

SEGURIDAD EN SISTEMAS OPERATIVOS

4º Grado en Informática

Curso 2018-19

Práctica 2.- Ingeniería inversa y vulnerabilidades

Sesión 1.- El formato ELF (*Executable and Linkable Format*) en Linux

Objetivo: Conocer la estructura de un binario ELF y las herramientas que nos permite analizarlo de cara a realizar ingeniería inversa, en especial de binarios sospechosos de contener *malware*.

1.- ELF

Esta es la primera sesión de dos dedicada al análisis y explotación de archivos ELF. En ella, nos centraremos en ver la estructura de un archivo ejecutable ELF y las herramientas básicas para ver su contenido. En la Sesión siguiente veremos las posibles explotaciones de este formato y las protecciones que se le han añadido (ASLR -*Address Space Layout Randomize*-, *Stack Canary*, Pila no ejecutable, RELRO -*Read-only Relocation*-, soporte PIE, codificación binario-texto, etc). No obstante, el formato sigue siendo vulnerable de cara a posibles infecciones víricas.

Un archivo ELF es el formato para los archivos ejecutables, bibliotecas y módulos de carga que se usa en Linux y otros sistemas¹ (reemplazando a los formatos COFF y a.out). Es un formato flexible y extensible y no está vinculado a ningún procesador o arquitecturas particulares.

Los instrumentos que podemos usar para visualizarlo son:

<code>readelf</code>	utilidad que muestra información sobre uno o varios ELFs suministrada en las binutils.
<code>elfdump</code>	orden para ver información de un ELF, disponible en Solaris y FreeBSD
<code>objdump</code>	permite visualizar el contenido de archivos en varios formatos entre ellos ELF. Utiliza la biblioteca de descriptores de archivos binarios como back-end para estructurar los ELFs.
<code>file</code>	permite mostrar alguna información sobre un ELF como la arquitectura del conjunto de instrucciones, si el código es reubicable, etc.
<code>stings</code>	muestra las cadenas de texto incluidas en un binario.

Cada archivo ELF² se compone de una cabecera al inicio del archivo y hasta dos tablas de cabeceras, que sirven para dotarlo de dos vistas: una *vista de segmentos* (la cual se indexa mediante un vector de estructuras denominadas *Tablas de Cabeceras de Programa*) y una *vista de secciones* (indexada por un vector de estructuras denominada *Tabla de Cabecera de Sección*), como muestra la Figura 1. La primera vista se encarga de indicarle al kernel de sistema operativo, o al enlazador en caso de que sea un binario dinámico, que porciones del archivo se cagan y en qué partes del espacio

1 Algunos sistemas que usan este formato son: Linux, Solaris, FreeBSD, HP-UX, QNX Neutrino, Haiko, PS 2 y 3, Game Cube, Wii, y algunos teléfonos móviles como Symbian, Motorola, etc.)

2 TIS Committee, "Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification. Versión 1.2" de 1995 y disponible en <http://www.cs.princeton.edu/courses/archive/spring11/cos217/reading/elf.pdf>

de direcciones del proceso, y cuales son los permisos de las mismas (lectura, escritura y ejecución). La segunda vista tiene como finalidad dar al enlazador información sobre qué bibliotecas se necesitan cargar, los símbolos a exportar e importar, etc.

