digital whisper
Technologic Papers

ELF - Executable Linkable Format

מאת dexr4de

הקדמה

במאמר זה נכיר לעומק את מבנה ה-ELF ובפרט עבור מערכות 64 ביט. נעבור על מושגים ומונחים שונים הקשורים לטעינת תהליכים.לבסוף, בתור דוגמה מובהקת נתאר את האופן בו נטען הקרנל של לינוקס (שהוא בעצמו קובץ ELF) בזמן העלייה של המחשב.

ELF - Executable Linkable Format, כפי שהשם מרמז הוא פורמט קובץ הרצה סטנדרטי שנבחר ב-1999 לפורמט הרשמי של מערכות Unix like. כשאנו אומרים "קובץ הרצה" הכוונה לא רק לקובץ שמכיל פקודות מכונה, מיפוי לכתובות זכרון, הצהרה על _start והניתן באופן מעשי להרצה, אלא גם, כפי שנראה בהמשך, לקבצי אוביקט, core dumps, shared libraries וכו'. ELF פורסם לראשונה במערכות System V ומשם תפס תאוצה ופרסום רב.

Elf format

נציג להלן את הפורמט באופן כללי ובהמשך נתעמק בכל חלק לחוד:

offset	0	1	2	3	4	5	6	7
0x0000	'0x7F'	'E'	'L'	'F'	EI_CLASS	EI_DATA	EI_VERSION	EI_OSABI
0x0008	EI_ABIVERSION	EI_PAD	0x0	0x0	0x0	0x0	0x0	0x0
0x0010	e_type		e_machine		e_version			
0x0018	e_entry							
0x0020	e_phoff							
0x0028	e_shoff							
0x0030	e_flags				e_ehsize		e_phentsize	
0x0038	e_phnum		e_shentsize		e_shnum		e_shstrndx	
0x0040	p_type				p_flags			

0x0048	**p_offset**	
0x0050	**p_vaddr**	
0x0058	**p_paddr**	
0x0060	**p_filesz**	
0x0068	**p_memsz**	
0x0070	**p_align**	
	More Elf64_Phdr's ...	
	Sections go here ...	
N + 0x00	sh_name	sh_type
N + 0x08	sh_flags	
N + 0x10	sh_addr	
N + 0x18	sh_offset	
N + 0x20	sh_size	
N + 0x28	sh_link	sh_info
N + 0x30	sh_addralign	
N + 0x38	sh_entsize	
	More Elf64_Shdr's	

כפי שניתן לראות, תחילת הקובץ מיוצגת על ידי ה-Elf header. תפקידו העיקרי הוא לשמש כ"תוכן העניינים" של הקובץ עצמו.

עוד ניתן להבחין ב-Program headers שמציינים את הסגמנטים (segments) שיימצאו בזמן הרצת הקובץ, וכן ב-Section table הנמצא בסוף, ומציין את המקטעים השונים בקובץ. לכאורה נראה כי יש דמיון רב בין segment ל-section, אך כפי שנראה בהמשך יש הבדל מהותי בין השניים. בינתיים נסתפק בכך שנגיד כי בזמן הרצה חלק מהsections מתאגדים ל-segment יחיד וחלקם אף נשמטים.

כשאנו אומרים סגנמט, אין הכוונה לאותו סגמנט שאנו רגילים לשמוע עליו, כזה שניתן לגשת אליו בעזרת סלקטורים וכו, אלא פשוט למרחב מסוים בזכרון בזמן ריצה.

וכמובן יש את גוף הקובץ, שבו אפשר לצפות שיהיו מקטעים הקשורים להרצה של הקובץ כמו text, data, bss. במאמר נכיר עוד כמה חדשים.

הערה: בהמשך מופיעות דוגמאות חיות שנועדו להמחיש את הנעשה בפועל. הדוגמאות כוללות הרצה של תכנית פשוטה ביותר שקומפלה עם shared library.

נשתמש ב-readelf כדי להדפיס חלקים רלוונטים להסבר:

file.c:

```
#include <stdio.h>

extern int myvar;
extern int get_magic();

int main(int argc, char **argv)
{
 printf("myvar = %d\n", myvar);
 printf("magic function returned %#04x\n", get_magic());

 return 0;
}

mylib.c:
int myvar = 123789;

int get_magic()
{
 return 0x1337;
}
```

תהליך הקומפילציה:

```
gcc -o mylib.so mylib.c -shared
gcc -o magic file.c ./mylib.so
```

בואו נצלול

חלק זה הוא היותר טכני שבמאמר ויש בו שימוש במבני נתונים. ניתן לראות אותם בפירוט בקוד מקור של הקרנל של לינוקס ב-include/upai/linux/elf.h. כמו כן אתר שאני מאוד ממליץ עליו כי הוא מכיל את כל ה-source code של הקרנל של לינוקס משלל גרסאות והוא מאוד נוח לגלישה:

http://lxr.free-electrons.com

לפני שנתחיל להגדיר את מבני הנתונים, על מנת להקל על קריאת הקוד נביא typedefs רלוונטיים:

```
/* 64-bit ELF base types */
typedef unsigned long long   Elf64_Addr;
typedef unsigned short       Elf64_Half;
typedef signed short         Elf64_Addr;
typedef unsigned long long   Elf64_Off;
typedef signed int           Elf64_Sword;
typedef unsigned int         Elf64_Word;
typedef unsigned long long   Elf64_Xword;
typedef signed long long     Elf64_Addr;
```

