International Journal of Innovative Research in Engineering & Management(IJIREM)
ISSN: 2350-0557, Volume-1, Issue-3, November-2014

Study and Analysis of ELF Vulnerabilities in Linux

Biswajit Sarma
Assistant professor, Department of Computer Science and Engineering,
Jorhat Engineering College,

Srishti Dasgupta
Final year student, Department of computer Science and Engineering,
Jorhat Engineering College

ABSTRACT

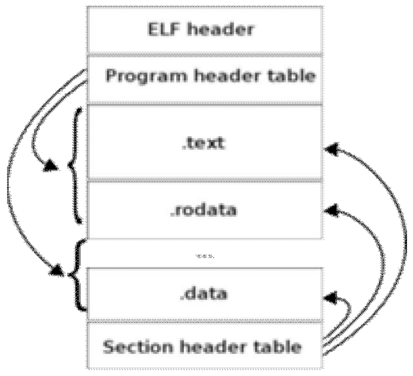
Internally the Linux kernel uses a binary format loader layer to implement the low level format dependent functionality of the execve() system call to replace current process by a new one or to execute a new process. The common execve() code contains just few helper functions used to load the new binary and leaves the format specific work to a specialized binary format loader. One of the Linux format loaders is the ELF (Executable and Linkable Format) loader. There are three header areas in an ELF files: the main ELF header, the program headers, and then the section headers. The program code lies between the program headers and the section headers. This paper will take a look at the Linux ELF file format and examine possibilities of file virus infectors.

Keywords

Linux, Execve, Loader, Virus.

1. INTRODUCTION

In computing, the Executable and Linkable Format (ELF, formerly called Extensible Linking Format) is a common standard file format for executables, object code, shared libraries, and core dumps. An executable file using the ELF file format consists of an ELF header, followed by a program header table or a section header table, or both. The ELF header is always at offset zero of the file. The offset of the program header table and the section header table in the file are defined in the ELF header.



2. THE ELF HEADER

2.1 The Main ELF File header

We start with the file header analysis. The following structures are available in elf.h.The fields of interest to us are e_entry, e_phoff,e_shoff, and the sizes given. e_entry specifies the location of _start, e_phoff shows us where the array of program headers lies in relation to the start of the executable, and e_shoff shows us the same for the section headers. Here is the structure of ELF file header.

```
typedef struct
{
unsigned char e_ident[EI_NIDENT];/* Magic number and other info */
  Elf32_Half    e_type;          /* Object file type */

  Elf32_Half    e_machine;       /* Architecture */

  Elf32_Word    e_version;       /* Object file version */

  Elf32_Addr    e_entry;         /* Entry point virtual address */

  Elf32_Off     e_phoff;         /* Program header table file offset */

  Elf32_Off     e_shoff;         /* Section header table file offset */

  Elf32_Word    e_flags;         /* Processor-specific flags */

  Elf32_Half    e_ehsize;        /* ELF header size in bytes */

  Elf32_Half    e_phentsize;     /* Program header table entry size */

  Elf32_Half    e_phnum;         /* Program header table entry count */

  Elf32_Half    e_shentsize;     /* Section header table entry size */

  Elf32_Half    e_shnum;         /* Section header table entry count */

  Elf32_Half    e_shstrndx;      /* Section header string table index */

} Elf32_Ehdr;
```

2.2 The Program Header

The next portion of the program are the ELF program headers. These describe the sections of the program that contain executable program code to get mapped into the program address space as it loads.

```
typedef struct

{

  Elf32_Word    p_type;               /* Segment type */

  Elf32_Off     p_offset;             /* Segment file offset */

  Elf32_Addr    p_vaddr;              /* Segment virtual address */

  Elf32_Addr    p_paddr;              /* Segment physical address */

  Elf32_Word    p_filesz;             /* Segment size in file */

  Elf32_Word    p_memsz;                /* Segment size in memory */

Elf32_Word    p_flags;              /* Segment flags */

Elf32_Word    p_align;              /* Segment alignment */

} Elf32_Phdr;
```

2.3 The ELF Body

The body of the ELF file excluding the header portion comes next. The actual locations and sizes of portions of the body are described by the program headers above, and these contain the executable instructions from our assembly file, as well as string constants and global variable declarations.

