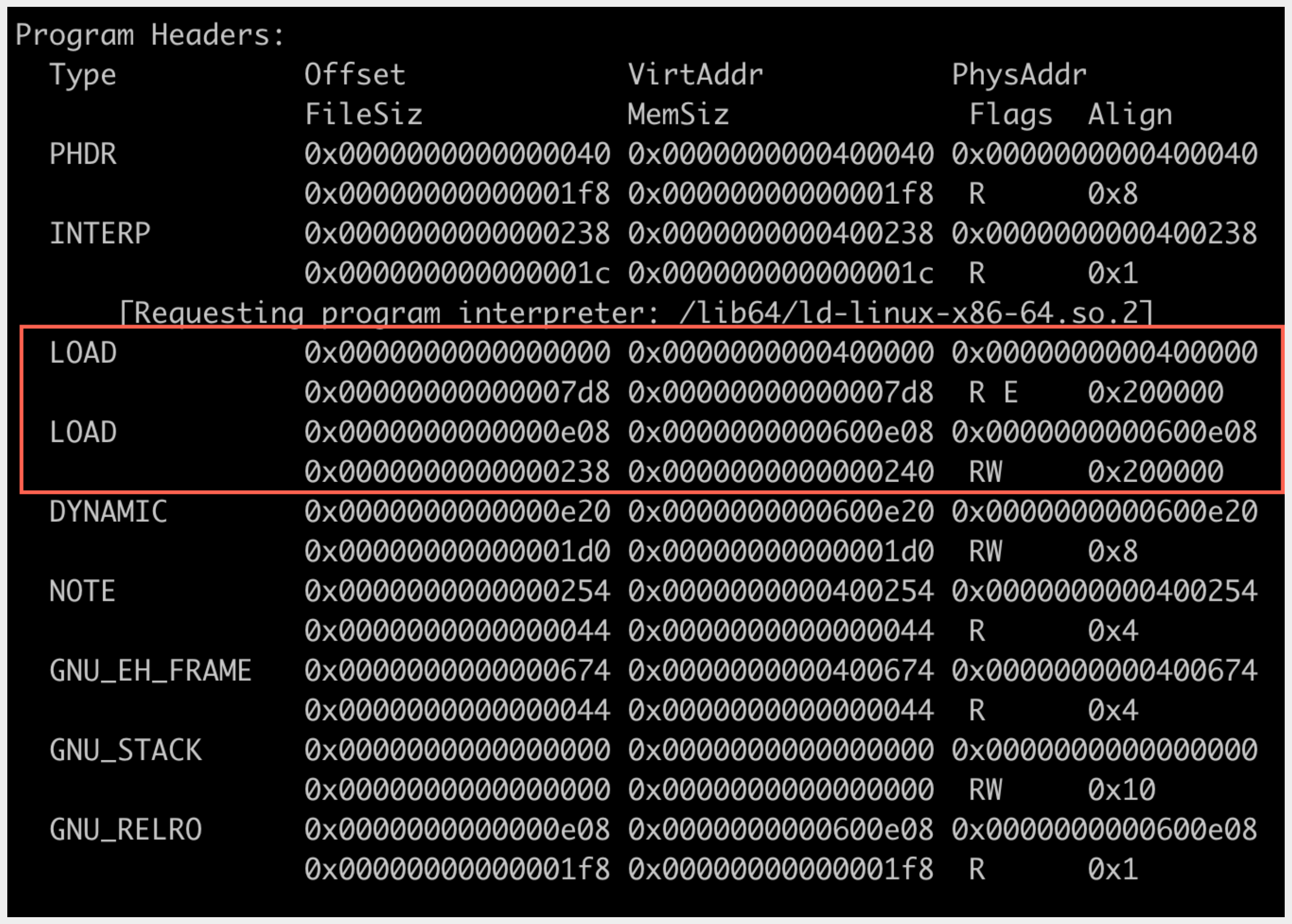
The Linux 0-day bug

談一下 PT_LOAD

`PT_LOAD`

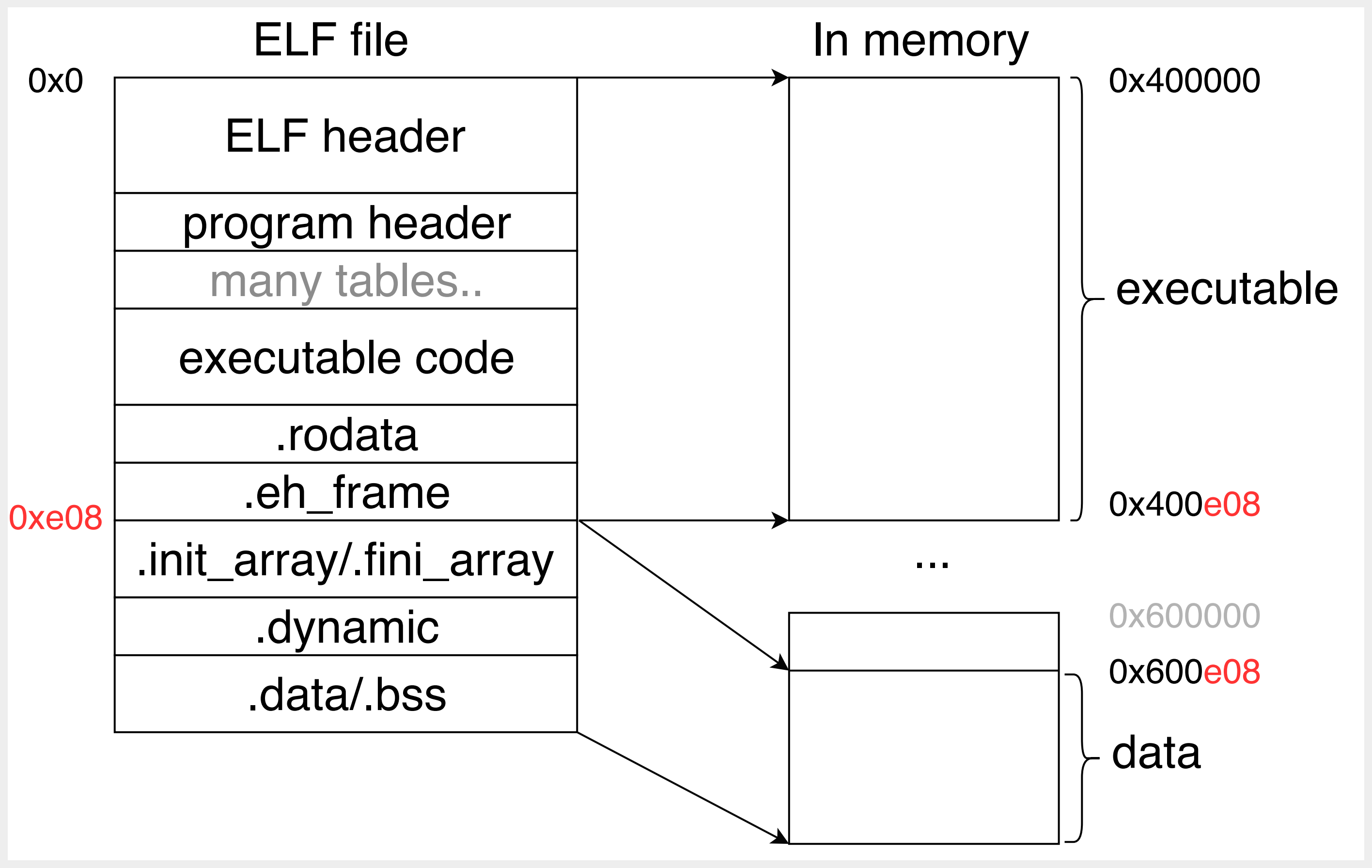
- 描述如何將 ELF 檔案映射到 memory
- 一般會有兩個 `PT_LOAD` entry

PT_LOAD



```
Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R      0x8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      0x1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x00000000000007d8 0x00000000000007d8  R E    0x200000
  LOAD           0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x0000000000000238 0x0000000000000240  RW     0x200000
  DYNAMIC        0x0000000000000e20 0x0000000000600e20 0x0000000000600e20
                 0x00000000000001d0 0x00000000000001d0  RW     0x8
  NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_EH_FRAME   0x0000000000000674 0x0000000000400674 0x0000000000400674
                 0x0000000000000044 0x0000000000000044  R      0x4
  GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
                 0x0000000000000000 0x0000000000000000  RW     0x10
  GNU_RELRO      0x0000000000000e08 0x0000000000600e08 0x0000000000600e08
                 0x00000000000001f8 0x00000000000001f8  R      0x1
```

Memory mapping



execve

linux/fs/binfmt_elf.c#load_elf_binary

#load_elf_binary

- Read and check ELF header
- Parse program header
  - `PT_INTERP`
  - `PT_LOAD`
  - `PT_GNU_STACK`
- Setup `AUXV`

