# Trabajo Práctico 3 - Memoria

## Práctica con Linux

### La estructura de ELF

¿Qué es ELF? ELF (Executable and Linking Format) es un formato de archivo que define cómo se compone y organiza un archivo objeto. Con esta información, su kernel y el cargador binario saben cómo cargar el archivo, dónde buscar el código, dónde buscar los datos inicializados, qué biblioteca compartida necesita ser cargada, etc. Los diferentes tipos de objetos ELF son:
Archivo reubicable (relocatable file): un archivo objeto que contiene código y datos adecuados para enlazar con otros archivos objeto para crear un archivo objeto ejecutable o compartido. En otras palabras, se puede decir que el archivo reubicable es una base para crear ejecutables y bibliotecas. Este es el tipo de archivo que obtiene si compila un código fuente como este:

```
$ gcc -c hello.c
```

Esto producirá hello.o, que es un archivo reubicable. El módulo del núcleo (con sufijo .o o .ko) es también una forma de archivo reubicable.
Archivo ejecutable: archivo objeto que contiene un programa apto para su ejecución. Sí, eso significa que su reproductor de mp3 XMMS, su reproductor de software vcd e incluso su editor de texto son todos archivos ejecutables ELF. Este también es un archivo familiar si compila un programa:

```
$ gcc -o hello hello.c
```

Después de asegurarse de que el bit ejecutable de «hello» está habilitado, puede ejecutarlo. La pregunta es, ¿qué hay del shell script? Un Script de shell no es ejecutable ELF, pero el intérprete sí lo es.

**Archivo de objeto compartido** (shared object file): Este archivo contiene código y datos adecuados para enlazar en dos contextos:

1. El editor de enlaces (link editor) puede procesarlo con otro archivo de objeto reubicable y compartido para crear otro archivo de objeto.
2. El enlazador dinámico (dynamic linker) lo combina con un archivo ejecutable y otros objetos compartidos para crear una imagen de proceso.

En pocas palabras, estos son los archivos que normalmente se ven con el sufijo .so (por lo general ubicados dentro de /usr/lib en la mayoría de las instalaciones de Linux).
¿Hay otra manera de detectar el tipo ELF? Si, la hay. En todo objeto ELF hay un archivo de cabecera que explica qué tipo de archivo es. La forma más sencilla y genérica de detectar qué tipo de archivo es el que tenemos es mediante el comando «file»; por ejemplo operando sobre nuestro

«hola, mundo» de esta manera:

```
ubuntu@ubuntu:~$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
dynamically linked, interpreter
/lib/ld-linux.so.2, for GNU/Linux 3.2.0,
BuildID[sha1]=5ce1e11509ddd875dd800a7251fc46f5050feff6, not stripped
ubuntu@ubuntu:~$
```

**Preprocesamiento:** Lo primero que hace el compilador es preprocesar el archivo fuente, esto es, interpretar todas las directivas de preprocesamiento que hayamos utilizado, como #define, #include, #ifdef, etc... y además, eliminará todos los comentarios que hayamos escrito en el archivo.
En el caso particular de nuestro «hello.c», incluirá el archivo stdio.h (cabecera de entrada/salida estándar), y eliminará los comentarios.
Preprocesemos nuestro ejemplo:

```
ubuntu@ubuntu:~$ gcc -E hello.c -o hello.i
```

El modificador «-E» permite especificar al compilador gcc que sólo preprocese, y que la salida sea escrita en el archivo «hello.i». La extensión .i es generalmente utilizada para archivos preprocesados.

Ahora, «hello.i» sigue siendo código fuente, pero si vemos su contenido encontraremos algo similar a esto:

```
ubuntu@ubuntu:~$ cat hello.i
...
extern int fileno_unlocked (FILE *__stream) __attribute__ ((__nothrow__
, __leaf__)) ; # 872 "/usr/include/stdio.h" 3 4
extern FILE *popen (const char *__command, const char *__modes) ;
extern int pclose (FILE *__stream);
extern char *ctermid (char *__s) __attribute__ ((__nothrow__ ,
__leaf__));
# 912 "/usr/include/stdio.h" 3 4
extern void flockfile (FILE *__stream) __attribute__ ((__nothrow__ ,
__leaf__)); extern int ftrylockfile (FILE *__stream) __attribute__
((__nothrow__ , __leaf__)) ; extern void funlockfile (FILE *__stream)
__attribute__ ((__nothrow__ , __leaf__)); # 942 "/usr/include/stdio.h"
3 4
# 2 "hello.c" 2
# 3 "hello.c"
int main()
{
printf("hello, world\n");
return 0;
}
```

Para ver el archivo en detalle es preferible utilizar el comando «less» en vez de «cat» porque, como habrá notado, la directiva «#include <stdio.h>» incluye todas esas líneas que vemos antes de nuestra declaración «int main()».

