

Lời mở đầu:

Tài liệu viết ra với mục đích trình bày một cách đơn giản nhất về ELF file. Kiến thức trong tài liệu phần lớn là lấy từ một số tài liệu nước ngoài và một vài quan điểm của bản thân. Do đó có thể còn nhiều thiếu sót, nhầm lẫn. Nếu các bạn muốn góp ý cho mình về nội dung tài liệu, hãy liên hệ với mình theo thông tin phía cuối để mình hoàn thiện tài liệu tốt hơn.

Xin cảm ơn!

1. Cấu trúc file và header:

ELF được gồm ba loại chính:

- **Relocation file:** giữ các section có chứa code và data. Nó sẽ được liên kết với các object file để tạo thành tệp thực thi (executable file) hay tệp đối tượng chia sẻ (shared object file).
- **Executable file:** giữ thông tin đầy đủ để OS có thể tạo một process image của chương trình và thực thi chương trình.
- **Shared object file:** giữ code và data cho việc liên kết trong 2 trường hợp. Trường hợp đầu, link-editor tạo một Shared object file mới từ Relocation file và Shared object file. Trường hợp thứ hai xảy ra trong liên kết động. Từ Executable file và các file Shared object file, OS tạo thành một process image.

ELF header
Program header table
Section 1
Section 2
...
Section header table

1.1. ELF header:

Elf header nằm ở bắt đầu của file, được dùng để xác định loại file elf và kiến trúc cụ thể của file.

Cấu trúc của ELF header có cấu trúc như sau:

```

#define EI_NIDENT          16

typedef struct {
    unsigned char    e_ident[EI_NIDENT];
    Elf32_Half       e_type;
    Elf32_Half       e_machine;
    Elf32_Word       e_version;
    Elf32_Addr       e_entry;
    Elf32_Off        e_phoff;
    Elf32_Off        e_shoff;
    Elf32_Word       e_flags;
    Elf32_Half       e_ehsize;
    Elf32_Half       e_phentsize;
    Elf32_Half       e_phnum;
    Elf32_Half       e_shentsize;
    Elf32_Half       e_shnum;
    Elf32_Half       e_shstrndx;
} Elf32_Ehdr;

```

- e_ident** : gồm chữ ký để nhận biết ELF file và cung cấp thêm một số thông tin.
- e_type** : xác định loại file ELF.
- e_machine** : xác định kiến trúc đề nghị.
- e_version** : xác định phiên bản tệp.
- e_entry** : virtual address của câu lệnh đầu tiên (Entry point).
- e_phoff** : địa chỉ offset của Program header table (nếu không có thì giá trị bằng 0).
- e_shoff** : địa chỉ offset của Section header table (nếu không có thì giá trị bằng 0).
- e_flag** : giữ giá trị cò dành riêng cho bộ xử lý.
- e_ehsize** : kích thước của ELF header (byte).
- e_phentsize** : kích thước của một mục trong Program header table (byte).
- e_phnum** : số lượng mục trong Program header table *số lượng segment*.
- e_shentsize** : kích thước của một mục trong Section header table (byte).
- e_shnum** : số lượng Section.
- e_shstrndx** : chỉ mục của Section chứa tên các Section.

1.2. Program header table:

Program header table: (nếu xuất hiện) cho hệ thống cách tạo process image
 thông tin về các Segment. Relocation file không cần thiết phải có mục này.

Program header table là một mảng các cấu trúc, mỗi phần tử mô tả thông tin về segment hệ điều hành cần biết để chuẩn bị cho quá trình thực thi chương trình.

```
typedef struct {
    Elf32_Word    p_type;
    Elf32_Off     p_offset;
    Elf32_Addr    p_vaddr;
    Elf32_Addr    p_paddr;
    Elf32_Word    p_filesz;
    Elf32_Word    p_memsz;
    Elf32_Word    p_flags;
    Elf32_Word    p_align;
} Elf32_Phdr;
```

p_type : loại segment (NULL/ LOAD/ DYNAMIC ...)

p_offset : địa chỉ offset của section đầu tiên được gộp trong segment.

p_vaddr : virtual address của segment trong bộ nhớ.

p_paddr : physical address được đề nghị của segment, thành phần này có thể sai với executable file và shared object.

p_filesz : kích thước (byte) trong file image của segment.

p_memsz : kích thước (byte) trong memory image của segment.

p_flags : giá trị cờ của segment.

p_align : thể hiện segment được căn chỉnh trong memory. **p_offset** sẽ bằng **p_vaddr** chia lấy dư cho **p_align**. Giá trị 0 hoặc 1 có nghĩa là không được căn chỉnh lại.

1.3. Section header table:

Section header table: chứa thông tin về các Section của file.

Section header table là một mảng các cấu trúc mô tả cho từng section, có kiểu dữ liệu như sau:

```
typedef struct {
    Elf32_Word    sh_name;
    Elf32_Word    sh_type;
    Elf32_Word    sh_flags;
    Elf32_Addr    sh_addr;
    Elf32_Off     sh_offset;
    Elf32_Word    sh_size;
    Elf32_Word    sh_link;
    Elf32_Word    sh_info;
    Elf32_Word    sh_addralign;
    Elf32_Word    sh_entsize;
} Elf32_Shdr;
```

sh_name : tên của section, giá trị là chỉ mục trong String table.

sh_type : phân loại nội dung của Section.

sh_flags : mô tả các thuộc tính khác.

sh_addr : (nếu xuất hiện) là giá trị virtual address đề nghị của section.

sh_offset : địa chỉ offset của section.

sh_size : kích thước (size) của section.

sh_link : giá trị được đối chiếu trong bảng dưới.

sh_info : giá trị được đối chiếu trong bảng dưới.

sh_addralign : Một vài section có địa chỉ ràng buộc alignment.

sh_entsize : Section giữ bằng các thành phần có kích thước cố định (ví dụ: symbol table). Trường này là giá trị kích thước (byte) mỗi phần trong đó.