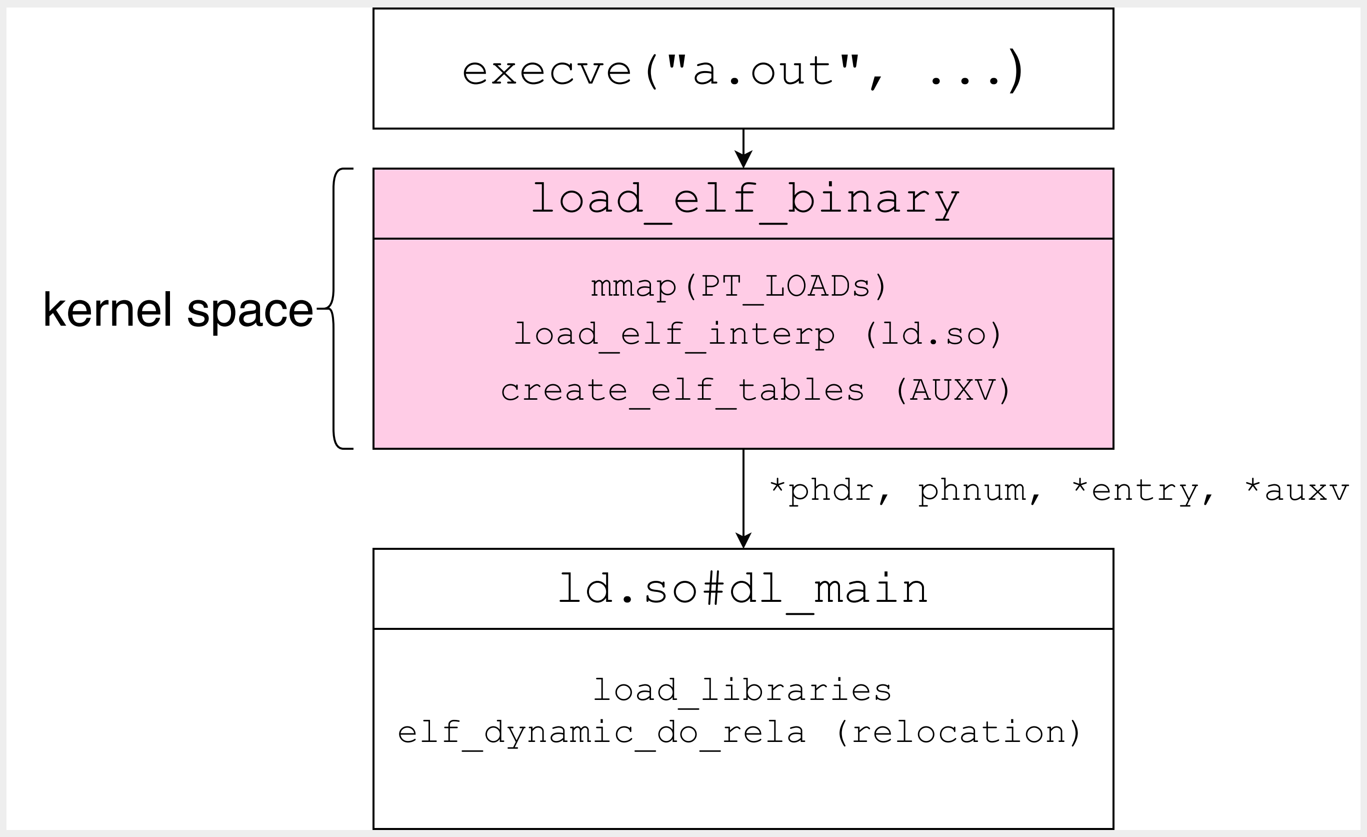
AUXV

AUXiliary Vector

傳遞一些資訊給 interpreter(ld.so)

- `AT_PHDR`
- `AT_ENTRY`
- `AT_UID`
- ...