Elf Header

כפי שמשתמע עד כה, חלק זה אכן אחראי על הצגת המידע הכללי אודות הקובץ הרצה. הוא מוגדר כך:

```
typedef struct elf64_hdr {
 unsigned char e_ident[EI_NIDENT]; /* ELF "magic number" */
 Elf64_Half e_type;
 Elf64_Half e_machine;
 Elf64_Word e_version;
 Elf64_Addr e_entry;   /* Entry point virtual address */
 Elf64_Off e_phoff;    /* Program header table file offset */
 Elf64_Off e_shoff;    /* Section header table file offset */
 Elf64_Word e_flags;
 Elf64_Half e_ehsize;
 Elf64_Half e_phentsize;
 Elf64_Half e_phnum;
 Elf64_Half e_shentsize;
 Elf64_Half e_shnum;
 Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

נתמקד בשדות היותר רלוונטיים (הכוונה היא שאני מניח כי למשל שדות כמו e_machine, אמורים להיות מובנים מאליו...)

- **e_ident** - כמו שהשם מרמז זהו החלק העקרי שמזהה את הקובץ. זהו מערך בגודל 16 בתים שכאשר ה4- בתים הראשונים הם הידועים בתור ELF magic number שהוא: '0x7F', 'E', 'L', 'F'" הבתים האחרים הם מוגדרים להיות אפסים וחלקם מורים על האם הקובץ הוא למשל big/little endian וכו.

- **e_type** - סוג הקובץ. נשתמש בפקודה: readelf -h magic mylib.so ואכן נראה את ההבדל בין השניים:

```
File: magic
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
```

```
File: mylib.so
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
```

- **e_entry** - הכתובת הוירטואלית ממנה מתחיל תהליך טעינת התכנית/מידע
- **e_phoff** - מתחילת הקובץ בו מתחילים ה-offset של ה-Program headers
- **e_shoff** - ה-offset מתחילת הקובץ שבו מתחיל ה-Section table.
- **e_ehsize** - גודל ה-Elf header, כרגע 64 בתים.
- **e_phentsize** - הגודל של Program header יחיד, כרגע 56 בתים.
- **e_phnum** - מספר ה-Program headers בקובץ.
- **e_shentsize** - גודל של Section header, בסטנדרט 64 בתים.
- **e_shnum** - מספר ה-Section headers בקובץ.
- **e_shstrndx** - האינקדס למקטע שמכיל את טבלת המחרוזות, String Table של הקובץ. כפי שנראה מפתחי פורמט ELF מצאו דרך יצירתית לשמור את שמות המקטעים בקובץ.

מפה נעשה קפיצה ל-Section Header Table. העניין הוא שאחרי שנהיה בקיאים בנושא המקטעים, נוכל להתפנות לתיאור הProgram headers שקשורים לטעינת הזכרון של התהליך.

Sections

אם נסתכל בפורמט בתחילת המאמר נראה כי ה-sections מהווים את בשר הקובץ ולא סתם כך. אם אתם שואלים, הייתי מגדיר מקטע בהקשר הזה כיחידת מידע בעלת משמעות מיוחדת. נעשה:

```
readelf -S magic
```

(נראה את המקטעים הרלוונטים ביותר למאמר):

```
[ 0]                   NULL             0000000000000000  00000000
     0000000000000000  0000000000000000           0     0     0
[ 1] .interp           PROGBITS         0000000000400238  00000238
     000000000000001c  0000000000000000   A       0     0     1
[ 2] .note.ABI-tag     NOTE             0000000000400254  00000254
     0000000000000020  0000000000000000   A       0     0     4
[ 3] .note.gnu.build-i NOTE             0000000000400274  00000274
     0000000000000024  0000000000000000   A       0     0     4
```

```
[ 5] .dynsym           DYNSYM           00000000004002d8  000002d8
     0000000000000150  0000000000000018   A       6     1     8
[ 6] .dynstr           STRTAB           0000000000400428  00000428
     00000000000000c8  0000000000000000   A       0     0     1
```

```
[ 9] .rela.dyn         RELA             0000000000400530  00000530
     0000000000000030  0000000000000018   A       5     0     8
[10] .rela.plt         RELA             0000000000400560  00000560
     0000000000000060  0000000000000018  AI       5    12     8
[11] .init             PROGBITS         00000000004005c0  000005c0
     000000000000001a  0000000000000000  AX       0     0     4
[12] .plt              PROGBITS         00000000004005e0  000005e0
     0000000000000050  0000000000000010  AX       0     0    16
[13] .text             PROGBITS         0000000000400630  00000630
     00000000000001b2  0000000000000000  AX       0     0    16
```

```
[22] .got              PROGBITS         0000000000600ff8  00000ff8
     0000000000000008  0000000000000008  WA       0     0     8
[23] .got.plt          PROGBITS         0000000000601000  00001000
     0000000000000038  0000000000000008  WA       0     0     8
[24] .data             PROGBITS         0000000000601038  00001038
     0000000000000010  0000000000000000  WA       0     0     8
[25] .bss              NOBITS           0000000000601048  00001048
     0000000000000008  0000000000000000  WA       0     0     8
```

```
[27] .shstrtab         STRTAB           0000000000000000  00001075
     0000000000000108  0000000000000000           0     0     1
[28] .symtab           SYMTAB           0000000000000000  00001180
     0000000000000648  0000000000000018          29    45     8
[29] .strtab           STRTAB           0000000000000000  000017c8
     0000000000000248  0000000000000000           0     0     1
```

במבט קל אכן אפשר לזהות מקטעים די מוכרים כמו .text, bss, data. אני מניח שאתם מכירים אותם. בהמשך המאמר ניגע גם באותם כל אלו המסתוריים. כאמור כל כזה מיוצג על ידי Section header, שמופיעים בסוף הקובץ ומוגדרים כך:

```
typedef struct elf64_shdr {
 Elf64_Word sh_name;   /* Section name, index in string tbl */
 Elf64_Word sh_type;   /* Type of section */
 Elf64_Xword sh_flags;        /* Miscellaneous section attributes */
 Elf64_Addr sh_addr;   /* Section virtual addr at execution */
 Elf64_Off sh_offset;         /* Section file offset */
 Elf64_Xword sh_size;         /* Size of section in bytes */
 Elf64_Word sh_link;   /* Index of another section */
 Elf64_Word sh_info;   /* Additional section information */
 Elf64_Xword sh_addralign; /* Section alignment */
```

