Introducción al Cracking en Linux.

Programa:	**Linux CrackMe (Level:2) by cyrex**
Descripción:	Primer contacto con el cracking en linux.
Dificultad:	Bajísima.
Herramientas:	Herramientas del sistema, EDB, GDB con DDD.
Objetivos:	Encontrar un serial correcto.

Cracker: [Juan Jose]	Fecha: 18/03/07

Introducción:

Por fin, después de un tiempo trabajando con linux he podido hacer algo de cracking en este S.O.; no es nada novedoso, pero la verdad es que voy a empezar desde cero en el análisis de programas en linux; ya que siendo un sistema libre ,donde casi todos los programas son gratis, nunca ha tenido mucho interés el cracking. Eso también se puede comprobar por las pocas herramientas especificas que he encontrado y sobretodo en la poca información que podemos encontrar sobre el tema. Y como siempre en español, todavía menos; lo poquito que he encontrado está en el magnífico ezine **Set** (Saqueadores Edición Técnica):

http://www.set-ezine.org/noticias/index.php

El trabajo CRACKING BAJO LINUX de SiuL+Hacky estaba en varias partes por lo que yo lo he juntado en un solo documento que acompaña a este tute.

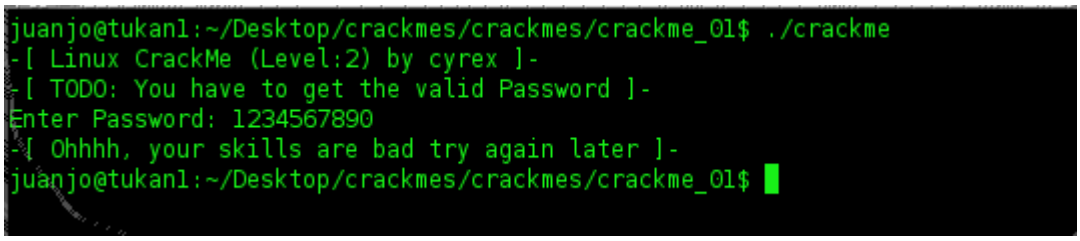
De todos modos, las pocas herramientas y la poca información no nos va a impedir poder hacer muchas cosas; pues como sabemos lo mas importante del cracking es nuestra habilidad y paciencia para utilizar el sistema en nuestro favor; y en eso linux nos va a ayudar; pues esta claro que contra mas conozcamos y manejemos los sistemas unix, mejor podremos estudiar y manipular los programas que corren sobre él.

Respecto a Linux, yo me he decantado por **Debian Etch**; pero en general cualquier distribución actual puede servirnos. La gran ventaja de Debian y todos sus derivados (como Ubuntu,Knoopix...) es la instalación de paquetes; que gracias al programa **apt-get** es muy sencilla, evitándonos todos los problemas de dependencias. También se puede usar **aptitude** (para consola) o **Synaptic** para el entorno gráfico; ambos basados en apt-get, que hacen el manejo de la instalación de programas más fácil.

Al Atake:

Como este crackme es muy fácil lo vamos a resolver con bastantes herramientas; es la mejor forma de empezar a manejarlas y ver como actúan. De esta manera tendremos una primera impresión de todas la posibilidades que nos ofrece linux, tanto si queremos usarlas para la ingeniería inversa o para el estudio de nuestros programas.

Primero vamos a verlo en acción, funciona dentro de una consola; por lo que hay que ir hasta carpeta donde tengamos el crackme; yo suelo utilizar **Konqueror** y cuando llegas a la carpeta con **F4** se te abrirá la consola en la posición que te interesa (también va muy bien MC y no hay que salirse de la consola). Primero miramos si tiene permiso de ejecución (si no, hay que dárselo **$ chmod u+x crackme**), lo ejecutamos colocando antes del nombre **./**
y vemos lo que nos pide:



Como vemos nos dice que tenemos que conseguir un serial valido; como hoy no tengo el día y no lo he acertado pues me dice que lo intente otra vez :-)

1. Herramientas del sistema.

GNU/Linux nos ofrece una gran cantidad de programas para analizar binarios; estos programas no siempre están instalados todos; pero en nuestro caso es fácil instalarlos ejecutando en una consola como root:

#aptitude install "programa que nos interesa"

En otras distribuciones habrá una forma similar y normalmente suelen venir en el CD o DVD de instalación.

Por cierto, todos estos programas se ejecutan desde una consola, en este caso desde **Konsole** pues utilizo **KDE** como escritorio.

1.a. Conociendo al enemigo: file.

Esta herramienta creo que esta presente en todos los Unix y nos da información del tipo de archivo que le pongamos como argumento:

$ file crackme

crackme: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0, dynamically linked (uses shared libs), for GNU/Linux 2.2.0, not stripped

Como vemos los archivos ejecutables en linux son del tipo **ELF,** muy diferente del formato PE de windows, que habrá que estudiar mas adelante, pues he encontrado muy poca información sobre su estructura. Además podemos ver que es un ejecutable **LSB**, osea que cumple el **Linux Standard Base**; que son unas serie de reglas para intentar compatibilizar las diferentes distribuciones y que permita que el programa se pueda ejecutar en cualquier sistema linux.
Respecto a **"dynamically linked"** se refiere a que en su compilación con **gcc** (compilador por defecto en linux) se ha hecho un linkado dinamico; osea que al ejecutarse y necesitar de librerias ajenas (**shared libs**= librerias compartidas); las buscará en tu ordenador. Este es el formato normal de compilar en Linux (para evitarlo debes colocar el argumento **-static** en tu compilación) y es lógico pues hace un ejecutable mas pequeño y rápido; pero también da lugar a las dependencias (una palabra maldita para los que luchamos con este S.O .); si al ejecutarse, las librería que necesita no están, no se podrá ejecutar y habrá que instalárselas para que funcione.
Por último vemos " **not stripped"**, esto significa que no se han eliminado los símbolos del fichero objeto; que son símbolos que genera el compilador y que nos van a dar mucha información. Si usamos un programa llamado **strip**, eliminara toda esa información y hará el ejecutable todavía mas pequeño y en la información saldrá **"stripped"** como es lógico. No confundir estos símbolos con los que busca **gdb** al iniciarse, esos símbolos es el código del programa que normalmente en los programas free de linux se tienen y se pueden utilizar, pero que nosotros nunca tendremos
Por cierto, en el caso de tener el código fuente, para ayudar a la depuración se puede añadir el flag **-g** al compilarlo, eso dejará mucha información para el debugger y nos ayudará a encontrar los problemas.