Flow of `execve`

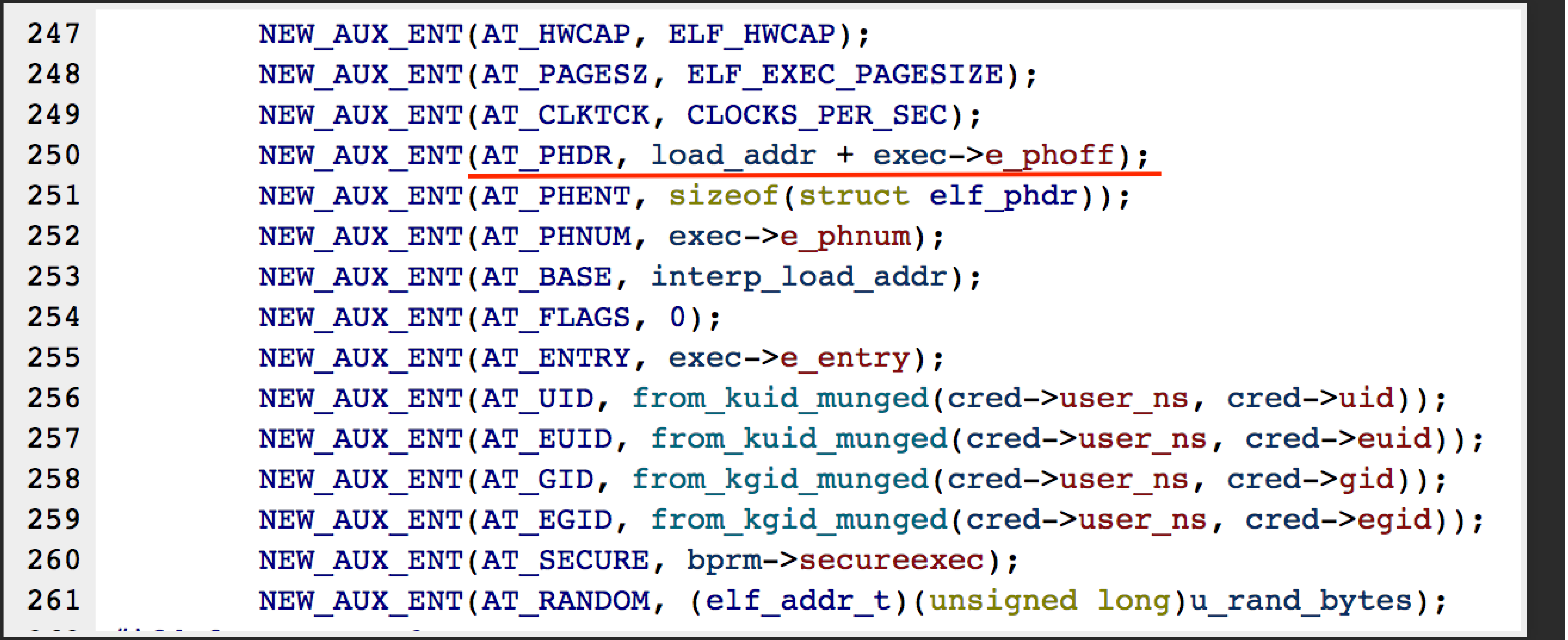


Bug🐛

- Kernel 計算 `AT_PHDR` 的方式不正確

洞

binfmt_elf.c#create_elf_tables



```
        NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
        NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
        NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
        NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
        NEW_AUX_ENT(AT_PHENT, sizeof(struct elf_phdr));
        NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
        NEW_AUX_ENT(AT_BASE, interp_load_addr);
        NEW_AUX_ENT(AT_FLAGS, 0);
        NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
        NEW_AUX_ENT(AT_UID, from_kuid_munged(cred->user_ns, cred->uid));
        NEW_AUX_ENT(AT_EUID, from_kuid_munged(cred->user_ns, cred->euid));
        NEW_AUX_ENT(AT_GID, from_kgid_munged(cred->user_ns, cred->gid));
        NEW_AUX_ENT(AT_EGID, from_kgid_munged(cred->user_ns, cred->egid));
        NEW_AUX_ENT(AT_SECURE, bprm->secureexec);
        NEW_AUX_ENT(AT_RANDOM, (elf_addr_t)(unsigned long)u_rand_bytes);
```

