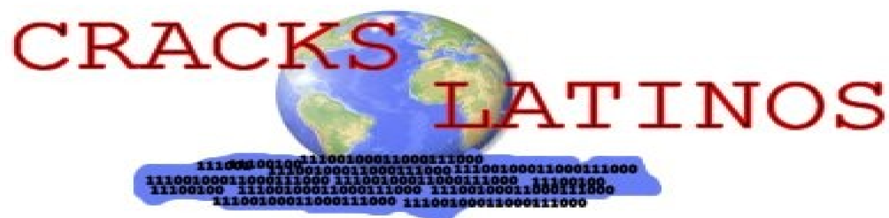


Introducción al Cracking en Linux 07 – ELF.

Programa:	**Crackme 888** Formaba parte de los crackmes de Damm Vulnerable Linux: http://www.4shared.com/file/9CxyCEoJ/Crackmes_DVLtar.html
Descripción:	Programa con protección antidebugger y formato elf corrupto.
Dificultad:	Media.
Herramientas:	Herramientas del sistema y GDB.
Objetivos:	Estudiar el formato ELF en un caso práctico e intentar solucionar el crackme.

Cracker: [Juan Jose]	Fecha: 30/11/2011

Introducción:

Seguimos con el manejo de binarios en **GNU/Linux**, en este caso vamos a profundizar en el formato de los ejecutables de Unix , **ELF** (hay otros como COOF o a.out, pero su uso es minoritario), que nos permitirá enfrentarnos a ejecutables que tiene dañada la cabecera elf, normalmente para evitar que los podamos estudiar y que en el mundo real (malware, troyanos, etc...) serán los que mas fácilmente nos vamos a encontrar.

De momento, vamos a seguir utilizando **gdb** como debugger, la verdad que ha sido una agradable sorpresa, pues hace dos años, gdb no solía admitir este tipo de programas con la cabecera elf corrupta; pero en esta ocasión ha funcionado perfectamente y será la herramienta que mas utilicemos en este tute.

Quien quiera profundizar en el tema, puede hallar todas las especificaciones técnica del formato ELF en este documento:

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

Al Atake:

Hasta ahora los crackmes que hemos utilizado en esta serie de tutoriales eran básicos, nos ayudaban a comenzar con el manejo de las herramientas que tenemos en linux. En este caso nos vamos a enfrentar con un programa muy depurado, creado para dificultar nuestro análisis, con ligera ofuscación de código y con protección antidebugger. Junto al programa viene un readme, que nos dice:

"make it print 'OK', no patching please"

Por tanto el objetivo es que nos devuelva un **OK** sin parchear. Vamos a ejecutarlo:

```
juanjo@tukan{~/Descargas/crk/crackmes_linux/888}$ ./888
NO
```

Solo nos dice un simple "**NO**", como todos los crackmes difíciles, no nos da pistas, ya sabemos que al enemigo ni agua ;-)

Por cierto, su nombre no es un misterio, si miramos su tamaño con ls -l vemos:

```
-rwxr-xr-x 1 juanjo juanjo    888 nov 15 12:30 888
```

Con la opción -l el comando ls nos dice los permisos del fichero, el número de enlaces que tiene, el nombre del propietario, el del grupo al que pertenece, el tamaño (en bytes), una marca de tiempo, y el nombre del fichero. Por tanto tenemos un programa con solo 888 bytes, que nos va a poner las cosas difíciles.

Primero, vamos a hacer una copia del ejecutable, en estos casos difíciles es interesante guardar el original:

```
$ cp 888 888.copia
```

Y para seguir con las buenas costumbres, vamos a desarrollar el estudio de este crackme en los tres pasos que ya conocemos; información, desensamblado y depuración.

1. Información.

1.a) File:

```
juanjo@tukan{~/Descargas/crk/crackmes_linux/888}$ file 888
888: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked
, corrupted section header size
```

Vemos que es un **ELF** de 32 bits, y lo mas llamativo es **"statically linked"**, que significa que ha sido enlazado estáticamente, por tanto este programa tiene todas las funciones incluidas en su propio código, es autónomo y no necesita de librerías exteriores. Esto suele ir acompañado de un aumento de tamaño, pero como veremos en este caso el programador ha hecho un gran trabajo.

Con **gcc** el enlazado por defecto es dinámico (como hemos visto en los anteriores crackmes), por lo que al compilar el programa debemos indicarlo con la opción **-static:**

```
$ gcc -static -o 888 888.c
```

File también nos comenta **"corrupted section header size"**, el tamaño de la cabecera de secciones esta corrupto. Para comprender lo que significa debemos tener mas información sobre el formato **ELF** de este programa.

1.b) Readelf; Nos da toda la información incluida en el binario sobre el formato **elf**.

Como ya hicimos, vamos a guardar toda la información en un archivo:

```
$ readelf -a 888 >>  informacion.txt
```

En este caso, al mirar el archivo vemos que hay poco datos:

```
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:               0x804823f
  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)        0x20
  Number of program headers:         1
  Size of section headers:           0 (bytes)
  Number of section headers:         0
  Section header string table index: 0

There are no sections in this file.

There are no sections in this file.

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x000000 0x08048000 0x08048000 0x00378 0x00398 RWE 0x1000

There is no dynamic section in this file.

There are no relocations in this file.

There are no unwind sections in this file.
```

Como vemos la cabecera de sección (**section headers**) no existe y por tanto este programa no tiene secciones. De esta forma vemos que el formato **ELF** es muy flexible y para ejecutarse un programa solo es

fundamental la cabecera de ELF(**ELF Header**) y como mínimo un segmento en la cabecera del programa (**Program Headers**). Lógicamente esto sólo se puede dar en un programa estático, pues como vimos en el anterior tute sobre el enlazado dinámico (las secciones **interp.hash, .dynsym, y .dynstr** son imprecindibles) un programa con enlazado dinámico no podría ejecutarse sin secciones.

Aunque **readelf** nos ha dado la información correcta, me parece muy interesante comprobarla manualmente y ver como se define en nuestro archivo **888** el formato **ELF**.
La estructura que vamos a ver la podéis encontrar en el archivo INCLUDE **elf.h**, que podemos encontrar en:

/usr/include/elf.h

Este archivo sera necesario definirlo en cualquier programa C donde queramos manejar este formato y en el esta incluido las versiones de 32 y 64 bits.

Por cierto, también me ha sido muy útil la traducción al español del ezine **Phrack nº 58 - 0x05** ***" Blindando el ELF: Encriptacion Binaria en la platforma UNIX"*** de donde he tomado la estructura ELF básica.

*******************FORMATO ELF***

Un ejecutable ELF suele tener una estructura típica,que podemos ver con esta tabla:

CABECERA ELF (ELF HEADER)
CABECERA DEL PROGRAMA (PROGRAM HEADERS) Define varios segmentos.
DATOS DEL PROGRAMA. Normalmente estructurado en secciones.
CABECERA SECCIONES. Una cabecera para cada sección

Esto es lo típico, pero no necesariamente siempre lo vamos a encontrar así, como vemos en este caso.

La cabecera del programa suele definir un segmento para cada sección y es la que define la colocación del ejecutable en memoria cuando se ejecuta, de modo que todos los bytes del ejecutable se coloquen en memoria de forma secuencial sin superponerse. Por tanto la estructura de un programa ELF cuando se convierte en un proceso se puede ver así:

CABECERA ELF (ELF HEADER)
SEGMENTO 1
SEGMENTO 2
SEGMENTO 3
....

Vamos a ver cada cabecera como se define y como se aplica en nuestro programa:

ELF Header o Cabecera ELF

```
#define EI_NIDENT (16)

typedef struct
{
  unsigned char e_ident[EI_NIDENT]; /* Numero magico y otra info */
  Elf32_Half    e_type;             /* Tipo de archivo object */
```

```
  Elf32_Half    e_machine;          /* Arquitectura */
  Elf32_Word    e_version;          /* Version de archivo objeto */
  Elf32_Addr    e_entry;            /* Direccion virtual de punto de entrada */
  Elf32_Off     e_phoff;            /* Program header table file offset */
  Elf32_Off     e_shoff;            /* Section header table file offset */
  Elf32_Word    e_flags;            /* Flags especificas del procesador */
  Elf32_Half    e_ehsize;           /* Tamaño de las cabeceras ELF en bytes */
  Elf32_Half    e_phentsize;        /* Program header table entry size */
  Elf32_Half    e_phnum;            /* Program header table entry count */
  Elf32_Half    e_shentsize;        /* Section header table entry size */
  Elf32_Half    e_shnum;            /* Section header table entry count */
  Elf32_Half    e_shstrndx;         /* Section header string table index */
} Elf32_Ehdr;
```

*** e_ident[EI_NIDENT]** Comienza en el offset 0 del archivo y son los primeros 16 bytes. Da una informacion basica sobre como tratar e interpretar el binario. En nuestro caso son:

```
            00                                              0f
            7f 45 4c 46 01 01 01 00-00 00 00 00 00 00 00 00
```

Cada byte tiene un significado:

```
#define EI_MAG0  0      /* Byte Index 0 de Identificacion de Archivo */---->7f
#define ELFMAG0 0x7f            /* Byte Numero Magico 0 */

#define EI_MAG1  1      /* Byte Index 1 de Identificacion de Archivo */---->45
#define ELFMAG1 'E'     /* Byte Numero Magico 1 */

#define EI_MAG2  2      /* Byte Index 2 de Identificacion de Archivo */---->4C
define ELFMAG2  'L'             /* Byte Numero Magico 2 */

#define EI_MAG3  3      /* Byte Index 3 de Identificacion de Archivo */---->46
#define ELFMAG3 'F'             /* Byte Numero Magico 3 */define

#define EI_CLASS 4   /* Byte Index de clase de archivo. Hay tres opciones: */
  #define ELFCLASSNONE    0               /* Clase invalida */
  #define ELFCLASS32      1               /* Objetos 32-bit */------------->01
  #define ELFCLASS64      2               /* Objetos 64-bit */

#define EI_DATA  5  /* Byte Index de codificación de datos. Tres opciones: */
  #define ELFDATANONE     0     /* Codificación de data invalida */
  #define ELFDATA2LSB     1     /* Complemento de 2, little endian */------>01
  #define ELFDATA2MSB     2     /* Complemento de 2, big endian */

#define EI_VERSION      6     /* Byte Index de version de archivo.Una opción */
  #define EV_CURRENT      1     /* Valor debe ser EV_CURRENT */------------>01
```

Los demás bytes no tienen utilidad.

Seguimos con la siguiente estructura de ELF header:

```
            10                                              1f
            02 00 03 00 01 00 00 00-3f 82 04 08 2c 00 00 00
```

*** e_type** describe como el binario debe ser utilizado. Como vemos es un valor Half por tanto son 2 bytes. Los siguientes son valores legales:

```
#define ET_NONE         0               /* No tipo archivo */
#define ET_REL          1               /* Archivo relocalizable */
#define ET_EXEC         2               /* Archivo ejecutable */----->02 00
#define ET_DYN          3               /* Archivo objeto compartido */
#define ET_CORE         4               /* Archivo Core */
```

* **e_machine** indica para que arquitectura esta propuesto el archivo objeto. Su tamaño es Half; 2 bytes.
La siguiente es una lista corta de los valores mas comunes:

```
#define EM_SPARC          2              /* SUN SPARC */
#define EM_386            3              /* Intel 80386 */----------->03 00
#define EM_SPARCV9       43              /* SPARC v9 64-bit */
#define EM_IA_64         50              /* Intel Merced */
```

* **e_version** indica que versión de ELF conforma el archivo ELF. Es un valor Word; 4 bytes. Actualmente debe ser seteado a EV_CURRENT, indicado e_ident[EI_VERSION], su valor es 1.----------- ------------------> 01 00 00 00

* **e_entry** contiene la dirección virtual relativa del punto de entrada al binario. Esta es tradicionalmente la función **_start()** que esta ubicada en el comienzo de la sección **.text**. Este campo solo tiene sentido para objetos ET_EXEX------------> 3f 82 04 08, al ser little endian-->0x0804823f

* **e_phoff** contiene el offset desde el principio del archivo a la primera Cabecera de Programa. Los offset en elf32 suelen tener 4 bytes--->2c 00 00 00
Por tanto la cabecera del programa comienza en **0x2c (44 bytes)**. Este campo es solo significante en ET_EXEC y objetos ET_DYN.

```
20                                             2f
00 00 00 00 00 00 00 00-34 00 20 00 01 00 00 00
```

* **e_shoff** contiene el offset desde el comienzo del archivo a la primera Cabecera de Sección. En este caso no hay secciones----------------> 00 00 00 00
Este campo es siempre útil para la ingeniería inversa, pero solo requerido en archivos ET_REL.

* **e_flags** contiene flags especificas del procesador. Tiene un tamaño Word, ocupa 4 bytes--> 00 00 00 00
Este campo no es usado en sistemas i386 o SPARC, por lo que puede ser ignorado tranquilamente.

* **e_ehsize** contiene el tamaño de la cabecera ELF. Esto es para chequeo de errores y debe ser seteado a sizeof(Elf32_Ehdr). Su tamaño son 2 bytes que corresponde al tamaño de elf header---------------------------------> 34 00
Como vemos su valor es de **0x34** que son **52 bytes**.

