Introducción al Cracking en Linux 08 – ELF(II).

Programa:	**Crackme cm1** Formaba parte de los crackmes de Damm Vulnerable Linux: http://www.4shared.com/file/9CxyCEoJ/Crackmes_DVLtar.html
Descripción:	Programa cuya protección antidebugger es el formato elf corrupto.
Dificultad:	Media.
Herramientas:	Herramientas del sistema, Python y GDB.
Objetivos:	En este tutorial vamos a profundizar en la vinculación dinámica y su estructura en el formato ELF.

Cracker: [Juan Jose]	Fecha: 20/12/2011

Introducción:

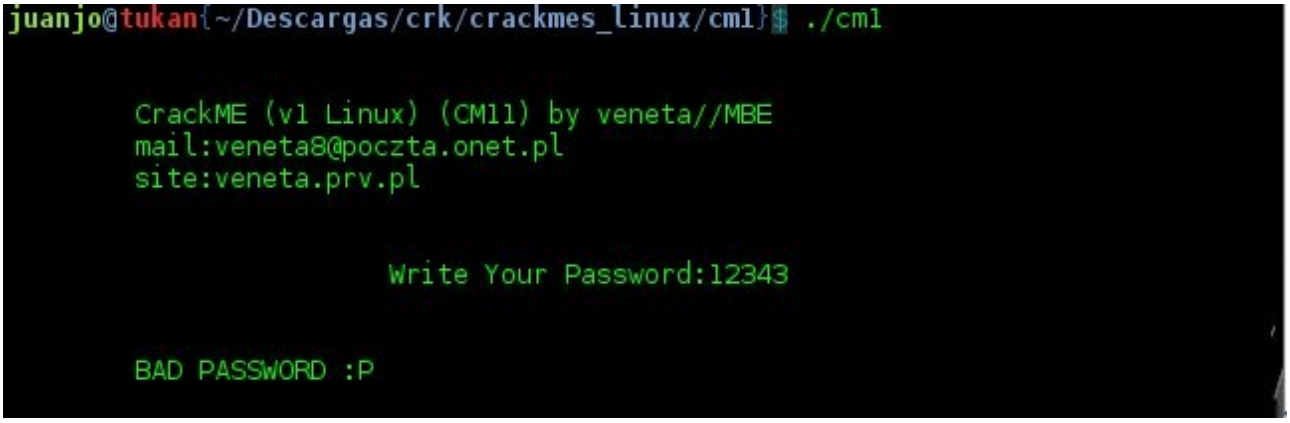
Como vimos en el tutorial nº6 de esta serie, al cargar un ejecutable ELF dinámico en memoria, es el enlazador dinámico el encargado de unir todas las llamadas a funciones externas con su librería correspondiente y después dar el control al programa. En la mayoría de los casos, toda la información necesaria para la vinculación dinámica se encuentra en varias secciones del ejecutable, desde una sección llamada **.interp** donde esta el nombre del enlazador dinámico, hasta otras como **.hash, .dynsym**, **.dynstr**, **.rel.dyn** y **.rel.plt**. Pero como ya sabemos, el formato ELF es muy flexible y justamente en este caso nos vamos a enfrentar con un crackme, **cm1**, definido como dinámico pero **sin** secciones; lo cual me ha motivado para intentar descubrir sus secretos a lo largo de este tutorial.

Empezamos con un poco de teoría, los binarios **ELF** en **GNU/Linux** se llaman ejecutables **"impuros"** (al igual que la mayoría de ejecutable PE de Windows); pues requieren código externo para funcionar. Este código externo se localiza en librerías compartidas o dinámicas (archivos con extensión **.so** en Linux y **.dll** en Windows); estas librerías se cargan en memoria como **objetos dinámicos** y se comparte entre las distintas aplicaciones que lo necesiten.

En la compilación de un programa, en la última fase, **linkado** o **enlazado**, se introduce todo lo necesario para que estas funciones sean accesibles cuando el programa lo necesite, pero es el kernel de linux, mediante unos módulos llamados cargadores, el que se encarga de montar y enlazar dinámicamente en memoria esas librerías para iniciar la ejecución. Por tanto en memoria debemos tener, además del programa , el cargador dinámico y las librerías necesarias.

Al Atake:

Primero vamos a ejecutar el programa **cm1** para ver su funcionamiento:



Como veis esta realizado por **veneta//MBE**; primero nos pide un serial, y aunque tenia los dedos cruzados (:-D), parece que el 12343 no es un serial valido, lo cual me lo confirma el programa con "BAD PASSWORD :P".

Como siempre vamos a dividir el estudio de este crackme en tres pasos; información , desensamblado y depuración:

1.Información.

1.a) Ldd:muestra las bibliotecas compartidas requeridas por cada programa

Se utiliza la orden "**ldd**" seguida del nombre del programa para ver que archivos son necesarios para su ejecución.:

```
$ ldd cm1
linux-gate.so.1 =>  (0xf77a3000)
libc.so.6 => /lib32/libc.so.6 (0xf7640000)
/lib/ld-linux.so.2 (0xf77a4000)
```

En este caso el cargador dinámico (runtime linker) es **ld-linux.so.2**; un ejecutable "**puro**" que se carga en memoria junto al programa y es el encargado de comunicar las llamadas externas a las librerías necesarias.
Respecto a las librerías, este programa necesita de **libc.so.6** (/lib32/libc.so.6) para poder ejecutar sus funciones C externas y **linux-gate.so.1**; que yo desconocía, aunque ya me había topado varias veces con ella en gdb sin saberlo. Esta librería es la forma moderna de realizar las llamadas al sistema , se carga en memoria y será la responsable de ejecutar las **syscall** mediante la función **__kernel_vsyscall**. En este aspecto, es una función dinámica ya que cualquier ejecutable que necesite una llamada al sistema puede utilizar esa función.

Curiosamente, la libreria **linux-gate.so.1** no existe; es lo que se denomina zona **vdso** (ya lo vimos en el manejo de radare); **Virtual Dynamic Shared Object** (VDSO), un área de memoria dinámica compartida por todos los procesos, empleada para hacer llamadas rápidas al sistema a través de la instrucción **sysenter**, que como vemos esta incluida en **__kernel_vsyscall**:

```
0xf7fdf430 <__kernel_vsyscall+16>:  pop    ebp
0xf7fdf431 <__kernel_vsyscall+17>:  pop    edx
0xf7fdf432 <__kernel_vsyscall+18>:  pop    ecx
0xf7fdf433 <__kernel_vsyscall+19>:  ret
0xf7fdf434: add    BYTE PTR [esi],ch
0xf7fdf436: jae    0xf7fdf4a0
--------------------------------------------------------------------------------

Catchpoint 1 (call to syscall 'write'), 0xf7fdf430 in __kernel_vsyscall ()
```

Como se ve, esta zona aparece cuando ponemos un brealpoint especial de gdb llamado **catchpoint**; que responde a diferentes condiciones, en este caso la orden es "**catch syscall 4**", parara el programa cuando ocurra la llamada al sistema (syscall) número **4;** que como podemos ver en **unistd_32.h** (/usr/include/asm/unistd_32.h) corresponde a **write.** Si continuamos un buen rato con **"ni"** llegaremos a donde se ejecuta la llamada al sistema realmente:

```
0xf7fdf420 <__kernel_vsyscall>:     push   ecx
0xf7fdf421 <__kernel_vsyscall+1>:   push   edx
0xf7fdf422 <__kernel_vsyscall+2>:   push   ebp
0xf7fdf423 <__kernel_vsyscall+3>:   mov    ebp,esp
0xf7fdf425 <__kernel_vsyscall+5>:   sysenter
```

Al llegar a **sysenter** comprobamos que funciona igual que las llamadas al sistema con **int80**; podemos ver el número de la syscall en **EAX** y los demás valores consecutivamente en **EBX**, **ECX**, **EDX**....etc.

```
     eax:00000004 ebx:00000001  ecx:08048326  edx:00000076     eflags:00000246
     esi:08048326 edi:00000000  esp:FFFFD48C  ebp:FFFFD48C     eip:F7FDF425
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s Z a P c
```

Para saber más de este archivo podemos dumpearlo desde gdb de esta forma:

```
gdb> dump memory dump.linux-gate 0xf7fdf000 0xf7fe0000
```

Como realmente no se el tamaño del archivo he escogido 0x1000 bytes desde la image base que según la dirección es **0xf7fdf000**, y lo he llamado "**dump.linux-gate**". Ahora podemos analizar el archivo, tanto su desensamblado con **hte** o con **objdump** podemos ver que funciones exporta:

```
$ objdump -T dump.linux-gate

dump.linux-gate:     file format elf32-i386

DYNAMIC SYMBOL TABLE:
ffffe420 g    DF .text  00000014  LINUX_2.5   __kernel_vsyscall
00000000 g    DO *ABS*  00000000  LINUX_2.5   LINUX_2.5
ffffe410 g    DF .text  00000008  LINUX_2.5   __kernel_rt_sigreturn
ffffe400 g    DF .text  00000009  LINUX_2.5   __kernel_sigreturn
```

Objdump con la opción **-T** nos muestra las funciones dinámicas del programa; como podemos ver por el manual de objdump ("**man objdump**"); *" Muestra las entradas de la tabla de símbolos*

dinámicos del fichero. Esto sólo tiene sentido para objetos dinámicos, como ciertos tipos de bibliotecas compartidas. Esto es similar a la información proporcionada por el programa `nm' cuando se le da la opción -D (--dynamic). "

De esta manera, ya tenemos claro la función de esta zona de memoria, que me había encontrado durante la depuración y no sabía su origen. Curiosamente esta librería es idea del propio **Linus Torvalds** y tuvo algunas críticas al principio; esencialmente por seguridad, ya que en todas las ejecuciones de un programa que utilizaba esta librería, la dirección de mapeo de esta función se mantenía constante; lo que podía facilitar la acción de exploit y demás programas que busquen una dirección fija donde saltar. Actualmente al ser aleatorias las direcciones de memoria del proceso, han dificultado su utilización Podéis saber mas en esta web:

http://www.trilithium.com/johan/2005/08/linux-gate/

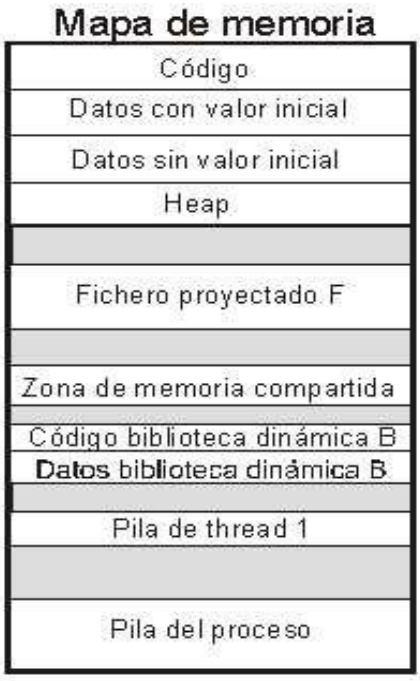
Incluso la entrada en **LKML.org** del propio **Linus Torvald** explicando su funcionamiento y que termina definiendose como "*I'm a disgusting pig, and proud of it to boot*." (Soy un cerdo asqueroso, y orgulloso de que se inicie.), jejeje, sin palabras:

http://lkml.org/lkml/2002/12/18/218

**********VASR**********

Esta variabilidad de la direcciónes de memoria donde se cargan las librerías de los programas dinámicos, es una protección que se ha implantado en el kernel **2.6** y en ejecutables realizados con **GCC**, para no dar facilidades a los exploit y demás malwares que podrían ayudarse de las direcciones fijas para su ejecución.

Se llama **Virtual Address Space Randomization**, es la creación de espacio de direcciones virtuales aleatorias en el proceso, que van cambiando con cada invocación del programa.
Además de las librerías externas, también cambia la dirección de la pila (stack), del cargador dinámico y de todas las memorias que pueden acompañar al proceso (como la zona VSDO):



El archivo de sistema que controla esta opción se encuentra en:

/proc/sys/kernel/randomize_va_space

Solo lo puede manipular el superusuario o root y se controla según sea el valor TRUE (1) o FALSE (0). En Debian el valor curiosamente es diferente:

```
# cat /proc/sys/kernel/randomize_va_space
2
```

Según parece valores diferentes de 0, significa que está activa. Para desactivarla se necesita ser root y se puede hacer con la siguiente orden:

```
# echo 0 > /proc/sys/kernel/randomize_va_space
```

****

1.b) File:

```
juanjo@tukan{~/Descargas/crk/crackmes_linux/cm1}$ file cm1
cm1: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linke
d (uses shared libs), corrupted section header size
```

Como vemos es un programa enlazado dinámicamente, por tanto usa librerías compartidas como hemos visto con la orden "**ldd**"; y tiene corrupta el tamaño de la cabecera de sección, en concreto es el valor del número de secciones como veremos con **readelf**.

1.c) Strings:nos muestra las cadenas de texto del ejecutable, con la opción **-t** nos dice la posición de la cadena en el programa, esta posición puede ser en decimal **"d"**, octal **"o"** o en hexadecimal **"x"**; esta última es la que usan los editores hexadecimales:

```
$ strings -t x cm1
    158 /lib/ld-linux.so.2
    16b libc.so.6
    175 _DYNAMIC
    17e _exit
    184 getchar
    18c write
    1de x^< t
    2fe veneta//mbe12345678.v
    314 yxt"J
    321 J678.
    328     CrackME (v1 Linux) (CM11) by veneta//MBE
    352     mail:veneta8@poczta.onet.pl
    36f     site:veneta.prv.pl
    385                 Write Your Password:
    39e     BAD PASSWORD :P
    3b2     OK Cracked Now mail me :)
```

1.d) Readelf; en este caso no nos va a dar toda la información necesaria; pero va a ser suficiente para comenzar el estudio del formato **ELF** en este programa. Empecé por utilizar la opción **-a (all)** para ver todo lo posible (y justamente fue lo contrario :-():

```
$ readelf -a cm1
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:               0x804819f
  Start of program headers:          44 (bytes into file) 0x2c
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes) 0x34
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           0 (bytes)
  Number of section headers:         344 (0x158)
  Section header string table index: 0
readelf: Error: Section headers are not available!
Abortado
```

Como vemos solo nos da la **cabecera ELF** con sus valores en decimal (yo he colocado los valores importantes en hexadecimal), para su estudio vamos a ir viendo esta estructura en el archivo, mediante el editor hexadecimal **hte**. Como ya sabemos por **e_ehsize** (Size of this header) la cabecera ELF ocupa **52** bytes, llegando hasta **0x34** en hexadecimal:

```
|00000000 7f 45 4c 46 01 01 01 00-00 00 00 00 00 00 00 00 |?ELF???         |
|00000010 02 00 03 00 01 00 00 00-9f 81 04 08 2c 00 00 00 |? ? ?   ????,   |
|00000020 00 00 00 00 00 00 00 00-34 00 20 00|03 00 00 00 |        4   ?   |
|00000030 58 01 00 00 |
                      |
                    0x34
```

Si nos fijamos en los valores relacionados con las secciones, hay valores erroneos; pues tanto el tamaño de la cabecera de sección **e_shentsize** (Size of section headers) como la tabla de los nombres de las secciones **e_shstrndx** (Section header string table index) su valor es 0; lo que indica que no están definidas las secciones o que no existen; pero no cuadra con el valor del número de secciones **e_shnum** (Number of section headers) que nos indica que hay **344** secciones (**0x158** en hexedecimal) , que provoca que **readelf** aborte su ejecución (readelf: Error: Section headers are not available!) y no siga analizando el archivo.