1.b nm

Como hemos visto, si no se ha utilizado **strip**, tenemos información del fichero objeto y el programa **nm** nos muestra esa información. Me he encontrado otros programas mejores y no iba a hablar de **nm**; pero me ha parecido interesante recordar una fuente interesante de información que nos da linux, es el comando **"man nombre_del_comando"**, a veces vamos directamente a internet sin mirarlo y cuando desesperado lo miramos; vemos la solución a nuestros problemas; por tanto para profundizar en todos los programas que comentaremos, mirad el **man**; pues aunque no siempre esta traducido es fácil ver lo que nos interesa. En este caso por ejemplo, vemos que con la opción **-D** nos dará todo lo relacionado con la librerías dinamicas:

```
$ nm -D crackme
         w __deregister_frame_info
         w __gmon_start__
08048604 R _IO_stdin_used
         U __libc_start_main
         U printf
         w __register_frame_info
         U scanf
         U strcmp
```

Pues nada mas y nada menos que nuestras amadas **APIs**, como vemos aquí casi todas son derivadas del lenguaje **C** (gran parte de linux esta hecho con ese lenguaje) y que no nos vendrá mal repasar sus funciones. En este caso llama la atención **strcmp,** que compara dos cadenas de caracteres; y me huele que sera

nuestra estrella invitada en este tute,jeje.

1.c. strings

Como hemos visto al ejecutar el programa hay un cartelito de chico malo **“[Ohhhh, your skills are bad try again later]”**; ¿y que hacemos cuando vemos algo así en un programa?? Buscamos las **strings referencias**; pues en linux haremos lo mismo; para ello tenemos el comando **strings** añadiendole como argumento el nombre del archivo, en este caso son muy pocas, pero conviene acostumbrarse a salvarlas a un archivo para poder repasarlo tranquilamente o incluso hacer búsquedas con cualquier editor de texto:

```
$ strings crackme > strings.out
```

Si vemos el archivo **strings.out**:

```
$ cat strings.out
/lib/ld-linux.so.2
libc.so.6
printf
__deregister_frame_info
strcmp
scanf
_IO_stdin_used
__libc_start_main
__register_frame_info
__gmon_start__
GLIBC_2.0
PTRhP
QVhP
[^_]
-[ Linux CrackMe (Level:2) by cyrex ]-
-[ TODO: You have to get the valid Password ]-
Enter Password:
47ghf6fh37fbgbgj
-[ Good, You're ready to begin linux reversing ]-
-[ Ohhhh, your skills are bad try again later ]-
```

Como veis, este crackme es una madre, tiene el valor correcto entre las strings del programa. Como os comente, ya no habría que hacer nada mas, si probamos ese serial vemos que es el correcto. Pero como hoy estamos viendo las herramientas, nos va a venir muy bien, es nuestro crackme de Cruehead para linux :-p

1.d. strace.

Este programa es mucho mas interesante, realiza un estudio dinámico del programa, pues lo ejecuta y nos da todas las llamadas al sistema que realiza el programa. De esta manera no dependemos tanto de ver si están o no los símbolos del objeto. La orden mas simple es esta:

```
$ strace -o strace.out ./crackme
```

La opción **-o** es para que mande la salida standar **“stdout”** al archivo con nombre strace.out, que colocará en el mismo directorio. En ese archivo podemos ver toda la información:

execve("./crackme", ["./crackme"], [/* 30 vars */]) = 0
uname({sys="Linux", node="tukan1", ...}) = 0
brk(0) = 0x804a000
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7f42000
access("/etc/ld.so.preload", R_OK) = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7f41000
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=81008, ...}) = 0
mmap2(NULL, 81008, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f2d000
close(3) = 0
access("/etc/ld.so.nohwcap", F_OK) = -1 ENOENT (No such file or directory)
open("/lib/tls/i686/cmov/libc.so.6", O_RDONLY) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\240O\1"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=1241580, ...}) = 0
mmap2(NULL, 1247388, PROT_READ|PROT_EXEC, MAP_PRIVATE|
MAP_DENYWRITE, 3, 0) = 0xb7dfc000
mmap2(0xb7f23000, 28672, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED|MAP_DENYWRITE, 3, 0x127) = 0xb7f23000
mmap2(0xb7f2a000, 10396, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f2a000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7dfb000
mprotect(0xb7f23000, 20480, PROT_READ) = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7dfb6c0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
munmap(0xb7f2d000, 81008) = 0
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 4), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7f40000
write(1, "-[Linux CrackMe (Level:2) by cy"..., 39) = 39
write(1, "-[TODO: You have to get the val"..., 47) = 47
fstat64(0, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 4), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|
MAP_ANONYMOUS, -1, 0) = 0xb7f3f000
write(1, "Enter Password: ", 16) = 16
read(0, "12345678\n", 1024) = 9
write(1, "-[Ohhhh, your skills are bad tr"..., 49) = 49
munmap(0xb7f40000, 4096) = 0
exit_group(0) = ?

Como veis, en este caso no nos sirve mucho; pero es una arma poderosa y además nos ayuda a comprender el funcionamiento del programa. Es curioso

que las llamadas del sistema son muy simbólicas y la mayoría son cómodas de entender.

1.e. ltrace.

