



Introducción al Cracking en Linux 06 – GDB(II).

Programa:	Linux Crackme 3 by g30rg3_x (Retos de Yashira.org) Descarga: http://www.yashira.org/ReTos/All/linux_crackme3_251.gz
Descripción:	Profundizaremos en GDB y Bash para resolverlo.
Dificultad:	Bajísima.
Herramientas:	Herramientas del sistema y GDB.
Objetivos:	Solucionar el crackme, encontrando el serial correcto de diferentes maneras.

Cracker: [Juan Jose]	Fecha: 09/11/2011
----------------------	-------------------

Introducción:

Continuamos con la solución del crackme usado en la parte N°5 de esta “Introducción al Cracking en Linux”. Para ello se podía utilizar IDA, pues realmente es un crackme muy sencillo, pero me ha interesado más hacerlo con GDB, que al ser una herramienta nativa de GNU/Linux, nos la vamos a encontrar siempre en los repositorios de cualquier distribución Linux y se puede utilizar incluso en aquellos sistemas donde no tengamos interfaz gráfica (servidores, de forma remota, etc...).

Para instalarlo en Debian (no suele estar instalado por defecto) solo tenemos que utilizar el gestor de paquetes de esta distribución **apt-get** y en una consola como **root** (o en otras distribuciones con “**sudo**”) ejecutar esta orden:

```
# apt-get install gdb
```

También se puede hacer de modo gráfico con el programa **Synaptic**, que lo podemos encontrar en el menú **Sistema > Administracion** de un escritorio **Gnome**; al ejecutarlo el sistema nos va a pedir la contraseña de root (como para cualquier tarea administrativa) y ya dentro de él podemos buscarlo e instalarlo.

Por cierto su manejo básico ya se explico en la parte 2 de esta serie de titulares, y por tanto, para el que nunca lo haya utilizado, convendría echarle una miradita:

Introduccion al Cracking en Linux 02 - GDB

http://www.4shared.com/document/AjNIQnGw/Introduccion_al_Cracking_en_Li.html

En realidad, en este tutorial, lo que quiero mostrar es algunas de las muchísimas posibilidades que nos da un sistema GNU/Linux y pongo **GNU** esta vez (tendría que hacerlo siempre, pero se me olvida :-); porque son todas las **herramientas GNU**; como **bash**, **gdb**, **gcc**, **binutils** (donde esta **objdump**) y muchas mas.... las que le han dado a este sistema su potencia. Para nosotros, curiosos de la informática, el terminal de GNU/Linux no debería ser sinónimo de martirio, sino todo un mundo de posibilidades, que en este caso lo enfocaremos al estudio de binarios, pero igualmente se pueden enfocar en el estudio de redes, programación, etc.....

Al Atake:

Primero vamos a ejecutar el programa **linux_crackme3**, para ver de que se trata, para ello vamos con un terminal a la carpeta donde lo tengamos, nos aseguramos de que tenga permiso de ejecución y lo ejecutamos:

```
juanjo@tukan[~/Descargas/crk/Crackme_Yashira/crackme3]$ ./linux_crackme3
Serial: 12345678790
Serial Invalido!
```

Es bastante clásico, nos pide un Serial, lo comprueba y como hoy no es mi día... me comenta que es un “**Serial Invalido!**”. Ahora vamos a estudiar este crackme y para ello vamos a dividir el proceso en tres pasos, **información**, **desensamblado** y **depuración**; de modo que nos sirva de referencia para futuros retos.

1.Información.

1.a)La orden **file**: determina el tipo de fichero y nos da una información básica del archivo.

```
juanjo@tukan[~/Descargas/crk/Crackme_Yashira/crackme3]$ file linux_crackme3
linux_crackme3: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.8, stripped
```

Por tanto es un archivo **ELF**(lo normal en linux) y es de 32 bits, además lo mas interesante es que esta “**stripped**”, por tanto han utilizado el programa **strip** para eliminar los símbolos del fichero objeto. Este programa se suele usar para quitar los símbolos de depuración, de esta forma:

```
strip --strip-debug fichero
```

Para quitar todos los símbolos, la orden es así:

```
strip --strip-all fichero
```

Con esta primera orden, vamos a iniciar la recopilación de información en un archivo de texto que mantendremos en la carpeta del programa:

```
$ file linux_crackme3 > informacion.txt
```

El símbolo > nos sirve para redireccionar la salida de un comando; normalmente la salida normal **stdout** es la pantalla, pero mediante el símbolo > vamos a volcar la salida hacia un archivo, que si no existe, la shell lo va a crear automáticamente y si existe con esta orden lo vamos a sobrescribir. En nuestro caso lo que queremos es ir añadiendo la información a un archivo (**informacion.txt**) sin destruir su contenido, por lo que en las demás ordenes utilizaremos el doble signo >>

1.b) Strings; muestra las cadenas de caracteres imprimibles que haya en el fichero :

```
$ strings -a -t x linux_crackme3  
114 /lib/ld-linux.so.2  
221 __gmon_start__  
230 libc.so.6  
23a _IO_stdin_used  
249 stdin  
24f printf  
256 fgets  
25c strlen  
263 __libc_start_main  
275 GLIBC_2.0  
388 PTRh  
395 QVh4  
44e Seri  
455 al V
```

La salida completa la vamos a poner en nuestro archivo, por si después nos hace falta:

```
$ strings -a -t x linux_crackme3 >> informacion.txt
```

Strings con la opción **-a** nos va a dar todas las referencias, no solo las que nos muestra el archivo al ejecutarlo, si no también el nombre de las funciones, librerías e incluso con que gcc fue creado. Con la opción **-t x** nos dará en que posición del archivo se encuentra cada string, siendo las opciones **-t o** para darnos la posición en octal, **-t d** para decimal y **-t x** para hexadecimal. Este último es el que he visto mas interesante por si queremos acceder a esa cadena en un editor hexadecimal, que por defecto los suele expresar también en hexadecimal (últimamente estoy utilizando [okteta](#) como editor hexadecimal, muy bueno y esta en los repositorios).

Todas las opciones y mas cosas de cada comando lo podéis ver en el terminal con la orden "**man comando**" y también "**info comando**"

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

```
$ readelf -a linux_crackme3 >> informacion.txt
```

De esta manera, con la opción **-a** no da toda la información; si quisiéramos solo una parte, se puede usar otras opciones:

- h** muestra la información de encabezado del archivo ELF
- S** muestra la información de los encabezados de sección
- s** muestra la tabla de símbolos
- r** muestra la información de relocación
- x** vuelca el contenido de la sección especificada

La última opción `-x` tiene posibilidades, podemos ver lo que contiene algunas secciones, para usarla primero debemos saber el **Nr.** de la sección, para ello utilizamos la opción `-S`:

```
$ readelf -S linux_crackme3
There are 27 section headers, starting at offset 0xa7c:

Section Headers:
[Nr] Name           Type           Addr           Off           Size          ES   Flg  Lk  Inf  Al
[ 0]                 NULL          00000000      0000000      0000000      00   A   0   0   0
[ 1] .interp          PROGBITS      08048114      000114       000013       00   A   0   0   1
[ 2] .note.ABI-tag    NOTE          08048128      000128       000020       00   A   0   0   4
[ 3] .hash            HASH          08048148      000148       000034       04   A   5   0   4
[ 4] .gnu.hash        GNU_HASH      0804817c      00017c       000024       04   A   5   0   4
[ 5] .dynsym          DYNAMIC       080481a0      0001a0       000080       10   A   6   1   4
[ 6] .dynstr          STRTAB        08048220      000220       00005f       00   A   0   0   1
[ 7] .gnu.version     VERSYM        08048280      000280       000010       02   A   5   0   2
[ 8] .gnu.version_1  VERNEED      08048290      000290       000020       00   A   6   1   4
[ 9] .rel.dyn         REL           080482b0      0002b0       000010       08   A   5   0   4
[10] .rel.plt         REL           080482c0      0002c0       000028       08   A   5  12   4
[11] .init            PROGBITS      080482e8      0002e8       000030       00  AX   0   0   4
[12] .plt             PROGBITS      08048318      000318       000060       04  AX   0   0   4
[13] .text           PROGBITS      08048380      000380       0002ec       00  AX   0   0  16
[14] .fini           PROGBITS      0804866c      00066c       00001c       00  AX   0   0   4
[15] .rodata         PROGBITS      080486a0      0006a0       000121       00   A   0   0  32
[16] .eh_frame       PROGBITS      080487c4      0007c4       000004       00   A   0   0   4
[17] .ctors          PROGBITS      080497c8      0007c8       000008       00  WA   0   0   4
[18] .dtors          PROGBITS      080497d0      0007d0       000008       00  WA   0   0   4
[19] .jcr            PROGBITS      080497d8      0007d8       000004       00  WA   0   0   4
[20] .dynamic        DYNAMIC       080497dc      0007dc       0000d0       08  WA   6   0   4
[21] .got            PROGBITS      080498ac      0008ac       000004       04  WA   0   0   4
[22] .got.plt        PROGBITS      080498b0      0008b0       000020       04  WA   0   0   4
[23] .data           PROGBITS      080498d0      0008d0       000008       00  WA   0   0   4
[24] .bss            NOBITS        080498d8      0008d8       00000c       00  WA   0   0   4
[25] .comment        PROGBITS      00000000      0008d8       0000d9       00   0   0   1
[26] .shstrtab       STRTAB        00000000      0009b1       0000cb       00   0   0   1

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
```

Interesante la columna **Flg**, que como vemos abajo (*Key to Flags*) nos indica si la sección es ejecutable (**X**), permite escritura (**W**) o contiene cadena de caracteres (**S**) entre otras opciones.

De esta manera podemos ver el dumpado de las diferentes secciones, como ejemplo, vamos a ver la sección **.rodata**, que como vemos por la orden anterior tiene el número **15**, de modo que la orden sería:

```
$ readelf -x 15 linux_crackme3

Hex dump of section '.rodata':
0x080486a0 03000000 01000200 00000000 00000000 .....
0x080486b0 00000000 00000000 00000000 00000000 .....
0x080486c0 47000000 64000000 6c000000 6a000000 G...d...l...j...
0x080486d0 65000000 69000000 2a000000 27000000 e...i...*...'....
0x080486e0 7b000000 60000000 64000000 2b000000 {...`...d...+...
0x080486f0 7f000000 64000000 63000000 6d000000 ....d...c...m...
0x08048700 7f000000 7d000000 7d000000 60000000 ....}...}...`...
```

```

0x08048710 34000000 79000000 77000000 37000000 4...y...w...7...
0x08048720 7b000000 76000000 69000000 7a000000 {...v...i...z...
0x08048730 3c000000 6e000000 7b000000 3f000000 <...n...{...?...
0x08048740 50000000 4e000000 4c000000 46000000 P...N...L...F...
0x08048750 04000000 48000000 53000000 5e000000 ....H...S...^...
0x08048760 08000000 40000000 44000000 5f000000 ....@...D..._...
0x08048770 49000000 5f000000 4b000000 5c000000 I..._...K...\....
0x08048780 51000000 5f000000 46000000 56000000 Q..._...F...V...
0x08048790 14000000 5b000000 59000000 17000000 ....[...Y.....
0x080487a0 5b000000 4b000000 5f000000 5e000000 [...K..._...^...
0x080487b0 4f000000 02000000 53657269 616c3a20 0.....Serial:
0x080487c0 00

```

2. Desensamblado.

2.a) Objdump: Muestra la información contenida en un archivo objeto, de modo que desensambla el segmento de código del archivo binario y lo analiza.