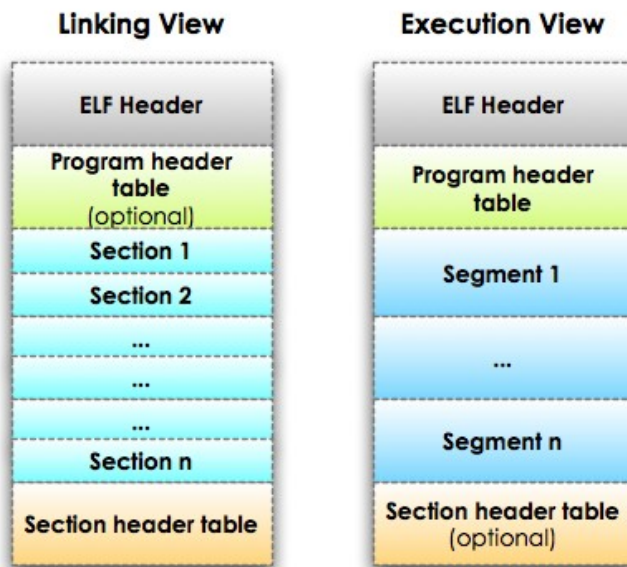


Figura 1.- Dos vistas de un ELF³.

Lo importante es que ambas tablas no cubren todo el binario, y en la tabla de segmentos hay zonas que incluso se solapan. Esto tiene como consecuencia que existen zonas no referenciadas por ninguna de las dos tablas, existen espacios vacíos que podríamos modificar de cara al funcionamiento del binario.

La Figura 2 muestra la cabecera de un segmento, denominada *cabecera de programa*, que especifica cómo y dónde puede cargarse en memoria y con qué permisos.

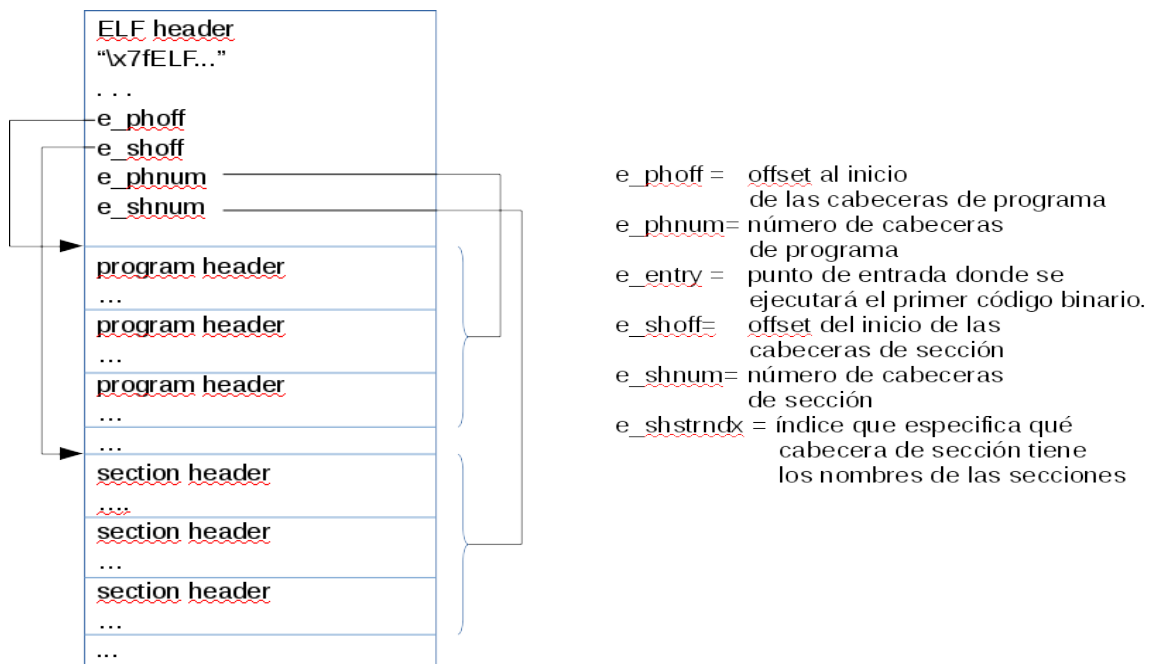


Figura 2.- Cabeceras de un binario ELF.

3 Imagen extraída de <http://www.sw-at.com/blog/2011/04/01/dissecting-executable-and-linking-format-elf/>.

Una de las secciones del ELF está destinada a contener la tabla de cadenas de cabeceras de sección, que es la concatenación de cadenas terminadas en nulos que indican los nombres de las secciones. Cada cabecera de sección tiene un desplazamiento en esta tabla de cadenas que indica su nombre, como muestra la Figura 3.

