INTRODUCCION AL ENSAMBLADOR BAJO GNU/LINUX
por YbY

-[0x0F]----------
-[Ensamblador bajo Linux]----------
-[by YbY]----------SET-23-

INTRODUCCION AL ENSAMBLADOR BAJO GNU/LINUX

Bueno, pues despues de daros un poco la tabarra con mi articulo sobre el MIPS R2000, vamos a continuar con el bajo nivel.
Esta vez va a pasar por nuestro laboratorio particular el ensamblador del 80x86 bajo el entorno GNU/Linux.
Aqui teneis los puntos que trataremos mas o menos:

---

1. Introduccion

Bueno, en primer lugar deciros que esto no pretende ser un sustituto de lo que la gente ya ha escrito (y muy bien, por cierto) sobre ensamblador bajo Linux.
El que escribe estas lineas no es, ni mucho menos, un experto en ensamblador. No dejo de ser un simple aprendiz y como aprendiz, escribo este tutorial para que otros aprendices aprendan. :)
De todas formas, comentar que la mayoria de la informacion que el que escribe ha encontrado sobre el tema (excepto el Linux Assembly HOWTO) estaba en ingles, asi que ahora ya no teneis excusa para no poneros con vuestro entorno favorito.

Bueno, en primer lugar vamos a ver los puntos (aparte de la simple curiosidad) que le pueden llevar a uno querer programar en ensamblador, habiendo actualmente lenguajes tan potentes como C:

- Querer programar un virus/troyano

- Optimizar partes de codigo extremas

- Querer aprender a muy bajo nivel como funciona un sistema operativo

- Optimizar un compilador que estas programando

...y bueno, creo que con esto ya os haceis una idea. Por supuesto, programar en ASM tambien tiene sus desventajas:

- perdida de la estructuracion en un programa
- portabilidad imposible
- mayor facilidad para equivocarse

Pero bueno, aunque luego no vayamos a utilizar realmente ensamblador para programar, pues por lo menos si que podremos tener una idea de lo que estamos viendo si hacemos algo de debugging o lo que sea..

En fin, me dejo ya de rollos y vamos ya a entrar en materia, porque va para rato...

2. El ensamblador

Este es un punto bastante polemico; mientras que unos se decantan por el AS (el GNU/Assembler), que utiliza la sintaxis AT&T, otros prefieren NASM (Netwide Assembler), que utiliza la sintaxis que hemos utilizado toda la vida bajo MS-DOS y bajo Win32.

La ventaja que tiene el usar el GNU AS, es que si por ejemplo, vais a programar en C y quereis meter codigo en ASM en medio de un programa en C (con las correspondientes directivas) os va a resultar muy facil.
La desventaja, por supuesto, es que si no estamos acostumbrados a la sintaxis AT&T, pues acabamos locos y al final no sabemos si el registro destino se ponia a la izquierda o a la derecha o si se ponian simbolos antes o...
Bueno, quiza los que no tengan ni idea de que es la sintaxis AT&T no entiendan de lo que hablo. Alla va un ejemplo:

```
 * Sintaxis AT&T *                              * Sintaxis Intel *
   movw %bx, %ax       ---------------------->      mov ax, bx
```

Esta instruccion lo que hace es pasar el contenido del registro BX al registro AX. La primera cosa que llama la atencion es que los registros se ponen al reves en una con respecto a la otra.
En la sintaxis AT&T, se indica el operando fuente a la izquierda, mientras que en la Intel lo indicamos a la derecha.
Otra cosa a tener en cuenta es el simbolo "%" que anteponemos en la sintaxis AT&T a los registros.
Por ultimo, a lo que nos indica de que instruccion se trata en si (la instruccion es la MOV), se le pone en la AT&T una letra mas para indicar con que estamos trabajando.
Si ponemos "w" indicamos que estamos trabajando con una palabra del procesador (que en los 80x86 es de 16 bits;
lo de "w" es de word = palabra en ingles).

Asimismo, ponemos "b" para byte (8 bits) y "l" para long (32 bits).
Aqui vamos a utilizar la sintaxis Intel, entre otras cosas porque es con la que mas soltura tengo.
De todas formas un dedicare un parrafillo como minimo a explicar en

detalle la sintaxis AT&T, ya que una gran cantidad de codigo que circula por ahi esta preparadito para compilar con el GAS.

Bueno, pues a parte del GAS, tenemos otras opciones, como ya he dicho antes.
La mas destacada es el NASM, que utiliza sintaxis Intel. El NASM esta disponible para muchos entornos diferentes, con lo cual lo tendremos muy comodo para pasarnos de uno a otro y eso. Tambien dedicare al NASM su correspondiente apartado. Otra opcion es el AS86, que no he probado en mi vida. Segun pone en el Assembly-HOWTO utiliza sintaxis Intel, salvo ligeras modificaciones en los modos de direccionamiento. Pero bueno, no lo trataremos aqui. Si alguien lo controla y se anima a escribir pues que lo haga :-)

De todos modos, deciros que lo importante, mas que el ensamblador, es la teoria que os explicare, asi que lo importante es que pilleis los conceptos para luego aplicarlos con el ensamblador con el que os sintais mas comodos..

3. NASM

3.1 Consiguiendo e instalando NASM

El website oficial de NASM es <http://www.cryogen.com/Nasm>.
Desde alli os podeis bajar la ultima version disponible.
Tambien teneis un servidor FTP en Francia:
<ftp.fr.kernel.org/pub/software/devel/nasm>
(directorios binaries y source)

La ultima version, a fecha de escribir esto es la 0.98.