**Compilación:** El siguiente paso es compilar nuestro código. El resultado de la compilación es un código binario no ejecutable, llamado código objeto, cuya extensión característica es un archivo «.o».

```
ubuntu@ubuntu:~$ gcc -c hello.i -o hello.o
```

Y si vemos el tipo de archivo, éste será un archivo binario ELF, pero no ejecutable, como sí lo era el anterior.

```
ubuntu@ubuntu:~$ file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not
stripped
ubuntu@ubuntu:~$
```

**Bibliotecas y enlace:** El siguiente paso para lograr que este código objeto se vuelva ejecutable, es el de enlazar (o «linkear», en la jerga) el objeto con las bibliotecas del sistema, las que utiliza.
En este caso todas las funciones incluidas en la cabecera «stdio.h» pertenecen a la biblioteca estándar de C, por lo que solamente deberemos extraer de la misma las funciones que queremos enlazar con nuestro objeto, y luego enlazarlo.
Pero podemos crear nuestro archivo de biblioteca de la siguiente forma:

```
ubuntu@ubuntu:~$ ar -cvr libhello.a hello.o
a - hello.o
ubuntu@ubuntu:~$ file libhello.a
libhello.a: current ar archive
```

El archivo «libhello.a» contendrá las funciones necesarias que debemos enlazar con nuestro «hello.o» para poder crear un ejecutable.
Ahora, solo bastará enlazar nuestro objeto con dicho archivo de biblioteca. Por cierto, es un archivo «.a», que viene del inglés «archive», y es una biblioteca de enlace estático, en contraposición con las bibliotecas de enlace dinámico, que en sistemas GNU/Linux se llaman «.so» de «shared object».
Obviamente éste es sólo un ejemplo trivial ya que se trata de un «hola, mundo». Esto lo vemos listando el contenido de «libhello.a»:

```
ubuntu@ubuntu:~$ ar -t libhello.a
hello.o
```

Si quisiéramos utilizar esta biblioteca estática que hemos creado deberíamos indicarle al compilador gcc la ruta al archivo «.a» mediante la opción «-L» y el nombre de la biblioteca con la opción «-l»:

```
ubuntu@ubuntu:~$ gcc hello.c -L/home/ubuntu/ -lhello -o hello
```

Si tuviéramos una aplicación grande, es buena idea crear bibliotecas para grupos de funciones que pueden ser utilizadas en varios sistemas, y en el resto de los sistemas podríamos simplemente incluirlas y utilizarlas, sin necesidad de que sean parte del código de nuestra aplicación.

La ventaja que ofrece esto es que podemos actualizar la biblioteca individualmente, y los cambios se verán reflejados en el resto de las aplicaciones que la utilicen inmediatamente luego de la re-compilación de las mismas (porque es un enlace estático), sin necesidad de implementar los cambios en cada una de las aplicaciones. Si hubiéramos creado una biblioteca de enlace dinámico «.so» no sería necesaria la re-compilación.
Hemos podido ver también la diferencia entre un archivo de cabecera y una biblioteca. Un archivo de cabecera, o header, es un archivo de texto con prototipos de funciones, definiciones de tipos de datos, macros #define, etc. Un archivo de biblioteca es binario compilado, un archivo «.a» o «.so», que sirve para enlazar con una aplicación final, pero que no necesariamente hemos escrito nosotros.

**readelf:** Suponiendo que tenemos instalado el paquete «binutils» (o «elfutils», según la distribución) podemos usar «readelf» para leer esta cabecera. Por ejemplo:

```
ubuntu@ubuntu:\~$ readelf -h /bin/ls
...
Type: EXEC (Executable file)
...
ubuntu@ubuntu:\~$ readelf -h /usr/lib/i386-linux-gnu/crt1.o
...
Type: REL (Relocatable file)
...
ubuntu@ubuntu:\~$ readelf -h /lib/i386-linux-gnu/libc-2.23.so
...
Type: DYN (Shared object file)
...
ubuntu@ubuntu:\~$
```

Para facilitarnos el estudio de ELF, podemos utilizar el siguiente programa simple C, modificando el que tenemos:

```c
/* test-elf.c */
#include <stdio.h>
int global_data = 4;
int global_data_2;
int main(int argc, char **argv){
  int local_data = 3;
  printf("hello, world\n");
  printf("global_data = %d\n", global_data);
  printf("global_data_2 = %d\n", global_data_2);
  printf("local_data = %d\n", local_data);
  return (0);
}
```

Y compilarlo así:

```
ubuntu@ubuntu:\~$ gcc -o test-elf test-elf.c
```

**Examen de la cabecera ELF**: El binario producido será nuestro objetivo de examen. Comencemos con el contenido de la cabecera ELF:

```
ubuntu@ubuntu:\~$ readelf -h test-elf
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: 0x80482c0
Start of program headers: 52 (bytes into file)
Start of section headers: 2060 (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: 28
Section header string table index: 25
```

**¿Qué nos dice este encabezado?**
Este ejecutable ha sido creado para la arquitectura Intel x86 de 32 bits (campos «Machine» y «Class»).
Cuando se ejecuta, el programa comenzará a ejecutarse desde la dirección virtual 0x8048340 (ver «Entry point address»). El prefijo «0x» aquí significa que es un número hexadecimal. Esta dirección no apunta a nuestro procedimiento main(), sino a un procedimiento llamado _start. Como sabemos, no hemos creado un procedimiento así llamado; el procedimiento _start es creado por el enlazador (linker) cuyo propósito es inicializar el programa.
Este programa tiene un total de 28 secciones (Number of section headers) y 7 segmentos (Number of program headers).
**¿Qué es la sección?** La sección es un área en el archivo objeto que contiene información que es útil para enlazar: código del programa, datos del programa (variables, matrices, cadenas), información de reubicación y otros. Por lo tanto, en cada área, se agrupan varias informaciones que tienen un significado distinto: la sección de código sólo contiene código, la sección de datos sólo contiene datos inicializados o no inicializados, etc. La tabla de cabecera de sección (SHT, Section Header Table) nos dice exactamente qué secciones tiene el objeto ELF pero al menos mirando en el campo «Number of section headers» (Número de cabeceras de sección) de arriba, se puede decir que

«test-elf» contiene 28 secciones.
Si la sección tiene significado para el binario, nuestro núcleo de Linux no lo ve de la misma manera. El kernel de Linux prepara varios VMA (Virtual Memory Area, área de memoria virtual) que contienen marcos de página virtualmente contiguos, como vemos en la Figura 3.1. Dentro de estos VMA, una o más secciones están mapeadas. Cada VMA en este caso representa un segmento ELF. ¿Cómo sabe el núcleo qué sección va a qué segmento? Esta es la función de la tabla de cabecera de programa (PHT, Program Header Table).