También hace un análisis dinámico del programa, ejecutándolo y viendo las llamadas a las librerías que realiza. Como es lógico, cada llamada a una librería corresponde a una **API** y esto siempre nos interesa. La forma mas sencilla de usarla es:

```
$ ltrace -o ltrace.out -i ./crackme
```

En este caso la opción **-o** salva la salida a un archivo ltrace.out y la opción **-i** es para que nos diga la dirección donde es llamada la librería, de forma que tengamos una referencia para colocar si hiciera falta nuestros breakpoint. Si vemos la salida **ltrace.out**:

```
[0x8048391] __libc_start_main(0x8048450, 1, 0xbfb813f4, 0x80484f0,
0x8048550 <unfinished ...>
[0x8048439] __register_frame_info(0x8049750, 0x8049854,
0xbfb81348, 0xb7dcbd02, 0x80484f0) = 0
[0x8048463] printf("-[ Linux CrackMe (Level:2) by cy"...) = 39
[0x8048473] printf("-[ TODO: You have to get the val"...) = 47
[0x8048483] printf("Enter Password: ")                  = 16
[0x8048497] scanf(0x80486a1, 0xbfb81358,0xbfb81358,0x80482f9,1)=
1
[0x80484ab] strcmp("1234567890", "47ghf6fh37fbgbgj")   = -1
[0x80484d3] printf("-[ Ohhhh, your skills are bad tr"...) = 49
[0x8048403] __deregister_frame_info(0x8049750, 0xb7ee43a0,
0x8048700, 0xbfb81344, 0xbfb81344) = 0
[0xffffffff] +++ exited (status 0) +++
```

Como veis de esta manera también queda resuelto, de forma mas elegante y clara. Por cierto, insistiendo en la información de "**man**" y para que veáis lo útil que puede ser; imaginaros que es un programa grandísimo y la información que nos da de esta forma ltrace necesita horas para digerir; pero si la protección es parecida podríamos probar de esta forma:

```
$ ltrace -e strcmp -o ltrace.out1 -i ./crackme
```

Con la opción **-e** solo buscará el API en concreto que le digamos y puede ahorrarnos trabajo:

```
[0x80484ab] strcmp("47ghf6fh37fbgbgj", "47ghf6fh37fbgbgj") = 0
[0xffffffff] +++ exited (status 0) +++
```

Como veis concreto y al grano. En este caso he colocado el serial correcto, para que veáis el significado del numero que aparece al final de cada linea; es el número que devuelve la función después de ejecutarse; **strcmp** devuelve **0** si la comparación es correcta y **-1** si es incorrecta.
Por cierto, la dirección 0x80484ab no es la dirección donde se llamo a la API, es la dirección de retorno al programa, donde como vemos ya sabe el valor de

retorno de la función. Esto lo comprobé con la siguiente herramientas, el debugger.

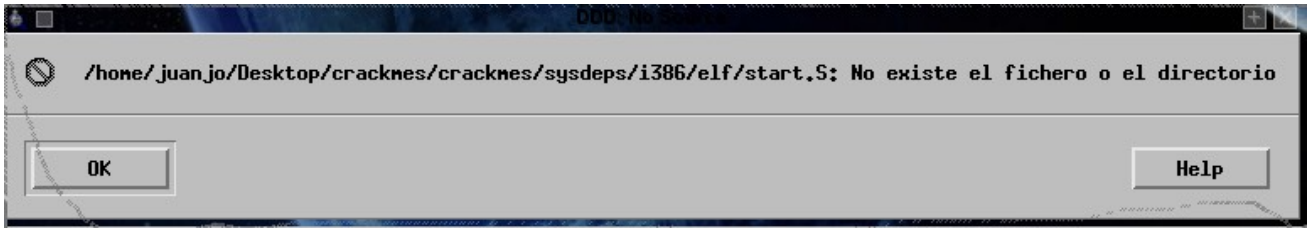
2.GDB y DDD.

En realidad **gdb** también es una herramienta del sistema; pues aunque no suele venir instalado, es el debugger de GNU, el mas completo y en el que se basan los otros que veremos. Pero también es complicado de utilizar; por lo que vamos a utilizarlo con **DDD**; que solo es un entorno gráfico para gdb. Esto nos ayudara a conocer las ordenes básicas de gdb y verlo todo un poco mejor ;-)