Para ver todas las opciones de este programa aconsejo mirar el "**man objdump**", pero en mi caso las opciones que mas he utilizado son:

- f muestra información básica y nos da el Entry Point del ejecutable
- p muestra el formato del archivo y las librerías que utiliza.
- h muestra las secciones del ejecutable
- d desensambla el código del segmento de texto
- D desensambla todo el programa incluyendo los datos
- G muestra la información de depuración de programas
- l muestra las direcciones en las líneas
- r muestra las entradas de relocalización
- R muestra las entradas de relocalización dinámica
- t muestra la tabla de símbolos
- M podemos indicar las opciones del desensamblado, por defecto el desensamblado es

att. Las diferentes opciones que nos ofrecen objdump con **-M** son:

- x86-64** Disassemble in 64bit mode
- i386** Disassemble in 32bit mode
- i8086** Disassemble in 16bit mode
- att** Display instruction in AT&T syntax
- intel** Display instruction in Intel syntax
- att-mnemonic** Display instruction in AT&T mnemonic
- intel-mnemonic** Display instruction in Intel mnemonic
- addr64** Assume 64bit address size
- addr32** Assume 32bit address size
- addr16** Assume 16bit address size
- data32** Assume 32bit data size
- data16** Assume 16bit data size
- suffix** Always display instruction suffix in AT&T syntax

En este caso, vamos a salvar todo el desensamblado a un archivo llamado **objdump.txt**, para poder analizar el código posteriormente y para tenerlo a mano mientras utilicemos el debugger:

```
$ objdump -f linux_crackme3
linux_crackme3:      file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x08048380
```

Con **-f** nos da una información muy básica, pero si queremos poner un breakpoint en el **EntryPoint** con **gdb**, ya sabemos que dirección poner, tened en cuenta que al no tener símbolos de depuración no nos sirve expresiones como "b main" o "b init" o "start", sino que debemos poner la dirección donde queremos colocar el breakpoint, en este caso "**b *0x08048380**" (ya sabéis siempre sin comillas.....).

Lo guardamos y creamos el archivo:

```
$ objdump -f linux_crackme3 > objdump.txt
```

Y continuamos con otras ordenes:

```
$ objdump -p linux_crackme3 >> objdump.txt
$ objdump -h linux_crackme3 >> objdump.txt
```

Por último vamos a generar el desensamblado:

```
$ objdump -d -l -M intel linux_crackme3 >> odjdump.txt
```

Con la opción **-d** nos dará el desensamblado de las secciones ejecutables, con **-l** nos dará la dirección de de cada instrucción, muy importante si queremos poner algún breakpoint y por último lo queremos en sintaxis **intel -M intel**, que es la que estamos acostumbrados, y la verdad, ya soy muy viejo para cambiar.

Ahora ya podemos estudiar el código, pero como mi idea es combinarlo con **gdb**, lo veremos al depurarlo.

2.b) Otros desensambladores. No los he utilizado y algunos están en fase alfa, pero gracias a "es.wikibooks.org" he podido ver algunas referencias:

Bastard Disassembler

Bastard es un potente y programable desensamblador para Linux y FreeBSD.

<http://bastard.sourceforge.net/>

ciadis

El nombre oficial de ciasdis es computer_intelligence_assembler_disassembler. Esta herramienta basada en Forth permite construir conocimiento sobre un cuerpo de código de manera interactiva e incremental. Es único en que todo el código desensamblado puede ser re-ensamblado exactamente al mismo código. Soporta 8080, 6809, 8086, 80386, Pentium I y DEC Alpha. Facilidades de scripting ayudan en al análisis de cabeceras de fichero Elf y MSDOS y hacen esta herramienta extensible. ciasdis para Pentium I está disponible como imagen binaria, las otras están en código fuente, cargables sobre lina Forth, disponible del mismo sitio.

<http://home.hccnet.nl/a.w.m.van.der.horst/ciadis.html>

lida linux interactive disassembler

un desensamblador interactivo con algunas funciones especiales como un criptoanalizador. Muestra referencias a cadenas, hace análisis de flujo de código, y no depende de objdump. Usa la librería de desensamblage Bastard para decodificar instrucciones individuales.

<http://lida.sourceforge.net>

ldasm

LDasm (Linux Disassembler) es un interfaz gráfico basado en Perl/Tk para objdump/binutils que intenta imitar el aspecto de W32Dasm. Busca referencias cruzadas (por ejemplo cadenas), convierte el código de GAS a un estilo parecido a MASM, traza programas y mucho más. Viene con PTrace, un logger para flujo de procesos.

<http://www.feedface.com/projects/ldasm.html>

3. Depuración.

En esta fase vamos a ejecutar el programa para poder verlo en acción, por tanto si vamos a estudiar un ejecutable sospechoso, debemos tener cuidado, hacedlo en un sistema virtual o en un entorno chroot para evitar problemas.

3.a) Ltrace; este programa permite interceptar y mostrar las llamadas a librerías dinámicas, por tanto nos muestra las API que utiliza el programa y puede ser una gran ayuda para indicarnos por donde podemos comenzar a atacar.

Sus posibilidades son muchas, ya sabéis "**man ltrace**" para verlas todas o "**ltrace --help**" para una ayuda rápida; yo normalmente lo ejecuto con esta orden:

```
$ ltrace -i -o ltrace.txt ./linux_crackme3
Serial: 1234567890
Serial Invalido!
```

Con la opción **-i** queremos que nos diga la dirección de donde se ejecuta la función, ideal para después con gdb poner breakpoint, la opción **-o** es para indicarle el archivo de salida y por último hay que ejecutar el programa **./linux_crackme3**.

Una vez ejecutado, podemos ver las funciones que se han ejecutado y desde donde, en el archivo **ltrace.txt**:

```
[0x80483a1] __libc_start_main(0x8048434, 1, 0xffe91c84, 0x80485e0,
0x80485d0 <unfinished ...>
[0x8048523] printf("Serial: ") = 8
[0x8048542] fgets("1234567890\n", 65, 0xf7754420) = 0xffe91a2b
[0x8048550] strlen("1234567890\n") = 11
[0x80485b7] printf("Serial Invalido!\n") = 17
[0xffffffff] +++ exited (status 0) +++
```

Llama mucho la atención la llamada a **strlen**, que también podemos ver con "**man strlen**" y nos

dice que "*strlen - calcula la longitud de una cadena de caracteres*", por tanto es fácil pensar que nuestro serial falso no cumple la primera condición para continuar y sera allí en **0x8048550** donde empezaremos con el debugger.

3.b) Strace; permite interceptar y grabar las llamadas al sistema.

En este caso no nos va a ser muy útil, pero es bueno saber como utilizarlo y además muy didáctico.

La orden es muy parecida a la anterior:

```
$ strace -i -o strace.txt ./linux_crackme3
[ Process PID=29913 runs in 32 bit mode. ]
Serial: 1234567890
Serial Invalido!
```

La opción **-i** para que nos diga la dirección desde donde se ejecuta, **-o** para salvar la salida en un archivo "**strace.txt**" y ejecutamos el crackme **./linux_crackme3**. Lo primero que nos comenta es el identificador del programa (PID) y que como el sistema es de 64 bits, el programa se ejecutará en el modo de 32 bits. Esto último sera el motivo por lo que en gdb seguiremos trabajando en formato de 32 bits, con `eax`, `edx`, etc....mientras que como veremos al final en un programa de 64 bits, los nombres de los registros han cambiado.

Si miramos el archivo **strace.txt** vemos las llamadas al sistema que se han producido en la ejecución del programa:

```
[ 7f867b58fb37] execve("./linux_crackme3", ["/linux_crackme3"], [/* 34
vars */]) = 0
[ f77906bd] brk(0) = 0x8c6d000
[ f77918c1] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such
file or directory)
[ f7791a03] mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xffffffff7778000
[ f77918c1] access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such
file or directory)
[ f7791784] open("/etc/ld.so.cache", O_RDONLY) = 3
[ f779174e] fstat64(3, {st_mode=S_IFREG|0644, st_size=99628, ...}) =
0
[ f7791a03] mmap2(NULL, 99628, PROT_READ, MAP_PRIVATE, 3, 0) =
0xffffffff775f000
[ f77917bd] close(3) = 0
[ f77918c1] access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such
file or directory)
[ f7791784] open("/lib32/libc.so.6", O_RDONLY) = 3
[ f7791804] read(3,
"\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\320m\1\0004\0\0\0"...
, 512) = 512
[ f779174e] fstat64(3, {st_mode=S_IFREG|0755, st_size=1327556, ...})
= 0
[ f7791a03] mmap2(NULL, 1337704, PROT_READ|PROT_EXEC, MAP_PRIVATE|
MAP_DENYWRITE, 3, 0) = 0xffffffff7618000
[ f7791a84] mprotect(0xf7758000, 4096, PROT_NONE) = 0
[ f7791a03] mmap2(0xf7759000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x140) = 0xffffffff7759000
[ f7791a03] mmap2(0xf775c000, 10600, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xffffffff775c000
```



