The elf in ELF

use 0-day to cheat all disassemblers

david942j @ CyberSEC 2019

This talk

- Tricks to cheat disassemblers
  - objdump, readelf, IDA Pro, etc.

IDA Pro

- The best tool for reverse-engineering
- Take it as examples in this talk

anti-reverse-engineering

- 反-逆向工程
- What you see is NOT what it really is
- IDA Pro 裡看起來無害 → 實際是惡意程式

Introduction to ELF

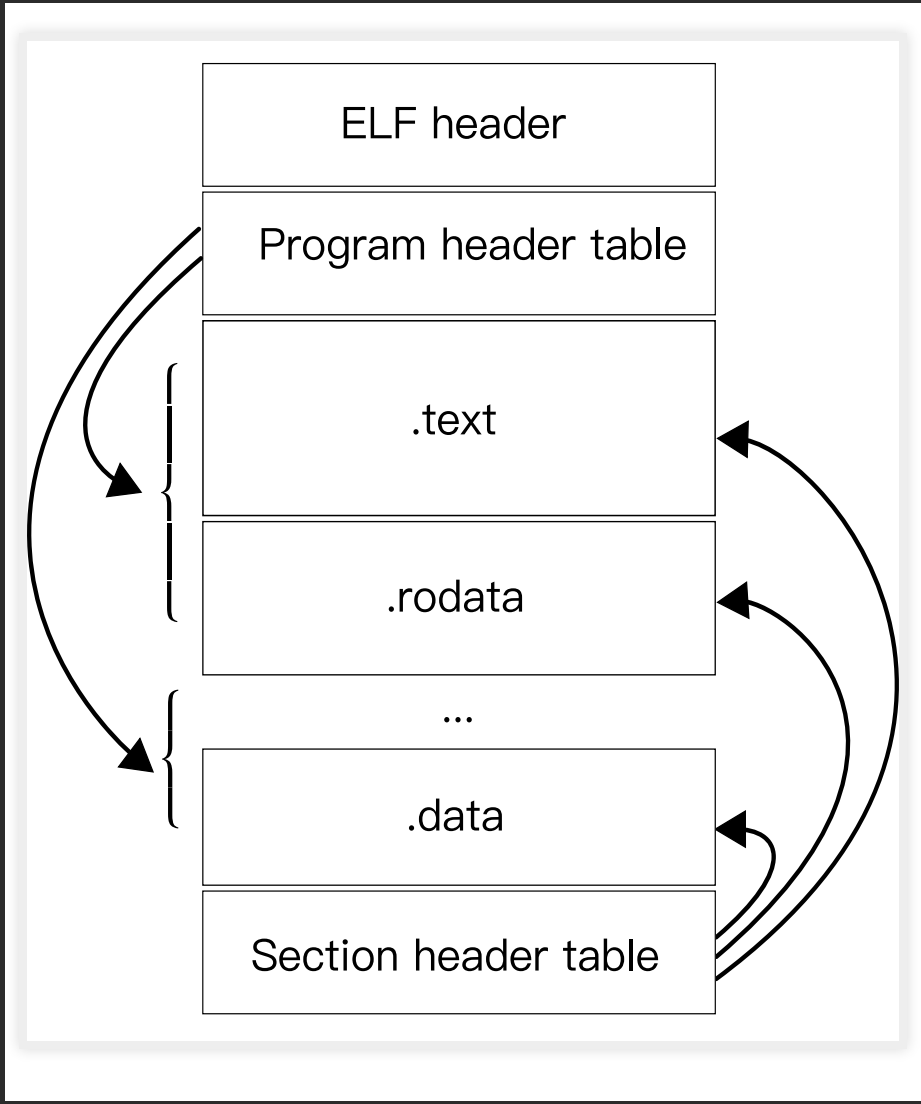
ELF

Executable and Linkable Format

- Linux 的執行檔格式

Header 有三個

- ELF header
- Section header (not important here)
- Program 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 section / program header 的位置

Section header

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

Program header

- 執行時期 需要的資訊
- Needed Libraries, Segment Permissions, etc.
- 包含一張 `_DYNAMIC` table