Bueno, una vez tengamos los fuentes (en un fichero llamado nasm-0.98.tar.gz o algo asi), vamos a instalarlos.
Si sabeis algo de Linux y sabeis como instalar un programa entonces pasad de leer esto; lo he explicado para que el manual este al alcance de todos. Bueno, pues una vez lo tenemos, lo copiamos a un directorio temporal y lo descomprimimos asi:

```
$ tar xvfz nasm-0.98.tar.gz
```

Esto nos creara un directorio llamado "nasm-0.98". Pues entramos a dicho directorio, y lo que vamos a hacer es compilar el programa:

```
$ ./configure
$ make
```

Ahora veremos como el ensamblador se compila con el gcc y una vez termine tendremos dos ficheros ejecutables listos para utilizar: nasm y ndisasm.
El primero es el ensamblador en si y el segundo es el desensamblador.
Ahora los copiaremos a un directorio que este en el PATH para poder llamarlos desde cualquier sitio:

```
$ cp nasm /usr/bin
$ cp ndisasm /usr/bin
```

Despues de esto solo nos queda copiar las paginas man para tenerlas disponibles con el comando man:

```
$ gzip nasm.man
```

```
$ cp nasm.man.gz /usr/man/man1/nasm.1.gz
$ gzip ndisasm.man
$ cp ndisasm.man.gz /usr/man/man1/ndisasm.1.gz
```

Si en vuestra distribucion teneis las paginas man en otro sitio,
realizad los cambios oportunos.
Con esto ya podemos teclear man nasm y man ndisasm y nos saldran las
respectivas paginas del manual, que nunca vienen mal en caso de apuro.

Otra cosa que trae el NASM en el TGZ es la documentacion oficial. En
las versiones anteriores a la 0.98 esta venia tal cual en el paquete,
pero en la 0.98, en el directorio doc, tenemos las fuentes, que tambien hay
que compilar, tecleando make. Con esto se nos crearan los ficheros
correspondientes a la documentacion en los formatos TXT, PS, RTF,
INFO, HTML, HPJ, etc., para podernoslos imprimir a gusto o hacer lo que
queramos (y todo sin pagar un duro :-) Tan solo deciros que el manual viene
MUY completo y que tiene (en la version PostScript) la friolera de 139
paginas.

En fin, ahora ya estamos listos para ensamblar nuestro primer programa.
Solo nos falta escribirlo ;-)

3.2 Introduccion al NASM

Esto va a ser una breve intro a este ensamblador para los que ya
controlan algo de TASM. Por supuesto, no voy a explicar, ni mucho
menos, todas las diferencias.
A los que quieran profundizar mas los remito al manual.

Una cosa que hay que destacar de Nasm, antes de meternos a ver las
diferencias, es que se pretende alejar de las sentencias tipo .IF
que muchos compiladores (por ej. el MASM) han ido introduciendo en su
sintaxis. De esta forma, se tiene mayor control sobre el codigo generado,
que es lo que nos interesa, ymas programando en ensamblador.
Bueno, veamos:

- Tama~o de los operandos:
  Para indicar el tama~o de los operandos de la operacion a realizar,
  Tasm utiliza la siguiente sintaxis:

```
  mov  eax, dword ptr [esi]   (32 bits)
  mov   ax,  word ptr [esi]   (16 bits)
  mov   ah,  byte ptr [esi]   (08 bits)
```

  En Nasm, en cambio, se hace asi:

```
  mov  eax, dword [esi]       (32 bits)
  mov   ax,  word [esi]       (16 bits)
  mov   al,  byte [esi]       (08 bits)
```

- Sistemas de numeracion:
  Nasm acepta la sintaxis de C y la de ASM, de modo que el numero de
  la bestia ;P lo podemos expresar como: 0x29Ah o 29Ah.

- Reservar memoria:
  Para reservar una double word en Tasm lo hariamos asi:

```
        Doble_palabra:    dd      DUP(?)
```

  Mientras que en Nasm lo hacemos de la siguiente forma:

```
        Doble_palabra:    resd    25
```

  (Lo mismo con las otras: db -> resb; dw -> resw).

Para reservar memoria e inicializarla a algun valor seria:
Handler: times 100 db 1

Esto lo que hace es decirle al ensamblador:
"escribe 100 veces db 1", de forma que lo que hace es reservar 100 bytes.

Esta sentencia se puede utilizar en otros contextos (siempre que se trate de repetir algo).

- Includes:
Para incluir a los programas ficheros con definiciones se hace asi:
%include "elf_header.inc"

- Indicando offsets:
En Tasm "mov eax, offset VirStart" copia la direccion de memoria donde reside la variable "VirStart" al registro EAX. Igualmente, podriamos indicarlo con "lea eax, VirStart".

En Nasm esto es diferente.
Cuando ponemos: "mov eax, VirStart"
nos referimos siempre a la direccion de VirStart, de forma que almacenariamos en EAX un puntero a VirStart.
Para indicar que mueva el contenido, en vez de la direccion de memoria lo indicamos en Nasm asi: mov eax, dword [VirStart]

- Etiquetas:
Olvidaros en Nasm de las directivas PROC y similares. Aqui, si tenemos que escribir una funcion, lo hacemos con etiquetas.
Otra cosa curiosa es que se pueden escribir etiquetas locales a otras etiquetas. Por ej.:

```
Inicio:
  . . .
  .principio:   (1)
  . . .

En_medio:
  . . .
  .principio:   (2)
  . . .
```

Esto es completamente valido. Para referenciar, desde cualquier parte de nuestro programa el punto (1), escribiriamos:
Inicio.principio (lo mismo con (2)).

