UNLEARNING SECURITY

Introducción a la ingeniería inversa x86

http://unlearningsecurity.blogspot.com

Daniel Romero Pérez

unlearnsecurity@gmail.com

Marzo del 2012

** Recopilación de entregas realizadas en http://unlearningsecurity.blogspot.com*

ÍNDICE

INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE I) 2
INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE II) 5
INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE III) 8
INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE IV) 11
INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE V) 14

INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE I)

Una de las temáticas que más me han llamado la atención y por varios motivos menos e tocado el mundo de la seguridad de la información, son la ingeniería inversa y el desarrollo de exploits. Me parece una tarea realmente interesante y con una gran cantidad de profesionales detrás de ella. Por ese y algún que otro motivo, he decidido realizar una serie de entradas en relación a dichos temas y así poder descubrir un poco como funciona todo este mundo.

Teniendo en cuenta mi desconocimiento del lenguaje ensamblador y de la arquitectura x86 empezaremos por lo más fácil. Así que iremos creando pequeños códigos en lenguaje C e interpretando en ensamblador que se está haciendo.

Pero antes de meternos de lleno, recordaremos **muy** por encima algunos registros básicos, las instrucciones principales y el funcionamiento de la pila.

Registros

Utilizados para facilitar la tarea en el procesado de las instrucciones, cómo para almacenar datos que se utilizaran posteriormente por las mismas. Estos son alguno de los registros básicos que existen:

- ***EAX (Accumulator register):*** Utilizado tanto para realizar cálculos, cómo para el almacenamiento de valores de retorno en "calls".
- ***EDX (Data register):*** Extensión de EAX, utilizada para el almacenamiento de datos en cálculos más complejos.
- ***ECX (Count register):*** Utilizado en funciones que necesiten de contadores, como por ejemplo bucles.
- ***EBX (Base register):*** Se suele utilizar para apuntar a datos situados en la memoria.
- ***ESI (Source index):*** Utilizado para la lectura de datos.
- ***EDI (Destination index):*** Utilizado para la escritura de datos.
- ***ESP (Stack pointer):*** Apunta a la cima de la pila “stack”.
- ***EBP (Base pointer:*** Apunta a la base de la pila “stack”.

Instrucciones

Son acciones predefinidas en el lenguaje ensamblador. Algunas de las más habituales de ver son:

- **Push:** Guarda el valor en la pila.
- **Pop:** Recupera valor de la pila.
- **Mov (dst, src):** Copia el operador “src” en el operador “dst”.
- **Lea (reg, src):** Copia una dirección de memoria en el registro destino (ej: EAX).
- **Add (o1, o2):** Suma los dos operadores y los almacena en el operador uno.
- **Sub (o1, o2):** Resta el valor del segundo operador sobre el primero y lo almacena en el primer operador.
- **Inc:** Incrementa en 1 el valor del operador indicado.
- **Dec:** Decrementa en 1 el valor del operador indicado.
- **And:** El resultado es 1 si los dos operadores son iguales, y 0 en cualquier otro caso.

- **Or:** El resultado es 1 si uno o los dos operadores es 1, y 0 en cualquier otro caso.
- **Cmp:** Compara dos operadores.
- **Jmp:** Salta a la dirección indicada.
- **Call:** Llama/Salta a la dirección/función indicada.
- **Nop:** Not Operation.

PILA (Stack)

La PILA o *Stack* es un conjunto de direcciones de memoria encargadas de almacena información de llamadas a funciones, variables locales, direcciones de retorno a funciones anteriores, entre otras tareas. La PILA es dinámica, por lo que va cambiando su tamaño dependiendo de la función a la cual se encuentre asociada, dispone de una estructura “First In, Last Out”, por lo que lo último que entra será lo primero en salir, delimitada siempre por su cima (ESP) y por su base (EBP).