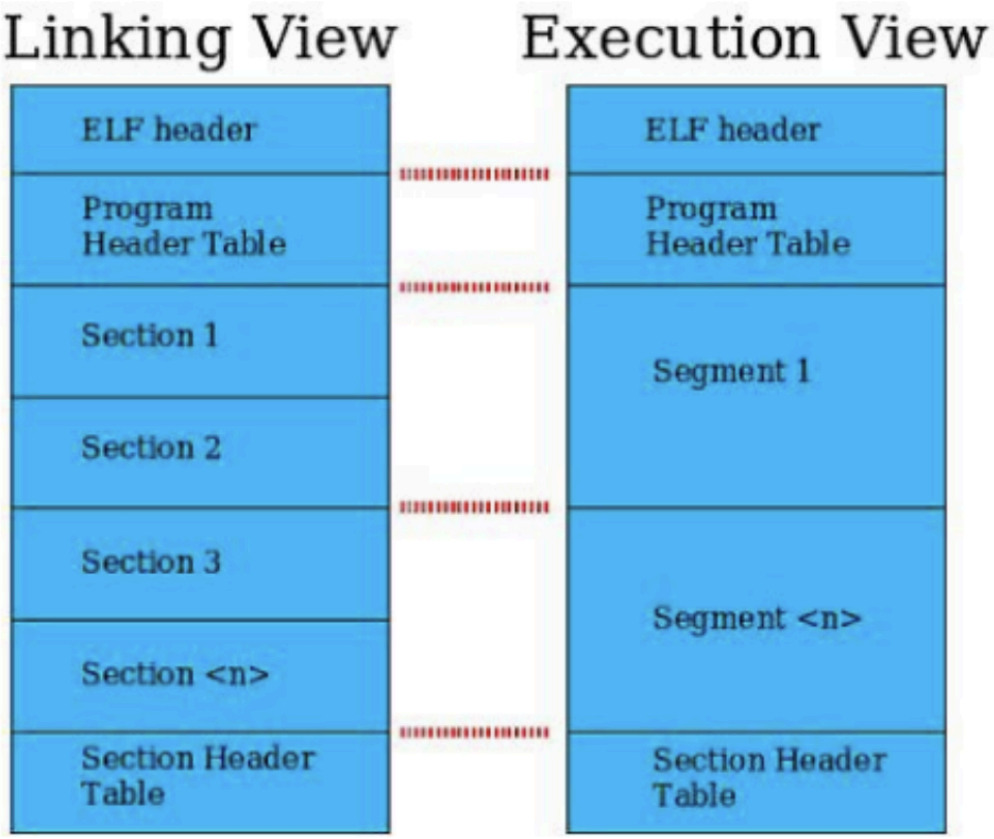


**Figura 3.1.:** Estructura ELF en dos puntos de vista diferentes

**Examen de la tabla de cabecera de sección (SHT)**: Veamos qué tipo de secciones existen dentro de nuestro programa:

```
ubuntu@ubuntu:~$ readelf -S test-elf
```

Hay 28 cabeceras de sección, comenzando en offset 0x80c:
Section Headers:

```
[Nr] Name Type Addr Off Size ES Flg Lk Inf Al ........
[ 4] .dynsym DYNSYM 08048174 000174 000060 10 A 5 1 4
```

```
........
[11] .plt PROGBITS 08048290 000290 000030 04 AX 0 0 4
[12] .text PROGBITS 080482c0 0002c0 0001d0 00 AX 0 0 4
........
[20] .got PROGBITS 080495d8 0005d8 000004 04 WA 0 0 4
[21] .got.plt PROGBITS 080495dc 0005dc 000014 04 WA 0 0 4
........
[22] .data PROGBITS 080495f0 0005f0 000010 00 WA 0 0 4
[23] .bss NOBITS 08049600 000600 000008 00 WA 0 0 4
........
[26] .symtab SYMTAB 00000000 000c6c 000480 10 27 2c 4
........
```

*(Las líneas se han numerado para referenciarlas)*

La sección .text es un lugar donde el compilador pone el código de los ejecutables. Como consecuencia, esta sección se marca como ejecutable («X» en el campo Flg). En esta sección, verá los códigos de máquina de nuestro procedimiento main():

```
ubuntu@ubuntu:~$ objdump -d -j .text test-elf
```

«-d» le dice a objdump que desensamble el código de máquina mientras que «-j» le dice que se enfoque sólo en una sección específica (en este caso, en la sección .text).
08048370 :

```
.......
8048397: 83 ec 08 sub $0x8, %esp
804839a: ff 35 fc 95 04 08 pushl 0x80495fc
80483a0: 68 c1 84 04 08 push $0x80484c1
80483a5: e8 06 ff ff ff call 80482b0
80483aa: 83 c4 10 add $0x10, %esp
80483ad: 83 ec 08 sub $0x8, %esp
80483b0: ff 35 04 96 04 08 pushl 0x8049604
80483b6: 68 d3 84 04 08 push $0x80484d3
80483bb: e8 f0 fe ff ff call 80482b0
.......
```