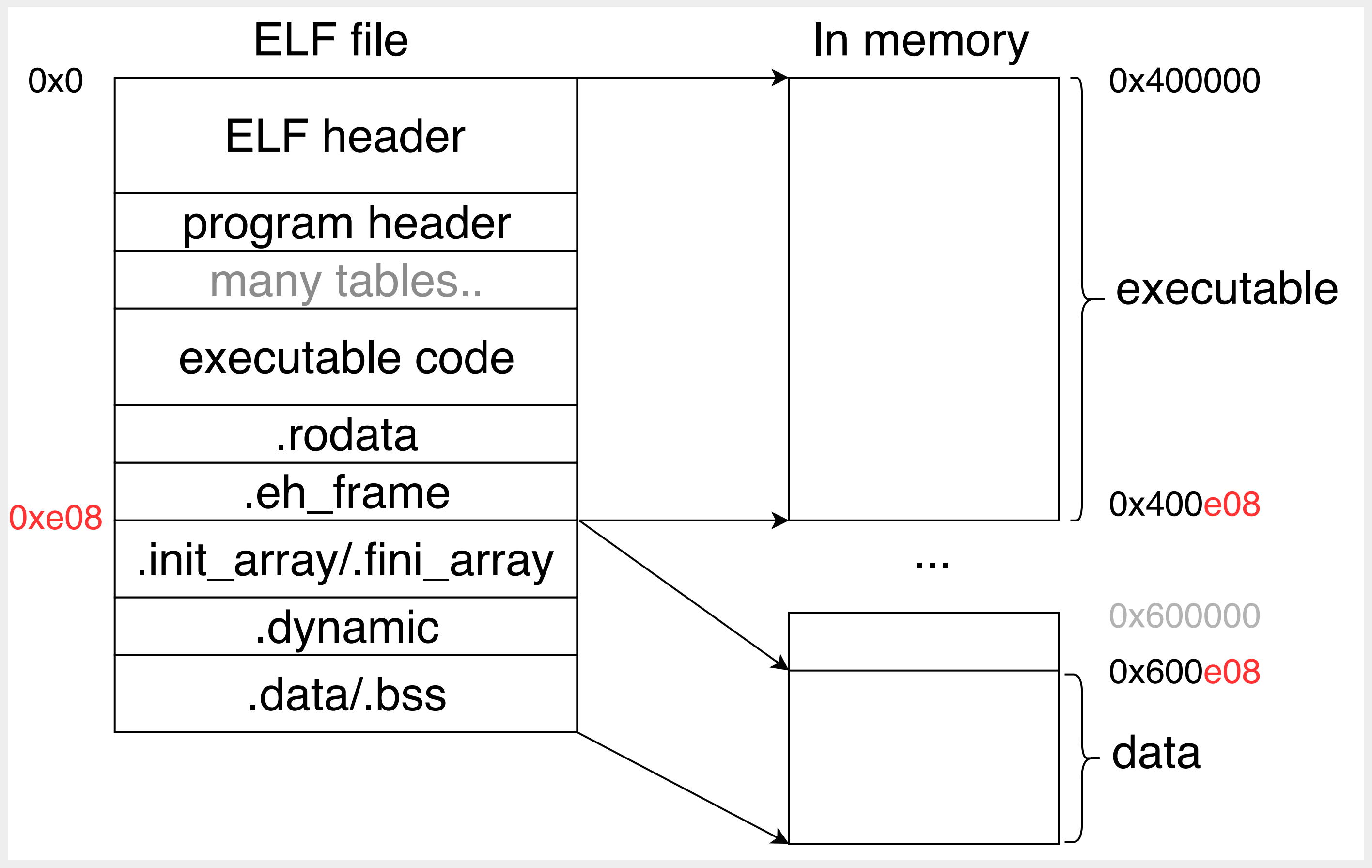
Normally

load_addr	exec->e_phoff	
0x400000	0x40	0x400040

`load_addr` is

The *first* LOADed address

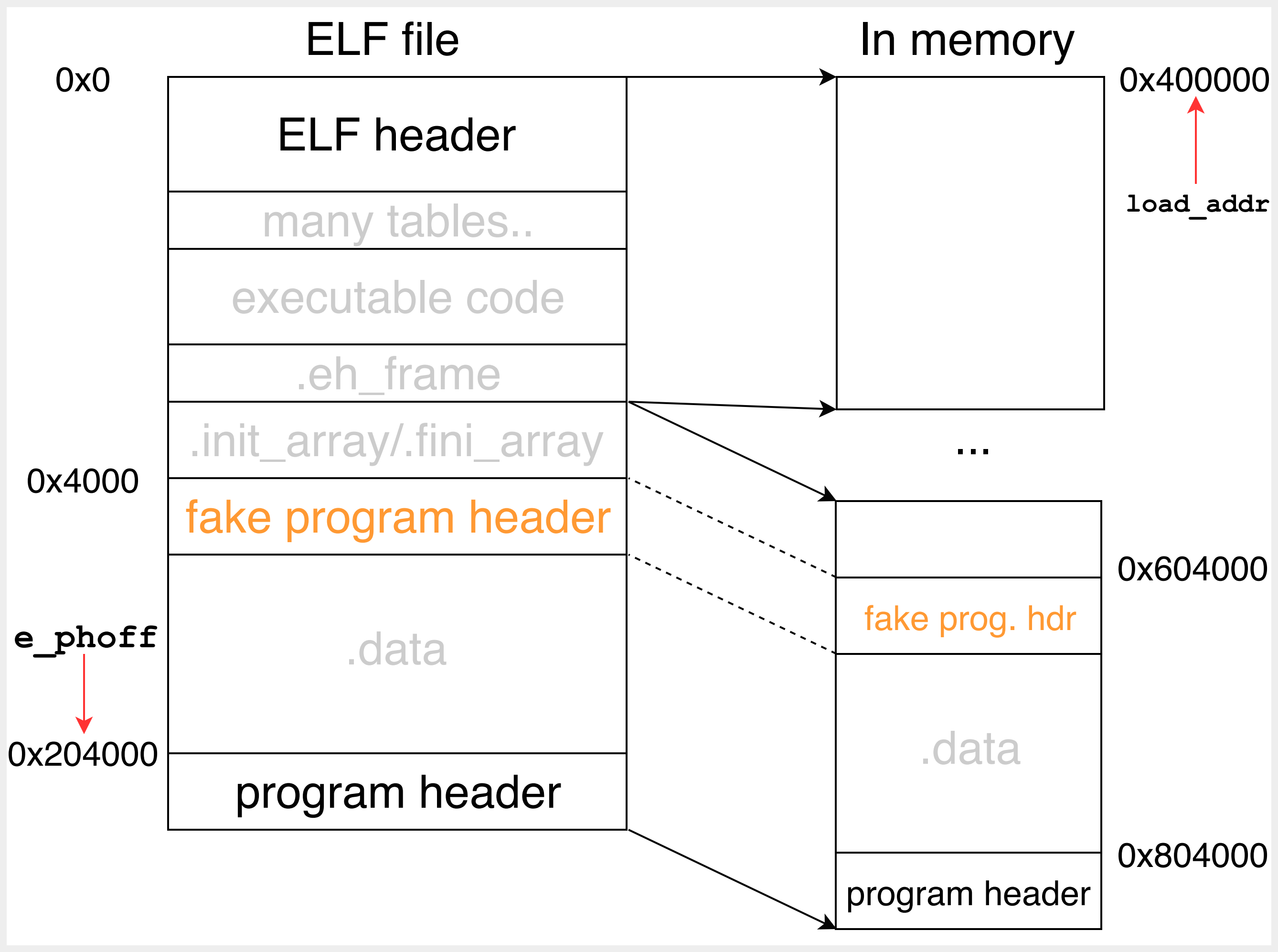
再看一次


ELF file
In memory
0x0
ELF header
program header
many tables..
executable code
.rodata
.eh_frame
0xe08
.init_array/.fini_array
.dynamic
.data/.bss
0x400000
executable
0x400e08
...
0x600000
0x600e08
data

