Introducción al Cracking en Linux 4 – Manejo de Radare.

Programa:	**Linux CrackMe1.**
Descripción:	Continuamos con el cracking en linux.
Dificultad:	Bajísima.
Herramientas:	**Radare by pancake.**
Objetivos:	Manejo básico de esta completa herramienta enfocada a la ingeniería inversa..

Cracker: [Juan Jose]	Fecha: 04/04/2009

Continuamos con **Radare**, vuelvo a recordar el ebook base de su manejo; donde encontrareis todo lo necesario:

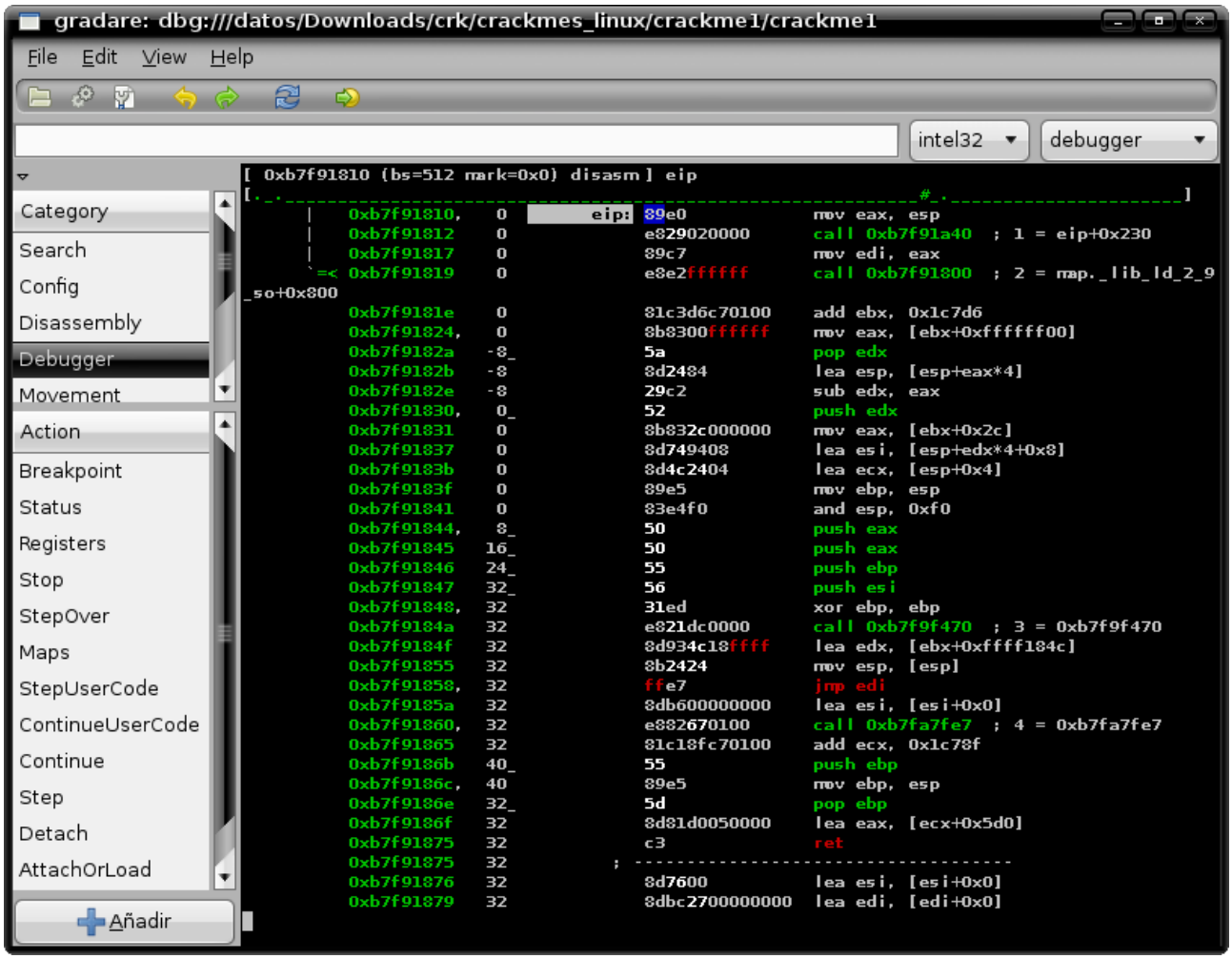
http://radare.nopcode.org/get/radare.pdf

Vamos a repasar lo básico y para hacerlo un poco mas entretenido, iremos viendo un crackme muy sencillo, que nos ayudará a ver radare en acción.

Mi primera recomendación es que tengáis claro que es un programa que actúa en linea de comandos; no tiene nada que ver con nuestro querido Ollydbg. En este caso, como muchos grandes programas de linux, no hay limites, pero su curva de aprendizaje, es mas alargada en el tiempo pues se necesita mas preparación. Espero que estas notas ayuden a empezar a utilizarlo, dando unas ideas generales enfocadas al estudio de binarios **ELF**.

Para empezar a utilizarlo, una vez instalado, solo necesitamos un terminal de **GNU/Linux**; que aconsejo estirar un poco, ancho de 110, para que las lineas de desensamblado salgan sin cortar, al igual que es bueno darle mas altura para poder ver el código bien. También hay un programa gráfico que engloba radare, se llama **gradare**, se puede ejecutar desde una linea de comando o con la combinación de teclas **ALT+F2**, escribes "**gradare**" (sin comillas) y te da una interfaz gráfica donde ejecutar radare, con mucha ayuda y se ve bastante bien; el problema es que el debugger me

ha dado problemas, por lo que creo que tiene que mejorar, pero es una buena manera de presentaros el programa:



Todo lo que veamos desde ahora en adelante se puede aplicar en gradare, solo tienes que tener en cuenta que por defecto te muestra el programa en modo visual; pero como ya veremos, podemos pasarle comandos con las mismas ordenes que en radare.

Configuración.

Como casi todos los programas de shell, se pueden configurar de dos maneras:

1.Desde la linea de comandos. Esta es la forma mas utilizadas, en un terminal linux ponemos la orden **radare -h** y nos va a dar la información que buscamos:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare -h**
radare [options] [file]
-s [offset] seek to the desired offset (cfg.seek)
-b [blocksize] change the block size (512) (cfg.bsize)
-i [script] interpret radare or ruby/python/perl/lua script
-p [project] load metadata from project file
-l [plugin.so] link against a plugin (.so or .dll)
-e [key=val] evaluates a configuration string
-d [program|pid] debug a program. same as --args in gdb
-f set block size to fit file size

-L list all available plugins
-w open file in read-write mode
-x dump block in hexa and exit
-n do not load ~/.radarerc and ./radarerc
-v same as -e cfg.verbose=false
-V show version information
-u unknown size (no seek limits)
-h this help message

Como vemos el formato es **radare [opciones] [file]**; si no colocamos ninguna opción el programa actúa como un **editor hexadecimal** y **desensamblador**, es la forma ideal para estudiar cualquier binario y para analizar el código muerto de programas sospechosos (malware, virus, etc...) sin poner en peligro el sistema.

La opción que mas vamos a utilizar es **-d**, **debugger**, que nos va a dar todas las posibilidades del editor hexadecimal y desensamblador y además podemos ver el código en acción. Esta será la opción que voy a explicar a lo largo de este texto. Como podemos ver la opción **-d** admite dos formas de referirse al archivo, una con el nombre del programa a estudiar, si el terminal esta abierto en la misma carpeta donde se encuentra el programa, con el nombre bastará:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ radare -d crackme1

Si no estamos en la misma carpeta debemos poner el **path** hasta el programa:

juanjo@cvtucan:~$ radare -d /datos/Downloads/crk/crackmes_linux/crackme1/crackme1

Yo suelo utilizar la primera opción con un navegador gráfico que me de la opción de abrir el terminal en el lugar donde este (konqueros, xfe, nautilus con un script.....) y ahorramos dedos ;-).

La segunda acción es abrir un proceso en ejecución, para ello debemos saber el **PID** del proceso, para lo cual podemos utilizar la orden **ps ax**:

juanjo@cvtucan:~$ **ps ax**
PID TTY STAT TIME COMMAND
1 ? Ss 0:03 init [2]
2 ? S< 0:00 [kthreadd]
3 ? S< 0:00 [migration/0]
4 ? S< 0:31 [ksoftirqd/0]
5 ? S< 0:00 [watchdog/0]
6 ? S< 0:00 [migration/1]
7 ? S< 0:44 [ksoftirqd/1]
8 ? S< 0:00 [watchdog/1]
9 ? S< 0:38 [events/0]
10 ? S< 0:09 [events/1]
11 ? S< 0:00 [khelper]
44 ? S< 0:03 [kblockd/0]
45 ? S< 0:00 [kblockd/1]
47 ? S< 0:00 [kacpid]
48 ? S< 0:00 [kacpi_notify]
123 ? S< 0:00 [kseriod]
163 ? S 0:02 [pdflush]
164 ? S< 0:15 [kswapd0]

165 ? S< 0:00 [aio/0]

Esto es solo una parte de la salida de este comando, como veis en la primera columna esta el **PID** de cada programa en ejecución, tomáis el dato de la victima y ejecutáis radare. Para evitar buscar entre tantos programas, es mejor que linux busque por ti, por ejemplo si me interesa saber el PID de **conky** (un monitor del sistema muy útil), hay que utilizar la orden **grep** junto a **ps ax**:

juanjo@cvtucan:~$ **ps ax | grep conky**
2722 pts/1 S+ 0:00 grep conky
6110** ? S 41:46 **conky

Como veis **grep** te muestra las lineas de la salida estandar de la orden **ps**, que coincidan con el patrón que le has dado, en esta caso salen dos con la palabra conky, pues grep como programa también se está ejecutando con esa opción, pero el que nos interesa es el **6110** que es conky:

juanjo@cvtucan:~$ **radare -d 6110**
Attached to pid '6110'.
PID = 6110
open debugger ro 6110