La sección .data contiene todas las variables inicializadas dentro del programa que no están dentro de la pila. «Inicializada» aquí significa que se le da un valor inicial como lo hicimos en «global_data». Si bien la variable «local_data» fue inicializada, su valor no está en .data ya que vive en la pila del proceso.
Aquí está lo que objdump encontró sobre la sección .data:

```
ubuntu@ubuntu:~$ objdump -d -j .data test-elf
.....
080495fc :
80495fc: 04 00 00 00 ....
.....
```

Una cosa que podemos concluir hasta ahora es que objdump amablemente hace la transformación de dirección a símbolo para nosotros. Sin mirar en la tabla de símbolos, sabemos que 0x08049424 es la dirección de global_data. Allí, vemos claramente que está inicializado con 4. Tenga en cuenta que los ejecutables comunes instalados por la mayoría de las distribuciones de Linux se han recortado, por lo que no hay ninguna entrada en su tabla de símbolos. Esto le dificulta a objdump la interpretación de las direcciones.
¿Y qué es .bss? BSS (Block Started by Symbol, bloque iniciado por el símbolo) es una sección donde se asignan todas las variables no inicializadas. Se podría pensar que «todo tiene un valor inicial». Cierto, en el caso de Linux, todas las variables sin inicializar se ponen a cero, por eso la sección .bss es sólo un montón de ceros. Para las variables de tipo carácter, eso significa carácter nulo. Conociendo este hecho, sabemos que a global_data_2 se le asigna 0 en tiempo de ejecución:

```
ubuntu@ubuntu:~$ objdump -d -j .bss test-elf
Disassembly of section .bss:
.....
08049604 :
8049604: 00 00 00 00 ....
.....
```

Anteriormente, mencionamos un poco sobre la tabla de símbolos. Esta tabla es útil para encontrar la correlación entre un nombre de símbolo (función no externa, variable) y una dirección. Usando «-s», readelf decodificará la tabla de símbolos:

```
ubuntu@ubuntu:~$ readelf -s ./test-elf
Symbol table '.dynsym' contains 6 entries:
Num: Value Size Type Bind Vis Ndx Name
.....
2: 00000000 57 FUNC GLOBAL DEFAULT UND printf@GLIBC_2.0 (2) .....
Symbol table '.symtab' contains 72 entries:
Num: Value Size Type Bind Vis Ndx Name
.....
49: 80495fc 4 OBJECT GLOBAL DEFAULT 22 global_data
.....
```

```
55: 08048370 109 FUNC GLOBAL DEFAULT 12 main
.....
59: 00000000 57 FUNC GLOBAL DEFAULT UND printf@@GLIBC_2.0 .....
61: 08049604 4 OBJECT GLOBAL DEFAULT 23 global_data_2
.....
```

«Value» indica la dirección del símbolo. Por ejemplo, si una instrucción se refiere a esta dirección (por ejemplo: pushl 0x80495fc), eso significa que se refiere a global_data. La función printf() se trata de forma diferente, ya que es un símbolo que hace referencia a una función externa. Recuerde que printf está definido en glibc, no dentro de nuestro programa. Más adelante, explicaremos cómo nuestro programa llama a printf.

**Examen de la tabla de cabecera del programa (PHT, Program Header Table)** Como se explicó anteriormente, el segmento es la forma en que el sistema operativo «ve» nuestro programa. Así, veamos cómo se segmentará nuestro programa:

```
ubuntu@ubuntu:~$ readelf -l test-elf
.....
There are 7 program headers, starting at offset 52
Program Headers:
Type Offset VirtAddr PhysAddr FileSiz MemSiz Flg Align [00] PHDR
0x000034 0x08048034 0x08048034 0x000e0 0x000e0 R E 0x4 [01] INTERP
0x000114 0x08048114 0x08048114 0x00013 0x00013 R 0x1 [02] LOAD 0x000000
0x08048000 0x08048000 0x004fc 0x004fc R E 0x1000 [03] LOAD 0x0004fc
0x080494fc 0x080494fc 0x00104 0x0010c RW 0x1000 [04] DYNAMIC 0x000510
0x08049510 0x08049510 0x000c8 0x000c8 RW 0x4 [05] NOTE 0x000128
0x08048128 0x08048128 0x00020 0x00020 R 0x4 [06] STACK 0x000000
0x00000000 0x00000000 0x00000 0x00000 RW 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 .eh_frame

03 .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04 .dynamic
05 .note.ABI-tag
06
```

