Introducción al Cracking en Linux 2.

Programa:	**Linux CrackMe (Level:3) by cyrex**
Descripción:	Profundizar en el debugger GDB.
Dificultad:	Bajísima.
Herramientas:	Herramientas del sistema y GDB.
Objetivos:	Solucionar el crackme, encontrando serial y archivo llave; y vencer el método antidebugging que implementa.

Cracker: [Juan Jose]	Fecha: 27/03/07

Introducción:

Seguimos disfrutando del cracking en linux; esta vez veremos una nueva herramienta, **objdump**; un volcador de información de los archivos **ELF**; y profundizaremos en el manejo de **gdb**, pues es una gran herramienta y aunque sea complicada, sus posibilidades son muchísimas, como podéis comprobar en este texto en ingles, que ha sido base para este trabajo:

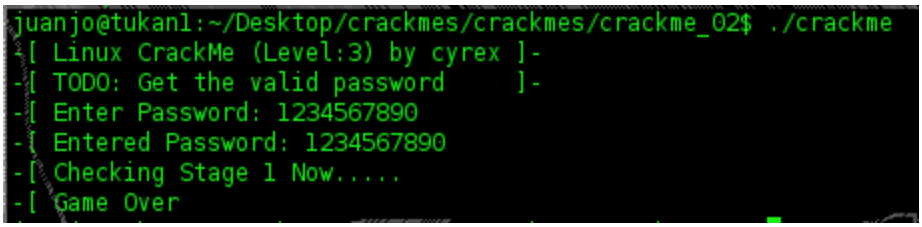
http://www.4shared.com/file/12702789/9894fd8/GNU_DEBUGGING_WITH_GDB.html

Por cierto, de momento dejaremos el **DDD**; no me gustó mucho su funcionamiento y además he comprobado que gdb, sin intermediarios es mas rápido y efectivo; como veremos a lo largo del tute :-)

Respecto a la utilidad de la ingeniería inversa en linux, estoy encontrado bastante información relacionada con las auditorías en servidores linux; donde muchas veces se encuentran archivos desconocidos; los cuales pueden ser virus,gusanos,exploit etc... que hay que analizar; y por tanto se aplicarán en ellos todo lo que estamos estudiando.

Al Atake

Hoy vamos a estudiar el segundo crackme de cyrex; donde podemos ver algo mas de complejidad en el código, con incluso un antidebugger para abrir boca; pero que al igual que el anterior es muy fácil y lo vamos a resolver con varias herramientas. Primero vamos a verlo en acción:



Como veis no es muy diferente al anterior; y sigo sin suerte, jeje, el día que un serial sea 1234567890 me retiro XD.

1º Estudiar el archivo: file

Como ya es rutina, primero hay que ver a que nos enfrentamos:

$ 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, stripped

Lo único diferente al anterior, es que está **stripped**, que como sabemos por el primer tute, es debido a que han utilizado el programa **strip** para eliminar los símbolos del fichero objeto; por lo que vamos a tener menos ayuda para el gdb.

2º Strings.

Como hemos visto hay bastantes cadenas de texto,por lo que este programa siempre tiene que ser una primera elección:

$ strings crackme
/lib/ld-linux.so.2
libc.so.6
printf
__deregister_frame_info
ptrace
strcmp
scanf
exit
fopen
_IO_stdin_used
__libc_start_main
__register_frame_info
__gmon_start__
GLIBC_2.1
GLIBC_2.0
PTRh
QVh

[^_]
Are you trying to Debug me?
-[Linux CrackMe (Level:3) by cyrex]-
-[TODO: Get the valid password]-
-[Enter Password:
-[Entered Password: %s
-[Checking Stage 1 Now.....
7gb5fjf8v4bg8fb34f
-[Stage 1 Cleared
-[Game Over
-[Checking Stage 2 Now....
/tmp/crackme_89nfnjfiefheufeue
-[Bad did you forgot something?
-[You have successfully reversed/cracked/sniffed This Crackme
-[Email me your solution to eth0@list.ru

Con esto nos podemos hacer una idea de como vencer el programa; vemos que hay que usar primero un serial (que es el mismo que en el crackme_1) y después buscará un archivo llave llamado crackme_89nfnjfiefheufeue, que debe estar colocado en la carpeta **/tmp**. Por tanto:

```
$ cd /tmp
$ touch crackme_89nfnjfiefheufeue
```

Y con eso si ejecutamos de nuevo el crackme y metemos los datos correctos:

-[Linux CrackMe (Level:3) by cyrex]-
-[TODO: Get the valid password]-
-[Enter Password: 7gb5fjf8v4bg8fb34f
-[Entered Password: 7gb5fjf8v4bg8fb34f
-[Checking Stage 1 Now.....
-[Stage 1 Cleared
-[Checking Stage 2 Now....
-[You have successfully reversed/cracked/sniffed This Crackme
-[Email me your solution to eth0@list.ru

Todo un éxito, ahora ya podemos ir a la parte interesante ;-)

3º Estudio del archivo elf con objdump.