Hay otra forma igual para ejecutar radare como debugger, si esta en la misma carpeta se debe poner:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare dbg://crackme1**

Y si no esta en la misma carpeta se debe poner el **path**:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare dbg:///bin/ls**

Tened en cuenta que el path es **/bin/ls .** Y también sirve con el **PID**:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare dbg://6709**

Otra opción que utilizaremos es **-w.** Si os fijáis en la salida anterior, vemos *"open debugger **ro** 6110"* significa que el debugger ha abierto el programa que corresponde con el PID 6110 como solo lectura, por lo tanto si queremos cambiar el código no podremos ; si ya tenemos abierto el programa con radare, podemos cambiar la variable de configuración correspondiente y eso nos permitirá escribir en el archivo, en ese caso se debe poner en el prompt del programa:

>**eval file.write=true**

Y ya podremos utilizar las orden correspondiente para parchear el ejecutable (que en este caso es write o **w** también). Pero si desde el principio ya sabemos que tendremos que cambiar el ejecutable, algo muy normal ;-), debemos iniciar radare con la opcion **-w**:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare -dw crackme1**
argv = ['crackme1',]
7 imports added
34 symbols added
Program 'crackme1' loaded.
PID = 7236
open debugger rw crackme1

Como es lógico, siempre es conveniente hacer una copia de seguridad del ejecutable antes de jugar con él, en linux desde la consola es tan fácil como hacer una copia del programa con otro nombre:

```
juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ cp crackme1
copia_crackme1
```

Por último otra opción muy interesante es **-e,** se utiliza para modificar de inicio una variable de configuración. Vamos a profundizar un poco en las variables de configuración, pues el programa funcionará de una manera u otra según los valores de estas variables; dentro del programa podemos ver estas variables globales con la orden **eval** o lo que es equivalente "**e**", primero podemos ver la ayuda de este comando:

```
[0xB7FD9810]> e?
Usage: e[m] key=value
 ereset          ; reset configuration
emenu            ; opens menu for eval
e scr.color = 1  ; sets color for terminal
```

Como veis la ayuda se consigue de cualquier orden con ponerle "?" delante (muy útil), de esta manera podemos darle un valor diferente a una variable concreta o ver todas las variables con **emenu**:

```
[0xB7FD9810]> emenu
- asm
- cfg
- child
- cmd
- dbg
- dir
- dump
- file
- graph
- gui
- io
 - range
 - scr
- search
- trace
- vm
- zoom
> _
```

Como veis el prompt del programa cambia, pues como hay muchos grupos de variables, esta esperando que elijamos alguno, en este caso vamos a ver **cfg**:

```
> cfg
- addrmod      =       4
- analdepth    =       6
- bigendian    =       false
- bsize        =       512
- count        =       0
- datefmt      =       %d:%m:%Y%H:%M:%S%z
```

```
- debug          =       false
- delta          =       4096
- editor         =       vi
- encoding       =       ascii
- fortunes       =       true
- inverse        =       false
- limit          =       0
- noscript       =       false
- rdbdir         =       TODO
- sections       =       true
- vbsize         =       1024
- vbsize_enabled         =       false
- verbose        =       1
> _
```

De esta manera vemos los valores que estamos usando y si queremos cambiar algunos, solo tendremos que salirnos de emenu con la orden que mas se utiliza "**quit**" o **q;** volveremos al prompt normal y pondremos la variable de esta manera:

```
> q
[0xB7FD9810]> e cfg.bsize=1000
```

Como veis la orden para cambiarlo es **e** y la forma de poner la variable es primero el grupo donde pertenece, **cfg** en este caso, y después el valor concreto que vamos a modificar, separados por un punto, **bsize** es el que hemos tomado como ejemplo, es el tamaño en bytes del bloque que muestra radare por defecto. Para comprobar que se ha producido el cambio se pone la misma orden sin asignarle ningún valor:

```
[0xB7FD9810]> e cfg.bsize
1000
```

Hay que fijarse que cada variable su valor es diferente, algunos son numéricos, otros valores booleanos (true,false)..., hay que tenerlo en cuenta a la hora de cambiarlos.

Estos cambios se pueden hacer al iniciar el programa y por tanto si nos interesa que el bloque que nos muestre radare en pantalla sea mas grande desde el momento de iniciar el programa utilizaremos la opción **-e** con la variable y su nuevo valor, con el ejemplo anterior seria:

```
juanjo@cvtucan:/$ radare  -e cfg.bsize=1000 crackme1
```

Otro ejemplo, podemos sustituir la opción **-w** de esta manera, es lo mismo:

```
juanjo@cvtucan:/$ radare  -e file.write=true crackme1
open rw crackme1
```

2.Desde un archivo de configuración. Entre las anteriores opciones que hemos visto había una que esta directamente relacionada a la configuración desde un archivo:

```
-n              do not load ~/.radarerc and ./radarerc
```

Si iniciamos el programa "**radare -n programa**" no se cargarán los archivos de configuración; por

tanto vemos que podemos tener diferentes variables globales configuradas con el valor que nos guste en esos archivos. El que mas se utiliza es el archivo oculto (pues tiene un punto delante del nombre) en nuestra **/home** llamado **.radarerc** (~/.radarerc) pero también podríamos hacer una configuración especifica para el programa que estemos analizando y colocarlo en la misma carpeta, en ese caso seria **./radarerc**.

La primera vez que iniciamos radare se genera un archivo **.radarerc** en nuestra home, donde coloca una serie de variables de configuración por defecto:

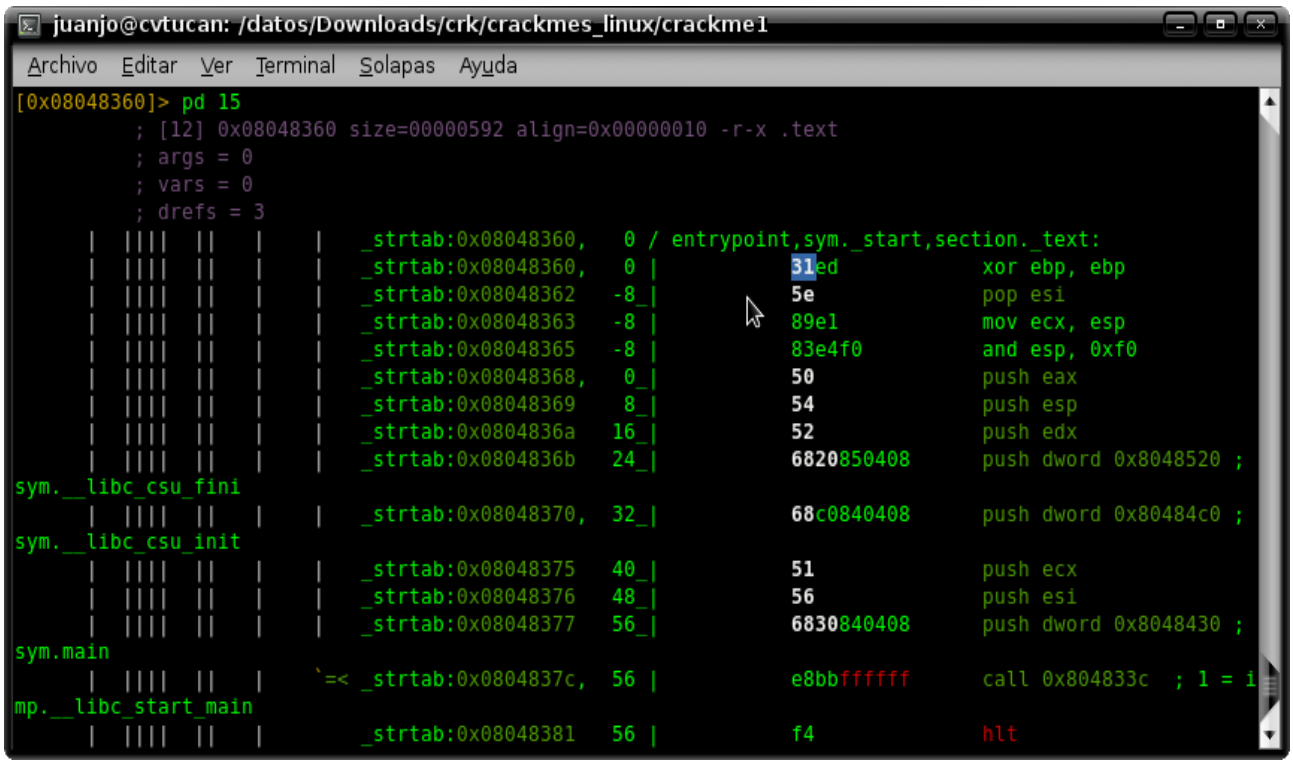
```
; Automatically generated by radare
e file.id=true
e file.flag=true
e file.analyze=true
e scr.color=true
```

Por tanto esta es otra forma de cambiar la configuración del programa, podemos colocar allí la variable y el valor que nos interese, y ese valor se mantendrá siempre, a menos que usemos la opción **-n** al ejecutar radare.

Vamos analizar estas opciones básicas. Si ejecutamos radare con el **crackme1** sin opciones veremos que el programa utilizara la configuración de **.radarerc** por defecto:

```
juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ radare crackme1
open ro crackme1
Message of the day:
Enhace your graphs by increasing the size of the block and graph.depth eval variable
Automagically identifying file...          ;file.id=true ---> Identificación del archivo
[Information]
ELF class:      ELF32
Data enconding:  2's complement, little endian
OS/ABI name:     linux
Machine name:    Intel 80386
Architecture:    intel
File type:       EXEC (Executable file)
Stripped:        No
Static:          No
Base address:    0x08048000
Automagically flagging file...             ;file.flag=true ---> Se crean los flags
7 imports added
34 symbols added
33 sections added
2 strings added
Analyzing program.....o.o.o.o.o...o.o.o.o.     .;file.analyze=true---> Se analiza el programa
functions: 14
data_xrefs: 22
code_xrefs: 11
[0x08048360]> _
```

Y si desensamblamos el código veremos que **src.color** esta activado:



Como es normal, podemos añadir las variables de configuración que queramos en el archivo .radarerc , de manera que serán ejecutadas al iniciarse el programa:

; Automatically generated by radare
e file.id=true
e file.flag=true
e file.analyze=true
e scr.color=true
;Debugger para en el entrypoint
e dbg.bep=entry

Como veis, para este tutorial solo he añadido una opción que le indica al debugger que al cargar el programa debe parar en el entrypoint. En este archivo se puede añadir comentarios siempre precedidos por **";"**, muy al estilo del assambler.