Ahora, veamos que parametros tenemos que indicarle a nasm para ensamblar nuestros programas. Si escribimos "nasm -h", nos aparecera lo siguiente:

```
------------------------------------------------------------------------
usage: nasm [-@ response file] [-o outfile] [-f format] [-l listfile]
            [options...] [--] filename
    or nasm -r   for version info

    -e          preprocess only (writes output to stdout by default)
    -a          don't preprocess (assemble only)
    -M          generate Makefile dependencies on stdout

    -E<file>    redirect error messages to file
    -s          redirect error messages to stdout

    -g          enable debug info
```

```
    -F format   select a debugging format

    -I<path>    adds a pathname to the include file path
    -P<file>    pre-includes a file
    -D<macro>[=<value>] pre-defines a macro
    -U<macro>   undefines a macro
    -w+foo      enables warnings about foo; -w-foo disables them
    where foo can be:
    macro-params   macro calls with wrong no. of params (default off)
    orphan-labels  labels alone on lines without trailing `:'
                   (default off)
    number-overflow numeric constants greater than 0xFFFFFFFF
                   (default on)

response files should contain command line parameters, one per line.

For a list of valid output formats, use -hf.
For a list of debug formats, use -f <form> -y.
```

Os paso a comentar los parametros mas importantes (al menos en un principio):

-f <format>
  Con esto le indicamos en que formato queremos ensamblar el programa:
  bin   --> forma binaria "a pelo"
  aout   --> para producir ficheros objeto a.out de Linux
  elf   --> para producir ficheros objeto elf de Linux

-o <output_name>
  Con esto le indicamos el nombre del fichero que generara.

-i <directory>
  Para especificar un directorio aparte donde buscar los includes.

Una vez creado el fichero objeto con nasm, habra que enlazarlo con el compilador de C de GNU (gcc) de la siguiente manera para obtener un ejecutable:

```
$ gcc <nombre_fichero_objeto> -o <nombre_del_ejecutable>
```

El siguiente shell script realiza esta tarea automaticamente:

```
<++> linuxasm/asm.sh
#!/bin/sh
# asm.h
# shell script para ensamblar/enlazar automaticamente
# parametros: nombre del fichero fuente sin la extension asm
nasm -f elf -o tmp.o $1.asm
gcc tmp.o -o $1
rm tmp.o
<-->
```

Con este script, para ensamblar y enlazar el fichero prog.asm se haria:

```
$ asm.sh prog
```

Con esto se crearia el ejecutable prog.

Bueno; con esto ya doy por terminada la introduccion al Nasm. Ya os digo que es muy recomendable que os mireis la documentacion si quereis controlar de verdad el ensamblador. De todas formas tambien os recomiendo

que antes os espereis a que toquemos brevemente el GAS, para ver cual os gusta mas y dedicarle mas tiempo.

4. GAS

4.1 Introduccion al GAS

No explicare donde conseguir el GAS porque es casi seguro que lo tengais ya listo para ensamblar en vuestra distribucion de Linux.
Si no es asi, dirigiros al website de vuestra distribucion o a los numeros FTP que hay por ahi con soft para Linux y seguro que lo encontrais.

Los parametros que se le pueden pasar mediante la linea de comandos al programa son practicamente los mismos que al gcc, salvo algunas matizaciones.
Aqui solo veremos los mas importantes. Para mas info: man as ;)
Con "-o" le indicais el fichero destino, como siempre, y con "-O" para optimizar.
Se puede utilizar directamente "as" para ensamblar un programa, solo que despues tendremos que utilizar ld (el enlazador de GNU) para indicarle que librerias tiene que utilizar y todo el rollo, por lo que lo mas comodo es utilizar gcc para ensamblar y enlazar al mismo tiempo de la siguiente forma:

```
$ gcc -O prog.asm -o prog
```

4.2 La sintaxis AT&T

Bueno, ahora os explicare mas o menos como pelearos con la sintaxis que nos trae de cabeza a los que aprendimos ASM bajo el DOS.

- Valores inmediatos:
  van precedidos por "$"
  ej.: movw $8, %ax (copia el valor inmediato 8 al registro AX).

- Registros:
  van precedidos por "%"
  ej.: movd %eax, %ecx (copia el contenido de EAX a ECX)

- Orden de los operandos:
  como habreis podido deducir es al reves que en la sintaxis Intel.
  ej.: mov ecx, eax --> movd %eax, %ecx
      (estas dos instrucciones copian el contenido de EAX a ECX).

- Tama~o de los operandos:
  se especifica posponiendo a los opnames los siguientes sufijos:

     b   -> byte (08 bits)
     w   -> word (16 bits)
     l   -> double word/long (32 bits)

  ej.: "pushl $5" introduce en la pila el valor inmediato 5, pero
       como un long (32 bits) para lo cual lo extiende de signo.
       En cambio "pushw $5" lo introduce extendiendole el signo solamente
       hasta los 16 bits (1 word).

  Nota: se supone que si ponemos un opname sin sufijo, el compilador

intenta buscar el tama~o de los operandos. De todas, formas, recomiendo ponerlo para tener claro que se esta haciendo.

- Saltos:
  los saltos largos se especifican con lcall o ljmp de la siguiente forma:
    lcall $seccion / ljmp $seccion
  o bien:
    lcall $offset / ljmp $offset
  hay que fijarse en que las direcciones que le pasemos son valores inmediatos y que por lo tanto hay que anteponerles el simbolo "$" (asi como Micro$oft indica que es *inmediato* el hundimiento de la compa~ia ;-))

- Referencias a memoria:
  esto se hace muy parecido al MIPS RX000:
    seccion:desplazamiento(base, indice, escala)
  De forma que la direccion de memoria resultante es (dentro de la seccion):
    base + desplazamiento + indice * escala
  ej.: si suponemos que tenemos el delta offset en EBP:
       movw %ax, elf_h(%ebp)
       lo que haria seria copiar el contenido de AX a la direccion de memoria indicada por EBP+elf_h.