Este programa suele estar es todas las distribuciones linux y es muy completo; podemos ver muchísimas información de los ejecutable de linux, del tipo **elf** y además nos va a servir como desensamblador. Vamos a ver las opciones que mas nos van a interesar, aunque como siempre quien busque mas información la puede encontrar en **"man objdump"**.
Primero vamos a analizar la cabecera del programa, veremos de esta manera toda la estructura de un ejecutable ELF:

```
$ objdump -x crackme
```

La información que da es muy amplia, cuando ponemos como opción **-x**, es como si pusiéramos las

opciones **-a -f -h -p -r -t**; vamos a ver cada una de sus partes:

$ objdump -f crackme

crackme: formato del fichero elf32-i386
arquitectura: i386, opciones 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
dirección de inicio 0x08048440

Como veis muestra una información muy escasa, pero es una forma rápida de tener el **Entry Point** 8048440

$ objdump -p crackme

crackme: formato del fichero elf32-i386

Encabezado del Programa:
PHDR off 0x00000034 vaddr 0x08048034 paddr 0x08048034 align 22**
filesz 0x000000e0 memsz 0x000000e0 flags r-x
INTERP off 0x00000114 vaddr 0x08048114 paddr 0x08048114 align 20**
filesz 0x00000013 memsz 0x00000013 flags r--
LOAD off 0x00000000 vaddr 0x08048000 paddr 0x08048000 align 212**
filesz 0x000009eb memsz 0x000009eb flags r-x
LOAD off 0x000009ec vaddr 0x080499ec paddr 0x080499ec align 212**
filesz 0x00000120 memsz 0x00000138 flags rw-
DYNAMIC off 0x00000a00 vaddr 0x08049a00 paddr 0x08049a00 align 22**
filesz 0x000000c8 memsz 0x000000c8 flags rw-
NOTE off 0x00000128 vaddr 0x08048128 paddr 0x08048128 align 22**
filesz 0x00000020 memsz 0x00000020 flags r--
STACK off 0x00000000 vaddr 0x00000000 paddr 0x00000000 align 22**
filesz 0x00000000 memsz 0x00000000 flags rwx

Ya iremos estudiando esta estructura mas adelante, pero de momento nos quedamos con lo marcado en rojo, que es la **Image Base** 8048000.

Sección Dinámica:
NEEDED libc.so.6
INIT 0x8048388
FINI 0x8048790
HASH 0x8048148
STRTAB 0x804824c
SYMTAB 0x804818c
STRSZ 0xa3
SYMENT 0x10
DEBUG 0x0
PLTGOT 0x8049ad8
PLTRELSZ 0x48
PLTREL 0x11

JMPREL 0x8048340
REL 0x8048338
RELSZ 0x8
RELENT 0x8
VERNEED 0x8048308
VERNEEDNUM 0x1
VERSYM 0x80482f0

Referencias de Versión:
requerido desde libc.so.6:
0x0d696911 0x00 03 GLIBC_2.1
0x0d696910 0x00 02 GLIBC_2.0

Como veis nos da información del formato del archivo que estudiamos y también podemos ver las librerías necesarias para su ejecución.

$ objdump -h crackme

crackme: formato del fichero elf32-i386

Secciones:
Ind Nombre Tamaño VMA LMA Desp fich Alin
0 .interp 00000013 08048114 08048114 00000114 20**
CONTENTS, ALLOC, LOAD, READONLY, DATA
1 .note.ABI-tag 00000020 08048128 08048128 00000128 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
2 .hash 00000044 08048148 08048148 00000148 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
3 .dynsym 000000c0 0804818c 0804818c 0000018c 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .dynstr 000000a3 0804824c 0804824c 0000024c 20**
CONTENTS, ALLOC, LOAD, READONLY, DATA
5 .gnu.version 00000018 080482f0 080482f0 000002f0 21**
CONTENTS, ALLOC, LOAD, READONLY, DATA
6 .gnu.version_r 00000030 08048308 08048308 00000308 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rel.dyn 00000008 08048338 08048338 00000338 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
8 .rel.plt 00000048 08048340 08048340 00000340 22**
CONTENTS, ALLOC, LOAD, READONLY, DATA
9 .init 00000017 08048388 08048388 00000388 22**
CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt 000000a0 080483a0 080483a0 000003a0 22**
CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text 00000350 08048440 08048440 00000440 24**
CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini 0000001b 08048790 08048790 00000790 22**

```
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .rodata       0000022b  080487c0  080487c0  000007c0  2**5
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 14 .data         00000010  080499ec  080499ec  000009ec  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 15 .eh_frame     00000004  080499fc  080499fc  000009fc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 16 .dynamic      000000c8  08049a00  08049a00  00000a00  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 17 .ctors        00000008  08049ac8  08049ac8  00000ac8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 18 .dtors        00000008  08049ad0  08049ad0  00000ad0  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 19 .got          00000034  08049ad8  08049ad8  00000ad8  2**2
                  CONTENTS, ALLOC, LOAD, DATA
 20 .bss          00000018  08049b0c  08049b0c  00000b0c  2**2
                  ALLOC
 21 .comment      00000124  00000000  00000000  00000b0c  2**0
                  CONTENTS, READONLY
 22 .note         0000003c  00000000  00000000  00000c30  2**0
                  CONTENTS, READONLY
```

Con la opción **-h** veremos las secciones del ejecutable; me llama la atención la cantidad de secciones que tienen los archivos **ELF** a diferencia de los **PE**, que no se suelen ver tantas. Es importante tener en cuenta que muchas secciones son añadidas por el compilador **gcc**; entre ellas habrá que estudiar las secciones **.init** y **.fini** , pues como su nombre indica son siempre ejecutadas al principio y al final del programa, y es el sitio donde el código de virus o malware se inyecta para pasar desapercibido. La sección **init** me ha recordado al TLS de windows, si colocas código allí se ejecutara antes del EP y no nos daremos cuenta. Teniendo en cuenta esto, veremos como es fácil de estudiar con **gdb** o con el código desensamblado que nos da **objdump**:

```
$ objdump -d crackme
```