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

Diễn giải **sh_link** và **sh_info**

1.4. Section:

Section: đại diện cho đơn vị nhỏ nhất được xử lý trong ELF file. Nó dữ thông tin quan trọng như: lệnh, dữ liệu, bảng symbol, relocation information ...

Segment: là một bộ các Section, đơn vị nhỏ nhất cho việc ánh xạ vào bộ nhớ.

1.4.1. Các section đặc biệt:

Ngoài các section thiết yếu như: .text, .bss, .data, .debug ... thì còn các section phụ trợ cho quá trình tạo process image và thực thi file.

Section	Mô tả
.dynamic	Giữ thông tin cho quá trình liên kết động.
.dynstr	Giữ các chuỗi cho liên kết động, thường các chuỗi đại diện cho các tên kết hợp cùng bảng symbol
.dynsym	Chứa bảng symbol.
.got	Chứa global offset table.
.hash	Chứa bảng băm của các symbol.
.plt	Chứa bảng liên kết thủ tục.
.relname / .relaname	Chứa thông tin relocation. Ví dụ: .rel.text , rela.text
.ststrtab	Chứa tên các Section.

.strtab	Chứa các chuỗi, đại diện cho các tên của symbol.
.symtab	Chứa bảng symbol.

1.4.2. String table:

String table là một bảng với các chuỗi ký tự có ký tự kết thúc là NULL. ELF file sử dụng các chuỗi này để đại diện cho tên của symbol hay tên của section. Mỗi chuỗi đều có một chỉ mục trong String table. Ký tự đầu tiên – Chỉ mục 0 trong bảng là một ký tự NULL. Tương tự, chỉ mục cuối cùng cũng là NULL. Một chuỗi có chỉ mục là 0, có thể là không có tên hoặc tên là NULL tùy ngữ cảnh.

Ví dụ dưới đây là một String table 25byte và một chuỗi trong bảng:

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	\0	n	a	m	e	.	\0	v	a	r
10	i	a	b	l	e	\0	a	b	l	e
20	\0	\0	x	x	\0					

Figure 1-15: String Table Indexes

Index	String
0	<i>none</i>
1	name.
7	Variable
11	able
16	able
24	<i>null string</i>

1.4.3. Symbol table:

Symbol table giữ thông tin cần thiết cho định vị (locate) và định vị lại (relocate) các symbol được chương trình định nghĩa hay tham chiếu. Đây là một mảng các thành phần có cấu trúc như dưới đây, thành phần đầu tiên có giá trị bằng 0.

```
typedef struct {
    Elf32_Word      st_name;
    Elf32_Addr      st_value;
    Elf32_Word      st_size;
    unsigned char   st_info;
    unsigned char   st_other;
    Elf32_Half      st_shndx;
} Elf32_Sym;
```

st_name : chỉ mục trong symbol string table là tên của symbol.

st_value : giá trị liên quan tới symbol (có thể là 1 giá trị, địa chỉ ...).

st_size : một số symbol có kích thước liên quan.

st_info : Loại và thuộc tính ràng buộc của symbol.

st_other : thường có giá trị 0.

st_shndx : mọi symbol table entry đều được xác định liên quan đến một section, thành phần này giữ chỉ mục tới section header table index.

1.4.4. Relocation:

Relocation là quá trình kết nối giữa các symbol tham chiếu và được định nghĩa.

Ví dụ: Khi chương trình gọi một hàm. Lệnh gọi phải chuyển quyền quyền điều khiển đến địa chỉ đích phù hợp. Các relocatable file phải có thông tin mô tả cách chỉnh sửa nội dung các section của nó. Điều này cho phép executable file và shared object file giữ các thông tin đúng đắn cho program image.

```

typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;

typedef struct {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
    Elf32_Sword    r_addend;
} Elf32_Rela;

```

r_offset : vị trí thực thi quá trình relocation. Với Relocatable file đây là giá trị địa chỉ offset. Với Executable file và Shared object file đây là virtual address.
r_info : chứa thông tin cả chỉ mục trong symbol table và loại relocation.
r_addend : chỉ định hằng số được thêm vào để tính giá trị lưu trữ trong trường relocatable.

Relocation section tham khảo 2 section khác: symbol table và một section để chỉnh sửa. **sh_info** và **sh_link** chỉ định mối quan hệ này.

2. The memory management in Linux:

Logicial address (Virtual address) : được tạo bởi CPU, là địa chỉ được tham chiếu vào một vị trí trong memory.

Physical address : địa chỉ được nhìn bởi Memory Unit, là địa chỉ thực tế trong memory.

Memory management Unit : là một phần cứng ánh xạ từ logicial address tới physical address.

Chia trang (Paging) : không gian logicial address được chia làm các khối kích cỡ cố định gọi là trang. Trang trong memory được đặt trong vị trí memory khác. Không gian physical address được chia làm các khối như nhau gọi là frame. Chuyển dịch địa chỉ được thực hiện bằng một bảng trang bởi MMU.

Bộ nhớ ảo (virtual memory) : được tạo bởi CPU. Chuyển đổi địa chỉ bằng cách ánh xạ một virtual address tới một physical address.

Mỗi tiến trình có không gian virtual address là 4Gb. Linux dành 3Gb cho chế độ người dùng và 1Gb cho Kernel mode.