* **e_phentsize** contiene el tamaño de una Cabecera de Programa. Esto es para chequeo de errores y debe ser seteado a sizeof(Elf32_Phdr).
Su tamaño es half, 2 bytes---> 20 00
Corresponde a 0x20 que son 32 bytes, como solo hay una cabecera de programa, ya sabemos que después de ella comenzara los datos del programa.

* **e_phnum** contiene el numero de cabeceras de Programa. La tabla de cabecera de programa son un e_phnum (en este caso solo 1) de elementos formados cada uno por un array de tamaño Elf32_Phdr (en este caso 32 bytes). Su tamaño son 2 bytes---> 01 00

* **e_shentsize** contiene el tamaño de una Cabecera de Sección. Esto es para chequeo de errores y debe ser seteado a sizeof(Elf32_Shdr). Su tamaño es Half, son 2bytes--> 00 00
En este caso este valor a 0 nos da el error en file.

```
30       34                                      3f
00 00 00 00 | 00 80 04 08-00 80 04 08 78 03 00 00
            |
      Fin Cabecera ELF
```

* **e_shnum** contiene el número de cabeceras de Sección. La cabecera de sección es un array de Elf32_Shdr (0 bytes) con elementos e_shnum (0 elementos). Su tamaño es de 2 bytes--> 00 00

* **e_shstrndx** contiene el número de la sección que contiene la tabla de strings de los nombres de las secciones. Son 2 bytes----------------> 00 00

Como vemos en el byte 0x34 (52) acabamos la cabecera ELF y si nos fijamos en los datos que nos ha dado readelf son todos correctos.

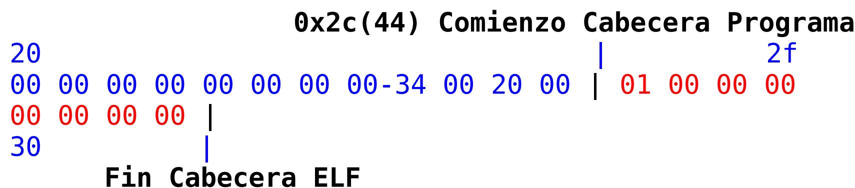
Normalmente a continuación del la cabecera ELF viene la cabecera del programa:

ELF Segmentos; cabecera del programa

Su estructura en un ELF de 32 bits es esta:

```
typedef struct
{
  Elf32_Word      p_type;                 /* Tipo de Segmento */
  Elf32_Off p_offset;          /* Offset de archivo de Segmento */
  Elf32_Addr      p_vaddr;          /* Dirección virtual de Segmento */
  Elf32_Addr      p_paddr;          /* Dirección fisica de Segmento */
  Elf32_Word      p_filesz;         /* Tamaño de segmento en archivo */
  Elf32_Word      p_memsz;          /* Tamaño de segmento en memoria */
  Elf32_Word      p_flags;          /* Flags de Segmento */
  Elf32_Word      p_align;          /* Alineación de Segmento */
} Elf32_Phdr;
```

Para saber donde comienza debemos buscar en el la cabecera ELF el valor **e_phoff**, que indica donde comienza la primera cabecera del programa, en este caso es **0x2C** que son 44 bytes, lo cual es raro pues nos lleva dentro de la cabecera ELF (llega hasta **0x32** -52 bytes-). Normalmente que se solapen estas dos cabeceras debería inutilizar al programa, pero en este caso los valores de cada campo se complementan y no dan problemas.



* **p_type** describe como tratar los contenidos de un segmento. Su tamaño es Word, por tanto son 4 bytes. Los siguientes son valores legales:

```
#define PT_NULL      0   /* Program header table entry sin uso */
#define PT_LOAD      1   /* Segmento de programa cargable */------> 01 00 00 00
#define PT_DYNAMIC   2   /* Informacion de linkeado dinamico */
#define PT_INTERP    3   /* Interpretador del programa */
#define PT_NOTE      4   /* Informacion auxiliar */
#define PT_SHLIB     5   /* Reservado */
#define PT_PHDR      6   /* Entrada para la tabla de cabeceras
                                en si misma. */
```

Como veis coincide con e_phnum y e_shentsize de la cabecera ELF.

* **p_offset** contiene el offset dentro del archivo del primer byte del segmento. Su tamaño es de 4 bytes---------------------------------> 00 00 00 00
Este segmento en realidad es todo el programa por tanto comienza en el byte 0 y coincide con los campos de la cabecera ELF e_shnum e_shstrndx.

*** p_vaddr** contiene la direccion virtual relativa que el segmento espera para ser cargado dentro de memoria--------------------------------> 00 80 04 08
Por tanto corresponde con 0x08048000, que sera la imagen base para este programa.

*** p_paddr** contiene la dirección física que el segmento espera para ser cargado en memoria. Este campo no tiene significado salvo el soporte de hardware y requiere esta información. Típicamente este campo es seteado a un valor 0 o es el mismo valor que p_vaddr:0x08048000---> 00 80 04 08

*** p_filesz** contiene el tamaño en bytes del segmento dentro del archivo. Su tamaño es Word, son 4 bytes---> 78 03 00 00
Vemos que su valor es 0x378, que son los 888 bytes del programa.

Fin Segmento 0x4C

```
40                                  |         4f
98 03 00 00 07 00 00 00-00 10 00 00 |b0 01 58 33
```

*** p_memsz** contiene el tamaño en bytes del segmento una vez cargado en memoria. Si el segmento tiene un p_memsz mas largo que p_filesz, el espacio que falta es inicializado a 0.Su tamaño es de 4 bytes-----> 98 03 00 00
Corresponde a 0x398 (920 bytes) por tanto todos los bytes después de los 888 son inicializados a 0.

*** p_flags** contiene los flags de protección de memoria para el segmento una vez cargado. Cualquier combinación bit wise OR'd de las siguientes , son valores legales:

```
#define PF_X            (1 << 0) = 1    /* Segmento es ejecutable */
#define PF_W            (1 << 1) = 2    /* Segmento es escribible */
#define PF_R            (1 << 2) = 4    /* Segmento es legible */
```

(Explicación mas adelante). Son cuatro bytes ---------------------> 07 00 00 00
En este caso el archivo se puede leer, escribir y ejecutar; 1+2+4 = 7(RWE).

*** p_align** contiene la alineación para el segmento en memoria. Si el segmento es de tipo PT_LOAD, entonces la alineación sera el tamaño de pagina esperado. Es un valor de 4 bytes-----------------------------------> 00 10 00 00
Por tanto su valor es 0x1000.

Este último campo llega hasta **0x4C (76)** por lo que de esta forma sabemos que los datos del programa empiezan en el byte 76. Esto también lo podríamos calcular sabiendo que la cabecera del programa comenzaba en **0x2C** y su tamaño era de **0x20** bytes.

Podemos comprobar que los valores son correctos, mirando los que nos da readelf:

Program Headers:

```
Type     Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
LOAD     0x000000 0x08048000 0x08048000 0x00378 0x00398 RWE 0x1000
```

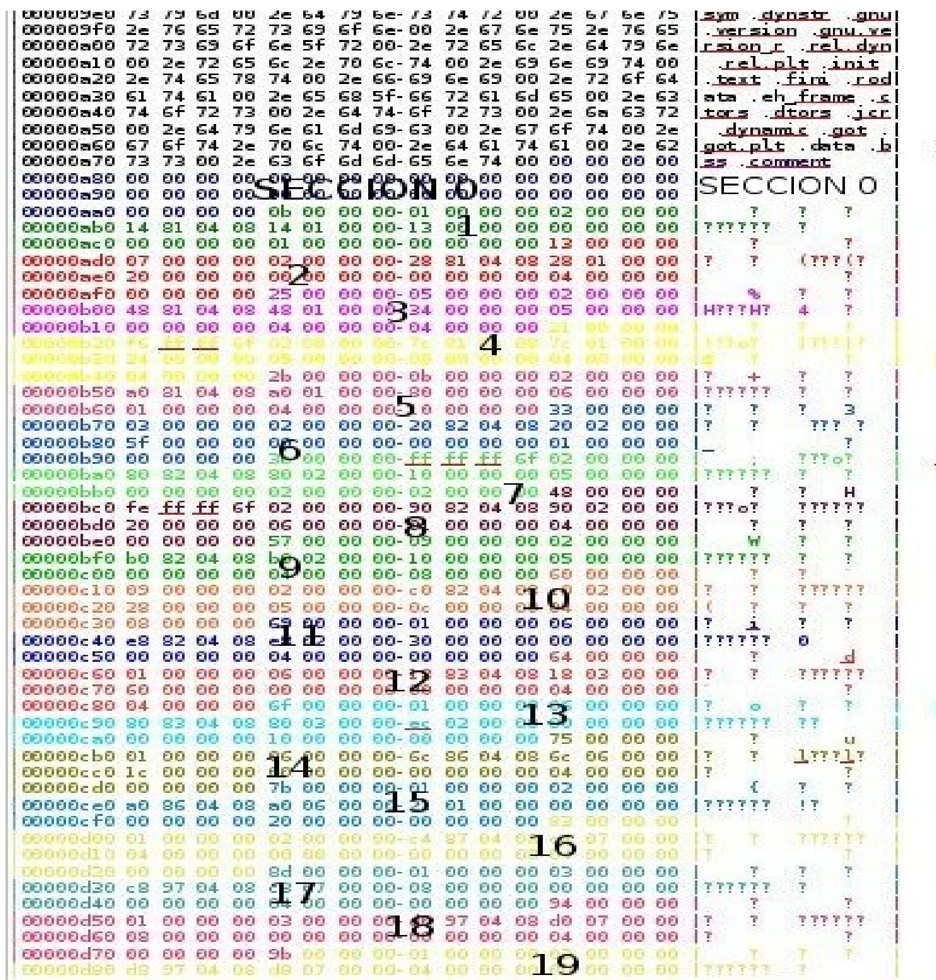
ELF Secciones

Para ver la zona de la cabecera de secciones, como es lógico no no sirve este programa. En este caso vamos a utilizar el programa **linux_crackme3** que estudiamos en la parte anterior (creíais que os ibais a librar,eeeh).

Primero debemos de mirar en la cabecera ELF para saber por donde empezar, por tanto con "**readelf -h linux_crackme3**" vemos:

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: 0x8048380
Start of program headers: 52 (bytes into file)
Start of section headers: 2684 (bytes into file)----Comienza la cabecera de seccion en el byte **2684** o **0xa7c**
Flags: 0x0
Size of this header: 52 (bytes)
Size of program headers: 32 (bytes)
Number of program headers: 7
Size of section headers: 40 (bytes)------------Cada sección tiene un tamaño de **40 bytes**
Number of section headers: 27----------En la cabecera hay **27** secciones
Section header string table index: 26----------En la sección **26** es donde se encuentra el índice de la tabla de nombres de cabecera de secciones.

Como vemos las cabeceras de secciones comienzan en **0xA7C (2684)** (suele ser al final del archivo), y cada cabecera de sección tiene **40 bytes**, vamos a comprobarlo con un editor hexadecimal:



```
00000d90 00 00 00 00 04 00 00 00-00 00 00 00 a0 00 00 00
00000da0 06 00 00 00 [illegible] 00 00 00-dc 97 04 08 dc 07 00 00      20
00000db0 d0 00 00 00 [illegible] [illegible] 00 00-00 00 00 00 04 00 00 00
00000dc0 08 00 00 00 a9 00 00 00-01 00 00 00 03 00 00 00
00000dd0 ac 98 04 08 ac 08 00 [illegible]-[illegible] 00 00 00 00 00 00 00      21
00000de0 00 00 00 00 04 00 00 00-04 00 00 00 ac 00 00 00
00000df0 01 00 00 00 03 00 00 00-b0 98 04 08 [illegible] 08 00 00      22
00000e00 20 00 00 00 00 00 00 00-00 00 00 00 [illegible] 00 00 00
00000e10 04 00 00 00 b7 00 00 00-01 00 00 00 03 00 00 00
00000e20 d0 98 04 08 [illegible] [illegible] 00 00-08 00 00 00 00 00 00 00      23
00000e30 00 00 00 00 04 00 00 00-00 00 00 00 bd 00 00 00
00000e40 08 00 00 00 03 00 00 00-d8 98 04 08 d8 08 00 00
00000e50 0c 00 00 00 00 00 00 00-[illegible] 00 00 00 04 00 00 00      24
00000e60 00 00 00 00 c2 00 00 00-01 00 00 00 [illegible] 00 00 00
00000e70 00 00 00 00 d8 08 00 00-d9 00 00 00 [illegible] 00 00 00      25
00000e80 00 00 00 00 01 00 00 00-00 00 00 00 01 00 00 00
00000e90 03 00 00 00 00 00 00 00-00 00 00 00 b1 09 00 00
00000ea0 cb 00 00 00 00 00 00 00-[illegible] 00 00 00 01 00 00 00      26
00000eb0 00 00 00 00
```

Hay **27** secciones pues comenzamos con la cabecera de **sección 0** que no tiene ningún valor:

La entrada 0 es la sección NULL: una sección que es seteada a 0 y no describe ninguna parte del binario.

```
                                                  0xa7c
                                                    |
                                                  | 00 00 00 00 |

|00000a80 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |

|00000a90 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |

|00000aa0 00 00 00 00
```

La primera cabecera de sección que ve el programa y la que primera que veremos nosotros, será la tabla de nombres de las secciones; esto es necesario para darle nombre a cada sección y como ya sabemos por la **cabecera ELF** es la **26**, que es la última.