Con la opción **-d** objdump desensambla las secciones que esperamos tenga código. Si tenemos sospechas de que pueda haber código oculto en otras secciones, debemos usar la opción **-D**.
No os voy a mostrar la salida pues es muy extensa; pero vemos que la opción determinada es el assambler **AT&T .**
*** ATT es el formato de assambler originario de Unix, como comente en el tute anterior se parece al HLA por lo menos en la forma de poner el origen y el destino de las instrucciones al contrario del formato de intel; pero para ser exactos el código que muestra gdb predeterminado es el formato ATT, es un error del primer tute ***
Por tanto, como yo por lo menos estoy muy acostumbrado a leer código en formato **intel**, debemos poner la opción **-M** , que nos da la posibilidad de elegir el tipo de assambler. Como es lógico esta opción no tiene sentido si no acompaña a la **-d** o **-D.**
Por último, nos interesa también que nos de las direcciones virtuales de cada instrucción, para poder parar después donde nos interese con un breakpoint, y para ello debemos colocar la opción **-l**; por lo que la forma mas completa de desensamblar un programa con objdump seria:

```
$ objdump -d -l -M intel  crackme
```

crackme: formato del fichero elf32-i386

Desensamblado de la sección .init:

```
08048388 <.init>:
 8048388:      55                   push   ebp
 8048389:      89 e5                mov    ebp,esp
 804838b:      83 ec 08             sub    esp,0x8
 804838e:      e8 d1 00 00 00       call   8048464 <fopen@plt+0x34>
 8048393:      e8 50 01 00 00       call   80484e8 <fopen@plt+0xb8>
 8048398:      e8 c3 03 00 00       call   8048760 <fopen@plt+0x330>
 804839d:      c9                   leave
 804839e:      c3                   ret
```

Solo he colocado la sección **init.** para que veáis el código en formato **intel**.

$ objdump -T crackme

crackme: formato del fichero elf32-i386

DYNAMIC SYMBOL TABLE:

```
080483b0  w   DF *UND*  00000039  GLIBC_2.0   __register_frame_info
00000000      DF *UND*  0000003f  GLIBC_2.0   strcmp
00000000      DF *UND*  0000003f  GLIBC_2.0   scanf
080483e0  w   DF *UND*  00000026  GLIBC_2.0   __deregister_frame_info
00000000      DF *UND*  00000084  GLIBC_2.0   ptrace
00000000      DF *UND*  000000fa  GLIBC_2.0   __libc_start_main
00000000      DF *UND*  00000039  GLIBC_2.0   printf
00000000      DF *UND*  000000d1  GLIBC_2.0   exit
00000000      DF *UND*  00000036  GLIBC_2.1   fopen
080487c4 g    DO .rodata        00000004  Base        _IO_stdin_used
00000000  w   D  *UND*  00000000              __gmon_start__
```

Con la opción **-T** vemos la tabla de símbolos dinámicos; que como vemos es una tabla de las **APIs** que se cargarán dinámicamente, y que siguiendo con la comparación con windows se parece mucho a nuestra querida **IAT**. He marcado las **APIs** mas interesantes, y que después con el gdb vamos a estudiar.

4º Ltrace y Strace.

Los estudiamos juntos pues ninguno puede completar su trabajo; ambos son parados por el antidebugger; al hacer un análisis dinámico ejecutando el programa son detectados y nos envía un mensaje:

Con **ltrace**: $ ltrace -o ltrace.out ./crackme
Are you trying to Debug me?

Si miramos **ltrace.out** vemos de todas formas donde esta el problema:

__libc_start_main(0x8048520, 1, 0xbfd7ade4, 0x80486a0, 0x8048700 <unfinished ...>
__register_frame_info(0x80499fc, 0x8049b0c, 0xbfd7ad38, 0xb7df0d02, 0x80486a0) = 0
ptrace(0, 0, 1, 0, 0xb7f22750) = -1
printf("Are you trying to Debug me?\n") = 28
__deregister_frame_info(0x80499fc, 0x8049ad8, 0xbfd7ad38, 0x8048509, 0x8049ad8) = 0
+++ exited (status 1) +++

Después de ejecutar **ptrace** nos salta el mensaje. Y si miramos **strace** el resultado es similar:

$ strace -o strace.out ./crackme
Are you trying to Debug me?

Y en el archivo **strace.out** encontramos:

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) = 0xb7f7b000
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) = 0xb7f7a000
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=82342, ...}) = 0
mmap2(NULL, 82342, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb7f65000
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) = 0xb7e34000
mmap2(0xb7f5b000, 28672, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x127) = 0xb7f5b000
mmap2(0xb7f62000, 10396, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0xb7f62000
close(3) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7e33000
mprotect(0xb7f5b000, 20480, PROT_READ) = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0xb7e336c0, limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0xb7f65000, 82342) = 0
ptrace(PTRACE_TRACEME, 0, 0x1, 0) = -1 EPERM (Operation not permitted)
fstat64(1, {st_mode=S_IFCHR|0600, st_rdev=makedev(136, 5), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb7f79000
write(1, "Are you trying to Debug me?\n", 28) = 28

munmap(0xb7f79000, 4096) = 0
exit_group(1) = ?

Se repite el asunto, al ejecutar **ptrace** vemos que es una operación no permitida y devuelve **-1**; lo cual provoca que nos detecte.

