

JUGANDO CON C, ASM Y SYSCALLS

Alejandro Hernández - @nitr0usmx

Tabla de Contenidos

- 1.-ASM.
 - 1.1.-Definición de ASM.
 - 1.2.-Arquitectura de computadoras.
 - 1.3.-Instrucciones básicas.
- 2.-SYSCALLS.
 - 2.1.-Definición de SYSCALL.
 - 2.2.-Ejemplos de Syscalls.
- 3.-EJEMPLOS
 - 3.1.-pariendo
 - 3.2.-FLE-ELFcorrupt
- 4.-CONCLUSION
- 5.-REFERENCIAS
- APÉNDICE A – CODIGOS
 - A.1.-pariendo.c
 - A.2.-pariendo.s
 - A.3.-FLE-ELFcorrupt.c
 - A.4.-FLE-ELFcorrupt.s

1.-ASM.

1.1.-Definición de ASM.

ASM (AsseMbly language) o lenguaje ensamblador, es el nivel de programación más bajo.

Los lenguajes de programación que conocemos están en diferentes niveles de abstracción, es decir, desde los de alto nivel como Visual Basic, Delphi, etc., hasta los de bajo nivel como ASM, pasando por lenguajes de medio nivel como Perl, C, Python, PHP, etc. Como vemos, el lenguaje más abstracto es el lenguaje ensamblador, ya que con este, podemos decirle que hacer directamente al microprocesador, como enviarle la instrucción que deseemos.

Claro, sabemos que las computadoras solamente entienden 1s y 0s (unos y ceros), pero he ahí el significado de los *códigos de operación* u *OPCODES* por su término en inglés, ya que estos son los números en si que el microprocesador entiende. Es por eso que cuando compilamos, enlazamos y creamos un ejecutable, nosotros podemos ver en su segmento de código los mencionados *OPCODES*, que no son ni más ni menos que las instrucciones que queremos que nuestro programa ejecute.

En conclusión, por medio del lenguaje ensamblador podemos hacer programas más rápidos, limpios, etc., y a la vez podemos combinarlo con la *API* que el sistema operativo nos ofrece. Por ejemplo, combinar ASM con la *API* de Windows o combinar ASM con syscalls en UNIX.

1.2.-Arquitectura de computadoras.

Como sabemos, una computadora tiene uno o más microprocesadores y cada uno de estos tiene una arquitectura. En si, la arquitectura es la forma de cómo un microprocesador trabaja internamente, la forma en como este busca y ejecuta las instrucciones, entre otras cosas.

En la actualidad existen muchas arquitecturas de computadoras. Quizás hayan escuchado cosas tales como “...*Con Sistema Operativo SUN OS 5.9 en Arquitectura SPARC o con Sist. Operativo IRIX bajo MIPS...*”, pues en estos ejemplos SPARC y MIPS son dos tipos de arquitecturas. La arquitectura más utilizada es la x86, ya que estos microprocesadores están en la mayoría de computadoras personales, de oficina, etc.

En este documento se tratarán ejemplos para arquitectura x86.

1.3.-Instrucciones básicas.

Y bien, veremos algunas instrucciones básicas que serán tratadas posteriormente en la programación de los ejemplos, pero antes de iniciar debemos saber que para programar en lenguaje ensamblador bajo x86 existen dos diferentes sintaxis, Intel y AT&T. Las diferencias entre sintaxis están fuera del contexto de este documento. Aquí utilizaremos la sintaxis AT&T.

Algunas instrucciones usadas en los ejemplos son:

xorl %regX, %regX

Esta instrucción pone en 0(cero) al registro *%regX*, es lo mismo que hacer *mov \$0, %regX* aunque más rápido ya que solamente se usa el mismo registro y no un valor inmediato, por lo cual toma menos ciclos de reloj.

movl fuente, destino Mueve 32 bits de fuente a destino

movw fuente, destino Mueve 16 bits de fuente a destino

movb fuente, destino Mueve 8 bits de fuente a destino

subl \$n, destino Resta n bytes a destino y el resultado lo almacena en destino

addl \$n, destino Suma n bytes a destino y el resultado lo almacena en destino

cmpl fuente, destino

Compara 32 bits entre fuente y destino, si son iguales, la flag de Z(zero flag) se activa

cmpb fuente, destino

Compara 8 bits entre fuente y destino, si son iguales, la flag de Z(zero flag) se activa

<i>je</i>	<i>etiqueta</i>	Si la flag Z(zero flag) está activada salta a etiqueta
<i>jne</i>	<i>etiqueta</i>	Si la flag Z(zero flag) no está activada, salta a etiqueta
<i>jmp</i>	<i>etiqueta</i>	Salta a etiqueta sin importar la flag Z(zero flag)
<i>pushl</i>	<i>valor</i>	Empuja a la pila el valor y al registro <i>%esp</i> se le restan 4.
<i>xchgl</i>	<i>%regX, %regY</i>	Intercambia los valores de los registros.

int \$0x80

Esta es la interrupción al sistema operativo, la cual toma el número de syscall que esté en el registro *%eax* y la ejecuta. Si dicha syscall necesita argumentos, estos van pasados en los registros *%ebx*, *%ecx*, *%edx*, *%esi*, *%edi*.