2.4 ELF Section Headers

The ELF section headers describe various sections of different names in an executable file. Each section has an entry in the section headers array, which is found at the bottom of the executable file and has the following format.

```
typedef struct
{

  Elf32_Word    sh_name;      /* Section name (string tbl index) */

  Elf32_Word    sh_type;      /* Section type */

  Elf32_Word    sh_flags;     /* Section flags */

  Elf32_Addr    sh_addr;    /* Section virtual addr at execution */

  Elf32_Off     sh_offset;      /* Section file offset */

  Elf32_Word    sh_size;            /* Section size in bytes */

  Elf32_Word    sh_link;            /* Link to another section */

  Elf32_Word    sh_info;        /* Additional section information */

  Elf32_Word    sh_addralign;          /* Section alignment */

  Elf32_Word    sh_entsize;   /* Entry size if section holds table */

} Elf32_Shdr;
```

The section headers are entirely optional. ELF is extremely flexible. Many of these sections can be shrunk, expanded, removed, etc. In fact, it is not outside the realm of possibility that some programs may deliberately make abnormal modifications. One can run .text section(only) of one ELf binary in another ELF binary. This is because how the linux (ELF) virus is written. Generally when we compile a c program one ELF binary is created and it has all the elements that create a process image in the memory and start executing. But the question is how the process is loaded in the memory. If we allocate some memory and place the binary ELF in that memory it is not going to be executed though it has all components of an ELF binary. Because the loader loads the ELF binary file in memory in a meaningful way so that the processor starts execution.

3. AN EXAMPLE

We have a C program tell.c

```
#include<stdio.h>
int main()
{
            printf("\n hello");
            return 0;
}
```

compile the program using the gcc compiler $gcc -o tell tell.
Using program readelf we can read this our ELF binary .(i.e tell)

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 03 00 00 00 00 00 00 00 00
  Class:                       ELF32
  Data:                        2's complement, little endian
 Version:                      1 (current)
  OS/ABI:                      UNIX - Linux
 ABI Version:                  0
  Type:                                EXEC (Executable file)
  Machine:                             Intel 80386
  Version:                             0x1
  Entry point address:                 0x8048310
  Start of program headers:            52 (bytes into file)
  Start of section headers:            2148 (bytes into file)
  Flags:                               0x0
  Size of this header:                 52 (bytes)
  Size of program headers:             32 (bytes)
```

Number of program headers:	8
Size of section headers:	40 (bytes)
Number of section headers:	30
Section header string table index:	27

All the header information are available. While loading ,the loader will read the first 4 bytes of the binary file and will confirm that this is an ELF binary file. Here Elf file type is EXEC (Executable file) and Entry point is 0x8048310.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  0x00530  0x00530
R E 0x1000
LOAD  0x000530  0x08049530  0x08049530  0x000fc  0x00104
RW 0x1000
DYNAMIC    0x000544   0x08049544   0x08049544   0x000c8
0x000c8 RW  0x4
NOTE    0x000148 0x08048148 0x08048148 0x00044 0x00044 R
0x4
GNU_EH_FRAME   0x0004ac 0x080484ac 0x080484ac 0x0001c
0x0001c R   0x4
GNU_STACK      0x000000 0x00000000 0x00000000 0x00000
0x00000 RW  0x4
```

```
Section to Segment mapping:
  Segment Sections...
   00
   01     .interp
   02     .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym
.dynstr .gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text
.fini .rodata .eh_frame_hdr .eh_frame
   03     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
   04     .dynamic
   05     .note.ABI-tag .note.gnu.build-id
   06     .eh_frame_hdr
   07
```

One of the important thing is Page size of the running machine. Our Machine page size is 4kb (0x1000=hex equivalent of 4096). We start with details of the program header.

```
Type   Offset   VirtAddr    PhysAddr    FileSiz   MemSiz
Flg  Align
PHDR   0x000034 0x08048034 0x08048034 0x00100 0x00100
R E  0x4
```

The program header itself taking 256 bytes(8x32=256),starting at offset 0x34 in the file and file size is 256bytes.(0x00100=hex equivalent of 256).The flage is Read and Execute.

```
Type    Offset    VirtAddr  PhysAddr   FileSiz    MemSiz
Flg Align
INTERP  0x000134 0x08048134 0x08048134 0x00013 0x00013
R   0x1
          [Requesting program interpreter: /lib/ld-linux.so.2]
```

The program is that which should be used to 'execute' the binary. Here, it reads as '/lib/ld-linux.so.2', which means some dynamic library linking will be required before we run the program. The flag is READ only.

```
Type    Offset   VirtAddr    PhysAddr   FileSiz   MemSiz
Flg    Align
LOAD  0x000000 0x08048000 0x08048000 0x00530 0x00530 R
E     0x1000
```

This line requests to read 0x530 bytes, starting at file's start and being 0x530 bytes large (that's virtually the whole file!), which will be read-only but executable. The file will appear to start at virtual address 0x08048000 for the program to work properly and we want the whole page in memory.

```
Type   Offset    VirtAddr     PhysAddr  FileSiz    MemSiz
Flg   Align
LOAD   0x000530  0x08049530  0x08049530  0x000fc  0x00104
RW  0x1000
```