Lo mismo pasaba con **gdb**; por lo que empecé a buscar información sobre técnicas anti-debugging en linux; como podéis comprender no hay mucha información sobre el tema, pero encontré una página en español muy interesante:

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

Nuestro caso corresponde al método 7:

"Cuando estamos siendo debuggeados por un debugger en modo usuario (RING-3 en arquitectura IA-32), como GDB o strace/ltrace, sólo es posible llamar a ptrace una vez por proceso. Por lo tanto, si intentamos trazarnos a nosotros mismos y esto provoca un error, es que ya hay otro proceso trazándonos"

Es fácil de entender, ya sabemos por Armadillo y su copymem2; que un programa que está siendo debuggeado no puede ser debuggeado por ningún otro programa; pues aquí es lo mismo si estamos traceando un programa y el mismo se intenta tracear, el error le indica nuestra presencia.
En la misma página nos dan un ejemplo:

```
#include <stdio.h>
#include <sys/ptrace.h>
int main( int argc, char *argv[] )
{
	if( ptrace(PTRACE_TRACEME, 0, 1, 0) < 0 )
	{
		printf("traced!\n");
		return 1;
	}
	printf("OK\n");
	return 0;
}
```

Como veis en la web falta la primera linea, seguramente un error, pues sin el fichero include **stdio.h** no se puede ejecutar la función **printf** y da un error al compilar.
Como el compilador **gcc** esta integrado en linux, para hacer pruebas sólo tenemos que salvar este ejemplo como **ptrace.o** con un editor de texto y ejecutar desde la consola:

```
$ gcc -g ptrace.o -o ptrace
```

De esta manera tenemos un ejecutable con esa protección en segundos, como veis para esto linux es genial ;-)
Vamos a estudiar ese programa para vencer la protección y posteriormente lo aplicaremos al crackme; para todo eso nada mejor que usar el debugger de

GNU y verlo todo en vivo y en directo :-)

5º GDB.

Para empezar vamos a conocer su manejo, por lo tanto es bueno saber que con solo darle a **Intro** repetiremos la orden última que hemos dado, lo cual es cómodo para cuando vayamos traceando, por ejemplo con **nexti**; solo debemos colocarla la primera vez, después con dar **Intro** se ejecutará la misma orden
aunque no salga en el prompt:

```
(gdb) ni      ;nexti
0x08048443 in ?? ()
1: x/i $pc  0x8048443 <fopen@plt+19>:   mov    %esp,%ecx
(gdb)                          ; aquí no se ve la orden pero se ha ejecutado
0x08048445 in ?? ()
1: x/i $pc  0x8048445 <fopen@plt+21>:   and    $0xfffffff0,%esp
```

Otro detalle es que como veis después de ejecutar una orden nos sale la instrucción siguiente que vamos a ejecutar, esto no está configurado por defecto, hay que indicarlo así:

(gdb) display/i $pc ; **display** mostrará lo que le mandemos cada vez que el programa se pare, esto si lo pensáis un poco da mucha posibilidades. Te puede ir mostrando cualquier variable para ver que valores va tomando. Con "**info display"** te muestra la lista de expresiones que tenemos activas; y si queremos quitar alguna lo hacemos con **"delete display nº".** Lo que nos encontramos después de **/** es la forma que quieres que te muestre la información, en este caso **i** como instrucción maquina, **s** como string, **x** como hexadecimal, **d** decimal con signo, **u** como decimal sin signo, **o** como octal, **t** como binario, **a** como dirección, **c** como constante y **f** en coma flotante.

Como se ve en el pequeño desensamblado de las instrucciones anteriores seguimos con el asambler de **ATT**; como nosotros queremos verlo en la forma de **intel** debemos indicarselo:

```
(gdb) set disassembly-flavor intel
```

También nos va a ayudar, que gdb usa cosas similares al **bash**, como es que guarda un historial de ordenes, que se pueden volver a mostrar dándole a las flechas arriba y abajo, y también si colocas una letra y le das al tabulador dos veces nos saldrá todas las ordenes que comience por esa letra. Una vez que encontremos la orden , podemos tener una pequeña ayuda de su funcionamiento con la orden **help**:

```
(gdb) help orden_que_nos_interesa
```

Si os habéis fijado, la mayoría de las ordenes importantes se pueden abreviar con la primera letra, como **next n** ,**step s**, **break b**, **run r** , **continue c** etc.... Para ver todas las posibilidades de **gdb**, ya tenemos el man y el pdf que os comente al inicio, así que yo voy a ir utilizando las opciones que me parecen

mas interesantes mientras vamos ejecutando el programa ptrace que hemos creado antes, de esta forma espero que esto no sea muy pesado.
Vamos a quitar la protección del programa ptrace. Para cargarlo se ejecuta desde una consola:

$ gdb -q ptrace ; la opción **-q** es para no ver el mensaje inicial
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".
(gdb)

De esta forma queda gdb preparado para recibir instrucciones, podemos inicialmente poner un breakpoint:

(gdb) b ptrace ; ya sabeis break
Function "ptrace" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (ptrace) pending.

Como veis la función no esta definida, pero deja el bpx pendiente, vamos a ver si funciona:

(gdb) r ;run
Starting program: /home/juanjo/Desktop/crackmes/crackmes/crackme_02/ptrace
Failed to read a valid object file image from memory.
Breakpoint 2 at 0xb7ec7996
Pending breakpoint "ptrace" resolved

Breakpoint 2, 0xb7ec7996 in ptrace () from /lib/tls/i686/cmov/libc.so.6
(gdb)

Ha funcionado, estamos en la función **ptrace** en la librería **libc.so.6**, vamos a ver el código:

(gdb) set disassembly-flavor intel ; se vera el assambler de intel.
(gdb) disas ; se vera el código por donde va el programa.
Dump of assembler code for function ptrace:
0xb7ec7990 <ptrace+0>: push ebp
0xb7ec7991 <ptrace+1>: mov ebp,esp
0xb7ec7993 <ptrace+3>: sub esp,0x14
0xb7ec7996 <ptrace+6>: mov DWORD PTR [ebp-4],edi
0xb7ec7999 <ptrace+9>: mov edi,DWORD PTR [ebp+8]

.....continua bastante, pero como no nos interesa, al llenar la pantalla nos dará la opción de continuar mostrando mas lineas con **intro** o salir con **q+intro.**
La orden **disas** sola, a veces no funciona en los programas sin símbolos, por lo que hay que poner las direcciones entre la cuales quieres desensamblar:

(gdb) disas 0x8048536 0x8048590

Como estamos en la función y lo que nos interesa es la salida, vamos a

continuar con **nexti ni**, similar al F8 de Olly, de esta forma se va a ejecutar la siguiente instrucción pero si llegamos a una función no entraremos en ella (si quisiéramos entrar deberíamos de utilizar **stepi si**, como el F7 de Olly):

(gdb) ni
0xb7ec7991 in ptrace () from /lib/tls/i686/cmov/libc.so.6
(gdb)
0xb7ec7993 in ptrace () from /lib/tls/i686/cmov/libc.so.6
(gdb)

De esta forma continuamos hasta salir de la función:

(gdb)
0x080483c9 in main () at ptrace.c:5
5 if(ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)

Ahora va a ejecutar la primera instrucción del programa. Si vemos los registros:

(gdb) info registers ; si quieres ver todos los registros **“info all-registers”**
eax 0xffffffff -1
ecx 0x0 0
edx 0x1 1
ebx 0xb7f2fff4 -1208811532
esp 0xbfb762c0 0xbfb762c0
ebp 0xbfb762d8 0xbfb762d8
esi 0x0 0
edi 0xb7f61cc0 -1208607552
eip 0x80483c9 0x80483c9 <main+53>
eflags 0x213 [CF AF IF]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51

Ya nos han pillado, si vemos el código con **disas**:

0x080483c4 <main+48>: call 0x80482b4 <ptrace@plt>
0x080483c9 <main+53>: test eax,eax
0x080483cb <main+55>: jns 0x80483e2 <main+78>

Si **eax** no es cero el salto se produce y nos echa fuera; por tanto lo mas fácil es que **eax** sea igual a **0**,para ello en gdb se puede hacer:

(gdb) print $eax ; print nos muestra el valor de eax
$1 = -1
(gdb) set $eax=0 ; con set le damos el nuevo valor

(gdb) info registers ; lo comprobamos
eax 0x0 0
ecx 0x0 0
edx 0x1 1
.....

Ahora ya hemos engañado al programa, podemos seguir pues el se creerá que la función **ptrace** ha funcionado correctamente. Para continuar usamos siempre **continue c**:

(gdb) c
Continuing.
OK

Program exited normally.
(gdb)

Conseguido!!!; aunque el programa ha terminado seguimos en gdb, si queremos se puede volver a cargar el programa con **exec**:

(gdb) exec ptrace ; también se podría haber cargado cualquier otro.
Si queremos durante la ejecución del programa reiniciar el proceso, primero debemos eliminar el proceso actual con "**kill**" y después volver a cargar el programa con **exec**.

Si por le contrario quieres salir de **gdb**, se usa la orden **quit**.

Vamos a seguir jugando con la protección pero en el crackme; de forma que veamos mas posibilidades de gdb, en este caso en un programa sin símbolos y que nos va a dar menos información. Ejecutamos en la consola:

$ gdb -q crackme
This GDB was configured as "i486-linux-gnu"...(no debugging symbols found)
Using host libthread_db library "/lib/tls/i686/cmov/libthread_db.so.1".

(gdb)

Primero vamos a poner un bpx en el **EP**:

(gdb) b *0x8048440 ; no olvidar el asterisco y el 0x en las direcciones hexadecimales.
Breakpoint 1 at 0x8048440
(gdb) r ; lo lanzamos con run
Starting program: /home/juanjo/Desktop/crackmes/crackmes/crackme_02/crackme
Failed to read a valid object file image from memory.
(no debugging symbols found)
(no debugging symbols found)

Breakpoint 1, 0x08048440 in ?? () ; ha parado en el breakpoint.

Vamos a ver el código:

(gdb) disas
No function contains program counter for selected frame.

Esto fue lo que os comente, hay que indicarlo con mas exactitud:

(gdb) disas 0x8048440 0x8048500
Dump of assembler code from 0x8048440 to 0x8048500:
0x08048440 <fopen@plt+16>: xor %ebp,%ebp
0x08048442 <fopen@plt+18>: pop %esi
0x08048443 <fopen@plt+19>: mov %esp,%ecx
0x08048445 <fopen@plt+21>: and $0xfffffff0,%esp
0x08048448 <fopen@plt+24>: push %eax

Asambler de ATT, lo cambiamos por el de Intel:

(gdb) set disassembly-flavor intel

Ahora si, con darle a la flecha arriba dos veces nos vuelve a salir la orden "**disas 0x8048440 0x8048500**" y ahora si lo vemos todo bien.

Vamos a continuar con **nexti**; pero como ya sabemos, para que nos muestre la instrucción siguiente que va a ejecutar, ponemos:

(gdb) display/i $pc

Y seguimos con **nexti** para ver como se ejecuta el programa, si en algún momento queremos ver la pila, podemos usar esta orden:

(gdb) x/5 $esp
0xbf8ca8a0: 0x08048700 0xb7f7fc40 0xbf8ca8ac 0xb7f8a4e4
0xbf8ca8b0: 0x00000001

Como vemos, con la orden **x** podemos ver cualquier zona de la memoria, su estructura es:

x/nfu dirección

Donde **n**=numero de unidades que va a mostrar.
f= el formato que utiliza para mostrar los datos, por sistema es hexadecimal---> **x**, decimal con signo-->**d**, decimal sin signo--->**u**, octal -->**o,** binario --->**t**, como direcciones -->**a**, constantes -->**c,** coma flotante ---> **f**, se ven las strings -->**s**, instrucciones maquinas -->**i.**
u= el tamaño de la unidad como byte -->**b,** dos bytes -->**h** (halfwords) cuatro bytes --> **w** (words, es el formato predeterminado), ocho bytes --> **g**