El mapeo es bastante sencillo. Por ejemplo, dentro del segmento número 02, hay 15 secciones mapeadas. La sección .text está mapeada en este segmento. Sus banderas son R y E, lo que significa que es legible (Readable) y ejecutable (Executable). Si ve W en la bandera del segmento, significa que se puede escribir (Writable).
Mirando la columna «VirtAddr», podemos descubrir la dirección de inicio virtual de cada segmento. Volviendo al segmento número 2, la dirección de inicio es 0x08048000. Más adelante en esta sección, descubriremos que esta dirección no es la dirección real del segmento en memoria. Puede ignorar el PhysAddr, porque en Linux siempre operan en modo protegido (en Intel/AMD 32 bits y 64 bits) por lo tanto la dirección virtual es lo que importa.
El segmento tiene muchos tipos, pero vamos a centrarnos en dos tipos:
**LOAD**: El contenido del segmento se carga desde el archivo ejecutable. «Offset» denota el offset o desplazamiento del archivo donde el kernel debería comenzar a leer el contenido del archivo. «FileSiz» nos dice cuántos bytes se deben leer del archivo.
Por ejemplo, el segmento número 2 es en realidad el contenido del archivo que comienza en offset 0 a 4fc (offset+filesiz). Para acelerar la ejecución, el contenido del archivo se lee bajo demanda, por lo que sólo se lee desde el disco si se hace referencia a él en tiempo de ejecución.
**STACK**: El segmento es área de pila. Es interesante ver que todos los campos excepto «Flg» y «Align» tienen un 0. ¿Es un error? No, es válido. Es tarea del núcleo decidir desde dónde empieza el segmento de pila y qué tan grande es. Recuerde que en los procesadores compatibles con Intel, la pila crece hacia abajo (la dirección disminuye cada vez que se «empuja» (push) un valor).
¿Desea ver la disposición real del segmento de proceso? Podemos usar el archivo /proc/<pid>/maps para revelarlo. <pid> es el PID del proceso que queremos observar. Antes de seguir adelante, tenemos un pequeño problema aquí. Nuestro programa de prueba se ejecuta tan rápido que termina antes de que podamos volcar la entrada /proc relacionada. Utilizamos gdb (GNUDebugger) para resolver esto. Puede usar otro truco cómo insertar sleep() antes de que llame return().

```
ubuntu@ubuntu:~$ gdb test-elf
(gdb) b main
Breakpoint 1 at 0x8048376
(gdb) r
Breakpoint 1, 0x08048376 in main ()
(gdb)
```

Mantenga esta sesión, abra otra consola y descubra el PID del programa «test-elf». Si quiere el camino rápido, escriba:

```
ubuntu@ubuntu:~$ cat /proc/$(pgrep test)/maps
...
[1] 0039d000-003b2000 r-xp 00000000 16:41 1080084 /lib/ld-2.3.3.so [2]
003b2000-003b3000 r--p 00014000 16:41 1080084 /lib/ld-2.3.3.so [3]
003b3000-003b4000 rw-p 00015000 16:41 1080084 /lib/ld-2.3.3.so [4]
003b6000-004cb000 r-xp 00000000 16:41 1080085 /lib/tls/libc-2.3.3.so
[5] 004cb000-004cd000 r--p 00115000 16:41 1080085
/lib/tls/libc-2.3.3.so [6] 004cd000-004cf000 rw-p 00117000 16:41
```

```
1080085 /lib/tls/libc-2.3.3.so [7] 004cf000-004d1000 rw-p 004cf000
00:00 0
[8] 08048000-08049000 r-xp 00000000 16:06 66970 /home/ubuntu/test-elf
[9] 08049000-0804a000 rw-p 00000000 16:06 66970 /home/ubuntu/test-elf
[10] b7fec000-b7fed000 rw-p b7fec000 00:00 0
[11] bffeb000-c0000000 rw-p bffeb000 00:00 0
[12] ffffe000-fffff000 ---p 00000000 00:00 0
...
```

*La salida que verá es similar a ésta. Las líneas han sido numeradas para su referencia.*

El comando «pgrep» busca procesos basados en su nombre y otros atributos.

Así que, en total, vemos 12 segmentos o VMA. Enfóquese en el primer y último campo. El primer campo denota el rango de direcciones VMA, mientras que el último campo muestra el archivo utilizado. ¿Ve la similitud entre la VMA #8 y el segmento #02 listado en PHT? La diferencia es que PHT dice que termina en 0x080484fc, pero en VMA #8, vemos que termina en 0x08049000. Lo mismo sucede entre el VMA #9 y el segmento #03; PHT dijo que comienza en 0x080494fc, mientras que el VMA comienza en 0x0804900.

Hay varios factores que debemos observar:

1. A pesar de que la VMA comenzó en una dirección diferente, las secciones relacionadas todavía están mapeadas en la dirección virtual exacta.
2. El núcleo asigna memoria por página y el tamaño de la página es de 4KB. Así, cada dirección de página es en realidad un múltiplo de 4 KB, por ejemplo: 0x1000, 0x2000 y así sucesivamente. Así, para la primera página de VMA #9, la dirección de la página es 0x0804900. O técnicamente hablando, la dirección del segmento se redondea hacia abajo (alineada) al límite de página más cercano.

Por último, ¿cuál es la pila? Es la indicada en VMA #11. Normalmente, el núcleo asigna varias páginas dinámicamente y se mapea a la dirección virtual más alta posible en el espacio de usuario para formar el área de pila. Simplemente hablando, cada espacio de direcciones del proceso se divide en dos partes (esto asume un procesador de 32 bits compatible con Intel): espacio de usuario y espacio de núcleo. El espacio de usuario está en el rango 0x0000000000-0xc0000000, mientras que el espacio del núcleo comienza a partir de 0xC000000000.

Por lo tanto, está claro que a la pila se le asigna un rango de direcciones cerca del límite 0xc0000000. La dirección final es estática, mientras que la dirección de inicio cambia según cuántos valores se almacenan en la pila.