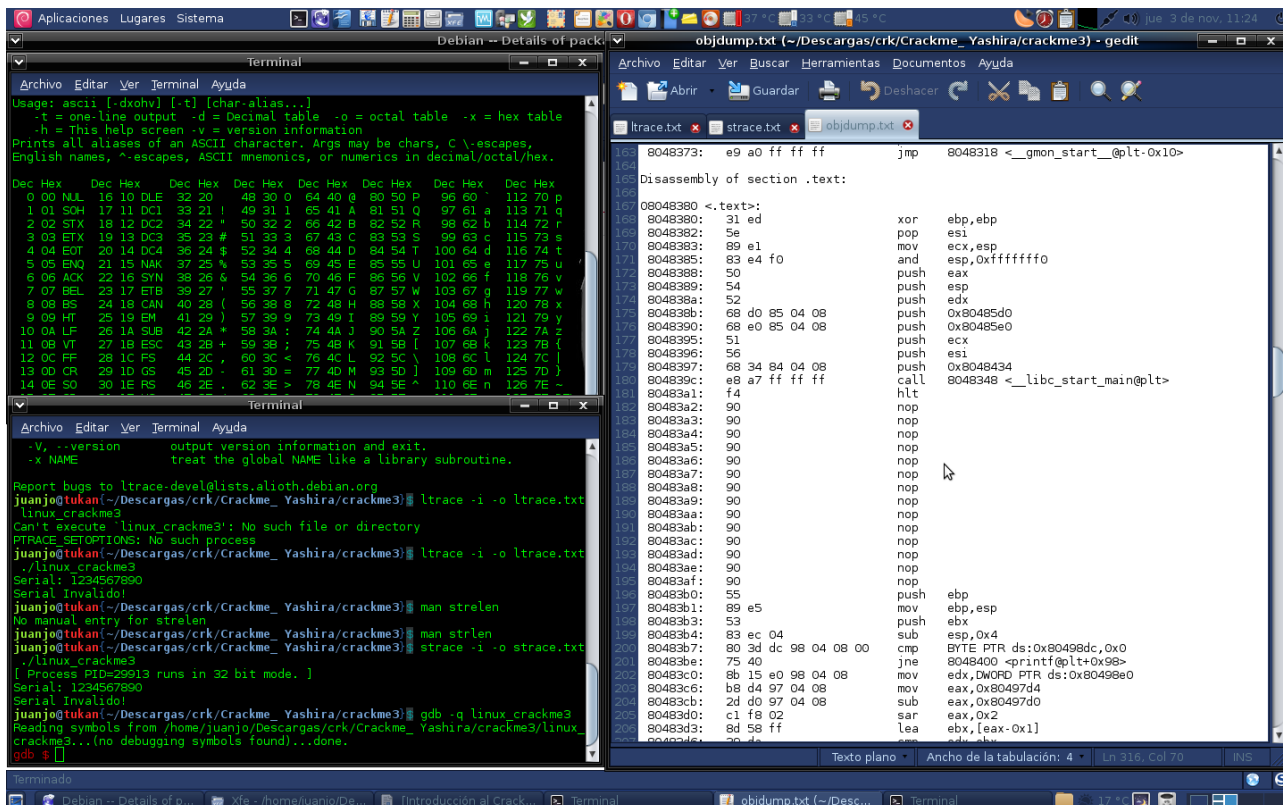
```

[ f77917bd] close(3) = 0
[ f7791a03] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7617000
[ f777d2df] set_thread_area(0xffc1d94c) = 0
[ f7791a84] mprotect(0xf7759000, 8192, PROT_READ) = 0
[ f7791a84] mprotect(0xf7797000, 4096, PROT_READ) = 0
[ f7791a41] munmap(0xf775f000, 99628) = 0
[ f777a430] fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
[ f777a430] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7777000
[ f777a430] fstat64(0, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
[ f777a430] mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xffffffff7776000
[ f777a430] write(1, "Serial: ", 8) = 8
[ f777a430] read(0, "1234567890\n", 1024) = 11
[ f777a430] write(1, "Serial Invalido!\n", 17) = 17
[ f777a430] exit_group(0) = ?

```

3.c) GDB; es el depurador estándar para el sistema operativo GNU.

Aunque hay varias interfaz gráficas para gdb, como ddd, kdbg, insight, nosotros vamos a hacerlo con el gdb ejecutándose en una consola, para trabajar yo suelo preparar un poco el escritorio, de forma que tengo dos terminales de Gnome, uno ejecutando gdb y otro para ver manuales o como se ve en la imagen, un programa llamado **ascii**, (bien simple, muestra una tabla con los valores ascii) y por otro lado **gedit** con el desensamblado del ejecutable(como tiene pestañas podemos tener muchos mas archivos de texto cargados, incluso un notas.txt para ir apuntando nuestras cositas.....



Ahora ya estamos preparados, vamos a ejecutar gdb:

```
$ gdb -q linux_crackme3
Reading symbols from /home/juanjo/Descargas/crk/Crackme_
Yashira/crackme3/linux_crackme3...(no debugging symbols found)...done.
gdb $
```

La opción **-q** es solo para que no nos de toda la información de gdb (version, copyright, etc...) al inicio. De esta manera, quedara un prompt tipo **gdb \$** esperando ordenes.

Ahora es el momento de configurar un poco gdb, para dejarlo de forma que nos sea mas cómodo trabajar con él. Mis opciones básicas son:

gdb \$ set disassembly-flavor intel

Con esta orden, el desensamblado que nos muestre gdb será en formato **Intel**, como esto va a ser siempre se puede colocar en un archivo en nuestra carpeta de usuario (home/usuario/) con el nombre **.gdbinit** (importante el punto al principio, pues es un archivo oculto) y no tendremos que colocarlo cada vez. Acompañando a este tutorial, os paso mi gdbinit, con las opciones que mejor me han funcionado y algunos macros que pueden ayudar, obtenidas de nuestro amigo Google.

gdb\$ display/5i \$pc

Esta orden va a hacer que nos muestre por donde vamos cada vez que ejecutemos una orden en **gdb**, en concreto las siguientes **5** instrucciones en ensamblador, pues **\$pc** es una variable interna que indica la siguiente instrucción que se va a ejecutar (igual que **\$eip**). **Display** es una orden que nos va a mostrar lo que queramos cada vez que se ejecute una orden, y como veremos es muy útil y tiene muchas posibilidades.

Se pueden poner todos los display que queramos, para manejarlos primero debemos saber que número tienen, si colocamos la orden **display** sola, nos listara todos:

gdb \$ display

1: x/5i \$pc

En este caso solo hay uno, si quisiéramos quitarlo, se usaría la orden:

gdb \$ undisplay 1

***Para todas las posibilidades, ya sabeis "**man gdb**" y para una info rápida, tenemos una especie de chuleta, muy cómoda para imprimirla y tenerla siempre a mano, en el mismo sistema:

/usr/share/doc/gdb/refcard.ps.gz

Para profundizar, tenemos la documentación que nos ofrecen en la pagina web del proyecto GDB:

<http://www.gnu.org/s/gdb/documentation/>

Ahora ya podemos comenzar a trabajar, para ello con la información que tenemos he pensado en ver donde nos lleva poner un breakpoint en la función **strlen** directamente:

gdb \$ b strlen

Breakpoint 1 at **0x8048358**

Si lo ponemos así, vemos que no coincide con la dirección que tomamos con **ltrace 0x8048550**, si miramos en el desensamblado vemos la explicación:

```

Disassembly of section .plt:

08048318 <__gmon_start__@plt-0x10>:
8048318: ff 35 b4 98 04 08      push  DWORD PTR ds:0x80498b4
804831e: ff 25 b8 98 04 08      jmp   DWORD PTR ds:0x80498b8
8048324: 00 00                  add   BYTE PTR [eax],al
...

08048328 <__gmon_start__@plt>:
8048328: ff 25 bc 98 04 08      jmp   DWORD PTR ds:0x80498bc
804832e: 68 00 00 00 00        push  0x0
8048333: e9 e0 ff ff ff        jmp   8048318 <__gmon_start__@plt-0x10>

08048338 <fgets@plt>:
8048338: ff 25 c0 98 04 08      jmp   DWORD PTR ds:0x80498c0
804833e: 68 08 00 00 00        push  0x8
8048343: e9 d0 ff ff ff        jmp   8048318 <__gmon_start__@plt-0x10>

08048348 <__libc_start_main@plt>:
8048348: ff 25 c4 98 04 08      jmp   DWORD PTR ds:0x80498c4
804834e: 68 10 00 00 00        push  0x10
8048353: e9 c0 ff ff ff        jmp   8048318 <__gmon_start__@plt-0x10>

08048358 <strlen@plt>:
8048358: ff 25 c8 98 04 08      jmp   DWORD PTR ds:0x80498c8
804835e: 68 18 00 00 00        push  0x18
8048363: e9 b0 ff ff ff        jmp   8048318 <__gmon_start__@plt-0x10>

08048368 <printf@plt>:
8048368: ff 25 cc 98 04 08      jmp   DWORD PTR ds:0x80498cc
804836e: 68 20 00 00 00        push  0x20
8048373: e9 a0 ff ff ff        jmp   8048318 <__gmon_start__@plt-0x10>

```

En realidad esta dirección forma parte de la sección **.plt** de los archivos **elf**, que como vemos es la **tabla de saltos** hacia las funciones de las librerías compartidas, que en linux son enlazadas dinámicamente en ejecución. Esto último lo he podido comprobar buscando la IAT, en windows estos saltos nos llevan a nuestra querida **IAT** (tabla de importaciones del programa), donde al compilar los programas, el linkador deja las direcciones y nombres de todas las funciones (APIs) que el programa va a utilizar; así que pensé en descubrir la IAT en linux; por tanto como el salto nos lleva a **0x80498c8** y esta dirección corresponde a la sección **.got.plt**, podemos verla con **readelf**:

```

& readelf -x 22 linux_crackme3

Hex dump of section '.got.plt':
 0x080498b0 dc970408 00000000 00000000 2e830408 .....
 0x080498c0 3e830408 4e830408 5e830408 6e830408 >...N...^...n...

```

Si nos fijamos en la dirección **80498c8**, corresponde a ese número marcado en rojo, que teniendo en cuenta que los valores en memoria tienen un formato **little indian**, se refiere la dirección **0804835e**, que como veis nos devuelve justo a la siguiente instrucción del salto anterior, parece que no va a ninguna función externa ¿?

```

8048358: ff 25 c8 98 04 08      jmp   DWORD PTR ds:0x80498c8
804835e: 68 18 00 00 00        push  0x18

```

Pero bueno, como tenemos el programa cargado en un debugger, vamos a ver si le encontramos explicación, primero vamos a reiniciar, para ello matamos el proceso con la orden "**kill**" y volvemos a cargar el proceso con la orden "**exec**":

```
gdb $ kill
gdb $ exec linux_crackme3
gdb $
```

Ahora si voy a poner un breakpoint en la llamada a **strlen**, de **0x08048550** (la que vimos en ltrace) aunque si miramos el desensamblado de esa zona:

```
804854b: e8 08 fe ff ff      call 8048358 <strlen@plt>
8048550: 83 f8 3f            cmp  eax,0x3f
```

Ltrace nos marca la dirección posterior de la llamada a **strlen**, pero en este caso vamos a seguir una llamada a esa función, por lo que debemos parar en **0x804854b**, por tanto ponemos un breakpoint en esa dirección y después comenzamos la ejecución del programa con la orden "**r**"(run):

```
gdb $ b *0x804854b
Breakpoint 1 at 0x804854b
gdb $ r
Serial: 123456

Breakpoint 1, 0x0804854b in ?? ()
1: x/5i $pc
0x804854b: call 0x8048358 <strlen@plt>
0x8048550: cmp  eax,0x3f
0x8048553: jne  0x8048592
0x8048555: mov  DWORD PTR [ebp-0x10],0x0
0x804855c: jmp  0x804858a
```

Como veis los display se mantienen si no salimos de **gdb**, por tanto la próxima instrucción será una llamada a **strlen@plt**, que es la zona que hemos visto antes. Para entrar en las llamadas debemos usar la orden "**si**":

```
gdb $ si
0x08048358 in strlen@plt ()
1: x/5i $pc
0x8048358 <strlen@plt>: jmp  DWORD PTR ds:0x80498c8
0x804835e <strlen@plt+6>: push 0x18
0x8048363 <strlen@plt+11>: jmp  0x8048318
0x8048368 <printf@plt>: jmp  DWORD PTR ds:0x80498cc
0x804836e <printf@plt+6>: push 0x20
```

Ahora continuaremos el salto con "**si**", no hace falta ponerlo otra vez, con **<Enter>** se vuelve a repetir y no aparece:

```
gdb $
0xf7ef10e0 in ?? () from /lib32/libc.so.6
1: x/5i $pc
0xf7ef10e0: push  esi
0xf7ef10e1: mov  eax,DWORD PTR [esp+0x8]
0xf7ef10e5: mov  ecx,eax
0xf7ef10e7: pxor xmm0,xmm0
```

```
0xf7ef10eb: mov     esi, eax
```

Como vemos la función entra en **0xf710e0** de **/lib32/libc-so.6** sin ningún paso intermedio ¿?