Nobody promises PHDR is located in the *first* `PT_LOAD`

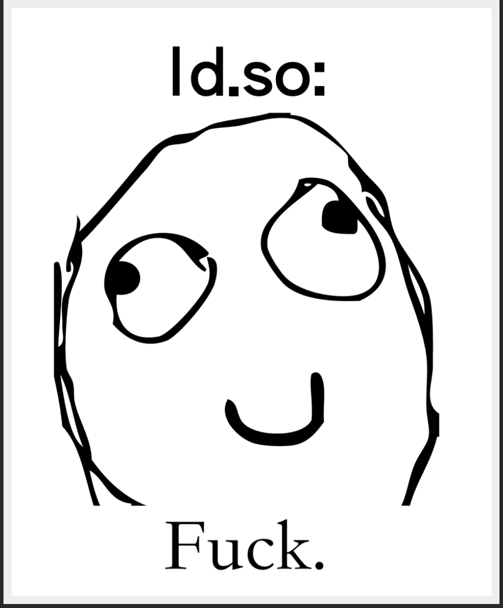
Put PHDR in the second `PT_LOAD`

天衣無縫


ELF file
In memory
0x0
ELF header
many tables..
executable code
.eh_frame
.init_array/.fini_array
0x4000
fake program header
e_phoff
.data
0x204000
program header
0x400000
load_addr
...
0x604000
fake prog. hdr
.data
0x804000
program header