Esta variable en concreto hay que tenerla en cuenta al analizar malware, pues en linux (como en el TLS de windows) el programa puede ejecutar código antes de llegar al entrypoint y podría ser peligroso tener esta opción por defecto.

Manejo de Radare

Vamos a continuar ya con cosas mas interesantes, para ello vamos a utilizar un crackme sencillo, del cual no conozco el autor y que en el readme que acompaña al programa solo pone esto:

"Level 1
Language C
Linux x86/ELF

This is an easy linux crackme. It will only print 'Solved' when certain conditions are met. Enjoy."

Como veis nos tiene que decir "**Solved**" en ciertas condiciones que debemos averiguar.
Como siempre primero vamos a ejecutarlo, para ver si nos da alguna pista mas:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ ./**crackme1**

[1]+ Stopped ./crackme1

Como veis el programa se queda parado y no da mas información; pero como esta en ejecución vamos a curiosear utilizando el debugger para attachearnos (palabro muy nuestro,jeje) al proceso. Primero hay que ver cual es su **PID**

juanjo@cvtucan:/datos/Downloads/crk/$ **ps ax | grep crackme1**
10709 *pts/1 T 0:00* ***./crackme1***
27481 pts/1 S+ 0:00 grep crackme1

Ahora vamos intentar enlazar con el debugger:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare -d 10709**
ptrace_attach: Operación no permitida
debug_init_maps: No existe el fichero o el directorio
rabin: Cannot open file (5402)
error: Cannot open 'dbg://5402'. Use -w to create

Bueno pues parece que el programa guarda algún as en la manga, así que vamos a verlo poco a poco con radare y así iremos viendo su manejo básico.

Primero quitaremos el proceso con la orden **kill,** como ya sabemos su **PID** solo tenemos que ejecutar:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **kill -9 10709**
[3]+ Terminado (killed) ./crackme1

El programa **kill** se encarga de enviar señales a lo programas, con el número **-9** se manda la señal kill (para no variar :-P) por lo que el programa termina y a diferencia de otras señales, este mensaje no puede ser bloqueado.

Ahora vamos a cargar el programa en el debugger y vamos a anlizarlo:

juanjo@cvtucan:/datos/Downloads/crk/crackmes_linux/crackme1$ **radare -d crackme1**
argv = ['crackme1',]
7 imports added
34 symbols added
Program 'crackme1' loaded.
PID = 3049
open debugger ***ro*** *crackme1*
Message of the day:
Trace until system calls with !contsc

Aquí vemos alguna información interesante, el programa tendra el PID nº **3049**, lo hemos abierto como solo lectura y se han encontrado **7 imports,** osea funciones importadas que es semejante a las APIs de windows. Como curiosidad están los mensajes del día, que la verdad a mi me parece muy

buenos, en este caso nos da una orden **“!contsc”**, que hará que el debugger continúe hasta encontrar una llamada al sistema. Si no te interesan, se pueden desactivar cambiando la variable de configuracion **cfg.fortunes** a valor “**false**”, tanto en el mismo programa o para hacerla definitiva se puede colocar en **~/.radarerc.**

Automagically identifying file...
[Information]
ELF class: ELF32
Data enconding: 2's complement, little endian
OS/ABI name: linux
Machine name: Intel 80386
Architecture: intel
File type: EXEC (Executable file)
Stripped: No
Static: No
Base address: 0x08048000

Aquí vemos que es un archivo **ELF32** ejecutable sobre arquitectura **intel** (interesante para el desensamblado) y un dato a apuntar es la **Image Base** que es **8048000**. También es interesante que no haya sido **stripped**; strip es un programa que elimina los símbolos del fichero objeto; en este caso tenemos esos símbolos, que genera el compilador y que nos van a dar mucha información. Respecto a “**Static: No”**, se refiere a que no ha sido compilado con librerías estáticas y que el linkado de las funciones se hace dinámicamente, lo cual es lo normal tanto en linux como en windows.

Automagically flagging file...
7 imports added
34 symbols added
33 sections added
2 strings added

El tema del **flagging** es una de las cualidades de radare, los flags son punteros a zonas del programa de interés, son como marcadores dentro del programa que podemos utilizar como referencias para cualquier comando y es lo primero que miraremos.

Analyzing program.....o.o.o..o...o.o..o.....o..v...o..o.o....
functions: 16
data_xrefs: 22
code_xrefs: 6
0xbff48000 - 0xbff5d000 rw-- 0x00015000 [**stack**]
0xb7f5b000 - 0xb7f5d000 rw-- 0x00002000 **/lib/ld-2.9.so**
0xb7f3f000 * 0xb7f5b000 r-x- 0x0001c000 **/lib/ld-2.9.so**
0xb7f3e000 - 0xb7f3f000 r-x- 0x00001000 [**vdso]**
0x08049000 - 0x0804a000 rw-u 0x00001000
/datos/Downloads/crk/crackmes_linux/crackme1/**crackme1**
0x08048000 - 0x08049000 r-xu 0x00001000
/datos/Downloads/crk/crackmes_linux/crackme1/**crackme1**

Este es el análisis del programa mas a fondo, donde vemos que hay **16** funciones, **22** referencias cruzadas a datos y **6** referencias cruzadas a código y por último nos muestra como se ha **mapeado** el programa en memoria,de cada zona nos dice inicio, final y tamaño. Vemos en que zona esta la pila (**stack**), las zonas de librería del sistema, **ld-2.9.so**, en este caso es una librería de enlace

dinámico, con una zona de escritura **rw** y otra de ejecucion **r-x** y lo mismo del programa, una de datos donde puede escribir el programa **rw-u** (user) y otra de código, donde se ejecuta el programa **r-xu.** He dejado para el final la zona **vdso**, una zona que en algunos kernels ha dado problemas de seguridad, como Denegación de Servicio (DoS) y escalamiento de privilegios . En realidad se llama **Virtual Dynamic Shared Object (VDSO)**, un área de memoria dinámica compartida por todos los procesos, empleada para hacer llamadas rápidas al sistema a través de la instrucción **sysenter** (en linux las syscall se hacían con la interrupción **int 80h**, pero actualmente se hacen con la instrucción sysenter que es mas rápida)

[0xB7F3F810]> _

Y por fin, este es el prompt del programa, que como veis ha pasado de la variable que le indicaba que tenía que ir al entry point,¿? …..son cosas que se irán mejorando.

Primero vamos a ver donde estamos, para ello la orden es "**print**" o "**p**" que nos muestra el código en diferentes formatos siempre a partir de la dirección que señale el prompt(en este caso es **0xB7F3F810**), si vemos la ayuda veremos que hay muchas opciones:

[**0xB7F3F810**]> **p?**
Available formats:
p% : print scrollbar of seek (null)
p= : print line bars for each byte (null)
pa : ascii (null)
pA : ascii printable (null)
pb : binary N bytes
pB : LSB Stego analysis N bytes
pc : C format N bytes
pd : disassembly N opcodes bsize bytes
pD : asm.arch disassembler bsize bytes
pe : double 8 bytes
pF : windows filetime 8 bytes
pf : float 4 bytes
pi : integer 4 bytes
pl : long 4 bytes
pL : long (ll for long long) 4/8 bytes
pm : print memory structure 0xHHHH
pC : comment information string
po : octal dump N bytes
pO : Overview (zoom.type) entire file
pp : cmd.prompt (null)
pr : raw ascii (null)
pR : reference (null)
ps : asm shellcode (null)
pt : unix timestamp 4 bytes
pT : dos timestamp 4 bytes
pu : URL encoding (null)
pU : executes cmd.user (null)
pv : executes cmd.vprompt (null)
p1 : p16: 16 bit hex word 2 bytes
p3 : p32: 32 bit hex dword 4 bytes
p6 : p64: 64 bit quad-word 8 bytes
p7 : print 7bit block as raw 8bit (null)

p8 : p8: 8 bit hex pair N byte
px : hexadecimal byte pairs N byte
pz : ascii null terminated (null)
pZ : wide ascii null end (null)