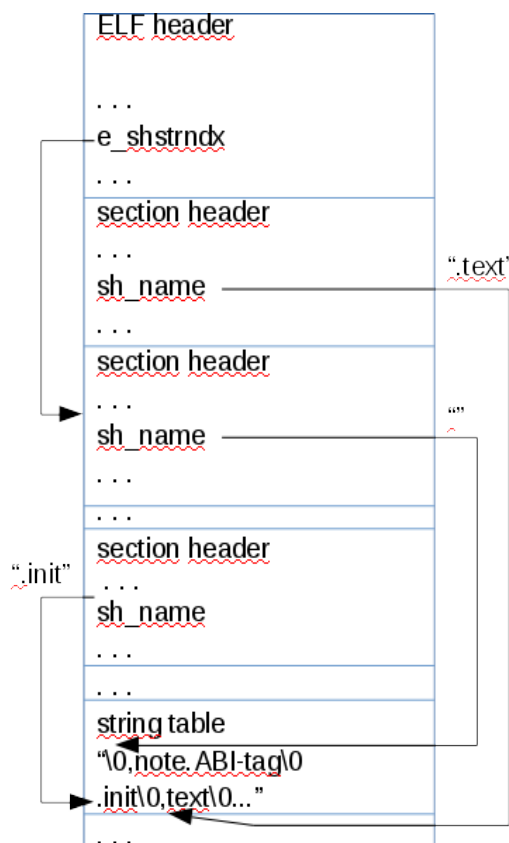


Figura 3.- Sección de nombres.

Los segmentos no tienen nombre, pero contienen de secciones que sí los tienen, así como tipos y permisos. Los tipos más interesantes son `LOAD` y `NOTE`. Un binario típico tiene dos segmentos `LOAD` y uno `NOTE`.

Un segmento **LOAD** tendrá permisos de lectura y ejecución y contiene el código del programa, en particular las secciones denominadas `.init`, `.text` y `.fini` que, respectivamente, contienen el código a ejecutar antes, durante y después de la ejecución del programa.

El otro segmento **LOAD** tiene permisos de lectura y escritura, contiene los datos del programa tales como los vectores estáticos, la lista de constructores y destructores, que aparecen en las secciones `.data`, `.ctors`, y `.dtors`.

El segmento NOTE contiene una única sección, la `.note.ABI-tag`. Si está presente, indica las versiones anteriores de Linux en la que el código es compatible. Esta sección y segmento, tal como especifica Linux Standard Base -LSB, no son necesario para al ejecución del programa. Asumiendo una versión del kernel reciente, esta información puede eliminarse.

La Figura 4 muestra la disposición típica de secciones y segmentos. Observar que un segmento en memoria puede ser mayor que lo es en disco. En tal caso, esta memoria extra se rellena a ceros.