_DYNAMIC

- 最重要功能：描述「要找函式庫裡的哪些函式」

Example

```
#include <stdio.h>
#include <iostream>
using std::cout;

int main() {
  char s[100] = {};
  scanf("%99s", s);                    // libc.so.6#scanf
  cout << "Hello, " << s << "!\n"; // libstdc++.so.6#std::cout
  return 0;
}
```

`_DYNAMIC`

- Need libraries: libc.so.6, libstdc++.so.6
- Need functions: `scanf` & `std::cout`
- `ld.so` 根據 `_DYNAMIC` 去找 function 位址

In this talk

- 欺騙 IDA Pro 解析假的 `_DYNAMIC` table
- e.g. IDA Pro 覺得是 `printf` 但其實是 `system`
- 0-day bug in Linux kernel
- Bug(?) in ld.so

The Linux 0-day bug

談一下 PT_LOAD

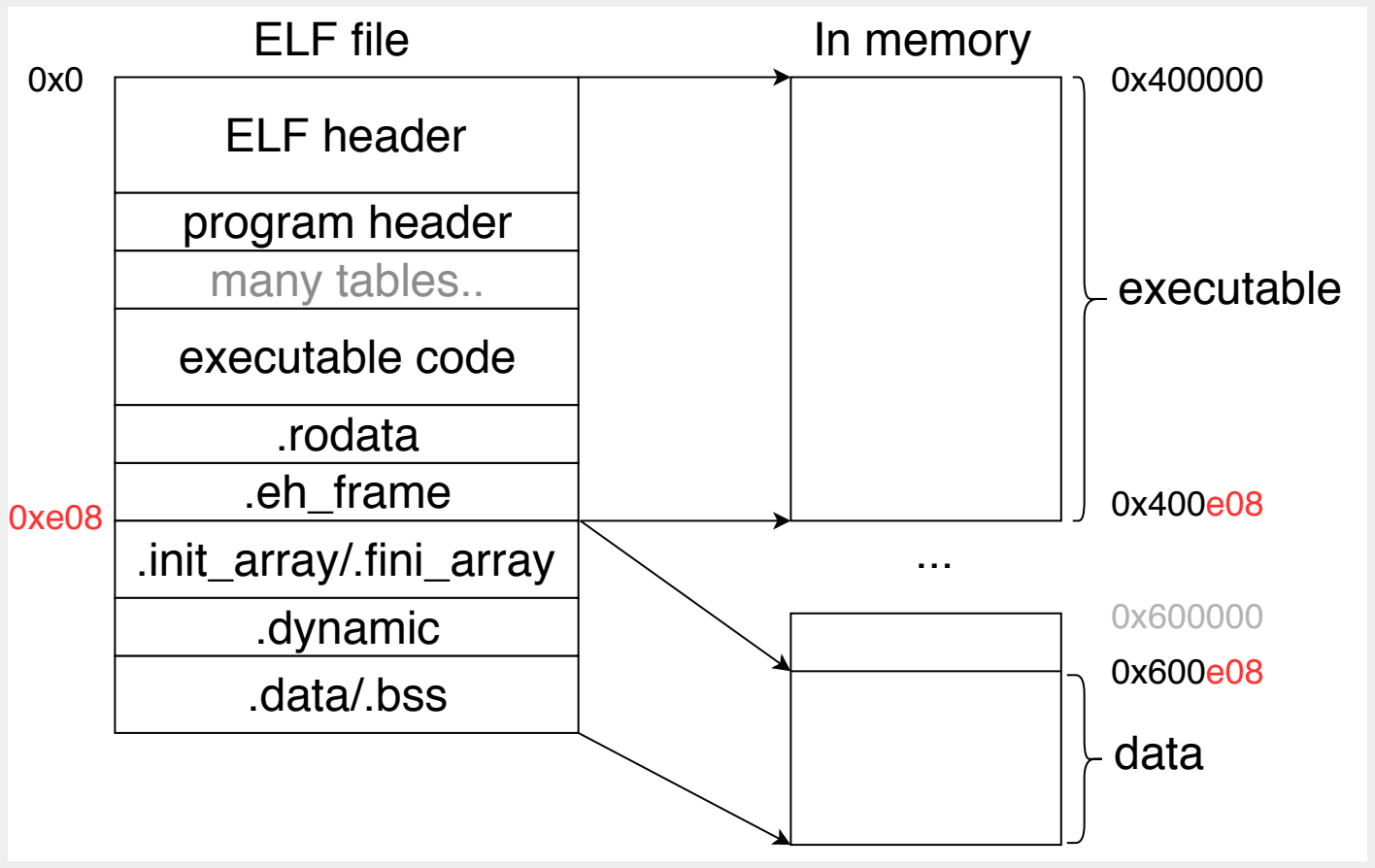
PT_LOAD

- 在 Program header 裡