Para empezar vamos a ver el **d**esensamblado de la zona donde estamos:

```
[0xB7F63810]> pd
|  ld,2.9.so_0:0xb7f63810,  0      eip: 89e0          mov eax, esp
|  ld,2.9.so_0:0xb7f63812   0         e829020000     call 0xb7f63a40  ; 1 = eip+0x230
|  ld,2.9.so_0:0xb7f63817   0         89c7           mov edi, eax
`=< ld,2.9.so_0:0xb7f63819   0       e8e2ffffff     call 0xb7f63800  ; 2 =
maps.ld,2.9.so_0+0x800
      ld,2.9.so_0:0xb7f6381e   0       81c3d6c70100   add ebx, 0x1c7d6
      ld,2.9.so_0:0xb7f63824,  0       8b8300ffffff   mov eax, [ebx+0xffffff00]
      ld,2.9.so_0:0xb7f6382a   -8_      5a            pop edx
      ld,2.9.so_0:0xb7f6382b   -8       8d2484        lea esp, [esp+eax*4]
      ld,2.9.so_0:0xb7f6382e   -8       29c2          sub edx, eax
      ld,2.9.so_0:0xb7f63830,  0_       52            push edx
      ld,2.9.so_0:0xb7f63831   0        8b832c000000  mov eax, [ebx+0x2c]
      ld,2.9.so_0:0xb7f63837   0        8d749408      lea esi, [esp+edx*4+0x8]
      ld,2.9.so_0:0xb7f6383b   0        8d4c2404      lea ecx, [esp+0x4]
      ld,2.9.so_0:0xb7f6383f   0        89e5          mov ebp, esp
      ld,2.9.so_0:0xb7f63841   0        83e4f0        and esp, 0xf0
      ld,2.9.so_0:0xb7f63844,  8_       50            push eax
      ld,2.9.so_0:0xb7f63845   16_      50            push eax
      ld,2.9.so_0:0xb7f63846   24_      55            push ebp
      ld,2.9.so_0:0xb7f63847   32_      56            push esi
      ld,2.9.so_0:0xb7f63848,  32       31ed          xor ebp, ebp
      ld,2.9.so_0:0xb7f6384a   32       e821dc0000    call 0xb7f71470  ; 3 = 0xb7f71470
      ld,2.9.so_0:0xb7f6384f   32       8d934c18ffff  lea edx, [ebx+0xffff184c]
      ld,2.9.so_0:0xb7f63855   32       8b2424        mov esp, [esp]
      ld,2.9.so_0:0xb7f63858,  32       ffe7          jmp edi
      ld,2.9.so_0:0xb7f6385a   32       8db600000000  lea esi, [esi+0x0]
      ld,2.9.so_0:0xb7f63860,  32       e882670100    call 0xb7f79fe7  ; 4 = 0xb7f79fe7
      ld,2.9.so_0:0xb7f63865   32       81c18fc70100  add ecx, 0x1c78f
      ld,2.9.so_0:0xb7f6386b   40_      55            push ebp
      ld,2.9.so_0:0xb7f6386c,  40       89e5          mov ebp, esp
      ld,2.9.so_0:0xb7f6386e   32_      5d            pop ebp
      ld,2.9.so_0:0xb7f6386f   32       8d81d00500    invalid
```

Si por el contrario quieres ver el código de otra zona deberíamos colocarlo de esta manera:

```
[0xB7F63810]> pd @  0x08048360          ;por ejemplo el desensamblado de 0x8048360
```

Siguiendo con el crackme1, vemos que estamos en la librería dinámica **ld.2.9.so**, que en realidad no nos interesa, vamos a continuar hasta el código del programa:

```
[0xB7F63810]> !contu
Stepping until user code...
[0xB7F63810]>_
```

Como veis parece que no se ha movido, esto es debido también a un pequeño error, pues la variable que regula esto es **scr.seek**:

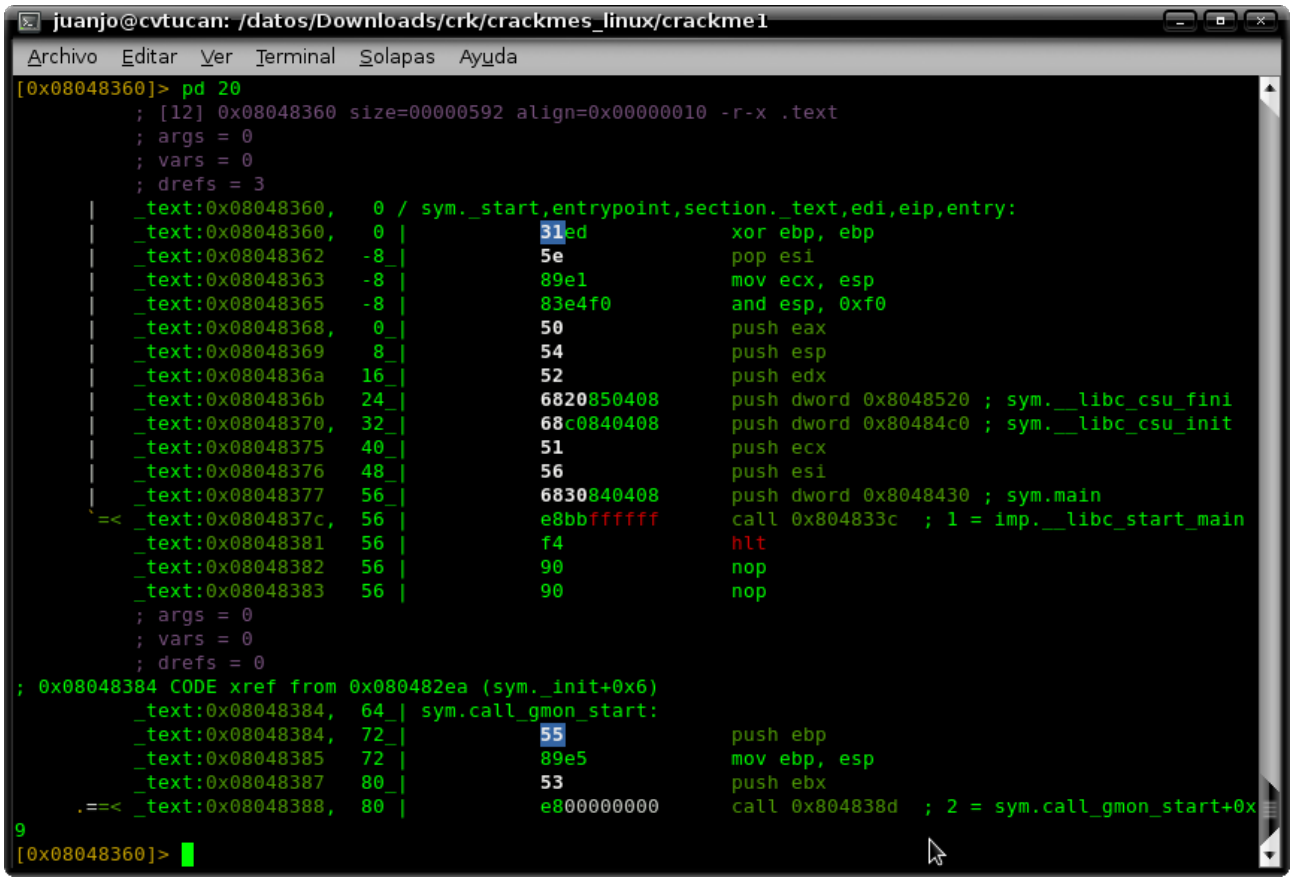
[0xB7F63810]> **e scr.seek**
eip

Como veis debería seguir el registro **eip**, pero el programa sigue en el punto inicial, **0xB7F63810** aunque el debugger haya avanzado; para solucionarlo vamos a ver otra de las ordenes que mas utilizaremos, **"seek"** o **"s"**, que nos permite movernos por todas las áreas de memoria del programa, (para mas ayuda **s?**). Por tanto con poner esto

[0xB7F63810]> **s eip**

Nos llevara al **entrypoint** del programa, que es donde ha parado con la orden **"contu**", por cierto todas las ordenes del debugger deben ir precedidas de un **"!"**

Vamos a ver donde estamos ahora, solo vamos a mostrar 20 lineas de desensamblado:



Asi se ve la orden **"pd 20"**, en este punto hay que explicar un poco algunas lineas interesantes:

_text:0x08048360, 0 / sym._start,entrypoint,section._text,edi,eip,entry:

Vemos que el programa ha definido esta zona como **sym._start** que esta en la sección **_text**, que son parte de los **flags**, que como ya comente son punteros a zonas del programa. Para ver esos flags se utiliza la orden **"f "**, pero como son muchas es preferible primero ver los grupos de marcadores

que tenemos, para ello se utiliza la orden **“fs”**:

```
[0x08048360]> fs
00   imports
01   symbols
02   elf
03   sections
04   strings
05   functions
06   regs
07   maps
```

Vamos comenzar por ver las funciones importadas, para ello las elegimos:

```
[0x08048360]> fs imports
```

Y ahora le decimos que no las enseñe:

```
[0x08048360]> f
0x0804830c   512    imp.getpid
0x0804831c   512    imp.puts
0x0804832c   512    imp.ptrace
0x0804833c   512    imp.__libc_start_main
0x0804834c   512    imp.exit
0x08048000   512    imp._Jv_RegisterClasses
0x08048000   512    imp.__gmon_start__
```

Con esto vemos que utiliza la función **getpid** y **ptrace**, ambas responsables de trucos antidebugger en linux, como ya vimos en la parte 2 y que podéis repasar en este enlace:

http://blog.txipinet.com/2006/10/05/37-tecnicas-anti-debugging-sencillas-para-gnu-linux/

De esta manera ya podemos ver porque no lo pudimos attachear, si el mismo programa utiliza ptrace sobre si mismo, ningún otro programa podrá debuggearlo, como pasaba en windows con protecciones como armadillo.

Ahora podemos ver los símbolos, estos se obtienen de los archivos objetos al compilar y nos va a dar las zonas calientes del programa:

```
[0x08048360]> fs symbols
[0x08048360]> f
0x08048384   512    sym.call_gmon_start
0x080496c8   512    sym.__CTOR_LIST__
0x080496d0   512    sym.__DTOR_LIST__
0x080496d8   512    sym.__JCR_LIST__
0x080495f8   512    sym.p_0
0x08049700   1      sym.completed_1
0x080483b0   1      sym.__do_global_dtors_aux
0x080483f0   1      sym.frame_dummy
0x080496cc   1      sym.__CTOR_END__
0x080496d4   1      sym.__DTOR_END__
0x080495fc   1      sym.__FRAME_END__
```

```
0x080496d8   1     sym.__JCR_END__
0x08048580   1     sym.__do_global_ctors_aux
0x08049600   1     sym._DYNAMIC
0x080485cc   4     sym._fp_hw
0x080495f0   4     sym.__fini_array_end
0x080495f4   4     sym.__dso_handle
0x08048520   80    sym.__libc_csu_fini
0x080482e4   80    sym._init
0x08048360   80    sym._start
0x080495f0   80    sym.__fini_array_start
0x080484c0   84    sym.__libc_csu_init
0x08049700   84    sym.__bss_start
0x08048430   142   sym.main
0x080495f0   142   sym.__init_array_end
0x080495f0   142   sym.data_start
0x080485b0   142   sym._fini
0x08049700   142   sym._edata
0x08048570   142   sym.__i686_get_pc_thunk_bx
0x080496dc   142   sym._GLOBAL_OFFSET_TABLE_
0x08049704   142   sym._end
0x080495f0   142   sym.__init_array_start
0x080485d0   4     sym._IO_stdin_used
0x080495f0   4     sym.__data_start
0x08048360   512   entrypoint
```

En esta caso vamos a centrarnos en una; **sym.main**, que se refiere a la función main en un programa escrito en C, esta función suele ser la inicial que escribe el programador y donde comienza a desarrollar su programa, también vemos el **entrypoint**, inicio del código ejecutable del programa. Por tanto ese flag (al igual que todos aunque esta vez no los utilicemos) son referencias y para radare será igual escribir 0x08048430 que sym.main.