Bueno, con esto uno se da cuenta de como los informaticos nos las apa~amos para fastidiar a los otros haciendo las cosas diferentes a ellos. Ejemplos de esto lo encontramos, entre otras cosas, en el "endianismo"
de esto lo encontramos, entre otras cosas, en el "endianismo".
Por supuesto, la sintaxis de los ensambladores no iba a ser menos ;)

4.3 GCC + ASM

Para compilar programas en C que incluye codigo en ensamblador entre su codigo (utilizando las sentencias asm, etc.), debeis hacerlo asi:

```
$ gcc -O2 -fomit-frame-pointer -m366 -Wall prog.c -o prog
```

Bueno, ahora que ya sabemos mas o menos como van los dos ensambladores mas populares y hemos escogido el mas nos gusta, vamos a programar nuestro primer programa en ensamblador (por fin! ;) Deciros que yo he escogido el NASM, y que la mayoria de codigo que escribire sera para NASM, aunque tendre clemencia de los "GASeros" y pondre tambien algo de codigo para este otro ensamblador.

5. Nuestro primer programa en Linux-ASM

Como no, vamos a ver el tipico "Hola, mundo!" para pillar los conceptos basicos del ensamblador bajo Linux. Alla va el codigo, y despues explicare cada parte:

```
<++> linuxasm/holamundo-nasm.asm
; HOLAMUNDO.ASM
global main
extern printf
```

```
section .data
mensaje    db   "Hola, soy un programa en Linux-ASM", 0Ah, 0

section .text
main:
                push dword mensaje
                call printf
                pop  eax
                ret
<-->
```

Ahora que ya os habeis dado de bruces contra el primer programa, voy a explicaros los puntos mas importantes:

- Con la directiva extern, indicamos, al igual que en Win32, las APIs que vamos a utilizar para el programa. En este caso, se trata de la funcion printf de C.

- Declaramos una etiqueta global llamada main. Esto es para que tanto el enlazador (el gcc) como el cargador del SO, sepan donde esta el punto de entrada del programa.

- Las directivas section se utilizan para declarar secciones en el ejecutable.
  En este caso tendremos dos: una de datos (.data) y otra de codigo (.text). Es parecido a las sentencias .code y .data del Tasm.

- Para llamar a una funcion, los parametros se apilan en orden inverso (el primer argumento lo apilas el ultimo).
  Esto es lo que se llama la sintaxis de llamada de C.
  Como a printf, en nuestro caso, la vamos a llamar con un solo parametro, lo apilamos y listo. Hay que tener en cuenta que tenemos que indicar el tama~o de lo que apilamos con las sentencias dword, word o byte. Para llamar a la funcion en si, se utiliza call (como en los demas entornos).
  A printf hay que pasarle un puntero a una cadena de caracteres que en C es un array de caracteres, con el ultimo caracter igual a 0.
  Recordemos tambien que en Nasm no se pone para indicar un puntero "push offset mensaje" como en Tasm, sino que cuando ponemos el nombre de la etiqueta de la variable ya nos referimos a la direccion de memoria y no al contenido (si no sabeis de lo que estoy hablando leeros la intro al Nasm de mas arriba otra vez :).)
  El 0Ah es el caracter de nueva linea (lo que en C es \n).

- Para retornar al sistema operativo, simplemente invocamos a ret.
  Para que os hagais una idea, esto es lo que pasa:

  1. El OS Loader carga vuestro programa con call.

  2. En la pila queda la direccion de retorno.

  3. Se ejecuta vuestro programa.

  4. Al ejecutarse la instruccion ret, se vuelve al punto desde donde habia sido llamado el programa.

  Esto es valido siempre que no modifiquemos la pila, ya que es donde se guarda la direccion de retorno cuando se ejecuta call. (Por eso es por lo que antes de invocar a ret ponemos un pop eax, para que el puntero de pila apunte otra vez a la direccion de retorno; restaurar la pila es una de las cosas importantes en la programacion en ASM bajo Linux).

Bueno, supongo que el programita os habra quedado claro; es muy basico.

Ahora vamos a ver el mismo programa, pero para el GAS:

<++> linuxasm/holamundo-gas.s

```
.main

.section data
mensaje:        .string  "Hola, mundo!\n"

.text
main:

        pushl $mensaje
        call  printf
        popl  %eax
        ret
```

<-->

No hay muchas diferencias significativas, excepto la sintaxis (como no ;))
Tambien hay que definir como global la etiqueta main (con .global) y
podemos definir las secciones como queramos, con la directiva .section.
Sin embargo, vienen dos ya predefinidas: .data y .text, para datos y codigo
respectivamente.

6. INT 80h

Probablemente muchos de vosotros echeis de menos las interrupciones
de cuando trabajabamos bajo DOS. Pues os tengo una sorpresa reservada:
las llamadas al sistema de Linux estan mapeadas en Linux en la
interrupcion software 80h.
Los numeros de las llamadas al sistema los podeis encontrar en el
fichero unistd.h, que esta entre el codigo fuente de Linux. En mi
distribucion estaba, concretamente, en /usr/src/linux/include/asm/unistd.h
Asi, este programa hace lo mismo que los anteriores, mediante una
llamada al sistema:

<++> linuxasm/holamundo-SC.asm

```
global main             ; definimos la etiqueta main

section data            ; seccion de datos
mensaje:        db      "Int 80h??? Si!!! :)", 0Ah, 0

section .text           ; seccion de codigo
main:
        mov  eax, 4             ; syscall 4 = write
        mov  ebx, 1             ; descriptor de fichero
        mov  ecx, mensaje       ; puntero a la cadena
        mov  edx, 21            ; tama~o de la cadena
        int  80h

        mov  eax, 1             ; syscall 1 = exit
        xor  ebx, ebx           ; ebx = 0 (codigo de error)
        int  80h
```

<-->

La forma de hacer una llamada al sistema es la siguiente:

1. Poner en EAX el numero de la llamada al sistema (unistd.h)

2. Poner los parametros de la llamada en los registros en el orden:
   EBX - ECX - EDX - ESI - EDI
3. Provocar la interrupcion con INT.