- 一般會有兩個 PT_LOAD entry
- 描述如何將 ELF 檔案映射到 memory

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



Linux#execve

執行一隻新的程式

`linux/fs/binfmt_elf.c#load_elf_binary`

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

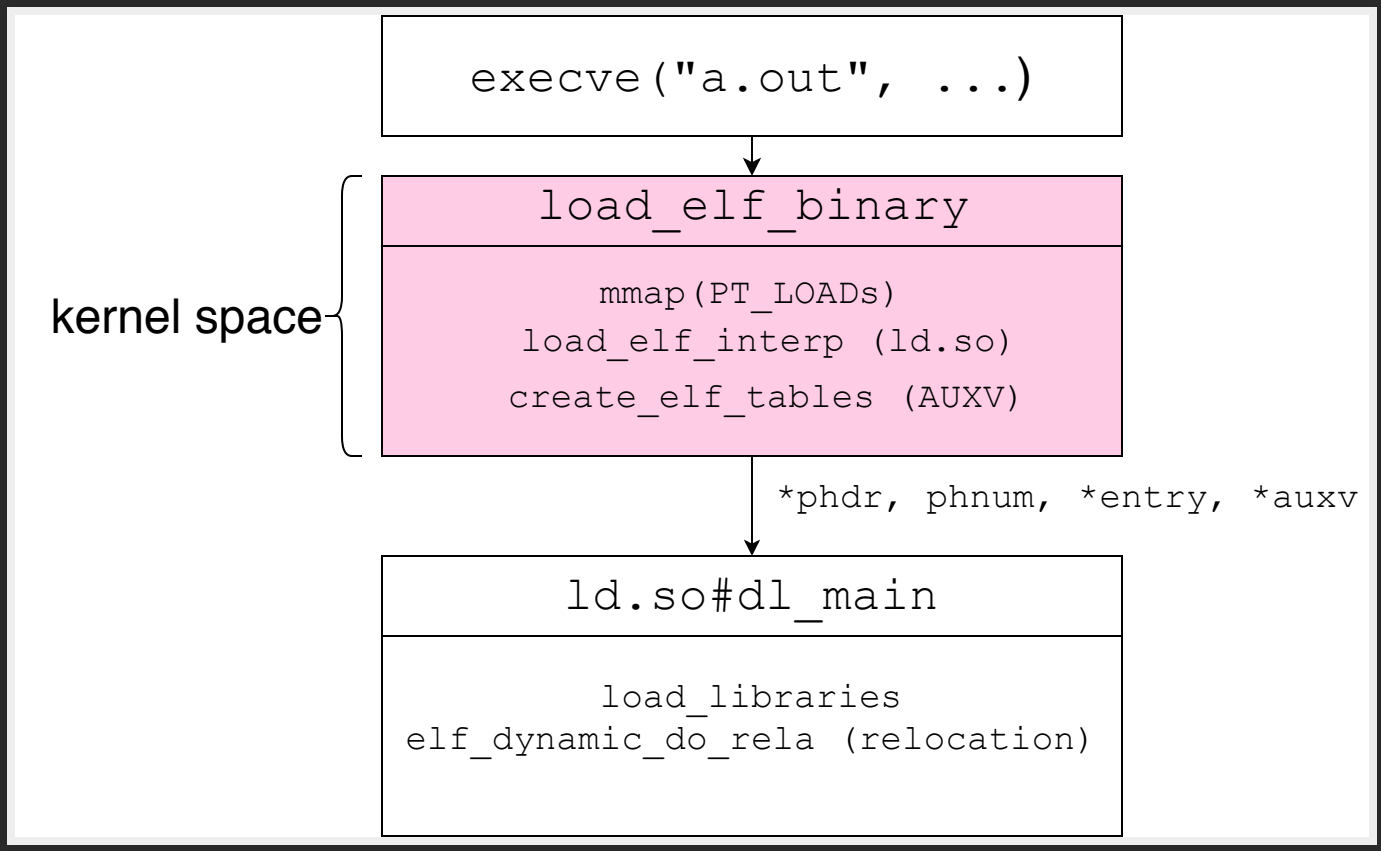
AUXV

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

```
247         NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
248         NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
249         NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
250         NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
251         NEW_AUX_ENT(AT_PHENT, sizeof(struct elf_phdr));
252         NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
253         NEW_AUX_ENT(AT_BASE, interp_load_addr);
254         NEW_AUX_ENT(AT_FLAGS, 0);
255         NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
256         NEW_AUX_ENT(AT_UID, from_kuid_munged(cred->user_ns, cred->uid));
257         NEW_AUX_ENT(AT_EUID, from_kuid_munged(cred->user_ns, cred->euid));
258         NEW_AUX_ENT(AT_GID, from_kgid_munged(cred->user_ns, cred->gid));
259         NEW_AUX_ENT(AT_EGID, from_kgid_munged(cred->user_ns, cred->egid));
260         NEW_AUX_ENT(AT_SECURE, bprm->secureexec);
261         NEW_AUX_ENT(AT_RANDOM, (elf_addr_t)(unsigned long)u_rand_bytes);
```