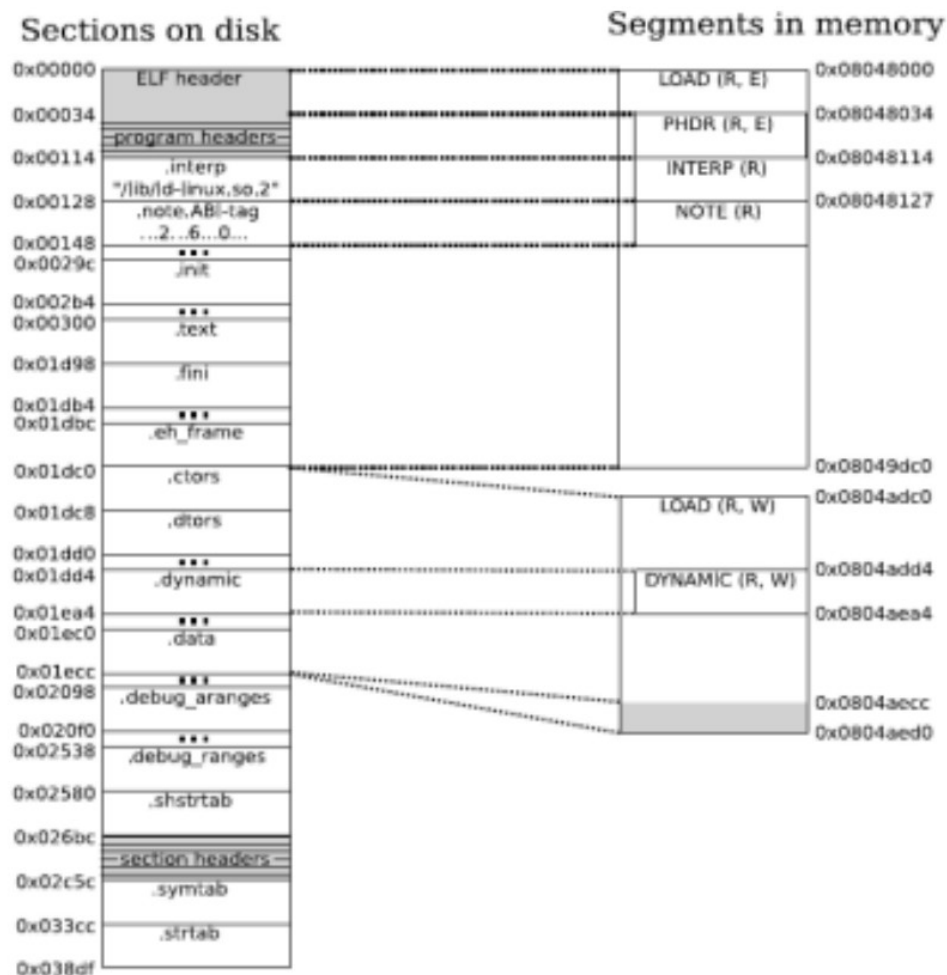


Figura 4.- Secciones y segmentos típicos de un ELF.

1.1 Diseccionando un archivo ELF

Podemos tomar un primer contacto con un archivo ELF con la orden file:

```
$ file /usr/bin/ls
```

```
/usr/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.16,
BuildID[sha1]=0x4cb12bd8e66f9102c0ba3a260298578b84dcad87, stripped
```

Que nos suministra información de que es un archivo ELF para una arquitectura de 32 bits y little-endian (LSB- Least Significant Bit). Además, nos indica que es “stripped” es decir mantiene información de depuración (fue compilador con la opción `-g` del compilador `gcc`). Podemos eliminar esta información y por tanto reducir el tamaño del ejecutable con la orden `strip`.

Los primeros 16 bits del archivo almacenan lo que se denomina número “mágico” que no es

otra cosa que una marca o identificador de que el archivo es un ejecutable. En concreto los cuatro primeros bytes contienen “\x7fELF”. Podemos ver la cabecera del archivo ELF con la orden:

```
$ readelf -h /usr/bin/ls
```

Encabezado ELF:

```
Mágico:  7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
Clase:                                     ELF32
Datos:                                     complemento a 2, little endian
Versión:                                   1 (current)
OS/ABI:                                    UNIX - System V
Versión ABI:                              0
Tipo:                                      EXEC (Fichero ejecutable)
Máquina:                                   Intel 80386
Versión:                                   0x1
Dirección del punto de entrada:           0x804c5d8
Inicio de encabezados de programa: 52  (bytes en el fichero)
Inicio de encabezados de sección: 115832 (bytes en el fichero)
Opciones:                                 0x0
Tamaño de este encabezado:                 52 (bytes)
Tamaño de encabezados de programa: 32 (bytes)
Número de encabezados de programa: 9
Tamaño de encabezados de sección: 40 (bytes)
Número de encabezados de sección: 31
Índice de tabla de cadenas de sección de encabezado: 30
```