Effect

- Kernel loads binary correctly
- While kernel cheats ld.so address of PHDR



因此

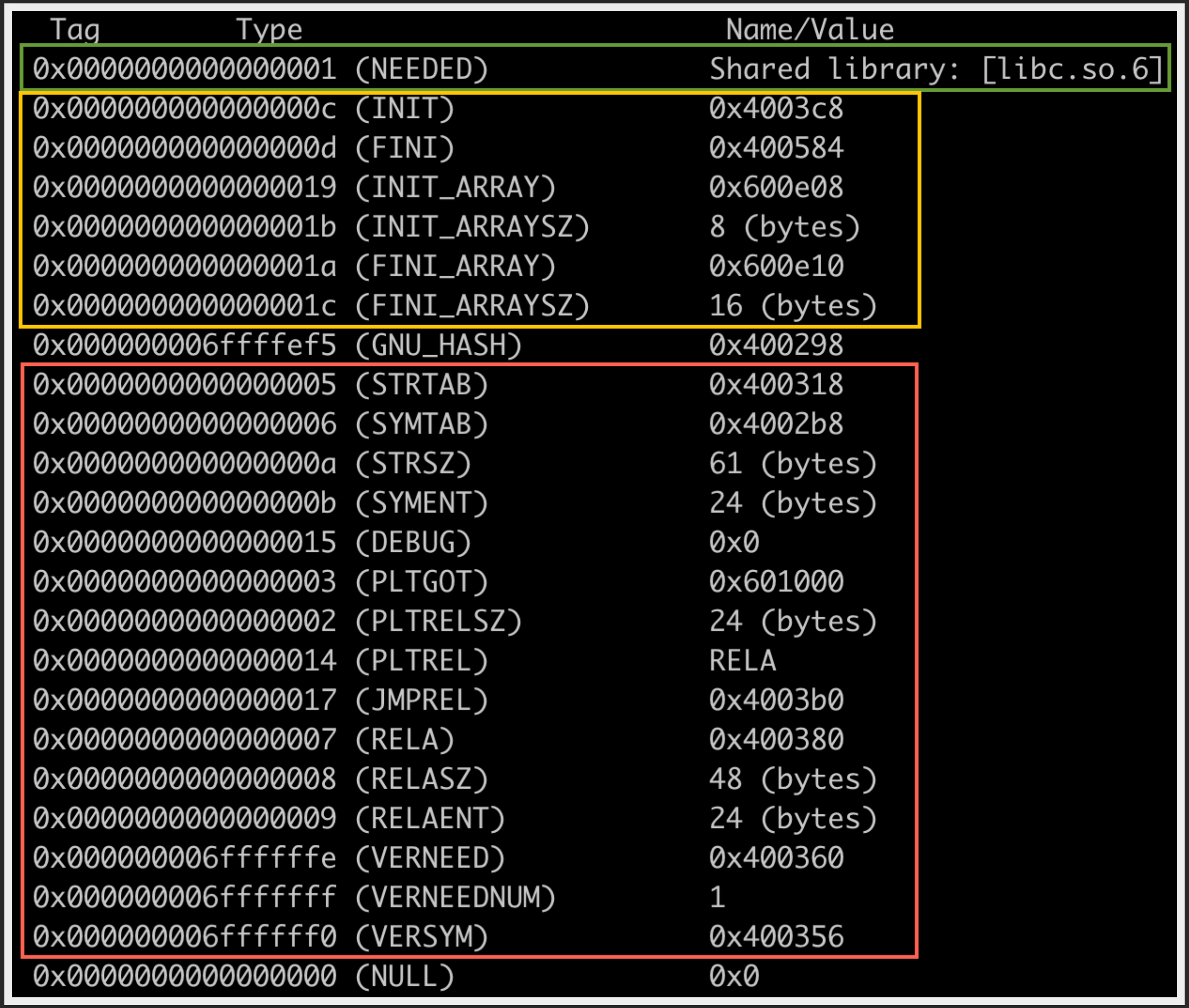
- ld.so 的行為跟反組譯工具預期完全不同

ld.so 會做什麼?

我們能騙什麼

- Load shared libraries
- Process dynamic relocation

Dynamic



```
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4003c8
 0x000000000000000d (FINI)               0x400584
 0x0000000000000019 (INIT_ARRAY)         0x600e08
 0x000000000000001b (INIT_ARRAYSZ)       8 (bytes)
 0x000000000000001a (FINI_ARRAY)         0x600e10
 0x000000000000001c (FINI_ARRAYSZ)       16 (bytes)
 0x000000006ffffef5 (GNU_HASH)           0x400298
 0x0000000000000005 (STRTAB)             0x400318
 0x0000000000000006 (SYMTAB)             0x4002b8
 0x000000000000000a (STRSZ)              61 (bytes)
 0x000000000000000b (SYMENT)             24 (bytes)
 0x0000000000000015 (DEBUG)              0x0
 0x0000000000000003 (PLTGOT)             0x601000
 0x0000000000000002 (PLTRELSZ)           24 (bytes)
 0x0000000000000014 (PLTREL)             RELA
 0x0000000000000017 (JMPREL)             0x4003b0
 0x0000000000000007 (RELA)               0x400380
 0x0000000000000008 (RELASZ)             48 (bytes)
 0x0000000000000009 (RELAENT)            24 (bytes)
 0x000000006ffffffe (VERNEED)            0x400360
 0x000000006fffffff (VERNEEDNUM)         1
 0x000000006ffffff0 (VERSYM)             0x400356
 0x0000000000000000 (NULL)               0x0
```

天衣無縫→瞞天過海2

- Forge relocation on `INIT_ARRAY/FINI_ARRAY`

做點更厲害的事情

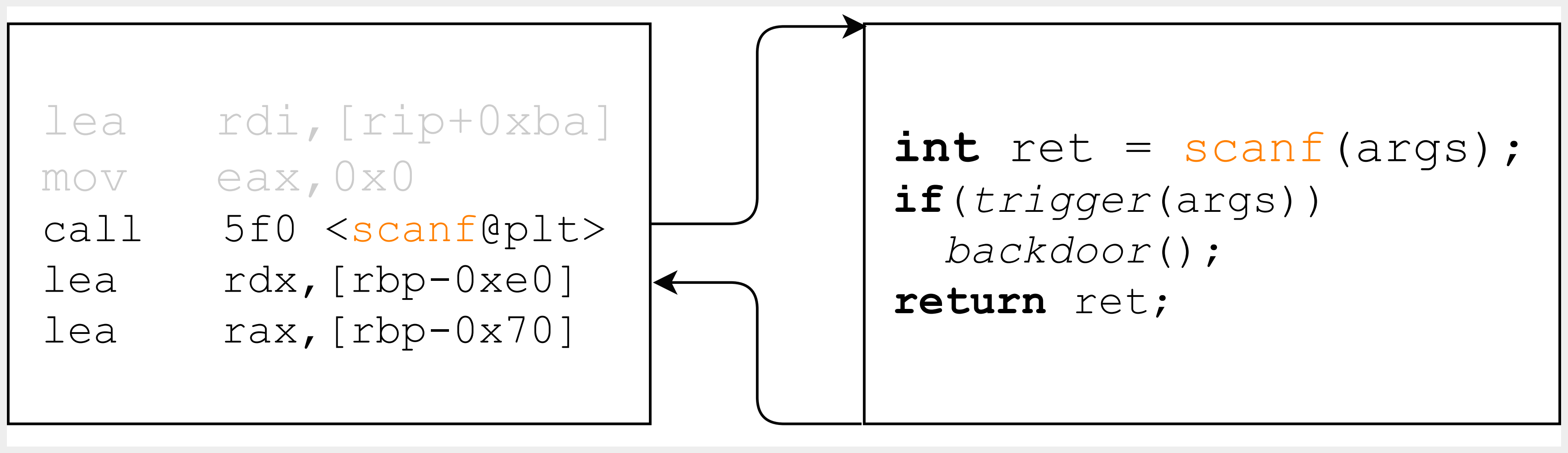
Relocation

- 也會用於呼叫 library 的函式
  - `printf/scanf`

假造 relocation table

- 以為即將 `scanf` 但其實跳後門
- 即使動態分析也不容易發現

後門



```
lea     rdi,[rip+0xba]
mov     eax,0x0
call    5f0 <scanf@plt>
lea     rdx,[rbp-0xe0]
lea     rax,[rbp-0x70]
```

```
int ret = scanf(args);
if(trigger(args))
  backdoor();
return ret;
```