2.-SYSCALLS.

2.1.-Definición de SYSCALL.

El núcleo está pensado para facilitarnos servicios relacionados con el sistema de ficheros y con el control de procesos. Las llamadas al sistema (syscalls) y su biblioteca asociada presentan la frontera entre los programas del usuario y el núcleo, esta biblioteca (*libc*) se enlaza por defecto al compilar cualquier programa en C.

Los programas en ensamblador pueden invocar directamente a las llamadas al sistema sin necesidad de ninguna biblioteca intermedia. Dichas llamadas se ejecutan en modo protegido (*kernel-mode*), y para entrar a este modo hay que ejecutar una interrupción (*int* \$0x80).

Las llamadas al sistema de UNIX tienen un formato estándar, tanto en la documentación que nos brinda el sistema sobre ellas (páginas del manual, sección 2), como en la forma de invocarlas. Cuando una syscall ha fallado, esta devuelve el valor -1 . [1]

Puedes ver la lista de syscalls en el archivo */usr/include/asm-generic/unistd.h*.

2.2.-Ejemplos de Syscalls.

En esta parte mencionaré las syscalls que usaré en los dos ejemplos. Como había mencionado, cada syscall puede o no recibir argumentos. En el lenguaje C, las llamadas se usan de la misma forma que las funciones, se pasan los argumentos entre paréntesis y separados por comas; en lenguaje ensamblador, estos se pasan por los registros *%ebx*, *%ecx*, *%edx*, *%esi*, *%edi* (recordemos que en *%eax* va el número de syscall a ejecutar). Los valores devueltos por lo general quedan almacenados en el registro *%eax*.

Ejemplos:

write(fd, ptr, len)

Escribe en *fd*, *len* bytes localizados en *ptr*.

Ejemplo: `write(1, buffer, strlen(buffer));`

read(fd, ptr, len)

Lee de *fd*, *len* bytes y los coloca en la memoria apuntada por *ptr*.

Ejemplo: `read(1, buffer, 1024);`

fork()

Se duplica así mismo, es decir, crea un proceso hijo. El valor devuelto en *%eax* es 0(cero) para el proceso hijo, y diferente de 0 para el padre.

Ejemplo: `if(fork() == 0){printf("hijo");}else{printf("padre");}`

exit(status)

Termina un programa y devuelve a su proceso padre el valor en *status*.

Ejemplo: `exit(-1);`

open(filename, mode)

Abre el archivo apuntado por *filename* con modo *mode*(lectura, escritura o ambos). Aunque el modo puede ser una combinación de varios valores más, aquí solamente trataremos estos que son los más simples. Esta syscall también puede recibir un 3er argumento, pero este es opcional. En el registro *%eax*, queda el valor devuelto que es un *fd* (file descriptor) o -1 en caso de error.

Ejemplo: `open("/etc/passwd", 2);` // Abre el archivo en modo lectura y escritura.

lseek(fd, offset, where)

Se mueve *offset* bytes desde *where* en el archivo apuntado por *fd*. Si *where* es 0, significa el inicio, si es 1 es en donde está actualmente y si es 2 es al final. *fd* debe ser un descriptor de archivo devuelto por las syscalls *open()* o *creat()*.

Ejemplo: `lseek(myfd, 32, 0);` // Se coloca en el byte 32 del archivo.

close(fd)

Cierra el archivo apuntado por *fd* y obviamente ya no se pueden hacer operaciones sobre el.

Ejemplo: `close(myfd);`

3.-EJEMPLOS

En este punto veremos 2 ejemplos, en los cuales se hará uso de las syscalls: `read()`, `write()`, `open()`, `close()`, `fork()`, `lseek()` y `exit()`.

El primer ejemplo (pariendo) es una demostración de los llamados “*fork-bombs*”. Un *fork-bomb* es un tipo de Denegación de Servicio contra una sistema que use la llamada al sistema `fork()`. Sabemos que el número de programas y procesos que una computadora puede atender tiene un límite. [2]

Cuando una bomba de forks es llamada, este crea tantos procesos que llenan la tabla de procesos del Sistema Operativo, y por consecuencia, cualquier proceso que intente ejecutarse no podrá llevarse a cabo.