```
 Elf64_Xword sh_entsize; /* Entry size if section holds table */
} Elf64_Shdr;
```

תיאור השדות הרלוונטיים:

- **sh_name** - לפני שנסביר את משמעות שדה זה, נכיר את ה-String Table, טבלת המחרוזות. בעצם זהו מקטע שלם (בדיוק כמו text, data) בעל מבנה דומה לזה שמופיע למטה:

.	*'\0'*	*p*	*r*	*e*	*t*	*n*	*i*	*.*	*'\0'*
t	*-*	*I*	*B*	*A*	*.*	*e*	*t*	*o*	*n*
g	*.*	*e*	*t*	*o*	*n*	*.*	*'\0'*	*g*	*a*

(Table continues...)

היא מתחילה בתו 0, ומכילה מחרוזת שזוהי בתורה גם מסתיימת ב-0. תפקידה בעיקרון לספק מידע רלוונטי שניתן יהיה להדפסה. כפי שניתן לראות בדוגמה, מופיעים שמות המקטעים, ושם טבלת המחרוזות הוא shstrtab. חשוב לציין שיכולות להיות עוד String Tables, שמציינות לא את שמות המקטעים, אלא למשל שמות של Symbols כפי שנראה בהמשך.

ובחזרה ל-sh_name, תפקידו הוא לספק אינדקס לתוך אותה טבלת מחרוזות בה מופיע השם של אותו מקטע במקרה זה. לדוגמה ב-Elf64_Shdr השני שמתאר את interp. שעליו נדבר בהמשך (ה-Elf64_Shdr הראשון הוא תמיד NULL), ערכו הוא 1, ובמקרה של Elf64_Shdr השלישי, ערך ה-sh_name הוא 9.

- **sh_type** - סוג המקטע. נציין כמה אופיניים:
  - **NULL** - באופן default ה-Elf64_Shdr הראשון הוא תמיד ריק.
  - **PROGBITS** - משמעותו היא שהמקטע מכיל מידע שאמור להיות "מעובד" למשל פקודות מכונה.
  - **NOBITS** - מסמל כי המקטע לא מכיל מידע בקובץ, אבל הוא מוקצה בזכרון. הדוגמה הכי טובה שאני יכול לחשוב עליה הוא מקטע ה-bss.
  - **STRTAB** - אני מניח שזוהי טבלת המחרוזות, לא?
  - **REL/RELA** - נראה בהמשך.
  - **NOTE** - מטרתו של מקטע זה היא לספק מידע אודות פרטים שונים כמו Build ID. בעזרת readelf -n magic נראה:

```
Displaying notes found at file offset 0x00000274 with length 0x00000024
:
  Owner                Data size       Description
  GNU                  0x00000014      NT_GNU_BUILD_ID (unique build I
D bitstring)
    Build ID: 45e4eb5bd57c4208bab432529504cd0fcdeb677c
```

  - **SYMTAB** - נראה עוד רגע!

מקטעים נבחרים

בתת חלק זה נעסוק במקטעים ספציפיים ובעלי חשיבות רבה.

Symbol Table

בואו נעשה סיור מוחין קטן לפני שנביא את ההגדרה למושג זה. במצב תאורטי אי שם, יש לי תכנית שעלתה לזכרון, אוקי, מה הלאה? ככל הנראה התכנית תרצה לקרוא לפונקציות, משתנים ומקומות שונים בזכרון. אבל כיצד היא תדע על קיומם? איך התכנית במקרה שלנו תדע שיש כזה משתנה 'myvar' ולא תחשוב שזה משתנה לא מוגדר (כמובן באופן תאורטי...)?

ובכן, אני מנחש שעליתם על זה כבר, אז כן, התכנית צריכה איזשהי רשימת מכולת שתכיל אזכור לגבי כל אותם פונקציות, משתנים ודברים אחרים, רבים וטובים או בקיצור Symbols. לאותה רשימה קוראים Symbol Table. טבלה זו בעצם תופסת מקטע שלם שמוקדש רק לה, כלומר מקטע שהוא פשוט אוסף של Elf64_Sym. ראוי לציין שיש כמה סוגי טבלאות כאלו. נתאר את symtab. נעשה readelf -s magic ונראה את symtab.:

```
49: 00000000004007e4      0 FUNC    GLOBAL DEFAULT   14 _fini
50: 0000000000000000      0 FUNC    GLOBAL DEFAULT  UND printf@@GLIBC_2.2.5
51: 0000000000601048      4 OBJECT  GLOBAL DEFAULT   25 myvar
52: 0000000000000000      0 FUNC    GLOBAL DEFAULT  UND __libc_start_main@@GLIBC_
53: 0000000000601038      0 NOTYPE  GLOBAL DEFAULT   24 __data_start
```

```
60: 0000000000601048      0 NOTYPE  GLOBAL DEFAULT   25 __bss_start
61: 0000000000400726     72 FUNC    GLOBAL DEFAULT   13 main
62: 0000000000000000      0 NOTYPE  WEAK   DEFAULT  UND _Jv_RegisterClasses
63: 0000000000000000      0 FUNC    GLOBAL DEFAULT  UND get_magic
```

ה-Elf64_Sym מוגדר כך:

```
typedef struct elf64_sym {
 Elf64_Word st_name;   /* Symbol name, index in string tbl */
 unsigned char st_info; /* Type and binding attributes */
 unsigned char st_other; /* No defined meaning, 0 */
 Elf64_Half st_shndx;         /* Associated section index */
 Elf64_Addr st_value;         /* Value of the symbol */
 Elf64_Xword st_size;         /* Associated symbol size */
} Elf64_Sym;
```

תיאור השדות:

- **st_name** ה-offset לתוך המקטע של טבלת המחרוזות (שימו לב זוהי אינה אותה טבלה כמו של Section Table). למקטע זה קוראים בתור ברירת מחדל strtab. ואם נסתכל במקטעים המצויינים למעלה, מספרו הוא 29.
- **st_info** - נסתכל בהערה המצויינת. מה שאפשר להבין זה ששדה זה כביכול מורה על שני סוגי מידע נפרדים, Type וBinding. נשאלת השאלה איך ערך אחד יכול להורות על שני ערכים שונים?

התשובה לכך נעוצה בכך שמחלקים את אותו שדה, לשני חלקים נפרדים: 4 ביטים מציינים את הType וה-4 האחרים מציינים את ה-Bind.

אם נסתכל בדוקומנטציה נראה את ה-macros הבאים:

```
#define ELF64_ST_BIND(i)           ((i) >> 4)
#define ELF64_ST_TYPE(i)           ((i) & 0xF)
#define ELF64_ST_INFO(b, t)  (((b) << 4) + ((t) & 0xF))
```

ניקח בתור דוגמה את השני: הוא מקבל משתנה i (הכוונה info, יותר נכון st_info), מבצע את מה שמבצע ומתקבל מספר. אותו מספר מתאים לאחד מאלו:

```
/* Symbol types */
#define STT_NOTYPE           0 /* No type specified */
#define STT_OBJECT           1 /* Data object */
#define STT_FUNC             2 /* Function entry point */
#define STT_SECTION   3 /* Symbol is associated with a section */
...
```

אני מנחש שה-readelf יעשה איזשהו:

```
switch (ELF64_ST_TYPE(symbol_table[i]->st_info)) {
case STT_NOTYPE:… break;
case STT_OBJECT:… break;
...
}
```

וכך ידפיס את מה שהוא הדפיס למעלה.

מן הסתם הוא הדבר תקף ל-ELF64_ST_BIND. קל גם לראות ש-ELF64_ST_INFO מבצע את הפעולה ההפוכה על type ו-binding.

בתמונה נראה כמצופה שmyvar הוא OBJECT וget_magic הוא FUNC.

st_other - פחות או יותר מתקבל מאותו עקרון כמו השדה הקודם רק פה שmacro הוא:

```
#define ELF64_ST_VISIBILITY(o) ((o) & 0x3)
```

לפי השם של ה-macro ופלט של readelf די ברור מה תפקידו של שדה זה.

Relocations - רלוקציות

גם פה נתחיל מדוגמה די פשוטה. לפני שהתכנית שלנו, magic עולה לזכרון, בניגוד למשתנים ופונקציות פנימיות שמוגדרים ב-file.c, המשתנים והפונקציות שנמצאים ב-mylib ובספריות חיצוניות אינם נמצאים נמצאים בו. מה שהולך בפועל הוא שהסביבה אמורה איכשהו לקשר בין מה שנמצא ב-mylib ל-magic בזמן טעינה, כדי שבסוף אותם משתנים ופונקציות ימצאו במרחב הזכרון ולכן אמור להיות אזכור שלהם ב-magic. באופן כללי אפשר להגיד רלוקציה היא קישור בין האזכור הסימבולי של דבר כלשהו לבין הגדרתו מחדש. ב-Elf הם מוגדרים כך:

```
typedef struct elf64_rel {
Elf64_Addr r_offset; /* Location at which to apply the action */
Elf64_Xword r_info; /* index and type of relocation */
} Elf64_Rel;
```

```
typedef struct elf64_rela {
Elf64_Addr r_offset; /* Location at which to apply the action */
Elf64_Xword r_info; /* index and type of relocation */
Elf64_Sxword r_addend; /* Constant addend used to compute value */
} Elf64_Rela;
```

נעשה readelf -r magic:

```
Relocation section '.rela.dyn' at offset 0x530 contains 2 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000600ff8  000400000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0
000000601048  000d00000005 R_X86_64_COPY     0000000000601048 myvar + 0

Relocation section '.rela.plt' at offset 0x560 contains 4 entries:
  Offset          Info           Type           Sym. Value    Sym. Name + Addend
000000601018  000200000007 R_X86_64_JUMP_SLO 0000000000000000 printf + 0
000000601020  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
000000601028  000400000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
000000601030  000600000007 R_X86_64_JUMP_SLO 0000000000000000 get_magic + 0
```

כצפוי myvar, get_magic ו-printf וכו שאינם מוגדרים ישירות ב-magic, אלא בספריות חיצוניות, ועל כן גם הם משתתפים בתהליך הרלוקציה.

תיאור השדות:

- **r_offset** - באופן כללי ניתן להסתפק שזה הכתובת בה תתבצע הרלוקציה. בתור דוגמה פשוטה ביותר, אם נעשה ב-gdb בזמן ריצה x/d 0x601048 יודפס ערכו של myvar שהוא כאמור 123789.
- **r_info** - בדומה ל-sh_info, גם שדה זה מורכב משני שדות. האחד מהם הוא type שמורה על סוג הרלוקציה, במקרה של myvar זה R_X86_64_COPY, כלומר בצורה "פשוטה" ערכו של משתנה זה מועתק. השני הוא האינדקס לSymbol Table, שמכילה מידע על אותו Symbol, ובין היתר את ה-offset ל-String Table ומשם אפשר להדפיס את השם של הרלוקציה. ההפרדה נעשית כך:

```
#define ELF64_R_SYM(i)              ((i) >> 32)
#define ELF64_R_TYPE(i)             ((i) & 0xffffffff)
```

- **r_addend** - נמצא רק בElf_Rela ותפקידו בעיקר לשמש כאיזשהו קבוע שמוסיפים ל-offset.

כל זאת הייצה סקירה קצרה בנושא רלוקציות, יש מגוון רחב של נושאים ובינהם חישוב רלוקציות וכתובות שכבר קשורים לסוג ארכיטקרטורה. מי שרוצה בהחלט מוזמן לקרוא על זאת ולהעשיר את הידע שלו.

GOT - Global Offset Table

זהו מקטע בהחלט מעניין. בעקרונו של דבר הוא שומר את הכתובות וה-offsets של ה-Symbols שאי אפשר לחשב את מיקומם בזמן הלינקוג'. ה-GOT בהחלט רלוונטי לרוב לכל הקשור לפונקציות וכו בספריות חיצוניות וכפי שנראה ה-PLT משתמש בו לשם קפיצה לתוך מימושן. נבהיר יותר על המושג בחלק הבא, ה-PLT.

PLT - Procedure Linkage Table

נחשוב על זאת, איך יתבצע הקישור בין פונקציות חיצוניות לתכנית שלנו? כיצד, למשל התכנית שלנו תקפוץ מהקריאה ל-printf למימוש שלה בספריה החיצונית?

אז כן, התשובה נעוצה בצורך של קיום של איזשהו תווך מקשר בין הקריאה למימוש. לאותו תווך מן הסתם קוראים PLT. מה שקורה בפועל הוא שכאשר נעשה printf, לא תהיה קפיצה ישירה למימוש של אותה פונקציה בספריה החיצונית, אלא תחילה מעבר ל-printf@plt כפי שניתן לראות בתרשים, ומשם כפי בעזרת הכתובות בתוך ה-GOT, תהיה קפיצה לתוך הספריה עצמה שנטענת בזמן ריצה. נציג את plt.:

```
Disassembly of section .plt:

00000000004005e0 <printf@plt-0x10>:
  4005e0:       ff 35 22 0a 20 00       push   QWORD PTR [rip+0x200a22]        # 601008 <_GLOBAL_OFFSET_TABLE_+0x8>
  4005e6:       ff 25 24 0a 20 00       jmp    QWORD PTR [rip+0x200a24]        # 601010 <_GLOBAL_OFFSET_TABLE_+0x10>
  4005ec:       0f 1f 40 00             nop    DWORD PTR [rax+0x0]

00000000004005f0 <printf@plt>:
  4005f0:       ff 25 22 0a 20 00       jmp    QWORD PTR [rip+0x200a22]        # 601018 <_GLOBAL_OFFSET_TABLE_+0x18>
  4005f6:       68 00 00 00 00          push   0x0
  4005fb:       e9 e0 ff ff ff          jmp    4005e0 <_init+0x20>

0000000000400600 <__libc_start_main@plt>:
  400600:       ff 25 1a 0a 20 00       jmp    QWORD PTR [rip+0x200a1a]        # 601020 <_GLOBAL_OFFSET_TABLE_+0x20>
  400606:       68 01 00 00 00          push   0x1
  40060b:       e9 d0 ff ff ff          jmp    4005e0 <_init+0x20>

0000000000400610 <__gmon_start__@plt>:
  400610:       ff 25 12 0a 20 00       jmp    QWORD PTR [rip+0x200a12]        # 601028 <_GLOBAL_OFFSET_TABLE_+0x28>
  400616:       68 02 00 00 00          push   0x2
  40061b:       e9 c0 ff ff ff          jmp    4005e0 <_init+0x20>

0000000000400620 <get_magic@plt>:
  400620:       ff 25 0a 0a 20 00       jmp    QWORD PTR [rip+0x200a0a]        # 601030 <_GLOBAL_OFFSET_TABLE_+0x30>
  400626:       68 03 00 00 00          push   0x3
  40062b:       e9 b0 ff ff ff          jmp    4005e0 <_init+0x20>
```

כעת אחרי שהכרנו את רוב המקטעים הרלוונטים, אפשר לעבור ולדון לגבי טעינת התהליך לזכרון.

Program headers

ELF משתמש ב-program headers שנקראים על ידי ה-loader של המערכת, וזאת כדי לציין כיצד התכנית אמורה להטען לזכרון הוירטואלי, מהם גבולותיו ואיזה מידע יהיה בו. הם מוגדרים כך:

```
typedef struct elf64_phdr {
Elf64_Word p_type;
Elf64_Word p_flags;
Elf64_Off p_offset;   /* Segment file offset */
Elf64_Addr p_vaddr;   /* Segment virtual address */
Elf64_Addr p_paddr;   /* Segment physical address */
Elf64_Xword p_filesz;        /* Segment size in file */
Elf64_Xword p_memsz;  /* Segment size in memory */
Elf64_Xword p_align;  /* Segment alignment, file & memory */
} Elf64_Phdr;
```

נעשה readelf -l magic:

```
Elf file type is EXEC (Executable file)
Entry point 0x400630
There are 9 program headers, starting at offset 64

Program Headers:
  Type           Offset             VirtAddr           PhysAddr
                 FileSiz            MemSiz              Flags  Align
  PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
                 0x00000000000001f8 0x00000000000001f8  R E    8
  INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
                 0x000000000000001c 0x000000000000001c  R      1
      [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
  LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
                 0x0000000000000954 0x0000000000000954  R E    200000
```

תיאור השדות:

- **p_type** - סוג הסגמנט בזכרון. כמה אפשרויות ששווה לדון בהן:
  - **PT_NULL** - מסמן כי הסגמנט לא בשימוש.
  - **PT_LOAD** - סגמנטים שאמורים להיות ממופים לזכרון לפני שהתכנית אמורה לרוץ.
  - **PT_DYNAMIC** - מציין כי הסגמנט מכיל מידע עבור ה-dynamic linker.
  - **PT_INTERP** - מציין כי הסגמנט מכיל את ה-path של ה-interpreter. בצילום נראה כי זהו ה-lib64/ld-linux-x86-64.so.2 / ובחזרה לאחורה נבחין גם כי זהו התוכן של מקטע הinterp. שהאינדקס שלו הוא 1. גודל הסגמנט הוא 0x1C וזהו אורך המחרוזת של הpath ועוד תו ה '\0'. מה הוא אותו interpreter? קודם כל נשלול את האפשרות שזה ה-interpreter, מפרש של שפות תכנות שאנחנו רגילים לשמוע עליו. אז בעקרונו של דבר, הכוונה פה היא בעצם לתכנית שיוצרת את ה-image הראשוני של התהליך. בנוסף תפקידו הוא למפות את הזכרון הנדרש לתהליך (למשל באיזו כתובת בדיוק יופיע מקטע ה-text וכו) וכמו כן לטעון בין היתר כל מה שקשור לספריות חיצוניות.
  - **PT_NOTE** - הסגמנט מכיל מידע נוסף שקשור כפי שראינו מקודם.

- **p_flags** - איזשהו bitmask שמציין את סוגי ההרשאות שיש לסגמנט. ההרשאות יכולות להיות קריאה, כתיבה והרצה.
- **p_offset** - ה-offset/מיקום מתחילת הקובץ של תחילת המידע על הסגמנט.
- **p_vaddr** - מספק לנו את הכתובת הוירטואלית הראשונה שאליה ממופה הסגמנט בזמן הרצת התהליך.
- **p_filesz** - גודל הסגמנט בקובץ.
- **p_memsz** - מציין את גודל הסגמנט בזכרון בזמן הרצה.
- **p_align** - מספק את ה-alignment של הסגמנט בזכרון. בהגדרה פשוטה, ואולי קצת לא מדויקת, אם למשל הalign הוא 8, אז תחילת המידע, משתנים, זכרון וכו יופיעו בכתובות זכרון שהן כפולות של 8.

Dynamic Linking

קבצי ELF יכולים להיות להטען כ-Dynamic Linking ועל כן אמור גם לזאת להיות אזכור בפורמט. במקטע ששמו dynamic. ניתן למצוא אוסף של Elf64_Dyn שמוגדרים כך:

```
typedef struct {
        Elf64_Sxword d_tag;    /* entry tag value */
        union {
                Elf64_Xword d_val;
                Elf64_Addr d_ptr;
        } d_un;
} Elf64_Dyn;
```

בעזרת readelf -d magic נראה:

```
Dynamic section at offset 0xe18 contains 25 entries:
  Tag        Type                         Name/Value
 0x0000000000000001 (NEEDED)             Shared library: [./mylib.so]
 0x0000000000000001 (NEEDED)             Shared library: [libc.so.6]
 0x000000000000000c (INIT)               0x4005c0
```

נתאר בקצרה את השדות:

d_tag - בפשטות, שדה זה מציין כיצד לפרש את הunion שבא בהמשך. כמה אפשרויות ששוה לדון בהן:

- **DT_NEEDED** - מסמל לרוב שיש צורך ב-shared object וכפי שניתן לראות בתמונה מדובר ב-mylib.so וב-libc שזוהי הספרייה הסטנדרתית של C. בתור ברירת מחדל יש להתייחס ל-d_val.
- **DT_INIT** - יש להתייחס לd_ptr ובו מצוין הכתובת של פונקצית האתחול.
- **DT_FINI** - כמו DT_INIT רק בהקשר של פונקצית הסיום.

נתייחס אל כל משתנה ב-union בנפרד:

- **d_val** - ערך שאפשר לפרשו בכמה דרכים.
- **d_ptr** - כתובת וירטואלית בזכרון.

אוקי, כעת כשאנו מציודים במספיק ידע ואנרגיות, אפשר להביא דוגמא מחיי היום יום. כפי שציינו מקודם, הקרנל של לינוקס הוא אכן גם קובץ ELF, רק קצת מיוחד. לפני שבאמת נכנס לתוך הפרטים המדויקים, כדאי שנציג את תהליך הboot בקצרה (אמנם ננסה להתמקד בפרטים תכנים, אך לא בפירוט רב למשל עד כדי ציון כתובות פרה-היסטוריות כמו 0x7C00, 0xFFFFFFF0) כדי לספק רקע מתאים.

אז מה היה בהתחלה?

יש לי מחשב "מכובה" (לידע כללי זה חלקית נכון, למשל יש את השעון החומרתי RTC - Real Time Clock שפועל כל עת, גם כשמחשב מכובה). בהנחה שמישהו לחץ על כפתור הpower, מועבר איזשהו איתות ל-CPU, דרך יציאה כלשהי. איתות זה גורם ל-CPU לבצע כמה פעולות, כשבסופו של דבר מורץ איזשהו firmware, או בעברית קושחה. אותו firmware מוכר לנו יותר כ-BIOS או UEFI. לצורך פשטות נמקד את התיאור לתהליך של ה-BIOS. לאחר בדיקות, הגדרות, למשל אצל ה-BIOS, בדיקת ה-POST, נבחר איזשהו bootable device, וממנו נקרא ה-MBR - Master Boot Record, שהוא הסקטור הראשון של אותו device (ה-UEFI בניגוד ל-BIOS יכול להשתמש ב-GUID GPT במקום הMBR, אך לא נתאר אותו פה).

עד פה אפשר להגיד שהחלק של עליית המחשב היה די כללי. מפה והלאה נכנסים מונחים שונים הקשורים לעולם הלינוקס.

ובחזרה, מבלי להתעמק במבנה ה-MBR, נציין שיש בו איזשהו boot code הנקרא boot.img,טבלת מחיצות, ולבסוף signature שערכו 0xAA55 שמורה ל-BIOS שזהו אכן bootable device. בגלל שאותו boot.img מוגבל מבחינת הגודל שלו (440 bytes), תפקידו, שעל פי השפה המקצועית מכונה Stage 1, הוא בין היתר הוא לבחור מחיצה מה-partition table, ולהעביר את השליטה למה שמכונה Stage 2 (יש מעבר על Stage 1.5, אבל הוא פחות רלוונטי פה). Stage 2 הוא מה שאנו מכירים בתור ה-GRUB, והוא אחראי על מרבית העבודה.

ה-GRUB בתורו, בורר מיקום מסויים בהארד דיסק, או ליתר דיוק מה שמכונה תיקיית ה-boot/. מה שקורה בפועל הוא ש-GRUB מתייעץ בקובץ קונפיגורציה (יותר נכון מפרסר אותו), בשם /boot/grub/grub.cfg.

אותו קובץ קונפיגורציה בעצם מהווה את התפריט של כל מערכות ההפעלה הקיימות על המחשב. נראה את חלקו הרלוונטי לנו:

```
if [ x$feature_platform_search_hint = xy ]; then
  search --no-floppy --fs-uuid --set=root --hint
0,msdos1 --hint-baremetal=ahci0,msdos1  77d0453a-

else
  search --no-floppy --fs-uuid --set=root 77d045

fi
linux   /boot/vmlinuz-4.2.0-16-generic root=UUID
9e39df2 ro  quiet splash $vt_handoff
initrd  /boot/initrd.img-4.2.0-16-generic
```

נשים לב לדברים הבאים:

- **vmlinuz-x.y.z** - אז בהחלט אנחנו מתקרבים לקרנל שלנו. מי שעוסק בדיבוג ובניית קרנלים, יודע שאחרי תהליך הקומפילציה של הקרנל של לינוקס יווצרו כמה קבצים. אחד מהם הוא ה-vmlinux שהוא קובץ הרצה, ELF המקורי של הקרנל (יש לשים לב שאי אפשר להריץ אותו ישירות. הכוונה הייתה שיש לו פורמט ELF). זהו הקרנל שאנו רגילים לשמוע עליו בתור האחראי לניהול תהליכים, scheduling זכרון והרשימה רק הולכת וגדלה. השני הוא ה-bzImage שהוא מכיל את הקרנל, vmlinux רק מכווץ ועוד כמה דברים. באופן כללי, ברגע שעושים make install בתוך ה-source code, ה-bzImage יועבר לתיקיית /boot, תחת השם vmlinuz-x.y.z. "באופן מפתיע", ה-vmlinuz הוא בעל פורמט PE. נראה זאת:

```
root@unknown:/boot# od -A d -t x1 vmlinuz-4.2.0-16-generic | grep '4d 5a'
0000000 4d 5a ea 07 00 c0 07 8c c8 8e d8 8e c0 8e d0 31
```

כאמור ה-magic number פה הוא MZ. אז מה הולך פה? תפקידו של אותו bzImage, הוא לעשות את כל הדברים שנחוצים לשם טעינת הקרנל, vmlinux. מלמד הקרנל המכווץ, אפשר למצוא בו קוד ו headers שאחראים בעיקר להכנת הסביבה (למשל מעבר בין modes שונים), ולטעינת הקרנל בשביל לעשות לו decompress.

שיטת הכיווץ הנפוצה ביותר של הקרנל היא gzip. נבדוק אותה בעצמנו:

```
root@unknown:/boot# od -A d -t x1 vmlinuz-4.2.0-16-generic | grep '1f 8b 08 00'
0018864 ac fe ff ff 1f 8b 08 00 00 00 00 00 02 03 ec fd
```

אם נסתכל בדוקומנטציה של gzip נראה שאכן 1f 8b הם ה-magic number של gzip header וה- 08 00 מציינים את אלגוריתם. הכיווץ שהוא DEFLATE ואת סוג המידע.

- initrd.img-x.y.z - לא נתעמק בפרטי הפרטים פה, אבל נציין שתפקידו של initrd הוא לאפשר בין היתר טעינה של מערכת קבצים זמנית של הקרנל ומודולים נוספים כדי להקל על תהליך ה-boot.

לאחר מכן מוצג התפריט הידוע של GRUB ובו המשתמש מתבקש לבחור מערכת הפעלה שברצונו לטעון. בהנחה שהמשתמש שלנו בחר מערכת לינוקס, ה-GRUB, מאתר את ה-vmlinuz-x.y.z של אותה מערכת (הקובץ קונפיגורציה של GRUB כאמור מספק מיקום של כל קרנל. ניתן לראות זאת על ידי מיליות כמו hdX, כאשר X מסמל את מספר המחיצה ו-hd מסמל hard drive, למי שתהה).

לאחר השתלשלות מעניינת של אירועים, שכוללים בין היתר מיפוי של הזכרון לחלקים מתאימים, אנו מוצאים את עצמנו "יחסית" בתוך הקרנל, או מה שנקרא בתוך הקובץ arch/x86/boot/header.S, שזוהי נקודת ההתחלה שלנו.

בהמשך אנחנו מגיעים ל-arch/x86/boot/compressed/misc.c ופה מתחילה ההתעסקות האמיתית עם ELF, בפונקציה decompress_kernel:

```
asmlinkage __visible void *decompress_kernel(void *rmode, memptr heap,
                             unsigned char *input_data,
                             unsigned long input_len,
                             unsigned char *output,
                             unsigned long output_len,
                             unsigned long run_size)
{
        . . .
        if (real_mode->screen_info.orig_video_mode == 7) {
                vidmem = (char *) 0xb0000;
                vidport = 0x3b4;
        } else {
                vidmem = (char *) 0xb8000;
                vidport = 0x3d4;
        }
        . . .
        debug_putstr("\nDecompressing Linux... ");
        decompress(input_data, input_len, NULL, NULL, output, NULL, error);
        parse_elf(output);
        /*
        * 32-bit always performs relocations. 64-bit relocations are only
        * needed if kASLR has chosen a different load address.
        */
        if (!IS_ENABLED(CONFIG_X86_64) || output != output_orig)
                handle_relocations(output, output_len);
        debug_putstr("done.\nBooting the kernel.\n");
}
```

בואו נעבור על הפונקציה:

בתחילתה יש אתחול של המשתנה vidmem. תפקידו הוא לאפשר פלט על המסך (I/O memory). בשלב זה נצטרך לכתוב הודעות למסך שמיוצג על ידי מערך של בתים. כצפוי יש הודעת דיבאג של "Decompressing Linux".

לאחר מכן קריאה ל-decompress ששמה מרמז על תכליתה. פונקציה זו תלויה שיטת כיווץ וכפי שניתן לראות היא ממומשת במקום אחר.

באותו קובץ misc.c נראה:

```
 ...
#ifdef CONFIG_KERNEL_GZIP
#include "../../../../lib/decompress_inflate.c"
#endif

#ifdef CONFIG_KERNEL_BZIP2
#include "../../../../lib/decompress_bunzip2.c"
#endif