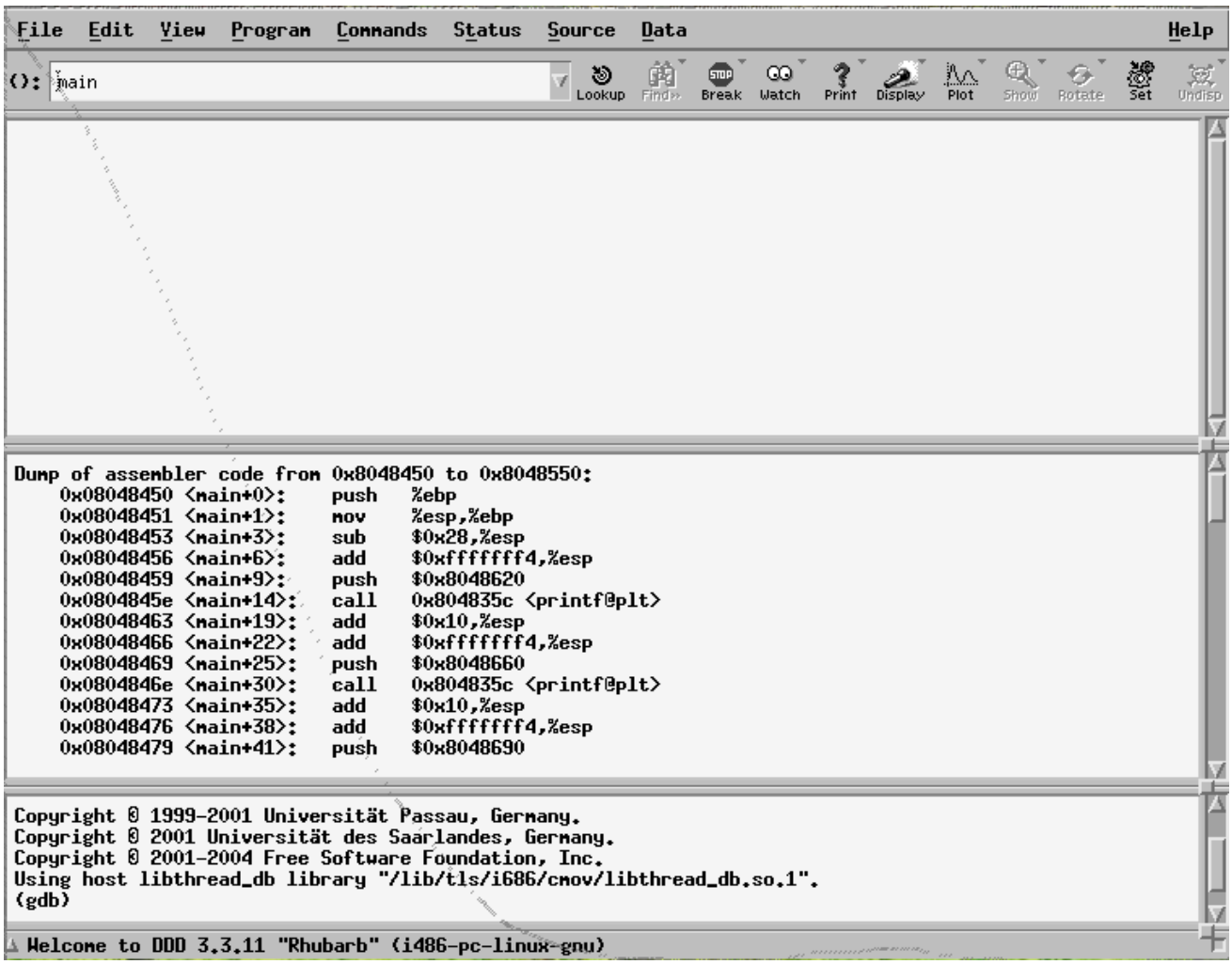
Para empezar, desde la consola en la carpeta del crackme ejecutamos:

$ ddd crackme

Hay dos motivos para hacerlo asi, primero es lo mas rápido y segundo, aunque sea un componente grafico con sus ventanitas y todo; las informaciones de salida de errores **“stderr”** son mas completos en consola y dan mucha información, esto nos vale siempre para linux, sobretodo, si como a veces pasa, el programa no arranca. En este caso, en la consola veréis muchos errores que no tienen importancia, y además, sale un mensaje de aviso:



Esto debe estar relacionado con los archivos de depuración, que como he comentado aparecen al compilar un programa con la opción **-g,** la verdad es que no lo se, pero el error no tiene importancia.
Lo aceptamos y el programa arranca sin problemas, como ya lo he ejecutado varias veces sale con el código:

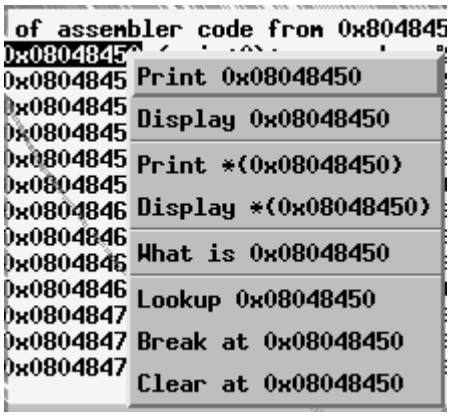


Realmente la primera vez que lo ejecutas se queda todo en blanco, pues no hay código fuente, y lo importante es que la ventana inferior debe quedar como veis; es el **gdb**, puro y duro, y se queda con el prompt **(gdb)** esperando recibir ordenes. Por tanto, podemos usar tanto el gdb directamente o lo que es mas comodo, mediante las opciones de ddd, vamos al menú **“View > Command tool”** y nos sale una ventanita con los comandos mas utilizados:

Lo primero es colocar un breakpoint en el **EP**, que como veis en la primera imagen es **8048450**; hay muchas formas de hacerlo, desde gdb:

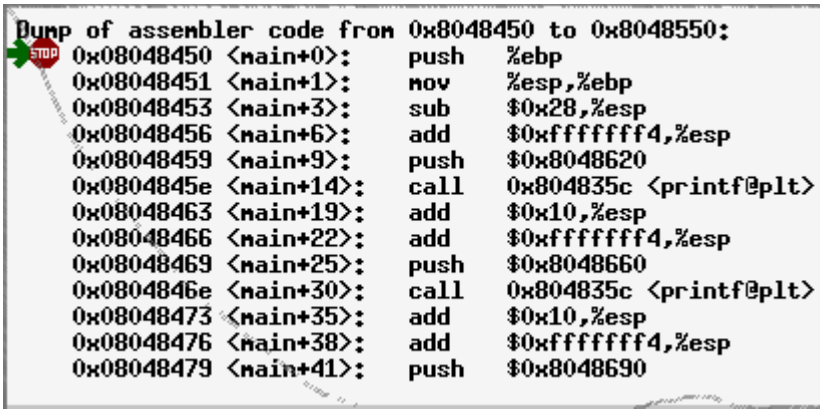
(gdb) break *0x08048450

Desde la zona de desamblado, se selecciona la dirección y con un click derecho tenemos varias opciones, entre ellas **break**:

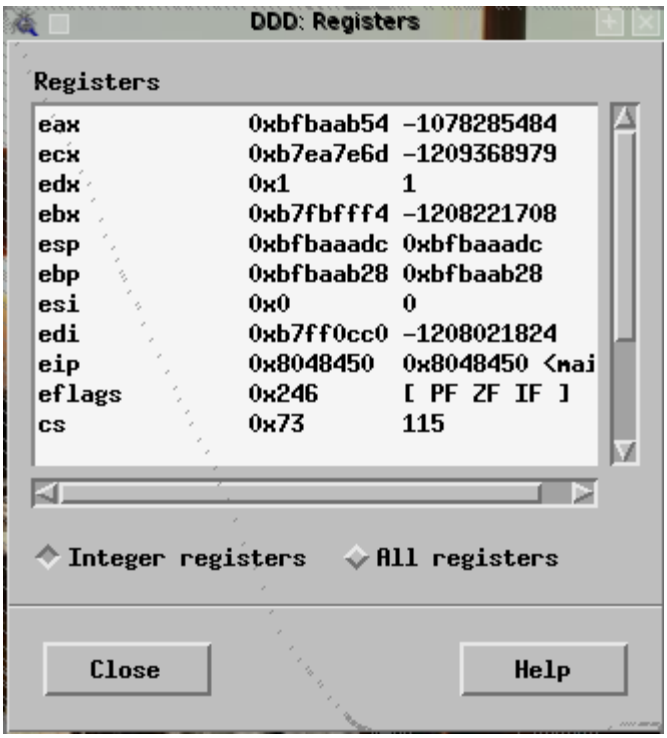


Y por último, desde el menu **“Source > Breakpoint”**, sale una ventana con todos los breakpoint que tenemos puestos, los puedes editar y puedes poner otros nuevos dándole al símbolo de stop.
De cualquier manera, una vez colocado el bpx se le da a **run**, en este punto es importante saber que DDD para todos los programas que funcionan en consola, como el crackme; va a llamar a **xterm** como consola predeterminada; por lo que como yo no la tenia instalada me dio algunos problemas, hasta que lo solucione con el clásico #aptitude install xterm

Por tanto, al darle a run nos aparecerá una ventana con xterm ejecutando el programa; aunque en este momento en blanco y el ddd parado en el bpx que hemos colocado:

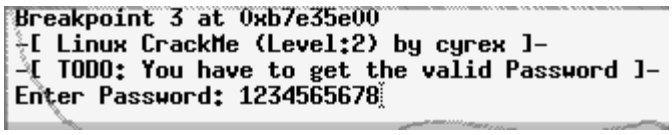


Como veis la flecha verde indica donde esta parado el programa y la señal de stop que hay colocado un bpx. Ahora es un buen momento para sacar los valores del registro, **“Status > Registers”**:



En el mismo menu de “status” hay otras opciones interesantes, como los threads, sygnals....etc.
Bueno vamos al asunto, sabemos por **ltrace** que la función **strcmp** esta en 0x80484ab, la buscamos en el source y vemos lo que os comente antes:

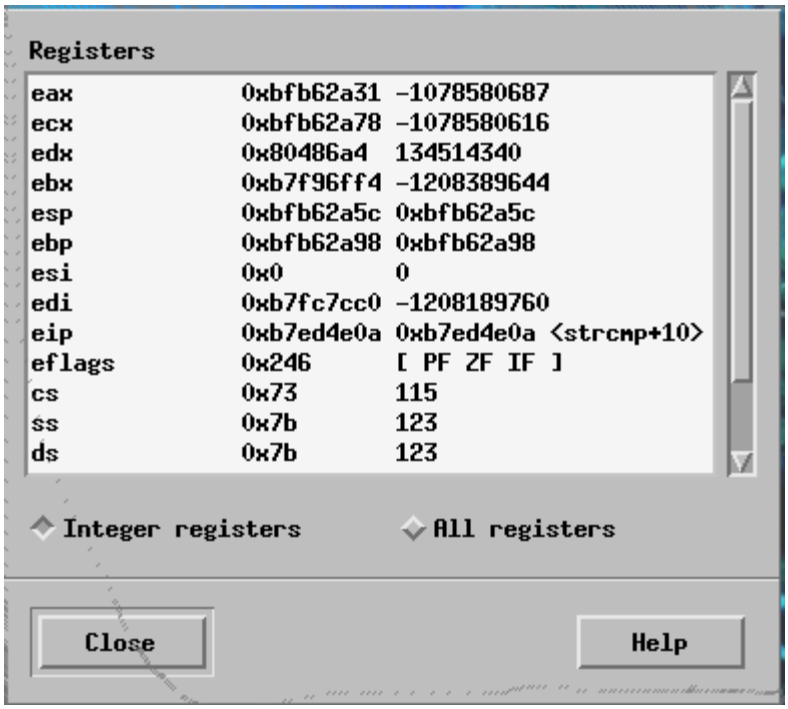
Como veis la posición 0x80484ab es la de retorno, por lo que colocamos un bpx en la llamada en 0x080484a6; ahora si se le puede dar a **“run”**; no se porque pero tuve que quitar el bpx del EP, pues no podía saltar la interrupción ¿? Para quitarlo, solo lo seleccionamos y con un clic derecho se le da a “**Clear...**”
Ahora si que se ejecuta el programa, alguna vez en la ventana de **xterm** y otra veces en la ventana de **(gdb)**; este ddd+gdb esta un poco loco |-)



Estemos donde estemos, colocamos el serial falso, no sin algún problemilla para escribirlo, damos **enter,** y si no hace cosas raras (que a veces lo hace.....) se parara en nuestra llamada.
Bueno una vez que pare en la llamada, entramos en ella con **stepi**, vemos que llega un momento que la zona de codigo se queda en blanco; pero si seguimos llegamos a la librería en este caso **/lib/tls/i686/cmov/libc.so.6** , pasamos por una zona intermedia que nos lleva a la función **strcmp** y nada mas empezar a

trasear llegamos a una comparación de dos registros:

Como veis en 0xb7ed4e0a compara **al** con el valor que indica **edx**. Por cierto, para liarlo todo un poco mas, el desensamblado que vemos es igual al formato que he visto en el cursillo de HLA (un saludo FL :-) , por lo que en Ollydbg seria **cmp al, byte ptr [edx],** osea el origen y destino están al revés. En el registro vemos:

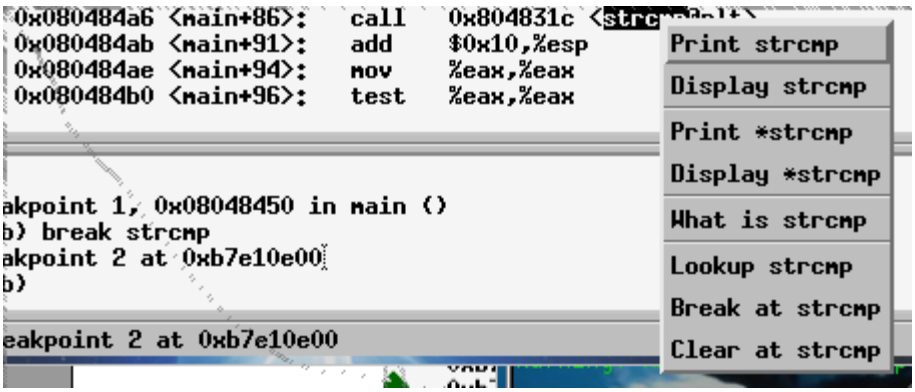


El valor de **al** es **31**, por lo tanto si miramos el valor que señala **edx:**

Esta orden es interesante para recordar, sobretodo si solo utilizamos gdb; **x / 5xw $edx** , se ven 5 doble word (w) en formato hexadecimal (x); lo que vemos es un codigo Ascii muy sospechoso, pues teniendo en cuenta que la estructura

de memoria es little indian (siempre se debe leer primero el byte menos significativo), podemos comprobar que aquí tenemos nuestro serial correcto 47ghf6fh37fbgbgj, y lo esta comparando byte a byte con el serial trucho.

Otra manera mas directa que admite **ddd,** es colocar el bpx directamente en la API, para ello en el código inicial seleccionamos la función y con un clic derecho le colocamos el bpx:



Como veis en gdb es un simple **break strcmp**; esto nos lleva directamente a la zona caliente y fácilmente veremos la comparación.

3.EDB.

Una agradable sorpresa para el cracking en linux y para los amantes del Ollydbg, es un programa que encontré en el foro de Woodmann, es un frontends para gdb pero la verdad muy conseguido y al igual que el Olly, da mucha información de forma rápida (tanto cuesta hacer las cosas asi?? que porque sea linux, todo tiene que ser complicado??grrrrrrrrrrr)
Lo podéis encontrar en:

http://www.codef00.com/projects.php#Debugger

Instalación:

En este caso el programa no esta en el repositorio de Debian, por lo que hay que compilarlo desde el código fuente, que es el paquete que nos hemos bajado, **debugger-0.8.12.tgz**; se descomprime en una carpeta y nos vamos a ella; lo primero es leerse el archivo **readme** (en otros casos es el archivo install), que nos indica que el programa necesita de las librerias **QT4**, de la versión 4.1 o posterior, por lo que la instalo pues yo tenia la **QT3**, y se compila con dos ordenes:

$ qmake
$ make

La primera no da problemas, pero la segunda nos da el error de que no encuentra las librerías de qt4, jeje, ya empezamos..Bueno la verdad que con

este problema ya me he enfrentado, todo esta relacionado con **alternatives**; que es la forma que Debian (no se si está en otras distribuciones) tiene configurado la elección del programa cuando hay varias opciones; por ejemplo, respecto a los navegadores, si tenemos instalado **firefox** y **konqueror**; con update-alternatives podemos decirle al sistema que ante un enlace a la web se lo mande a un programa u otro. Para ello se ejecuta como root:

 # update-alternatives --config x-www-browser

Hay 3 alternativas que proveen `x-www-browser'.

```
  Selección     Alternativa
-----------------------------------------------
*+        1    /usr/bin/konqueror
          2    /usr/bin/iceape
          3    /usr/bin/iceweasel
```

Pulse <Intro> para mantener el valor por omisión [*] o pulse un número de selección: 3
Se utiliza `/usr/bin/iceweasel' para proveer `x-www-browser'.

Como veis el asterisco indica el programa que esta predeterminado: si te interesa mantenerlo al darle a Intro, se queda la opción como estaba, pero si pones el número de otra opción, el sistema te abrirá siempre todos los enlaces con el que tu eligas, yo en este caso he elegido **iceweasel**, que no es mas que firefox en Debian, que le han cambiado el nombre por los problemas de copyright y esas cosas.....

Por tanto, en este caso el sistema esta predispuesto a utilizar qt3 y tuve que cambiarlo en varias puntos. Para ello se ejecuta:

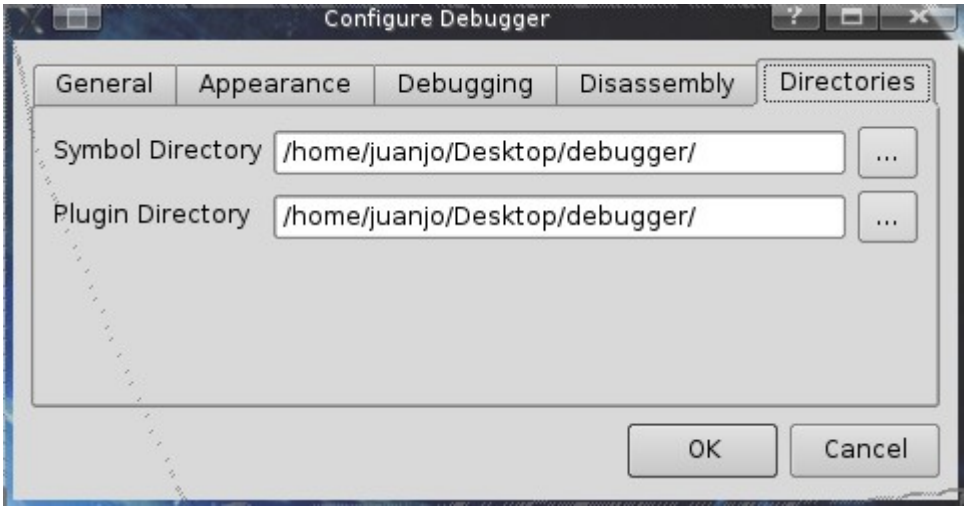
 # update-alternatives –all

Y donde salia las dos opciones, elegía qt4 y listo. Vuelvo a repetir el qmake y el make y el programa se compila perfectamente, quedando los ejecutables, librerías y plugins en la misma carpeta donde lo hemos descomprimido.