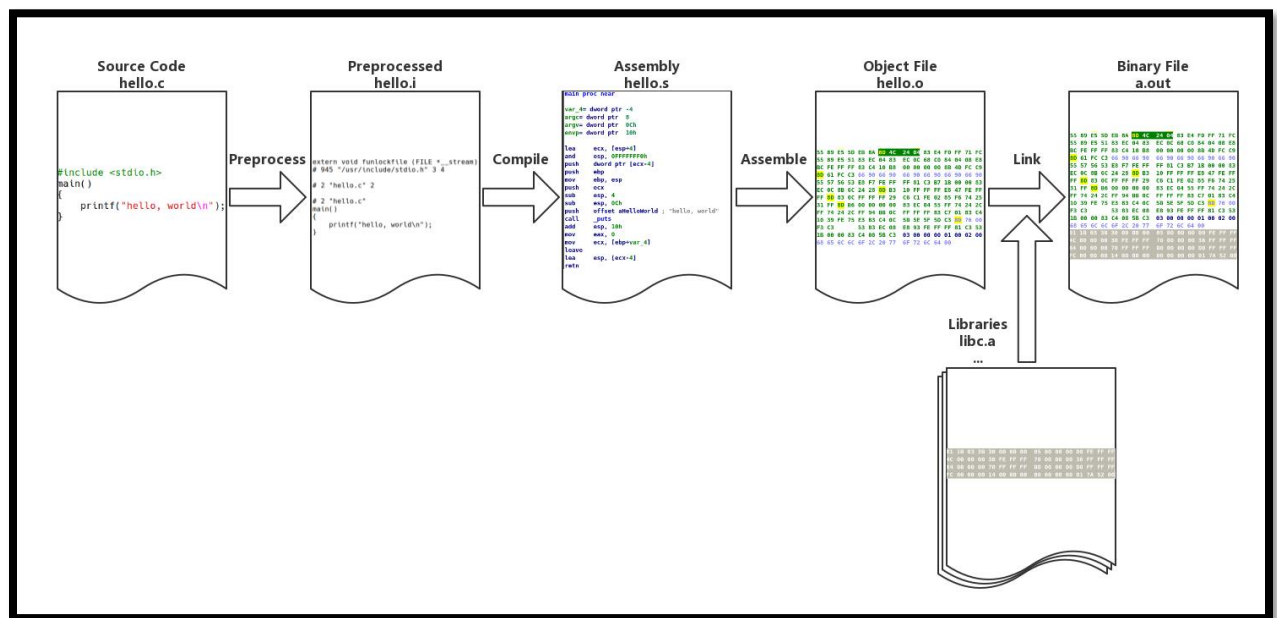
3. Program loading:

Để compile một chương trình “Hello world!” trong Linux ta thường dùng câu lệnh dưới đây:

```
# include <stdio.h> main() { printf ( "hello, world\n" ); }
```

```
$gcc hello.c ./a.out hello world
```

Quá trình trên có thể chia làm 4 bước: Preprocessing, Compilation, Assembly, Linking.



Vì ELF file phải tham gia vào cả hai quá trình : Linking chương trình và thực thi chương trình. Để thuận tiện, file cho phép hai cách đọc file riêng lẻ song song về nội dung file.

Object File Format

Linking View

ELF header
Program header table <i>optional</i>
Section 1
...
Section <i>n</i>
...
...
Section header table

Execution View

ELF header
Program header table
Segment 1
Segment 2
...
Section header table <i>optional</i>

*Từ Section header table và Program header table, ta có thể sắp xếp các Section vào Segment tương ứng *

```

root@kali: ~/Desktop/IDA
File Edit View Search Terminal Help
root@kali:~/Desktop/IDA# readelf -l crackme

Elf file type is EXEC (Executable file)
Entry point 0x80484a0
There are 8 program headers, starting at offset 52

Program Headers:
  Type           Offset             VirtAddr           PhysAddr          FileSiz MemSiz  Flg Align
  PHDR           0x000034          0x08048034         0x08048034        0x00100 0x00100 R E 0x4
  INTERP         0x000134          0x08048134         0x08048134        0x00013 0x00013 R   0x1
    [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000          0x08048000         0x08048000        0x00a3c 0x00a3c R E 0x1000
  LOAD           0x000f0c          0x08049f0c         0x08049f0c        0x00124 0x00130 RW 0x1000
  DYNAMIC         0x000f20          0x08049f20         0x08049f20        0x000d0 0x000d0 RW 0x4
  NOTE           0x000148          0x08048148         0x08048148        0x00020 0x00020 R   0x4
  GNU_STACK      0x000000          0x00000000         0x00000000        0x00000 0x00000 RW 0x4
  GNU_RELRO      0x000f0c          0x08049f0c         0x08049f0c        0x000f4 0x000f4 R   0x1

Section to Segment mapping:
Segment Sections...
 00
 01      .interp
 02      .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu
u.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
 03      .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
 04      .dynamic
 05      .note.ABI-tag
 06
 07      .ctors .dtors .jcr .dynamic .got
root@kali:~/Desktop/IDA#

```

Base address : Executable file và Shared object file có một giá trị base address, đây là địa chỉ virtual address nhỏ nhất của file trong memory image. Để tính giá trị này, ta lấy p_vaddr của segment loại LOAD.

Hệ thống sẽ sao chép các segment của file tới các segment của bộ nhớ một cách hợp lý.

Một tiến trình không yêu cầu một trang vật lý trừ khi nó tham chiếu tới một trang logic trong khi thực thi, tiến trình thường để lại nhiều trang không được tham chiếu (unreferenced). Do giảm sự trì hoãn từ việc đọc vật lý, nên hiệu suất hệ thống gia tăng. Để đạt được điều này trong thực tế, Executable file và Shared object file phải có image segment với file offset và virtual address là đồng dư, với modulo là kích thước một trang. Giá trị **p_align** trong của mỗi Program header phải là bội của giá trị modulo trên. (xem lại giá trị **p_align** trong Section header table).

Ví dụ sau về một tập tin thực thi có base address là 0x80000000 (2Gbytes).

Executable File

File Offset	File	Virtual Address
0	ELF header	
Program header table		
	Other information	
0x100	Text segment	0x8048100
	...	
	0x2be00 bytes	0x8073eff
0x2bf00	Data segment	0x8074f00
	...	
	0x4e00 bytes	0x8079cff
0x30d00	Other information	
	...	