El segundo ejemplo (FLE-ELFcorrupt) es simplemente un programa que deja inservible a un binario ELF[3] cambiando ciertos bytes en el.

Ambos códigos completos están en el Apéndice A.

3.1.-pariendo

Para comprender fácilmente el funcionamiento de este programa lo analizaremos en un nivel de abstracción más alto, esto es, que he pasado el código en ensamblador a código en C. Iremos analizando parte por parte el código en C, para así comprenderlo más fácilmente en ensamblador.

En la primera parte imprimimos [write()] un mensaje para preguntar si está seguro de crashear el sistema. Leemos [read()] 4 bytes y los guardamos en la variable *esp*, y luego hacemos un cast para guardar solamente un byte en la variable *cl*. Luego comparamos dicho carácter; si este fue 'y' o 'Y' saltamos al símbolo *fuck_100p*, de no ser así saltamos a *g00d_bye* con lo cual terminamos[exit()] el programa.

```
main(){
    write(1, "Crash the system?(y/n): ", 24);

    int esp, eax;
    char cl;

    read(1, &esp, 4);
    cl = (char) esp;

    if(cl == 'y' || cl == 'Y')
        goto fuck_100p;
    goto g00d_bye;
```

Ahora, recordemos que fork() devuelve un 0(cero) en el hilo de ejecución del hijo y un valor diferente de 0(cero) al padre. Entonces, creamos un hijo[*fork()*], y si el valor devuelto es 0 saltamos a *childmsg* y si no a *fuck_100p* (padre).

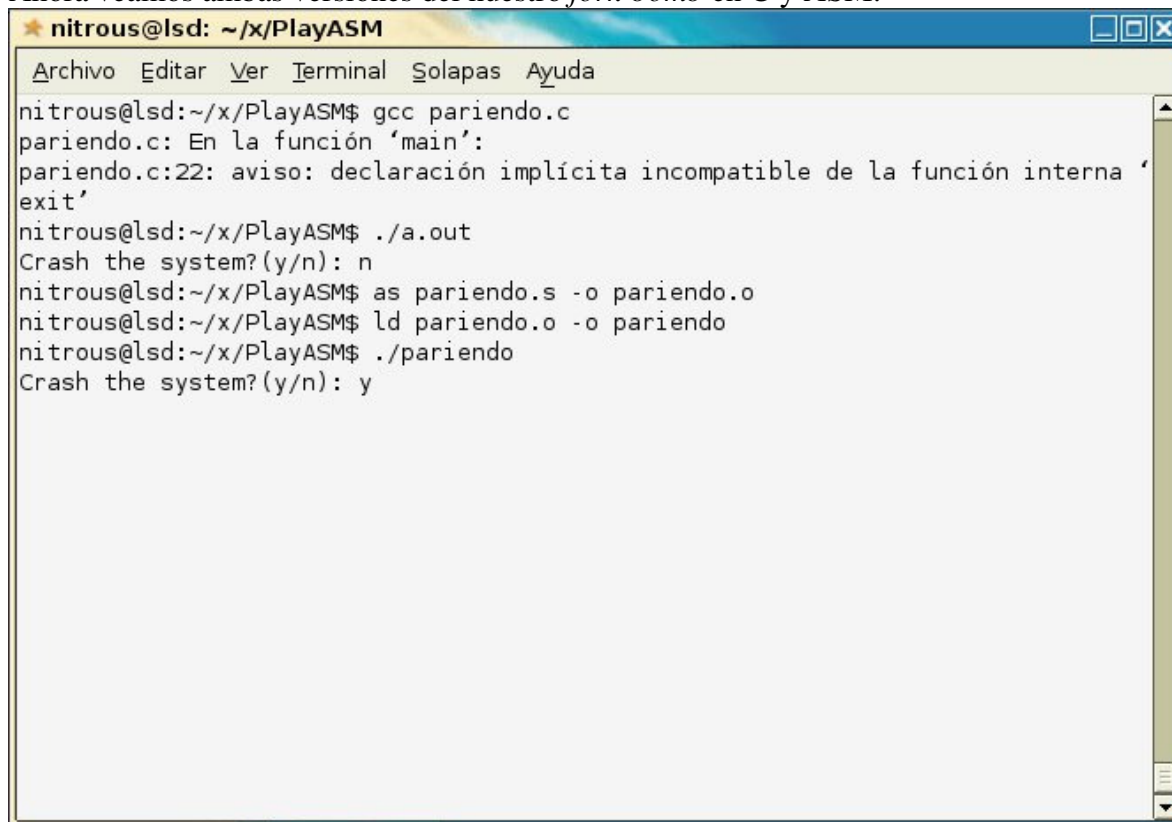
```
fuck_100p:
    if((eax = fork()) == 0)
        goto childmsg;
    goto fuck_100p;

g00d_bye:
    exit(0);
```