Vamos a echar un vistazo a esa función sin mover el debugger:

```
[0x08048360]> s sym.main  .
[0x08048430]> pd              ;vemos por el prompt que estamos en 0x8048430
; framesize = 1048
; args = 1
; vars = 1
; drefs = 2
; 0x08048430 DATA xref from 0x08048377 (sym._start+0x17)
|  ||  _text:0x08048430,   8_/ sym.main: 55            push ebp
|  ||  _text:0x08048431    0 |        89e5            mov ebp, esp
       ; Stack size +1048
|  ||  _text:0x08048433  1048_|       81ec18040000    sub esp, 0x418
       ; Get var248
|  ||  _text:0x08048439  1048 |       8975f8          mov [ebp-0x8], esi ;
elf.program_headers+0xc4
|  ||  _text:0x0804843c, 1048 |       83e4f0          and esp, 0xf0
|  ||  _text:0x0804843f  1048 |       897dfc          mov [ebp-0x4], edi
|  ||  _text:0x08048442  1048 |       bfd4850408      mov edi, 0x80485d4 ; str.__gmon_start__
|  |`=< _text:0x08048447  1048 |       e8c0feffff      call 0x804830c  ; 1 = imp.getpid
|  |   _text:0x0804844c, 1048 |       89442404        mov [esp+0x4], eax
```

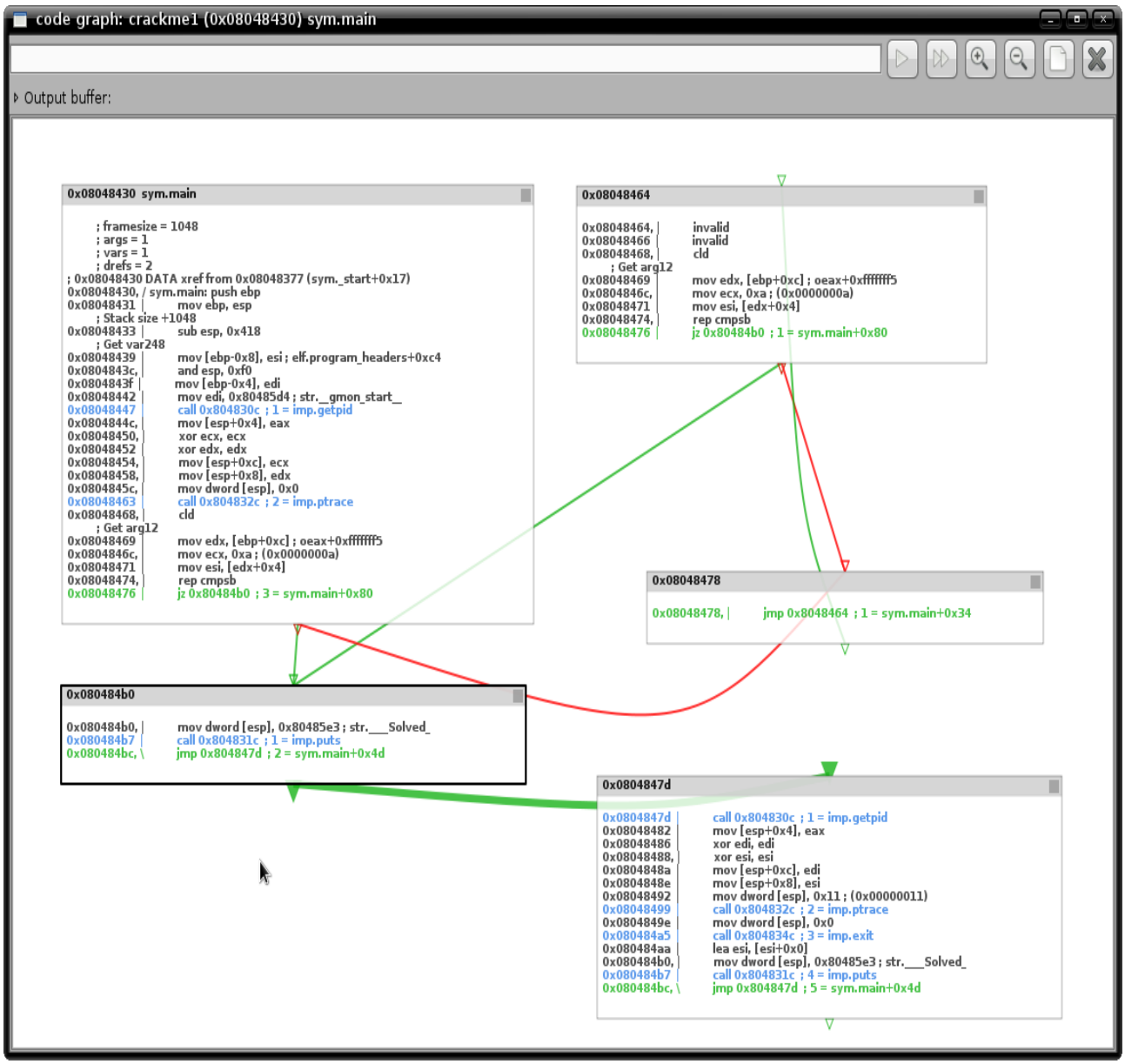
```
| |   _text:0x08048450, 1048 |        31c9          xor ecx, ecx
| |   _text:0x08048452  1048 |        31d2          xor edx, edx
| |   _text:0x08048454, 1048 |        894c240c      mov [esp+0xc], ecx
| |   _text:0x08048458, 1048 |        89542408      mov [esp+0x8], edx
| |   _text:0x0804845c, 1048 |        c7042400000000  mov dword [esp], 0x0
| `==< _text:0x08048463  1048 |        e8c4feffff    call 0x804832c  ; 2 = imp.ptrace
||    _text:0x08048468, 1048 |        fc            cld
      ; Get arg12
||    _text:0x08048469  1048 |        8b550c        mov edx, [ebp+0xc] ; oeax+0xfffffff5
||    _text:0x0804846c, 1048 |        b90a000000    mov ecx, 0xa ; (0x0000000a)
||    _text:0x08048471  1048 |        8b7204        mov esi, [edx+0x4]
||    _text:0x08048474, 1048 |        f3a6          rep cmpsb
||.===< _text:0x08048476  1048 |        7438          jz 0x80484b0  ; 3 = sym.main+0x80
| `====< _text:0x08048478, 1048 |        e9e7ffffff    jmp 0x8048464  ; 4 = sym.main+0x34
 `=====< _text:0x0804847d  1048 |        e88afeffff    call 0x804830c  ; 5 = imp.getpid
  |    _text:0x08048482  1048 |        89442404      mov [esp+0x4], eax
  |    _text:0x08048486  1048 |        31ff          xor edi, edi
  |    _text:0x08048488, 1048 |        31f6          xor esi, esi
  |    _text:0x0804848a  1048 |        897c240c      mov [esp+0xc], edi
  |    _text:0x0804848e  1048 |        89742408      mov [esp+0x8], esi
```

En este caso vemos que **pd** NO muestra toda la zona completa; pues sin valor numérico la orden **p (print)** nos va a mostrar los datos que corresponden al valor de **block size** (que como sabemos podemos cambiar con la variable **cfg.bsize**); en estos casos podemos hacer que **block size** se adapte a un flag en concreto, la orden sería:

```
[0x08048430]> bf  sym.main                ;la f es que va a tomar el tamaño de un flag
block size = 142
[0x08048430]> pd                          ;ahora podemos volver a verla completa
```

De ambas maneras hemos visto que **sym.main** es una zona muy interesante, se ven las dos funciones problemáticas y un salto condicional después de una comparación de cadena. Es un buen momento para hacer un análisis gráfico de esta zona:

```
[0x08048430]> ag
```



Como veis la orden **"a"** es analizar y como todas con **"a?"** podemos ver sus posibilidades, entre ellas está **"ag"** que es el **análisis gráfico**. Si os fijáis, aunque se ve un poco pequeño, el primer cuadro grande es la parte de función **main** que hemos visto con la orden **pd** anterior.
Todos los saltos condicionales están en verde y están al final de cada cuadro pues según sus posibles acciones nos llevarán a diferentes zonas de código, si el salto se produce debemos seguir la linea verde y si la condición del salto no se produce, el programa continuará por la linea roja.