**¿Cómo se hace referencia a una función?**

Si un programa llama a una función que reside dentro de su propio ejecutable, todo lo que tiene que hacer es simple: simplemente llamar al procedimiento. Pero, ¿qué pasa si llama a algo como printf() que está definido dentro de la biblioteca («librería») compartida glibc?

No discutiremos profundamente sobre cómo funciona realmente el enlazador dinámico (dynamic linker), sino que nos enfocaremos en cómo se implementa el mecanismo de llamada dentro del propio ejecutable.

Cuando un programa quiere llamar a una función, lo hace siguiendo este flujo:

1. Hace un salto a la entrada relevante en la tabla de enlace de procedimiento PLT (Procedure Linkage Table).
2. En PLT, hay otro salto a una dirección mencionada en la entrada relacionada con la tabla de desplazamientos globales (GOT, Global Offset Table).
3. Si esta es la primera vez que se llama la función, se sigue el paso 4. Si no es así, sigue al paso 5.
4. La entrada GOT relacionada contiene una dirección que indica la siguiente instrucción en PLT. El programa saltará a esta dirección y luego llamará al enlazador dinámico para resolver la dirección de la función. Si se encuentra la función, su dirección se pone en la entrada GOT relacionada y, a continuación, se ejecuta la propia función.
   Por lo tanto, otra vez se llama a la función, GOT ya tiene su dirección y PLT puede saltar directamente a la dirección. Este procedimiento se denomina «unión perezosa» (lazy binding, en inglés); todos los símbolos externos no se resuelven hasta el momento en que realmente se necesiten (en este caso, cuando se llama a una función). Se salta al paso 6.
5. Ir a la dirección mencionada en GOT. Es la dirección de la función, por lo que PLT ya no se utiliza.
6. La ejecución de la función ha finalizado. Volver a la siguiente instrucción del programa principal.

Como siempre, mirar dentro del ejecutable es la mejor manera de explicarlo:

```
ubuntu@ubuntu:~$ objdump -d -j .text test-elf
```

Verá el siguiente fragmento de código:

```
.....
08048370 :
.....
804838f: e8 1c ff ff ff call 80482b0
Lo que tenemos en 0x80482b0 es:
080482b0 :
80482b0: ff 25 ec 95 04 08 jmp *0x80495ec
80482b6: 68 08 00 00 00 push $0x8
80482bb: e9 d0 ff ff ff jmp 8048290 <_init+0x18>
```

Como se puede ver, el salto en 0x80482b0 es un salto indirecto ('*' delante de la dirección). Por lo tanto, para ver dónde va a saltar, debemos echar un vistazo a 0x80482b0. Las conjeturas son, o esta dirección está en la sección .got o en .got.plt. Mirando hacia atrás en SHT, es claro que debemos revisar .got.plt. Usamos readelf para hacer volcado hexadecimal porque hace reordenación de números por nosotros:

```
ubuntu@ubuntu:~$ readelf -x 21 test-elf
Hex dump of section '.got.plt':
```

```
0x080495dc 080482a6 00000000 00000000 08049510 ................
0x080495ec 080482b6 ....
```

*(Nota: la primera columna es dirección virtual)*

Los datos de esta dirección se describen en la quinta columna, no en la segunda! Por lo tanto, de derecha a izquierda, la dirección está en orden ascendente.

¡Bingo! Tenemos «080482b6» aquí. En otras palabras, volvemos a PLT y ahí eventualmente saltamos a otra dirección. Aquí es donde se inicia el trabajo del enlazador dinámico, así que lo omitiremos. Asumiendo que el enlazador dinámico ha terminado su trabajo mágico, la entrada GOT relacionada ahora contiene la dirección de printf().

**Herramienta alternativa para inspeccionar la estructura de ELF** Además de contar con readelf y objdump, existe otra herramienta llamada Beye (originalmente Biew). Esto es en realidad un visor de archivos pero es capaz de analizar la estructura ELF. Puede obtener el programa fuente de http://beye.sourceforge.net/ y compilarlo usted mismo. Normalmente Beye (o Biew) está incluido en el Live CD de Linux orientado a hacking como Phlak https://sourceforge.net/projects/phlakproject/. Consulte el sitio web y los documentos empaquetados para obtener información sobre cómo compilar e instalar Beye.

**Ofrece una pantalla GUI basada en «curses»**. La navegación entre secciones, la comprobación del encabezado ELF, la lista de símbolos y otras tareas son ahora sólo cuestión de pulsar ciertos atajos de teclado y listo.
Por ejemplo, puede listar los símbolos y saltar directamente a la dirección del símbolo. Aquí, intentamos saltar a main(). Primero ejecute a Beye:

```
ubuntu@ubuntu:~$ beye test-elf
```

Presione Ctrl+A seguido de F7 para ver la tabla de símbolos. Para no perder tiempo en la tabla, pulse de nuevo F7 para abrir el menú «Buscar cadena». Escriba «main» y pulse Intro. Una vez que la entrada resaltada es lo que está buscando, simplemente presione Enter y Beye saltará a la dirección de main() como se ve en la Figura 3.2. No olvide cambiar al modo de desensamblador (presione F2 para seleccionarlo) para que pueda ver la interpretación de alto nivel de los «opcodes». Como normalmente nos referimos a la dirección virtual, no al desplazamiento de archivos, es mejor cambiar a la vista de dirección virtual. Pulse Ctrl+C seguido de F6 y seleccione «Local». Ahora, lo que ve en la columna de la izquierda es la dirección virtual.