Para entender correctamente el funcionamiento de la PILA vamos a ver un ejemplo. Imaginemos que disponemos de dos funciones “main()” e “imprimir_dato()”, donde la segunda se llamada dentro del código de la primera.

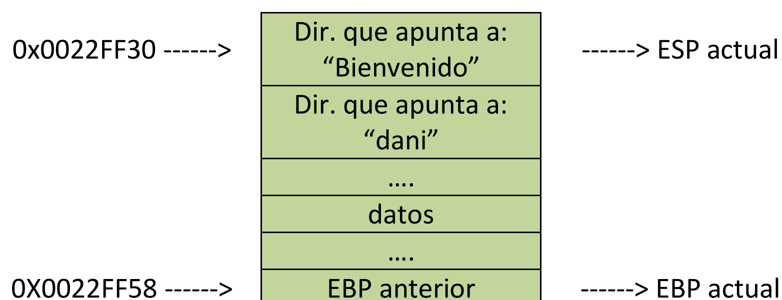
FUNCIÓN 1:

```
int main(int argc,char **argv)
{
    char nombre[4] = "dani";

    imprimir_dato();

    printf("Bienvenido %s!\n", nombre);
}
```

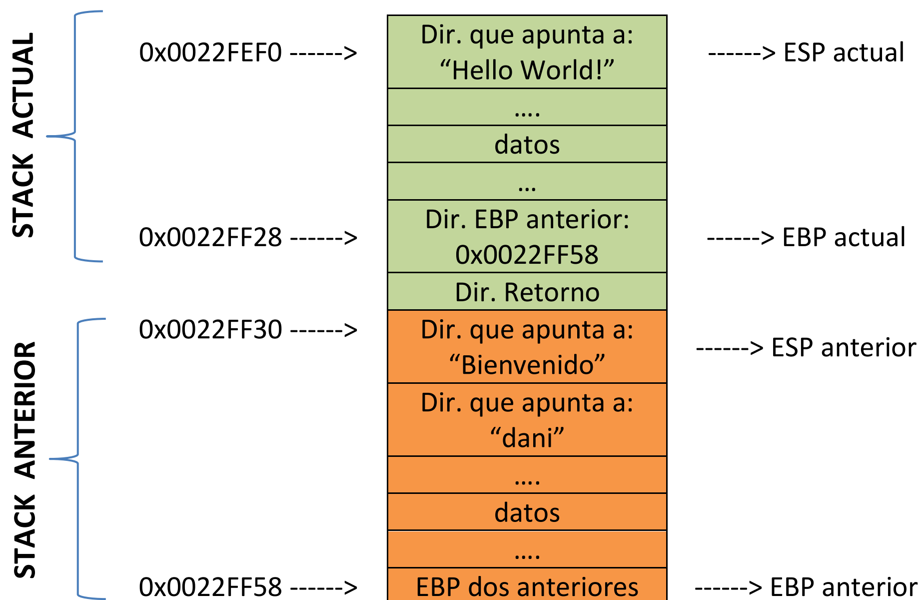
FUNCIÓN 2:

```
void imprimir_dato()
{
    char dato[12] = "Hello World!";
    printf("%s\n", dato);
}
```

Vamos a ver como queda la *Stack* justo antes de realizar la llamada a **printf** de la “función 1”.



Ahora vamos a visualizar el estado de la pila justo antes de la llamada al **printf** de la “función 2”.



Si os fijáis, la *Stack* a medida que va habiendo funciones dentro de funciones, se van apilando de forma decreciente la nueva PILA sobre la anterior, dejando entre medias la famosa **"dirección de retorno"** encargada en este caso de volver la aplicación al punto exacto de la "función 1" después de haber llamado a "función 2".

Por lo tanto, podemos concretar que cada función que se esté ejecutando en una aplicación tendrá asociada su "propia" *Stack.*

Existen multitud de registros e instrucciones más, simplemente he nombrado algunas de las que más vamos a ver. A medida que vayamos viendo nuevas las iremos comentando y explicando para poder entender correctamente el código.

INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE II)

Continuando con la serie de introducción a la ingeniería inversa, y una vez visto un pequeño resumen de algunos de los registros, comentado algunas de las instrucciones más vistas y un resumen de cómo funciona la *Stack*, pasamos a ver como podemos interpretar código en ensamblador y no perdernos en el intento.

Vamos a ver unos cuantos ejemplos de códigos en lenguaje C y su equivalencia en lenguaje ensamblador. *(Para el desensamblado de la aplicación podéis utilizar cualquiera de las muchísimas herramientas que existen:* ***OllyDbg, IDA Pro, Immunity Debugger, gdb, radare,*** *etcétera,* en futuros posts haré una pequeña mención a cada una de ellas detallando sus comandos y funciones básicas).

1 - Hello World

Comenzaremos con el típico "Hello World".

```c
#include <stdio.h>

int main(int argc,char **argv)
{
    printf("Hello World\n");
}
```

```
.text:004013C0  push    ebp
.text:004013C1  mov     ebp, esp
.text:004013C3  and     esp, 0FFFFFFF0h
.text:004013C6  sub     esp, 10h
.text:004013C9  call    sub_401A00
.text:004013CE  mov     dword ptr [esp], 403064h
.text:004013D5  call    puts
.text:004013DA  leave
.text:004013DB  retn

.rdata:00403064 aHelloWorld db 'Hello World',0
```

Tal y como se puede observar, el código es bastante simple de entender.

- **0x004013C6:** Hasta dicha dirección nos encontramos con el prologo de la función (inicialización de la misma).
- **0x004013C9:** Nos encontramos con la instrucción **CALL** que seguramente la habrá introducido el compilador para llamar a vete a saber tu qué ;)
- **0x004013CE:** Se puede observar como se copia (**MOV**) el valor **0x00403064** asociada a la cadena "Hello World" en **ESP** (cima de la pila), por lo que ya tenemos en la PILA el valor que necesitamos.
- **0x004013D5:** Se llame a la función **PUTS,** la cual cogerá de la cima de la pila el valor que va a imprimir por pantalla ("Hello World").
- **0x004013DA:** Tenemos el epílogo, donde se restaura la pila y retornamos a la función anterior.

2 - Hello World (Argumentos)

Pasamos a ver como se manejan los argumentos que le pasamos a una función.

```
#include <stdio.h>

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

```
.text:004013C0 push    ebp
.text:004013C1 mov     ebp, esp
.text:004013C3 and     esp, 0FFFFFFF0h
.text:004013C6 sub     esp, 10h
.text:004013C9 call    sub_401A00
.text:004013CE mov     eax, [ebp+arg_4]
.text:004013D1 add     eax, 4
.text:004013D4 mov     eax, [eax]
.text:004013D6 mov     [esp], eax
.text:004013D9 call    puts
.text:004013DE leave
.text:004013DF retn
```

Ya que el código es similar al anterior pasaremos a detallar lo más relevante.

- **0x004913CE y 0x004913D1:** Copia el valor de "EBP+arg_4" que equivale a "argv" en **EAX,** acto seguido incrementa el valor de **EAX** en 4 para así poder acceder a "argv[1]" donde se encuentra la dirección que apunta al valor que se ha pasado por argumento a la aplicación.
- **0x004913D4, 0x004913D6 y 0x004913D9**: Se copia el contenido de la dirección almacenada en **[EAX]** a **EAX**. Y para finalizar copiamos **EAX** en la cima de la pila para que la función **PUTS** pueda imprimir por pantalla el texto introducido.

Aunque parezca lioso al leerlo, os recomiendo *debuggear* paso a paso la aplicación fijándoos bien en como se van rellenando los registros y la pila.

3 - Función FOR

Vamos a ver como podemos detectar que se está realizando un bucle en nuestro código.