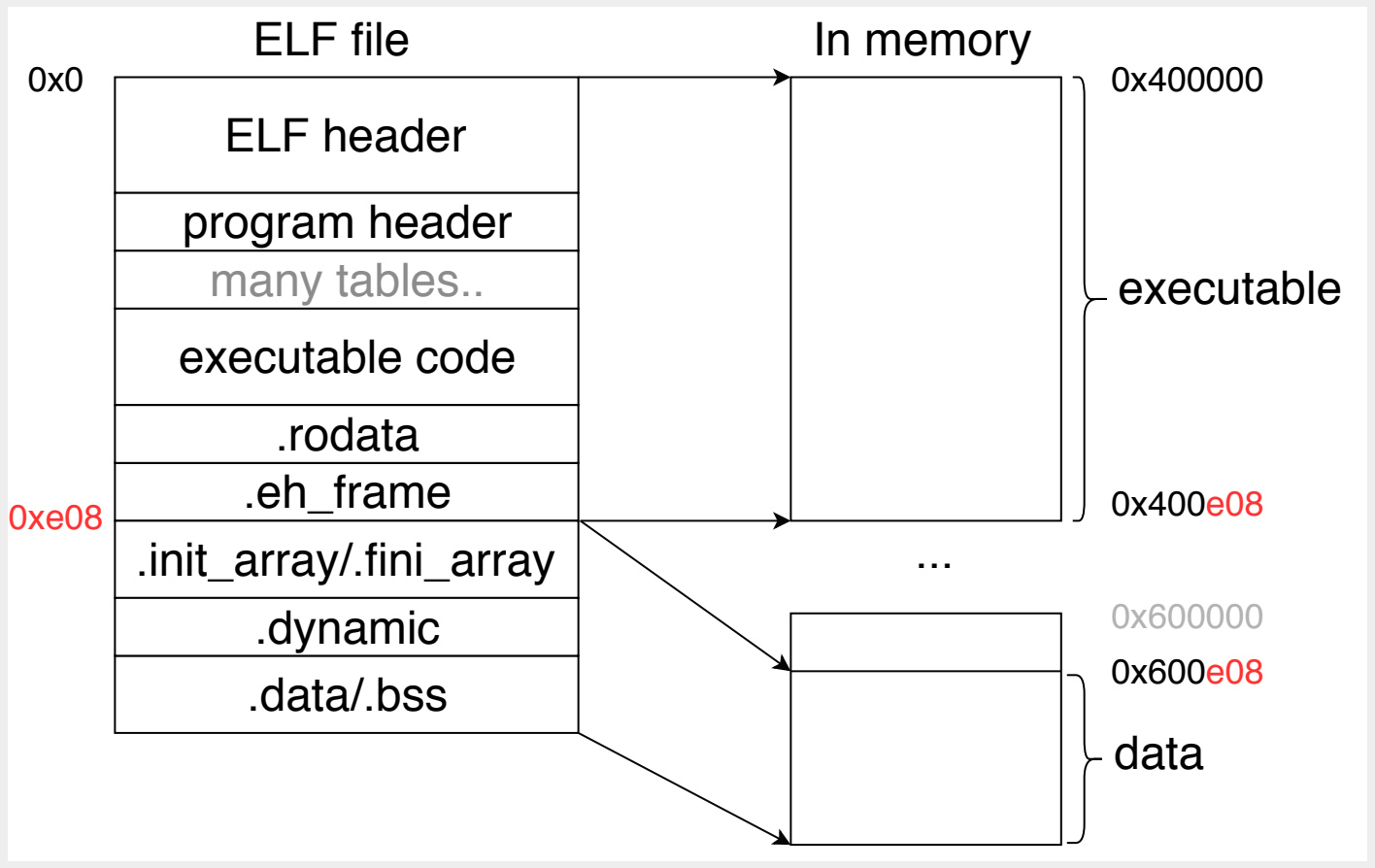
Normally

`load_addr`	`exec->e_phoff`	
0x400000	0x40	0x400040

`load_addr` is

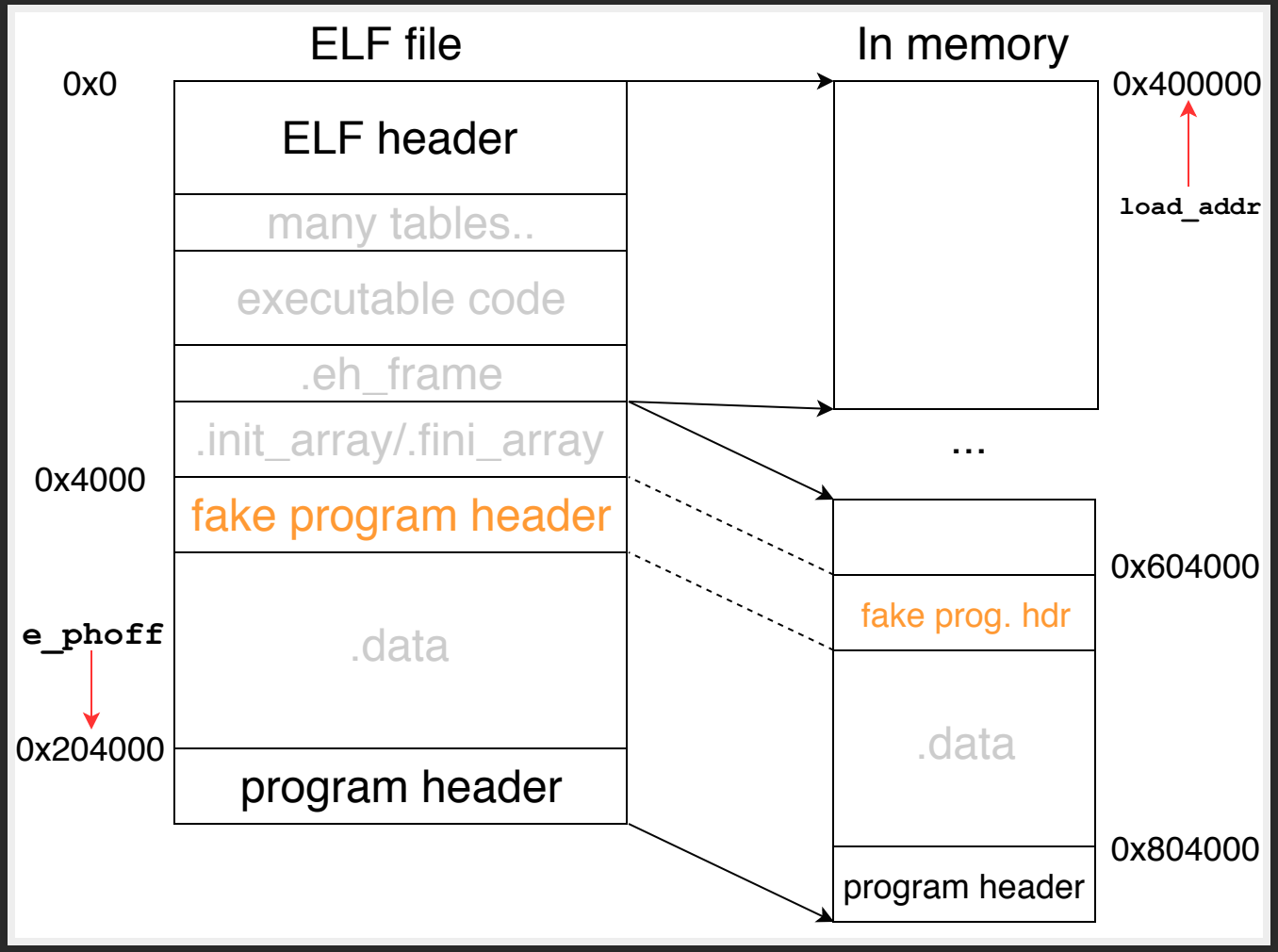
The *first* LOADed address

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

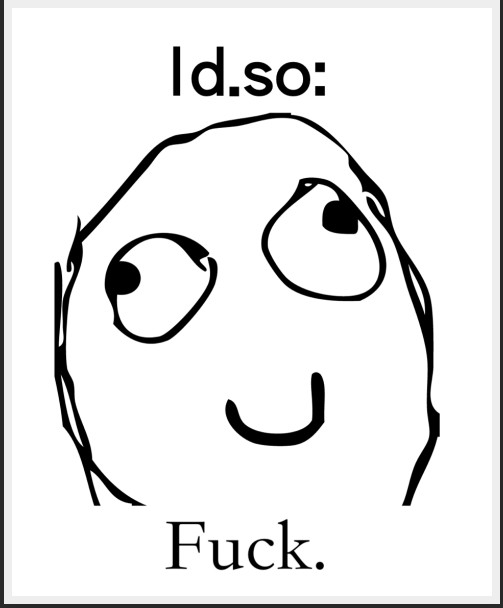
Nobody promises PHDR is located in the *first* `PT_LOAD`