La estructura de una cabecera de sección, es asi:

```
typedef struct
{
  Elf32_Word    sh_name;          /* Nombre de Sección (string tbl index) */
  Elf32_Word    sh_type;          /* Tipo de Sección */
  Elf32_Word    sh_flags;         /* Flags de Sección */
  Elf32_Addr    sh_addr;          /* Sección virtual addr en ejecución */
  Elf32_Off     sh_offset;        /* Offset de Sección de Archivo */
  Elf32_Word    sh_size;          /* Tamaño de Sección en bytes */
  Elf32_Word    sh_link;          /* Link a otra sección */
  Elf32_Word    sh_info;          /* Información adicional de sección */
  Elf32_Word    sh_addralign;     /* Alineamiento de Sección */
  Elf32_Word    sh_entsize;       /* Tamaño de Entrada si la sección */
} Elf32_Shdr;                     /* mantiene tabla. */
```

Habrá una por cada sección como hemos comprobado. Vamos a ver la cabecera de la tabla de nombres:

```
                                                  | 01 00 00 00 |

|00000e90 03 00 00 00 00 00 00 00-00 00 00 00 b1 09 00 00 |

|00000ea0 cb 00 00 00 00 00 00 00-00 00 00 00 01 00 00 00 |

|00000eb0 00 00 00 00
```

*** sh_name** contiene un índice dentro de los contenidos de la sección de la tabla de string **e_shstrndx**. Este indice es el principio de un string terminado en null para ser usado como el nombre de la sección.--> 01 00 00 00

En este caso es el nombre que esta colocado en el byte 1 de la tabla, para saber el nombre debemos saber donde comienza esta sección, eso lo veremos con **sh_offset.**

Hay nombres reservados, los mas importantes serían:

```
.text       Codigo objeto Ejecutable
.rodata     Strings solo lectura
.data       Datos "static" inicializados
.bss        Datos "static" inicializados a 0.
```

*** sh_type** contiene el tipo de sección, ayudando a la aplicación inspectora a determinar como interpretar los contenidos de las secciones---> 03 00 00 00
Los siguientes son valores legales:

```
#define SHT_NULL         0         /* Entrada de tabla de cabeceras de
                                      seccion sin uso */
#define SHT_PROGBITS     1         /* Datos del Programa */
#define SHT_SYMTAB       2         /* Tabla de Simbolos */
#define SHT_STRTAB       3         /* Tabla String */
#define SHT_RELA         4         /* Entrada reubicada con addends */
#define SHT_HASH         5         /* Tabla de Simbolos Hash */
#define SHT_DYNAMIC      6         /* Informacion de linkeado dinamico */
#define SHT_NOTE         7         /* Notas */
#define SHT_NOBITS       8         /* Espacio de programa sin data (bss) */
#define SHT_REL          9         /* Entradas de reubicacion, no addends */
#define SHT_SHLIB        10        /* Reservado */
#define SHT_DYNSYM       11        /* Tabla de simbolos de linker dinamico */
```

*** sh_flags** contienen un bitmap definiendo como los contenidos de la sección deben ser tratados en tiempo de ejecución).------> 00 00 00 00
Cualquier valor bitwise OR'd de los siguientes es legal:

```
#define SHF_WRITE       (1 << 0)        /* Escribible */
#define SHF_ALLOC       (1 << 1)        /* Ocupa memoria durante ejecucion */
#define SHF_EXECINSTR   (1 << 2)        /* Ejecutable */
```

El signo **<<** es desplazamiento de bits a la izquierda, a la izquierda del signo << vemos sobre que valor se va a trabajar (**01** en todos los casos) y a la derecha vemos cuantos desplazamientos se harán en cada caso. Por tanto el valor final pueden ser una suma de cada uno de estos valores:

(1 << 0), si hacemos 0 desplazamientos a la izquierda de 1 es **1** binario
(1 << 1), hacemos 1 desplazamiento a la izquierda de 1 es **10** binario, que es **2** en decimal.
(1 << 2), hacemos 2 desplazamientos a la izquierda de 1 es **100** binario, que es **4** en decimal.
En este caso, esta sección no se utiliza en tiempo de ejecución, su valor es 0.

*** sh_addr** contiene la dirección virtual de la sección en ejecución--> 00 00 00 00
En este caso no tiene.

*** sh_offset** contiene el offset desde el principio del archivo al primer byte de la seccion--> b1 09 00 00

Como vemos esta tabla esta a partir del byte 0x9b1 (2481):

```
|000009b0 00 00 2e 73 68 73 74 72-74 61 62 00 2e 69 6e 74 |  .shstrtab .int|
|000009c0 65 72 70 00 2e 6e 6f 74-65 2e 41 42 49 2d 74 61 |erp .note.ABI-ta|
|000009d0 67 00 2e 67 6e 75 2e 68-61 73 68 00 2e 64 79 6e |g .gnu.hash .dyn|
|000009e0 73 79 6d 00 2e 64 79 6e-73 74 72 00 2e 67 6e 75 |sym .dynstr .gnu|
|000009f0 2e 76 65 72 73 69 6f 6e-00 2e 67 6e 75 2e 76 65 |.version .gnu.ve|
|00000a00 72 73 69 6f 6e 5f 72 00-2e 72 65 6c 2e 64 79 6e |rsion_r .rel.dyn|
|00000a10 00 2e 72 65 6c 2e 70 6c-74 00 2e 69 6e 69 74 00 | .rel.plt .init |
|00000a20 2e 74 65 78 74 00 2e 66-69 6e 69 00 2e 72 6f 64 |.text .fini .rod|
|00000a30 61 74 61 00 2e 65 68 5f-66 72 61 6d 65 00 2e 63 |ata .eh_frame .c|
|00000a40 74 6f 72 73 00 2e 64 74-6f 72 73 00 2e 6a 63 72 |tors .dtors .jcr|
|00000a50 00 2e 64 79 6e 61 6d 69-63 00 2e 67 6f 74 00 2e | .dynamic .got .|
|00000a60 67 6f 74 2e 70 6c 74 00-2e 64 61 74 61 00 2e 62 |got.plt .data .b|
|00000a70 73 73 00 2e 63 6f 6d 6d-65 6e 74 00 00 00 00 00 |ss .comment     |
```

Si volvemos a mirar el primer campo de esta sección **sh_name** comprobamos que su valor era 1. Comprobamos que el siguiente byte (01) después de 0x9b1 (que es el 00) comienza el nombre de esta sección-----------------**.shstrtab**

* **sh_size** contiene el tamaño en bytes de la sección--------> cb 00 00 00
Por tanto su tamaño es 0xcb (203 bytes).

* **sh_link** es usado para linkear secciones asociadas juntas-> 00 00 00 00
Es usado para linkear una tabla de string a una sección cuyos contenidos requieren una tabla de string para correcta interpretación, p.e. tablas de símbolos.

* **sh_info** es usado para contener información extra para asistir en la edición de link. Este campo tiene exactamente dos usos, indicando que sección aplica una reubicación para secciones SHT_REL[A], y mantener el máximo número de elementos mas uno dentro de la tabla de símbolos-------> 00 00 00 00

* **sh_addralign** contiene el requerimiento de alineación de los contenidos de sección típicamente **0/1** (ambos significan no alineación) o **4** que indica que si necesita alineación---> 01 00 00 00
Como vemos esta sección no necesita alineación.

* **sh_entsize**, si la sección mantiene una tabla, contiene el tamaño de cada elemento. Utilizado para chequeo de errores.---------------> 00 00 00 00

Como vemos todo coincide con lo que nos dice readelf de esta sección:

[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	Al
[26]	.shstrtab	STRTAB	00000000	0009b1	0000cb	0	0	0	0	1

De esta manera podemos ver otras secciones, por ejemplo la 1:

```
                    0b 00 00 00-01 00 00 00 02 00 00 00 |
|00000ab0 14 81 04 08 14 01 00 00-13 00 00 00 00 00 00 00 |
|00000ac0 00 00 00 00 01 00 00 00-00 00 00 00
```

* **sh_name**--> 0b 00 00 00
Como sabemos que la tabla de nombres de secciones comienza en 0x9b1, el nombre correspone a 0x9b1+0xb= 0x9bc, que corresponde a .interp

* **sh_type**---> 01 00 00 00
Corresponde a **#define SHT_PROGBITS** 1 /* Datos del Programa */

* **sh_flags**--> 02 00 00 00
#define SHF_ALLOC **(1 << 1)** el signo << es desplazamiento de bits a la

```
izquierda, por tanto si desplaza 1 byte a la izquierda de 1, seria 10 que es 2
en binario.
Si miramos una que tenga tanto la opción alloc como ejecutable, como .init
seria, 2 por el alloc y 4 por el ejecutable. Por tanto el valor final es 6 (AX).
Lo he comprobado y coincide (06 00 00 00).

* sh_addr -----------------------------------------------------> 14 81 04 08
La direccion virtual de la sección sera 08048114.

* sh_offset----------------------------------------------------> 14 01 00 00
Esta en el offset 0x114 (276) .

* sh_size------------------------------------------------------> 13 00 00 00
Es muy pequeña, solo tiene 0x13 bytes, 19 en decimal:

      0x08048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
      0x08048124 2e3200                              .2.

* sh_link------------------------------------------------------> 00 00 00 00
No enlaza a ninguna sección.

* sh_info------------------------------------------------------> 00 00 00 00
Si la otra es 0, esta también.

* sh_addralign-------------------------------------------------> 01 00 00 00
No necesita alineacioón.

* sh_entsize---------------------------------------------------> 00 00 00 00

También coincide con lo que nos dice readelf de esta sección:

    [Nr] Name          Type            Addr     Off    Size   ES Flg Lk Inf Al
    [ 1] .interp       PROGBITS        08048114 000114 000013 00   A  0   0  1

**********************************************************************
```

2. Desensamblado.

Con este crackme objdump no funciona, seguramente por la cabecera corrupta, así que tuve que buscar alternativas:

2.a)Ndisasm. Es el desensamblador que viene con **nasm**, analiza binarios 80x86 y funciona desde consola.

Su funcionamiento es muy sencillo, como siempre vamos a guardar el resultado en un archivo:

```
$ ndisasm 888 > ndisasm.txt
```

Lo mejor es que nos da un código assambler tipo Intel y no tiene problemas con las cabeceras corruptas. Si miramos el código con gedit vemos que las direcciones son la hexadecimal del archivo:

```
000001F1  0000            add [bx+si],al
000001F3  90              nop                 ; Entrypoint 0x804823f
000001F4  08C0            or al,al
```

En este caso como ya sabemos la image base por **p_vaddr**, es **0x08048000** y que ese segmento (que es todo el archivo) se mapea en memoria desde el byte **0**, podemos colocar la direcciones virtuales con la opción **"Buscar y reemplazar texto"** sustituyendo **00000** por **08048**, quedando de

esta manera:

```
0804823D  0000          add [bx+si],al
0804823F  90            nop                    ; Entrypoint
08048240  08C0          or al,al
```

Por cierto, este entrypoint creo que **nop** va a ser muy útil :-)

También se puede utilizar **ndisasm** desde un punto del programa, para evitar zonas que no nos interese, en este caso como sabemos que el todo el header llega hasta el byte **76 (0x46)** podemos utilizar la orden:

```
$ ndisasm -e 76 888 > ndisasm1.txt
```

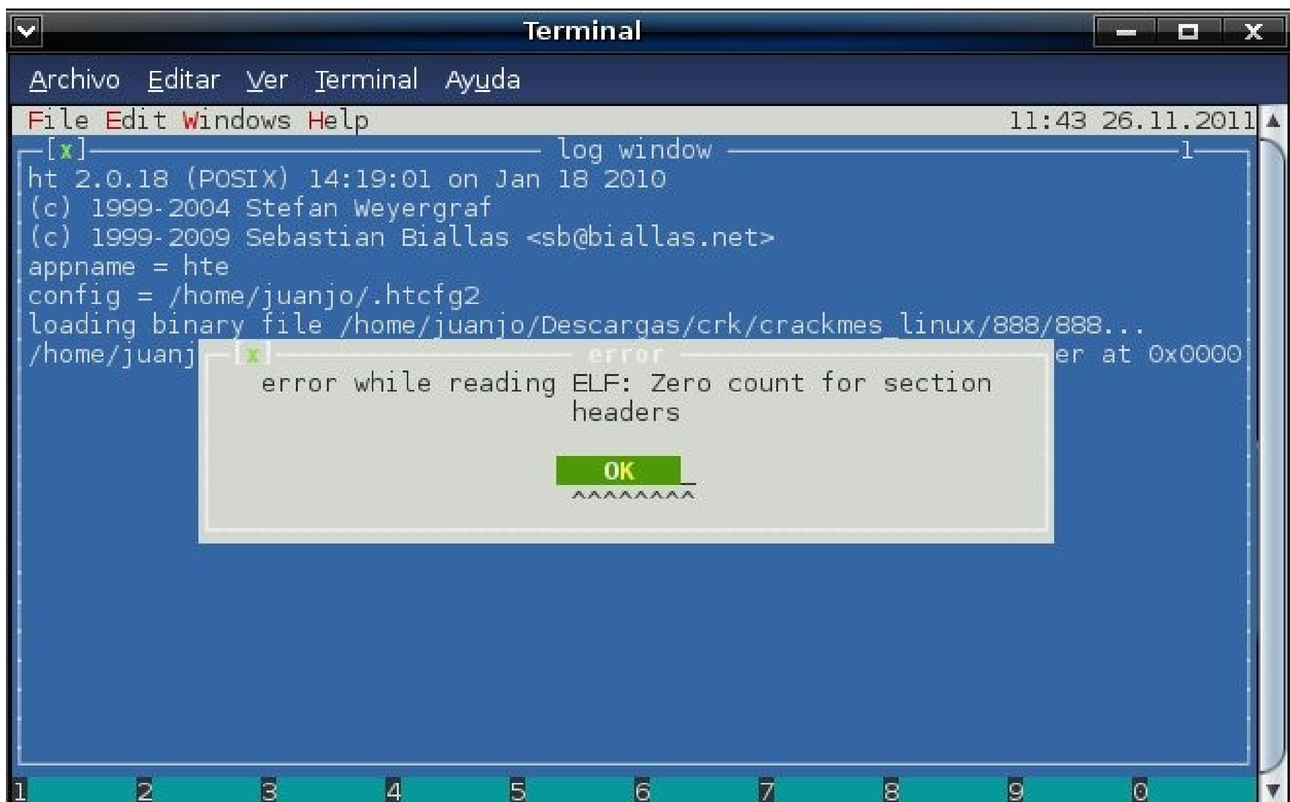
2.b) Hte. Es un editor de ejecutables que me ha sorprendido, funciona muy bien y para nuestros objetivos es genial. Es muy parecido a **Hiew,** funciona desde la consola y tiene varias opciones que le han hecho ir directamente a mis herramientas favoritas.