#ifdef CONFIG_KERNEL_LZMA
#include "../../../../lib/decompress_unlzma.c"
#endif
. . .
```

כעת יש קריאה לפונקציה parse_elf שמצויה באותו קובץ ומוגדרת כך:

```
static void parse_elf(void *output)
{
#ifdef CONFIG_X86_64
 Elf64_Ehdr ehdr;
 Elf64_Phdr *phdrs, *phdr;
#else
 Elf32_Ehdr ehdr;
 Elf32_Phdr *phdrs, *phdr;
#endif
 void *dest;
 int i;

 memcpy(&ehdr, output, sizeof(ehdr));
 if (ehdr.e_ident[EI_MAG0] != ELFMAG0 || // '0x7F'
        ehdr.e_ident[EI_MAG1] != ELFMAG1 ||      // 'E'
        ehdr.e_ident[EI_MAG2] != ELFMAG2 ||      // 'L'
        ehdr.e_ident[EI_MAG3] != ELFMAG3) {      // 'F'
         error("Kernel is not a valid ELF file");
         return;
 }
 debug_putstr("Parsing ELF... ");

 phdrs = malloc(sizeof(*phdrs) * ehdr.e_phnum);
 if (!phdrs)
         error("Failed to allocate space for phdrs");

 memcpy(phdrs, output + ehdr.e_phoff, sizeof(*phdrs) * ehdr.e_phnum);

 for (i = 0; i < ehdr.e_phnum; i++) {
         phdr = &phdrs[i];
         switch (phdr->p_type) {
         case PT_LOAD:
#ifdef CONFIG_RELOCATABLE
                 dest = output;
                 dest += (phdr->p_paddr - LOAD_PHYSICAL_ADDR);
#else
                 dest = (void *)(phdr->p_paddr);
#endif
                 memcpy(dest,
                        output + phdr->p_offset,
                        phdr->p_filesz);
                 break;
         default: /* Ignore other PT_* */ break;
         }
 }
 free(phdrs);
}
```

נסביר כל חלק וחלק: בהתחלה יש הצהרה על headers שונים שרלוונטיים למערכת, בין אם זה 32bit או 64bit, ה-preprocessor יודע לעשות העתק הדבק מתאים. לאחר מכן יש העתקה פשוטה של ה- Elf header מהקרנל עשינו לו decompress מקודם למשתנה שלנו. ואז יש השוואה של הארבעה בתים ראשונים לmagic number של פורמט ה-ELF. כצפוי אם לא תהיה התאמה, תודפס הודעת השגיאה: "Kernel is not a valid ELF file".

כפי שציינו מקודם, ה-program headers, מציינים את האופן בו נטען התהליך, ואת מרחב הזכרון שלו. גם פה ניתן לראות, יש העתקה של אותם program headers, למערך מוקצה דינמית.

לאחר מכן יש לולאה שעוברת על כל header כזה.

במידה וסוג של header הוא PT_LOAD, כלומר שהסגמנט אמור להיטען לזכרון הדברים הבאים קורים:

- כך או כך, יש קבלה של איזשהו base address של הסגמנט. כפי שניתן לראות, אותה כתובת תלויה ב-p_paddr, כלומר הכתובת הפיזית של של הסגמנט. העניין הוא שבשלב זה עדין לא מופעל מנגנון הוירטואליזציה, (השמה של כתובת ה-base של page global table לאוגר cr3, והתעסקות עם ביטים באוגר cr0 כפי שניתן לראות בקובץ /arch/x86/kernel/head_64.S) ולכן נטען רק לשם את המידע.
- העתקה בעזרת פונקציה memcpy ל-dest את התוכן של אותו סגמנט שגודלו p_filesz.

לבסוף יש הדפסה של הודעת הצלחה ואנחנו ממשיכים במסענו. לאחר סדרה של אתחולים שונים (למשל הזכרון הוירטואלי), אנחנו מגיעים לפוקנציה start_kernel שמוגדרת ב-/init/main.c שגם היא אחראית להגדרות שונות כמו של זכרון, זמן. זו בתורה קוראת לrest_init שיוצרת את תהליך ה-init, והוא מריץ את הפונקציה kernel_init ובה תהיה נקודת הסיום שלנו:

```
static int __ref kernel_init(void *unused)
{
        …
        if (!try_to_run_init_process("/sbin/init") ||
                !try_to_run_init_process("/etc/init") ||
                !try_to_run_init_process("/bin/init") ||
                !try_to_run_init_process("/bin/sh"))
         return 0;
        …
}
```

סיכום

במאמר זה ניסיתי להציג ולתאר לעומק עד כמה שאפשר, שלל נושאים הקשורים ל-ELF. מן הסתם שאף פעם אי אפשר לכסות את כל הידע הקיים, אבל בהחלט כן השתדלתי לעסוק בדברים החשובים ביותר. תחום שלצערי לא נגעתי בו הוא כל הקשור ל-exploitation. כמובן תמיד אפשר לדבר על הדברים היותר מוכרים כמו מגוון buffer overflows, ret2libc, ROP וכו', אולם מה שתפס את תשומת ליבי בכל היכרותי עם תחום זה הוא דבר הנקרא GOT overwrite. בעקרונו של דבר, התוקף מנסה לשנות ולדרוס פה את הערכים ב-GOT בכדי לנצל זאת למטרותיו, ובמקום שהתכנית תריץ את מה שהיא צריכה להריץ, היא מריצה את מה שברצונו של התוקף. כמובן שתיאור זה היה פחות מעל קצה המזלג, ומי שרוצה בהחלט מוזמן לקרוא על זאת ועל עוד התקפות מעניינות.

ובתור מילים באמת אחרונות, אני חייב לציין שבאמת נהנתי לכתוב מאמר זה. אם מצאתם משהו לא מובן, לא נכון וכל טעות, קטנה ככל שתהיה או שאתם רוצים ליצור איתי קשר, ניתן לפנות אלי באימייל:

dexr4de@gmail.com