Cuando lleguemos al momento de la depuración veremos a fondo que poner un valor muy alto en el numero de secciones es un **metodo antidebugger** específico para nuestro querido **GDB,** lo que nos pondrá difícil utilizarlo aunque como ya sabemos, no imposible ;-)

También vemos que ha provocado que **readelf** no haya terminado su análisis; pues si nos fijamos en la cabecera del programa, vemos que existe el valor **e_phoff**(Start of program headers) pero está sin analizar; para estos casos debemos usar las opciones especificas de **readelf**; en este caso **"-l"** que solo mirará la cabecera del programa:

```
$ readelf -l cm1

Elf file type is EXEC (Executable file)
Entry point 0x804819f
There are 3 program headers, starting at offset 44

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  INTERP         0x000158 0x08048158 0x08048158 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x003cd 0x00403 RWE 0x1000
  DYNAMIC        0x0000a8 0x080480a8 0x080480a8 0x00050 0x00050 RW  0x4
```

Esta parte ya la vimos en el anterior tutorial, por lo que iremos mas rápido. Con esta orden nos muestra el valor del **Entry point**, 0x804819F , y nos dice que hay tres cabeceras del programa que comienzan en el byte **44** (0x2C). Al igual que el crackme anterior, el inicio de la primera cabecera del programa esta en **0x2C** (44), donde se solapa con el final de la cabecera **ELF** (llega hasta **0x34**) y como también sabemos, el tamaño de cada cabecera del programa, **e_phentsize,** es de **32 bytes**, por lo que podemos ver cada cabecera individualmente:

1º cabecera del programa

```
                                                0x2C
                  Fin de la Cabecera ELF 0x34     |
                          |                       |03 00 00 00 |        4   ?   |
|00000030 58 01 00 00 | 58 81 04 08-58 81 04 08 13 00 00 00 |X?  X???X????   |
|00000040 13 00 00 00 04 00 00 00-01 00 00 00
```

Los valores coincide con la información que nos da readelf:

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  INTERP         0x000158 0x08048158 0x08048158 0x00013 0x00013 R   0x1
```

El valor de **p_type** (03 00 00 00) nos esta indicando como manejar el contenido de este segmento:

```
#define PT_INTERP       3            /* Interpretador del programa */
```

En concreto si miramos el offset **0x158** vemos que esta dándonos el nombre del cargador dinámico **/lib/ld-linux.so.2**, que como ya sabemos es el responsable de la vinculación dinámica:

```
                             0x158
                               |
                                2f 6c 69 62 2f 6c 64 2d |??????  /lib/ld-|
|00000160 6c 69 6e 75 78 2e 73 6f-2e 32 00 |           |linux.so.2
                                          |
                                        0x16b
```

Ahora entendemos el motivo del solapamiento de la cabecera ELF con esta primera cabecera del programa; el valor **0x158** es imprescindible para la ejecución del programa, ya que sabemos que un programa dinámico no podría funcionar sin un enlazador dinámico y justamente este valor indica el offset donde se encuentra; por tanto el valor antidebugger que indica que hay 344 secciones (0x158) no se puede borrar sin dejar el programa inservible (muy listo...). Ya veremos mas adelante como solucionar este problema.

2º cabecera del programa

```
                                                  01 00 00 00 |?   ?   ?   ?   |
|00000050 00 00 00 00 00 80 04 08-00 80 04 08 cd 03 00 00 |     ??? ?????  |
```

```
|00000060 03 04 00 00 07 00 00 00-00 10 00 00
```

Si vemos el valor **p_type** (01 00 00 00) comprobamos que es **PT_LOAD**; un segmento de un programa ejecutable y que corresponde al segmento que carga el programa en memoria como vemos con **readelf**:

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x003cd 0x00403 RWE 0x1000
```

3º cabecera del programa

```
                                                  02 00 00 00 |??  ?    ?  ?   |
|00000070 a8 00 00 00 a8 80 04 08-a8 80 04 08 50 00 00 00 |?   ????????P   |
|00000080 50 00 00 00 06 00 00 00-04 00 00 00 |
                                              |
                                            0x8c
```

El valor **p_type** (02 00 00 00) de esta cabecera de programa nos indica que a este segmento le corresponde un tipo **PT_DYNAMIC**:

#define PT_DYNAMIC 2 /* **Informacion de linkeado dinamico** */

Como vemos este segmento tiene la información necesaria para realizar el enlazado dinámico y por tanto vamos a verlo en profundidad para comprender su estructura. Primero debemos de encontrar el segmento, para ello utilizamos los datos que nos da su cabecera de programa en **readelf**:

```
      Type       Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
DYNAMIC  0x0000a8 0x080480a8 0x080480a8 0x00050 0x00050 RW  0x4
```

Este segmento comienza a partir del offset **0xa8** (168) , su dirección virtual cuando se cargue en memoria sera **0x80480a8** y su tamaño tanto en el archivo (**FileSize**) como en memoria (**MemSize**) sera de **0x50** (80 bytes); se permite la lectura y escritura (**RW**) en memoria. Por tanto debemos buscar la zona que comienza en **0xA8** y ocupa **80 bytes**, llegando hasta el offset **0xf8**:

```
                                 0Xa8
                                   |
                                   |01 00 00 00 01 00 00 00 |?       ?   ?   |
|000000b0 04 00 00 00 8c 80 04 08-05 00 00 00 6a 81 04 08 |?   ?????   j???|
|000000c0 06 00 00 00 f0 80 04 08-0a 00 00 00 28 00 00 00 |?   ?????   (   |
|000000d0 0b 00 00 00 10 00 00 00-11 00 00 00 40 81 04 08 |?   ?   ?   @???|
|000000e0 12 00 00 00 18 00 00 00-13 00 00 00 08 00 00 00 |?   ?   ?   ?   |
|000000f0 00 00 00 00 00 00 00 00 |
                                  0xf8
```

Toda la información necesaria para manejar este tipo de segmentos están en las especificaciones del formato ELF:

http://pdos.csail.mit.edu/6.828/2005/readings/elf.pdf

Eso si en Inglés, así que perdonad si la traducción entre Google y yo es defectuosa; pero en general he conseguido descubrir el significado de todos los valores:

Dynamic Section

Si un archivo objeto participa en la vinculación dinámica, la tabla de cabecera del programa tendrá un elemento de tipo **PT_DYNAMIC**. Este segmento contiene la sección **.dynamic**. Un

símbolo especial, **_dynamic,** es la etiqueta de esta sección, que contiene una matriz de las siguientes estructuras:

```
Dynamic Structure

  typedef struct {
      Elf32_Sword d_tag;
      union {
          Elf32_Sword  d_val;
          Elf32_Addr   d_ptr;
      } d_un;
  } Elf32_Dyn;

  extern Elf32_Dyn _DYNAMIC[];
```

Esta función indica que para cada objeto perteneciente a esta estructura, hay un valor **d_tag** que controla la interpretación del valor **d_un**, que puede contener a su vez un valor **d_val** o **d_ptr.**

Primero hay que definir que son los valores d_val y d_ptr:

*** d_val**

Representa un valor entero de tamaño word (**4 bytes**) en **ELF32** con diferentes interpretaciones.

***d_ptr**

Este objeto representa una dirección virtual en **ELF32** (son **4 bytes**).

Teniendo en cuenta esto, existe una tabla de valores para **d_tag,** donde se indica según sea un ejecutable o una biblioteca compartida, la necesidad obligatoria de que aparezca ese tag, y se marca como "**mandatory**" o puede ser un tag cuya presencia es opcional, y se marca como "**optional**":

```
Dynamic Array Tags, d_tag

  Name              Value  d_un     Executable   Shared Object
  ====              =====  ====     ==========   =============
  DT_NULL             0    ignored  mandatory    mandatory
  DT_NEEDED           1    d_val    optional     optional
  DT_PLTRELSZ         2    d_val    optional     optional
  DT_PLTGOT           3    d_ptr    optional     optional
  DT_HASH             4    d_ptr    mandatory    mandatory
  DT_STRTAB           5    d_ptr    mandatory    mandatory
  DT_SYMTAB           6    d_ptr    mandatory    mandatory
  DT_RELA             7    d_ptr    mandatory    optional
  DT_RELASZ           8    d_val    mandatory    optional
  DT_RELAENT          9    d_val    mandatory    optional
  DT_STRSZ            10   d_val    mandatory    mandatory
  DT_SYMENT           11   d_val    mandatory    mandatory
  DT_INIT             12   d_ptr    optional     optional
  DT_FINI             13   d_ptr    optional     optional
  DT_SONAME           14   d_val    ignored      optional
  DT_RPATH            15   d_val    optional     ignored
  DT_SYMBOLIC         16   ignored  ignored      optional
  DT_REL              17   d_ptr    mandatory    optional
  DT_RELSZ            18   d_val    mandatory    optional
  DT_RELENT           19   d_val    mandatory    optional
```

```
DT_PLTREL          20 d_val      optional     optional
DT_DEBUG           21 d_ptr      optional     ignored
DT_TEXTREL         22 ignored    optional     optional
DT_JMPREL          23 d_ptr      optional     optional
DT_LOPROC     0x70000000  unspecified  unspecified  unspecified
DT_HIPROC     0x7fffffff  unspecified  unspecified  unspecified
```

Para hacerlo de forma práctica vamos a obtener los valores que el segmento **DYNAMIC** nos da para nuestro crackme. Para ello se van tomando valores en parejas 4 bytes + 4 bytes, siendo los primeros 4 bytes un valor entero que nos indique a que **tag** corresponde y los segundos, según la tabla anterior, le corresponderá un valor **d_val** o **d_ptr** con un significado deferente:

```
01 00 00 00 01 00 00 00
```

El valor **01** es **DT_NEEDED**; en este caso le corresponde un valor **d_val**, osea un entero sin signo, que corresponde al desplazamiento hacia una cadena de caracteres terminado en null(00), que indican una librería necesaria (needed). Esta cadena se encuentra en el tag **DT_STRTAB** y en esta caso corresponde al valor **libc.so.6.** Es un tag **opcional**.

```
04 00 00 00 8c 80 04 08
```

El valor **04** corresponde a **DT_HASH**; corresponde un valor **d_ptr** y se refiere a la dirección virtual de la tabla hash de símbolos. Esta tabla hash se utiliza para encontrar los símbolos de la tabla de símbolos referenciados por el elemento **DT_SYMTAB**.
En este caso se encuentra en la dirección 0x0804808C. Es **obligatorio** su existencia.

```
05 00 00 00 6a 81 04 08
```

El valor **05** corresponde con **`DT_STRTAB;`** Este elemento contiene la dirección de la tabla de cadenas, por tanto es un valor **d_ptr**. Nombres de los símbolos, nombres de las bibliotecas, y otras cadenas residen en esta tabla. En nuestro caso esta en 0x0804816A, que corresponde al offset **0x16A**:

```
                                    0x16a
                                      |
                                       00 6c 69  62 63 2e |           libc.|
|00000170 73 6f 2e 36 00 5f 44 59-4e 41 4d 49 43 00 5f 65 |so.6 _DYNAMIC _e|
|00000180 78 69 74 00 67 65 74 63-68 61 72 00 77 72 69 74 |xit getchar writ|
|00000190 65 00 |                                         |e
                |
            0x193 (403)
```

Como veis en la posición **1** (**0x16b**) esta la cadena de la librería NEEDED que hemos visto al principio. Es **obligatorio**.

```
06 00 00 00 f0 80 04 08
```

El valor **06** corresponde con **`DT_SYMTAB;`** este elemento contiene la dirección de la **tabla de símbolos,** le corresponde un valor **d_pt**, que en este caso señala a 0x080480F0. DT_SYMTAB contiene entradas **Elf32_Sym** para los archivos de 32bit, que veremos a continuación. Es **obligatorio**.

```
0a 00 00 00 28 00 00 00
```

El valor **10** (0x0a)corresponde a **`DT_STRSZ;`** este elemento contiene el tamaño en bytes de la tabla

de cadenas; tiene un valor **d_val** y es complementario a **DT_STRTAB**, pues era donde nos decía su comienzo y ahora sabemos su tamaño, ocupa **0x28 bytes** que son 40 en decimal. Es **obligatorio**.

```
0b 00 00 00 10 00 00 00
```

El valor **11** (0x0b)corresponde con **DT_SYMENT;** este elemento contiene el tamaño en bytes de una entrada de la tabla de símbolos. Corresponde a **0x10**, por tanto cada elemento ocupa **16 bytes**. Es **obligatorio** y complementario a DT_SYMTAB.

```
11 00 00 00 40 81 04 08
```

El valor **17** (0x11) corresponde con **DT_REL;** este elemento contiene la dirección de una tabla de reubicación,Un archivo objeto puede tener varias secciones de reubicación. Cuando se construye la tabla de reubicación de un archivo ejecutable o un archivo de objetos compartidos, el editor de enlace concatena las secciones para formar una sola tabla. A pesar de que las secciones permanecen independientes en el archivo de objeto, el enlazador dinámico ve una sola tabla. Cuando el enlazador dinámico crea la imagen de proceso de un archivo ejecutable o agrega un objeto compartido en la imagen del proceso, entonces lee la tabla de reubicación y realice las acciones asociadas. Si este elemento está presente, la estructura dinámica también debe tener **DT_RELSZ** y el elemento **DT_RELENT**. En este caso la tabla de reubicación se localiza en 0x08048140

```
12 00 00 00 18 00 00 00
```

El valor **18** (0x12)corresponde con **DT_RELSZ**; este elemento contiene el tamaño en bytes, de la reubicación **DT_REL** y como vemos es un valor **d_val 0x18**, que indica un tamaño de **24 bytes**.

```
13 00 00 00 08 00 00 00
```

El valor **19 (0x13)** corresponde con **DT_RELENT**; este elemento contiene el tamaño en bytes de cada entrada de la sección de reubicación **DT_REL**. En nuestro caso serán **8 bytes** lo que ocupe cada entrada.

```
00 00 00 00 00 00 00 00
```

El valor **0** corresponde a **DT_NULL**; una entrada con una etiqueta **DT_NULL** que marca el final de la matriz **_dynamic** y es **obligatoria**.