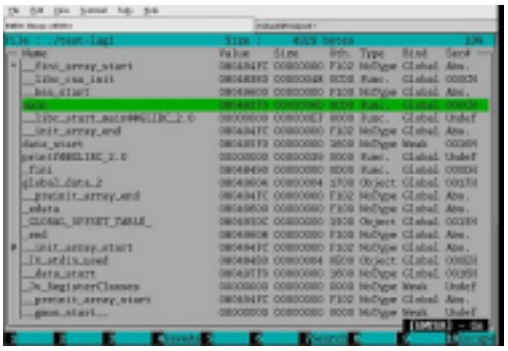

**Figura 3.2.:** Lista de todos los símbolos en Biew

## Estadísticas de la memoria virtual en Linux

El término «intercambio» (swap) se refiere estrictamente a guardar el proceso entero, no una parte, al disco. Esta forma de intercambio es rara de encontrar, sino que se usa en combinación con la paginación.
Un page-in es un hecho común, normal y no es motivo de preocupación. Cuando arranca una aplicación, su imagen se va cargando en memoria haciendo page-in, esto es normal. Pero el

page-out puede ser una señal de problemas. Cuando el núcleo detecta que se está quedando sin memoria intenta liberarla haciendo page-out. Si bien esto puede ocurrir brevemente y de tanto en tanto, si esta situación se vuelve constante podemos caer en hiperpaginación.
Tanaka (2005) nos muestra cómo el programa vmstat, tal como su nombre sugiere,muestra estadísticas de la memoria virtual. Cuánta memoria virtual hay, cuánta está libre y la actividad de paginación; y se pueden observar los page-in y los page-out a medida que ocurren.
Es mejor si nos muestra la tabla actualizándola periódicamente, el retardo o delay se lo podemos pasar como primer parámetro y como segundo, el conteo para que no nos lo muestre indefinidamente. Por ejemplo, si colocáramos el comando:

```
ubuntu@ubuntu:~$ vmstat 5 10
```

podríamos ver una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system--
-----cpu------ r b swpd free buff cache si so bi bo in cs us sy id wa
st 0 0 0 1183880 104564 545096 0 0 56 15 344 365 3 2 92 2 0 0 0 0
1179308 104572 549592 0 0 0 4 272 329 1 1 98 0 0 0 0 0 1183572 104580
545112 0 0 0 7 247 266 0 0 98 1 0 2 0 0 1183612 104592 545092 0 0 0 8
249 285 0 0 98 1 0 0 0 0 1183644 104592 545080 0 0 0 0 214 232 0 0 99 0
0 1 0 0 1183652 104592 545072 0 0 0 18 266 314 1 0 99 0 0 0 0 0 1183652
104600 545108 0 0 0 14 208 265 1 0 98 1 0 0 0 0 1183768 104600 545064 0
0 0 1 212 276 1 0 99 0 0
0 0 0 1183644 104604 545072 0 0 0 1 190 256 1 0 98 1 0 0 1 0 1173544
104612 553040 0 0 1589 29 539 501 3 2 90 5 0
```

En la que se nos muestran 10 líneas de estadísticas tomadas cada 5 segundos. Las columnas están explicadas en el manual (man vmstat) pero las que nos interesan son free, si y so. La columna free nos muestra la cantidad de memoria libre, si nos muestra los page-in y so los page-out.
Es decir que nos muestra que hay suficiente memoria libre. Como en este momento no estamos comenzando a ejecutar una aplicación nueva, no vemos page-in; luego tampoco vemos page-outs. Pero si comenzáramos a ejecutar varias aplicaciones que requirieran de mucha memoria, y colocáramos nuevamente el mismo comando, la situación cambiaría y nos mostraría una salida similar a ésta:

```
procs -----------memory---------- ---swap-- -----io---- -system--
-----cpu------ r b swpd free buff cache si so bi bo in cs us sy id wa
st 1 0 13344 1444 1308 19692 0 168 129 42 1505 713 20 11 69
1 0 13856 1640 1308 18524 64 516 379 129 4341 646 24 34 42
3 0 13856 1084 1308 18316 56 64 14 0 320 1022 84 9 8
```

Note los valores no nulos de so, están indicando que no hay suficiente memoria física y el núcleo está generando page-outs. Como ya vimos anteriormente, podemos usar top y ps para ver cuáles son

los procesos que más recursos están consumiendo. Pero también podemos usar el mismo programa vmstat para ver otras estadísticas interesantes. Instamos al lector a que experimente libremente una vez que haya leído las páginas del manual.
En Linux contamos con los comandos administrativos swapon y swapoff para activar y desactivar respectivamente dispositivos o archivos definidos como áreas de intercambio.
En el pseudo sistema de archivos accesible en el directorio /proc tenemos al archivo /proc/swaps que mide el espacio de intercambio y su utilización. Por ejemplo, en un sistema con una única partición de intercambio, si colocamos este comando:

```
ubuntu@ubuntu:~$ cat /proc/swaps
```

la salida será similar a ésta:

```
Filename Type Size Used Priority
/dev/sda5 partition 2104472 0 -1
```

En la que podemos ver el nombre del archivo de intercambio, el tipo de espacio de intercambio, el tamaño total y la cantidad de espacio en uso, medidos en kilobytes. La columna de Prioridad es útil cuando se usan varios archivos o particiones de intercambio; a menor número mayor es la preferencia de uso.

### El archivo /proc/pid/maps

El sistema operativo Linux ofrece un tipo de sistema de archivos muy especial: el sistema de archivos proc. Este sistema de archivos no tiene soporte en ningún dispositivo. Su objetivo es poner a disposición del usuario datos del estado del sistema en forma de archivos. Esta idea no es original de Linux, ya que casi todos los sistemas UNIX la incluyen. Sin embargo, Linux se caracteriza por ofrecer más información del sistema que el resto de variedades de UNIX. En este sistema de archivos se puede acceder a información general sobre características y estadísticas del sistema, así como a información sobre los distintos procesos existentes. La información relacionada con un determinado proceso se encuentra en un directorio que tiene como nombre el propio identificador del proceso (pid). Así, si se pretende obtener información de un proceso que tiene un identificador igual a “1234”, habrá que acceder a los archivos almacenados en el directorio “/proc/1234/”. Para facilitar el acceso de un proceso a su propia información, existe, además, un directorio especial, denominado “self”. Realmente, se trata de un enlace simbólico al directorio correspondiente a dicho proceso. Así, por ejemplo, si el proceso con identificador igual a “2345” accede al directorio “/proc/self/”, está accediendo realmente al directorio “/proc/2345/”.
En el directorio correspondiente a un determinado proceso existe numerosa información sobre el mismo. Sin embargo, en esta práctica nos vamos a centrar en el archivo que contiene información sobre el mapa de memoria de un proceso: el archivo “maps”. Cuando se lee este archivo, se obtiene una descripción detallada del mapa de memoria del proceso en ese instante. Como ejemplo, se incluye a continuación el contenido de este archivo para un proceso que ejecuta el programa “cat”.

Coloque el comando:

```
ubuntu@ubuntu:~$ cat /proc/self/maps
```

Y verá una salida similar a ésta:

```
08048000-0804a000 r-xp 00000000 08:01 65455 /bin/cat
0804a000-0804c000 rw-p 00001000 08:01 65455 /bin/cat
0804c000-0804e000 rwxp 00000000 00:00 0
40000000-40013000 r-xp 00000000 08:01 163581 /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 08:01 163581 /lib/ld-2.2.5.so
40022000-40135000 r-xp 00000000 08:01 165143 /lib/libc-2.2.5.so
40135000-4013b000 rw-p 00113000 08:01 165143 /lib/libc-2.2.5.so
4013b000-4013f000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Cada línea del archivo describe una región del mapa de memoria del proceso. Por cada región aparece la siguiente información:
Rango de direcciones virtuales de la región (en la primera línea, por ejemplo, de la dirección «08048000» hasta «0804a000»).
Protección de la región: típicos bits «r» (permiso de lectura), «w» (permiso de escritura) y «x» (permiso de ejecución).
Tipo de compartimiento: «p» (privada) o «s» (compartida). Hay que resaltar que en el ejemplo todas las regiones son privadas.

**Desplazamiento de la proyección en el archivo**. Por ejemplo, en la segunda línea aparece «00001000» (4096 en decimal), lo que indica que la primera página de esta región se corresponde con el segundo bloque del archivo (o sea, el byte 4096 del mismo).
Los siguientes campos identifican de forma única al soporte de la región. En el caso de que sea una región con soporte, se especifica el dispositivo que contiene el archivo (en el ejemplo, «08:01») y su nodo-i (para el comando «cat, 65455»), así como el nombre absoluto del archivo. Si se trata de una región sin soporte, todos estos campos están a cero.
A partir de la información incluida en este ejemplo, se puede deducir a qué corresponde cada una de las nueve regiones presentes en el ejemplo de mapa de proceso:

- Código del programa. En este caso, el comando estándar «cat».
- Datos con valor inicial del programa, puesto que están vinculados con el archivo ejecutable.
- Datos sin valor inicial del programa, puesto que se trata de una región anónima que está contigua con la anterior.
- Código de la biblioteca «ld», encargada de realizar todo el tratamiento requerido por las bibliotecas dinámicas que use el programa.
- Datos con valor inicial de la biblioteca «ld».
- Código de la biblioteca dinámica «libc», que es la biblioteca estándar de C usada por la

mayoría de los programas.
- Datos con valor inicial de la biblioteca dinámica «libc».
- Datos sin valor inicial de la biblioteca dinámica «libc».
- Pila del proceso.

Habiendo llegado a este punto, experimente libremente, investigando el mapa de memoria de otros procesos. Por ejemplo, si se cambia al directorio

```
ubuntu@ubuntu:~$ cd /proc
```

puede ver que hay un directorio por cada PID. Experimente investigando qué es lo que contiene el mapa de memoria del proceso cuyo PID es «1», luego con el de su propio intérprete de comandos «bash». Puede dejar ejecutando en una consola el proceso «top» y desde otra ver el mapa de memoria. Tome notas.

### El programa mapa.c

El siguiente programa ilustra los segmentos

```c
#include <stdio.h>
#include <stdlib.h>

int uig;
int ig=5;
int func(){
  return 0;
}
int main(){
  int local;
  int *ptr;
  ptr=(int *) malloc(sizeof(int));
  printf("Una direccion del BSS: %p\n", &uig);
  printf("Una direccion del segmento de datos: %p\n", &ig);
  printf("Una direccion del segmento de codigo: %p\n", &func);
  printf("Una direccion del segmento de pila: %p\n", &local);
  printf("Una direccion del monticulo: %p \n", ptr);
  printf("Otra direccion de la pila: %p\n", &ptr);
  free(ptr);
  return 0;
}
```