Ej. (gdb) x/8dh 0xbf8ca8a0

0xbf8ca8a0: -30976 2052 -960 -18441 -22356 -16500 -23324 -18440

(gdb) x/8ih 0xbf8ca8a0

0xbf8ca8a0: add BYTE PTR [edi-0x3bff7fc],al
0xbf8ca8a6: div DWORD PTR [edi-0x40735754]
0xbf8ca8ac: in al,0xa4
0xbf8ca8ae: clc
0xbf8ca8af: mov bh,0x1
0xbf8ca8b1: add BYTE PTR [eax],al
0xbf8ca8b3: add BYTE PTR [ecx],ah
0xbf8ca8b5: leave

Vamos a seguir ahora, resolviendo la protección antidebugger, de una forma mas elegante, para ello ponemos un bpx en la llamada **ptrace**:

(gdb) b ptrace
Breakpoint 2 at 0xb7eef996
(gdb) c ;continuamos
Continuing.
Breakpoint 2, 0xb7eef996 in ptrace () from /lib/tls/i686/cmov/libc.so.6
1: x/i $pc 0xb7eef996 <ptrace+6>: mov DWORD PTR [ebp-4],edi

Ahí hemos parado y nos muestra la primera linea dentro de la función. Ahora es un buen momento para probar la función **backtrace bt**:

(gdb) bt
#0 0xb7eef996 in ptrace () from /lib/tls/i686/cmov/libc.so.6
#1 0x08048536 in ?? ()
#2 0x00000000 in ?? ()

Como vemos esta orden es similar al **Call Stack** de Olly, donde vemos todas las llamadas que hay en la pila y que nos han llevado hasta aquí.
Esta formada por varias lineas, que se llaman **frame** y están numeradas, las última es la marcada con **#0**, en este caso la **#1** nos indica de que parte del programa venimos, lo cual nos podría interesar. Si colocamos la orden **frame** sin argumentos nos saldrá la función donde estamos:

(gdb) frame
#0 0xb7eef996 in ptrace () from /lib/tls/i686/cmov/libc.so.6

Ahora como hicimos en el programa ptrace, podríamos trasear hasta la final de la función y el valor que retorne, que será **-1** cambiarlo por **0**, pero gdb tiene una forma mucho mas cómoda, la orden **return**, que evitará que se ejecute la Call, equilibrará la pila y podemos darle como argumento el valor que queramos que devuelva la función; es perfecta para estas ocasiones:

(gdb) return 0
Make selected stack frame return now? (y or n) y
#0 0x08048536 in ?? ()

Nos pregunta algo, se le responde que yes y listo, ya esta solucionada la llamada a **ptrace**. Lo comprobamos con la orden **print p**:

```
(gdb) p $eax
$5 = 0
```

Todo correcto. Vamos a ver donde estamos:

```
(gdb) disas 0x8048536 0x8048590
Dump of assembler code from 0x8048536 to 0x8048590:
0x08048536 <fopen@plt+262>:     add    esp,0x10
0x08048539 <fopen@plt+265>:     mov    eax,eax
0x0804853b <fopen@plt+267>:     test   eax,eax
0x0804853d <fopen@plt+269>:     jge    0x8048560 <fopen@plt+304>
```

Si seguimos con **nexti** hasta el salto:

```
(gdb)
0x0804853d in ?? ()
1: x/i $pc  0x804853d <fopen@plt+269>:  jge    0x8048560 <fopen@plt+304>
```

En este caso, hemos cambiado el valor de **eax** para que este salto se produzca; pero si no lo hubiésemos hecho, podríamos todavía modificar la ejecución del programa, obligándole a saltar con la orden **jump**:

```
(gdb) jump *0x8048560   ;hay que poner un asterisco
Continuing at 0x8048560.
-[ Linux CrackMe (Level:3) by cyrex ]-
-[ TODO: Get the valid password     ]-
-[ Enter Password:
```

Como veis se ha ejecutado el programa, aunque no hubiésemos cambiado el valor de eax. Otra forma de cambiar la dirección del programa es con la variable **$pc**, es la siguiente dirección a ejecutar, por lo que con la orden **set** podemos darle el valor que queramos:

```
(gdb) set $pc= 0x8048560
```

También podría ser con **$eip** de toda la vida. :-) Por cierto, todas las variables, los registros lo son, se deben poner colocándole delante el signo **$**.

Ahora vamos a reiniciar el programa con **kill** y **exec crackme** y vamos a resolver el crackme dentro de **gdb**, ya que la protección sabemos como pasarla. En realidad esto no es mas que una excusa para seguir estudiando los comandos mas importantes:

```
(gdb) info file
Symbols from "/home/juanjo/Desktop/crackmes/crackmes/crackme_02/crackme".
Local exec file:
```

`/home/juanjo/Desktop/crackmes/crackmes/crackme_02/crackme', file type elf32-i386.
Entry point: 0x8048440
0x08048114 - 0x08048127 is .interp
0x08048128 - 0x08048148 is .note.ABI-tag
0x08048148 - 0x0804818c is .hash
0x0804818c - 0x0804824c is .dynsym
0x0804824c - 0x080482ef is .dynstr
0x080482f0 - 0x08048308 is .gnu.version
0x08048308 - 0x08048338 is .gnu.version_r
0x08048338 - 0x08048340 is .rel.dyn
0x08048340 - 0x08048388 is .rel.plt
0x08048388 - 0x0804839f is .init
0x080483a0 - 0x08048440 is .plt
0x08048440 - 0x08048790 is .text
0x08048790 - 0x080487ab is .fini
0x080487c0 - 0x080489eb is .rodata
0x080499ec - 0x080499fc is .data
0x080499fc - 0x08049a00 is .eh_frame
0x08049a00 - 0x08049ac8 is .dynamic
0x08049ac8 - 0x08049ad0 is .ctors
0x08049ad0 - 0x08049ad8 is .dtors
0x08049ad8 - 0x08049b0c is .got
0x08049b0c - 0x08049b24 is .bss

La orden **info** teniendo como argumentos algunos comandos pueden dar mucha información interesante, en este caso con **file** nos da información del archivo que estamos ejecutando,aprovechamos para tomar nota del EP. Si tenéis curiosidad poned **info set**, veréis gran parte de la configuración de GDB. Otro info útil es **info program**; este solo te dice en que posición tenemos el programa:

(gdb) info program
The program being debugged is not being run.