Podemos ver las secciones que componen el archivo ELF con la orden:

```
$ readelf -S /usr/bin/ls
```

Hay 31 encabezados de sección, comenzando en el desplazamiento: 0x1c478:

Encabezados de Sección:

[Nr]	Nombre	Tipo	Direc	Desp	Tam	ES	Opt	En	Inf	Al
[0]		NULL	00000000	000000	000000	00		0	0	0
[1]	.interp	PROGBITS	08048154	000154	000013	00	A	0	0	1
[2]	.note.ABI-tag	NOTE	08048168	000168	000020	00	A	0	0	4
[3]	.note.gnu.build-id	NOTE	08048188	000188	000024	00	A	0	0	4
[4]	.hash	HASH	080481ac	0001ac	000430	04	A	6	0	4
. . .										

Entre ellas encontramos:

.text	Sección de código
.data, .bss, .rodata	Secciones de datos inicializados, no inicializados y de solo lectura
.comment	Comentarios
.rel.*	Tablas de reubicación
.symtab	Tablas de símbolos
.shstrtab	Tablas de cadenas que almacenan los nombres de cada sección
.strtab	Tablas de cadenas

Ejercicio 1.- Construye y compila un programa simple, por ejemplo uno similar al “Hola, Mundo”, en dos versiones: una en C y otra en C++.

(a) Consulta los manuales, o en Internet que contienen las secciones .interp, .got, got.ptl.

(b) Compara los ELFs de las dos versiones listando las secciones ¿hay alguna diferencia relevante respecto a las secciones de un programa compilado con gcc? ¿qué contienen las secciones .ctors y .dtors?

(c) Con la opción `readelf -r` podemos ver las secciones de reubicación. Indicar que contienen estas secciones.

La Tabla de cabeceras de sección no se carga en memoria, dado que ni el kernel ni el cargador dinámico son capaces de utilizarla. Para cargar el archivo en memoria se utilizan las cabeceras de programa para suministrar la información necesaria. Estas las podemos ver con `readelf -W -l`. Por ejemplo:

```
$ readelf -W -l /usr/bin/ls
```

El tipo del fichero elf es EXEC (Fichero ejecutable)

Punto de entrada 0x804c5d8

Hay 9 encabezados de programa, empezando en el desplazamiento 52

Encabezados de Programa:

Tipo	Desplaz	DirVirt	DirFísica	TamFich	TamMem	Opt	Alin
PHDR	0x000034	0x08048034	0x08048034	0x00120	0x00120	R E	0x4
INTERP	0x000154	0x08048154	0x08048154	0x00013	0x00013	R	0x1
[Se solicita el intérprete de programa: /lib/ld-linux.so.2]							
LOAD	0x000000	0x08048000	0x08048000	0x1b034	0x1b034	R E	0x1000
LOAD	0x01be68	0x08064e68	0x08064e68	0x004d8	0x0111c	RW	0x1000
DYNAMIC	0x01bec4	0x08064ec4	0x08064ec4	0x00108	0x00108	RW	0x4
NOTE	0x000168	0x08048168	0x08048168	0x00044	0x00044	R	0x4
GNU_EH_FRAME	0x017f24	0x0805ff24	0x0805ff24	0x0073c	0x0073c	R	0x4
GNU_STACK	0x000000	0x00000000	0x00000000	0x00000	0x00000	RW	0x4
GNU_RELRO	0x01be68	0x08064e68	0x08064e68	0x00198	0x00198	R	0x1

mapeo de Sección a Segmento:

Segmento Secciones...

. . .

Como podemos ver cada cabecera se corresponde con un segmento donde podemos encontrar secciones dentro de él. Pero ¿cómo funciona?. Al inicio, cuando el kernel ve el segmento `INTERP`, carga el primer segmento `LOAD` en la dirección virtual especificada, después carga los segmentos del programa intérprete (`/lib64/ld-x86-64.so.2`) y salta al punto de entrada del interprete. Tras lo cual, el cargado toma el control y carga la bibliotecas especificadas en `LD_PRELOAD` y también los segmentos `DYNAMIC` necesarios del ejecutable (podemos verlos con `readelf -d`).