Para explicar esto vamos a ver que pasa cuando ejecutamos un programa **ELF**. Primero el kernel analiza el archivo y lo carga en la memoria virtual del usuario. Si la aplicación ha sido enlazada a librerías compartidas (esto lo vimos con la orden **file "dynamically linked (uses shared libs)"**), la aplicación debe contener el nombre del enlazador dinámico que se debe utilizar. En este momento el kernel transfiere el control al enlazador dinámico y no al programa. Este cargador dinámico es el responsable de cargar todas las librerías compartidas necesarias y de resolver todas las relocalaciones o movimientos de funciones que quedasen pendientes. A continuación, si que se transfiere el control al programa y se ejecutará. Es por tanto el enlazador dinámico, el que nos hace pasar del salto en la sección **.plt** a la función en si misma de forma transparente.

Para ver las funciones que debe recolocar el enlazador dinámico, podemos utilizar **readelf** con la opción **-r**:

```
$ readelf -r linux_crackme3

Relocation section '.rel.dyn' at offset 0x2b0 contains 2 entries:
  Offset      Info      Type           Sym.Value   Sym. Name
080498ac  00000106  R_386_GLOB_DAT 00000000   __gmon_start__
080498d8  00000705  R_386_COPY     080498d8   stdin

Relocation section '.rel.plt' at offset 0x2c0 contains 5 entries:
  Offset      Info      Type           Sym.Value   Sym. Name
080498bc  00000107  R_386_JUMP_SLOT 00000000   __gmon_start__
080498c0  00000207  R_386_JUMP_SLOT 00000000   fgets
080498c4  00000307  R_386_JUMP_SLOT 00000000   __libc_start_main
080498c8  00000407  R_386_JUMP_SLOT 00000000   strlen
080498cc  00000507  R_386_JUMP_SLOT 00000000   printf
```

Como podemos ver, la dirección que corresponde a **strlen** es justamente el valor **080498c8**, que era el destino del salto que vimos en la sección **.plt**, que como hemos visto nos llevaba a **.got.plt**; donde creíamos que estaba la IAT y en realidad solo estaba una dirección de retorno, ya que como hemos visto la IAT se crea dinámicamente.

Por tanto la clave es el enlazador dinámico, que como hemos visto debe indicarlo el ejecutable y eso lo podemos ver en la primera sección del ejecutable **.interp**:

```
$ readelf -x 1 linux_crackme3

Hex dump of section '.interp':
 0x08048114 2f6c6962 2f6c642d 6c696e75 782e736f /lib/ld-linux.so
 0x08048124 2e3200    .2.
```

Como vemos en este caso es **ld-linux.so.2** el enlazador dinámico, que sera el responsable de enlazar el programa con **libc-so.6** como hemos visto en **gdb**.

Por último, y para redondear este pequeño estudio de los ejecutables **elf**, debemos tener en cuenta otras secciones como **.hash**, **.dynsym**, y **.dynstr**, que son una pequeña tabla de símbolos usadas por el enlazador dinámico para poder recolocar las funciones que se ejecutan dinámicamente. Por lo visto, la tabla de **.hash** son utilizadas para poder encontrar rápidamente los símbolos necesarios en la sección **.dynsym** y las cadenas de caracteres (strings) en **.dynstr**:

```
$ readelf -x 6 linux_crackme3
```

```
Hex dump of section '.dynstr':
```

```
0x08048220 005f5f67 6d6f6e5f 73746172 745f5f00  .__gmon_start__
0x08048230 6c696263 2e736f2e 36005f49 4f5f7374  libc.so.6._IO_st
0x08048240 64696e5f 75736564 00737464 696e0070  din_used.stdin.p
0x08048250 72696e74 66006667 65747300 7374726c  rintf.fgets.strl
0x08048260 656e005f 5f6c6962 635f7374 6172745f  en.__libc_start_
0x08048270 6d61696e 00474c49 42435f32 2e3000    main.GLIBC_2.0.
```

Continuamos, como hemos visto el breakpoint en `strlen`, nos ha dado la posibilidad de estudiar los archivos **ELF**, pero no es lo que buscábamos, así que primero debemos borrarlo. Para ello, primero con la orden **b** vemos los breakpoint que tenemos colocados y el número que le corresponde y con **delete** lo borramos. Y ahora si que ponemos el breakpoint en **0x8048550** como vimos con **ltrace**:

```
gdb $ b
Breakpoint 2 at 0xf7ef10a0
gdb $ delete 2
gdb $ b *0x8048550
Breakpoint 6 at 0x804854b
```

Continuamos con la orden **"c"** y llegamos al punto que nos interesaba:

```
gdb $ c
Serial: 123432566

Breakpoint 1, 0x0804854b in ?? ()
1: x/5i $pc
0x8048550: cmp     eax,0x3f
0x8048553: jne     0x8048592
0x8048555: mov     DWORD PTR [ebp-0x10],0x0
0x804855c: jmp     0x804858a
```

Si añadimos otro display, **"display \$eax"**, siempre me mostrará el valor de **eax**, y como sabemos que **strlen** devuelve la longitud de la cadena en **eax**, podemos verlo en este caso:

```
gdb $
0x08048550 in ?? ()
2: $eax = 0xa
1: x/5i $pc
0x8048550: cmp     eax,0x3f
0x8048553: jne     0x8048592
0x8048555: mov     DWORD PTR [ebp-0x10],0x0
0x804855c: jmp     0x804858a
0x804855e: mov     eax,DWORD PTR [ebp-0x10]
```

Nuestro serial truco es muy corto, **0xa** y como vemos por la instrucción que se va ejecutar en **0x8048550** el serial correcto debe tener **0x3f** caracteres, por tanto debemos arreglar esa diferencia para que la comparación se cumpla y para eso se utiliza la orden **set**, asignándole un nuevo valor a la variable **\$eax**:

```
gdb $ set $eax=0x3f
gdb $ p $eax
$1 = 0x3f
```

Con **p** (print) podemos ver que el valor es correcto, por tanto el salto "**jne 0x8048592**" (salta si la comparación no es igual) no se produce y podemos continuar estudiando la parte que nos interesa en **0x8048555**.

Aprovecho el dato **0x3f**, para hacer otro inciso, como es lógico cuando se ve un dato en hexadecimal, buscamos la manera de pasarlo a diferentes formatos; en linux hay muchas posibilidades, desde la calculadora a programas específicos tanto en consola como gráficos; pero he visto que **gdb** tiene integrado **python** y me ha parecido interesante comentarlo. De momento yo lo he utilizado mas para cambios de base y de calculadora, pero no he mirado que grado de integración tiene, pues seria muy útil para hacer script dentro de gdb, aunque de momento no se como¿?

Para ver su uso en gdb utilizamos la orden **help**, que si la ejecutamos sola vemos que tiene muchas posibilidades:

```
gdb $ help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
```

Por tanto con poner "**help running**" nos dará todas las ordenes que se puede utilizar para movernos por el código con el debugger, y de esa misma forma podemos obtener una ayuda rápida para todos los comandos, en el caso de **python** sería:

```
gdb $ help python
Evaluate a Python command.

The command can be given as an argument, for instance:

    python print 23

If no argument is given, the following lines are read and used
as the Python commands. Type a line containing "end" to indicate
the end of the command.
```

Por tanto se puede hacer de dos maneras, con python y argumentos o ejecutar python y nos dará una linea de comandos, donde las ordenes terminarán con **end**. Por cierto, a diferencia de la consola de

Python, en gdb no nos muestra los resultados por defecto, sino que los muestra si tu lo indicas con la orden **print**. Vamos a ver unos cuantos ejemplos:

```
Cambiar de hexadecimal a decimal
gdb $ python
>print int('0x3f',16)
>end
63
Cambiar de decimal a hexadecimal
gdb $ python
>print hex(63)
>end
0x3f
Pasar de Ascii a decimal
gdb $ python
>print ord('a')
>end
97
Pasar de decimal a Ascii
gdb $ python
>print chr(97)
>end
a
Pasar de Ascii a hexadecimal
gdb $ python
>print(hex(ord('a')))
>end
0x61
Pasar de hexadecimal a Ascii
gdb $ python
>print(chr(int('0x61',16)))
>end
a
```

De modo que el serial debe tener **63** caracteres, vamos a ver si podemos averiguar cuales son, para ello vamos a continuar en gdb con la orden **"ni"** e intentaremos entender el código, para una mejor comprensión fui poniendo comentarios en el archivo **objdump.txt**, y siguiéndolos creo que se visualiza mejor:

```
8048550: 83 f8 3f      cmp    eax,0x3f      ; la longitud del serial debe ser 0x3f
8048553: 75 3d        jne   8048592 <printf@plt+0x22a> ; si no es igual Serial
Invalido
```

De momento este salto no se produce, continuamos en **0x8048555**:

```
8048555: c7 45 f0 00 00 00 00 mov   DWORD PTR [ebp-0x10],0x0; se pone a cero el
contador, es la expresión i=0 de un bucle tipo for
804855c: eb 2c        jmp   804858a <printf@plt+0x222> ; seguimos en 804858a
804858a: 83 7d f0 3d  cmp   DWORD PTR [ebp-0x10],0x3d ; sera el límite para el
contador i<0x3d
804858e: 7e ce        jle   804855e <printf@plt+0x1f6> ; si es menor seguimos con el
```


bucle en **804855e**

8048590: eb 07 jmp 8048599 <printf@plt+0x231> ; hemos terminado el bucle, seguimos en 8048599

Como vemos el contador que está en **[ebp-0x10]** va a ir aumentando hasta alcanzar el valor **0x3d**, en este caso no **0x3f**, eso es debido a que en el terminal al pedirnos el **Serial:**, y ponerlo, damos **<intro>** y sin darnos cuenta se añade otro caracter que no vemos, es un salto de carro **\n**

804855e: 8b 45 f0 mov eax,DWORD PTR [ebp-0x10] ; movemos el contador a **eax** ¿? en la primera vuelta vale cero

8048561: 0f b6 84 05 63 fe ff movzx eax,BYTE PTR [ebp+eax*1-0x19d] ; movemos a **eax** el primer valor de mi serial falso

Para comprobar esto último debemos utilizar la orden **"x"** en **gdb**:

```
gdb $ x/s ($ebp+$eax*1-0x19d)
0xfffffd23b: "123432566\n"
```

Ahora comprobamos porque la longitud del serial es **0x3f** mientras que el limite del bucle es **0x3d**, pues como había comentado, hay que añadirle el salto de carro **\n** como un carácter más.