Como esta en los repositorios, para instalarlo solo tenemos que usar **apt-get** o **aptitude** como **root**:

```
# aptitude install ht
```

Como veis en los repositorios esta como **ht,** pero para ejecutarlo desde la consola debemos poner "**hte**", si le añadimos como opción el nombre del programa aparecerá en su modo hexadecimal:

```
$ hte 888
```



Como veis el también detecta que la cabecera de sección no existe; le damos a **<Enter>** y continua:

Su manejo es muy fácil, solo voy a comentar tres opciones que me han sido muy útiles. Con **F6 (6mode)** nos aparece un menú donde tenemos cuatro opciones:

Si le damos a **"text"** nos servirá para descubrir las cadenas de texto (strings), en **"some statictext"** nos da información sobre el ejecutable y para el final dejo **"disas/x86"** que es la que nos interesa, pues nos da el desensamblado que buscábamos:

```
Archivo  Editar  Ver  Terminal  Ayuda
File Edit Windows Help Local-Disasm                                   11:53 26.11.2011
-[x]----------- /home/juanjo/Descargas/crk/crackmes_linux/888/888 -----------2--
 0000004b 00b001583342          add         [eax+42335801], dh
 00000051 66877c2404            xchg        [esp+0x4], di
 00000056 30c0                  xor         al, al
 00000058 31c9                  xor         ecx, ecx
 0000005a f7d1                  not         ecx
 0000005c fc                    cld
 0000005d f2ae                  repnz scasb
 0000005f f7d1                  not         ecx
 00000061 49                    dec         ecx
 00000062 89c8                  mov         eax, ecx
 00000064 877c2404              xchg        [esp+0x4], edi
 00000068 c20400                ret         0x4
 0000006b e988010000            jmp         0x1f8
 00000070 57                    push        edi
 00000071 bfb1810408            mov         edi, 080481b1
 00000076 b902000000            mov         ecx, 0x2
 0000007b b090                  mov         al, 0x90
 0000007d f3aa                  repz stosb
 0000007f 5f                    pop         edi
 00000080 a18c830408            mov         eax, [0804838c]
-- view 0x00000080/128 --------------------------------------------------------
1help  2save  3open  4edit  5goto  6mode  7search 8use16  9viewin.0quit
```

Con **F5 (5goto)** podemos ir a la dirección que nos interese, teniendo en cuenta que si colocamos el numero normal, por ej. 255 lo considera decimal y para el formato hexadecimal debemos colocarlo precedido de 0x, por ej.0xff.

Por último, con **F4 (4edit)** podemos cambiar los bytes fácilmente y con **F2** salvamos los cambios. Me ha gustado mucho que si editamos el archivo en modo asm al cambiar el código, el desensamblado se transforma según los opcodes que pongamos.

3. Depuración.

Aquí comenzamos lo divertido, eso si en este punto y con este tipo de programas es fundamental tener a mano todas las teorías de assambler que podamos, pues es en el código donde vamos a poder descubrir todos los secretos de este crackme y como resolverlo. Ya sabéis que para el cracking (tanto en linux como en windows) la base en ensamblador es fundamental!!!.

1.a) Strace. Vamos a ver las llamadas al sistema, ya que al no tener librerías dinámicas (motivo por el cual ni pongo ltrace) su relación con el sistema se debe realizar con syscall:

```
$ strace -i -o strace.txt ./888
```

Si miramos el archivo **strace.txt** vemos:

```
[    7f3080c69b37] execve("./888", ["./888"], [/* 34 vars */]) = 0
[        8048377] signal(SIGTRAP, 0x804820a) = 0 (SIG_DFL)
[        8048377] write(1, "NO\r\n", 4) = 4
[        8048377] _exit(0)            = ?
```

Como vemos ha dado poca información, observamos como salta una señal en **0x804820a (SIGTRAP)** y después llama a **write** para mostrarnos el mensaje **"NO"**.

Las llamadas al sistema **signal**, **write** e incluso **exit** la veremos cuando comencemos con el debugger.

1.b) GDB, seguimos con este gran debugger, veremos muchas opciones que no habíamos utilizado y profundizaremos en el manejo del **registro de flags** o **registro de banderas**; que como veremos da mucho juego.

Primero cargaremos el programa en gdb:

```
$ gdb -q 888
```

Ahora podemos colocar los **display** que nos gustan, para trabajar con mas información:

```
gdb $ display/5i $pc
gdb $ display $esi
gdb $ display $edx
gdb $ display $ecx
gdb $ display $ebx
gdb $ display $eax
```

En este programa he decidido poner todos estos **displays**, esencialmente porque vamos a trabajar con las llamadas al sistema.

La forma de hacer una llamada al sistema en linux es mediante la interrupción 80, para ello debemos llevar un orden:

1. Poner en **EAX** el número de la llamada al sistema. Para saber el número debemos mirar el archivo **unistd.h** que normalmente esta en:

```
/usr/include/asm/unistd.h
```

En un sistema de **64 bits** nos vamos encontrar dos versiones de este archivo:

```
/usr/include/asm/unistd_32.h
/usr/include/asm/unistd_64.h
```

2. Poner los parámetros de la llamada en los registros en este orden:

EBX - ECX - EDX - ESI - EDI

Para saber los parámetros podemos buscar la syscalls en las páginas de **man** o directamente en la página de kernel.org:

http://www.kernel.org/doc/man-pages/online_pages.html

3. Provocar la interrupción con **INT 80**.

Mas adelante veremos varios ejemplos prácticos.

Aunque se que no va a ser de utilidad, lo primero que hice fue poner un **breakpoint** en el **entrypoint *0x0804823f** e iniciamos la ejecución con **"r"**:

```
gdb> b *0x0804823f
gdb> r
```

```
Program received signal SIGTRAP, Trace/breakpoint trap.

0x080482d3 in ?? ()
6: $eax = 0x80482d2
5: $ebx = 0x5
4: $ecx = 0x804820a
3: $edx = 0x0
2: $esi = 0x0
1: x/5i $pc
0x80482d3:  mov    eax,ds:0x8048380
0x80482d8:  test   eax,eax
0x80482da:  je     0x8048325
0x80482dc:  sub    eax,0x53504f4f
0x80482e1:  call   eax
gdb>
```

Como vemos hemos recibido una señal de **SIGTRAP**; normalmente este sucede cuando ponemos un breakpoint, pues el debugger coloca en la dirección que nos interese el opcode **CC** que provoca una interrupción 3 (**int 3**), que el debugger captura y entonces nos muestra el programa parado en ese punto.

En este caso, es el programador el que lo ha colocado, seguramente para detectar el debugger y evitar que lo podamos depurar. SI miramos la dirección que se acaba de ejecutar **0x80482d2** vemos lo que acabo de comentar:

```
gdb> x/b 0x80482d2
0x80482d2:  0xcc
gdb> x/i 0x80482d2
0x80482d2:  int3
```

Como ya sabemos con **"x"** vemos la memoria, en este caso con **"b"** vemos los bytes y con **"i"** la instrucción en ensamblador que corresponde.
Si miramos el desensamblado del programa original, **ndisasm.txt**, y buscamos los bytes **CC**, los encontramos en esta dos direcciones; **0x8048221** y **0x8048209**; esto solo nos sirve para comprobar que el programa se automodifica en ejecución y habría que tener una imagen del programa en este momento. Para ello vamos a usar el comando **dump** de gdb:

```
gdb> help dump
Dump target code/data to a local file.

List of dump subcommands:

dump binary -- Write target code/data to a raw binary file
dump ihex -- Write target code/data to an intel hex file
dump memory -- Write contents of memory to a raw binary file
dump srec -- Write target code/data to an srec file
dump tekhex -- Write target code/data to a tekhex file
dump value -- Write the value of an expression to a raw binary file

Type "help dump" followed by dump subcommand name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
gdb> dump memory dump.bin 0x8048000 0x8048398
```

De esta manera salvamos el programa que esta en memoria (**dump memory**) a un archivo llamado

dump.bin, y como sabemos por el formato **ELF,** el segmento que esta en memoria comienza en **0c8048000** y termina **0x8048398.** Ahora con nuestro editor hexadecimal y desensamblador **hte** podemos verlo y analizarlo.

Estamos parados en **0x80482d2,** si estudiamos el desensamblado que tenemos a continuación vemos que la cosa no pinta bien:

```
      0x80482d3:  mov     eax,ds:0x8048380 ; mueve a eax el valor que hay en
0x8048380. Si miramos con gdb vemos que no hay ningún valor:
```

```
gdb> x/4b 0x8048480
0x8048480:  0x00  0x00  0x00  0x00
```

```
      0x80482d8:  test    eax,eax ; con test comprueba si EAX vale 0, como es
nuestro caso
      0x80482da:  je      0x8048325 ; en nuestro caso continuaremos en
0x8048325
      0x80482dc:  sub     eax,0x53504f4f ; al valor que había en 0x8048480 (0)
le restamos 0x53504f4f; en este caso nos daría un valor negativo
      0x80482e1:  call    eax ; la resta anterior nos debe dar una dirección del
programa donde continuar,  que de momento no podemos saber.
```

Si de todos modos, quisiéramos hacer pruebas y continuar sin que se produzca el salto "**je 0x8048325**" podríamos hacerlo parcheando el código; pero en general esto es muy pesado, lo mas cómodo es cambiar el flag adecuado para que no se produzca el salto y eso nos da pie para explicar el registro **eflags** y como manejarlo en **gdb**:

***************************EFLAGS******************************

Los registros que ya conocemos son los registros de propósitos generales (**eax, ebx**....) , pero hay un registro muy especial, llamado **registro de flags** o **registro de banderas**, donde algunos de sus bits tienen un significado especial según este activo **(1)** o no **(0)** para la ejecución del programa; ya que su valor sirve para tomar decisiones.

Si miramos en este momento su valor (antes de ejecutar el **"test eax,eax"**) mediante la orden "**info registers**":

```
gdb> info registers
eax            0x0     0x0
ecx            0x804820a    0x804820a
edx            0x0     0x0
ebx            0x5     0x5
esp            0xffffd4c4   0xffffd4c4
ebp            0x0     0x0
esi            0x0     0x0
edi            0x0     0x0
eip            0x80482d8    0x80482d8
eflags         0x206   [ PF IF ]
cs             0x23    0x23
ss             0x2b    0x2b
ds             0x2b    0x2b
es             0x2b    0x2b
fs             0x0     0x0
gs             0x0     0x0
gdb> python print bin(0x206)
0b1000000110
```

Como vemos su valor es 0x206 y **gdb** nos indica que esta activo el **flag de paridad** (PF) y el **flag de interrupciones** (IF). Como vamos a trabajar a nivel de bits, debemos saber el valor binario de **0x206**, que en este caso es **0b1000000110.** Tanto en sistemas de 32 bits como en 64 bits, en este registro se miran los mismos bits, solo que en cada caso sera un registro de 32 o de 64 bits. Como nuestro crackme es de 32 podemos representarlo así:

```
N°                                              B A 9 8 7 6   4   2   0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|0|0|0|0|1|1|0
                                                O D I T S Z   A   P   C

C: Acarreo P: Paridad A: Acarreo Auxiliar Z: Cero S: Signo
T: Trampa I: Interrupciones habilitadas D: Direccion O: Overflow
```

Las siguientes instrucciones van a afectar a los flag de esta manera:

```
0x80482d8:  test   eax,eax    ;como EAX=0 la comparación activa el bit de Cero (Z)
0x80482da:  je     0x8048325  ; si el bit de Cero esta a 1 se produce el salto
0x80482dc:       sub    eax,0x53504f4f
```

Como sabéis muchas instrucciones pueden afectar a los flags, tanto sumas, divisiones, restas...., y la mayoría suelen modifican a los registros implicados (sub eax,0x10, quita 0x10 al valor de eax). En este caso solo queremos saber si **EAX** es igual a **0** o no, pero sin modificar EAX, para ello en ensamblador hay dos instrucciones, **test** y **cmp**. **Test** (comprobar) lo que realiza es un **AND lógico** entre los valores que indiques en la instrucción, mientras que **cmp** (comparar) realiza una resta (**SUB**) entre los valores que indiques. En ambos caso solo se afecta al **registro de flags**, dándonos la información necesaria para actuar mediante un **salto condiciona**l. Normalmente la combinación de estas instrucciones es la forma que tiene el compilador de emular construcciones de mas alto nivel, como " IF......THEN....", "FOR" o "WHILE", en ensamblador.