Aunque en inglés vemos que aunque esta cargado no lo hemos ejecutado todavía. Vamos a colocar un breakpoint en el EP:

(gdb) tb *0x8048440
Breakpoint 6 at 0x8048440

En este caso he utilizado **tbreak** que es un breakpoint pero solo se usa una vez, como es para el **EP** con eso tenemos suficiente. Para saber que breakpoint tenemos puestos se usa la orden:

(gdb) info break

Num	Type	Disp	Enb	Address	What
6	breakpoint	del	y	0x08048440	<fopen@plt+16>

Como veis es el bpx número 6, ya que normalmente los bpx y hbpx, **gdb** los guarda mientras nos salgamos de la sesión con **quit**. Para borrar los bpx que faltaban he usado la orden:

(gdb) delete break nº_orden ; si colocas un **delete break**, te los borrara todos.

Seguimos, damos a **run r** y el programa para en el EP. Si miramos la situación del programa vemos que el bpx se ha eliminado una vez se ha utilizado:

(gdb) info program
Using the running image of child process 30288.
Program stopped at 0x8048440.
It stopped at a breakpoint that has since been deleted.

Ahora debemos parar en la llamada a ptrace, para ello vamos a utilizar un **hardware breakpoint**:

(gdb) hb ptrace
Hardware assisted breakpoint 8 at 0xb7eee996
(gdb) c
Continuing.

Breakpoint 8, 0xb7eee996 in ptrace () from /lib/tls/i686/cmov/libc.so.6
(gdb)

El hardware breakpoint ha funcionado, se usa igual que los bpx, se puede poner en una dirección o en una función, y también funciona el **thb**, que son un hardware bpx de un solo uso. Una vez parado en la función **ptrace**, eliminamos la protección:

(gdb) return 0
Make selected stack frame return now? (y or n) y
#0 0x08048536 in ?? ()

Ahora vamos a buscar la API **strcmp** para encontrar el serial:

(gdb) b strcmp
Breakpoint 3 at 0xb7f00e00
(gdb) c
Continuing.
-[Linux CrackMe (Level:3) by cyrex]-
-[TODO: Get the valid password]-
-[Enter Password: 123456789
-[Entered Password: 123456789
-[Checking Stage 1 Now.....

Breakpoint 3, 0xb7f00e00 in strcmp () from /lib/tls/i686/cmov/libc.so.6

Seguimos con **nexti** hasta la comparación como vimos en el anterior tute:

(gdb) ni
0xb7f00e0a in strcmp () from /lib/tls/i686/cmov/libc.so.6
1: x/i $pc 0xb7f00e0a <strcmp+10>: cmp al,BYTE PTR [edx] ;esto es lo que ese va a ejecutar.

Ahora es el momento de ver los valores de los registros:

(gdb) p/x $eax ; pongo x pues quiero verlo en hexadecimal.
$1 = 0xbfe6c931

En al tenemos nuestro primer valor del serial falso en **al,** 31 (1 en Ascii) y en la dirección indicada por **edx**:

(gdb) x/s ($edx) ; estoy buscando una string -->s
0x80488b6 <_IO_stdin_used+242>: "7gb5fjf8v4bg8fb34f"

Si no queremos ver el valor de una variable o dirección, sino usarla como puntero, para ver lo que señala se debe colocar entre paréntesis, como en este caso; en assambler son corchetes.

Continuamos con **ni** hasta el final de la función **strcmp**, como los seriales no son iguales, el valor que retorna es **-1**, vamos a cambiarlo para terminar de solucionar el crackme sin reiniciar:

(gdb) print $eax
$2 = -1
(gdb) set $eax=0
(gdb) print $eax
$3 = 0

Ahora seguimos buscando el archivo llave, que como hemos visto lo intenta abrir con la llamada al sistema **open** (con fopen no me funcionó):

(gdb) b open
Breakpoint 5 at 0xb7e764b0
(gdb) c
Continuing.
-[Stage 1 Cleared
-[Checking Stage 2 Now....

Breakpoint 5, 0xb7e764b0 in open () from /lib/tls/i686/cmov/libc.so.6
1: x/i $pc 0xb7e764b0 <open>: cmp DWORD PTR gs:0xc,0x0(gdb) b open

Si miramos la pila, veremos los argumentos que le ha pasado el programa a **open**:

(gdb) x/4 $esp

0xbff0b184: "\202\211\004\b"
0xbff0b18d: ""
0xbff0b18e: ""
0xbff0b18f: ""

No es lo que buscamos, vamos a ver los registros:

(gdb) info registers
eax 0x8048920 134514976
ecx 0x0 0
edx 0x1b6 438
ebx 0xb7ee6ff4 -1209110540
esp 0xbff0b184 0xbff0b184
ebp 0xbff0b1ac 0xbff0b1ac
esi 0x804a008 134520840
edi 0x8 8
eip 0xb7e764b0 0xb7e764b0 <open>
eflags 0x246 [PF ZF IF]
cs 0x73 115
ss 0x7b 123
ds 0x7b 123
es 0x7b 123
fs 0x0 0
gs 0x33 51

Es curioso que el valor de eax esté en el ejecutable, seguramente es un puntero a una string interesante ;-)

(gdb) x/s ($eax)
0x8048920 <_IO_stdin_used+348>: "/tmp/crackme_89nfnjfiefheufeue"

Resuelto; en este caso como el archivo seguía colocado en tmp, el resultado de open es correcto y el crackme termina bien:

gdb) c
Continuing.
-[You have successfully reversed/cracked/sniffed This Crackme
-[Email me your solution to eth0@list.ru

Program exited normally.