With more bits to load, (likely to be .data section because the flag

is WR) it is noticeable that the 'filesize' and 'memsize' differ,

which means the .bss section will actually be allocated through this statement, but left as zeroes while 'real' data only occupy first 0xfc bytes starting at virtual address 0x8049530. Now we calculate the memory size 104. If we add the size of .ctors to .bss we will find 0x104. This information is stored in one page.

```
Type          Offset     VirtAddr     PhysAddr      FileSiz
MemSiz   Flg   Align
DYNAMIC      0x000544  0x08049544  0x08049544  0x000c8
0x000c8    RW  0x4
```

The dynamic sections are used to store information used in the dynamic linking process, such as required libraries and relocation entries.

```
Type       Offset     VirtAddr  PhysAddr  FileSiz  MemSiz
Flg  Align
```

```
NOTE       0x000148 0x08048148 0x08048148 0x00044 0x00044
R     0x4
```

NOTE sections contain information left by either the programmer(comment lines) or the linker, for most programs linked using the GNU `ld` linker it just says 'GNU'.

```
Type                 Offset      VirtAddr    PhysAddr   FileSiz
MemSiz  Flg Align
GNU_EH_FRAME   0x0004ac 0x080484ac 0x080484ac 0x0001c
0x0001c   R   0x4
```

that's for Execption Handler information, in case we should link against some C++ binaries at execution (afaik).

```
Type                 Offset      VirtAddr     PhysAddr   FileSiz
MemSiz   Flg  Align
GNU_STACK       0x000000 0x00000000 0x00000000 0x00000
0x00000  RW  0x4
```

The stack information of the process. The most important thing at this level is the entry point. In one place we find that

```
Type              Offset      VirtAddr       PhysAddr       FileSiz
MemSiz  Flg    Align
LOAD              0x000000  0x08048000  0x08048000  0x00530
0x00530 R E 0x1000
```

But the entry point is 0x8048310 . From above we know that the starting address is 0x8048000 . So, from where does this 0x310 come from? If we look at the section of the ELF header we see that the .text section address is 0x8048310. So the execution starts at the address 0x8048310 section [13]. There are 30 section headers, starting at offset 0x864:

```
Section Headers:
[Nr] Name              Type        Addr      Off       Size    ES   Flg
Lk Inf Al
 [ 0]                      NULL    00000000 000000 000000    00
0    0    0
[ 1] .interp          PROGBITS 08048134    000134 000013 00    A
0   0  1
[ 2] .note.ABI-tag     NOTE        08048148 000148 000020 00   A
0   0  4
[ 3] .note.gnu.build-i NOTE        08048168 000168 000024 00   A
0   0  4
  [ 4] .gnu.hash   GNU_HASH    0804818c 00018c 000020 04    A
5   0  4
 [ 5] .dynsym       DYNSYM       080481ac 0001ac 000050 10   A
6   1  4
 [ 6] .dynstr          STRTAB              080481fc 0001fc 00004c 00
A  0   0  1
 [ 7] .gnu.version    VERSYM        08048248 000248 00000a 02
A  5   0  2
 [ 8] .gnu.version_r  VERNEED 08048254 000254 000020 00    A
6   1  4
 [ 9] .rel.dyn           REL                08048274 000274 000008 08
A  5   0  4
 [10] .rel.plt         REL              0804827c 00027c 000018 08   A
5  12  4
 [11] .init          PROGBITS     08048294 000294 000030 00  AX
0   0  4
 [12] .plt           PROGBITS      080482c4 0002c4 000040 04  AX
0   0  4
 [13] .text        PROGBITS       08048310 000310 00016c 00  AX
0   0 16
  [14] .fini      PROGBITS       0804847c 00047c 00001c 00   AX
0   0  4
 [15] .rodata     PROGBITS        08048498 000498 000013 00   A
0   0  4

[16] .eh_frame_hdr  PROGBITS  080484ac 0004ac 00001c 00   A
0   0  4
 [17] .eh_frame    PROGBITS    080484c8 0004c8 000068 00   A
0   0  4
 [18] .ctors          PROGBITS        08049530 000530 000008 00
WA  0   0  4
 [19] .dtors       PROGBITS     08049538 000538 000008 00  WA
0   0  4
 [20] .jcr          PROGBITS     08049540 000540 000004 00  WA
0   0  4
 [21] .dynamic   DYNAMIC    08049544 000544 0000c8 08  WA
6   0  4
 [22] .got       PROGBITS       0804960c 00060c 000004 04  WA
0   0  4
 [23] .got.plt       PROGBITS         08049610 000610 000018 04
WA  0   0  4
 [24] .data         PROGBITS          08049628 000628 000004 00
WA  0   0  4
 [25] .bss         NOBITS          0804962c 00062c 000008 00  WA
0   0  4
 [26] .comment      PROGBITS    00000000 00062c 00013b 00
0   0  1
  [27] .shstrtab          STRTAB        00000000 000767 0000fc 00
0   0  1
  [28] .symtab      SYMTAB           00000000 000d14 000410 10
29  45  4
 [29] .strtab           STRTAB           00000000 001124 0001fc 00
0   0  1

Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor
specific)
```