Respecto a **x** es la orden para ver la memoria en **gdb**, todas sus opciones se pueden ver con **"help x"**, en concreto con **s** vamos a ver **string** o cadena de caracteres, en este caso el serial falso que hemos puesto. Si veis el desensamblado debemos indicar **[ebp+eax*1-0x19d]**, para que **gdb** lo entienda solo hay que anteponer a los nombres de los registros el signo **\$**, pues para **gdb** son variables, y sustituir los corchetes por paréntesis; **(\$ebp+\$eax*1-0x19d)**. Seguimos:

8048569: 0f be d0 movsx edx,al ; movemos el byte mas significativo del primer valor que estaba en **eax** a **edx**

804856c: 8b 45 f0 mov eax,DWORD PTR [ebp-0x10] ; movemos el contador a **eax** ¿? en la primera vuelta vale cero

804856f: 8b 84 85 a4 fe ff ff mov eax,DWORD PTR [ebp+eax*4-0x15c] ; movemos a **eax** el primer valor del serial correcto

8048576: 33 45 f0 xor eax,DWORD PTR [ebp-0x10] ; xoreamos el primer valor correcto con el valor del contador

Si miramos en **[ebp+eax*4-0x15c]** con **gdb**, veremos los valores que se utilizan para generar el serial correcto:

```
gdb x/62wx ($ebp+$eax*4-0x15c)
0xfffffd27c: 0x00000047 0x00000064 0x0000006c 0x0000006a
0xfffffd28c: 0x00000065 0x00000069 0x0000002a 0x00000027
0xfffffd29c: 0x0000007b 0x00000060 0x00000064 0x0000002b
0xfffffd2ac: 0x0000007f 0x00000064 0x00000063 0x0000006d
0xfffffd2bc: 0x0000007f 0x0000007d 0x0000007d 0x00000060
0xfffffd2cc: 0x00000034 0x00000079 0x00000077 0x00000037
0xfffffd2dc: 0x0000007b 0x00000076 0x00000069 0x0000007a
0xfffffd2ec: 0x0000003c 0x0000006e 0x0000007b 0x0000003f
```

```

0xffffd2fc: 0x00000050 0x0000004e 0x0000004c 0x00000046
0xffffd30c: 0x00000004 0x00000048 0x00000053 0x0000005e
0xffffd31c: 0x00000008 0x00000040 0x00000044 0x0000005f
0xffffd32c: 0x00000049 0x0000005f 0x0000004b 0x0000005c
0xffffd33c: 0x00000051 0x0000005f 0x00000046 0x00000056
0xffffd34c: 0x00000014 0x0000005b 0x00000059 0x00000017
0xffffd35c: 0x0000005b 0x0000004b 0x0000005f 0x0000005e
0xffffd36c: 0x0000004f 0x00000002

```

Como veis hemos buscado en memoria $x / 62$ word en formato hexadecimal **62wx**, curiosamente el formato hexadecimal por defecto son 4 bytes, por lo que hubiese dado el mismo resultado con la expresión $x/62x$. Tened en cuenta que debemos seguir lo que nos comenta gdb en su ayuda:

"Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes)"

****También podemos encontrar en internet la definición de un byte, dos byte como una palabra (word) y 4 bytes es doubleword ¿?****

Por tanto, copiamos todos estos valores en gedit y los guardamos como **base.txt**. Estos valores serán la base para diferentes soluciones que he planteado, en este caso sabemos que estos valores se le hace **xor** con el contador y va dando un resultado que el programa pone en **EAX**, mientras que el valor de nuestro serial esta en **EDX** y los compara:

8048579: **39 c2** **cmp edx,eax** ; compara el valor correspondiente a nuestro serial con el valor xoreado

804857b: **74 09** **je 8048586 <printf@plt+0x21e>** ; si es igual se continua el ciclo en **8048586**

En este punto ya sabemos varias cosas, el primer valor es **0x47**, en ascii es **G** y como no pusimos ese valor en nuestro serial, para continuar con el bucle debemos hacer que la comparación se cumpla, cambiando el valor de **\$edx** en este caso:

```

gdb $
0x08048579 in ?? ()
2: $eax = 0x47
1: x/5i $pc
0x8048579: cmp     edx,eax
0x804857b: je     0x8048586
0x804857d: mov    DWORD PTR [ebp-0x14],0x0
0x8048584: jmp    0x8048599
0x8048586: add    DWORD PTR [ebp-0x10],0x1
gdb $ p $edx
$1 = 0x31
gdb $ set $edx=0x47
gdb $ p $edx
$2 = 0x47
gdb $ b *0x8048579
Breakpoint 2 at 0x8048579
gdb $

```

Además,hemos colocado un breakpoint en la comparación (b *0x8048579) pues en ese punto vamos a tener siempre el valor correcto del serial en **eax**. Seguimos:

8048586: **83 45 f0 01** **add** **DWORD PTR [ebp-0x10],0x1** ; se incrementa el contador en una unidad **i++**
804858a: **83 7d f0 3d** **cmp** **DWORD PTR [ebp-0x10],0x3d** ; sera el límite para el contador **i<0x3d**
804858e: **7e ce** **jle** **804855e <printf@plt+0x1f6>** ; si es menor seguimos con el bucle en **804855e**
8048590: **eb 07** **jmp** **8048599 <printf@plt+0x231>** ; hemos terminado el bucle, seguimos en **8048599**

Por terminar el estudio del código, ya solo queda ver cuando terminamos el bucle y continuamos hacia la función **printf**:

8048599: **83 7d ec 01** **cmp** **DWORD PTR [ebp-0x14],0x1** ; se comprueba si ha habido error
804859d: **75 0d** **jne** **80485ac <printf@plt+0x244>** ; si no es igual, hay error, se lleva a Serial Invalido; si es igual continuamos
804859f: **8d 45 dc** **lea** **eax,[ebp-0x24] ; x/s (\$ebp-24) 0xffb66964: "Serial Valido!\n"**
80485a2: **89 04 24** **mov** **DWORD PTR [esp],eax**
80485a5: **e8 be fd ff ff** **call** **8048368 <printf@plt>**; se muestra en el terminal
80485aa: **eb 0b** **jmp** **80485b7 <printf@plt+0x24f>**
80485ac: **8d 45 ca** **lea** **eax,[ebp-0x36] ; x/s (\$ebp-36) 0xffb66952: "Serial Invalido!\n"trace**
80485af: **89 04 24** **mov** **DWORD PTR [esp],eax**
80485b2: **e8 b1 fd ff ff** **call** **8048368 <printf@plt>** ;se muestra en el terminal.

De esta manera, ya lo tenemos resuelto, si continuamos con **"c"**, se ira parando en el breakpoint de **"cmp edx,eax"**, vamos tomando el valor de **eax**, lo pasamos a su valor **ascii** y así tenemos el serial correcto.

```

gdb $ c

Breakpoint 2, 0x08048579 in ?? ()
2: $eax = 0x65
1: x/5i $pc
0x8048579: cmp     edx,eax
0x804857b: je     0x8048586
0x804857d: mov    DWORD PTR [ebp-0x14],0x0
0x8048584: jmp    0x8048599
0x8048586: add    DWORD PTR [ebp-0x10],0x1

```

El siguiente valor es **0x65** que corresponde la valor **ascii "e"**.

El único problema es que como el valor de **edx** no coincide, deberíamos cambiarlo cada vuelta con **set**, como esto es una lata, lo ideal sería transformar el salto condicional de **0x804857b** en un salto

incondicional, para ello primero tenemos que hacer que en el archivo se pueda escribir, eso podemos hacerlo al ejecutar gdb con la opción `--write` o como en nuestro caso, desde dentro de gdb con la orden `"set write on"` y hacemos el cambio:

```
gdb $ set write on
gdb $ x/i 0x804857b
0x804857b: je      0x8048586
gdb $ set {char}0x804857b=0xeb
gdb $ x/i 0x804857b
0x804857b: jmp     0x8048586
```

Vamos a explicarlo, uso `"x/i 0x804857b"` de modo que `x` nos muestra instrucciones en ensamblador de la zona que vamos a cambiar, y así comprobar que el parcheado ha sido correcto. Respecto a la orden `"set {char}0x804857b=0xeb"` como podéis comprobar en gdb para parchear se hace a nivel de opcodes, lo cual limita mucho sus posibilidades prácticas, pero para pequeñas cosas puede ser útil. Para saber los opcodes miramos en `objdump.txt` y vemos el salto condicional y uno no condicional:

`804857b: 74 09 je 8048586 <printf@plt+0x21e>` ; vemos que el opcode de salto condicional es `0x74` y `0x09` es el desplazamiento

`8048584: eb 13 jmp 8048599 <printf@plt+0x231>`; el opcode del salto incondicional es `0xeb` y el desplazamiento es `0x13`. Por tanto, he cambiado `0x74` por `0xeb` y así el salto en `0x804857b` siempre se produce aunque la comparación no sea correcta.

Del termino que se ponga entre las `{ }` dependerá la cantidad de bytes que se puedan parchear, los valores que podemos poner y su correspondencia en bytes lo vemos con este cuadro:

Tipo	Bytes	Nombre	Rango de valores
int	*	entero simple	depende del sistema.
unsigned int	*	entero simple sin signo	depende del sistema.
char	1	entero tipo char	-128 a 127
unsigned char	1	entero tipo char sin signo	0 a 255
short	2	entero corto	-32.768 a 32.767
unsigned short	2	entero corto sin signo	0 a 65,535
long	4	entero largo	-2.147.483.648 a 2.147.483.647
unsigned long	4	entero largo sin signo	0 a 4.294.967.295
enum	2	entero corto	-32.768 a 32.767

Lo único que puede variar es `int`, en mi caso con `{int}` me cambiaba el byte correctamente pero también afectaba a varios bytes seguidos, por lo que como solo queremos cambiar un byte debemos usar el tipo `{char}` que en este caso funcionó correctamente.

Respecto a los opcode, si por cualquier motivo (shellcode, curiosidad, etc..) quisiéramos guardarlos la orden en gdb sería esta:

```
gdb> x/32bx 0x804857b
0x804857b: 0xeb 0x09 0xc7 0x45 0xec 0x00 0x00 0x00
0x8048583: 0x00 0xeb 0x13 0x83 0x45 0xf0 0x01 0x83
0x804858b: 0x7d 0xf0 0x3d 0x7e 0xce 0xeb 0x07 0xc7
0x8048593: 0x45 0xec 0x00 0x00 0x00 0x00 0x83 0x7d
```

Solo haría falta saber desde donde, en este caso **0x804857b** y la cantidad, en este caso **32 bytes**; esto último se puede averiguar con `objdump` sin problemas.