De esta manera vemos que si el salto en **0x08048476** se produce:

_text:0x08048474, 1048 | f3a6 rep cmpsb
*_text:**0x08048476** 1048 | 7438 **jz 0x80484b0** ;*

Nos lleva a la zona **0x0804 84b0** donde carga la string **Solved** y nos la manda al terminal:

*_text:**0x080484b0**, 1048 | c70424e3850408 mov dword [esp], 0x80485e3 ; **str.____Solved_***
*_text:0x080484b7 1048 | e860feffff call 0x804831c ;; 8 = **imp.puts***

De este modo, ya hemos encontrado el camino,ahora vamos a ver que es lo necesario para que ese salto se ejecute.
Para ello vamos a seguir usando el debugger, que se había quedado en el **entrypoint**:

```
[0x08048360]> pd 10                ;vemos 10 lineas de código
; [12] 0x08048360 size=00000592 align=0x00000010 -r-x .text
; args = 0
; vars = 0
; drefs = 3
|  _text:0x08048360,   0 / sym._start,entrypoint,section._text,edi,eip,entry:
|  _text:0x08048360,   0 |        31ed          xor ebp, ebp
|  _text:0x08048362   -8_|        5e            pop esi
|  _text:0x08048363   -8 |        89e1          mov ecx, esp
|  _text:0x08048365   -8 |        83e4f0        and esp, 0xf0
|  _text:0x08048368,   0_|        50            push eax
|  _text:0x08048369    8_|        54            push esp
|  _text:0x0804836a   16_|        52            push edx
|  _text:0x0804836b   24_|        6820850408    push dword 0x8048520 ; sym.__libc_csu_fini
|  _text:0x08048370,  32_|        68c0840408    push dword 0x80484c0 ; sym.__libc_csu_init
|  _text:0x08048375   40_|        51            push ecx
```

Como esta zona no interesa vamos a llevar al debugger hasta **sym.main**, para ello podemos poner un breakpoint en ese flag:

```
[0x08048360]> !bp sym.main
new hw breakpoint 0 at 0x8048430
```

Y ahora continuamos para que pare en ese punto:

```
[0x08048360]> !cont
pre-Breakpoint restored 08048360
HW breakpoint hit!
debug_dispatch_wait: RET = 0 WS(event)=0 INT3_EVENT=2 INT_EVENT=3
CLONE_EVENT=6
=== cont: tid: 27948 event: 0, signal: 5 (SIGTRAP). stop at 0x08048430
```

El programa ha parado , pero el prompt sigue sin seguir el valor de eip, así que debemos hacerlo manualmente:

```
[0x08048360]> s eip
[0x08048430]> _
```

En este punto quiero recordar que la ayuda del programa es la orden **“?”** ; si queremos ver por ejemplo todas las posibilidades de la orden **“!”** que es para ejecutar comandos, se coloca **“!?”.**
Pero además el **“?”** también sirve para evaluar expresiones, como pasa en Olly en el CommandBar:

```
[0x08048430]> ? eip
0x8048430 134513712d 1001102060o 00110000   128,0M
```

Como veis os da el valor de eip en diferentes formatos. También se puede utilizar para hacer cálculos o ver el contenido de una zona de memoria, aquí os pongo unos cuantos ejemplos:

```
[0x08048430]> ? sym.main
0x8048430 134513712d 1001102060o 00110000   128,0M

[0x08048430]> ? [8048430]
0xffffffff 4294967295d 37777777777o 11111111   3,0G

[0x08048430]> ? eip+15
0x804843f 134513727d 1001102077o 00111111   128,0M

[0x08048430]> ? eip+0x15
0x8048445 134513733d 1001102105o 01000101   128,0M
```

Continuamos, habíamos parado en el inicio de la función **main** por el breakpoint, por tanto para poder seguir debemos primero quitar el breakpoint:

```
[0x08048430]> !bp -sym.main
breakpoint at 0x8048430 dropped
```

En zonas tan interesantes, es bueno conocer que radare incorporá una maquina virtual para hacer análisis de código, ésta se ejecuta con la orden **"av";** primero debemos saber los valores que tienen los registros virtuales que vamos a usar, esto se verá con la orden **"avr"**:

```
[0x08048430]> avr
        ; ax=eax&0xffff
        ; al=eax&0xff
        ; ah=eax&0xff00
        ; ah=ah>8
.int32  eax = 0xbfd20ac4
.int16  ax = 0x00000ac4
.int8   al = 0x000000c4
.int8   ah = 0x0000000a
.int32  ebx = 0xb7ee4ff4
.int32  ecx = 0xe2d1bc91
.int32  edx = 0x00000001
.int32  esi = 0x080484c0
.int32  edi = 0x080485d4
.int32  eip = 0x08048332
.int32  esp = 0xbfd20a2c
.int32  ebp = 0xbfd20a38
.bit    zf = 0x00000000
.bit    cf = 0x00000000
```

Y después teniendo en cuenta estos valores podemos ver como cambiarían los registros en una serie de opcodes que queramos valorar sin debuggear, para ello se utiliza la orden **"avx"** seguido con el número de lineas de ensamblador que queramos analizar , el programa las ejecutará virtualmente y nos dirá como se modificarían los registros, vemos un ejemplo en esta zona:

```
[0x08048430]> avx 10
Emulating 10 opcodes
MMU: cached
Importing register values
0x08048430, eip:
```

```
; framesize = 1048
; args = 1
; vars = 1
; drefs = 2
  ; 0x08048430 DATA xref from 0x08048377 (sym._start+0x17)
  0x08048430, / push ebp
  ; esp=esp-4
  ; [esp]=ebp
   ;==> [0xbfd20a38] = bfd20a98  ((esp]))
  ; write bfd20a98 @ 0xbfd20a38
  0x08048431  | ebp = esp
  ; ebp = esp
; Stack size +1048
  0x08048433  | esp -= 0x418
  ; esp -= 0x418
; Get var248
  0x08048439  | [ebp-0x8] = esi ; elf.program_headers+0xc4
  ; [ebp-0x8] = esi
  ;==> [0xbfd20a30] = 80484c0  ((ebp-0x8]))
  ; write 80484c0 @ 0xbfd20a30
  0x0804843c, | esp &= 0xf0
  ; esp &= 0xf0
  0x0804843f  | [ebp-0x4] = edi
  ; [ebp-0x4] = edi
   ;==> [0xbfd20a34] = 8048360  ((ebp-0x4]))
  ; write 8048360 @ 0xbfd20a34
  0x08048442  | edi = 0x80485d4 ; str.__gmon_start__
  ; edi = 0x80485d4
  0x08048447  | call 0x804830c  ; 1 = imp.getpid
  ; esp=esp-4
   ; [esp]=eip+5
   ;==> [0xbfd20a34] = 8048451  ((esp]))
   ; write 8048451 @ 0xbfd20a34
  ; eip=0x804830c
  0x0804830c, imp.getpid:
  ; 0x0804830c CODE xref from 0x0804847d (eip+0x4d)
  ; 0x0804830c CODE xref from 0x08048447 (eip+0x17)
  0x0804830c,   goto dword near [0x80496e8]
  ; goto dword near [0x80496e8]
  Unknown opcode
  0x08048312    push dword 0x0
  ; esp=esp-4
  ; [esp]=dword
  ;==> [0xbfd20a30] = ffffffff  ((esp]))
  ; write ffffffff @ 0xbfd20a30
  [0x08048430]> _
```

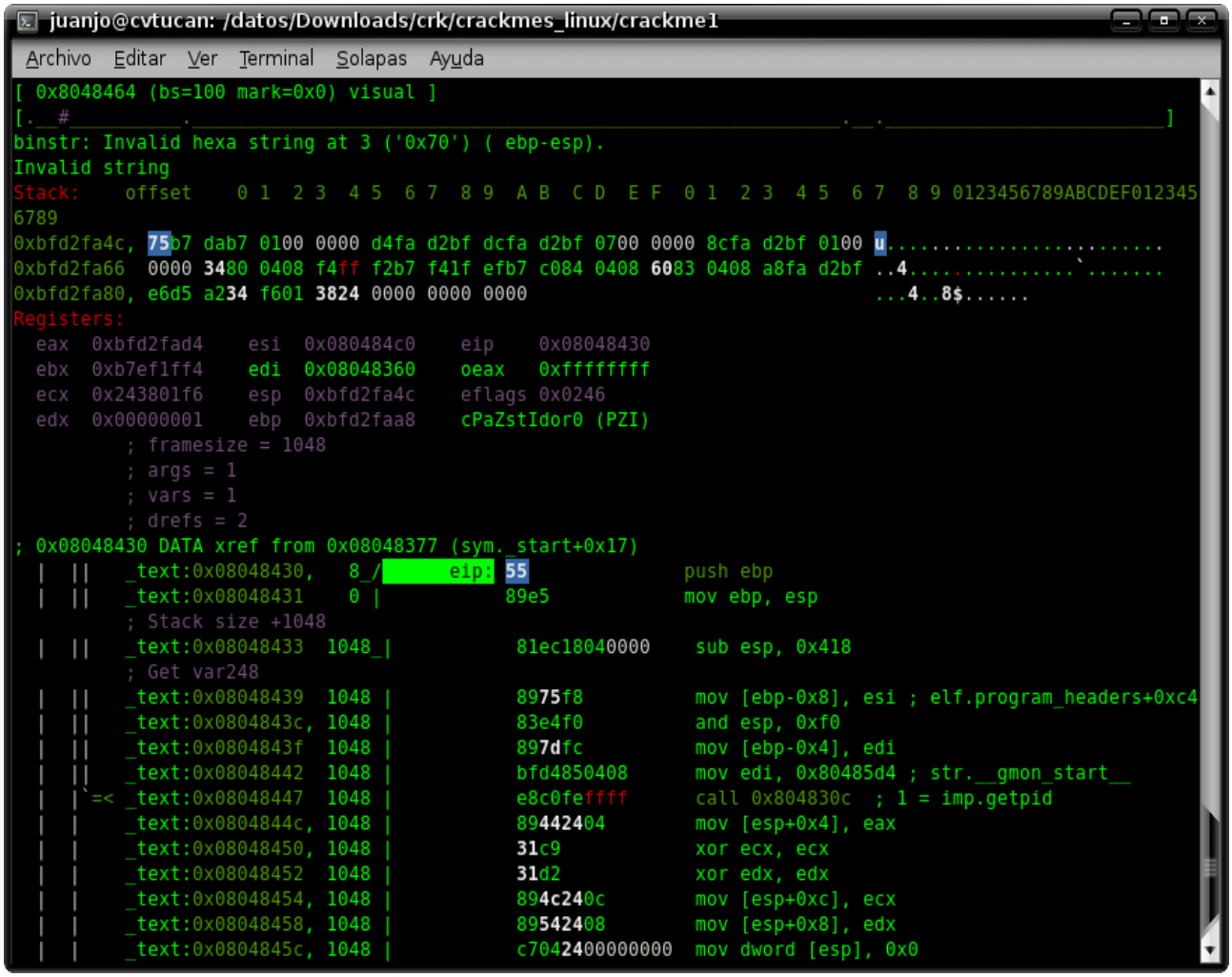
En este caso, donde hay calls, puede que no sea muy útil, pero en zonas de código donde se maneje seriales o muchos datos, creo que puede ser una gran ayuda.