Si continuamos con **"ni"** para que se ejecute el "**test eax,eax**", el registro de flags ha cambiado:

```
eflags:00000246
gdb> python print bin(0x246)
0b1001000110
gdb> p $eflags
$2 = [ PF ZF IF ]
```

En este caso:

```
N°                                              B A 9 8 7 6   4   2   0
0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|1|0|0|1|0|0|0|1|1|0
Flags                                           O D I T S Z   A   P   C
```

Para manejar los **eflags** nos va a hacer falta tener claro las operaciones lógicas:

AND			OR			XOR		
0	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	1	1
1	0	0	1	0	1	1	0	1
1	1	1	1	1	1	1	1	0

NOT	
0 1	1 0

Como veis cada operación lógica puede tener una finalidad, normalmente con **AND** vemos el estado del bit; si enfrentamos un bit a 1 con un bit desconocido, sabremos según el resultado que ese bit esta activado si nos da **1** y que no está activado si nos da **0**. **OR** nos sirve siempre para activar cualquier bit, de modo que un bit a 1 transformará siempre el bit desconocido a **1** con esta operación. Con **XOR** vamos a hacer todo lo contrario, sirve para desactivar un bit (ponerlo a 0), de modo que un bit a 1 enfrentado a un bit cualquiera con XOR siempre lo dejará a **0.**

Teniendo esto claro, ya podemos intentar ver el estado de los bits del **eflags** con **gdb**:

```
gdb> p (($eflags >> 6) & 1 );ZF
$3 = 0x1
```

De esta manera vemos que el **flag cero (Z)** está activado por la instrucción **"test eax,eax"**. Con **p** "print" vemos el resultado de **(($eflags >>6) & 1)**; primero obtenemos el bit sexto de **$eflags**, ya sabemos que >> es desplazamiento de bits, en este caso miramos el bit **6** del registro **eflags** sin modificarlo; y como no sabemos su estado le hacemos un **AND lógico** (**&**) con **1**; solo si esta a 1 devolverá un **0x1** como resultado. En este caso como EAX vale 0, sabíamos que estaría activado.

De igual manera, podemos averiguar el estado de cualquier bit:

```
gdb> p (($eflags >> 2) & 1 );PF
$10 = 0x1
gdb> p (($eflags >> 9) & 1 );IF
$11 = 0x1
gdb> p (($eflags >> 1) & 1 );No se utiliza, pero también esta a 1
$12 = 0x1
gdb> p (($eflags >> 0) & 1 );CF no esta activado
$13 = 0x0
```

Ahora que ya sabemos averiguar el estado del bit; vamos a ver como se pueden modificar. Para ello nos hace falta un bit activado en la posición que nos interese para cada **flags**. Los valores se obtienen colocando un 1 en la posición que nos interese y los demás valores serán 0 para no tocar ningún bit mas:

```
B A 9 8 7 6   4   2   0
0 0 0 0 0 0 0 0 0 0 0 1 = 0x1
0 0 0 0 0 0 0 0 0 1 0 0 = 0x4
0 0 0 0 0 0 0 1 0 0 0 0 = 0x10 (16)
0 0 0 0 0 1 0 0 0 0 0 0 = 0x40 (64)
0 0 0 0 1 0 0 0 0 0 0 0 = 0x80 (128)
0 0 0 1 0 0 0 0 0 0 0 0 = 0x100 (256)
0 0 1 0 0 0 0 0 0 0 0 0 = 0x200 (512)
0 1 0 0 0 0 0 0 0 0 0 0 = 0x400 (1024)
1 0 0 0 0 0 0 0 0 0 0 0 = 0x800 (2048)
O D I T S Z   A   P   C
```

Para el flag **Z** nos hace falta el valor **0x40 (64)**, de modo que podemos hacer un **XOR lógico "^"** para desactivarlo y si nos hiciera falta, un **OR lógico (|)** para activarlo:

```
gdb> set $eflags = $eflags^0x40          ;hacemos un XOR lógico con 0x40
gdb> p $eflags
$5 = [ PF IF ]

gdb> set $eflags = $eflags|0x40          ; hacemos un OR lógico con 0x40
gdb> p $eflags
$6 = [ PF ZF IF ]
```

En **gdb** tenemos que usar la orden **set** para darle un nuevo valor a la variable **$eflags** y con " **p $eflags"** confirmamos el cambio.

Acordarse de todo esto es complicado, así que vamos a utilizar macros en **.gdbinit** para tener el cambio de cualquier flag muy a mano. Una macro en gdbinit debe comenzar por definir su nombre con **"define"**, que será la orden que debemos recordar, después poner todos los comandos necesarios y terminar con **"end"**:

```
define cfz
 if (($eflags >> 6) & 1 )
  set $eflags = $eflags&~0x40
 else
  set $eflags = $eflags|0x40
 end
end
document cfz
change Zero Flag
end
```

Como veis, este código no es mio, lo he tomado de un gdbinit que me he bajado de la red, **dotgdbinit**; donde aplican este método para cambiar los flags y ya están todos configurados, como veis en este caso es **cfz** (**cambiar flag z**) y utiliza un bucle **IF...ELSE** para cambiar el flag; primero con **AND** comprueba el estado del flag; si es **1** lo pondrá a 0 con **XOR (&~)** y si es un **0** lo activará con **OR (|).**

Por cierto, es raro el símbolo que utilizan para XOR, **"&~"** (no se si es símbolo att ¿?) pero como habéis visto en las ordenes que he puesto en **gdb**, el símbolo **"^"** para XOR funciona perfectamente y podéis sustituirlo sin problemas.

Además, **dotgdbinit** trae muchas comodidades para trabajar con gdb, yo solo tuve que añadirle al final la opción de ensamblador en formato intel "**set disassembly-flavor intel**" para empezar a utilizarlo. Si os interesa lo he subido aquí:

http://www.4shared.com/document/oUD8TJ8S/dotgdbinit.html

Una vez descargado, tenéis que ponerlo en vuestra **/home** con el nombre **.gdbinit** (renombrar el anterior por si no os gusta) y veréis los cambios al ejecutar **gdb:**

```
  eax:080481B4 ebx:00000005  ecx:0804820A  edx:00000000     eflags:00000216
  esi:00000000  edi:00000000    esp:FFBF5724  ebp:00000000     eip:080482E1
  cs:0023   ds:002B   es:002B   fs:0000   gs:0000   ss:002B          o d I t s z A P c
[002B:FFBF5724]---------------------------------------------------------[stack]
FFBF5754 : DF 67 BF FF  FF 67 BF FF - 0B 68 BF FF  FB 6C BF FF .g...g...h...l..
FFBF5744 : 16 67 BF FF  67 67 BF FF - A2 67 BF FF  B5 67 BF FF .g..gg...g...g..
FFBF5734 : 97 66 BF FF  C4 66 BF FF - FB 66 BF FF  06 67 BF FF .f...f...f...g..
FFBF5724 : 56 66 BF FF  00 00 00 00 - 62 66 BF FF  84 66 BF FF Vf......bf...f..
[002B:080481B4]---------------------------------------------------------[ data]
```

```
080481B4 : 8F 05 88 83  04 08 58 09 - C0 74 07 35  41 52 47 53 ......X..t.5ARGS
080481C4 : 75 F4 85 C0  74 01 E8 8B - 1C 24 09 DB  74 2E E8 7B u...t....$..t..{
[0023:080482E1]-----------------------------------------------------[ code]
0x80482e1:    jmp   0x80482e1
0x80482e3:    push  0x8048366
0x80482e8:    push  0x8048225
0x80482ed:    push  0x80481a1
0x80482f2:    push  0x8054c54
0x80482f7:    bt    ecx,0x16
-------------------------------------------------------------------------
```

Con la orden **"help user"** tendréis una ayuda sobre todas las macros que incorpora, para cambiar cada **flags** hay una orden específica y fácil de recordar:

cfa -- Change Auxiliary Carry Flag
cfc -- Change Carry Flag
cfd -- Change Direction Flag
cfi -- Change Interrupt Flag
cfo -- Change Overflow Flag
cfp -- Change Carry Flag
cfs -- Change Sign Flag
cft -- Change Trap Flag
cfz -- Change Zero Flag

****

Continuamos con el debugger; esta vez no cambiaremos el flag Z para evitar el salto , pues de todas maneras no podríamos continuar sin el valor que falta en 0x8048380 . Por tanto continuamos después de " **je 0x8048325** ":

```
0x08048325 in ?? ()
6: $eax = 0x0
5: $ebx = 0x5
4: $ecx = 0x804820a
3: $edx = 0x0
2: $esi = 0x0
1: x/5i $pc
0x8048325:  int    0x3
0x8048327:  mov    eax,0xdeadc0de
0x804832c:  push   eax
0x804832d:  mov    eax,ds:0x8048384
0x8048332:  mov    ebx,esp
```

Nos encontramos otra interrupción, como la anterior, la maneja **gdb** y no el programa; seguramente el programa tendrá una rutina que estamos evitando; por lo que estas interrupciones son un método antidebugger donde hemos caido.
Por cierto, el utilizar el valor hexadecimal **0xdeadc0de** también parece indicarnos que no vamos por buen camino, que amable.....

Para no alargar el tema, vamos a ir viendo las zonas de código mas interesantes, normalmente vamos siguiendo con la orden **"ni"** y analizando la memoria del programa con **"x"**, hasta que llegamos a la primera llamada al sistema:

```
0x08048375 in ?? ()
9: $edi = 0x0
8: $esi = 0x0
7: $edx = 0x4
6: $ecx = 0xffffd4b0
5: $ebx = 0x1
4: $eax = 0x4
1: x/5i $pc
0x8048375:  int    0x80
0x8048377:  ret
0x8048378:  add    al,BYTE PTR [eax]
0x804837a:  add    BYTE PTR [eax],al
0x804837c:  add    BYTE PTR [eax],al
```

Venimos de "**0x8048359: call   0x8048375**" y justo antes ha ido preparando los registros para la syscall. Primero para saber que función es, miramos el valor de **EAX,** en este caso **0x4**, y en el archivo **unistd_32.h** vemos que es write:

#define __NR_write   4

Para saber su estructura lo miramos con la orden **"man 2 write"**:

"NOMBRE
write - escribe a un descriptor de fichero

SINOPSIS
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t num);

DESCRIPCIÓN
write escribe hasta num bytes en el fichero referenciado por el descriptor de fichero fd desde el búfer que comienza en buf. POSIX requiere que un read() que pueda demostrarse que ocurra después que un write() haya regresado, devuelva los nuevos datos. Observe que no todos los sistemas de ficheros son conformes con POSIX.
VALOR DEVUELTO
En caso de éxito, se devuelve el número de bytes escritos (cero indica pues que no se ha escrito nada). En caso de error, se devuelve -1 y se pone un valor apropiado en errno. Si num es cero y el descriptor de fichero se refiere a un fichero regular, se devolverá 0 sin que se cause ningún otro efecto. Para un fichero especial, los resultados no son transportables."

Lo importante aquí es **(int fd, const void *buf, size_t num)**; de modo que necesitamos primero **"int fd"** el descriptor del fichero donde vamos a escribir, este valor ira a **EBX**, después **"const void *buf"** que es la dirección del buffer donde esta el texto que vamos a mostar, ira en **ECX** y por último, **"size_t num"** el número de bytes que vamos a escribir en **EDX.** Siguiendo este orden, ya podemos ejecutar la **interrupción 80** para que realice la llamada al sistema.

En assambler la cosa podría quedar así:

```
mov eax,linux.sys_write;
mov ebx, fd;
mov  ecx, buf;
mov edx, count;
int( $80 );
```

En nuestro caso, podemos ver que **EBX** tiene el valor **1,** pues en linux es el descriptor de salida, y normalmente (si no hay redirección ">") se refiere a la pantalla.

****
Descriptores básicos:
stdin: Archivo estandar de entrada, su descriptor es el **0** y es donde los programas leen su entrada.
stdout: Archivo estandar de salida, su descriptor es el **1** y es donde los programas envían sus resultados.
stderr: Archivo estandar de error, su descriptor es el **2** y es a donde los programas envían sus salida de errores.