Después de la reubicación, el cargador invoca todas las funciones de biblioteca `INIT` y salta al punto de entrada del ejecutable.

En estático, hay menos cosas que hacer ya que el kernel solo carga los segmentos `LOAD` en la dirección virtual y salta al punto de entrada.

Ejercicio 2.- Mira en el manual en línea o en Internet las opciones de la orden `objdump`, e indica:

(a) qué opciones nos permiten ver la información que nos suministra `readelf`.

(b) qué otras opciones nos permite realizar `objdump` desde el punto de la ingeniería inversa.

Los sistemas Linux utilizan el pseudo-sistema de archivos `/proc` para mostrar la información del kernel al resto del sistema. En concreto, existen una serie de directorios cuyo nombre es `/proc/<PID>`, donde `<PID>` representa el `PID` de un proceso en el sistema, que nos muestran información de los procesos que se están ejecutando. En especial, el archivo `/proc/<PID>/maps` muestra una línea por cada uno de los segmentos del proceso en ejecución. Por ejemplo,

```
% cat /proc/3276/maps
08048000-080da000 r-xp 00000000 08:02 1181970 /bin/bash
080da000-080db000 r--p 00091000 08:02 1181970 /bin/bash
080db000-080dd000 rw-p 00092000 08:02 1181970 /bin/bash
080dd000-080e4000 rw-p 00000000 00:00 0
084e4000-0860b000 rw-p 00000000 00:00 0 [heap]
b7359000-b735b000 rw-p 00000000 00:00 0
b735b000-b735d000 r-xp 00000000 08:02 923011 /usr/lib/gconv/ISO8859-1.so
...
```

En la información anterior, la primera columna indica las direcciones virtuales de inicio y fin de la región de memoria, a continuación se muestra los bits de protección de la región (r, w, y x) seguidos de una letra que indica si la región es el resultado de una proyección o mapeo privado (p) o compartido (s). La tercera columna representa el desplazamiento dentro del archivo donde se ha realizado el mapeo. La cuarta columna es el número principal y secundario del dispositivo donde reside el archivo origen de la proyección. La quinta y sexta columna representan el inodo y nombre del archivo, respectivamente, desde el cual se ha realizado la proyección.

Ejercicio 3.- Modifica el programa realizado en el ejercicio anterior para que el programa se detenga durante un rato, por ejemplo, con un `sleep()`, al objeto de que podamos visualizar el archivo `/proc/<PID>/maps` de su ejecución. Ahora analiza la información de su ELF para entrever cómo se ha construido dicho proceso a través de la información del ELF, por ejemplo, las direcciones y permisos de las regiones de texto y datos, etc.

Bibliografía

- [You95] Eric Youngdale, "The ELF Object File Format by Dissection", *Linux Journal*, 1 de mayo de 1995, disponible en <http://www.linuxjournal.com/article/1060>.
- [Bat12] BatchDrake, "Infección de ejecutables en Linux: ELF y código PIC (1/6)", *Security by Default*, Abril 2012, disponible en <http://www.securitybydefault.com/2012/04/infeccion-de-ejecutables-en-linux-elf-y.html>
- [O'Neill16] Ryan "elfmaster" O'Neil, *Learning Linux Binary Analysis*, 2016, Packt Publishing.
- [MCA08] C. Malinf, E. Casey y J. Aquilina, "Chapter 8: File Identification and Profiling: Initial Analysis of a Suspect File on a Linux System", en *Malware Forensics: Investigating and Analyzing Malicious Code*, Syngress, 2008.
- [CMCDragonkai] "Understanding the Memory Layout of Linux Executables", accedido 21/11/2017, disponible <https://gist.github.com/CMCDragonkai/10ab53654b2aa6ce55c11cfc5b2432a4>