Asi, en el ejemplo, para imprimir por pantalla una cadena utilizamos
la llamada al sistema write:

ssize_t sys_write(unsigned int fd, const char *buf, size_t count)

Resumiendo:

1. Colocamos en EAX el numero de la llamada (4).
2. Colocamos en EBX el descriptor de fichero a utilizar (el
   descriptor de fichero 1 es STDOUT en Linux; la salida estandar).
3. Colocamos en ECX un puntero a la cadena.
4. Colocamos en EDX el tama~o de la cadena (incluido el \n y el 0).
5. INT 80h

Para salir:

1. Colocamos en EAX el numero de la llamada (1).
2. Colocamos en EBX el codigo de salida (0).
3. INT 80h

Por supuesto, eso de tener que especificar la llamada mediante
numeros es muy engorroso, asi que lo mejor es utilizar un fichero
engorroso, asi que lo mejor es utilizar un fichero
include al estilo de los que Jacky Qwerty / 29A ;) tiene para Win32
con todas las definiciones y eso.
Os aconsejo los de Konstantin Boldyshev (que son los unicos que yo
conozco).
Si a Green Legend le parece bien incluiremos un TGZ con los mas
importantes, y si no de todas formas en el apendice teneis donde
encontrarlos.
Circulan por ahi listas en HTML con todas las syscalls y sus
parametros y eso. Os pondre las URLs en el correspondiente apendice.

7. El formato ELF

Bueno, ahora que ya sabemos escribir mas o menos cualquier programilla
sencillo en ensamblador, vamos a detenernos un poco en el estudio del
formato de ficheros ejecutables ELF. Por supuesto, esto sera
solamente una breve introduccion; para mas informacion os remito a la
documentacion oficial.
Para nuestras practicas necesitaremos un editor hexadecimal.
El mejor que hay para Linux ahora mismo es el BIEW, que es un clon
para Linux del popular HIEW para entornos DOS. La ultima version
disponible es la 5.1.1.
Otra opcion es el que viene con el KDE (khexdit) o el que viene
con el GNOME (GHex). Seguramente habra mas, pero estos son los mas
conocidos.
Para estudiar el formato ELF, lo mejor que podemos hacer es coger un
fichero en dicho formato e ir viendo, con el mencionado editor hexadecimal,
que es cada cosa en la cabecera.
Pues eso vamos a hacer. Ahora pillamos el ultimo "hola mundo" que hemos
hecho (el que utilizaba llamadas a las funciones) y vamos a "diseccionarlo" ;)
Sin embargo, si lo enlazamos como hemos hecho antes, el gcc nos metera mucha
"basura" en el ejecutable, asi que lo compilaremos asi:

```
$ nasm -f elf holamundo-SC.asm
$ gcc -Wall -s -nostdlib -o holamundo-SC holamundo-SC.o
```

El compilador dara un error, pero no debemos preocuparnos, ya que el programa funcionara igual, y por supuesto la cabecera seguira siendo valida para nuestros propositos. Ahora lo que vamos a hacer va a ser editarla con el editor hexadecimal.

Os voy a poner aqui la estructura del ELF header para que podais ir siguiendola en el fichero que estais editando:

<++> linuxasm/elf-header.txt

ESTRUCTURA DEL ELF HEADER

CAMPO	OFFSET(hex.)	QUE ES??
e_ident	0	Firma y diferentes flags
e_type	10	Tipo de fichero
e_machine	12	Maquina para el que fue creado
e_version	14	Version de ELF header
e_entry	18	Punto de entrada (virtual address)
e_phoff	1C	Offset del program header
e_shoff	20	Offset del sections header
e_flags	24	Otros flags
e_ehsize	28	Tama~o del ELF header
e_phentsize	2A	Tama~o de una entrada en el prog h
e_phnum	2C	Numero de entradas en el prog h
e_shentsize	2E	Tama~o de una entrada en el sec h
e_shunum	30	Numero de entradas en el sec h
e_shstrndx	32	Numero de entrada del nombre de la sec

<-->

El ELF header ocupa en el fichero ELF 52 bytes en total. Lo primero que tenemos es la firma. Al igual que en los de DOS la firma era MZ (o ZM), aqui la firma es 7F 45 4C 46.
O sea, que si por alguna extra~a razon ;) queremos averiguar si un fichero es un ejecutable, pues comparamos sus primeros 4 bytes con estos.
Despues vienen 12 bytes mas con diferentes flags, que por el momento no nos interesan. A continuacion, tenemos el campo e_type, que nos dice el tipo de fichero ELF del que se trata.
En nuestro caso vale 02, ya que es un fichero ejecutable. Luego, en e_machine tenemos el tipo de maquina que se necesita para ejecutar el fichero.
Para los 80386+ el numero es el 3. En el siguiente campo tenemos el valor 1, que no es mas que la version de ELF header (se supone que se esta investigando sobre nuevas versiones para hacer los ELF aun mas flexibles).