Lo único que hace *childmsg* es imprimir[*write()*] su mensaje y de nuevo hacer un salto a *fuck_100p* y con esto se logra hacer un ciclo infinito, un proceso padre teniendo hijos indefinidamente.

```
childmsg:
    write(1, "I'm a child !\n", 14);
    goto fuck_100p;
}
```

Ahora veamos ambas versiones del nuestro *fork-bomb* en C y ASM:



```
nitrous@lsd: ~/x/PlayASM
Archivo  Editar  Ver  Terminal  Solapas  Ayuda
nitrous@lsd:~/x/PlayASM$ gcc pariendo.c
pariendo.c: En la función 'main':
pariendo.c:22: aviso: declaración implícita incompatible de la función interna 'exit'
nitrous@lsd:~/x/PlayASM$ ./a.out
Crash the system?(y/n): n
nitrous@lsd:~/x/PlayASM$ as pariendo.s -o pariendo.o
nitrous@lsd:~/x/PlayASM$ ld pariendo.o -o pariendo
nitrous@lsd:~/x/PlayASM$ ./pariendo
Crash the system?(y/n): y
```

Figura 1 – Compilación y ejecución de *pariendo.c* y *pariendo.s*

3.2.-FLE-ELFcorrupt

Al igual que en el ejemplo anterior, he pasado el código en ensamblador a C el cual explico a continuación:

Declaro dos variables, *MAGIC_YEAH* y *MAGIC_HELL*, la primera tiene el valor correcto de los primeros 4 bytes de un archivo ELF válido, mientras que *MAGIC_HELL* contiene otro valor no válido (creo que en los comentarios del código se entiende esto, y he ahí el significado del nombre del código).

```
int main(int argc, char **argv)
{
    int MAGIC_YEAH = 0x464c457f; //0x7f'ELF'
    int MAGIC_HELL = 0x454c467f; //0x7f'FLE'
```

Se compara el primer argumento dado al programa, si este es NULL se salta a *usage* el cual simplemente imprime[write()] el modo de uso de dicho programa y termina[exit()] su ejecución.

```
    if(argv[1] == NULL)
        goto usage;
```

Abrimos el archivo[open()] y el descriptor devuelto lo asignamos a la variable *eax*. Si el valor devuelto es menor que 0(error), saltamos a la etiqueta *erropen* la cual imprime un mensaje de error y termina la

ejecución del programa, en otro caso, si el valor es mayor que 0 entonces procedemos a verificar si el archivo es un ELF válido (*checkifelf*).

```
int eax, edi, esp;
if((eax = open(argv[1], 2)) < 0)
    goto erropen;
edi = eax;
goto checkifelf;
```

```
erropen:
write(1, "Cannot open() file\n", 19);
goto exit;
```

En este punto, leemos[*read()*] los primeros 4 bytes del archivo y los guardamos en *esp*, luego comparamos dicho valor con *MAGIC_YEAH* y si estos son diferentes saltamos a *notelf*, el cual imprime un error, salta *closefile*(que simplemente cierra[*close()*] el archivo) y termina el programa[*exit()*]. En caso de que el valor leído y *MAGIC_YEAH* sean iguales (ELF válido) simplemente nos queda infectar (*goto infect*).

```
checkifelf:
read(edi, &esp, 4);
if(esp != MAGIC_YEAH)
    goto notelf;
goto infect;
```

```
notelf:
write(1, "This is not an ELF file\n", 24);
goto closefile;
```

```
usage:
write(1, "I need an ELF file as argument\n", 31);
goto exit;
```

En el momento que leímos los primeros 4 bytes del archivo, el puntero al descriptor va avanzando, por lo tanto, si queremos modificar los primeros 4 bytes debemos regresar al principio del archivo. Esto lo llevamos acabo con la llamada *lseek()*. Y para finalizar, escribimos[*write()*] *MAGIC_HELL* al inicio del archivo, cerramos[*close()*] el archivo y terminamos el programa[*exit()*].