****

En **ECX** tenemos el valor **"0xffffd4b0 ",** la dirección del **buffer** que podemos ver con **x**:

```
gdb $ x 0xffffd4c0
0xffffd4c0:0x0a0d4f4e
gdb $ x/s 0xffffd4c0
0xffffd4c0:  "NO\r\n\"\326\377\377"
```

Por último, en **EDX** tenemos el contador de bytes , que tiene un valor de **4**, pues como se ve en el buffer va a escribir **"NO"** con un salto de linea **\n** y un retorno de carro **\r,** que son los 4 bytes.

Si continuamos con **"ni"** lo podemos ver en gdb:

```
gdb $ ni
NO
```

Continuamos y llegamos a la última llamada al sistema:

```
0x08048375 in ?? ()
6: $eax = 0x1
5: $ebx = 0x0
4: $ecx = 0xffffd4c0
3: $edx = 0x4
2: $esi = 0x0
1: $pc = (void (*)()) 0x8048375
```

En este caso **EAX** tiene el valor de 1:

#define __NR_exit 1

Esta función no tiene mas parámetros, en ensamblador sería algo así:

```
mov eax,  linux.sys_exit;
int( $80 );
```

Con esta orden se termina el proceso actual, como podemos ver con gdb:

```
gdb $ ni

Program exited normally.
```

Como ya sospechabamos el programa nos ha detectado y nos dice que "NO", por tanto el breakpoint en el entrypoint no sirve y debemos buscar otro punto de entrada anterior.

Si nos fijamos ambas llamadas al sistema que hemos visto tenían la **interrrupción 80** en **0x8048375** y si además buscamos sus opcodes **0xcd 0x80** en el archivo que hemos dumpeado (**dump.bin**) solo aparecen en esa posición. Por tanto es de suponer que se utilicen es todas las llamadas al sistema, vamos a verlo cargando de nuevo el programa y le colocaremos un breakpoint en 0x8048375:

```
gdb $ exec 888
gdb $ display/5l $pc
gdb $ display $esi
gdb $ display $edx
gdb $ display $ecx
gdb $ display $ebx
gdb $ display $eax
gdb $ b *0x8048375
Breakpoint 1 at 0x8048375
gdb $ r
```

Inmediatamente para en la primera llamada al sistema:

```
Breakpoint 1, 0x08048375 in ?? ()
6: $eax = 0x30
5: $ebx = 0x5
4: $ecx = 0x804820a
3: $edx = 0x0
2: $esi = 0x0
1: $pc = (void (*)()) 0x8048375
```

EAX tiene un valor de **0x30** que es **48** en decimal, pues es la forma como lo encontramos en **unistd_32.h**:

#define __NR_signal 48

Ahora vamos a ver su estructura con **"man 2 signal"**:

Por cierto, si os preguntáis por el **2** después de **man**, hay que saber que estas funciones a veces tienen muchas entradas en el manual de linux **"man"**, de forma que si queremos verlas todas se podría utilizar la orden **"man -a signal"** y nos mostrará todas las entradas una tras otra, pero en casos concretos podemos averiguar donde buscar con el manual de man **"man man"** que nos dice:

"La siguiente tabla muestra los números de sección del manual y los tipos de páginas que contienen.

1 Programas ejecutables y guiones del intérprete de órdenes
2 Llamadas del sistema (funciones servidas por el núcleo)
3 Llamadas de la biblioteca (funciones contenidas en las bibliotecas del sistema)
4 Ficheros especiales (se encuentran generalmente en /dev)
5 Formato de ficheros y convenios p.ej. I/etc/passwd
6 Juegos
7 Paquetes de macros y convenios p.ej. man(7), groff(7).
8 Órdenes de admistración del sistema (generalmente solo son para root)
9 Rutinas del núcleo [No es estándar] "

Por tanto con **"man 2 signal"** vemos lo que nos interesa:

NOMBRE
signal - manejo de señales en ANSI C

SINOPSIS
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

DESCRIPCIÓN
La llamada al sistema signal() instala un nuevo manejador de señales para la señal con número signum. El manejador de señales queda establecido a sighandler que puede ser una función especificada por el usuario o bien SIG_IGN o SIG_DFL.

Cuando llega una señal con número signum ocurre lo siguiente. Si el manejador correspondiente está establecido a SIG_IGN, la señal es ignorada. Si el manejador está establecido a SIG_DFL, se realiza la acción por defecto asociada a la señal (vea signal(7)). Finalmente, si el manejador está establecido a una función sighandler lo primero que se hace es o bien restablecer el manejador a SIG_DFL o un bloqueo de la señal que depende de la implementación, invocando después a sighandler con el argumento signum.

Usar una función manejadora de señales para una señal se llama "atrapar la señal".

Por tanto la llamada tiene dos parametros; **(int signum, sighandler_t handler)**; donde **"int signum"** es la señal para la que se crea el manejador de señales, en este caso esta en **EBX** y es **0x5.** Para saber a que señal corresponde, la mejor opción es buscar el programa de linux que se encarga de las señales, **kill**, y en concreto con la opción **-l** nos lista todas las que hay:

```
$ kill -l
 1) SIGHUP   2) SIGINT   3) SIGQUIT  4) SIGILL   5) SIGTRAP
 6) SIGABRT  7) SIGBUS   8) SIGFPE   9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT     17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG  24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM     27) SIGPROF 28) SIGWINCH      29) SIGIO   30) SIGPWR
31) SIGSYS  34) SIGRTMIN      35) SIGRTMIN+1    36) SIGRTMIN+2    37) SIGRTMIN+3
38) SIGRTMIN+4    39) SIGRTMIN+5    40) SIGRTMIN+6    41) SIGRTMIN+7    42) 
SIGRTMIN+8
43) SIGRTMIN+9    44) SIGRTMIN+10   45) SIGRTMIN+11   46) SIGRTMIN+12   47) 
SIGRTMIN+13
48) SIGRTMIN+14   49) SIGRTMIN+15   50) SIGRTMAX-14   51) SIGRTMAX-13   52) 
SIGRTMAX-12
53) SIGRTMAX-11   54) SIGRTMAX-10   55) SIGRTMAX-9    56) SIGRTMAX-8    57) 
SIGRTMAX-7
58) SIGRTMAX-6    59) SIGRTMAX-5    60) SIGRTMAX-4    61) SIGRTMAX-3    62) 
SIGRTMAX-2
63) SIGRTMAX-1    64) SIGRTMAX
```

Ahora solo queda saber cual es el motivo para que se de la señal **SIGTRAP**, esto lo vemos en otra página del manual **" man 7 signal"** :

SIGTRAP 5 Core Trampa de traza/punto de ruptura

Por tanto la señal se produce con los puntos de ruptura , que en este caso los pone el programa al colocar el byte **CC** en diferentes zonas, como podemos ver en este código que nos encontrarnos un poco después de pasar la llamada al sistema de signal:

```
4: $ecx = 0x804820a
3: $ebx = 0x80482d2
2: $eax = 0xcc
1: x/5i $pc
0x80482bf:  mov BYTE PTR [ebx],al ;introduce un punto de ruptura en 0x80482d2
0x80482c1:  xchg   ebx,eax
0x80482c2:  pop    ebx
0x80482c3:  dec    eax
0x80482c4:  jne    0x80482d0
```

Después solo tiene que llevar el programa hasta ese punto:

```
0x080482cc in ?? ()
4: $ecx = 0x804820a
3: $ebx = 0x5
2: $eax = 0x80482d2
1: x/5i $pc
0x80482cc: push   eax
0x80482cd: ret
```

Normalmente la orden **ret** se utiliza para volver de una función, (pues en la cima de la pila se ha subido la dirección de regreso, cuando se ha entrado en una función con "**Call**"); pero también puede ser una instrucción de salto con solo subir la dirección donde queremos que vaya el programa a la pila (**push eax)** y posteriormente con "**ret**" el programa continuara, en nuestro caso en **0x80482d2**, donde ha colocado el punto de interrupción y se producirá una **SIGTRAP**.

Con la syscall **signal** se ha colocado un manejador de está señal, que es **sighandler_t handler**, que tenemos en **ECX** y es **0x804820a**. Por tanto el programa sin debugger, genera esa señal y pasa por esa zona, mientras que con debugger no pasamos, pues esa señal la maneja el depurador y nos pillan.

Si vemos el código en **0x804820a**:

```
gdb $ disas 0x804820a  0x8048250
Dump of assembler code from 0x804820a to 0x8048250:
0x0804820a: inc    DWORD PTR ds:0x804837c
0x08048210: mov    DWORD PTR ds:0x8048380,0x5b54d103
0x0804821a: sub    eax,eax
0x0804821c: inc    eax
0x0804821d: neg    eax
0x0804821f: js     0x8048223
0x08048221: int3
```

Primero incrementará en 1 el valor en **0x804837C** (que después comprueba para saber que el programa ha pasado por aquí) y después mueve el valor **0x5b54d103** a una zona de memoria **0x8048380**, esta parte es complementaria con la zona que vimos al principio:

```
            0x80482d3:  mov    eax,ds:0x8048380 ; si no hay debugger el valor en
ds:ox8048380 es 0x5b54d103
            0x80482d8:  test   eax,eax
            0x80482da:  je     0x8048325
            0x80482dc:  sub    eax,0x53504f4f ; si hacemos la resta 0x5b54d103 -
0x53504f4f = 0x80481B4
            0x80482e1:  call   eax ; quedaría como Call 0x80481B4
```

De este modo sabemos como salvar la primera protección, podemos realizar los cambios que se realizan en **804820A** y continuar con el debugger (esta opción me dio problemas) o para estar mas seguros, hacer que el programa sin debugger llegue a **0x80482E1 (call eax)**, hacer que pare allí con un bucle infinito y entonces engancharnos al proceso con **gdb (attach)**. Para esto vamos a utilizar la copia del programa **888.copia**, la editamos con **hte** y nos iremos al byte **0x2E1**:

```
080482E1  FFD0       call ax------> cambiar ffd0 por ebfe
```

Los opcodes **EB FE** indican un salto incondicional hacia el mismo sitio de donde es ejecutado, por lo que el programa queda en un bucle infinito; en nuestro caso seria un **"jmp 80482E1"** o como quedo en **hte:**

```
000002e1 EBFE        jmp         0x2e1
```

De esta manera el programa pasará la zona antidebugger (int3) y nosotros seguiremos con la parte que nos interesa , eso si, antes hay que volver a cambiar los bytes **EB FE** por los originales **FF D0.**

Por tanto en una consola ejecutamos el programa **888.copia** y en otra nos attacheamos con **gdb**:

```
$ ps ax | grep 888.copia
32185 pts/3    R+     0:13 ./888.copia
32296 pts/2    R+     0:00 grep 888.copia
$ gdb -q 888.copia 32185

Reading symbols from
/home/juanjo/Descargas/crk/crackmes_linux/888/888.copia...(no debugging
symbols found)...done.
Attaching to program:
/home/juanjo/Descargas/crk/crackmes_linux/888/888.copia, process 32185
0x080482e1 in ?? ()
```

Primero debemos saber el **PID** de 888.copia, para ello utilizamos la orden **"ps ax"** con el filtro **"grep 888.copia"**, que nos da como PID **32185**. Después con **gdb** cargamos el programa seguido de su PID para que se pueda enganchar al proceso. Como veis el programa queda parado en el bucle que habíamos colocado en **0x80482E1**.

Ahora solo queda restablecer los bytes originales para poder continuar:

```
gdb $ set {short}0x80482e1=0xd0ff
gdb $ x/2b 0x80482e1
0x80482e1:  0xff  0xd0
gdb $ x/i 0x80482e1
0x80482e1:  call   eax
gdb $ p $eax
$2 = 0x80481b4
```

Solo comentar que **{short}** permite el cambio de dos bytes y que esos dos bytes debido al formato **"little indian"** se deben poner al contrario de como se colocan en memoria: **0xd0ff.** De todos modos, siempre debemos asegurarnos que todo esta correcto con **"x/i 0x80482e1"**.

Como vemos la dirección que nos señala **EAX** es la misma que habíamos calculado: **0x80481B4**, que sera por donde continuamos con la orden **"ni"** hasta que llegamos a este punto:

```
_____________________________________________________________________________
     eax:00000000 ebx:FFCA068F  ecx:FFFFFFFF  edx:00000000  eflags:00000246
     esi:00000000 edi:FFCA068F  esp:FFCA02B8  ebp:00000000     eip:0804805D
     cs:0023  ds:002B  es:002B  fs:0000  gs:0000  ss:002B    o d I t s Z a P c
[002B:FFCA02B8]-------------------------------------------------------[stack]
```

```
FFCA02E8 : 0B 08 CA FF  FB 0C CA FF - 21 0D CA FF  71 0D CA FF ........!...q...
FFCA02D8 : A2 07 CA FF  B5 07 CA FF - DF 07 CA FF  FF 07 CA FF ................
FFCA02C8 : FB 06 CA FF  06 07 CA FF - 16 07 CA FF  67 07 CA FF ...........g....
FFCA02B8 : D7 81 04 08  00 00 00 00 - B1 06 CA FF  C4 06 CA FF ................
[002B:FFCA068F]---------------------------------------------------------[ data]
FFCA068F : 4F 52 42 49  54 5F 53 4F - 43 4B 45 54  44 49 52 3D ORBIT_SOCKETDIR=
FFCA069F : 2F 74 6D 70  2F 6F 72 62 - 69 74 2D 6A  75 61 6E 6A /tmp/orbit-juanj
[0023:0804805D]---------------------------------------------------------[ code]
0x804805d:  repnz scas al,BYTE PTR es:[edi]
0x804805f:  not    ecx
0x8048061:  dec    ecx
0x8048062:  mov    eax,ecx
0x8048064:  xchg   DWORD PTR [esp+0x4],edi
0x8048068:  ret    0x4
--------------------------------------------------------------------------------
---
0x0804805d in ?? ()
```

Como veis ya estamos utilizando el nuevo .**gdbinit,** la siguiente instrucción que se va a ejecutar es **"repnz scas al, BYTE PTR es:[edi]"**

L a instrucción "**scas** " sirve para examinar una cadena, de modo que va tomando los valores señalados por **EDI** , byte a byte, y los va cargando en **AL**; **"scas al, BYTE PTR es:[edi]"** . con el prefijo **repnz** lo que conseguimos es que **scas** se repita mientras que no sea cero y como sabemos que una string debe acabar en 00; lo que haremos es ir tomando todas las strings señaladas por **EDI**.