Program Header Segments

Member	Text	Data
p_type	PT_LOAD	PT_LOAD
p_offset	0x100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	unspecified	unspecified
p_filesz	0x2be00	0x4e00
p_memsz	0x2be00	0x5e24
p_flags	PF_R+PF_X	PF_R+PF_W+PF_X
p_align	0x1000	0x1000

File offset và Virtual address trong ví dụ là đồng dư của modulo 4KB (0x1000), ta có 4 trang dữ liệu.

- Trang đầu chứa ELF header, Program header và thông tin khác.
- Trang thứ hai chứa mã lệnh.
- Trang thứ ba chứa dữ liệu.
- Trang cuối cùng là các thông tin khác không liên quan trong quá trình khởi chạy tiến trình.

Về mặt logic, hệ thống thực thi quyền truy cập bộ nhớ như mỗi segment là hoàn chỉnh và riêng biệt. Địa chỉ của segment được điều chỉnh để mỗi trang logic trong

không gian địa chỉ có một bộ quyền truy cập đơn lẻ. Trong ví dụ trên, phần tệp giữ phần cuối của Text và đầu của Data được ánh xạ hai lần: Một lần tại virtual address cho Text segment, lần nữa tại virtual address cho Data segment.

Phần cuối của Data segment yêu cầu được xử lý đặc biệt cho dữ liệu không được khởi tạo, hệ thống xác định bắt đầu với giá trị 0. Do đó nếu trang Data cuối cùng bao gồm các thông tin không có trong trang logic, các dữ liệu không liên quan bị đặt giá trị bằng 0, không phải là nội dung khác xác định.

Process Image Segments		
Virtual Address	Contents	Segment
0x8048000	<i>Header padding</i> 0x100 bytes	Text
0x8048100	Text segment ...	
	0x2be00 bytes	
0x8073f00	<i>Data padding</i> 0x100 bytes	
0x8074000	<i>Text padding</i> 0xf00 bytes	Data
0x8074f00	Data segment ...	
	0x4e00 bytes	
0x8079d00	Uninitialized data 0x1024 zero bytes	
0x807ad24	<i>Page padding</i> 0x2dc zero bytes	

Một khía cạnh trong nạp segment khác biệt giữa Executable file và Shared object file. Executable file thường chứa mã tuyệt đối. Để cho quá trình thực thi là chính xác, segment phải được đặt trong virtual address được chỉ định trong Executable file. Do đó, hệ thống sử dụng giá trị **p_vaddr** không đổi giống như virtual address. Mặt khác, segment của Shared object file thường chứa đoạn mã có vị trí độc lập. Virtual address của segment thay đổi với từng tiến trình mà không làm mất hiệu lực thực thi. Tuy nhiên, hệ thống chọn virtual address cho tiến trình riêng lẻ, nó cũng

duy trì một relative position. Do mã độc lập vị trí sử dụng một địa chỉ tuyệt đối giữa segment, sự chênh lệch giữa virtual address trong bộ nhớ phải trùng sự chênh lệch giữa virtual address được xác định trong file. Bảng dưới trình bày virtual address của Shared object file trong nhiều tiến trình, minh họa vị trí tương đối liên tục và base address:

Example Shared Object Segment Addresses			
Source	Text	Data	Base Address
File	0x200	0x2a400	0x0
Process 1	0x80000200	0x8002a400	0x80000000
Process 2	0x80081200	0x800ab400	0x80081000
Process 3	0x900c0200	0x900ea400	0x900c0000
Process 4	0x900c6200	0x900f0400	0x900c6000

4. Liên kết động (Dynamic linking / Strip symbol):

4.1. Dynamic Linker:

Khi dựng một tệp thực thi dùng liên kết động, link editor thêm một thành phần Program header loại PT_INTERP vào tệp thực thi, nói với hệ thống rằng gọi dynamic linker như thông dịch viên chương trình.

Exec(BA_OS) hợp tác cùng Dynamic linker hợp tác để tạo thành process image, điều này cần qua các bước sau:

- Thêm segment của Executable file vào process image.
- Thêm segment của Shared Object file vào process image.
- Tiến hành relocation Executable file và Shared Object file tương ứng.
- Đóng bộ mô tả file đã được dùng để đọc Executable file, nếu nó được trao cho Dynamic linker.
- Chuyển quyền điều khiển cho chương trình, làm như thể chương trình nhận trực tiếp quyền từ exec(BA_OS).

Link editor cũng xây dựng các dữ liệu khác nhau hỗ trợ quá trình liên kết động.

- Section .dynamic (SHT_DYNAMIC): cấu trúc được đặt tại đầu của section giữ địa chỉ tới thông tin khác của quá trình liên kết động.
- Section .hash (SHT_HASH): giữ symbol hash table.
- Section .got và .plt (SHT_PROGBITS) giữ hai bảng riêng biệt: Global offset table, Procedure linkage table.

Vì mọi chương trình tuân thủ theo chuẩn ABI đều import các dịch vụ hệ thống cơ bản từ các Shared object library, nên liên kết động tham gia khi thực thi chương trình theo ABI.

Shared object có thể đặt tại virtual memory address khác với địa chỉ trong Program header table. Dynamic linker di chuyển memory image, cập nhật địa chỉ tuyệt đối trước ứng dụng lấy lại quyền điều khiển. Tuy nhiên, vẫn có trường hợp, địa chỉ trong Program header table chỉ định trùng với địa chỉ tuyệt đối. Nếu process environment chứa một biến với tên LD_BIND_NOW cùng giá trị khác null thì Dynamic linker tiến hành relocation toàn bộ trước khi chuyển quyền điều khiển cho chương trình. Ngược lại, Dynamic linker được cho phép đánh giá mục Procedure linkage table một cách lười biếng, do đó tránh việc phân giải symbol và chi phí relocation các hàm không được gọi

4.2. Dynamic Section:

Nếu một object file tham gia quá trình liên kết động, Program header table sẽ có một thành phần loại PT_DYNAMIC. Segment này chứa section .dynamic. Một symbol đặc biệt, _DYNAMIC, chứa cấu trúc sau:

```
typedef struct {
    Elf32_Sword    d_tag;
    union {
        Elf32_Word    d_val;
        Elf32_Addr    d_ptr;
    } d_un;
} Elf32_Dyn;

extern Elf32_Dyn _DYNAMIC[];
```

Cho mỗi đối tượng, **d_tag** điều khiển cách diễn giải của **d_un**. (*)

d_val : đại diện một giá trị nguyên.

d_ptr: đại diện giá trị một địa chỉ. Như giới thiệu ở các phần trước, virtual address của file không giống với virtual address lúc thực thi. Khi diễn giải địa chỉ trong một cấu trúc động, dynamic tính toán địa chỉ tuyệt đối dựa vào giá trị gốc của file và base address. Để nhất quán, file không chứa các mục relocation để địa chỉ đúng trong cấu trúc động.