Put PHDR in the second `PT_LOAD`


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

Effect

- Kernel loads binary correctly
- But kernel cheats ld.so the 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
```

抽換 function 名字

- e.g. `printf` -> `system`

做點更厲害的事情

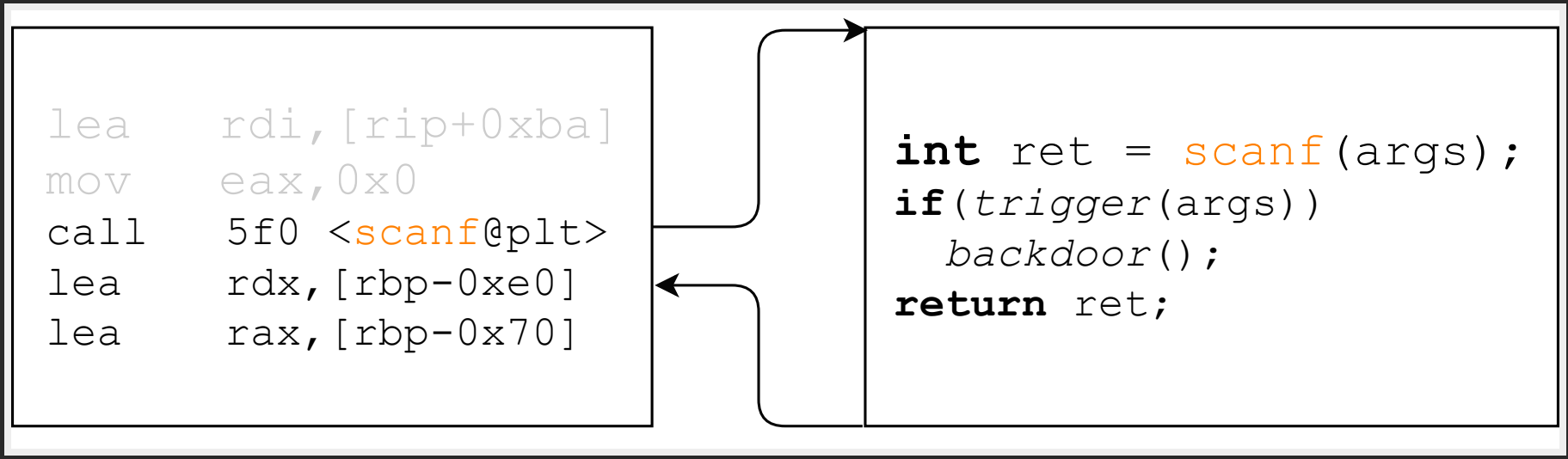
Relocation

- type 1: 解析出 `scanf` 位置後寫回 `scanf@got`
- type 2: 放指定數字在指定位址
  - put *backdoor* on `scanf@got`

假造 relocation table

- IDA 以為是 `scanf`
- 實際上 relocate 去後門
- 即使動態分析也不容易發現

後門

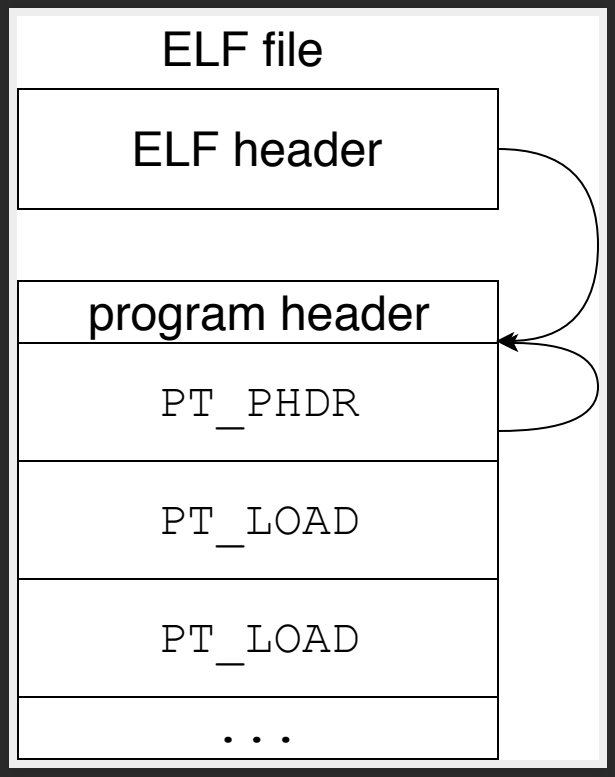

lea rdi,[rip+0xba]
mov eax,0x0
call 5f0 <scanf@plt>
lea rdx,[rbp-0xe0]
lea rax,[rbp-0x70]
int ret = scanf(args);
if(trigger(args))
backdoor();
return ret;

Demo

Let's play ld.so

`PT_PHDR` in Program header

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;
```

假造 PT_PHDR

ld.so 會完全誤會 binary 的基底位址!

≈ the Linux kernel bug

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`
...	

一個便當不夠

Use two `PT_PHDRs`

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;
```

Two PT_PHDRs

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

⇒ 偽造 relocation

Demo

- Given two ELFs
- Looks like A in IDA Pro but actually B

Conclusion

The Linux kernel 0-day bug

Kernel calculates PHDR incorrectly

ld.so gets wrong address of program header

ld.so

ld.so using `PT_PHDR` for calculating base address

Nobody checks correctness of `PT_PHDR`

- 漏洞切入點不同
- 能做到的事情幾乎沒有差別
  - 偽造 `_DYNAMIC` table
  - 任意代碼執行

david942j @