Si miramos la zona de memoria que nos señala **EDI:**

```
gdb $ x/100s $edi
0xffc506b2:  "SSH_AGENT_PID=2307"
0xffc506c5:  "GPG_AGENT_INFO=/tmp/seahorse-qYhPJn/S.gpg-agent:2321:1"
0xffc506fc:  "TERM=xterm"
0xffc50707:  "SHELL=/bin/bash"
0xffc50717:  "XDG_SESSION_COOKIE=2a17fb62c7c71781076f0b7200000006-
1318504216.442190-1626032974"
0xffc50768:  "GTK_RC_FILES=/etc/gtk/gtkrc:/home/juanjo/.gtkrc-1.2-gnome2"
0xffc507a3:  "WINDOWID=85983392"
0xffc507b5:  "GNOME_KEYRING_CONTROL=/tmp/keyring-k3UYNz"
0xffc507df:  "GTK_MODULES=canberra-gtk-module"
0xffc507ff:  "USER=juanjo"
..................................................................
```

Continua, pero con esto es suficiente para saber que estamos viendo las **variables de la shell,** esto además lo podemos comprobar con la orden **"export -p"** en la consola sin argumentos, donde veremos las mismas variables.

Esto es importante, porque ya sabemos como interactuar con el programa, seguramente tendremos que incluir un valor especial en esta variables para resolverlo.

Si seguimos observando el código, vamos viendo que busca el programa:

```
0x80481d7: cmp    eax,0x4
0x80481dc: jl     0x80481cb
0x80481de: cmp    eax,0x40
0x80481e3: jg     0x80481cb
```

Con la instrucción **"repnz scas"** nos queda el número de caracteres que hemos examinado hasta el valor cero en **ECX**, por tanto tenemos la longitud de la string en **ECX**, el programa lo pasa a **EAX** y en la zona anterior comprueba que la cadena no sea menor de **0x4** ni mayor de **0x40 (64)**.

```
0x80481ed:  cmp    ecx,0x79656b
0x80481f3:  jne    0x80481cb
```

En **ECX** ha introducido los tres primeros caracteres de la cadena que está analizando el programa, en este caso **0x42524f (BRO)** que corresponde a la variable **"ORBIT_SOCKETDIR=/tmp/orbit-juanjo"** y lo compara con **0x79656B (yek)**, por tanto esta buscando una cadena en la variable de la shell que comience con **key**. Como no lo encuentra de momento terminamos este primer intento.

Para introducir una variable nueva en la shell se utiliza la orden **export**, seguida del nombre de la clave y con el signo "=" se le asigna un valor, en este caso pruebo de esta manera:

```
$ export key=123456789
```

Ahora reiniciamos **888.copia** y nos enganchamos con **gdb**. Ahora podemos continuar viendo lo que espera el programa de la key:

```
0x080480da: cmp    al,0x30
0x080480dc: jl     0x80480f3
0x080480de: cmp    al,0x39
0x080480e0: jg     0x80480f3
```

Aquí vemos que solo busca números, tanto si el código ascii es menor de **30 (0)** o mayor de **39 (9)** nos vamos fuera. En este caso hemos tenido suerte, seguimos:

```
0x080480e2: sub    al,0x30
0x080480e4: xchg   ebx,eax
0x080480e5: mul    ecx
0x080480e7: xchg   ebx,eax
0x080480e8: add    ebx,eax
0x080480ea: jmp    0x80480d5
```

Esta parte es importante, la llamare **Función_1** y hay que tener claro que es lo que hace:

`0x080480e2:    sub    al,0x30` ; en **AL** esta nuestro primer valor (**1**), por lo que al restarle 30 va a dejar el mismo valor en hexadecimal que hemos puesto en decimal (**0x1**)
`0x080480e4:    xchg   ebx,eax ;` cambia el valor que hay en **EAX** y en **EBX** respectivamente.
Ahora en EAX esta una variable que llamaremos **<TOTAL>** y que al principio vale **0** y en **EBX** el valor **0x1.**
`0x080480e5:    mul    ecx` ; la multiplicación se realiza entre **EAX** (donde esta <TOTAL>) y el operando que pongamos en la instrucción, en este caso **ECX**, que siempre contendrá una constante **0xA**. El resultado se obtiene en **EAX**. En este caso **0*0xA=0**

`0x080480e7:    xchg   ebx,eax ;` vuelve a cambiar los valores, lo que estaba en **EAX,** el valor <TOTAL> que vale 0 pasa a **EBX**, y lo que había en **EBX** (0x1) pasa a **EAX.**

`0x080480e8:    add    ebx,eax ;` suma al valor **<TOTAL>** el valor de **EAX** que era **0x1,** por tanto termina este bucle valiendo **1**.

Si hacemos otra vuelta veremos mas claro como se va formando un valor **<TOTAL>** para nuestra **key:**

`0x080480e2:    sub    al,0x30 ;` en **AL** queda nuestro segundo valor **0x2**.
`0x080480e4:    xchg   ebx,eax ;` en EAX tenemos el valor **<TOTAL>**, vale 0x1 y en **EBX** se pone 0x2.

```
0x080480e5:     mul     ecx   ;
0x080480e7:     xchg    ebx,eax ;
0x080480e8:     add     ebx,eax ;
```

0x080480e5: mul ecx ; (<TOTAL> * 0xA) se guarda en **EAX**
0x080480e7: xchg ebx,eax ; <TOTAL> que vale **0xA** se pasa a **EBX**
0x080480e8: add ebx,eax ; se le suma a **<TOTAL>** nuestro segundo valor **0x2.**

Y así consecutivamente pasan todos los valores dando un **<TOTAL> final**, que en este momento no importa, pero que ya sabemos como se forma.

Si seguimos después de la **Función_1**, llegamos a un sitio interesante:

```
0x8048158: test   eax,eax
0x804815a: js     0x8048196 ; por si es un valor negativo¿?
0x804815c: pop    esi
0x804815d: xor    ecx,ecx
0x804815f: mov    bl,BYTE PTR [esi+0x3] ;  toma el signo = de "key=12345."
0x8048162: sub    bl,0x31
0x8048165: je     0x8048182--------->continuamos en 8048182
0x8048167: dec    bl   ; si era key2 aqui quedara 0
0x8048169: je     0x8048170--------->continuamos en 0x8048170
0x804816b: jmp    0x8048197
```

Nos centramos en :

0x804815f: mov bl,BYTE PTR [esi+0x3] ; toma el signo = de "key=123456789", pero después le resta el valor **0x31 (1)** y si es igual (se activa el flag Z) nos lleva a 0x8048182:

```
0x08048182: or     DWORD PTR ds:0x804838c,0x1
0x0804818c: add    ecx,0x8048390
0x08048192: mov    DWORD PTR [ecx],eax
0x08048194: jmp    0x8048197
```

Por tanto parece que el valor después de key es 1, lo que indica que el programa espera una **key1**, además la zona **0x8048182** parece muy interesante, tomamos nota de que se hace **OR lógico** entre el valor que hay en la dirección **0x804838C** y el valor **1,** que normalmente si el valor original era 0 sería como ponerlo a **1**.

SI volvemos al código inicial se intuye que también debe haber una **key2**; si fuera así, el valor que obtenemos en **0x804815f** de una posible key2 sería el **2**, que en ascii es **0x32** y estaría en **BL**:

0x8048162: sub bl,0x31 ; 0x32 menos 0x31 es igual a 0x1
0x8048165: je 0x8048182 ; no hay salto
0x8048167: dec bl ; si quitamos una unidad a 0x1 queda a 0
0x8048169: je 0x8048170 ; se produce el salto y continuamos en 0x8048170

```
0x08048170: or     DWORD PTR ds:0x804838c,0x2
0x0804817a: add    ecx,0x4
0x08048180: jmp    0x804818c
```

Como veis hace un **OR lógico** entre el valor que hay en **0x804838c** y **0x2**, que en binario es 10; y si ya teníamos el valor **0x1** de la **key1**, 01 en binario; nos quedará como valor binario 11, **0x3** en hexadecimal, debido a la **key2.**

Por tanto, vamos a reiniciar el proceso pero antes debemos quitar la key anterior de la variable de shell, para ello se utiliza la orden **export** pero en este caso con la opción **-n**:

```
$ export -n key
```

Y ahora introducimos las dos key que espera el programa:

```
$ export key1=12345

$ export key2=67890
```

Ejecutamos el archivo **888.copia**, restablecemos los bytes del bucle infinito, y llegamos a la zona de la **Función_1**, donde esta vez anotamos el valor **<TOTAL_1>** del **key1** que es 0x3039 y el **<TOTAL_2>** de la **key2** que es 0x10932.

Si continuamos esperando la resolución de programa, nos encontramos que vuelve a preparar otra trampa para el debugger con otra llamada al sistema signal:

```
eax:00000030 ebx:00000008  ecx:08048070  edx:00000000

0x8048375:  int    0x80 ; otra trampa:
0x8048377:  ret
```

Tenemos de nuevo en EAX el valor **0x30**, que ya sabemos es **48:**

#define __NR_signal 48

Y se define como:

NOMBRE
signal - manejo de señales en ANSI C

SINOPSIS
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);

Por tanto en **EBX** tenemos la señal que va a manejar , es este caso la **8**, que con la ayuda de **"man 7 signal"** sabemos que corresponde a:

SIGFPE 8 Core Excepción de coma flotante

En **ECX** tenemos el manejador de la señal, que en este caso está en 0x08048070 y que como ya veremos es esencial pasar por allí para resolver el crackme.

Si seguimos con el debugger vemos que se va a producir una situación que dispara la señal **8** en **0x804819D**:

```
0x804819d: mov    eax,ds:0x804838c ; mueve el valor 0x3 a EAX.
0x80481a2: and    eax,0x3 ; se asegura que haya ese valor
0x80481a7: dec    eax      ; deja 0x2 en EAX
0x80481a8: je     0x80481b3 ; si es cero vamos a error.
0x80481aa: dec    eax      ; deja 0x1 en EAX
0x80481ab: je     0x80481b3 ; si es cero vamos a error.
0x80481ad: dec    eax      ; deja 0 en EAX
0x80481ae: jne    0x80481b3 ; si no es cero vamos a error.
0x80481b0: xchg   ecx,eax   ; mueve el valor 0 a ECX
0x80481b1: div    ecx ; salta la excepción por dividir por 0...eax/0
0x80481b3: ret
```

Como he ido explicando, si hemos pasados la **key1** y la **key2** tenemos el valor **0x3** en **0x804838C** y si os

fijáis en todo el desarrollo del código a la fuerza te lleva a la excepción; pues todos los saltos que nos salvarían a **0x80481b3** son callejones sin salida comprobados.

Por tanto si terminamos esta función tendremos el mismo problema que al principio,al **dividir por 0** se genera una excepción de coma flotante, esta señal 8 la manejará el debugger en lugar del programa y no podremos resolver el crackme pues no pasaremos por **0x8048070**, que es el manejador de la señal 8 si no hubiera depurador.

La solución será igual que al principio, vamos a colocar los opcodes del salto sobre si mismo, **EB FE**, en la dirección **0x8048070**, después nos desengancharemos del programa para que se produzca la excepción y se quede en un bucle infinito en **0x8048070**.