4.3. Global Offset Table:

Mã độc lập vị trí nói chung không thể chứa địa chỉ tuyệt đối. Global offset table giữ địa chỉ tuyệt đối trong các dữ liệu riêng, làm cho các địa chỉ khả dụng mà không ảnh hưởng đến vị trí độc lập và khả năng chia sẻ của chương trình. Một chương trình gọi Global offset table bằng việc sử dụng địa chỉ độc lập và lấy ra giá trị tuyệt đối. Do đó chuyển hướng địa chỉ độc lập đến địa chỉ tương đối. Ban đầu, Global offset table giữ các thông tin theo yêu cầu của các mục relocation. Sau khi hệ thống tạo các segment bộ nhớ cho các object file có thể nạp, dynamic linker xử lý các mục relocation, một số chúng nó loại R_386_GLOB_DAT gọi đến Global offset table. Dynamic linker xác định các symbol liên quan, tính giá trị tuyệt đối của chúng, đặt các mục trong bảng bộ nhớ với giá trị thích hợp. Tuy rằng, địa chỉ tuyệt đối là không biết khi dynamic linker dựng một object file, dynamic linker biết toàn bộ địa chỉ segment và có thể tính toán địa chỉ tuyệt đối của các symbol.

Nếu chương trình yêu cầu truy cập trực tiếp đến địa chỉ tuyệt đối của symbol, symbol sẽ có một mục trong Global offset table. Bởi vì, Executable file và Shared object file có global offset table riêng biệt, địa chỉ của một symbol có thể xuất hiện trong nhiều bảng. Dynamic linker xử lý tất cả việc sắp đặt global offset table trước khi đưa quyền truy cập cho câu lệnh trong process image, do đó đảm bảo địa chỉ tuyệt đối là khả dụng trong quá trình thực thi.

Mục 0 trong bảng giá trị dành cho việc giữ địa chỉ của cấu trúc động, gọi là symbol _DYNAMIC. Điều này cho phép tìm cấu trúc động của nó mặc dù chưa tiến hành xử lý các mục relocation. Điều này đặc biệt quan trọng với Dynamic linker, nó tự khởi tạo chính nó mà không dựa vào các chương trình khác để relocation process image của nó. Trong kiến trúc Intel 32 bit, mục 1 và 2 cũng được để dành.

Hệ thống phải chọn các địa chỉ segment khác nhau cho cùng một object file trong các chương trình khác nhau. Nó cũng phải chọn các địa chỉ thư viện khác nhau cho việc thực thi khác nhau của cùng chương trình giống nhau. Tuy nhiên, memory segment không bị thay đổi trong mỗi lần process image được tạo. Các memory segment được đặt trong virtual address cố định.

4.4. Procedure Linkage Table:

Nhiều như Global offset table chuyển địa chỉ độc lập thành các vị trí tuyệt đối. Procedure linkage table chuyển các lời gọi hàm tới vị trí tuyệt đối. Link editor không thể thực hiện chuyển (giống như lời gọi hàm) một Executable hoặc Shared object tới vị trí khác. Hậu quả là, link editor sắp xếp để có thể chuyển hướng điều khiển tới một mục trong Procedure linkage table. Trong kiến trúc hệ thống V(5), procedure linkage table được đặt trong text nhưng chúng sử dụng địa chỉ trong Global offset table riêng tư. Dynamic linker xác định đích của địa chỉ tuyệt đối và chỉnh sửa memory image phù hợp của Global offset table.

Dynamic linker do đó có thể chuyển hướng mà không ảnh hưởng đến vị trí độc lập và khả năng chia sẻ của chương trình. Executable file và Shared object file có Procedure linkage table riêng biệt.

Absolute Procedure Linkage Table

```
.PLT0:pushl got_plus_4
      jmp    *got_plus_8
      nop; nop
      nop; nop
.PLT1: jmp    *name1_in_GOT
      pushl  $offset@PC
.PLT2: jmp    *name2_in_GOT
      push   $offset
      jmp    .PLT0@PC
      ...
```

Position-Independent Procedure Linkage Table

```
.PLT0:pushl 4(%ebx)
      jmp    *8(%ebx)
      nop; nop
      nop; nop
.PLT1: jmp    *name1@GOT(%ebx)
      pushl  $offset
      jmp    .PLT0@PC
.PLT2: jmp    *name2@GOT(%ebx)
      pushl  $offset
      jmp    .PLT0@PC
      ...
```

Dynamic linker và chương trình hợp tác để giải quyết các lời gọi hàm qua Procedure linkage table và Global offset table:

- Khi lần đầu tạo memory image của chương trình, dynamic linker đặt chỉ mục thứ hai và ba trong global offset table là giá trị xác định.