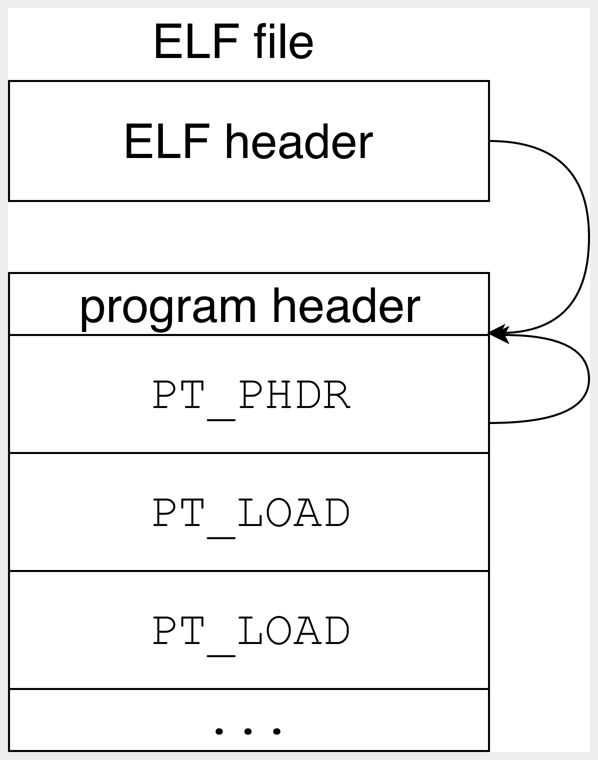
Demo

偷天換日

Let's play ld.so

`PT_PHDR` in PHDR

`PT_PHDR` points to itself



glibc/elf/rtld.c#1147

```
for (ph = phdr; ph < &phdr[phnum]; ++ph)
   switch (ph->p_type)
     {
     case PT_PHDR:
       /* Find out the load address.  */
       main_map->l_addr = phdr - ph->p_vaddr;
       break;
     case PT_DYNAMIC:
       /* This tells us where to find the dynamic section,
          which tells us everything we need to do.  */
       main_map->l_ld = main_map->l_addr + ph->p_vaddr;
       break;
```

Forge `PT_PHDR`

ld.so will completely misunderstand base of binary!

≈ 天衣無縫

Program header for kernel ≠ for ld.so

不好用?

- ld.so 誤會 binary 的基底位址
- 影響到的事情太多
    - 要修正非常多表的位址

原本的 program header

PT_PHDR	main_map->l_addr = phdr - ph->p_vaddr
PT_LOAD	
PT_LOAD	
PT_DYNAMIC	main_map->l_ld = main_map->l_addr + ph->p_vaddr
...	

偷天换日

Use two `PT_PHDR`

glibc/elf/rtld.c#1147

```
for (ph = phdr; ph < &phdr[phnum]; ++ph)
   switch (ph->p_type)
     {
     case PT_PHDR:
       /* Find out the load address.  */
       main_map->l_addr = phdr - ph->p_vaddr;
       break;
     case PT_DYNAMIC:
       /* This tells us where to find the dynamic section,
          which tells us everything we need to do.  */
       main_map->l_ld = main_map->l_addr + ph->p_vaddr;
       break;
```

偷天换日

PT_PHDR	main_map->l_addr = phdr - ph->p_vaddr
PT_DYNAMIC	**main_map->l_ld** = main_map->l_addr + ph->p_vaddr
PT_PHDR	main_map->l_addr = phdr - ph->p_vaddr
PT_LOAD	
PT_LOAD	
...	

偽造 dynamic

`INIT_ARRAY/FINI_ARRAY/Relocation`

≈ 天衣無縫

Conclusion

瞞天過海

1. IDA Pro trusts section header
2. Not using relocation for `INIT/FINI_ARRAY`

天衣無縫

Kernel calculates PHDR incorrectly

ld.so get wrong address

偷天换日

ld.so using `PT_PHDR` for calculating base address

Nobody checks correctness of `PT_PHDR`

三種技巧

- 漏洞切入點不同
- 能做到的事情幾乎沒有差別
  - 任意代碼執行

Demo

- Give me two ELFs
- Looks like A in IDA pro but actually B

david942j @

GitHub twitter Blogger facebook