Ahora que ya nos aseguramos que el salto después de la comparación (`cmp edx,eax`) siempre se produce, podemos continuar con "`c`", e irá parando en el breakpoint, donde tomaremos, uno tras otro, los valores de `eax` hasta obtener el serial correcto. Si como yo sois muy flojos, jeje, y no os apetece ir apuntando, podemos activar el **logging** en gdb, para que sea de esa manera como nos quede guardada toda esa información:

```
gdb $ set logging on
gdb $ c

Breakpoint 2, 0x08048579 in ?? ()
2: $eax = 0x6e
1: x/5i $pc
0x8048579: cmp     edx,eax
0x804857b: jmp     0x8048586
0x804857d: mov     DWORD PTR [ebp-0x14],0x0
0x8048584: jmp     0x8048599
0x8048586: add     DWORD PTR [ebp-0x10],0x1
```

Y continuamos con `<Enter>` durante 60 veces

De esta manera, tendremos un archivo en la misma carpeta del crackme, donde ejecutamos gdb, con el nombre **gdb.txt**, donde ira guardando todo lo que nos enseñe en pantalla, como el **display \$eax**, que en este caso es lo que nos interesa.

La verdad es que no lo hice (he dicho que soy muy vago,jeje) me pareció mas divertido hacerlo de otra manera:

- Solución en BASH.

Bash suele ser el intérprete de comandos de los sistemas Linux, aunque debemos saber que en algunas distribuciones o sistemas Unix podemos encontrar otros como **sh**, **ksh** o **csh**.

Para acceder a bash podemos utilizar una interfaz gráfica como el **terminal de gnome**, la **konsole** (KDE) o el clásico **xterm** o ir a un terminal del sistema **tty**, si estamos en el sistema gráfico podemos acceder con la combinación de teclas **Ctrl+Alt+F1**, **Ctrl+Alt+F2**, hasta **F6** y allí tendremos que hacer **login** para tener un prompt donde trabajar.

Estos terminales pueden ser útiles cuando hay bloqueo de la parte gráfica, podemos acceder a uno de ellos, ver que programa esta causando problemas (**ps ax**) y matarlo (**kill -9 PID_del_proceso**). Por cierto, tened en cuenta que la interfaz gráfica (Gnome, KDE.....) siempre se crea en el **tty 7**, por lo que para volver a ella, usamos la combinación **Ctrl+Alt+F7**.

De cualquier manera, el interprete de comando **Bash** se puede usar de dos formas; mediante una serie de ordenes desde la consola o podemos poner todas las ordenes en un archivo, darle permiso de ejecución y tendremos un **script de Bash**. De ambas maneras, tenemos a nuestra disposición un lenguaje de programación completo, en este caso interpretado, con el que podemos unir y combinar todas las herramientas de **GNU/linux** disponibles en un terminal.

Entremos en materia, primero debemos obtener los valores con los que trabajar, para ello salvamos los valores hexadecimales que utiliza el crackme en el archivo **base.txt**:

```
0x00000047 0x00000064 0x0000006c 0x0000006a
0x00000065 0x00000069 0x0000002a 0x00000027
0x0000007b 0x00000060 0x00000064 0x0000002b
0x0000007f 0x00000064 0x00000063 0x0000006d
0x0000007f 0x0000007d 0x0000007d 0x00000060
0x00000034 0x00000079 0x00000077 0x00000037
0x0000007b 0x00000076 0x00000069 0x0000007a
0x0000003c 0x0000006e 0x0000007b 0x0000003f
0x00000050 0x0000004e 0x0000004c 0x00000046
0x00000004 0x00000048 0x00000053 0x0000005e
0x00000008 0x00000040 0x00000044 0x0000005f
0x00000049 0x0000005f 0x0000004b 0x0000005c
0x00000051 0x0000005f 0x00000046 0x00000056
0x00000014 0x0000005b 0x00000059 0x00000017
0x0000005b 0x0000004b 0x0000005f 0x0000005e
0x0000004f 0x00000002
```

Para trabajar con ello, con gedit o cualquier editor de texto, creamos un array con los valores hexadecimales sin ceros ni x, quedando así:

```
(47 64 6c 6a 65 69 2a 27 7b 60 64 2b 7f 64 63 6d 7f 7d 7d 60 34 79 77 37 7b 76 69 7a 3c 6e 7b 3f
50 4e 4c 46 04 48 53 5e 08 40 44 5f 49 5f 4b 5c 51 5f 46 56 14 5b 59 17 5b 4b 5f 5e 4f 02)
```

****Es una sola linea****

Lo primero que se me ocurrió, es pasar los valores hexadecimales a decimales, pues pensé que me darían menos problemas trabajar con ello; para ello vamos a ver como cambiamos de base en **bash**:

```
$ echo $(( 2#11011 ))
27
```

Se pueden usar tanto doble parentesis como corchetes, lo importante es poner antes de la almohadilla # sobre que base estamos trabajando, en este caso base 2 o binaria, y después el número que vamos a comprobar. El resultado siempre nos lo da en decimal, que es justamente lo que andaba buscando. Vamos a ver mas ejemplos:

```
$ echo ${8#12}
10
$ echo $((16#3d))
62
$ echo ${16#A}
10
```

Primero debemos declarar el array en la consola donde vamos a trabajar, para ello solo hay que ponerle un nombre y asignarle con el signo = el array entre paréntesis:

```
$ array=(47 64 6c 6a 65 69 2a 27 7b 60 64 2b 7f 64 63 6d 7f 7d 7d 60 34 79
77 37 7b 76 69 7a 3c 6e 7b 3f 50 4e 4c 46 04 48 53 5e 08 40 44 5f 49 5f 4b
5c 51 5f 46 56 14 5b 59 17 5b 4b 5f 5e 4f 02)
```

Ahora podemos acceder a los valores del array de esta manera:

```
$ echo ${array[@]}
47 64 6c 6a 65 69 2a 27 7b 60 64 2b 7f 64 63 6d 7f 7d 7d 60 34 79 77 37 7b
76 69 7a 3c 6e 7b 3f 50 4e 4c 46 04 48 53 5e 08 40 44 5f 49 5f 4b 5c 51 5f
46 56 14 5b 59 17 5b 4b 5f 5e 4f 02
```

Así los vemos todos y si queremos acceder a uno de ellos, solo hay que recordar que el primer valor su orden es 0, por tanto para ver el último valor sería 61:

```
$ echo ${array[61]}
02
$ echo ${array[0]}
47
```

Ahora vamos a ejecutar un **for** para que nos pase todo el array a valores decimales:

```
$ for (( i = 0 ; i < ${#array[@]} ; i++ ))
> do
> echo ${16#${array[$i]}}
> done
71
100
108
106
101
.
.
.
```

Como veis al poner la orden **for** o cualquier palabra reservada (while,if,...), el prompt cambia a ">" que nos indica que esta esperando las ordenes siguiente del bucle y solo nos dará el resultado cuando pongamos **done** que indica el final del bucle. La variable **i** no hace falta declararla, de modo que **`\${16#\${array[\$i]}}** equivale a pasar de hexadecimales a decimales todos los valores del array..

Como veis **echo** nos ha devuelto cada valor en una linea, los podíamos copiar, pero ya sabemos que podemos direccionar la orden a un archivo. Para ello, al darle a las flechas arriba y abajo, lo que hacemos es recorrer el historial de ordenes de **bash** (también están en nuestro directorio personal en el archivo oculto **.bash_history**), que nos evita tener que repetirlos. Si le damos a la flecha para arriba, vemos que **bash** ha unido todas las ordenes anteriores en una linea y eso nos facilita direccionarlo a un archivo:

```
$ for (( i = 0 ; i < ${#array[@]} ; i++ )); do echo ${16#${array[$i]}};
done > decimal.txt
```

En el archivo **decimal.txt** pasamos todos los valores a una linea para hacer otro array de valores decimales y de esta manera los declaramos en la consola:

```
$ decimal=(71 100 108 106 101 105 42 39 123 96 100 43 127 100 99 109 127
125 125 96 52 121 119 55 123 118 105 122 60 110 123 63 80 78 76 70 4 72 83)
```



```
94 8 64 68 95 73 95 75 92 81 95 70 86 20 91 89 23 91 75 95 94 79 2)
```

Ahora a todos los decimales le vamos a realizar **xor** con el valor del contador que es **i**, para ello usamos un bucle **for**, teniendo en cuenta que el simbolo de xor es "^":

```
$ for (( i = 0 ; i < ${#decimal[@]} ; i++ ))
> do
> echo $(( ${decimal[$i]} ^ $i ))
> done
71
101
110
105
97....
```

Como ha salido en una columna, vamos a salvarlos a un archivo **xor.txt** y allí crear un array:

```
$ for (( i = 0 ; i < ${#decimal[@]} ; i++ )); do echo $(( ${decimal[$i]} ^ $i ))
); done > xor.txt
```

Ahora solo queda pasarlo a valores ascii, para ello vamos a utilizar **awk**, un gran lenguaje de procesamiento, con innumerables posibilidades y que como casi todas las herramientas unix se pueden integrar en un script. Primero declaramos el array de **xor.txt**:

```
$ xor=(71 101 110 105 97 108 44 32 115 105 110 32 115 105 109 98 111 108
111 115 32 108 97 32 99 111 115 97 32 115 101 32 112 111 110 101 32 109 117
121 32 105 110 116 101 114 101 115 97 110 116 101 32 110 111 32 99 114 101
101 115 63)
```

Ahora con un bucle **for** y **awk** pasamos todos los valores a **ascii**, de modo que ahora si tendremos nuestro serial correcto:

```
$ for (( i = 0 ; i < ${#xor[@]} ; i++ ))
> do
> echo ${xor[$i]} | awk '{printf "%c", $1}'
> done
Genial, sin simbolos la cosa se pone muy interesante no crees?
```

Ahora a toro "pasao", podemos ver que tanto el xor como pasarlo a ascii se podría hacer en un solo paso con los valores en decimal:

```
$ for (( i = 0 ; i < ${#decimal[@]} ; i++ )); do echo $(( ${decimal[$i]} ^ $i ))
) | awk '{printf "%c", $1}'; done
Genial, sin simbolos la cosa se pone muy interesante no crees?
```

En esta orden llama la atención el signo **|** o **tubería**, que es muy útil pues las salida de un programa puede ser enviado como entrada a otro, de forma que el resultado de "**echo \$((\${decimal[\$i]} ^ \$i))**" será la entrada de datos para **awk**. **Awk** en este caso lo único que hace es presentar con **printf** lo que recibe de **echo**, y ya sabemos que con **printf** podemos darle formato. Los valores que admite la sentencia **printf** en **awk** son varios (obtenido de "**man awk**"):

"La sentencia printf

Las versiones AWK de la sentencia printf y de la función sprintf()

(véase a continuación) aceptan la siguiente especificación para formatos de conversión:

%c Un carácter ASCII. Si el argumento usado para %c es numérico, es tratado como un carácter e imprimido. De otro modo, el argumento se asume que es una cadena, y solo se imprime el primer carácter de esa cadena.

%d

%i Un número decimal (la parte entera).

%e

%E Un número real de coma flotante de la forma [-]d.ddddde[+-]dd.
El formato %E usa E en vez de e.

%f Un número real de coma flotante de la forma [-]ddd.ddddd.

%g

%G Usese la conversión %e o %f, la que sea más corta, suprimiendo los ceros no significativos. El formato %G usa %E en vez de %e.