This is the section header information from that ELF binary. So the file start it section from 0x8048000 and load the file. If we calculate the size from starting to the section .eh_frames we get 0x530, which is in one page.

4. Finding the vulnerabilities in the ELF

We have a little information about the ELF file and now we can think of a plane where we can run the .text section of one program to another ELF.

Plane one: At first we thought of adding a .text section at EOF of one working ELF and then of running this. The output is not the same as that of the natural execution done earlier . This is because that .text section is not part of the process , so the entry point is at proper place and it executes normally.

Plane two: Then we noticed the hole in virtual memory between the code segment and the data one. But we were not able to use it. Overwriting on section is not used as it is not safe.

Plane three:Finally we decided to look around the segments, not the section. Adding a new segment is possible but when we tried to relocate the program header we encountered a few hurdles. So we decided to enlarge one segment. As we wanted to code a non-destructive virus, we chose to append to the file, not to overwrite a part of it. The only way then is to enlarge a segment. As we appended to the file, the only enlargeable segment was the data one. The new virtual entry point is easy to calculate from the virtual address of the data segment. We were able to run code before the host.

Plane four:The data segment size is not the same in memory and in the file. It now contains the rest of the file in memory, with our viral code appended. we thought the segmentation faults came from that. And we were right. The .bss section problem came up. Normally, this section's flag is SHT_NOBITS. This section should not contain bytes from the programs. But we now understand why the data segment stops just before the text section,i.e, .txtsegment in the file; the program bytes are copied in memory up to our code. And the .bss section has to be filled of zero. So we decided to overwrite the .bss section with zeros at run time before returning to the host. We encountered segmentation fault once again.

Plane five: An alternative is to add the .text section at the end of working ELF and then try to change the ELF program header so that the new .text section is a part of the process and it gets executed. But we try the above plane and fail because of the memory allotment. We are not able to allot memory properly to that section. Suddenly we find that the NOTE section in the ELF program header is doing nothing. Since this >text section is a LOAD section and it is like the same as the first LOAD in the ELF so, we copy that LOAD to NOTE. Thus, we have now three LOAD sections.

```
[root@localhost 1]# readelf -l lsn


Elf file type is EXEC (Executable file) Entry point 0x08048310
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 0x00530 0x00530 R E 0x1000
  LOAD           0x000530 0x08049530 0x08049530 0x000fc 0x00104 RW  0x1000
  DYNAMIC        0x000544 0x08049544 0x08049544 0x000c8 0x000c8 RW  0x4
  LOAD           0x000000 0x08048000 0x08048000 0x00530 0x00530 R E 0x1000  [*]
  GNU_EH_FRAME   0x0004ac 0x080484ac 0x080484ac 0x0001c 0x0001c R   0x4
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
```

[*] shows that we are able to replace the NOTE program header with a new LOAD program header. If one looks at this program header carefully the flags are R and E ,i.e., read and execute. So modification is possible in ELF program header. But if we have a look at the entry point of the program ,it is still 0x08048310 and if we run the program , it will execute normally. We even are not sure what that 3rd LOAD does. So we have to modify the 3rd LOAD so that the process has the 3rdLOAD able section. We now try to modify the 3rd LOAD section . We know its offset already. So, we have to LOAD the 3rd loadable section but since we are new to this field of research , we will load the whole file now.

At the end we try to modify the entry point according to the newly LOAD section(by adding the extra bytes so it skips to appropriate place) . So the program starts with execution of the new LOAD section.

5. CONCLUSION

We have studied the internal details of the ELF file format. The main intention in this paper is to discuss how the process is created in the system from the ELF file and also we have tried to find a vulnerable point in the ELF file format. These information can be used for further research work in different fields related to ELF file format.

REFERENCES

[1] Ryan O'Neill "Extending the ELF Core Format for Forensics Snapshots" in Leviathan Research November 2014
[2] Ian Lance Taylor "A New ELF Linker" in iant@google.com
[3] Marius Van Oers "LINUX VIRUSES – ELF FILE FORMAT" in VIRUS BULLETIN CONFERENCE, SEPTEMBER 2000
[4] wikipedia.org