Modo Visual

Ha llegado el momento de debuggear el programa y ver que pasa en directo; para ello lo mas cómodo es utilizar el **modo Visual** de radare. Para entrar en este modo se utiliza la orden **"Visual"** o "**V**" (acordaros que habíamos quitado el breakpoint en el inicio de sym.main, pues hasta que no lo quitas no avanzará el debugger):

[0x08048430]> **Visual**

Y nos aparece de está manera:



Si no os sale así, debéis dar a la tecla **"p"** o **"P"** para avanzar o retroceder en los diferentes formatos de presentación que tiene el modo visual, hay hexadecimal, binario, codigo Ascci, desensamblador y el mas interesante para usar el debugger el que os muestro, que como veis se ve el código desensamblado abajo, marcando el **EIP**,por encima vemos los **registros** y por encima la **pila** del programa parada en **ESP**.

En el modo visual las teclas de manejo varían, lo principal que hay que recordar es la ayuda, que en este caso también es **"?"**, y para ver los comandos del debugger en este modo la orden también es **"!"**, pero os voy a indicar los que mas he usado:

F7 ---> Avanzamos un paso entrando en las funciones (step into), también con **s**

F8 ---> Avanzamos un paso sin entrar en las Calls (step over), también **S.**
F9 ---> Continuar la ejecución
F1 ---> Ayuda
F2 ---> Coloca un breakpoint donde marque seek o si tenemos el cursor cargado, para ello se utiliza la tecla **"c"**, podemos seleccionar la dirección con el cursor y poner un breakpoint con F2.

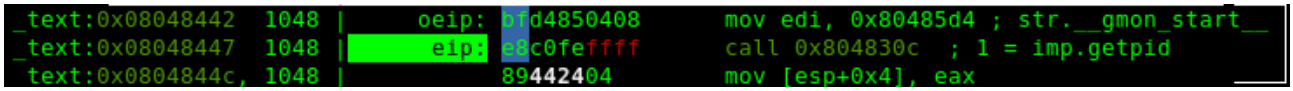
Para moverse en el código o mover el cursor se utilizan estas teclas:

h ---> Ir a la izquierda **j** ---> Ir hacia abajo **k** ---> Ir hacia arriba **l** ---> Ir hacia la derecha

F3 ---> Manera rapida de ejecutar la orden **wx**, sirve para escribir en el punto que estamos , por ejemplo para nopear un salto, **wx 90 90 90**
F4 ---> Continua hasta pasar una repetición o un bucle.
F6 ---> Continuar hasta una syscall (llamada al sistema).
F10 ---> Continuar hasta código del usuario, es lo mismo que la orden en modo normal que ya hemos usado **!contu.**

En este modo, se pueden introducir los comandos normales, para ello debemos pulsar la tecla **":"**, al estilo del editor **vi** ; de esta manera sale el prompt para que pongas la orden, te da el resultado y al pulsar cualquier tecla, vuelves al modo visual.

Bueno, vamos al lío; continuamos la ejecución del **crackme1** con **F8**, pues no queremos entrar en las Calls , y así llegamos a la primera función importada **getpid**:



Como veis el programa siempre nos daré el valor de **eip** y el anterior eip, que el programa llama **oeip**; parece sin sentido, pero tened en cuenta que si hay un salto, puede que nos interese saber de donde venimos.

Ahora pasamos la llamada a **getpid** con **F8** y vemos que la función devuelve un valor en **EAX**:

eax 0x000054c9

Este es el valor hexadecimal, vamos averiguar el valor decimal, para ello damos a **":"** y en el prompt ponemos:

:> **? 0x54c9**
0x54c9 **21705d** 52311o 11001001 21,0K

--press any key--

Ese es el PID del crackme1, pero podemos asegurarnos, para ello volvemos a dar a **":"** y como ya comente podemos enlazar a cualquier orden de **bash** anteponiendo el simbolo **"!"** o en algunas ocasiones hay que hacerlo con una doble admiración **"!!"**como en este caso:

:> **!!ps ax | grep crackme1**
3742 pts/1 S+ 0:00 sh -c ps ax | grep crackme1
3746 pts/1 S+ 0:00 grep crackme1
21704 pts/1 S+ 0:10 radare -d crackme1

21705 pts/1 T+ 0:00 **crackme1**

--press any key--

Como veis tenemos muchas posibilidades sin salir de radare, incluso como todas estas funciones importadas se utilizan en script tenemos información sobre ellas con la orden **man**, podemos usarla directamente o en el modo visual después de darle a **":"**

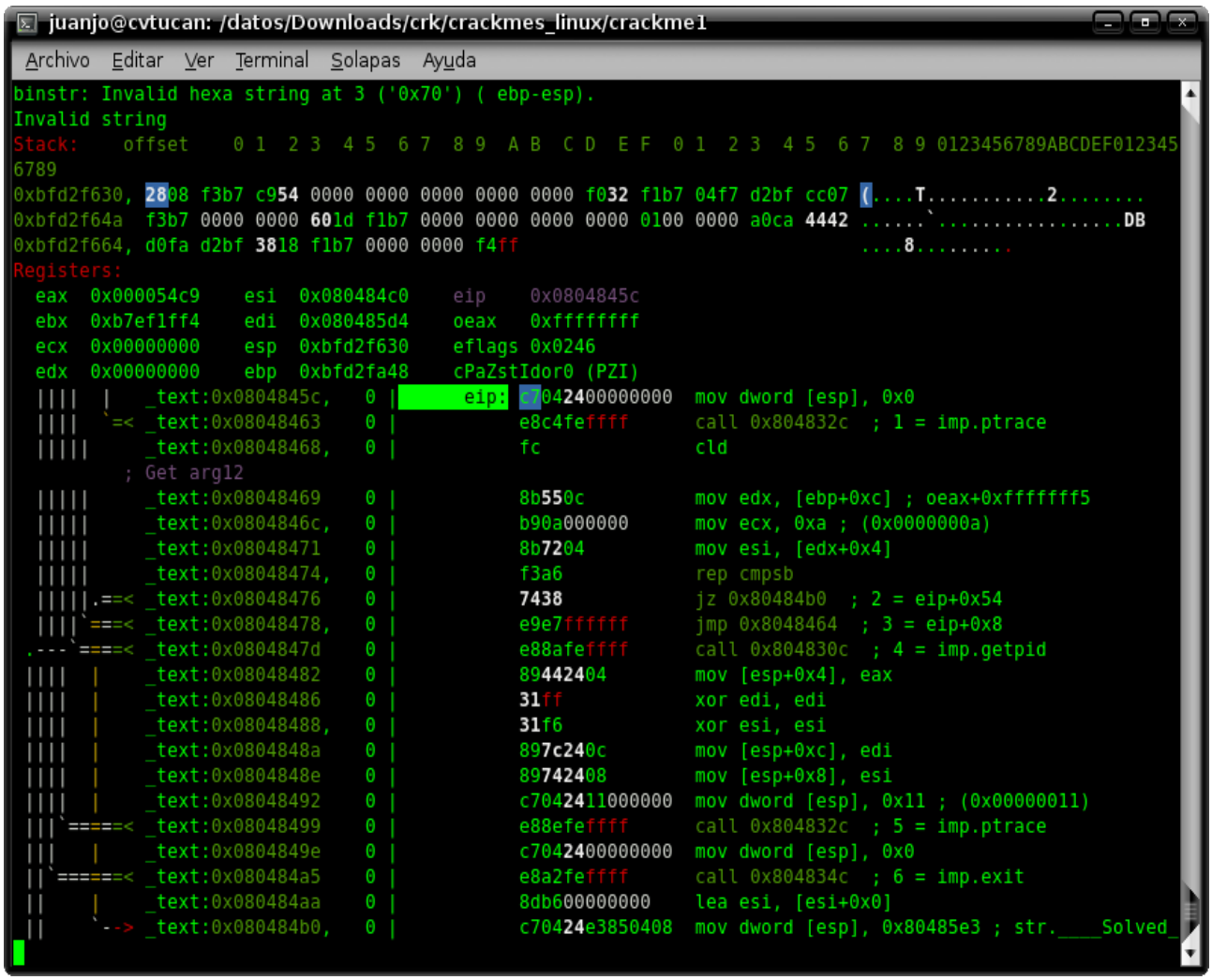
:> **!man getpid**

Para salir de man, debemos presionar **"q"** y después tocar cualquier tecla para volver al modo visual.