Podemos comprobar nuestros resultados gracias a **readelf** y su opción **-d** que nos muestra la sección dinámica:

```
$ readelf -d cm1

Dynamic section at offset 0xa8 contains 10 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x00000004 (HASH)                       0x804808c
 0x00000005 (STRTAB)                     0x804816a
 0x00000006 (SYMTAB)                     0x80480f0
 0x0000000a (STRSZ)                      40 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000011 (REL)                        0x8048140
 0x00000012 (RELSZ)                      24 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
```

```
0x00000000 (NULL)                       0x0
```

Como hemos visto, aunque este programa no definía en la cabecera ELF ninguna sección, las secciones existen pues son necesarias para realizar el enlace de las funciones externas, que necesita el programa en el momento de la ejecución. Todo este trabajo lo realiza el cargador dinámico, **ld-linux.so.2**, y gracias a esta sección **_dynamic** ya tiene todo lo necesario para comenzar.

Lo primero sera buscar que funciones necesita el programa, para ello existe la sección **HASH** que le indica a **ld-linux.so.2** como encontrar los elementos en la tabla de símbolos:

Tabla HASH

Una tabla hash contiene elementos tipo **Elf32_Word**, por tanto son **4 bytes**, y sirven para apoyar el acceso a la tabla de símbolos.

En este caso la tabla se localiza en **0x8C**:

```
                                              0x8c
                                                |
                                                |01 00 00 00 |P   ?   ?   ?   |
00000090 05 00 00 00 04 00 00 00-00 00 00 00 02 00 00 00 |?   ?       ?   |
000000a0 03 00 00 00 00 00 00 00 |
                                 |
                               0xa8
```

Su estructura es la siguiente, en este orden y teniendo en cuenta que cada valor ocupa 4 bytes:

```
nbucket                 01 00 00 00
nchain                  05 00 00 00
bucket[0]               04 00 00 00
...
bucket[nbucket - 1]
chain[0]                00 00 00 00
...                     02 00 00 00
chain[nchain - 1]       03 00 00 00
```

El array **bucket** (un tipo de array por casillero) contiene un numero de entradas **nbucket,** y el array de **cadena** (chain) contiene un número de entradas **nchain**; teniendo en cuenta que ambos índices comienzan en **0**.El número de entradas de la tabla de símbolos debe ser igual a **nchain**, de modo que los índices de la tabla de símbolos también puede seleccionar las entradas correspondientes en la tabla de cadenas.

Aunque lo veremos mas adelante, la **tabla de símbolos** es **0** sin nombre, **1 _Dynamic**, **2 _exit**, **3 getchar** y **4 write.**

Por tanto en nuestro caso, **nbucket** es **01** y **nchain** es **05** (la tabla de símbolos también tiene 5 elementos comenzando por el 0). Después comienza todas las entradas bucket que en este caso solo hay una, **bucket[0]** que tiene el valor **04** que corresponde al símbolo **getchar** (es su número de orden, pues como símbolo es el 3). A continuación tenemos el **array chain**, siendo **chain[0]** el valor **0** con valor NULL, **chain[1]** tiene el valor **02** que corresponde con **_exit** y el **chain[2]** con el valor **03** que corresponde con **write**.

Según he podido comprobar, el **array bucket** determina los símbolos que necesita una librería externa (en nuestro caso solo hay uno, **getchar**) mientras que el **array chain** cubre funciones como llamadas al sistema (como es nuestro caso con **write** y **_exit**) y nombres de bibliotecas compartidas

(**_Dynamic**).

Tabla de SIMBOLOS

La tabla de símbolos de un archivo objeto contiene información necesaria para localizar y reubicar las definiciones simbólicas de un programa y sus referencias. Un símbolo índice de la tabla es un subíndice dentro de esta matriz. El **indice 0** designa tanto la primera entrada en la tabla y sirve como símbolo no definido.

Cada entrada de la tabla de símbolos tiene esta estructura:

```
typedef struct {
     Elf32_Word st_name;
     Elf32_Addr st_value;
     Elf32_Word st_size;
     unsigned char   st_info;
     unsigned char   st_other;
     Elf32_Half st_shndx;
 } Elf32_Sym;
```

*** st_name**

Es un índice a una cadena de caracteres en la tabla de nombres de símbolos de un archivo objeto, que contiene las representación en caracteres del nombre del símbolo. Si el valor es distinto de cero, representa un índice en la tabla de strings que le da el nombre del símbolo. De lo contrario, la entrada de la tabla de símbolos no tiene nombre.

Estos nombre son los mismo que se dan en el lenguaje C.

*** st_value**

Este miembro da el valor del símbolo asociado. según el contexto, esto puede ser un valor absoluto, una dirección, etc; los detalles se muestran a continuación.

*** st_size**

Muchos símbolos tienen asociados los tamaños. Por ejemplo,el tamaño de un objeto de datos es el número de bytes que contiene el objeto. Este miembro contiene 0 si el símbolo no tiene un tamaño o un tamaño desconocido.

*** st_info**

Este miembro especifica el tipo de símbolo y atributos de enlace.
La lista de los valores y significados aparece a continuación. El siguiente código muestra cómo manipular los valores.

```
#define ELF32_ST_BIND(i)    ((i)>>4)
#define ELF32_ST_TYPE(i)    ((i)&0xf)
#define ELF32_ST_INFO(b, t) (((b)<<4)+((t)&0xf))
```

Los resultados de cada código se deben comparar a su tabla correspondiente, **st_bind** y **st_type** que os pongo a continuación:

***ST_BIND**

Cada símbolo determina como se vera la vinculación y su comportamiento; por tanto estos símbolos determinan como se van a comportar entre diferentes archivos objetos:

Symbol Binding, ELF32_ST_BIND

Name	Value
STB_LOCAL	0
STB_GLOBAL	1
STB_WEAK	2
STB_LOPROC	13
STB_HIPROC	15

*** STB_LOCAL**

Los símbolos locales no son visibles fuera del archivo objeto que contiene su definición. Símbolos locales del mismo nombre pueden existir en varios archivos sin interferir unos con otros.

*** STB_GLOBAL**

Los símbolos globales son visibles para todos los archivos de objetos que se combinan. Un archivo de definición de un símbolo global va a satisfacer a otro archivo con referencia indefinida, siendo el símbolo global el mismo.

*** STB_WEAK**

Los símbolos débiles se asemejan a los símbolos globales, pero tienen sus definiciones con menor prioridad.

*** STB_LOPROC through STB_HIPROC**

Valores en este rango de inclusión son reservados para el procesador específico .

***ST_TYPE ; Tipos de Símbolos:**

Los tipos de los símbolos nos da una clasificación general de las entidades a las que están asociados:

ELF32_ST_TYPE

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13

STT_HIPROC 15

*** STT_NOTYPE**

El tipo de símbolo no se especifica.

*** STT_OBJECT**

El símbolo está asociado con un objeto de datos, como una variable, un matriz, etc

*** STT_FUNC**

El símbolo está asociado con una función o código ejecutable.

*** STT_SECTION**

El símbolo está asociado con una sección. Las entradas de la tabla de símbolos de este tipo existen principalmente para el traslado y, normalmente, tienen **STB_LOCAL** vinculante.

*** STT_FILE**

Convencionalmente, el nombre del símbolo le da el nombre del archivo de origen asociados con el archivo de objeto. Un símbolo de archivo tiene STB_LOCAL vinculante, si su índice de la sección es SHN_ABS, y que precede a otros símbolos STB_LOCAL para el archivo, si está presente.

*** STT_LOPROC través STT_HIPROC**

Valores en este rango de inclusión son reservados para el procesador específico.

*** st_other**

Este miembro actualmente tiene valor 0 y no tiene ningún significado definido.

*** st_shndx**

Cada entrada de la tabla de símbolos es definida en relación con algún sector; este valor tiene el correspondiente índice de la tabla en la cabecera de la sección. Como se puede ver algunos índices de sección tienen un significado especial.
El valor **índice** de la sección, indica como es la relación del símbolo con su sector correspondiente:

*** SHN_ABS**

El símbolo tiene un valor absoluto que no va a cambiar debido a la reubicación.

*** SHN_COMMON**

Es el símbolo de las etiquetas de un bloque común que aún no ha sido asignado.
El valor del símbolo da restricciones de alineación, de forma similar a un miembro **sh_addralign.** Es decir, el editor de enlace asigna el almacenamiento para el símbolo en una dirección que es un múltiplo de **st_value**. El tamaño del símbolo indica cuántos bytes son necesarios.

*** SHN_UNDEF**

Indica que el símbolo no está definido. cuando el enlazador combina este archivo con otro objeto que define el símbolo indicado, las referencias de este archivo al símbolo es vinculado a su definición real.

```
Special Section Indexes

Name              Value
 ====              =====
SHN_UNDEF            0
SHN_LORESERVE    0xff00
SHN_LOPROC       0xff00
SHN_HIPROC       0xff1f
SHN_ABS          0xfff1
SHN_COMMON       0xfff2
SHN_HIRESERVE    0xffff
```

Con toda esta información ya estamos preparados para descubrir la tabla de símbolos del programa **cm1**, que en este caso no nos lo muestra **readelf** (se utiliza la orden **"readelf -s"**).
Para saber donde localizar esta sección utilizamos la información que nos ha dado **_dynamic**, siendo **DT_SYMTAB** quien nos indica que la sección comienza en **0xF0** y con **DT_SYMENT** sabemos que cada entrada de esta tabla tiene un tamaño de **16 bytes**.

```
000000f0 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |                |
00000100 0b 00 00 00 a8 80 04 08-00 00 00 00 11 00 f1 ff |?   ????    ? ??|
00000110 14 00 00 00 00 00 00 00-00 00 00 00 12 00 00 00 |?           ?   |
00000120 1a 00 00 00 00 00 00 00-00 00 00 00 22 00 00 00 |?           "   |
00000130 22 00 00 00 00 00 00 00-00 00 00 00 22 00 00 00 |
```

1º Entrada

```
00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00
```

st_name: No tiene nombre, son 4 bytes ----> 00 00 00 00
st_value; no tiene valor asociado, son 4 bytes --->00 00 00 00
st_size; tamaño 0, son 4 bytes ------------------>00 00 00 00
st_info; de aquí obtenemos el tipo NOTYPE y el valor UNDEF, un solo byte -> 00
st_other; no tiene significado, un solo byte-------------> 00
st_shndx; valor UNDEF , son dos bytes--------------------> 00 00

```
Num:    Value  Size Type    Bind   Vis      Ndx Name
0:    00000000   0 NOTYPE  LOCAL  DEFAULT   UND
```

2º Entrada

```
0b 00 00 00 a8 80 04 08-00 00 00 00 11 00 f1 ff
```

st_name, su valor corresponde al byte número **11 (0x0B)** de la tabla de cadena de caracteres **DT_STRTAB**, que corresponde con **_DYNAMIC**-----> 0b 00 00 00

st_value; en este caso es una dirección virtual que indica donde comienza este símbolo; como ya sabemos es **0x080480A8** -----> a8 80 04 08

st_size: en este caso el tamaño no esta definido ----> 00 00 00 00

st_info; su valor es **0x11**, para saber su significado debemos aplicarle su código correspondiente:

#define ELF32_ST_BIND(i) ((i)>>4) //4 desplazamientos de bits a la derecha de (0x11)
>>> hex(0x11>>4)
'0x1' ---> Corresponde con el valor **STB_GLOBAL**, por tanto este símbolo sera visible para todos los objetos relacionados

#define ELF32_ST_TYPE(i) ((i)&0xf) // AND lógico entre el valor 0x11 y 0xf
>>> hex(0x11&0xf)
'0x1'---> Corresponde con el tipo **STT_OBJECT,** por tanto este símbolo esta asociado a un matriz de datos, que es la sección **_dynamic** que hemos visto

#define ELF32_ST_INFO(b, t) (((b)<<4)+((t)&0xf)) , este código debe darnos el valor que hemos encontrado (0x11 es st_info) tomando cada parte del byte independiente, en este caso 0x1 y 0x1:

>>> hex((0x1<<4)+(0x1&0xf))
'0x11'

st_other; su valor es 0.

st_shndx: su valor es **ff f1** ----------> F1 FF
Corresponde con el índice de sección **SHN_ABS** , por tanto su valor es absoluto y no se va a ser modificado.

Resumiendo, lo ponemos ver como lo presenta **readelf**:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
1:	**08048086**	**0**	**OBJECT**	**GLOBAL**	**DEFAULT**	**ABS**	**_DYNAMIC**

Solo queda por aclarar la columna **"Vis"**, que corresponde a los atributos de visibilidad (que no he encontrado en las especificaciones ELF que tengo); en la mayoría de los casos no se tienen en cuenta y siempre se ponen como **"Default"** ya que normalmente la visibilidad esta indicada por el valor **ST_BIND**; ya que si es Global sera visible, mientras que si es Local solo sera visible dentro del archivo objeto.
Los atributos de visibilidad son de cuatro tipo, por defecto (**DEFAULT**), protegido (**PROTECTED)**, oculto (**HIDDEN**) o interno (**INTERNAL**) pero realmente no se como están definidos en el formato ELF.

3º Entrada

```
14 00 00 00 00 00 00 00-00 00 00 00 12 00 00 00
```

st_name; corresponde al byte **0x14 (20)** de la tabla **DT_STRTAB** que coincide con **_exit**
st_value; no hay valor definido.
st_size; no hay valor definido-
st_info; tenemos el valor **0x12** por lo tanto:

#define ELF32_ST_BIND(i) ((i)>>4)
>>> hex(0x12 >> 4)
'0x1'----> Corresponde al valor **STB_GLOBAL**

#define ELF32_ST_TYPE(i) ((i)&0xf)
>>> hex(0x12&0xf)
'0x2'----> Corresponde al tipo **STT_FUNC**, por tanto este símbolo esta asociado a una función o código ejecutable.