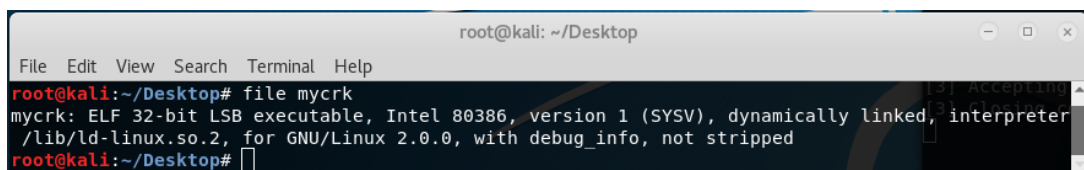
- Nếu procedure linkage table có vị trí độc lập, địa chỉ của Global offset table phải đặt trong %ebx. Mỗi shared object file trong process image có Procedure linkage table của riêng nó, điều khiển chuyển dịch một chỉ mục Procedure linkage table chỉ từ cùng object file. Kết quả là, hàm gọi chịu trách nhiệm thiết lập Global offset table dựa vào thanh ghi trước khi gọi chỉ mục Procedure linkage table.
- Để minh họa, giả sử chương trình gọi name1, chuyển điều khiển đến nhãn .PLT1.
- Câu lệnh đầu tiên nhảy đến địa chỉ chỉ mục cho name1 trong Global offset table. Ban đầu, Global offset table giữ địa chỉ sau lệnh pushl, không phải là địa chỉ thực của name1.
- Kết quả là, chương trình đẩy một relocation offset vào stack. Relocation offset là một giá trị 32bit trong bảng relocation. Chỉ mục relocation được chỉ định có loại R_386_JMP_SLOT, và offset sẽ xác định chỉ mục Global offset table được sử dụng trong lệnh jump trước. Chỉ mục relocation cũng chứa một chỉ số symbol table, do đó nói với dynamic linker symbol nào được gọi.
- Sau khi đẩy relocation offset chương trình sẽ nhảy về .PLT0. Lệnh pushl đặt giá trị thứ hai của Global offset table trên stack, do đó cung cấp cho dynamic linker một từ để xác định thông tin. Chương trình sau đó nhảy đến địa chỉ thứ ba của Global offset table, tại đây sẽ chuyển điều khiển cho dynamic linker.
- Dynamic linker nhìn vào chỉ mục relocation được chỉ định, tìm giá trị symbol và lưu địa chỉ thật cho name1 trong Global offset table của nó và chuyển quyền điều khiển cho đích mong muốn.
- Thực thi tiếp theo của Procedure linkage table chuyển trực tiếp đến name1, không có gọi dynamic linker lần hai. Câu lệnh jump tại .PLT1 sẽ chuyển tới name1 thay vì câu lệnh pushl.

4.5. Phân tích ví dụ:

Target: mycrk

Link: <https://crackmes.one/crackme/5ab77f6633c5d40ad448cbfe>

Dùng lệnh “file” thần thánh để kiểm tra thông tin của file thực thi:



```

root@kali: ~/Desktop
File Edit View Search Terminal Help
root@kali:~/Desktop# file mycrk
mycrk: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux.so.2, for GNU/Linux 2.0.0, with debug_info, not stripped
root@kali:~/Desktop#

```

Ta có thể thấy đây là file thực thi ELF 32 bit kiến trúc chip Intel 80386. File dạng liên kết động và chưa được stripped. Phần sau, mình sẽ thử làm thêm một file liên kết tĩnh xem có sự khác biệt gì.

Hiện ELF header:

```
root@kali: ~/Desktop
File Edit View Search Terminal Help
root@kali:~/Desktop# readelf -h mycrk
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                                ELF32
  Data:                                      2's complement, little endian
  Version:                               1 (current)
  OS/ABI:                                UNIX - System V
  ABI Version:                           0
  Type:                                  EXEC (Executable file)
  Machine:                               Intel 80386
  Version:                               0x1
  Entry point address:                   0x8048300
  Start of program headers:              52 (bytes into file)
  Start of section headers:              6936 (bytes into file)
  Flags:                                  0x0
  Size of this header:                   52 (bytes)
  Size of program headers:               32 (bytes)
  Number of program headers:              7
  Size of section headers:               40 (bytes)
  Number of section headers:              33
  Section header string table index:     30
root@kali:~/Desktop#
```

Show Program header table:

```
root@kali: ~/Desktop
File Edit View Search Terminal Help
root@kali:~/Desktop# readelf -l mycrk
Elf file type is EXEC (Executable file)
Entry point 0x8048300
There are 7 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E  0x4
  INTERP         0x000114 0x08048114 0x08048114 0x00013 0x00013 R    0x1
    [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x00531 0x00531 R E  0x1000
  LOAD           0x000534 0x08049534 0x08049534 0x00108 0x0010c RW  0x1000
  DYNAMIC         0x000544 0x08049544 0x08049544 0x000c8 0x000c8 RW   0x4
  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R    0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RWE  0x4

Section to Segment mapping:
Segment Sections...
 00
 01      .interp
 02      .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_
r .rel.dyn .rel.plt .init .plt .text .fini .rodata
 03      .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
 04      .dynamic
 05      .note.ABI-tag
 06
```