Seguimos con **F8** hasta la siguiente función, vemos que terminamos la pantalla que os he mostrado al principio del modo visual,pero el programa no actualiza la ventana al seguir el EIP,(sigue sin seguir el valor de scr.seek =eip), por lo tanto lo tenemos que hacer nosotros, le damos a **":"** y ponemos la orden:

:> **s eip**

De esta manera continuamos en la zona correcta:



Seguimos con **F8** hasta pasar la llamada a **ptrace**, (podemos mirar su manual con **!man ptrace**), y vemos que el valor de EAX es -1.

eax= 0xffffffff

Esto indica que la función ha devuelto un error pues **ptrace** no puede actuar sobre un programa que esta siendo analizada por un debugger. Puede que esto sea lo que debemos cambiar para resolver el crackme, por tanto vamos a cambiar **EAX** con la orden **reg**, por tanto damos a **":"** y ponemos**:**

:> **!reg eax=0**
eax=0

--press any key--

De esta manera cambiamos el valor devuelto por la función y así no nos detecta. Antes de seguir con **F8** vemos una instrucción un poco extraña, **CLD**, que ahora mismo no recuerdo. Pero radare nos puede ayudar con **rsc**, que es el programa para hacer script, y que tiene una pequeña ayuda de assambler:

:> **!rsc adict cld**
cld eflags[DF] = 0

--press any key--

Con la orden **"rsc adict"** vemos que esta orden pone a cero el flags de direcciones **DF,** si vemos en los registros el que corresponde a eflags vemos su valor en dos formatos:

eflags 0x0213
CpAzstIdor0 (CAI)

La forma inferior con letras es la mejor forma de representar los diferentes flags, de esta manera vemos que el flag de direcciones ya esta a cero (**d**), por lo que la orden CLD no va a cambiar nada.

Por cierto, si queremos cambiar algún flag, se utiliza la orden **reg** de esta manera:

:> **!reg eflags=D** **;** de esta manera he activado el flag de direcciones
eflags=D

--press any key--

Debemos saber que los flags se ponen a **0** si es minúscula y sera **1** si es mayúscula, en este caso, ponemos a 1 el flag D y nos queda:

CpAzstIDor0

Os recuerdo los flags que podemos manejar:

C = flag de Acarreo activo
p = flag de paridad
A = flag de acarreo Auxiliar activo
z = flag de cero
s = flag de signo
t = flag de trampa
I = flag de interrupción activo
D = flag de dirección activo
o = flag de overflow (desbordamienteo)

Como esto era solo un ejemplo, lo volvemos a cambiar y continuamos con **F8**, vamos a ver sin mas paradas si con el cambio de EAX después de ptrace es suficiente.

Pero no hay suerte,en este punto, el programa no avanza:

_plt:0x08048474, 0 | f3a6 rep cmpsb
_plt:0x08048476 0 | 7438 jz 0x80484b0 ; 2 = eip+0x48

Si nos salimos del modo visual con **"q"** y queremos continuar, vemos cual es el error:

[0x0804845C]> **!cont**
Segmentation fault!
debug_dispatch_wait: RET = 0 WS(event)=0 INT3_EVENT=2 INT_EVENT=3 CLONE_EVENT=6
*=== cont: tid: 31112 event: 0, signal: 11 (SIGSEGV). stop at **0x08048474***

De esta manera vemos que se ha producido un excepción que no podemos saltar, por lo tanto la solución es que la orden "**rep cmpsb"** se cumpla para que se produzca el salto hacía **0x80484b0** y nos muestre **_Solved.**

La orden **cmps** es una instrucción de cadena (todas ellas acaban en **s** de **string**) en este caso va a comparar una serie de bytes (**cmpsb**) de un origen, que esta señalado por **ESI**, con unos bytes en el destino, que esta señalado por **EDI,** afectando a los flags. La utilidad real de esta orden es cuando la acompañamos con **rep**, que hará repetir la comparación entre esas cadenas de bytes hasta que se cumpla la condición o hasta que se acabe la cadena de bytes, esto último esta marcado por el registo **ECX** (se le llama registro contador) que es el que indica la cantidad de bytes que queremos comparar.
En nuestro caso, los bytes en **EDI** y **ESI** deben ser iguales, para que pase esta comparación y el flag Cero (**Z)** se active, lo que provocará que el salto **jz 0x80484b0** se produzca**.**

Como no podemos continuar, vamos a volver a recargar el crackme, para ello hay una orden que nos evita salirnos de radare:

```
[0x0804845C]> !load                ;curiosamente al recargar si que para en el entrypoint
argv = [ 'crackme1', ]
entry at: 0x8048360
Continue until (0x8048360
) = 0x08048360
pre-Breakpoint restored b7fd8810
HW breakpoint hit!
debug_dispatch_wait: RET = 0 WS(event)=0 INT3_EVENT=2 INT_EVENT=3
CLONE_EVENT=6
=== cont: tid: 32137 event: 0, signal: 5 (SIGTRAP). stop at 0x08048360
pre-Breakpoint restored 08048360
7 imports added
34 symbols added
[0x0804845C]>              ; el programa para donde habíamos llegado con el último "s eip" cuando estabamos en el modo visual ¿?
[0x0804845C]> s eip        ;pero en realidad el debugger esta en el entrypoint
[0x08048360]> _
```

Como esta parte no nos interesa, vamos a poner un breakpoint en la función main:

```
[0x08048360]> !bp sym.main
new hw breakpoint 0 at 0x8048430
```

Por cierto, como pasa en la consola de linux, radare guarda un historial de los comandos: por lo que con dar a las flechas hacia arriba y hacia abajo veremos todas las ordenes que hemos ido introduciendo.

Ahora continuamos para que pare en el breakpoint con la orden **cont:**

```
[0x08048360]> !cont
pre-Breakpoint restored 08048360
HW breakpoint hit!
debug_dispatch_wait: RET = 0 WS(event)=0 INT3_EVENT=2 INT_EVENT=3
CLONE_EVENT=6
```

=== cont: tid: 32137 event: 0, signal: 5 (SIGTRAP). **stop at 0x08048430**
El punto de ruptura ha funcionado, vamos a ir donde señala EIP:

[0x08048360]> **s eip**

Para poder continuar con el debugger en **modo visual,** debemos quitar el breakpoint, vamos a aprovechar el historial de ordenes, con solo darle tres veces a la flecha hacia arriba, nos aparece la orden para poner el bpx, y así sólo debemos colocar un guión para quitarlo:

[0x08048430]> [0x08048430]> **!bp -sym.main**
breakpoint at 0x8048430 dropped

Ahora entramos en modo Visual con la orden **"V"**, y repetimos todo el proceso hasta llegar a la comparación:

_plt:0x08048474, 0 | f3a6 rep cmpsb
_plt:0x08048476 0 | 7438 jz 0x80484b0 ; 2 = eip+0x48

Nos fijamos en los registros:

Registers: eax 0xffffffff **esi 0x00000000** eip 0x08048474
ebx 0xb7fb6ff4 **edi 0x080485d4** oeax 0xffffffff
ecx 0x0000000a esp 0xbf8f2910 eflags 0x0213
edx 0xbf8f2db4 ebp 0xbf8f2d28 CpAzstIdor0 (CAI)

Como veis el valor devuelto por ptrace en **EAX** no lo he modificado esta vez, pues creo que solo era una protección para evitar ser attacheado; la solución pasa por cumplirse la instrucción **rep compsb**, y ni en **EDI** ni en **ESI** hay relación con **EAX**; por lo tanto la clave es que la comparación entre los bytes en **EDI** y **ESI** sea igual, como vemos que ESI vale **0**, al realizarse la comparación provocaba la excepción. Como ya hemos explicado en **ECX** esta al cantidad de bytes que va a comparar, en este caso **0xa (10d)**.

La solución mas sencilla es que tanto EDI como ESI apunten a los mismos bytes, de esta manera estamos seguro que la comparación se cumple. Vamos a utilizar la orden **"reg"** para modificar los registros, como estamos en modo visual para poner una orden damos a **":"** y colocamos la orden:

:> **!reg esi=edi**
esi=edi

--press any key--

De esta manera tanto **EDI** como **ESI** señalan a **0x80485d4**; ahora continuamos con **F8** y vemos que pasamos la comparación y el salto se produce, llevándonos al lugar que buscábamos:

```
Invalid string
Stack:    offset    0 1  2 3  4 5  6 7  8 9  A B  C D  E F  0 1  2 3  4 5  6 7  8 9 0123456789ABCDEF012345
6789
0xbf8f2910, 0000 0000 897d 0000 0000 0000 0000 0000 f082 fdb7 e429 8fbf cc57 .....}..............)...W
0xbf8f292a  ffb7 0000 0000 606d fdb7 0000 0000 0000 0000 0100 0000 a0ca 4442 ......`m..............DB
0xbf8f2944, b02d 8fbf 3868 fdb7 0000 0000 f44f                             .-..8h.......O
Registers:
  eax  0xffffffff    esi  0x080485de    eip    0x080484b0
  ebx  0xb7fb6ff4    edi  0x080485de    oeax   0xffffffff
  ecx  0x00000000    esp  0xbf8f2910    eflags 0x0246
  edx  0xbf8f2db4    ebp  0xbf8f2d28    cPaZstIdor0 (PZI)
   | ||   _text:0x080484b0,   0 |      eip: c70424e3850408  mov dword [esp], 0x80485e3 ; str.____Solved_
   | |`=< _text:0x080484b7    0 |           e860feffff      call 0x804831c  ; 1 = imp.puts
   | `==< _text:0x080484bc,   0 \           ebbf            jmp 0x804847d  ; 2 = oeip+0x7
   |      _text:0x080484be    0             90              nop
   |      _text:0x080484bf    0             90              nop
          ; framesize = 12,0
          ; args = 0
          ; vars = 0
          ; drefs = 0
```

Y si seguimos con F8 nos sale:

[!] Solved! ;me costo encontrarlo, estaba fuera de la zona visual

Conclusión

Bueno como veis el crackme era bastante simple, pero nos ha venido bien para ver algunas cosas de esta gran herramienta, radare, que como también habéis visto esta en fase de desarrollo y tiene varias cosas que pulir.
También he de decir que solo hemos visto una pequeña parte de sus posibilidades, poco a poco iremos haciendo mas pruebas, con otros crackmes mas difíciles y espero contároslo en futuros tutoriales

Si alguien tiene dudas o ve cosas incorrectas, no dudéis en comentarlo en la mejor lista de cracking del universo:

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

Agradecimiento: Sin duda al gran Maestro, Ricardo Narvaja y a todos los CracksLatinoS.