#define ELF32_ST_INFO(b, t) (((b)<<4)+((t)&0xf))
>>> hex((0x1<<4)+(0x2&0xf))
'0x12'

st_other; vale 0, no esta definido

st_shndx; vale 0 se define como **SHN_UNDEF,** por tanto las referencias de este archivo al símbolo es vinculado a su definición real, que en otras palabras y como veremos con GDB, el símbolo **_exit** se unirá directamente con la función **_exit** de una librería externa por medio del cargador dinámico.

Resumiendo:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
2:	**0**	**0**	**FUNC**	**GLOBAL**	**DEFAULT**	**UND**	**_exit**

4º Entrada

```
1a 00 00 00 00 00 00 00-00 00 00 00 22 00 00 00
```

st_name; su valor es **0x1a (26)** de **DT_STRTAB** que corresponde al nombre **getchar**.
st_value; 00 00 00 00
st_size; 00 00 00 00
st_info; tenemos un valor **0x22**

```
#define ELF32_ST_BIND(i)     ((i)>>4)
>>> hex(0x22 >> 4)
'0x2'----> Corresponde al valor STB_WEAK, semejante a Global pero con menos
prioridad ¿?

#define ELF32_ST_TYPE(i)     ((i)&0xf)
>>> hex(0x22&0xf)
'0x2'----> Corresponde al tipo STT_FUNC, por tanto este símbolo esta asociado a
una función o código ejecutable.

#define ELF32_ST_INFO(b, t)  (((b)<<4)+((t)&0xf))
>>> hex((0x2<<4)+(0x2&0xf))
'0x22'
```

st_other; no esta definido.

st_shndx; su valor es **0** que corresponde a **SHN_UNDEF.** En este caso el símbolo **getchar** se unirá a su función correspondiente de **libc.so.6.**

Resumiendo:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
3:	**0**	**0**	**FUNC**	**WEAK**	**DEFAULT**	**UND**	**getchar**

5º Entrada

```
22 00 00 00 00 00 00 00-00 00 00 00 22 00 00 00
```

Es igual que la anterior, solo que **st_name** corresponde a la función **write**; por tanto podemos resumir:

Num:	Value	Size	Type	Bind	Vis	Ndx	Name
4:	**0**	**0**	**FUNC**	**WEAK**	**DEFAULT**	**UND**	**write**

Con esto acabamos el tema de los símbolos, que como vemos es una forma de darle características claras a cada uno de los objetos (la mayoría funciones externas) al cargador dinámico, para que sepa su nombre, que relación tiene con los demás objetos y sus características mas importantes.
Por tanto, ya solo queda crear la zona concreta donde se realice esa vinculación a nivel de código, y para eso se utiliza la **Relocation**; que traduciremos como reubicación, y que justamente define todo el proceso final del enlazado dinámico.

RELOCATION:

La reubicación es el proceso de conexión de las referencias simbólicas con las definiciones simbólicas. Por ejemplo, cuando un programa llama a una función,la instrucción de llamada asociada debe transferir el control a la adecuada dirección de destino en la ejecución. En otras palabras, los archivos reubicables debe tener la información que se describe para modificar el contenido de sus secciones, lo que permite a ejecutables y archivos objetos compartidos, mantener la información adecuada para la imagen del programa en un proceso.

Hay dos estructuras definidas para cada entrada de reubicación:

```
typedef struct {
     Elf32_Addr r_offset;
     Elf32_Word r_info;
 } Elf32_Rel;

 typedef struct {
     Elf32_Addr r_offset;
     Elf32_Word r_info;
     Elf32_Sword     r_addend;
 } Elf32_Rela;
```

*** r_offset**

Este miembro ofrece el lugar en que se aplicará la acción de reubicación. Para un archivo reubicable, el valor es el desplazamiento de bytes desde el principio de la sección de la unidad de almacenamiento afectada por la reubicación. Para un archivo ejecutable o un objeto compartido, el valor es la dirección virtual de la unidad de almacenamiento afectados por la reubicación.

*** r_info**

Este miembro ofrece tanto el índice de la tabla de símbolos con respecto al cual el traslado se va a hacer, y el tipo de reubicación que se va a aplicar.
Por ejemplo, la entrada de una instrucción de llamada de reubicación tendría el índice de la tabla de símbolos de la función que se llama. Si el índice es **STN_UNDEF** (índice símbolo indefinido), la reubicación utiliza 0 como valor símbolo y los tipos de reubicación específicos del procesador.

Cuando el texto se refiere al tipo de reubicación de una entrada o al índice de la tabla de símbolos , es el resultado de aplicar **ELF32_R_TYPE** o **ELF32_R_SYM**, respectivamente, a los miembros **r_info** de la entrada. Para ello seguimos este código:

```
# define ELF32_R_SYM (i) ((i)>> 8)
# define ELF32_R_TYPE (i) ((unsigned char) (i))
# define ELF32_R_INFO (s, t) ((s) <<8 + (unsigned char) (t))
```

*** r_addend**

Este miembro especifica un sumando constante que se utiliza para calcular el valor que debe ser almacenados en el campo reubicable.

En nuestro caso, sabemos por **_DYNAMIC** que tenemos una sección **REL** , localizada a partir del byte **0x140** (0x8048140), con un tamaño definido por **RELSZ** de **0x18 bytes (24)** y donde cada entrada, **RELENT**, tiene un tamaño de **8 bytes.**
También sabemos al ser una sección **REL** y no RELA, que nos corresponde la primera estructura que hemos visto para cada entrada de reubicación:

```
typedef struct {
      Elf32_Addr r_offset;
      Elf32_Word r_info;
  } Elf32_Rel;
```

Con el editor hexadecimal se puede ver así:

```
|00000140 dc 83 04 08 01 04 00 00-d8 83 04 08 01 03 00 00 |??????  ??????  |
|00000150 ec 83 04 08 01 02 00 00-|
                                  |
                               0x158
```

1º Entrada `dc 83 04 08 01 04 00 00`

***r_offset**, se corresponde con la dirección 0x080483DC
***r_info**, en este caso su valor es 00 00 04 01

Se toma el valor **r_info** para obtener el valor **R_SYM**:

ELF32_R_SYM (i) ((i)>> 8) //8 desplazamiento de bits a la derecha de (i)
>>> hex(0x401>>8)
'0x4' -----> Por tanto esta reubicación esta relacionado con el símbolo numero **4**, que como sabemos por la tabla de símbolos es **write**.

ELF32_R_TYPE (i) ((unsigned char) (i)); en este caso solo nos indica que miremos un valor char sin signo (1 byte), que corresponde con el byte de mas peso, **0x01**. Para saber que tipo de reubicación le corresponde tenemos una tabla que nos lo indica:

Relocation Types

Name	Value	Field	Calculation
R_386_NONE	0	none	none
R_386_32	**1**	**word32**	**S + A**
R_386_PC32	2	word32	S + A - P

```
R_386_GOT32          3     word32      G + A - P
R_386_PLT32          4     word32      L + A - P
R_386_COPY           5     none        none
R_386_GLOB_DAT       6     word32      S
R_386_JMP_SLOT       7     word32      S
R_386_RELATIVE       8     word32      B + A
R_386_GOTOFF         9     word32      S + A - GOT
R_386_GOTPC          10    word32      GOT + A - P
```

En este caso la reubicación es del tipo **R_386_32**.

ELF32_R_INFO (s, t) ((s) <<8 + (unsigned char) (t))
En este caso tenemos dos valores, (s) de sym que como hemos visto es 0x4 y (t) de type que es 0x1. Por tanto en Python vemos:
>>> hex((0x4<<8)+0x1)
'0x401' ----> Es el valor correcto de **r_info** que hemos encontrado

2º Entrada d8 83 04 08 01 03 00 00

***r_offset**, se corresponde con la dirección 0x080483D8
***r_info**, en este caso su valor es 00 00 03 01. Sin hacer nada ya podemos ver que el valor de **r_type** es **1** y su símbolo asociado es **03**. Lo comprobamos:

ELF32_R_SYM (i) ((i)>> 8)
>>> hex(0x301>>8)
'0x3' ----> Corresponde al simbolo nº3 que corresponde a **getchar**.

ELF32_R_TYPE (i) ((unsigned char) (i))
Nos indica el tipo nº1 que corresponde con **R_386_32**.

# define ELF32_R_INFO (s, t) ((s) <<8 + (unsigned char) (t))
>>> hex((0x3<<8)+0x1)
'0x301' -----> **r_info.**

3º Entrada ec 83 04 08 01 02 00 00

***r_offset**, se corresponde con la dirección 0x080483EC
***r_info**, en este caso su valor es 00 00 02 01. No nos vamos a repetir, el valor de **r_type** es **1** que corresponde con **R_386_32** y su símbolo asociado es **02** que según la tabla de símbolos corresponde con **_exit.**

Para terminar podemos colocar todos estos datos estilo **readelf**:

```
Offset     Info    Type              Sym.Value   Sym. Name
080483dc  00000401 R_386_32          00000004      write
080483d8  00000301 R_386_32          00000003     getchar
080483ec  00000201 R_386_32          00000002      _exit
```

2. Desensamblado.

En este programa no me ha sido necesario ver el código desensamblado; para las pocas veces que lo

he necesitado me ha sido útil el programa **hte** con el modo **disasm/x86**.

3. Depuración.

3.1)Metodo Antidebugger.

Como ya hemos visto, este programa corrompe la cabecera ELF para protegerlo de miradas curiosas; de modo que elimina las secciones para evitar el uso de **objdump** y también provoca un trabajo incompleto en **readelf**. Pero el punto fuerte es colocar un valor muy alto en el número de secciones, que provoca que **GDB** no pueda depurarlo:

```
      $ gdb -q cm1
"/home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1": not in executable format:
No se reconoce el formato del fichero
gdb> r
No executable file specified.
Use the "file" or "exec-file" command.
gdb>
```

Podemos comprobar que el metodo antidebugger solo es debido al número elevado de secciones, para ello abrimos el ejecutable con **hte**, nos vamos al byte **0x30** y vemos el valor 58 01, con **F4** editamos el archivo, cambiamos estos valores por 00 00 y salvamos con **F2.** Si ahora cargamos el archivo en GDB:

```
      $ gdb -q cm1
Reading symbols from /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1...(no
debugging symbols found)...done.
gdb> r
/bin/bash: /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1: no se puede
ejecutar el fichero binario
/bin/bash: /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1: Conseguido
During startup program exited with code 1.
gdb>
```

Como veis, el programa ahora no nos da ningún error, pero como ya sabemos no se puede ejecutar pues el valor 0x158 también es el valor del offset donde esta el nombre del cargador dinámico; y sin este valor un programa dinámico no se puede ejecutar. Por tanto volvemos a dejar el programa **cm1** como estaba con **hte** y buscamos otras maneras de resolver el problema; y como ya sabemos, en informática siempre hay diferentes soluciones para un mismo problema:

a) Usar otros debugger. Pues si, esta sería la solución mas fácil, tanto **IDA** como **radare** no se ven afectado por este método. Radare se encuentra el problema pero continua (lo que debería hacer gdb¿No?) y se puede depurar sin problemas.

Con **IDA** he probado de forma remota tanto con el servidor propio, **linux_server** como con **gdbserver.** Esto último me dio una pista, si **gdbserver** no daba problemas, puede que también se pueda depurar con **gdb de modo remoto**, para ello tenemos que ejecutar primero el servidor:

```
$ gdbserver --remote-debug localhost:2345 ./cm1
```

Y por otro lado, ejecutamos **gdb** sin cargar ningún programa y una vez dentro, ejecutamos esta orden:

```
gdb> target remote localhost:2345
```

Y funcionó, se puede depurar sin problemas. De todos modos no lo he utilizado pues este método tiene algunas limitaciones (como no poder usar los catchpoint sobre syscall ¿?) que me hicieron seguir buscando.

b) Arreglar la cabecera ELF manualmente. Esta parte ha sido mas de prueba y de camino, jugar un poco con el formato ELF, ya que el resultado no ha sido perfecto :-P

Para ello abrí el archivo con un editor hexadecimal, en este caso con **okteto,** que me ha funcionado mejor que hte para esto. Una vez abierto, como tenemos pestañas, abrimos un archivo nuevo y vamos copiando, primero la **cabecera ELF** completa, los **52** primeros bytes donde aprovecho para quitar el valor 58 01, después copio los primero **8 bytes** de la **primera cabecera del programa** (los que se solapaban con el final del header elf) dejando ahí si el valor 58 01 y los añado al ejecutable nuevo; por último tomo el resto del archivo desde el byte **52** y lo copio a continuación.

De esta manera tenemos un ejecutable 8 bytes mas grande sin problemas de solapamiento y sin antigdb; el problema es que esto implica cambiar todas las direcciones virtuales, tanto el **entry point**, como en el segmento **LOAD** cambiar los valores del tamaño y direcciones, y también en todas las secciones dinámicas que hemos visto. Como veis esta solución no es práctica y puede tener muchos errores (como así fue). Por último, salvamos el nuevo archivo como **cm1.elf** y lo probamos:

```
      $ ./cm1.elf
[J678.

      CrackME (v1 Linux) (CM11) by veneta//MBE
      mail:veneta8@poczta.onet.pl
      site:veneta.prv.pl


                  Write Your P12345
Violación de segmento
```

Como veis hay errores en las strings y después de poner el serial nos salta una **"Violación de segmento"**; puede que me haya dejado algún valor sin cambiar o que esto sea una locura, pero bueno lo he intentado ;-)

c) Solución de Gera. El gran Maestro **Gerardo Richarte**, en la lista de correo **Crackslatinos**, nos ofreció hace unos años una solución para estos archivos corruptos. Yo pedí en su momento ayuda para solucionar este tipo de problemas (justamente era este mismo crackme) y gracias a la colaboración de **NCR+,** el gran **Gera** nos mando un script en **Python** que solucionaba este problema, él comentaba que funcionaría en programas estáticos y que no sabía su resultado con programas dinámicos.
Por mi parte, yo guarde todo lo que nos mando por si en el futuro me era necesario; para quien lo

quiera, script + librerias necesarias + mensaje de correo de Gera explicando su funcionamiento, lo he subido aquí:

http://www.4shared.com/file/DkQ5LGt6/Arreglando_ELFbyGeratar.html

El script en python de Gera, se llama **fixefl.py**, para su funcionamiento necesita dos archivos mas, **elf.py** y **exelib.py.** Todos ellos los debemos mover a la carpeta donde se vaya a utilizar y para aplicarlo utilizamos esta orden:

```
$ python fixefl.py cm1 cm1.ok
```

Como vemos, debemos poner el nombre del script y como argumentos el nombre del archivo original y el nombre del archivo con las modificaciones realizadas, en este caso **cm1.ok.**
El resultado en este caso ha sido perfecto,seguramente porque aunque es un elf dinámico, como hemos visto, tiene un formato un poco especial. Al final esta es la solución que he utilizado para comenzar con la depuración con GDB.