Show Section header table:

```
root@kali: ~/Desktop
File Edit View Search Terminal Help
root@kali:~/Desktop# readelf -S mycrk
There are 33 section headers, starting at offset 0x1b18:

Section Headers:
[Nr] Name                Type              Addr             Off             Size            ES Flg Lk Inf Al
[ 0]                      NULL              00000000         000000         000000          00  0  0  0  0
[ 1] .interp                PROGBITS          08048114         000114         000013          00  A  0  0  1
[ 2] .note.ABI-tag          NOTE              08048128         000128         000020          00  A  0  0  4
[ 3] .hash                  HASH              08048148         000148         000030          04  A  4  0  4
[ 4] .dynsym                DYNSYM            08048178         000178         000070          10  A  5  1  4
[ 5] .dynstr                STRTAB            080481e8         0001e8         000066          00  A  0  0  1
[ 6] .gnu.version            VERSYM            0804824e         00024e         00000e          02  A  4  0  2
[ 7] .gnu.version_r          VERNEED           0804825c         00025c         000020          00  A  5  1  4
[ 8] .rel.dyn                REL               0804827c         00027c         000008          08  A  4  0  4
[ 9] .rel.plt                REL               08048284         000284         000018          08  A  4  11 4
[10] .init                  PROGBITS          0804829c         00029c         000017          00  AX 0  0  4
[11] .plt                   PROGBITS          080482b4         0002b4         000040          04  AX 0  0  4
[12] .text                  PROGBITS          08048300         000300         0001f0          00  AX 0  0 16
[13] .fini                  PROGBITS          080484f0         0004f0         00001b          00  AX 0  0  4
[14] .rodata                PROGBITS          0804850c         00050c         000025          00  A  0  0  4
[15] .data                  PROGBITS          08049534         000534         00000c          00  WA 0  0  4
[16] .eh_frame              PROGBITS          08049540         000540         000004          00  A  0  0  4
[17] .dynamic                DYNAMIC           08049544         000544         0000c8          08  WA 5  0  4
[18] .ctors                 PROGBITS          0804960c         00060c         000008          00  WA 0  0  4
[19] .dtors                 PROGBITS          08049614         000614         000008          00  WA 0  0  4
[20] .jcr                   PROGBITS          0804961c         00061c         000004          00  WA 0  0  4
[21] .got                   PROGBITS          08049620         000620         00001c          04  WA 0  0  4
[22] .bss                   NOBITS            0804963c         00063c         000004          00  WA 0  0  4
[23] .comment                PROGBITS          00000000         00063c         00007e          00  0  0  1
[24] .debug_aranges          PROGBITS          00000000         0006c0         000058          00  0  0  8
[25] .debug_pubnames         PROGBITS          00000000         000718         000025          00  0  0  1
[26] .debug_info             PROGBITS          00000000         00073d         00096e          00  0  0  1
[27] .debug_abbrev           PROGBITS          00000000         0010ab         000124          00  0  0  1
[28] .debug_line             PROGBITS          00000000         0011cf         0001ca          00  0  0  1
[29] .debug_str              PROGBITS          00000000         001399         00065f          01  MS 0  0  1
[30] .shstrtab               STRTAB            00000000         0019f8         00011e          00  0  0  1
[31] .symtab                 SYMTAB            00000000         002040         0006b0          10  32 82 4
[32] .strtab                 STRTAB            00000000         0026f0         000341          00  0  0  1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
L (link order), O (extra OS processing required), G (group), T (TLS),
C (compressed), x (unknown), o (OS specific), E (exclude),
p (processor specific)
root@kali:~/Desktop#
```

Ngoài ra, readelf còn hỗ trợ nhiều tính năng khác như dump section, hiển thị các thông tin khác của ELF file. Dùng “man readelf” hoặc “readelf –help” để tìm hiểu thêm.

Đem file bỏ vào IDA, bật tab Segment ta thấy như sau:

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
LOAD	08048000	0804829C	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.init	0804829C	080482B3	R	.	X	.	L	dword	0004	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	080482B3	080482B4	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.plt	080482B4	080482F4	R	.	X	.	L	dword	0005	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	080482F4	08048300	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.text	08048300	080484F0	R	.	X	.	L	para	0006	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.fini	080484F0	0804850B	R	.	X	.	L	dword	0007	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	0804850B	0804850C	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.rodata	0804850C	08048531	R	.	.	.	L	dword	0008	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.data	08049534	08049540	R	W	.	.	L	dword	0009	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.eh_frame	08049540	08049544	R	.	.	.	L	dword	000A	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	08049544	0804960C	R	W	.	.	L	mempage	0002	public	DATA	32	0000	0000	0009	0000	0000
.ctors	0804960C	08049614	R	W	.	.	L	dword	000B	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.dtors	08049614	0804961C	R	W	.	.	L	dword	000C	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.jcr	0804961C	08049620	R	W	.	.	L	dword	000D	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.got	08049620	0804963C	R	W	.	.	L	dword	000E	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.bss	0804963C	08049640	R	W	.	.	L	dword	000F	public	BSS	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.prgend	08049640	08049641	?	?	?	.	L	byte	0010	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...
extern	08049644	08049658	?	?	?	.	L	dword	0011	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...
abs	08049658	08049674	?	?	?	.	L	dword	0012	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...

Sau đó, ta đặt breakpoint tại địa chỉ 0x080483F1. Chạy file, khi dừng tại breakpoint ta mở cửa sổ Segments:

Name	Start	End	R	W	X	D	L	Align	Base	Type	Class	AD	es	ss	ds	fs	gs
LOAD	08048000	0804829C	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.init	0804829C	080482B3	R	.	X	.	L	dword	0004	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	080482B3	080482B4	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.plt	080482B4	080482F4	R	.	X	.	L	dword	0005	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	080482F4	08048300	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.text	08048300	080484F0	R	.	X	.	L	para	0006	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.fini	080484F0	0804850B	R	.	X	.	L	dword	0007	public	CODE	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	0804850B	0804850C	R	.	X	.	L	mempage	0001	public	CODE	32	0000	0000	0009	0000	0000
.rodata	0804850C	08048531	R	.	.	.	L	dword	0008	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
myrk	08048531	08049000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
myrk	08049000	08049534	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
.data	08049534	08049540	R	W	.	.	L	dword	0009	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.eh_frame	08049540	08049544	R	.	.	.	L	dword	000A	public	CONST	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
LOAD	08049544	0804960C	R	W	.	.	L	mempage	0002	public	DATA	32	0000	0000	0009	0000	0000
.ctors	0804960C	08049614	R	W	.	.	L	dword	000B	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.dtors	08049614	0804961C	R	W	.	.	L	dword	000C	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.jcr	0804961C	08049620	R	W	.	.	L	dword	000D	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.got	08049620	0804963C	R	W	.	.	L	dword	000E	public	DATA	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.bss	0804963C	08049640	R	W	.	.	L	dword	000F	public	BSS	32	FFFFFF...	FFFFFF...	0009	FFFFFF...	FFFFFF...
.prgend	08049640	08049641	?	?	?	.	L	byte	0010	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...
myrk	08049641	08049644	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
extern	08049644	08049658	?	?	?	.	L	dword	0011	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...
abs	08049658	08049674	?	?	?	.	L	dword	0012	public		32	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...	FFFFFF...
myrk	08049674	0804A000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
libc_2.27.so	F7D81000	F7DCA000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
libc_2.27.so	F7DCA000	F7F84000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
libc_2.27.so	F7F84000	F7F85000	?	?	?	D	.	byte	0000	public	DATA	32	0000	0000	0000	0000	0000
libc_2.27.so	F7F85000	F7F87000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
libc_2.27.so	F7F87000	F7F88000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
debug001	F7F88000	F7F8B000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
debug002	F7FAC000	F7FAE000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
[vvar]	F7FAE000	F7FB1000	R	.	.	D	.	byte	0000	public	CONST	32	0000	0000	0000	0000	0000
[vdso]	F7FB1000	F7FB3000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
ld_2.27.so	F7FB3000	F7FB4000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
ld_2.27.so	F7FB4000	F7FD9000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
ld_2.27.so	F7FDA000	F7FDB000	R	.	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
ld_2.27.so	F7FDB000	F7FDC000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000
[stack]	FFD8D000	FFDAE000	R	W	X	D	.	byte	0000	public	CODE	32	0000	0000	0000	0000	0000