Mas cosas útiles de gdb.

Si miráis el documento que os mostré al inicio, las posibilidades son muchas y algunas no acabo de entenderlas bien; pero he seleccionado algunas opciones que nos puede interesar:

(gdb) set logging on ; salva todos los comandos de gdb en un archivo.

(gdb) set write on ; con esta orden antes de ejecutar el programa nos permitirá poder parchear el ejecutable. ¿Cómo hacerlo....? es algo que no he averiguado todavía, si alguien tiene idea pues nos vendrá muy bien; por ahora estoy utilizando **Biew**,un editor hexadecimal bastante completo.

Watchpoints ; son un tipo de breakpoint un tanto especiales, se podría comparar con un bpx condicional. La orden es "**watch expresión**" de forma que si la expresión se cumple el programa se parará. Os quería poner algún ejemplo en el programa, pero no me ha llegado a funcionar bien, además de que tarda mucho, no he conseguido que me pare. En este caso la orden fue:

(gdb) watch $eax=-1 ; como la función **ptrace** nos devuelve ese valor, esperaba que el programa parase y entonces poder cambiarlo; pero no lo he conseguido, habrá que seguir haciendo pruebas porque tiene posibilidades. Con **info break** veremos tanto los bpx como los hardware bpx y también los watchpoints y catchpoints que veremos a continuación. Todos se pueden eliminar con **delete break nº**

Catchpoint; otro tipo de breakpoint; en este caso el programa parará cuando ocurra el evento que se pone como argumento:

(gdb) catch evento

Como eventos mas interesantes, aunque no los he probado:

throw The throwing of a C++ exception ; creo entender que parará cuando salte una excepción de C++

catch The catching of a C++ exception; creo que parará cuando capture una excepción de C++ ¿? Queda pendiente su utilización.

Hay otros eventos muy interesantes, como **load libname**; que obliga a parar el programa cuando se carga una librería dinámica; pero según pone sólo se puede utilizar en HP-UX.

Tracepoints; esta orden no tiene que ver con los bpx; es una forma de obtener información de un punto concreto del programa sin necesidad de pararlo. Es muy parecido al "trace into" de Olly, aunque localizado en un punto. Para colocarlo se usa la orden **trace tr**:

(gdb) trace argumento ; como argumentos se puede poner tanto direcciones como funciones, al igual que los bpx.

Con **info trace**, veremos los tracepoint que tenemos colocados, para borrarlos, usamos **delete tracepoint nº**.

Podemos ejecutarlos cuando queramos, pues como es lógico, tenerlos activos siempre hará al programa ir muy lento:

(gdb) enable tracepoint nº ; si no colocamos número se activarán todos los tracepoint.

(gdb) disable tracepoint nº; igual que el anterior.

Una vez colocado el **tracepoint** tenemos que decirle que tiene que hacer, para ello usamos la orden **actions**. Aquí podemos ir dándole ordenes de lo que tiene que hacer cada vez que llegue al punto que nos interesa; lo mas utilizado es **collect**; que recogerá información del argumento que le demos, ejemplos, **$regs**, recoge el valor de los registros, **$args**, todos los argumentos de las funciones, **$locals**, las variables locales. Cuando hayamos terminado de darle ordenes lo indicamos con **end**.

Por último, cuando queramos que el tracepoint empiece a funcionar, utilizaremos **tstart** y para parar **tstop**.

Para ver lo que hemos ido recolectando se usa las ordenes **tfind**, para encontrar algo concreto y **tdump**, nos vuelca todo lo que hemos obtenido con el tracepoint que ha actuado el último.

Vamos a ver todo esto con un ejemplo:

```
(gdb) trace ptrace
Tracepoint 1 at 0xb7f22996
(gdb) actions
Enter actions for tracepoint 1, one per line.
End with a line saying just "end".
> collect $regs
> end
```

Como veis he puesto un tracepoint en la función **ptrace**, de modo que cada vez que el programa pase por ahí tome los valores de todos los registros. Normalmente el tracepoint por defecto esta activado, esto lo podemos ver con:

```
(gdb) info trace
Num Enb Address    PassC StepC What
1       y   0xb7f22996 0     0      <ptrace+6>
  Actions for tracepoint 1:
      collect $regs
      end
```

Bajo Enb (enable) vemos que si (y) está activado. Ahora lo podemos ejecutar:

```
(gdb) tstart
Trace can only be run on remote targets.
```

Y aquí se acabo todo, la idea es buena pero parece que solo se puede utilizar de forma remota ¿? Lo cual es un poco raro, porque si se puede utilizar de forma remota, también debería de poder hacerse de forma local. Digo lo mismo, si alguien sabe como activarlo que lo diga.
Podría ser que haya que instalar gdb con alguna opción específica ??; al instalarlo con aptitude install gdb, se instala con la opciones básicas y puede

que no sea suficiente.

Conclusión.

Creo que aunque básico hemos dado un buen repaso al debugger por excelencia de Linux. Yo creo que era fundamental tener una buena base con gdb, tanto para el que vaya a utilizar frontends graficos basados en él, como para entender la depuración de programas en linux.
Han quedado muchas incógnitas, que intentaremos ir solucionando en próximos trabajos, al igual que veremos otras herramientas que puedan completar o mejorar las que hemos visto hasta ahora.

Si alguien quiere aportar ideas, correcciones,soluciones etc..:

cvtukan(arroba)gmail.com

También me podéis localizar 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.....