Aunque toda la información la encontrareis en el correo que **Gera** mando a la lista **Crackslatinos,** vamos a ver como ha solucionado nuestro problema. Para ello comprobamos la cabecera ELF con **readelf**:

```
            $ readelf -h cm1.ok
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:               0x804819f
  Start of program headers:          52 (bytes into file)
  Start of section headers:          0 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         4
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0
```

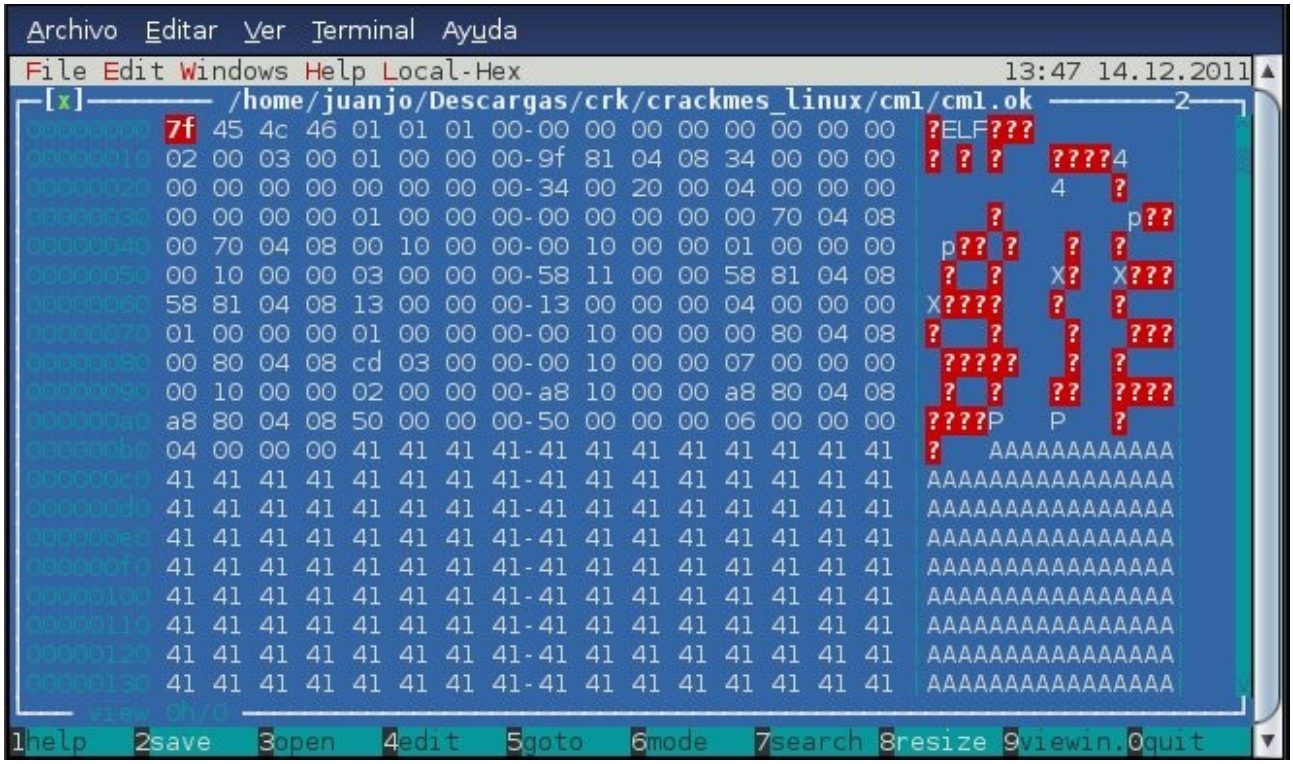
Lo primero, es que el número de secciones es 0 y ya tenemos solucionada la protección antidebugger. Además ha quitado el solapamiento del header ELF y la primera cabecera del programa; que comienza a partir del byte 52 (y no 44 como antes).
Lo importante de esta idea es que no ha tocado nada del archivo original, por lo que mantiene el Entry point original y sus direcciones intactas. La única diferencia es en el número de **cabeceras del programa** que han pasado de 3 a 4; vamos a verlas con **readelf**:

```
            $ readelf -l cm1.ok

Elf file type is EXEC (Executable file)
Entry point 0x804819f
There are 4 program headers, starting at offset 52

Program Headers:
```

```
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08047000 0x08047000 0x01000 0x01000   E 0x1000
  INTERP         0x001158 0x08048158 0x08048158 0x00013 0x00013 R   0x1
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD           0x001000 0x08048000 0x08048000 0x003cd 0x01000 RWE 0x1000
  DYNAMIC        0x0010a8 0x080480a8 0x080480a8 0x00050 0x00050 RW  0x4
```

Como vemos se ha creado otro segmento, de tipo **LOAD** , con una Image Base en **0x08047000** y de un tamaño de **0x1000** bytes. Todo lo demás es igual al original,solo ha modificado la MemSize del **LOAD** original en 0x1000 bytes, seguramente para que el alineamiento sea correcto.

Si vemos el archivo **cm1.ok** con un editor hexadecimal lo entendemos todo:



Si os fijáis, aquí solo tenemos la cabecera **ELF** y las 4 entradas de la cabecera del programa; estando el resto del archivo ocupado por **0x41** (A). De este modo toda la parte dinámica esta en el ELF antiguo, sin problemas de direcciones virtuales, pues se carga en memoria a partir de **0x08048000**; y queda todo como en el original, como vemos con **readelf** y la opción **-d**:

```
            $ readelf -d cm1.ok

Dynamic section at offset 0x10a8 contains 10 entries:
  Tag        Type                         Name/Value
 0x00000001 (NEEDED)                     Shared library: [libc.so.6]
 0x00000004 (HASH)                       0x804808c
 0x00000005 (STRTAB)                     0x804816a
 0x00000006 (SYMTAB)                     0x80480f0
 0x0000000a (STRSZ)                      40 (bytes)
 0x0000000b (SYMENT)                     16 (bytes)
 0x00000011 (REL)                        0x8048140
 0x00000012 (RELSZ)                      24 (bytes)
 0x00000013 (RELENT)                     8 (bytes)
 0x00000000 (NULL)                       0x0
```

Con este archivo, vamos a ver como se realiza la vinculación dinámica en directo y si podemos intentaremos solucionar el crackme.

3.2) Strace; con este programa vemos las llamadas al sistema, y en este caso me interesa comprobar en que punto comienza la ejecución del programa y sus llamadas al sistema a diferencia de las ejecutadas por el sistema al cargar el programa. Como siempre la orden es:

```
$ strace -i -o strace.txt ./cm1
```

Con estos datos y el debugger con la orden "**catch syscall**" (sin ningún valor para en todas las llamadas al sistema) pude comprobar que el programa comenzaba después de la llamada a **munmap** (*munmap - ubica o elimina ficheros o dispositivos en memoria*):

```
[    7f0161af9b37] execve("./cm1", ["./cm1"], [/* 34 vars */]) = 0
[        f77b06bd] brk(0)                  = 0x98f6000
[        f77b18c1] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file
or directory)
[        f77b1a03] mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xfffffffff7798000
[        f77b18c1] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file
or directory)
[        f77b1784] open("/etc/ld.so.cache", O_RDONLY) = 3
[        f77b174e] fstat64(3, {st_mode=S_IFREG|0644, st_size=104943, ...}) = 0
[        f77b1a03] mmap2(NULL, 104943, PROT_READ, MAP_PRIVATE, 3, 0) =
0xfffffffff777e000
[        f77b17bd] close(3)                = 0
[        f77b18c1] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file
or directory)
[        f77b1784] open("/lib32/libc.so.6", O_RDONLY) = 3
[        f77b1804] read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320m\1\0004\0\0\0"..., 512) =
512
[        f77b174e] fstat64(3, {st_mode=S_IFREG|0755, st_size=1327556, ...}) = 0
[        f77b1a03] mmap2(NULL, 1337704, PROT_READ|PROT_EXEC, MAP_PRIVATE|
MAP_DENYWRITE, 3, 0) = 0xfffffffff7637000
[        f77b1a84] mprotect(0xf7777000, 4096, PROT_NONE) = 0
[        f77b1a03] mmap2(0xf7778000, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED|MAP_DENYWRITE, 3, 0x140) = 0xfffffffff7778000
[        f77b1a03] mmap2(0xf777b000, 10600, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xfffffffff777b000
[        f77b17bd] close(3)                = 0
[        f77b1a03] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xfffffffff7636000
[        f779d2df] set_thread_area(0xff8592cc) = 0
[        f77b1a84] mprotect(0xf7778000, 8192, PROT_READ) = 0
[        f77b1a84] mprotect(0xf77b7000, 4096, PROT_READ) = 0
[        f77b1a41] munmap(0xf777e000, 104943) = 0
Comienza la ejecucion del programa desde el entrypoint 0x804819f
[        f779a430] write(1, "\n\n\tCrackME (v1 Linux) (CM11) by "..., 118) =
118
[        f779a430] fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1
), ...}) = 0
[        f779a430] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xfffffffff7797000
[        f779a430] read(0, "12345\n", 1024) = 6
[        f779a430] write(1, "\n\n\tBAD PASSWORD :P\n\n", 20) = 20
[        f779a430] exit_group(4151114644) = ?
```

Mi idea era parar el programa antes de llegar al **entrypoint**, por si ejecutaba código antes y pensé en encontrar una syscall que me dejase cerca de la zona que me interesaba. No siempre será esta

llamada al sistema **munmap** la última; pero de momento nos quedamos con esta idea por si nos sirve en el futuro.

3.3) GDB.

Cargamos el programa **cm1.ok** en **gdb** y vemos que no nos da ningún aviso de mal formato:

```
            $ gdb -q cm1.ok
Really redefine built-in command "frame"? (y or n) [answered Y; input not from
terminal]
Really redefine built-in command "thread"? (y or n) [answered Y; input not from
terminal]
Really redefine built-in command "start"? (y or n) [answered Y; input not from
terminal]
Reading symbols from /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1.ok...(no
debugging symbols found)...done.
gdb>  b *0x804819f
Breakpoint 1 at 0x804819f
```

Como estoy utilizando el **dotgdbinit** y ha redefinido algunos comandos, me salen estos avisos sin importancia. Lo primero es colocar un breakpoint en el **EntryPoint** en 0x0804819F y comenzar con la opción **"r"**:

```
gdb> r
─────────────────────────────────────────────────────────────────────────────
     eax:F7FFD900 ebx:F7FFCFF4  ecx:F7FF83CD  edx:F7FEEB60     eflags:00000282
     esi:FFFFD4BC edi:0804819F  esp:FFFFD4B0  ebp:00000000     eip:0804819F
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t S z a p c
[002B:FFFFD4B0]-------------------------------------------------------[stack]
FFFFD4E0 : A1 D7 FF FF  C1 D7 FF FF - CD D7 FF FF  BD DC FF FF ................
FFFFD4D0 : DA D6 FF FF  2A D7 FF FF - 65 D7 FF FF  77 D7 FF FF ....*...e...w...
FFFFD4C0 : 75 D6 FF FF  88 D6 FF FF - BF D6 FF FF  CF D6 FF FF u...............
FFFFD4B0 : 01 00 00 00  1E D6 FF FF - 00 00 00 00  53 D6 FF FF ............S...
[002B:FFFFD4BC]-------------------------------------------------------[ data]
FFFFD4BC : 53 D6 FF FF  75 D6 FF FF - 88 D6 FF FF  BF D6 FF FF S...u...........
FFFFD4CC : CF D6 FF FF  DA D6 FF FF - 2A D7 FF FF  65 D7 FF FF ........*...e...
[0023:0804819F]-------------------------------------------------------[ code]
0x804819f:  pop    esi
0x80481a0:  pop    ebx
0x80481a1:  mov    ebx,0x80483d0
0x80481a6:  mov    ebp,0x8048192
0x80481ab:  push   0x0
0x80481ad:  push   DWORD PTR [ebx+0x1c]
------------------------------------------------------------------------------

Breakpoint 1, 0x0804819f in ?? ()
```

Ya lo tenemos parado en el breakpoint, vamos a continuar con **"ni"** hasta llegar a la primera **Call**:

```
─────────────────────────────────────────────────────────────────────────────
     eax:F7FFD900 ebx:080483D0  ecx:F7FF83CD  edx:F7FEEB60     eflags:00000246
     esi:08048326 edi:00000000  esp:FFFFD4A4  ebp:08048192     eip:080481C3
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s Z a P c
[002B:FFFFD4A4]-------------------------------------------------------[stack]
FFFFD4D4 : 2A D7 FF FF  65 D7 FF FF - 77 D7 FF FF  A1 D7 FF FF *...e...w.......
FFFFD4C4 : 88 D6 FF FF  BF D6 FF FF - CF D6 FF FF  DA D6 FF FF ................
FFFFD4B4 : 00 00 00 00  00 00 00 00 - 53 D6 FF FF  75 D6 FF FF ........S...u...
FFFFD4A4 : 01 00 00 00  26 83 04 08 - 76 00 00 00  94 4B F1 F7 ....&...v....K..
[002B:08048326]-------------------------------------------------------[ data]
```

```
08048326 : 0A 0A 09 43  72 61 63 6B - 4D 45 20 28  76 31 20 4C ...CrackME (v1 L
08048336 : 69 6E 75 78  29 20 28 43 - 4D 31 31 29  20 62 79 20 inux) (CM11) by
[0023:080481C3]-------------------------------------------------------------[ code]
0x80481c3:  call   DWORD PTR [ebx+0xc]
0x80481c6:  add    esp,0xc
0x80481c9:  dec    eax
0x80481ca:  cmp    edi,0x13
0x80481d0:  jne    0x80481d3
0x80481d2:  ret
--------------------------------------------------------------------------------
0x080481c3 in ?? ()
```

Nos encontramos con una llamada a un valor localizado en **EBX** mas un desplazamiento, en este caso **0x80483D0** + **0xC**, que nos llevará a la dirección que se encuentre en 0x80483DC.