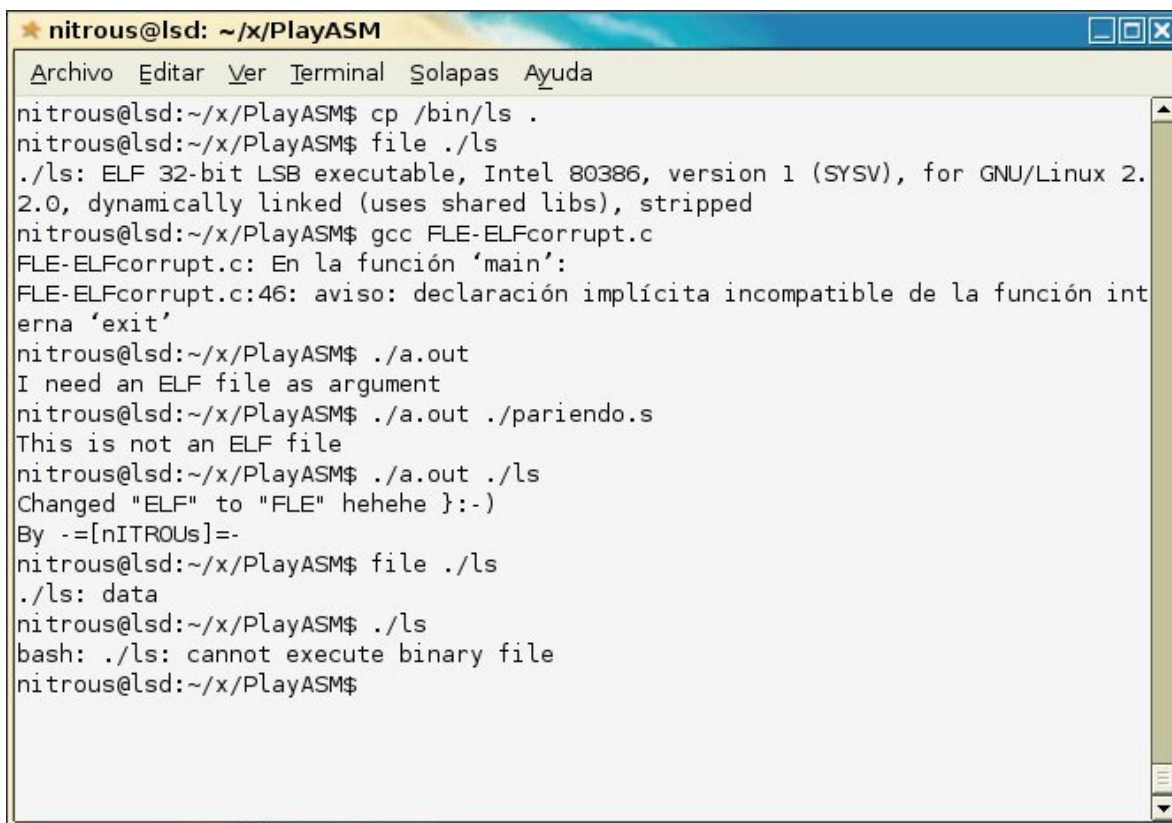
```
infect:
write(1, "Changed \"ELF\" to \"FLE\" hehehe }:-)\nBy ==[nITROUs]==\n", 52);
```

```
lseek(edi, 0, 0);
esp = MAGIC_HELL;
write(edi, &esp, 4);
```

```
closefile:
close(edi);
```