Primero vamos a ver los bytes que hay en **0x8048070:**

```
gdb> x/2b 8048070
0x8048070:  0x57  0xbf
gdb> set{short}0x8048070=0xfeeb
gdb> x/i 0x8048070
0x8048070:  jmp    0x8048070
```

Los bytes **0x57** y **0xbf** se sustituyen por **EB EF** con **set{short}0x8048070=0xfeeb** , lo comprobamos con **x/i** y ya estamos preparados para dejar correr al programa:

```
gdb> detach
gdb> attach 27270
```

Con la orden **detach** el programa sigue su camino. Sin salir de gdb, la orden **attach** acompañado del **PID** del proceso que queremos enganchar nos devuelve el control sobre **888.copia:**

```
     eax:00000008 ebx:00000008  ecx:00000000  edx:00000000     eflags:00000246
     esi:00000000 edi:00000000  esp:FFA12A7C  ebp:00000000     eip:08048070
     cs:0023  ds:002B  es:002B  fs:0000  gs:0000  ss:002B    o d I t s Z a P c
[002B:FFA12A7C]---------------------------------------------------------[stack]
FFA12AAC : 00 00 00 00  70 80 04 08 - 00 00 00 00  00 00 00 00 ....p...........
FFA12A9C : 00 00 00 00  64 2D A1 FF - 08 00 00 00  00 00 00 00 ....d-..........
FFA12A8C : 2B 00 00 00  2B 00 00 00 - 00 00 00 00  00 00 00 00 +...+...........
FFA12A7C : 00 54 74 F7  08 00 00 00 - 00 00 00 00  00 00 00 00 .Tt.............
[002B:FFA12A7C]---------------------------------------------------------[ data]
FFA12A7C : 00 54 74 F7  08 00 00 00 - 00 00 00 00  00 00 00 00 .Tt.............
FFA12A8C : 2B 00 00 00  2B 00 00 00 - 00 00 00 00  00 00 00 00 +...+...........
[0023:08048070]---------------------------------------------------------[ code]
0x8048070:  jmp    0x8048070
0x8048072:  mov    cl,0x81
0x8048074:  add    al,0x8
0x8048076:  mov    ecx,0x2
0x804807b:  mov    al,0x90
0x804807d:  rep stos BYTE PTR es:[edi],al
------------------------------------------------------------------------------
0x08048070 in ?? ()
```

Ahora dejamos todo como estaba y continuamos con la parte final del programa:

```
gdb> set {short}0x8048070=0xbf57
gdb> x/2i $pc
0x8048070:  push   edi
0x8048071:  mov    edi,0x80481b1
```

El código desde **0x8048070** es fundamental, por eso lo vamos a ver poco a poco, y llamaremos a

esta zona **Función_2**:

`0x8048070: push   edi`
`0x8048071: mov    edi,0x80481b1 ;` mueve la dirección **0x80481b1** a **EDI**, en esa dirección esta la instrucción que provocaba la excepción al dividir por 0
`0x8048076: mov    ecx,0x2 ;` sera el contador
`0x804807b: mov    al,0x90 ;` mueve a **AL** el byte **0x90** que como sabemos es el opcode de **NOP**, es un opcode que no tiene ninguna función y por eso es tan útil, en este caso sirve para eliminar una instrucción (nopear).
`0x804807d: rep stos BYTE PTR es:[edi],al ;` va a colocar los bytes **0x90** en la zona que indica **EDI** (**0x80481b1**) con **STOS**, con **REP** cada transferencia de byte hará decrecer **ECX** en una unidad hasta que llegue a 0, por tanto solo va a nopear 2bytes. De esta manera evita que se vuelva a producir la excepción en 0x80481b1
`0x804807f: pop    edi ;` la cima de la pila se pone en EDI
`0x8048080: mov    eax,ds:0x804838c ;` mueve el valor que hay en **0x804838c** a **EAX**
`0x8048085: and    eax,0x3 ;` solo si hay un 0x3 en **EAX,** al realizar un **AND lógico** frente a otro 0x3, no se producirán cambios en **EAX**. ¿?
`0x804808a: cmp    eax,0x3 ;` ahora comprueba que el valor en **EAX** es **0x3**, esta es la señal de que ha habido dos keys, **key1** y **key2**.
`0x804808f: jne    0x80480c3 ;` si la resta no activa el flag cero **Z**, es que no era 0x3 y por tanto nos lleva por mal camino.
`0x8048091: mov    eax,ds:0x8048390 ;`mueve a **EAX** el valor que ha generado con la **key1= 3039**, será nuestro **<TOTAL_1>**
`0x8048096: mov    ecx,DWORD PTR ds:0x8048394 ;`mueve el valor que ha generado con la **key2 = 10932**, será **<TOTAL_2>**
`0x804809c: cmp    ecx,0x1000 ;` compara **<TOTAL_2>** con **0x1000**
`0x80480a2: jl     0x80480c3 ;` no debe ser menor de **0x1000** para ser correcto.
`0x80480a4: cmp    eax,0x1000 ;`compara **<TOTAL_1>** con **0x1000**
`0x80480a9: jl     0x80480c3 ;` no debe ser menor de **0x1000** para ser correcto.
`0x80480ab: mul    ecx ;` multiplica **EAX <TOTAL_1>** con **ECX <TOTAL_2>** y guarda el resultado en **EAX.**
`0x80480ad: mov    ecx,0x10001 ;` mueve el valor **0x10001** a **ECX**
`0x80480b2: div    ecx ;` el dividendo es **EAX,** que contiene el resultado de **<TOTAL_1> * <TOTAL_2>**, el divisor es **ECX** (se indica en la operación) que contiene el valor **0x10001**; el resultado se guarda en **EAX** y el resto se guarda en **EDX**:

```
EAX= (<TOTAL_1> * <TOTAL_2>) |_ECX=0x10001
                              COCIENTE en EAX
          RESTO=EDX
```

`0x80480b4: mov    eax,edx ;` mueve el valor del resto a **EAX**
`0x80480b6: dec    eax ;` quita una unidad a **EAX**
`0x80480b7: jne    0x80480c3 ;` si no es **0**, no es correcto y vamos fuera. Por tanto la condición es que al dividir el resultado de (<TOTAL_1> * <TOTAL_2>) con 0x10001 debe dar un resto que sea **1.**
Como es lógico, en esta vuelta no daba el resto correcto, por lo que para comprobar si quedaba alguna otra sorpresa, evite este salto **"jne 0x80480ce"** con las dos formas posibles; tanto activando el flag **Z** con **cfz** como poniendo el valor 0 en **EAX**:

```
gdb> cfz         ; suficiente para que no se produzca el salto
gdb> p $eflags
$1 = [ PF ZF IF ]
```

```
gdb> set $eax=0 ; para asegurarme
gdb> p $eax
$2 = 0x0
gdb> ni   ; continuamos
```

De esta manera si el programa comprobaba después el valor de **EAX,** este sería **0.** Ahora podemos seguir hasta terminar la **Función_2:**

```
0x80480b9: push   0x4c53d075 ; sube este valor a la pila
0x80480be: jmp    0x8048103 : continuamos en 0x8048103
0x80480c3: ret
```

Si seguimos el programa realiza una última llamada al sistema un poco especial:

```
0xf7745400 <__kernel_sigreturn>:       pop    eax
0xf7745401 <__kernel_sigreturn+1>:      mov    eax,0x77
0xf7745406 <__kernel_sigreturn+6>:      int    0x80
```

Es curioso, ejecuta código en la pila, sube el valor **0x77** a **EAX** y ejecuta la **int80.**

```
gdb> python print int('0x77',16)
119
```

Vemos la syscall 119:

#define __NR_sigreturn 119

Si miramos **"man 2 sigreturn"**:

NOMBRE
sigreturn - regresa desde el manejador de señales y limpia el marco de pila

SINOPSIS
int sigreturn(unsigned long __unused);

DESCRIPCIÓN
Cuando el núcleo de Linux crea la estructura de pila para el manejador de señales, inserta una llamada a sigreturn en la estructura de pila aunque el manejador de señales llamará a sigreturn a su vuelta. Esta llamada a sigreturn limpia la pila aunque el proceso puede restaurar desde donde fue interrumpido por la señal.

VALOR DEVUELTO
sigreturn nunca regresa.

PRECAUCIÓN
La llamada sigreturn es usada por el núcleo para inplementar el manejador de señales. Nunca debe ser llamada directamente. Mejor aún, el uso específico del argumento __unused varía dependiendo de la arquitectura.

CONFORME A
sigreturn es específico para Linux y no debe ser usado en programas que deban ser portados.

Parece que quiere cuadrar la pila después de la última señal ¿? no se muy bien porque, pero si

continuamos el programa va directamente a la zona que provoco la ultima señal:

```
0x80481b1:  nop
0x80481b2:  nop
0x80481b3:  ret
```

Que en este caso, como hemos pasado por el manejador de la señal 7, **0x08048070,** ya se nopeo y no volverá a aparecer esa excepción.

Si seguimos llegamos a esta última trampa:

```
0x8048325:  int    0x3  ; la trampa final grrrrrrrrrrrr
```

Como creo que la cosa ya esta solucionada, en este caso voy a desenganchar el debugger del programa con la orden **"detach"** y vemos el resultado en el terminal donde ejecutamos **888.copia**:

```
$ ./888.copia
OK
```

De momento ya hemos descubierto todos los secretos del crackme, solo nos queda encontrar dos valores de la **key1** y la **key2** que cumplan con las condiciones del programa tanto al pasar la **Funcion_1** como tras la **Función_2.**

Solución:

Primero pensé en dejar el tutorial así y terminar el crackme mas adelante. Pero pensando un poco y con algo de suerte encontré la solución.

Como buen reverse vamos a a comenzar por el final, si nos fijamos en la división de la **Función_2**:

```
EAX= (<TOTAL_1> * <TOTAL_2>) |_ECX=0x10001
                              COCIENTE en EAX
           RESTO=EDX=1
```

Hay dos cosas seguras, el valor que buscamos es un "casi" múltiplo de **0x10001**, y digo casi pues al dividirlo por 0x10001 debe dar un resto de **1**.
Mi primera idea fue **0x10001** como **<TOTAL_1>** y **0x10002** como **<TOTAL_2>**; pero claro esto sería valido si ambos valores se sumaran; pero en este caso se multiplican. De todos modos hice la prueba multiplicando, para ello use **python** como calculadora:

```
>>> 10001*10002
100030002
>>> 100030002%10001
0
```

El signo **%** en python te calcula el resto de una división, en este caso al ser un múltiplo perfecto de 10001 el resto es **0** y no nos vale.
Aquí fue donde San ZEN (patrón de los crackers,jejeje) me echo una mano, hice pruebas variando los términos y encontré estos valores:

```
>>> 10002*10002
100040004
>>> 100040004%10001
1
```

Correcto el resto es **1**, me saltaron dudas pues el programa maneja valores hexadecimales, pero no hay diferencia si los valores se pasan a hexadecimal, siguen siendo correctos:

```
>>> hex(0x10002*0x10002)
'0x100040004'
>>> hex(0x100040004%0x10001)
'0x1'
```

Por tanto el valor **<TOTAL_1>** como **<TOTAL_2>** son iguales (0x10002), por lo que solo nos queda conseguir con la **Función_1** una serie de números que nos de como resultado **0x10002**.

Si nos fijamos en la **Función_1** es una serie de multiplicaciones con una constante **0xA**, por tanto vamos a dividir **0x10002** por **0xA** varias veces hasta que nos de el valor mas bajo posible, que lógicamente será nuestro primer número:

gdb> python print hex(0x10002//0xa) ; signo de división //
0x1999
Hay que tener en cuenta si es una división exacta, para ello vemos el resto con **%:**
gdb> python print hex(0x10002%0xa)
0x8
Ahora todo cuadra, si miráis la **Función_1** vemos que después de multiplicar por **0xA** se le sumaba el valor exacto que habíamos puesto en la key, ese valor es el resto de cada una de estas divisiones, por tanto el valor 8 hay que añadirlo como último número.

gdb> python print hex(0x1999//0xa)
0x28f
gdb> python print hex(0x1999%0xa)
0x3---->Hay que añadirlo

gdb> python print hex(0x28f//0xa)
0x41
gdb> python print hex(0x28f%0xa)
0x5---->Hay que añadirlo

gdb> python print hex(0x41//0xa)
0x6----> Este es el primer numero
gdb> python print hex(0x41%0xa)
0x5---->hay que añadirlo, será el segundo número

Por tanto, para comprobarlo vamos a seguir los valores según la **Funcion_1**, teniendo en cuenta que tenemos el valor **<TOTAL_1>** acumulado en cada bucle:

6----> 0*0xa=0 ; 0+0x6=**0x6** <TOTAL_1>=0x6
5----> 0x6*0xa=0x3c como la división no es exacta se le debe sumar el resto 0x41%0xa=05; 0x3c+5=**0x41** <TOTAL_1>=0x41
5----> 0x41*0xa=0x28a como la división no es exacta se le debe sumar el resto 0x28f%0xa=0x5 ; 0x28a+5=**0x28f** <TOTAL_1>=0x28f
3----> 0x28f*0xa=0x1996 como la división no es exacta se le debe sumar el resto 0x1999%0xa=0x3 ; 0x1996+3=**0x1999** <TOTAL_1>=0x1999
8----> 0x1999*0xa=0xfffa como la división no es exacta se le debe sumar el resto 0x10002%0xa=0x8 ; 0xfffa+0x8=**0x10002** **<TOTAL_1>=0x10002**

Esto nos da que tanto la **key1** como la **key2** su valor es **65538**, vamos a comprobarlo:

```
juanjo@tukan{~/Descargas/crk/crackmes_linux/888}$ export key1=65538
juanjo@tukan{~/Descargas/crk/crackmes_linux/888}$ export key2=65538
juanjo@tukan{~/Descargas/crk/crackmes_linux/888}$ ./888
OK
```

!!!!Conseguido¡¡¡

Conclusión:

Con la solución de este crackme hemos dado un gran repaso al formato **ELF**, que sera el que nos encontremos en GNU/Linux, hemos continuado profundizando en **GDB** y he intentado explicar muchas instrucciones de assambler. Espero que se haya entendido, pues el lenguaje ensamblador es básico para el cracking.

Agradecimiento:

Sin duda al gran Maestro, Ricardo Narvaja y a todos los CracksLatinoS, una gran lista para aprender.

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

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