Este valor ya lo hemos visto en la reubicación:

```
Offset      Info     Type              Sym.Value   Sym. Name
080483dc  00000401 R_386_32           00000004      write
080483d8  00000301 R_386_32           00000003     getchar
080483ec  00000201 R_386_32           00000002      _exit
```

Por tanto estamos ante una llamada hacía una función externa del ejecutable, que en este caso al final nos llevará a una llamada al sistema. Antes de ejecutarse el programa, el enlazador dinámico se ha encontrado con todo el sistema de **Relocation** y la **tabla de símbolos** y ha colocado lo necesario en los puntos de reubicación, en este caso un tipo **R_386_32**. Este tipo indica una reubicación directa a una función de 32 (no hay tabla de saltos) y por el valor de **st_shndx**, **UNDEF**, de la tabla de símbolos, sabemos que le corresponde un símbolo no definido y por tanto el cargador lo enlazará directamente con su función correspondiente.

Si miramos la zona indicada por EBX con la orden **"x"** podemos ver en que consiste:

```
gdb> x/10w $ebx
0x80483d0:  0x00000000  0x00000000  0xf7edc3e0  0xf7f3b030
0x80483e0:  0x00000000  0x00000000  0x00000000  0xf7f14b94
0x80483f0:  0x00000000  0x00000000
```

Por tanto **0x80483DC** corresponde con **0xf7f3b030** , que sera donde nos lleve la **Call** en **0x80481c3**; que vamos a seguir con la orden **"si"**, antes para estar seguro de no pasarnos vamos a poner un breakpoint en la zona de retorno 0x80481c6:

```
gdb> b *0x80481c6
Breakpoint 2 at 0x80481c6
gdb> si
-------------------------------------------------------------------------------
     eax:F7FFD900 ebx:080483D0  ecx:F7FF83CD  edx:F7FEEB60     eflags:00000246
     esi:08048326 edi:00000000  esp:FFFFD4A0  ebp:08048192     eip:F7F3B030
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s Z a P c
[002B:FFFFD4A0]-------------------------------------------------------------[stack]
FFFFD4D0 : D9 D6 FF FF  29 D7 FF FF - 64 D7 FF FF  77 D7 FF FF ....)...d...w...
FFFFD4C0 : 74 D6 FF FF  87 D6 FF FF - BE D6 FF FF  CE D6 FF FF t...............
FFFFD4B0 : 94 4B F1 F7  00 00 00 00 - 00 00 00 00  52 D6 FF FF .K..........R...
FFFFD4A0 : C6 81 04 08  01 00 00 00 - 26 83 04 08  76 00 00 00 ........&...v...
[002B:08048326]-------------------------------------------------------------[ data]
08048326 : 0A 0A 09 43  72 61 63 6B - 4D 45 20 28  76 31 20 4C ...CrackME (v1 L
08048336 : 69 6E 75 78  29 20 28 43 - 4D 31 31 29  20 62 79 20 inux) (CM11) by
```

```
[0023:F7F3B030]-------------------------------------------------------[ code]
0xf7f3b030: cmp    DWORD PTR gs:0xc,0x0
0xf7f3b038: jne    0xf7f3b05c
0xf7f3b03a: push   ebx
0xf7f3b03b: mov    edx,DWORD PTR [esp+0x10]
0xf7f3b03f: mov    ecx,DWORD PTR [esp+0xc]
0xf7f3b043: mov    ebx,DWORD PTR [esp+0x8]
------------------------------------------------------------------------------
0xf7f3b030 in ?? ()
```

Ahora debemos estar en la función **write** externa, normalmente **gdb** debería informarnos sobre a que librería le corresponde este código; pero no es el caso (0xf7f3b030 in ??); por tanto vamos utilizar un programita que me ha sido muy útil, **pmap**, el cual nos informa del mapa de memoria de un proceso; para usarlo necesitamos el identificador del proceso **PID** pues el proceso debe estar en ejecución, para ello utilizamos la orden **"ps ax"** con el filtro **grep**:

```
      $ ps ax | grep cm1.ok
 9944 pts/1    S+     0:00 gdb -q cm1.ok
10334 pts/1    T      0:00 /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1.ok
12067 pts/2    S+     0:00 grep cm1.ok
```

De esta manera tenemos que el PID de **cm1.ok** es **10334**, que es el único argumento que necesita **pmap**:

```
            $ pmap 10334
10334:   /home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1.ok
0000000008047000      4K --x--
/home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1.ok
0000000008048000      4K rwx--
/home/juanjo/Descargas/crk/crackmes_linux/cm1/cm1.ok
00000000f7e7b000      4K rwx--    [ anon ]
00000000f7e7c000   1280K r-x--  /lib32/libc-2.11.2.so
00000000f7fbc000      4K -----  /lib32/libc-2.11.2.so
00000000f7fbd000      8K r-x--  /lib32/libc-2.11.2.so
00000000f7fbf000      4K rwx--  /lib32/libc-2.11.2.so
00000000f7fc0000     12K rwx--    [ anon ]
00000000f7fdd000      8K rwx--    [ anon ]
00000000f7fdf000      4K r-x--    [ anon ] ---------->linux.gate.so.1
00000000f7fe0000    112K r-x--  /lib32/ld-2.11.2.so
00000000f7ffc000      4K r-x--  /lib32/ld-2.11.2.so
00000000f7ffd000      4K rwx--  /lib32/ld-2.11.2.so
00000000fffe9000     84K rwx--    [ stack ]
 total             1536K
```

La dirección 0xF7F3B030 corresponde a **/lib32/libc-2.11.2.so**; por tanto el cargador dinámico teniendo en cuenta la zona de reubicación (**080483dc**) y la tabla de símbolos (**write**) ha unido el programa con este código en **libc**. En concreto, debido al valor índice de la sección, **st_shndx** que tenia valor 0, **SHN_UNDEF,** sabemos que tiene que unirlo a la misma función **write** de **/lib32/libc-2.11.2.so**. Vamos a ver si coincide, para ello utilizamos **objdump** con la opción **-T**:

```
$ objdump -T /lib32/libc-2.11.2.so | grep write
00065720 g    DF .text  0000014d  GLIBC_2.2   _IO_wdo_write
000bf030  w   DF .text  0000007a  GLIBC_2.0   __write
00108fa0 g    DF .text  00000033 (GLIBC_2.0)  _IO_do_write
00067ae0 g    DF .text  00000033  GLIBC_2.1   _IO_do_write
000a58d0  w   DF .text  000000c6  GLIBC_2.1   __pwrite64
000c61a0  w   DF .text  000000aa  GLIBC_2.0   writev
000a5730 g    DF .text  000000cd  GLIBC_PRIVATE __libc_pwrite
```

```
000c6890 g    DF .text  00000107  GLIBC_2.10  pwritev
000ce3c0 g    DF .text  0000002e  GLIBC_2.7   eventfd_write
0005d690  w   DF .text  00000179  GLIBC_2.0   fwrite
000c6af0 g    DF .text  000000f1  GLIBC_2.10  pwritev64
00067b20 g    DF .text  000000a2  GLIBC_2.1   _IO_file_write
00109090 g    DF .text  0000006f (GLIBC_2.0)  _IO_file_write
0005d690 g    DF .text  00000179  GLIBC_2.0   _IO_fwrite
000a5730  w   DF .text  000000cd  GLIBC_2.1   pwrite
000626a0 g    DF .text  000000ca  GLIBC_2.1   fwrite_unlocked
000a58d0  w   DF .text  000000c6  GLIBC_2.1   pwrite64
000bf030  w   DF .text  0000007a  GLIBC_2.0   write
```

Si tenemos en cuenta que con **pmap** sabemos que la imagen base de **libc-2.11.2.so** es 0x0f7e7c000 (es la entrada de mas tamaño) y le añadimos el desplazamiento de la función **write** 0x0bf030, nos encontramos con la dirección de reubicación 0xF7F3B030 que es la función **write** de libc:

```
>>> hex (0xf7e7c000 + 0xbf030)
'0xf7f3b030'
```

Podemos ir mas lejos, si tenemos en cuenta que es una llamada al sistema y como vimos al principio este programa utiliza **linux-gate.so.1**; podemos poner un punto de interrupción (**"b"**) en la función **__kernel_vsyscall** para comprobarlo. En este caso, como no he reiniciado el programa, su dirección no se ha modificado, por lo que si ponemos un breakpoint en la dirección de **sysenter** comprobaremos que estamos en lo correcto:

```
gdb> b *0xf7fdf425
Breakpoint 3 at 0xf7fdf425
gdb> x/i 0xf7fdf425
0xf7fdf425 <__kernel_vsyscall+5>:   sysenter
```

Después de comprobar que el breakpoint esta en la zona correcta con **"x/i"**, continuamos con **"c"**:

```
gdb> c
──────────────────────────────────────────────────────────────────────────
     eax:00000004 ebx:00000001  ecx:08048326  edx:00000076     eflags:00000246
     esi:08048326 edi:00000000  esp:FFFFD48C  ebp:FFFFD48C     eip:F7FDF425
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s Z a P c
[002B:FFFFD48C]-------------------------------------------------------[stack]
FFFFD4BC : 52 D6 FF FF  74 D6 FF FF - 87 D6 FF FF  BE D6 FF FF R...t...........
FFFFD4AC : 76 00 00 00  94 4B F1 F7 - 00 00 00 00  00 00 00 00 v....K..........
FFFFD49C : D0 83 04 08  C6 81 04 08 - 01 00 00 00  26 83 04 08 ............&...
FFFFD48C : 92 81 04 08  76 00 00 00 - 26 83 04 08  53 B0 F3 F7 ....v...&...S...
[002B:08048326]-------------------------------------------------------[ data]
08048326 : 0A 0A 09 43  72 61 63 6B - 4D 45 20 28  76 31 20 4C ...CrackME (v1 L
08048336 : 69 6E 75 78  29 20 28 43 - 4D 31 31 29  20 62 79 20 inux) (CM11) by
[0023:F7FDF425]-------------------------------------------------------[ code]
0xf7fdf425 <__kernel_vsyscall+5>:   sysenter
0xf7fdf427 <__kernel_vsyscall+7>:   nop
0xf7fdf428 <__kernel_vsyscall+8>:   nop
0xf7fdf429 <__kernel_vsyscall+9>:   nop
0xf7fdf42a <__kernel_vsyscall+10>:  nop
0xf7fdf42b <__kernel_vsyscall+11>:  nop
--------------------------------------------------------------------------

Breakpoint 3, 0xf7fdf425 in __kernel_vsyscall ()
```

Si miramos el archivo **unistd_32.h** vemos que estamos en el sitio correcto, pues el valor de **EAX** es **4**:

```
#define __NR_write          4
```

Como ya sabemos, en **EBX** esta el descriptor **1** que corresponde al archivo de salida estandar, en este caso la pantalla; en **ECX** está la zona de buffer, que estamos viendo en la zona de **[data]** y por último en **EDX** la cantidad de bytes que nos va a mostrar.

Si seguimos con **"c"**, vemos que aparece en el terminal el texto donde nos pide el password y queda parado en la zona de retorno:

```
                Write Your
Password:_____________________________________________________________________
_________
     eax:00000076 ebx:080483D0  ecx:08048326  edx:00000076     eflags:00000203
     esi:08048326 edi:00000000  esp:FFFFD4A4  ebp:08048192     eip:080481C6
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s z a p C
[002B:FFFFD4A4]-----------------------------------------------------------[stack]
FFFFD4D4 : 29 D7 FF FF  64 D7 FF FF - 77 D7 FF FF  A1 D7 FF FF )...d...w.......
FFFFD4C4 : 87 D6 FF FF  BE D6 FF FF - CE D6 FF FF  D9 D6 FF FF ................
FFFFD4B4 : 00 00 00 00  00 00 00 00 - 52 D6 FF FF  74 D6 FF FF ........R...t...
FFFFD4A4 : 01 00 00 00  26 83 04 08 - 76 00 00 00  94 4B F1 F7 ....&...v....K..
[002B:08048326]-----------------------------------------------------------[ data]
08048326 : 0A 0A 09 43  72 61 63 6B - 4D 45 20 28  76 31 20 4C ...CrackME (v1 L
08048336 : 69 6E 75 78  29 20 28 43 - 4D 31 31 29  20 62 79 20 inux) (CM11) by
[0023:080481C6]-----------------------------------------------------------[ code]
0x80481c6:  add    esp,0xc
0x80481c9:  dec    eax
0x80481ca:  cmp    edi,0x13
0x80481d0:  jne    0x80481d3
0x80481d2:  ret
0x80481d3:  lea    edi,[ebx+0x20]
--------------------------------------------------------------------------------

Breakpoint 2, 0x080481c6 in ?? ()
```

Si continuamos con **"ni"** llegaremos a la siguiente función reubicada:

```
_____________________________________________________________________________
     eax:00000075 ebx:080483D0  ecx:08048326  edx:00000076     eflags:00000297
     esi:00000012 edi:080483F0  esp:FFFFD4B0  ebp:08048192     eip:080481D9
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t S z A P C
[002B:FFFFD4B0]-----------------------------------------------------------[stack]
FFFFD4E0 : A1 D7 FF FF  C1 D7 FF FF - CD D7 FF FF  BD DC FF FF ................
FFFFD4D0 : D9 D6 FF FF  29 D7 FF FF - 64 D7 FF FF  77 D7 FF FF ....)...d...w...
FFFFD4C0 : 74 D6 FF FF  87 D6 FF FF - BE D6 FF FF  CE D6 FF FF t...............
FFFFD4B0 : 94 4B F1 F7  00 00 00 00 - 00 00 00 00  52 D6 FF FF .K..........R...
[002B:080483F0]-----------------------------------------------------------[ data]
080483F0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
08048400 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
[0023:080481D9]-----------------------------------------------------------[ code]
0x80481d9:  call   DWORD PTR [ebx+0x8]
0x80481dc:  or     eax,eax
0x80481de:  js     0x804823e
0x80481e0:  cmp    al,0x20
0x80481e2:  je     0x80481d9
0x80481e4:  cmp    al,0x9
--------------------------------------------------------------------------------
0x080481d9 in ?? ()
```

Si os fijáis todas estas llamadas tienen el mismo formato, comprobamos que el valor que señala

EBX+0x8 es 0x80483d8 y nos señala a una función en **libc** como hemos visto antes:

```
gdb> x/w $ebx+0x8
0x80483d8:  0xf7edc3e0
```