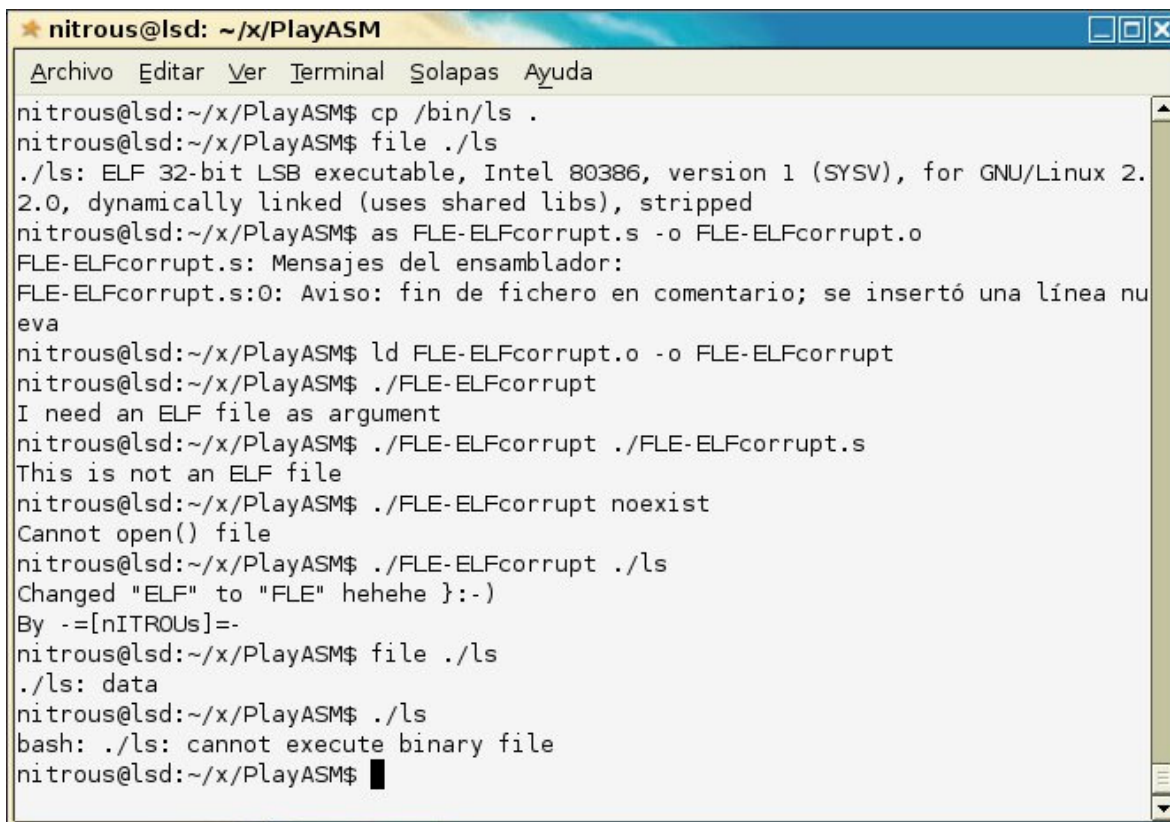
```
exit:
exit(0);
}
```

Aquí vemos la ejecución de ambos programas, en C y en ensamblador:



```
nitrous@lsd: ~/x/PlayASM
Archivo  Editor  Ver  Terminal  Solapas  Ayuda
nitrous@lsd:~/x/PlayASM$ cp /bin/ls .
nitrous@lsd:~/x/PlayASM$ file ./ls
./ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.0, dynamically linked (uses shared libs), stripped
nitrous@lsd:~/x/PlayASM$ gcc FLE-ELFcorrupt.c
FLE-ELFcorrupt.c: En la función 'main':
FLE-ELFcorrupt.c:46: aviso: declaración implícita incompatible de la función interna 'exit'
nitrous@lsd:~/x/PlayASM$ ./a.out
I need an ELF file as argument
nitrous@lsd:~/x/PlayASM$ ./a.out ./pariendo.s
This is not an ELF file
nitrous@lsd:~/x/PlayASM$ ./a.out ./ls
Changed "ELF" to "FLE" hehehe }:-)
By -=[nITROUs]=-
nitrous@lsd:~/x/PlayASM$ file ./ls
./ls: data
nitrous@lsd:~/x/PlayASM$ ./ls
bash: ./ls: cannot execute binary file
nitrous@lsd:~/x/PlayASM$
```

Figura 2 – Compilando y ejecutando *FLE-ELFcorrupt.c*



```
nitrous@lsd: ~/x/PlayASM
Archivo Editar Ver Terminal Solapas Ayuda
nitrous@lsd:~/x/PlayASM$ cp /bin/ls .
nitrous@lsd:~/x/PlayASM$ file ./ls
./ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.
2.0, dynamically linked (uses shared libs), stripped
nitrous@lsd:~/x/PlayASM$ as FLE-ELFcorrupt.s -o FLE-ELFcorrupt.o
FLE-ELFcorrupt.s: Mensajes del ensamblador:
FLE-ELFcorrupt.s:0: Aviso: fin de fichero en comentario; se insertó una línea nu
eva
nitrous@lsd:~/x/PlayASM$ ld FLE-ELFcorrupt.o -o FLE-ELFcorrupt
nitrous@lsd:~/x/PlayASM$ ./FLE-ELFcorrupt
I need an ELF file as argument
nitrous@lsd:~/x/PlayASM$ ./FLE-ELFcorrupt ./FLE-ELFcorrupt.s
This is not an ELF file
nitrous@lsd:~/x/PlayASM$ ./FLE-ELFcorrupt noexist
Cannot open() file
nitrous@lsd:~/x/PlayASM$ ./FLE-ELFcorrupt ./ls
Changed "ELF" to "FLE" hehehe }:-)
By -=[nITROUs]=-
nitrous@lsd:~/x/PlayASM$ file ./ls
./ls: data
nitrous@lsd:~/x/PlayASM$ ./ls
bash: ./ls: cannot execute binary file
nitrous@lsd:~/x/PlayASM$ █
```

Figura 3 – Compilando y ejecutando *FLE-ELFcorrupt.s*

4.-CONCLUSION

Y bien, como hemos visto no es tan difícil hacer programas simples como estos, tan solo es leer el manual de cada syscall y practicar.

Espero se hayan comprendido los códigos en C, así que será fácil comprender los códigos en ASM. He puesto varios comentarios en los programas en ensamblador, propios del lenguaje de programación.