Por último como dice el readme hay que enlazar los simbolos, para ello se ejecuta un script que acompaña al código:

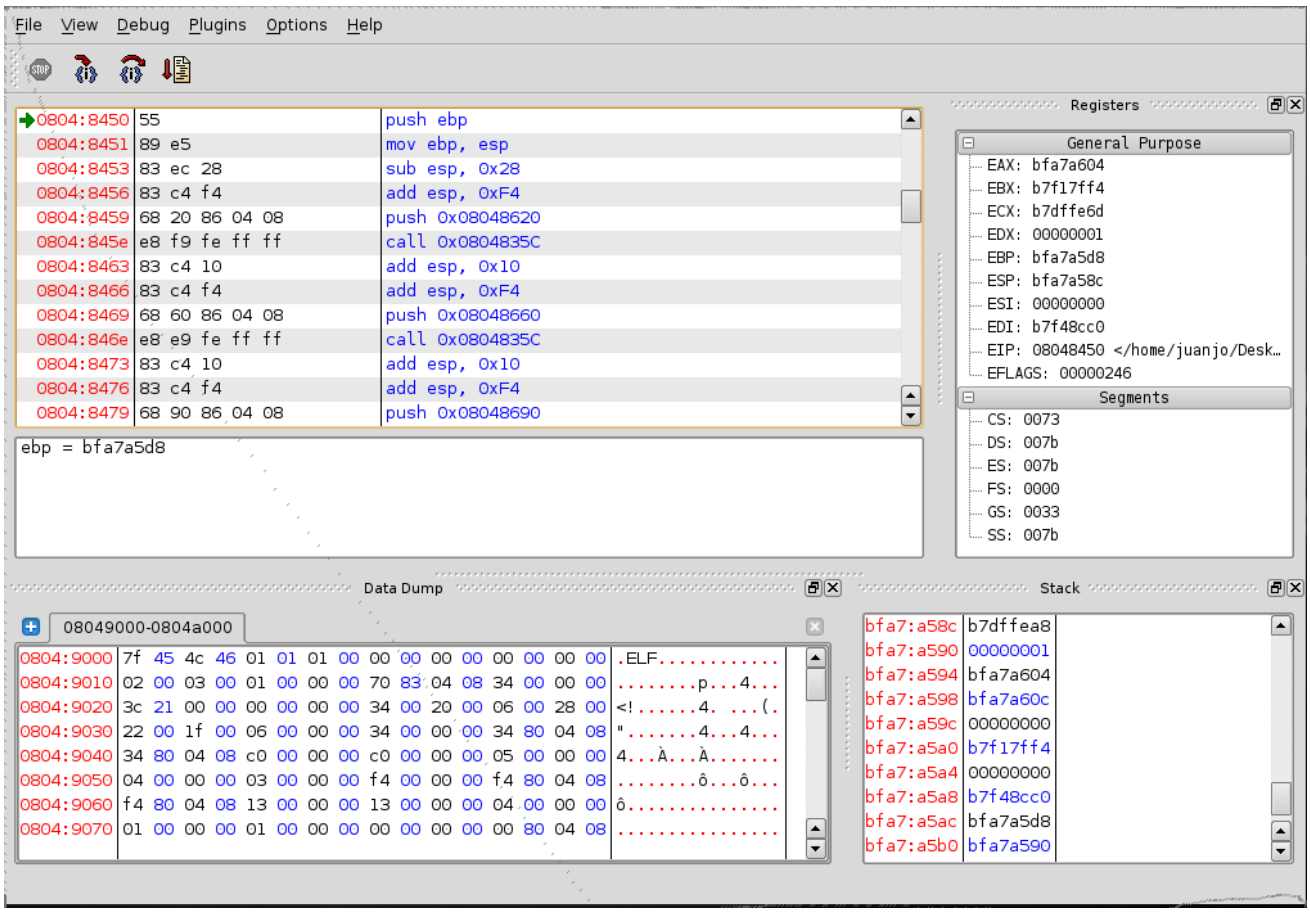
$ ./make_symbolmap.sh

Ahora ya podemos ejecutar el archivo **edb** y nos sale el programa; la primera vez nos pide que configuremos los directorios de los símbolos y plugins:

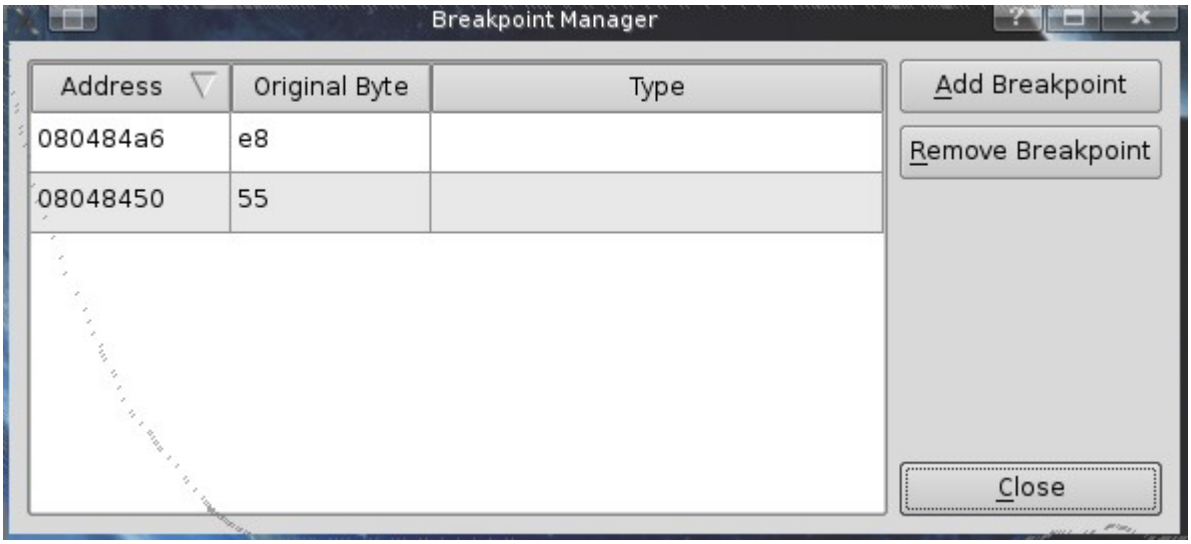


Aquí tuve un pequeño problema, pues como en Olly, puse la dirección de ambas carpetas; **/home/juanjo/Desktop/debugger/symbols** y **/home/juanjo/Desktop/debugger/ plugins**; pero el programa no encontraba algo y se cerraba; haciendo pruebas comprobé, como veis en la imagen anterior, que había que poner la dirección de la carpeta principal, donde se encuentran tanto los simbolos como los plugins. De esta manera, se ejecuta sin problemas.

Ahora ya toca disfrutarlo, vamos al menú **“File > Open”** y elegimos el crackme, y nos sale una imagen que nos agrada bastante ;-)

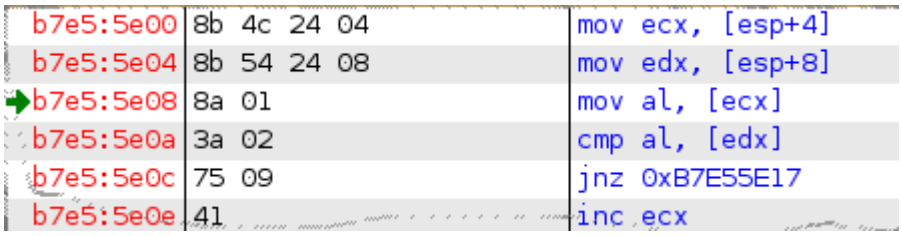


Como veis la cosa promete; pero hay que tener en cuenta que le queda bastante que mejorar; de momento en estos crackmes que se ejecutan en un terminal, la consola no aparece y no podemos meter los datos. De todas maneras el crackme se ejecuta perfectamente, y podemos llegar a la misma zona que con gdb. Si colocamos un bpx en la dirección de la llamada a **strcmp** 0x080484a6; esto se hace en el menú **“Plugins > Beakpoint manager”**:

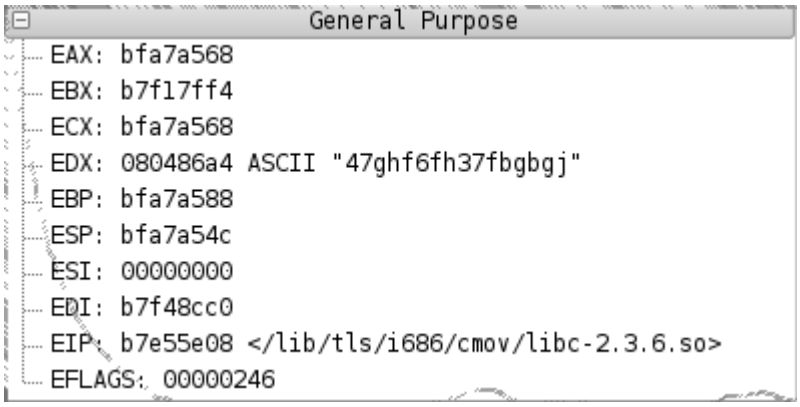


Como veis tuve que poner el bpx en el EP; pues paraba antes, y también he

puesto el bpx en la llamada a **strcmp**, solo es necesario darle a **“Add Breakpoint”** y darle la dirección; después le damos a **F9** y nos para sin problemas. A partir de aquí seguimos con **F7** al principio y después con **F8** hasta llegar a la zona caliente:



Como veis la estructura del ensamblado es igual a la que estamos acostumbrados en el Olly, y si miramos en los registros vemos que estamos en la zona correcta:



Se pueden ver algunos detalles, no es exactamente la misma dirección que con gdb (0xb7ed4e0a) y la librería tampoco es exactamente la misma (lib/tls/i686/cmov/libc.so.6), aunque esto último tiene menos importancia pues libc.so.6 es un enlace simbólico a libc-2.3.6.so. Por lo tanto comprobamos que EDB funciona muy bien y nos ha llevado al mismo sitio; tiene además muchos mas plugins interesantes que ya iremos viendo y seguro que este programa mejorara poco a poco hasta convertirse en un gran debugger de linux.
Solo nos queda felicitar a Evan Teran (http://www.codef00.com/) por su gran trabajo y que esperemos siga mejorando.

Conclusión.

Como siempre el tute me ha salido bastante mas largo de lo que quisiera, espero que se haya entendido y que sea un principio para ver muchos mas tutoriales sobre cracking en linux.
Por mi parte, me he dejado muchas mas herramientas que ver, tanto editores hexadecimales como desensambladores y mas herramientas del sistema; por lo que espero tener tiempo para continuar con las dos cosas que mas me gustan, el cracking y linux ;-)
Como veis estoy empezando; cualquier sugerencia,idea, corrección, nuevas herramientas....etc lo podemos ver en la lista:
http://groups.google.com/group/CrackSLatinoS.

Agradecimiento: Sin duda al gran Maestro, Ricardo Narvaja y a todos los CracksLatinoS; que como siempre digo, son los mejores.....