Por tanto, si tenemos en cuenta la reubicación:

080483d8 00000301 R_386_32 00000003 **getchar**

Esta es la llamada a **getchar,** para comprobarlo entramos con "**si**"; no sin antes poner un breakpoint en la zona de retorno **0x80481dc**:

```
gdb> b *0x80481dc
Breakpoint 4 at 0x80481dc
gdb> si
─────────────────────────────────────────────────────────────────────────────
     eax:00000075 ebx:080483D0  ecx:08048326  edx:00000076     eflags:00000297
     esi:00000012 edi:080483F0  esp:FFFFD4AC  ebp:08048192     eip:F7EDC3E0
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t S z A P C
[002B:FFFFD4AC]-------------------------------------------------------[stack]
FFFFD4DC : 77 D7 FF FF  A1 D7 FF FF - C1 D7 FF FF  CD D7 FF FF w...............
FFFFD4CC : CE D6 FF FF  D9 D6 FF FF - 29 D7 FF FF  64 D7 FF FF ........)...d...
FFFFD4BC : 52 D6 FF FF  74 D6 FF FF - 87 D6 FF FF  BE D6 FF FF R...t...........
FFFFD4AC : DC 81 04 08  94 4B F1 F7 - 00 00 00 00  00 00 00 00 .....K..........
[002B:080483F0]-------------------------------------------------------[ data]
080483F0 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
08048400 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
[0023:F7EDC3E0]-------------------------------------------------------[ code]
0xf7edc3e0: push   ebp
0xf7edc3e1: mov    ebp,esp
0xf7edc3e3: sub    esp,0x14
0xf7edc3e6: mov    DWORD PTR [ebp-0xc],ebx
0xf7edc3e9: call   0xf7e92aaf
0xf7edc3ee: add    ebx,0xe2c06
-------------------------------------------------------------------------------
0xf7edc3e0 in ?? ()
```

De nuevo estamos en **/lib32/libc-2.11.2.so** ; ahora debemos saber si es la función **getchar**. Para ello hay que localizar la función en **libc**, ver el desplazamiento, añadírselo a la image base que tenemos con **pmap** y así obtenemos la dirección de esta función (con lo fácil que es con Olly.jeje...).

Primero utilizaremos **objdump** con la opción **-T** para ver donde está la función **getchar**:

```
           $ objdump -T /lib32/libc.so.6 | grep getchar
00062460 g    DF .text  00000039  GLIBC_2.0   getchar_unlocked
000603e0 g    DF .text  000000f7  GLIBC_2.0   getchar
```

No os preocupéis por el nombre de la libreria C, dentro de **/lib32/** es la misma , pues libc.so.6 es un enlace a libc-2.11.2.so:

```
           $ objdump -T /lib32/libc-2.11.2.so | grep getchar
00062460 g    DF .text  00000039  GLIBC_2.0   getchar_unlocked
000603e0 g    DF .text  000000f7  GLIBC_2.0   getchar
```

Ahora debemos encontrar la image base de esta librería en el proceso con **pmap**, aunque hay varias entradas, debemos tener en cuenta que el archivo completo solo es una, en concreto la de mayor tamaño:

00000000f7e7c000 1280K r-x-- /lib32/libc-2.11.2.so

Por tanto con python obtenemos la dirección correcta, sumando a la image base el desplazamiento:

>>> hex(0xf7e7c000+0x603e0)
'0xf7edc3e0'

Por tanto el cargador dinámico ha colocado la dirección de la función **getchar** ,'0xf7edc3e0', en la reubicación ya que se encontró un valor índice de la sección **st_shndx** como **SHN_UNDEF** que se lo indicaba.

Como **getchar** es una función que permite leer un carácter desde el teclado sin que se muestre en pantalla,vamos a colocarle un breakpoint y la veremos en acción:

```
gdb> b *0xf7edc3e0
Breakpoint 5 at 0xf7edc3e0
gdb> c
123456
```

Al continuar el programa queda esperando nuestro serial, se lo ponemos y ahora si funciona el breakpoint en **getchar**:

```
─────────────────────────────────────────────────────────────────────────
     eax:00000031 ebx:080483D0  ecx:00000031  edx:F7FC0358     eflags:00000206
     esi:00000011 edi:080483F1  esp:FFFFD4AC  ebp:08048192     eip:F7EDC3E0
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    o d I t s z a P c
[002B:FFFFD4AC]-------------------------------------------------------[stack]
FFFFD4DC : 77 D7 FF FF  A1 D7 FF FF - C1 D7 FF FF  CD D7 FF FF w...............
FFFFD4CC : CE D6 FF FF  D9 D6 FF FF - 29 D7 FF FF  64 D7 FF FF ........)...d...
FFFFD4BC : 52 D6 FF FF  74 D6 FF FF - 87 D6 FF FF  BE D6 FF FF R...t...........
FFFFD4AC : F3 81 04 08  94 4B F1 F7 - 00 00 00 00  00 00 00 00 .....K..........
[002B:080483F1]-----------------------------------------------------[ data]
080483F1 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
08048401 : 00 00 00 00  00 00 00 00 - 00 00 00 00  00 00 00 00 ................
[0023:F7EDC3E0]-----------------------------------------------------[ code]
0xf7edc3e0: push   ebp
0xf7edc3e1: mov    ebp,esp
0xf7edc3e3: sub    esp,0x14
0xf7edc3e6: mov    DWORD PTR [ebp-0xc],ebx
0xf7edc3e9: call   0xf7e92aaf
0xf7edc3ee: add    ebx,0xe2c06
--------------------------------------------------------------------------

Breakpoint 5, 0xf7edc3e0 in ?? ()
```

Como vemos, ha tomado el primer valor **0x31** (1) y si continuamos ira tomando el resto de valores hasta completar el serial que hemos introducido.

Una vez que volvemos al punto de retorno de esta función, **0x80481dc**, continuamos con **"ni"** y nos encontramos con la cantidad de operaciones que le realiza al serial (las veremos después); como esta vez tampoco hemos dado con el password correcto, pasamos otra vez por la función **write** (BAD PASSWORD :P) y llegaremos a la última reubicación.

Por cierto en el camino saltará varias veces el breakpoint de **sysenter**; para evitar esto sin quitarlo

esta la opción de **gdb "disable"** que seguida del numero de breakpoint lo inhabilita hasta que nos interese:

```
Breakpoint 3, 0xf7fdf425 in __kernel_vsyscall ()
gdb> disable 3
```

Lo comprobamos con la orden **"info break"**:

```
gdb> info break
Num     Type           Disp Enb Address    What
1       breakpoint     keep y   0x080481c3
        breakpoint already hit 1 time
2       breakpoint     keep y   0x080481c6
3       breakpoint     keep n   0xf7fdf425 <__kernel_vsyscall+5>
        breakpoint already hit 1 time
4       breakpoint     keep y   0x080481dc
6       breakpoint     keep y   0x080481c6
7       breakpoint     keep y   0x0804823f
```

Si te fijas en la columna "**Enb**" (enable) aparece como "**n**" (not enable). Para volver a usarlo se usa la orden **"enable"** mas el número de breakpoint.

Seguimos y ya sin interrupciones llegamos cerca de la última llamada cuando se va a cerrar el programa:

```
──────────────────────────────────────────────────────────────────────────────
     eax:E7E00EF5 ebx:080483D0  ecx:0000000F  edx:00000014     eflags:00000A02
     esi:0804839C edi:00000013  esp:FF9C977C  ebp:08048192     eip:08048238
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    O d I t s z a p c
[002B:FF9C977C]-------------------------------------------------------[stack]
FF9C97AC : D5 B7 9C FF  FF B7 9C FF - 1F B8 9C FF  2B B8 9C FF .............+...
FF9C979C : 27 B7 9C FF  37 B7 9C FF - 87 B7 9C FF  C2 B7 9C FF '...7...........
FF9C978C : B0 B6 9C FF  D2 B6 9C FF - E5 B6 9C FF  1C B7 9C FF ................
FF9C977C : 00 00 00 00  94 5B 64 F7 - 00 00 00 00  00 00 00 00 .....[d.........
[002B:0804839C]-------------------------------------------------------[ data]
0804839C : 0A 0A 09 42  41 44 20 50 - 41 53 53 57  4F 52 44 20 ...BAD PASSWORD
080483AC : 3A 50 0A 0A  0A 0A 09 4F - 4B 20 43 72  61 63 6B 65 :P.....OK Cracke
[0023:08048238]-------------------------------------------------------[ code]
0x8048238:  push   DWORD PTR [ebx+0x1c]
0x804823b:  push   edx
0x804823c:  jmp    0x80481c0
0x804823e:  push   eax
0x804823f:  call   DWORD PTR [ebx+0x1c]
0x8048242:  push   ebx
------------------------------------------------------------------------------
0x08048238 in ?? ()
gdb>
```

El problema es que esta llamada a **_exit** nunca se ejecuta, pues en **0x804823c** hay un salto incondicional que lleva a una llamada al sistema **_exit_group** sin pasar por la reubicación.
Pero bueno, todo tiene solución, podemos nopear ese salto incondicional y así llegaremos a la última reubicación sin problemas:

```
gdb> set {short}0x804823c=0x9090
```

Con esta orden cambiamos los dos bytes (usamos short) de **0x804823c** por dos instrucciones **nop** , que al no hacer nada, son perfectas para eliminar una instrucción. Lo comprobamos con la orden **x/i**

y vemos que todos ha quedado bien :

```
     gdb> x/6i $pc
0x804823b:  push   edx
0x804823c:  nop
0x804823d:  nop
0x804823e:  push   eax
0x804823f:  call   DWORD PTR [ebx+0x1c]
0x8048242:  push   ebx
```

Vamos a continuar hasta la llamada en **0x804823f** con **"ni"**:

```
-------------------------------------------------------------------------
     eax:E7E00EF5 ebx:080483D0  ecx:0000000F  edx:00000014     eflags:00000A02
     esi:0804839C edi:00000013  esp:FFC73260  ebp:08048192     eip:0804823F
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    O d I t s z a p c
[002B:FFC73260]-----------------------------------------------------------[stack]
FFC73290 : 37 47 C7 FF  87 47 C7 FF - C2 47 C7 FF  D5 47 C7 FF 7G...G...G...G..
FFC73280 : D2 46 C7 FF  E5 46 C7 FF - 1C 47 C7 FF  27 47 C7 FF .F...F...G..'G..
FFC73270 : 94 9B 68 F7  00 00 00 00 - 00 00 00 00  B0 46 C7 FF ..h..........F..
FFC73260 : F5 0E E0 E7  14 00 00 00 - 94 9B 68 F7  00 00 00 00 ..........h.....
[002B:0804839C]-----------------------------------------------------------[ data]
0804839C : 0A 0A 09 42  41 44 20 50 - 41 53 53 57  4F 52 44 20 ...BAD PASSWORD
080483AC : 3A 50 0A 0A  0A 0A 09 4F - 4B 20 43 72  61 63 6B 65 :P.....OK Cracke
[0023:0804823F]-----------------------------------------------------------[ code]
0x804823f:  call   DWORD PTR [ebx+0x1c]
0x8048242:  push   ebx
0x8048243:  mov    edi,esi
0x8048245:  mov    ecx,DWORD PTR [edi]
0x8048247:  mov    ebx,DWORD PTR [edi+0x4]
0x804824a:  xor    edi,edi
-------------------------------------------------------------------------

Breakpoint 2, 0x0804823f in ?? ()
```

Recordamos la reubicación:

```
080483ec  00000201 R_386_32          00000002      _exit
```

En este caso **EBX+0x1C** corresponde a **0x80483EC**, si entramos en la función nos debe llevar a 0xf7689b94:

```
-------------------------------------------------------------------------
     eax:E7E00EF5 ebx:080483D0  ecx:0000000F  edx:00000014     eflags:00000A02
     esi:0804839C edi:00000013  esp:FFC7325C  ebp:08048192     eip:F7689B94
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    O d I t s z a p c
[002B:FFC7325C]-----------------------------------------------------------[stack]
FFC7328C : 27 47 C7 FF  37 47 C7 FF - 87 47 C7 FF  C2 47 C7 FF 'G..7G...G...G..
FFC7327C : B0 46 C7 FF  D2 46 C7 FF - E5 46 C7 FF  1C 47 C7 FF .F...F...F...G..
FFC7326C : 00 00 00 00  94 9B 68 F7 - 00 00 00 00  00 00 00 00 ......h.........
FFC7325C : 42 82 04 08  F5 0E E0 E7 - 14 00 00 00  94 9B 68 F7 B.............h.
[002B:0804839C]-----------------------------------------------------------[ data]
0804839C : 0A 0A 09 42  41 44 20 50 - 41 53 53 57  4F 52 44 20 ...BAD PASSWORD
080483AC : 3A 50 0A 0A  0A 0A 09 4F - 4B 20 43 72  61 63 6B 65 :P.....OK Cracke
[0023:F7689B94]-----------------------------------------------------------[ code]
0xf7689b94: mov    ebx,DWORD PTR [esp+0x4]
0xf7689b98: mov    eax,0xfc
0xf7689b9d: call   DWORD PTR gs:0x10
0xf7689ba4: mov    eax,0x1
```

```
0xf7689ba9: int    0x80
0xf7689bab: hlt
--------------------------------------------------------------------------------
0xf7689b94 in ?? ()
```

Estamos en la **libc**, según **pmap** le corresponde esta dirección de memoria:

```
00000000f75f1000   1280K r-x--  /lib32/libc-2.11.2.so
```

Para los observadores, podéis comprobar que la dirección base de **libc-2.11.2.so** ha cambiado en esta última parte del tutorial, al tener que reiniciar porque no llegaba a esta reubicación, hemos podido comprobar que las direcciones de memoria van cambiando aleatoriamente.