Gracias

5.-REFERENCIAS

- [1] Francisco M. Márquez. (2004). *UNIX – Programación Avanzada* (3ª Edición). Madrid, España: ALFAOMEGA.
- [2] *Fork-bomb* - http://en.wikipedia.org/wiki/Fork_bomb
- [3] TIS Committee. (1995). *Executable and Linking Format(ELF) Specification* (Version 1.2).

APÉNDICE A – CODIGOS

A.1.-*pariendo.c*

```
#include<stdio.h>

main(){
    write(1, "Crash the system?(y/n): ", 24);

    int esp, eax;
    char cl;

    read(1, &esp, 4);
    cl = (char) esp;

    if(cl == 'y' || cl == 'Y')
        goto fuck_100p;
    goto g00d_bye;

fuck_100p:
    if((eax = fork()) == 0)
        goto childmsg;
    goto fuck_100p;

g00d_bye:
    exit(0);

childmsg:
    write(1, "I'm a child !\n", 14);
    goto fuck_100p;
}
```

A.2.-*pariendo.s*

```
# I'm bored... Pariendo...Hijos [fork()]s xD
# This is a Mother with least 1 million of childs inside =>
# ... should be enough to consume the system resources
#
# Wait a few seconds and try to launch any program ;)
#
# $as pariendo.s -o pariendo.o
# $ld pariendo.o -o pariendo
# $./pariendo
#
# nitrous[at]danitrous[dot]org
# 13/Jul/2005

# *-~*-~* DATA SECTION *-~*-~*
.section .data
CHILD:      .ascii      "I'm a child !\n"
CONTI:      .ascii      "Crash the system?(y/n): "
             .equ       STDIN, 0
             .equ       STDOUT, 1
```

```

.equ SYS_EXIT, 1
.equ SYS_FORK, 2
.equ SYS_READ, 3
.equ SYS_WRITE, 4
.equ OK_MIN, 'y'
.equ OK_MAY, 'Y'

# -*-*-* TEXT SECTION *-*-*-
.section .text
.globl _start
_start:
    xorl %eax, %eax
    xorl %ebx, %ebx
    xorl %ecx, %ecx
    xorl %edx, %edx

question:
    movb $SYS_WRITE, %al
    movb $STDOUT, %bl
    movl $CONTI, %ecx
    movb $0x18, %dl # CONTI string length
    int $0x80

    movb $SYS_READ, %al
    movb $STDIN, %bl
    subl $0x4, %esp
    movl %esp, %ecx
    movb $0x4, %dl
    int $0x80

    movb (%esp), %cl
    addl $0x4, %esp
    cmpb $OK_MIN, %cl
    je fuck_l00p
    cmpb $OK_MAY, %cl
    je fuck_l00p

    jmp g00d_bye

fuck_l00p:
    xorl %eax, %eax # CLEAR THE LATEST RETURN OF fork()
    movb $SYS_FORK, %al
    int $0x80
    cmpl $0x0, %eax
    je childmsg # CHILD PROCESS
    jne fuck_l00p # PARENT PROCESS

g00d_bye:
    movb $SYS_EXIT, %al
    int $0x80

childmsg:
    movb $SYS_WRITE, %al
    movb $STDOUT, %bl
    movl $CHILD, %ecx
    movb $0xe, %dl # CHILD string length

```

```
int    $0x80
jmp    fuck_100p
```

A.3.-FLE-ELFcorrupt.c

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

```
int main(int argc, char **argv)
{
```

```
    int MAGIC_YEAH = 0x464c457f;
    int MAGIC_HELL = 0x454c467f;
```

```
    if(argv[1] == NULL)
        goto usage;
```

```
    int eax, edi, esp;
    if((eax = open(argv[1], 2)) < 0)
        goto erropen;
    edi = eax;
    goto checkifelf;
```

erropen:

```
    write(1, "Cannot open() file\n", 19);
    goto exit;
```

checkifelf:

```
    read(edi, &esp, 4);
    if(esp != MAGIC_YEAH)
        goto notelf;
    goto infect;
```

notelf:

```
    write(1, "This is not an ELF file\n", 24);
    goto closefile;
```

usage:

```
    write(1, "I need an ELF file as argument\n", 31);
    goto exit;
```

infect:

```
    write(1, "Changed \"ELF\" to \"FLE\" hehehe }:-)\nBy ==[nITROUs]=-\", 52);
```

```
    lseek(edi, 0, 0);
    esp = MAGIC_HELL;
    write(edi, &esp, 4);
```