Despues viene un campo MUY importante ;) El campo e_entry indica la direccion donde esta el punto de entrada: el codigo que se ejecutara.
En mi caso vale 00 00 80 80.

Los demas campos se refieren a otras partes del fichero ELF (el numero de secciones, la direccion del program header, etc.). Basicamente, un fichero ELF es esto:

```
  Linking View                      Execution View
  ============                      ==============
  ELF header                        ELF header
  Program header table (optional)   Program header table
  Section 1                         Segment 1
  ...                               Segment 2
  Section n                         ...
  Section header table              Section header table (optional)
```

A la izquierda tenemos el fichero tal y como esta en el disco al enlazarlo, y a la derecha tal y como se vera cuando se transforme en un proceso.

Para mas info sobre el ELF header os podeis mirar el fichero: /usr/include/linux/elf.h

Para acabar con esta breve introduccion al formato ELF (para que al menos sepais la estructura basica de lo que ejecutais) os presento un programa que nos puede ser util para ver la estructura de un fichero ELF. Por supuesto podria hacer uno en ASM, pero como soy asi de perro utilizare uno que viene con todas las distribuciones de Linux: objdump.

Para aprender como se usa este programilla, pues vamos a pillar el fichero ese que hemos ensamblado antes (el hola.asm "capado" ;) y vamos a ver que hace con el.

Con el parametro -a, objdump nos muestra breve sobre la cabecera y con -f nos muestra informacion un poco mas completa sobre estas:

```
$ objdump -f hola

hola:     file format elf32-i386
architecture: i386, flags 0x00000102:
EXEC_P, D_PAGED
start address 0x08048080
```

Para mostrar aun mas informacion sobre las cabeceras, lo invocamos con la opcion -x.

Como era de esperar, el fichero es un ELF para plataformas 80386+ :P

Con el parametro -d, nos muestra el codigo desensamblado:

```
$ objdump -d hola

hola:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
 8048080:       b8 04 00 00 00          mov    $0x4,%eax
 8048085:       bb 01 00 00 00          mov    $0x1,%ebx
 804808a:       b9 9f 80 04 08          mov    $0x804809f,%ecx
 804808f:       ba 15 00 00 00          mov    $0x15,%edx
 8048094:       cd 80                   int    $0x80
 8048096:       b8 01 00 00 00          mov    $0x1,%eax
 804809b:       31 db                   xor    %ebx,%ebx
 804809d:       cd 80                   int    $0x80
```

Como he dicho, desensambla *el codigo*, que se corresponde en nuestro programejo a la seccion .text. Como podeis observar, el codigo es el mismo que hemos escrito antes (en sintaxis AT&T, por supuesto). Esto es asi porque era un programa en ensamblador, pero probad a haced uno en C y vereis las virguerias que hace a veces, que se podrian optimizar al 100% (y eso que el gcc es un compilador bastante apa~ao...)

Para ver todas las secciones:

```
$ objdump -D hola


hola:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
 8048080:       b8 04 00 00 00          mov    $0x4,%eax
 8048085:       bb 01 00 00 00          mov    $0x1,%ebx
 804808a:       b9 9f 80 04 08          mov    $0x804809f,%ecx
 804808f:       ba 15 00 00 00          mov    $0x15,%edx
 8048094:       cd 80                   int    $0x80
 8048096:       b8 01 00 00 00          mov    $0x1,%eax
 804809b:       31 db                   xor    %ebx,%ebx
 804809d:       cd 80                   int    $0x80
Disassembly of section data:

0804809f <data>:
 804809f:       49                      dec    %ecx
 80480a0:       6e                      outsb  %ds:(%esi),(%dx)
 80480a1:       74 20                   je     0x80480c3
 80480a3:       38 30                   cmp    %dh,(%eax)
 80480a5:       68 3f 3f 3f 20          push   $0x203f3f3f
 80480aa:       53                      push   %ebx
 80480ab:       69 21 21 21 20 3a       imul   $0x3a202121,(%ecx),%esp
 80480b1:       29 0a                   sub    %ecx,(%edx)
        ...
```

En fin, con esto mas o menos ya sabeis analizar que es cada cosa en un fichero ELF, sobre todo en lo que respecta a la cabecera ELF propiamente dicha. Os recomiendo que os mireis la pagina man de objdump, que explica todas las opciones de este programa.

8. VIRUS

Bueno, pues a pesar de no estar demasiado generalizados, los virus para UNIX en general, y Linux en particular son algo *perfectamente* viable. De hecho, los administradores de sistemas UNIX ya pueden empezar a temblar si las tecnicas viricas se empiezan a desarrollar en serio para sus sistemas. Por supuesto, el colmo de un virus para Unix es que sea ejecutado por el usuario root, ya que se puede decir, literalmente, que tiene el poder sobre el maldito sistema. Y el concepto de virus nos lleva mas alla que los troyanos tan desarrollados hasta hora en el mundillo Linux.
Un troyano no deja de ser un troyano, y tienes que tener suerte para que root lo ejecute, pero un virus puede ir expandiendose a traves de los ficheros en los que tenga permisos (al principio de usuarios con poco poder en el sistema y despues poco a poco ir subiendo de niveles, hasta, probablemente que sea ejecutado por el root).
Un virus decente para Linux si que seria verdaderamente un virus porque muestra el ascenso a traves de las capas de seguridad hasta hacerse con el

poder absoluto (y dejarlo en manos de su creador ;))
Por lo tanto, lo mas interesante es programar un hibrido troyano/virus, que se vaya expandiendo y que cuando sea ejecutado por un usuario con cierto nivel, desempe~e sus acciones.
Una idea interesante seria tener un virus con diferentes modulos. Cada uno se ejecutaria segun los permisos del usuario que ha ejecutado el portador.
Pero bueno, me estoy yendo por las ramas y esto es un mero articulo de introduccion. Ademas, habra mucha gente que no tenga ni idea de que es un virus por lo que voy a empezar desde el principio.