%o Un número sin signo en octal (entero.)

%s Una cadena de caracteres.

%x

%X Un número sin signo en hexadecimal (entero.) El formato %X usa ABCDEF en vez de abcdef.

%% Un carácter % ; no se utiliza ningún argumento. "

Por tanto con **printf " %c "** transforma cada valor que ha mandado "echo" a su valor en ascii y lo presenta en pantalla.

Por cierto, creo que al final esto ha tenido mas trabajo que seguir el valor en **gdb**, pero y lo que hemos aprendido; eso no tiene precio :-))

- Solución en C.

C es un lenguaje de programación muy longevo, se comenzó a utilizar en los sistemas Unix y también es parte de GNU/Linux. Es un lenguaje de medio nivel con características que le hacen ideal para controlar sistemas a bajo nivel, por lo que es muy útil para desarrollar sistemas operativos.

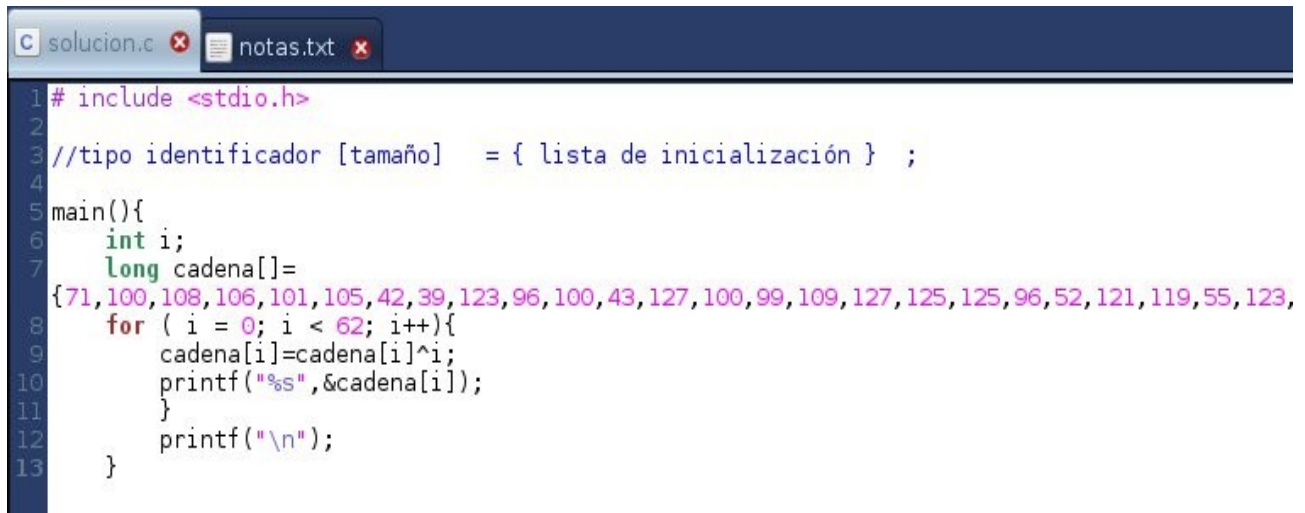
En este caso vamos a utilizar el programa **GCC (Gnu C compiler)** que en su origen era solo un compilador para C, pero actualmente se ha ampliado y da soporte a otros lenguajes, como C++ (g++), Java (gcj), fortran (gfortran), etc.... de modo que ahora se debe llamar **GNU Compiler Collection** (colección de compiladores GNU) pues es mucho mas completo.

En mi caso, estoy empezando por decimonovena vez (:-P) a programar en C y tenía claro que debía hacer cosas prácticas para avanzar, pues es la única manera de darle sentido a tanta teoría y tantas normas.

Vamos a hacer un programa que nos resuelva el crackme, para ello solo necesitamos un editor de texto, en este caso **gedit**, una vez escrito el código lo guardamos con extensión **.c** y desde el terminal ejecutamos **gcc** con esta orden para compilarlo:

```
$ gcc -g -o solucion solucion.c
```

Con la opción **-g** se guardarán los símbolos de depuración, con **-o** le damos nombre al archivo de salida, si por cualquier motivo olvidáis esta parte **gcc** le da el nombre **a.out** por defecto al programa que genera, y por último ponemos el archivo con el código, que en este caso se llama **solución.c**:



```
1 #include <stdio.h>
2
3 //tipo identificador [tamaño] = { lista de inicialización } ;
4
5 main(){
6     int i;
7     long cadena[]=
8     {71,100,108,106,101,105,42,39,123,96,100,43,127,100,99,109,127,125,125,96,52,121,119,55,123,
9     for ( i = 0; i < 62; i++){
10        cadena[i]=cadena[i]^i;
11        printf("%s",&cadena[i]);
12    }
13    printf("\n");
14 }
```

Como veis el programa no puede ser mas sencillo, declaramos el **array** (esta todo en la misma linea 7) como una variable **long** (como **int** no funcionaba) y con un bucle **for** le hago **xor** (signo **^**) a cada valor del array con su valor correspondiente del contador **i**; por último usamos **printf** para darle formato, en este caso con **%s** muestra una cadena de caracteres (string). Por último, pongo un salto de carro **\n** con **printf** para que quede bonito.

La parte que me dio mas problemas fue "**printf("%s",&cadena[i]);**" pues yo colocaba solo **cadena[i]**, cuyos valores eran los que quería imprimir pero el resultado eran unos valores muy raros, hasta que comprobé que debía pasarle la dirección de la cadena[i] y eso se representa anteponiendo el símbolo **&**. Un simbolito me tuvo entretenido toda una tarde,jeje

Como veis en la imagen anterior, el editor de texto puede ser cualquiera, pero viene bien que tenga numeración de líneas, pues los errores de compilación nos lo dan por linea, y así es fácil ver donde buscar. En este caso, al compilarlo no hay problemas y tenemos un ejecutable llamado **solucion** que nos da el resultado que ya conocemos:

```
$ ./solucion
Genial, sin simbolos la cosa se pone muy interesante no crees?
```

3.d) GDB Remoto.

Por último, vamos a ver como usar **gdb** en modo remoto, tanto para poder depurar un archivo que se encuentra en otro ordenador, como hacerlo en el mismo de forma remota, pues hay

opciones de gdb que solo funcionan de este modo (tracepoint...).

Para ello, como ya sabemos necesitamos un servidor, que nos presente el programa con el que vamos a trabajar, en este caso **gdbserver**, y un cliente como el mismo **gdb**, que depure el programa de forma remota.

Primero es importante que tanto en el lado del servidor como en el lado cliente se encuentre el programa. En este caso vamos a utilizar el programa **solucion**, que hemos compilado manteniendo los símbolos de depuración (opción **-g**) y veremos como estos símbolos son útiles para ver como se muestra en ensamblador las estructuras que ponemos en el código del programa.

- **Servidor**, se ejecuta igual que como vimos en el tutorial sobre IDA:

```
$ gdbserver --remote-debug localhost:2345 ./solucion
Process ./solucion created; pid = 28516
Listening on port 2345
```

-**Cliente**, se ejecuta gdb con el programa, esto ayuda a cargar los símbolos y después le damos la orden para que comience la depuración remota:

```
$ gdb -q solucion
Reading symbols from /home/juanjo/Descargas/crk/Crackme_
Yashira/crackme3/solucion...done.
gdb $ target remote localhost:2345
0x00007fe92cc0caf0 in ?? () from /lib64/ld-linux-x86-64.so.2
gdb $
```

Con la orden "**target remote**", gdb se conectara con el servidor, pero claro para ello necesita saber la dirección del servidor (**localhost** pero también puede ser una dirección ip, en este caso 127.0.0.1) y el puerto donde se crea la conexión (**2345**) separados por dos puntos.

Ahora trabajamos en la parte del cliente, gdb, ponemos los display que nos gusta y continuamos hasta la función **main**:

```
gdb $ b main
Breakpoint 1 at 0x400540: file solucion.c, line 7.
gdb $ display/5i $pc
gdb $ c

Breakpoint 1, main () at solucion.c:7
7      long cadena[]
={71,100,108,106,101,105,42,39,123,96,100,43,127,100,99,109,127,125,125,96,
52,121,119,55,123,118,105,122,60,110,123,63,80,78,76,70,4,72,83,94,8,64,68,
95,73,95,75,92,81,95,70,86,20,91,89,23,91,75,95,94,79,2};
1: x/5i $pc
0x400540 <main+12>:   lea    rdx,[rbp-0x210]
0x400547 <main+19>:   mov    ebx,0x400720
0x40054c <main+24>:   mov    eax,0x3e
0x400551 <main+29>:   mov    rdi,rdx
0x400554 <main+32>:   mov    rsi,rbx
```

Como se puede ver, el comienzo del código sirve para manejar el array, de modo que esta preparando los registros para moverlos a una zona de memoria.

Por cierto, antes de continuar, como estoy en Debian squeeze de 64 bits, **gcc** compila por defecto

ejecutables de 64 bits, que como podéis comprobar tiene nomenclatura diferente en los registros, podemos verlos usando la orden "**info registers**":

```
gdb $ info registers
rax      0x7fe92cc08ec8      0x7fe92cc08ec8
rbx      0x0      0x0
rcx      0x0      0x0
rdx      0x7fff2e622fa8      0x7fff2e622fa8
rsi      0x7fff2e622f98      0x7fff2e622f98
rdi      0x1      0x1
rbp      0x7fff2e622eb0      0x7fff2e622eb0
rsp      0x7fff2e622ca0      0x7fff2e622ca0
r8       0x7fe92cc07300      0x7fe92cc07300
r9       0x7fe92cc19dc0      0x7fe92cc19dc0
r10      0x0      0x0
r11      0x7fe92c8c9b50      0x7fe92c8c9b50
r12      0x400450      0x400450
r13      0x7fff2e622f90      0x7fff2e622f90
r14      0x0      0x0
r15      0x0      0x0
rip      0x400540      0x400540 <main+12>
eflags   0x206 [ PF IF ]
cs       0x33      0x33
ss       0x2b      0x2b
ds       0x0      0x0
es       0x0      0x0
fs       0x0      0x0
gs       0x0      0x0
fctrl    0x37f     0x37f
fstat    0x0      0x0
ftag     0xffff     0xffff
fiseg    0x0      0x0
fioff    0x0      0x0
foseg    0x0      0x0
fooff    0x0      0x0
fop      0x0      0x0
mxcsr    0x1f80 [ IM DM ZM OM UM PM ]
```

Respecto a esto, **gcc** trabajando en un sistema de 64 compila por defecto en 64 bits, pero si te interesa puedes compilar tanto en 32 como en 64, para ello esta la opción **-m**, de modo que se pondrá **-m32** o **-m64** (sin espacios) para obtener un ejecutable de 32 o de 64. Vamos a comprobarlo:

```
$ gcc -m32 -o solucion32 solucion.c
In file included from /usr/include/features.h:378,
                 from /usr/include/stdio.h:28,
                 from solucion.c:1:
/usr/include/gnu/stubs.h:7:27: error: gnu/stubs-32.h: No existe el fichero
o el directorio
```

He usado el mismo código pero el archivo de salida lo he cambiado **-o solucion32**. Como vemos nos da un error, le falta el fichero **stubs-32.h**. Una visita rápida en Google y descubro que en Debian este archivo se encuentra en el paquete **libc6-dev-i386** , gracias a <http://packages.debian.org> veo que es el paquete correcto para mi kernel (amd64):

```
File: /usr/include/gnu/stubs-32.h packages: libc0.1-dev [kfreebsd-i386], libc0.1-dev-i386 [kfreebsd-amd64], libc6-dev [not amd64, kfreebsd-amd64, kfreebsd-i386], libc6-dev-i386 [amd64]
```

Lo instalamos como root:

```
# aptitude install libc6-dev-i386
Se instalarán los siguiente paquetes NUEVOS:
gcc-4.4-multilib{a} gcc-multilib{a} lib32gomp1{a} libc6-dev-i386
```

Como vemos, además va a instalar otros paquetes por dependencias, en concreto **gcc-4.4-multilib**, **gcc-multilib** y **lib32gomp1**, lo que parece que ha solucionado el problema. Lo vamos a comprobar:

```
$ gcc -m32 -o solucion32 solucion.c
$ file solucion32
solucion32: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
$ file solucion
solucion: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.18, not stripped
```

Ahora ha compilado sin problemas y vemos con **file**, que tenemos un archivo de 32b y otro de 64b.