closefile:

```
    close(edi);
```

```
exit:
    exit(0);
}
```

A.4.-FLE-ELFcorrupt.s

```
#####
#
# FLE-ELFcorrupt.s
#
# Just a lame ELF crasher... It replace the magic #
# number 0x7f'ELF' by 0x7f'FLE' and obviously the #
# binary goes to hell.
#
# $as FLE-ELFcorrupt.s -o FLE-ELFcorrupt.o
# $ld FLE-ELFcorrupt.o -o FLE-ELFcorrupt
# $./anyelf
# Hello World
# $./FLE-ELFcorrupt ./anyelf
# $./anyelf
# Cannot execute
#
# nitrous[at]conthackto[dot]com[dot]mx
# 29/11/2005
#####
```

```
.section .data
    #GLOBAL VARS
    .equ SDTIN, 0
    .equ STDOUT, 1
    .equ STDERR, 2
    .equ SYS_EXIT, 1
    .equ SYS_READ, 3
    .equ SYS_WRITE, 4
    .equ SYS_OPEN, 5
    .equ SYS_CLOSE, 6
    .equ SYS_LSEEK, 19
    .equ O_RDWR, 2
    .equ SEEK_SET, 0
    .equ MAGIC_YEAH, 0x464c457f #0x7f 'ELF'
    .equ MAGIC_HELL, 0x454c467f #0x7f 'FLE'
    .equ NULL, 0x00000000
```

```
NOARG:
    .ascii      "I need an ELF file as argument\n"
LENNOARG = . - NOARG
```

```
ERROPEN:
    .ascii      "Cannot open() file\n"
LENERROPEN = . - ERROPEN
```

```
NOELF:
    .ascii      "This is not an ELF file\n"
LENNOELF = . - NOELF
```

```
INF:
```

```

        .ascii      "Changed \"ELF\" to \"FLE\" hehehe }:-)\nBy -
=[nITROUs]=-\n"
LENINF = . - INF

.section .text
.globl _start
_start:
    #CLEAR REGISTERS
    xorl %eax, %eax
    xorl %ebx, %ebx
    xorl %ecx, %ecx
    xorl %edx, %edx
    xorl %esi, %esi
    xorl %edi, %edi

    movl 8(%esp), %esi    # %esi = argv[1]
    cmpl $NULL, %esi    # if(argv[1] == NULL) { goto usage; }
    je   usage #jump equal(%esi == NULL) to usage

    jmp  openfile # else jump to openfile

usage:
    movb $SYS_WRITE, %al
    movb $STDERR, %bl
    movl $NOARG, %ecx
    movl $LENNOARG, %edx
    int  $0x80 # write(1, NOARG, LENNOARG);

    jmp  exit

openfile:
    xorl %eax, %eax
    movb $SYS_OPEN, %al
    movl %esi, %ebx
    movb $O_RDWR, %cl
    int  $0x80 # %eax = open(argv[1], 2);

    movl %eax, %edi    # %edi = %eax //returned file descriptor

    cmpl $0x00, %edi
    jl   erropen # if(%edi < 0) { goto erropen; }

    jmp  checkifelf # else jump to checkifelf

erropen:
    xorl %eax, %eax
    xorl %ebx, %ebx

    movb $SYS_WRITE, %al
    movb $STDERR, %bl
    movl $ERROPEN, %ecx
    movl $LENERROPEN, %edx
    int  $0x80 # write(1, ERROPEN, LENERROPEN);

    jmp  exit

```

```

checkifelf:
    movb  $SYS_READ, %al
    movl  %edi, %ebx
    movl  %esp, %ecx
    movl  $0x4, %edx
    int   $0x80 # read(%edi, (%esp), 4);

    cmpl  $MAGIC_YEAH, (%esp)
    jne   notelf # if(esp != MAGIC_YEAH){ goto notelf;}

    jmp   infect # else jump to infect

notelf:
    movb  $SYS_WRITE, %al
    movb  $STDERR, %bl
    movl  $NOELF, %ecx
    movl  $LENNOELF, %edx
    int   $0x80 # write(1, NOELF, LENNOELF);

    jmp   closefile

infect:
    movb  $SYS_WRITE, %al
    movb  $STDOUT, %bl
    movl  $INF, %ecx
    movl  $LENINF, %edx
    int   $0x80 # write(1, INF, LENINF);

    movb  $SYS_LSEEK, %al
    movl  %edi, %ebx
    xorl  %ecx, %ecx
    movl  $SEEK_SET, %edx
    int   $0x80 # lseek(%edi, 0, 0);

    movb  $SYS_WRITE, %al
    pushl $MAGIC_HELL
    movl  %esp, %ecx
    movl  $0x4, %edx
    int   $0x80 # write(%edi, (%esp), 4);

closefile:
    movb  $SYS_CLOSE, %al
    xchgl %edi, %ebx
    int   $0x80 # close(%edi);

exit:
    movb  $SYS_EXIT, %al
    xorl  %ebx, %ebx
    int   $0x80 # exit(0);

```