Ahora vamos a ver si se corresponde con la función **_exit** de **libc-2.11.2.so**, para ello la buscamos con **objdump -T**:

```
     $ objdump -T /lib32/libc-2.11.2.so | grep exit
0002fa20 g    DF .text  0000003c  GLIBC_2.10  __cxa_at_quick_exit
0002f5c0 g    DF .text  0000002f  GLIBC_2.0   exit
00098b94 g    DF .text  00000018  GLIBC_2.0   _exit
000f5f30 g    DF .text  00000043  GLIBC_2.0   svc_exit
0002f9f0 g    DF .text  0000002f  GLIBC_2.10  quick_exit
0002f830 g    DF .text  0000003b  GLIBC_2.1.3 __cxa_atexit
00107bb0 g    DF .text  0000003b (GLIBC_2.0)  atexit
00143164 g    DO .data  00000004  GLIBC_2.1   argp_err_exit_status
000db190 g    DF .text  00000046  GLIBC_2.0   pthread_exit
001430cc g    DO .data  00000004  GLIBC_2.0   obstack_exit_failure
0002f5f0  w   DF .text  00000058  GLIBC_2.0   on_exit
000e0950 g    DF .text  00000005  GLIBC_2.2   __cyg_profile_func_exit
```

Por tanto, el desplazamiento es **0x98b94** y la image base **0x0f75f1000**:

```
>>> hex (0x0f75f1000 +  0x98b94)
'0xf7689b94'
```

Obtenemos la dirección donde nos encontramos ahora mismo **0xF7689B94.**

También la dirección de **linux-gate.so.1** ha cambiado y si intentamos acceder a la dirección que antes habíamos utilizado, nos da un error de acceso:

```
gdb> x/i 0xf7fdf425
0xf7fdf425: Cannot access memory at address 0xf7fdf425
```

Curiosamente, no hace falta buscarla pues en la dirección **0xf7689b9d**, tenemos una llamada de la forma" Call DWORD PTR gs:0x10", en la cual si entramos nos encontramos en la misma zona de **linux-gate**; y podemos llegar hasta **sysenter** y ver que llamada al sistema ejecuta:

```
-------------------------------------------------------------------
     eax:000000FC ebx:E7E00EF5  ecx:0000000F  edx:00000014     eflags:00000A02
     esi:0804839C edi:00000013  esp:FFC73258  ebp:08048192     eip:F7754420
     cs:0023  ds:002B  es:002B  fs:0000  gs:0063  ss:002B    O d I t s z a p c
[002B:FFC73258]-----------------------------------------------------[stack]
FFC73288 : 1C 47 C7 FF  27 47 C7 FF - 37 47 C7 FF  87 47 C7 FF .G..'G..7G...G..
FFC73278 : 00 00 00 00  B0 46 C7 FF - D2 46 C7 FF  E5 46 C7 FF .....F...F...F..
FFC73268 : 94 9B 68 F7  00 00 00 00 - 94 9B 68 F7  00 00 00 00 ..h.......h.....
```

```
FFC73258 : A4 9B 68 F7  42 82 04 08 - F5 0E E0 E7  14 00 00 00 ..h.B...........
[002B:0804839C]-------------------------------------------------------[ data]
0804839C : 0A 0A 09 42  41 44 20 50 - 41 53 53 57  4F 52 44 20 ...BAD PASSWORD
080483AC : 3A 50 0A 0A  0A 0A 09 4F - 4B 20 43 72  61 63 6B 65 :P.....OK Cracke
[0023:F7754420]-------------------------------------------------------[ code]
0xf7754420: <__kernel_vsyscall>:    push   ecx
0xf7754421: <__kernel_vsyscall+1>:  push   edx
0xf7754422: <__kernel_vsyscall+2>:  push   ebp
0xf7754423: <__kernel_vsyscall+3>:  mov    ebp,esp
0xf7754425: <__kernel_vsyscall+5>:  sysenter
0xf7754427: <__kernel_vsyscall+7>:  nop
--------------------------------------------------------------------------
0xf7754420 in ?? ()
```

También en este caso la syscall es **_exit_group** pues el valor en EAX es 0xFC (**252**) que provocara el cierre del programa.

```
#define __NR_exit_group            252
```

De todos modos,si os fijáis en la función **0xf7689b94**, si quitamos la llamada a sysenter también en este caso llegaríamos a la llamada a **_exit,** con el método tradicional**:**

```
0xf7689ba4:mov    eax,0x1
0xf7689ba9:int    0x80
```

Con esto, acabamos de ver la última entrada de reubicación y como se realiza todo el proceso de enlazado dinámico en este programa.

Solución del crackme.

Realmente el crackme no lo he terminado pues creo que su solución pasa por utilizar la fuerza bruta y en este punto, no tengo muchas ganas de enfrentarme a todo ese proceso (ya sabéis la famosa F1ACA).

De todos modos su desarrollo si que lo vamos a ver y os dejo mis notas en el desensamblado que he ido tomando.

Para empezar, estamos en la dirección de retorno de la llamada a la función **getcha**r, **0x80481dc**,que es la que toma los valores del password que hemos puesto, eso si de uno en uno, generándose un bucle hasta obtener todos los valores. Como referencia tened en cuenta que el valor que puse fue **"123456789"**:

```
0x80481dc: or     eax,eax  ; eax=31
0x80481de: js     0x804823e ; salta si el flag de signo este a 1
0x80481e0: cmp    al,0x20
0x80481e2: je     0x80481d9 ; salta si es igual a 0x20
0x80481e4: cmp    al,0x9
0x80481e6: jb     0x80481ec; salta si es menor
0x80481e8: cmp    al,0xe
0x80481ea: jb     0x80481d9  ; salta si es menor
0x80481ec: stos   BYTE PTR es:[edi],al ; carga el valor 0x31 a edi:080483F0
0x80481ed: dec    esi ; decrementa una unidad a esi:00000012
0x80481ee: je     0x8048203
0x80481f0: call   DWORD PTR [ebx+0x8]
```

Este bucle se repite hasta que toma los 10 valores, siendo el último el salto de linea **\n** que sin darnos cuenta se incluye al presionar **<Intro>.** Después continua repitiendo un bucle parecido al anterior con la longitud del serial que está en **EAX**:

```
0x80481f3: or     eax,eax  ; EAX vale 0xA
0x80481f5: js     0x8048203
0x80481f7: cmp    al,0x20  ; compara si vale 0x20
0x80481f9: je     0x8048203 ; no salta
0x80481fb: cmp    al,0x9
0x80481fd: jb     0x80481ec  ; no es menor que 0x9, no salta
0x80481ff: cmp    al,0xe
0x8048201: jae    0x80481ec  ;no es mayor o igual a 0xe
0x8048203: mov    BYTE PTR [edi],0x0
0x8048206: lea    esi,[ebx+0x20]
0x8048209: call   0x8048242 ; entramos con "si"-->
```

No he encontrado un numero de caracteres adecuado;pero puede que aquí, la solución correcta tome alguno de estos salto. En mi caso llegamos al final y entramos en la función en **0x8048242**, que es una autentica locura:

```
0x8048242: push   ebx
0x8048243: mov    edi,esi  ;el inicio de la cadena en esi:080483F0 pasa a EDI
0x8048245: mov    ecx,DWORD PTR [edi] ; mueve 34333231 a ECX
0x8048247: mov    ebx,DWORD PTR [edi+0x4] ; mueve 38373635 a EBX
0x804824a: xor    edi,edi  ; borra EDI
0x804824c: mov    al,bl  ; mueve 0x35 a al
0x804824e: mov    ah,cl; mueve 0x31 a ah quedando eax:00003135
0x8048250: rol    eax,0x10 ; rota hacia la izquierda 0x10 (16 bits) quedando como
31350000
0x8048253: mov    al,bh  ; mueve 0x36 a AL
0x8048255: mov    ah,ch; mueve 32 a AH quedando eax:31353236
0x8048257: xor    eax,0xa589ca11 ; quedando como eax:94BCF827
0x804825c: mov    edi,0x80482fe ;
0x8048261: shl    eax,0x5 ; desplazamiento lógico a la izquierda 5 veces queda
eax=979F04E0
0x8048264: xor    DWORD PTR [edi],eax  ; xor del valor que hay edi con eax; (EDI)
termina valiendo:0xf2f16196
```

Hasta aquí termina una fase, pues como vemos al final los valores señalados por EDI **0x80482FE**, que se han obtenido de hacer el XOR con EAX, serán los que compare con los valores correctos.
.

```
0x8048266: add    edi,0x4 ; seguimos EDI apuntando a edi:08048302
0x804826c: rcr    ebx,0x13  ; rotar  hacia la derecha con acarreo 13 veces, de modo
que ebx:38373635 ahora vale ebx:CD8D4706
0x804826f: and    ebx,0x211333  ; ebx termina valiendo ebx:00010302
0x8048275: sub    edx,edx ; edx=0
0x8048277: xor    edx,ebx  ; xor 0 con 10302 por tanto edx vale 0x10302 tambien.
0x8048279: xor    edx,0x87654321  ; 0x10302 xor 0x87654321; edx termina
valiendo edx:87644023
0x804827f: xor    DWORD PTR [edi],edx  ; xor del valor que hay edi con eax; (EDI)
termina valiendo:0xa84b2157
```

Otra fase, continua colocando valores contiguos a los anteriores y señalados por EDI.

```
0x8048281: and    ebx,0x2f13af92 ; ebx termina valiendo ebx:00010302
0x8048281: and    ebx,0x2f13af92 ; ebx termina valiendo ebx:00010302
0x8048287: add    ebx,0x12623345 ; ebx termina valiendo ebx:12633647
0x804828d: sub    esi,esi  ; borra esi
0x804828f: xor    esi,ebx ; xor 0 con 12633647, queda esi como esi:12633647
0x8048291: ror    esi,0x39 ; esi:319B2389
0x8048294: add    edi,0x4  ; edi señala ahora a edi:08048306
0x804829a: xor    DWORD PTR [edi],esi
```

Otra tanda de valores que coloca en la dirección indicada por EDI después de hacer un XOR. EDI va por **0x8048306**

```
0x804829c: and    ebx,0x2113ff98  ; queda ebx:00033600
0x80482a2: add    ebx,0x5623a5ff  ; queda ebx:5626DBFF
0x80482a8: xor    esi,ebx  ; xor esi:319B2389 con ebx:5626DBFF quedando
esi:67BDF876
0x80482aa: ror    esi,0x39  ; queda esi:DEFC3B33
0x80482ad: rol    ecx,0x56  ; ecx:8C4D0CCC
0x80482b0: ror    ebx,0x7 ; ebx:FEAC4DB7
0x80482b3: xor    ecx,ebx ; ecx:72E1417B
0x80482b5: rcl    ecx,0x32 ;ecx:05ECE5C2
0x80482b8: add    edi,0x4 ; edi:0804830A
0x80482be: mov    DWORD PTR [edi],ecx ;
0x80482c0: sub    edx,edx  ; borra edx
0x80482c2: mov    dx,ax  ; eax:979F04E0 queda edx como edx:000004E0
0x80482c5: neg    edx  ; edx:FFFFFB20
0x80482c7: shl    edx,0x8 ; edx:FFFB2000
0x80482ca: and    edx,0xdeadc8b3  ; edx:DEA90000
0x80482d0: mov    esi,edx  ; esi:DEA90000
0x80482d2: imul   esi  ; eax:979F04E0 * esi:DEA90000
0x80482d4: rcr    esi,0x15 ;  esi:90000EF5
0x80482d7: xor    ecx,esi; ecx:05ECE5C2 XOR  esi:90000EF5  queda
ecx:95ECEB37
0x80482d9: xor    ecx,eax ; ecx:95ECEB37 XOR  eax:77E00000 queda
ecx:E20CEB37
0x80482db: xor    eax,esi  ;eax:E7E00EF5
0x80482dd: xor    edx,ebx ;edx:F33BB4E3
0x80482df: xor    edx,0x12345678  ; edx:E10FE29B
0x80482e5: rol    edx,0x19 ; edx:37C21FC5
0x80482e8: add    edx,eax ; edx:1FA22EBA
0x80482ea: add    edx,esi  ; edx:AFA23DAF
0x80482ec: add    edi,0x4
0x80482f2: mov    DWORD PTR [edi],edx; mueve edx:AFA23DAF a [edi:0804830E]
0x80482f4: xor    DWORD PTR [edi],eax  ; eax:E7E00EF5
```

Aquí acaba otra tanda y yo ya estaba aburrido; solo he puesto los valores mas relevantes.

```
0x80482f6: pop    ebx  ; saca de la pila ebx:080483D0
0x80482f7: mov    edx,0x1d ; edx:0000001D
```

```
0x80482fc: ret
```

Acaba esta función y ya llegamos al final, este **ret** nos lleva a **0x804820e**:

```
0x804820e: mov    esi,0x80482fe ; dirección donde esta el resultado del serial trucho
0x8048213: mov    edi,0x8048312 ; dirección donde están los valores correctos.
0x8048218: mov    ecx,0x10 ; el contador son 16
0x804821d: cld    ; pone a 0 el flag de dirección
0x804821e: repz cmps BYTE PTR ds:[esi],BYTE PTR es:[edi] ; repite la
```

comparación siempre que sean iguales y este activado el flag cero Z

```
0x8048220: mov    esi,0x80483b0
0x8048225: je     0x8048231 ; si la comparación anterior es correcta nos lleva al
```

mensaje **"OK Cracked Now mail me :)"** Por tanto si cambiamos este salto por uno incondicional admitirá cualquier serial.

En el momento de la comparación los valores en **EDI** son los importantes, serían ellos con los que habría que empezar a intentar revertir o ser el objetivo de un a taque bruteforce:

```
gdb> x/16b $edi
0x8048312:  0x76  0x88  0x79  0x78  0x74  0x22  0x4a  0xa8
0x804831a:  0xe4  0xc0  0x7c  0x00  0xe3  0x5b  0xb1  0x4a
```

Mientras que los valores obtenidos con el serial 123456789, están en **ESI**:

```
gdb> x/16b $esi
0x80482fe:  0x96  0x61  0xf1  0xf2  0x57  0x21  0x4b  0xa8
0x8048306:  0xe4  0x41  0xfe  0x00  0xc2  0xe5  0xec  0x05
```

También habría que ver si este camino es el indicado, a lo mejor es mas fácil de lo que parece; pero de momento lo dejaremos aquí, y si alguien se anima pues espero que este desensamblado le pueda ayudar.

Conclusión:

Con este crackme le hemos dado un repaso a la vinculación dinámica del formato **ELF**, hemos continuado profundizando en **GDB** y he intentado explicar como funciona internamente el sistema GNU/Linux, dentro del ámbito del estudio de binarios. Espero se haya entendido todo :-)

Agradecimiento:

Sin duda a los grandes Maestros, Ricardo Narvaja, Gerardo Richarte**,** +NCR/CRC! [ReVeRsEr], a todos los CracksLatinoS y claro a ti, que has llegado al final de este extenso tutorial (uffff).

Cualquier comentario a cvtukan(arroba)gmail.com o en la lista de Crackslatinos.

http://groups.google.com/group/CrackSLatinoS