Click vào, ta có thấy “libc_2.27.so” cũng là một file ELF. Tìm kiếm trường e_entry trong ELF header của file (các bạn có thể dùng file elf101.pdf như một bản đồ để dò vị trí này). Ta thấy, giá trị của nó là 0x00019B20 (có thể khác với mỗi máy khác nhau), điều này hoàn toàn đã được giải thích ở phần Program loading.

Reset lại IDA, nạp lại file. Tại lệnh 0x080483F1, click vào `_print` ta nhảy tới 0x800482E4.

```
.plt:080482E4      jmp     ds:printf_ptr
```

Nhãn này chính là một thành phần trong bảng Global offset table. Từ đây, ta có thể nhận thấy, sau quá trình liên kết động. Giá trị của `printf_ptr` sẽ được gán bằng giá trị ô nhớ của hàm `printf`.

Đặt breakpoint tại địa chỉ 0x080482E4, rồi trace từng lệnh các bạn sẽ thấy điều thú vị. Do trình độ reverse của mình có hạn, nên mình cũng chỉ biết được câu lệnh cuối cùng trong segment của target gọi đến địa chỉ hàm `printf` là 0x080482BA. Hàm `printf` nằm tại “`ld_2.27.so`” và địa chỉ ô 0x8049624 và 0x8049628 cũng được ghi từ lệnh của Shared object file này.

Tương tự, các bạn có thể phân tích hàm “`scanf`”.

Từ đây, ta có thể hiểu đơn giản của quá trình hệ điều hành thực thi file ELF: OS nạp Executable file vào địa chỉ được chỉ định trong file, sau đó nạp các Shared object file cần thiết và thực hiện quá trình Relocation. Với các thông tin được mô tả tại Executable file, chương trình thực hiện quá trình liên kết động. Quá trình liên kết động có thể nói đơn giản là OS tìm kiếm địa chỉ các hàm chương trình thực thi gọi trong các Shared object file và gán các địa chỉ này vào một bảng trong Executable file. Sau này, khi gọi hàm chương trình chỉ việc nhảy đến địa chỉ này.

Để hiểu sâu hơn về quá trình này, đọc thêm tài liệu `dsowhowto.pdf` đi kèm.

4.6. ELF file liên kết tĩnh:

Ở đây, mình sẽ lấy ví dụ trong quyển ‘Ctf all in one’:

在 1.5.1 节 C语言基础 中我们看到了从源代码到可执行文件的全过程，现在来看一个更复杂的例子。

```
#include<stdio.h>

int global_init_var = 10;
int global_uninit_var;

void func(int sum) {
    printf("%d\n", sum);
}

void main(void) {
    static int local_static_init_var = 20;
    static int local_static_uninit_var;

    int local_init_val = 30;
    int local_uninit_var;

    func(global_init_var + local_init_val +
        local_static_init_var );
}
```

然后分别执行下列命令生成三个文件：

```
gcc -m32 -c elfDemo.c -o elfDemo.o

gcc -m32 elfDemo.c -o elfDemo.out

gcc -m32 -static elfDemo.c -o elfDemo_static.out
```

đi theo hàm print trong func, ta có thể nhận thấy rằng đoạn code của printf là các section trong ELF file. Chạy file và so sánh giữa hai cửa sổ Program Segmentation, ta thấy không có khác biệt quá nhiều.

Do đó, thay vì phải thực hiện quá trình liên kết động, file liên kết tĩnh chỉ đơn giản là đưa tất cả các hàm của Shared object file cần dùng vào chính đoạn mã của chính nó. Tất nhiên, điểm bất lợi đầu tiên của dung lượng tăng cao và khi hệ thống có thay đổi cũng ảnh hưởng đến khả năng thực thi của file.

TỔNG KẾT:

Rất cảm ơn các bạn khi đã chịu khó đọc tài liệu mình viết đến đây. Do năng lực bản thân có hạn, mình không chắc đã truyền tải hoàn toàn kiến thức ở những tài liệu vào đây. Tập đính kèm là các tài liệu tham khảo, sau khi có được cái nhìn cơ bản nhất các bạn có tiếp tục đọc chúng. Nếu có đóng góp hay phản bác về vấn đề gì trong tài liệu, hãy phản hồi cho mình!

Tài liệu tham khảo:

- Executable and Linkable Format (ELF).
- ELF101 a Linux executable walkthrough – Ange Albertini.



Email: diepnh96@gmail.com

github : github.com/dn9guy3n/ELFfile4Vietnamese

Hà Nội, 24/09/2018