Seguimos con el de 64, continuamos con "**ni**" y podemos ver como ha preparando los registros para llegar a esta instrucción "**rep**":

```
gdb $
0x0000000000040055a 7 long cadena[]
={71,100,108,106,101,105,42,39,123,96,100,43,127,100,99,109,127,125,125,96,
52,121,119,55,123,118,105,122,60,110,123,63,80,78,76,70,4,72,83,94,8,64,68,
95,73,95,75,92,81,95,70,86,20,91,89,23,91,75,95,94,79,2};
1: x/5i $pc
0x40055a <main+38>: rep movs QWORD PTR es:[rdi],QWORD PTR ds:[rsi]
0x40055d <main+41>: mov DWORD PTR [rbp-0x14],0x0
0x400564 <main+48>: jmp 0x4005b9 <main+133>
0x400566 <main+50>: mov ecx,DWORD PTR [rbp-0x14]
0x400569 <main+53>: mov eax,DWORD PTR [rbp-0x14]
```

Aquí podemos ver como esta pasando los valores del **array** desde la dirección que marca **rsi** hasta la dirección que marca **rdi**:

```
gdb $ x/62g $rsi
0x400720 <C.0.2048>: 71 100
0x400730 <C.0.2048+16>: 108 106
0x400740 <C.0.2048+32>: 101 105
0x400750 <C.0.2048+48>: 42 39
0x400760 <C.0.2048+64>: 123 96
0x400770 <C.0.2048+80>: 100 43
0x400780 <C.0.2048+96>: 127 100
0x400790 <C.0.2048+112>: 99 109
0x4007a0 <C.0.2048+128>: 127 125
0x4007b0 <C.0.2048+144>: 125 96
0x4007c0 <C.0.2048+160>: 52 121
```

0x4007d0	<C.0.2048+176>:	119	55
0x4007e0	<C.0.2048+192>:	123	118
0x4007f0	<C.0.2048+208>:	105	122
0x400800	<C.0.2048+224>:	60	110
0x400810	<C.0.2048+240>:	123	63
0x400820	<C.0.2048+256>:	80	78
0x400830	<C.0.2048+272>:	76	70
0x400840	<C.0.2048+288>:	4	72
0x400850	<C.0.2048+304>:	83	94
0x400860	<C.0.2048+320>:	8	64
0x400870	<C.0.2048+336>:	68	95
0x400880	<C.0.2048+352>:	73	95
0x400890	<C.0.2048+368>:	75	92
0x4008a0	<C.0.2048+384>:	81	95
0x4008b0	<C.0.2048+400>:	70	86
0x4008c0	<C.0.2048+416>:	20	91
0x4008d0	<C.0.2048+432>:	89	23
0x4008e0	<C.0.2048+448>:	91	75
0x4008f0	<C.0.2048+464>:	95	94
0x400900	<C.0.2048+480>:	79	2

En este caso, como son valores de 64 bits, debemos colocar en la orden **x** el formato de **giant** (8 bytes), en concreto **62g**.

Si seguimos con la orden **"ni"** en esta instrucción **"rep"** no avanzamos, pues son 62 instrucciones pasando los valores de **rsi** a **rdi**, para estos casos viene muy bien la orden **"until"**, de modo que estando dentro de estas instrucciones repetitivas con **"until"** las pasamos de una vez:

```
gdb $ until
8      for ( i = 0; i < 62; i++){
1: x/5i $pc
0x40055d <main+41>:  mov    DWORD PTR [rbp-0x14],0x0
0x400564 <main+48>:  jmp    0x4005b9 <main+133>
0x400566 <main+50>:  mov    ecx,DWORD PTR [rbp-0x14]
0x400569 <main+53>:  mov    eax,DWORD PTR [rbp-0x14]
0x40056c <main+56>:  cdqe
```

Ahora podemos ver como continua coincidiendo el código con el desensamblado, comenzamos con el bucle **for**, y en concreto el contador que está en **[rbp-0x14]** se inicializa a 0, sería por tanto la parte **i=0**. Seguimos con **"ni"**:

```
gdb $
0x0000000000004005b9      8      for ( i = 0; i < 62; i++){
1: x/5i $pc
0x4005b9 <main+133>:  cmp    DWORD PTR [rbp-0x14],0x3d
0x4005bd <main+137>:  jle    0x400566 <main+50>
0x4005bf <main+139>:  mov    edi,0xa
0x4005c4 <main+144>:  call  0x400430 <putchar@plt>
0x4005c9 <main+149>:  add    rsp,0x208
```

Aquí pone el limite del contador, lo compara con **0x3d** y siempre que sea menor o igual se producirá el salto, en caso contrario se termina el bucle. Seguimos:

```
gdb $
9      cadena[i]=cadena[i]^i;
1: x/5i $pc
```

```

0x400566 <main+50>:  mov    ecx,DWORD PTR [rbp-0x14]
0x400569 <main+53>:  mov    eax,DWORD PTR [rbp-0x14]
0x40056c <main+56>:  cdqeq
0x40056e <main+58>:  mov    rdx,QWORD PTR [rbp+rax*8-0x210]
0x400576 <main+66>:  mov    eax,DWORD PTR [rbp-0x14]

```

Aquí comenzamos a realizar el xor de los valores del array con el contador, si seguimos veremos el proceso completo:

```

gdb $
0x00000000000040056e      9          cadena[i]=cadena[i]^i;
1: x/5i $pc
0x40056e <main+58>:  mov    rdx,QWORD PTR [rbp+rax*8-0x210]
0x400576 <main+66>:  mov    eax,DWORD PTR [rbp-0x14]
0x400579 <main+69>:  cdqeq
0x40057b <main+71>:  xor    rdx,rax
0x40057e <main+74>:  movsxd rax,ecx

```

Ya podemos imaginarnos que **[rbp+rax*8-0x210]** son los valores del array , se pasan a **rdx**; mientras que el contador se pasa a **rax** y se hace un **xor** entre ellos.

En este punto, voy a probar los **tracepoint**, un tipo especial de breakpoint que solo se pueden usar en forma remota, y que nunca acaba de funcionar.

Como vemos en la instrucción **0x40057b** se realiza el **xor** entre el valor del array y el contador (**xor rdx,rax**), quedando el valor definitivo en **rdx**, por tanto si nos interesara recolectar esos valores se puede utilizar un **tracepoint**, que podríamos poner en la instrucción siguiente al xor, **0x40057e**, y allí decirle que guarde los valores de **\$rdx**. Los tracepoint se ponen así:

```

gdb $ trace *0x40057e
Tracepoint 2 at 0x40057e: file solucion.c, line 11.
gdb $ actions
> collect $rdx
> end
gdb $ tstart
Target does not support this command.

```

Con la orden **"trace"** lo colocas en la dirección que te interese y con la orden **"actions"** le dices que debe hacer cada vez que pare, en este caso **"collect \$rdx"**, osea que guarde el valor de **\$rdx** y terminas con la orden **"end"**. Para comenzar a utilizarlo se usa **"tstart"**, pero vemos que no lo podemos utilizar con este target ¿? **"Target does not support this command"**.

Bueno, tampoco funciona, ni en remoto ni con símbolos; no se para que lo ponen si no se puede utilizar :-)

Seguimos viendo el código con **"ni"**

```

gdb $
8          for ( i = 0; i < 62; i++){
1: x/5i $pc
0x4005b5 <main+129>:  add    DWORD PTR [rbp-0x14],0x1
0x4005b9 <main+133>:  cmp    DWORD PTR [rbp-0x14],0x3d
0x4005bd <main+137>:  jle    0x400566 <main+50>
0x4005bf <main+139>:  mov    edi,0xa
0x4005c4 <main+144>:  call  0x400430 <putchar@plt>

```

Aquí terminamos el primer ciclo del **for** con el incremento del contador **i++** y también se valora que no pase de **0x3d**.

De momento con el bucle **for** nos quedamos, en otros tutoriales espero continuar con diferentes estructuras de control de un programa en C.

Por el momento, con este primer contacto de un ejecutable de 64 bits de forma remota, es suficiente. Si continuamos con "**c**" el programa termina, saliendo el resultado final en la ventana donde ejecutamos el **gdbserver**:

```
putpkt ("0K#9a"); [noack mode]
getpkt ("vCont;c"); [no ack sent]
Genial, sin simbolos la cosa se pone muy interesante no crees?
Child exited with status 10
putpkt ("W0a#e8"); [noack mode]
GDBserver exiting
```

Como hemos visto es interesante el tema de los símbolos, pues de esta manera podemos ir viendo diferentes estructuras de programación como se ensamblan realmente, lo que nos ayudará en los casos que tengamos que realizar un estudio de ingeniería inversa en un programa desconocido.

Sería una manera de unir el lenguaje C y el reversing en Linux, tal como hizo Ricardo Narvaja en su excelente curso de **C y Reversing** en windows:

<http://www.ricardonarvaja.org/WEB/C%20Y%20REVERSING/>

Conclusión:

Con esto acabo todo lo que quería comentar sobre GNU/Linux, su shell y como manejarse en el estudio de ejecutables, hemos visto en acción a **gdb** de forma local y remota, con un programa de 32 y otro de 64 bits, y en fin, esto es solo una ínfima parte de todas las posibilidades que tenemos en Linux, espero que quien haya llegado hasta aquí (¡¡valiente!!) le sirva para interesarse en este tema y como mínimo que haya entendido algo.

Agradecimiento:

Sin duda al gran Maestro, Ricardo Narvaja y a todos los CracksLatinoS; que nunca dejan de sorprenderme.



Cualquier comentario a [cvtukan\(arroba\)gmail.com](mailto:cvtukan(arroba)gmail.com) o en la lista de Crackslatinos.

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