Ah! y otra cosa: que conste que no pretendo fomentar la programacion de virus, pero he de admitir que se trata de una MUY buena forma de aprender sobre las interioridades de tu Sistema Operativo ;)

8.1 Concepto general de virus

Un virus no es mas que un parasito de ficheros ejecutables. La idea general es esta:

1. El parasito se a~ade a si mismo en alguna parte del portador, que no es mas que un fichero ejecutable.

2. Asimismo, modifica el punto de entrada en el que se empezara a ejecutar el codigo del portador, de forma que apunte a su propio codigo.

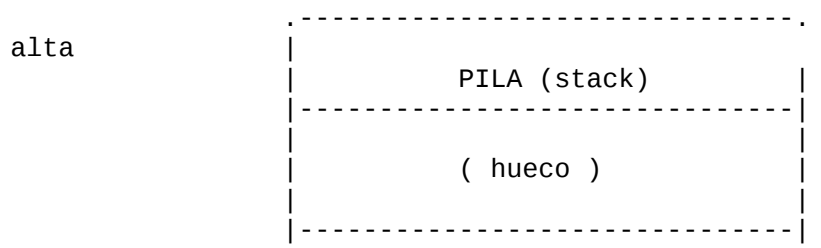
3. Al final del codigo del virus, debemos incluir una instruccion que pase el control de nuevo al portador, para que el usuario no sospeche.

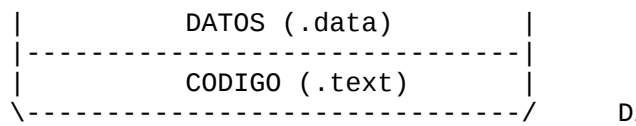
Y asi de simple es esto de los virus. El problema es que eso de "a~adir el codigo en alguna parte del portador" y lo demas presenta sus problemillas que discutiremos a continuacion... Ademas, deciros que la idea anteriormente expuesta corresponde a los virus llamados "runtime", puesto que al ejecutarse es cuando realizan la infeccion. Despues estan los virus residentes, etc., pero bueno, eso ya se vera... ;)

8.2 Primera (y ultima por ahora) aproximacion

Lo que sigue va a estar fuertemente basado en los magnificos documentos que ha escrito Silvio Cesare (un ejemplo de ellos lo teneis en el articulo "Shared Library Redirection via ELF PLT Infection" de la ultima Phrack (la 56))
Asi que nadie me venga ahora con que todo esto ya lo ha dicho el, porque ya lo se; pero esto es un articulo de introduccion, y viene bien apoyarse en trabajos tan buenos como los de Silvio. Al final, en el apendice, teneis donde encontrar los susodichos manuales, por si alguien necesita mas ;-)

Lo primero que tenemos que entender para aclararnos las ideas respecto a la infeccion, es el concepto de proceso. Un proceso es, a grandes rasgos, un programa ejecutandose en memoria:

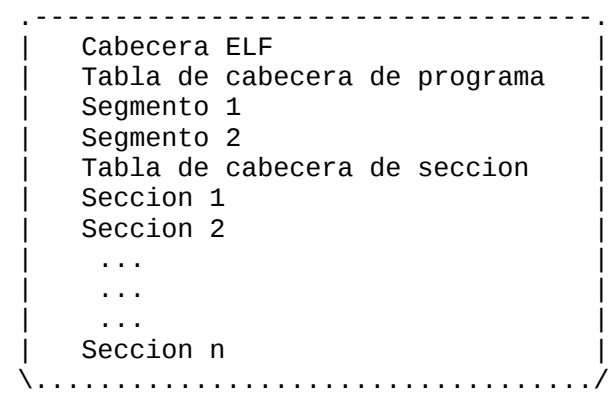




Dir. de mem + baja

Como vemos, un proceso se compone de tres segmentos: codigo, datos y pila. En el segmento de codigo, se puede leer y ejecutar; en el de datos, solamente leer y escribir. Cada segmento, puede estar compuesto por una o mas paginas (1 pagina = 1 "cacho" de 4 Kb). Recordemos, ademas, que la pila "crece" hacia abajo, hacia la seccion de datos.
(Para mas info sobre esto de la pila os animo a que os leais el articulo de Doing sobre el Stack Overflow publicado en SET ;-))
En un fichero ELF (el fichero en disco no el proceso en memoria), estos segmentos (el de codigo y el de datos) suelen estar tal cual, primero el de codigo y despues el de datos (generalmente y en circunstancias normales)

Antes, vimos una descripcion del formato ELF muy basada en su cabecera principal. Pues bien ahora veremos como es la estructura general de un fichero ELF:



Los segmentos, son los que hemos mencionado antes (text y data) y tienen asociadas las dos primeras secciones. Las restantes secciones, nos dan otra clase de informacion, como las tablas de simbolos, etc. Los puntos de entrada (en virtual address) del program header y el section header, estan en la cabecera principal del fichero ELF (ver el esquema de antes).
La idea principal, es que los segmentos de codigo y datos no estan pegados directamente, si no que existe un cierto espacio llamado "padding".
La cuestion es meter el codigo del virus en ese espacio entre el segmento de codigo y el de datos:

```
Cabecera ELF modificada
Tabla de la cabecera de programa
Segmento 1 (text y codigo del virus)
Segmento 2 (data)
Tabla de la cabecera de seccion
Seccion 1
Seccion 2
.......
Seccion n
```

Por lo tanto, tal y como explica Silvio Cesare, la forma de incluir a nuestro virus en el "lote", seria:

- Incrementar p_shoff (que nos indicaba el offset en el que se encontraba la tabla de cabecera de programa), teniendo en cuenta el tama~o del parasito que hemos a~adido.

- Hallar la cabecera de programa del segmento de codigo y:
  - Incrementar p_filesz, que nos indica el tama~o que ocupa el codigo fisicamente
  - Incrementar p_memsz, que nos indica el tama~o que ocupa el codigo cuando esta en memoria

- Para cada cabecera de programa cuyo segmento esta despues del de codigo (que es donde hemos introducido el virus):
  - Incrementar p_offset, que nos indica el offset del segmento en el fichero

- Para cada cabecera de seccion cuya seccion este despues de nuestra insercion:
  - Incrementar sh_offset, para tener en cuenta el nuevo codigo

- Insertar el virus en si en el fichero

Facil, verdad? ;P Supongo que no sabreis lo que son muchos de estos campos, ya que me he saltado muchas cabeceras en la explicacion, pero creo que vale para que os hagais una idea de como hacerlo.
De todas formas, decir que con esto no se infecta un ELF correctamente, por unas historias que hay con un campo llamado p_vaddr, entre otras cosas... Pero es que este articulo estaba dedicado al ensamblador; a servir de punto de despegue para los que quieren meterse en el tema, asi que considero que por ahora ya hay bastante sobre virus. Quien quiera investigar mas que se mire los docs que pongo mas abajo... De todas formas, no descarto que cuando domine un poco mas el tema me anime a escribir otro articulo exclusivamente dedicado a los virus, con algun virus programado por mi. Pero bueno, eso es otra historia... ;-)

9. "Bibliografia"

Esto es una lista con todo lo que necesitais para seguir adelante. Con este manual y leyendo todo lo que expongo aqui, podeis pillar un nivel decente en la programacion en ensamblador bajo Linux:

- Linux Assembly (http://www.linuxassembly.org)
  Esto es un site entero dedicado al ASM bajo Linux. Ahi podeis conseguir numerosos ejemplos, programas utiles (hasta un traductor de sintaxis AT&T/Intel ;) etc, etc. Tambien tienen una lista de correo; para suscribirse mandar un mensaje en blanco a la direccion: <linux-assebly-subscribe@egroups.com>

- Pagina de Silvio Cesare (http://www.big.net.au/~silvio)
  Pagina del principal investigador de tecnicas viricas bajo Linux. Leyendo sus documentos y analizando sus virus podeis empezar a programar los vuestros propios ;) Por supuesto, si alguien se anima a escribir algo sobre el tema y mandarlo a SET, yo sere el primero que aprendera sobre ello :) En esta pagina tambien podeis encontrar algunos articulos muy interesantes escritos tambien por Silvio (este tio es una ca~a) sobre las interioridades del kernel o tecnicas anti-debugging.

- Lista de correo sobre programacion de virus (http://virus.beergrave.net)
  Pese a que mucha gente dice que esta subscrita, yo lo intente y parece estar out por el momento :-? En fin, seguiremos probando...

- "Sistemas Operativos: Dise~o e Implementacion"
  Este es el libro sobre las interioridades de los SSOO por excelencia Se analiza en profundidad el MINIX, SO creado por el autor (Andrew

S. Tanenbaum) para la ocasion y que fue en el que se inspiro Linus Torvalds para programar las primeras versiones del kernel del Linux :) Este libro viene bien para saber lo que es una llamada al sistema, como chuta lo de los procesos, etc.

- "PC Assembly Language" de Paul A. Carter.
  Es un libro electronico en formato PostScript sobre la programacion en ASM en general, pero como todos los ejemplos estan hechos con el Nasm, pues viene bien para practicar con ese ensamblador. Ademas, no cuesta un centimo ;) Lo podeis conseguir en la pagina de su autor: <http://www.comsc.ucok.edu/~pcarter>

- Linux Assembly HOWTO
  No hace falta que os diga que es y donde conseguirlo... :P

- Pagina oficial de NASM (Netwide Assembler)
  Os la he puesto arriba pero os la digo otra vez para que lo tengais todo junto: <http://www.cryogen.com/Nasm> En el siguiente site FTP tambien podeis encontrar la ultima version: <ftp.fr.kernel.org/pub/software/devel/nasm>

Y bueno, ademas de todo esto, siempre teneis los motores de busqueda habituales para estas ocasiones ;)

10. Despedida

Bueno, pues se acabo lo que se daba :) Espero que os haya gustado; se que es muy basico y que posiblemente muchas cosas que se explican ya las sabiais, pero creo que cumple perfectamente el objetivo de ser un articulo de introduccion a un determinado tema. Como he dicho antes, no descarto la posibilidad de escribir articulos para cada una de las partes exclusivamente, a medida que vaya adquiriendo mas nivel, pero bueno, ya veremos... ;) Os animo a que me reporteis todos los fallos que encontreis en el documento en la siguiente direccion de correo: <yby@linuxfan.com>

Los saludos y eso tambien seran bien recibidos, por supuesto. Tambien animo a que encripteis vuestros mensajes con la siguiente llave publica:

[Daemon: Podeis encontrar la clave PGP de YbY al final de la revista]

Pues nada mas; a seguir aprendiendo y a demostrar que el hacking es algo mas que joder sistemas y manipular paginas web...

YonderBoY (YbY)
<yby@linuxfan.com>

EOF

Fonte: http://mgu7mxp3vxzfj7hs.onion/