```
#include <stdio.h>

int main(int argc,char **argv)
{
    char nombre[6] = "daniel";
    int i;

    for(i = 0; i < 10; i++)
    {
        printf("Hola %s!\n", nombre);
    }
}
```

```
1
.text:004013C0 push    ebp
.text:004013C1 mov     ebp, esp
.text:004013C3 and     esp, 0FFFFFFF0h
.text:004013C6 sub     esp, 20h
.text:004013C9 call    sub_401A30
.text:004013CE mov     dword ptr [esp+22], 696E6164h
.text:004013D6 mov     word ptr [esp+26], 6C65h
.text:004013DD mov     dword ptr [esp+28], 0
.text:004013E5 jmp     short loc_4013FF
.text:004013E7 ; ---------------------------------
.text:004013E7
2
.text:004013E7 loc_4013E7:
.text:004013E7 lea     eax, [esp+22]
.text:004013EB mov     [esp+4], eax
.text:004013EF mov     dword ptr [esp], offset ; "Hola %s!\n"
.text:004013F6 call    printf
.text:004013FB inc     dword ptr [esp+28]
.text:004013FF
3
.text:004013FF loc_4013FF:
.text:004013FF cmp     dword ptr [esp+28], 9
.text:00401404 jle     short loc_4013E7
4
.text:00401406 leave
.text:00401407 retn
.text:00401407 sub_4013C0 endp
```

Si os fijáis, podemos definir claramente cuatro partes de código que realizan funciones distintas.

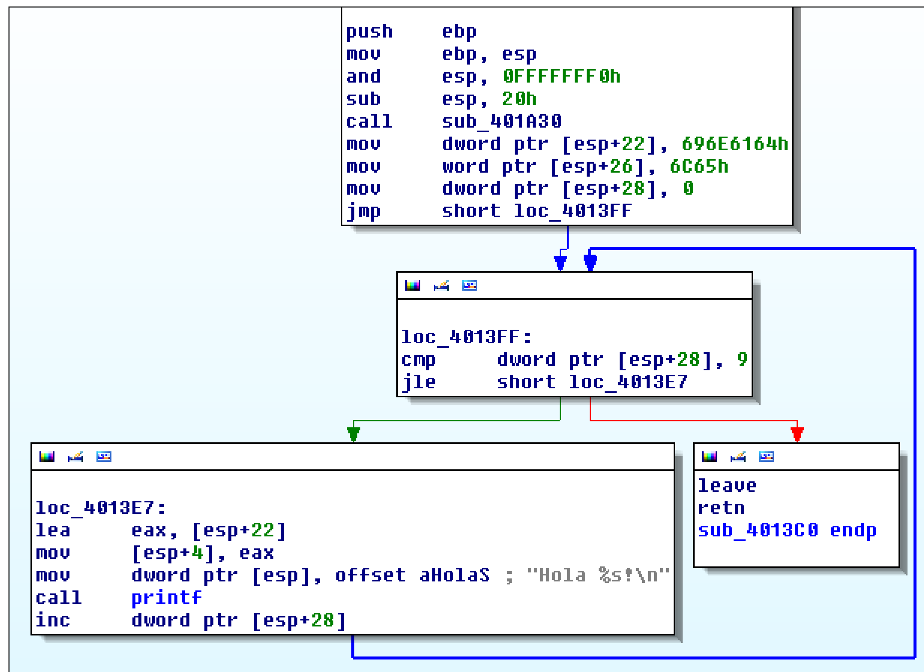
1. Prologo y declaración de variables.
2. **(loc_4013E7)**: Encargado de preparar los parámetros y llamar a la función **printf**. De esta parte podemos destacar la instrucción **INC**, la cual incrementará en uno el valor indicado **[esp+28]**.
3. **(loc_4013FF)**: Parte encargada de verificar el estado del bucle. En esta parte nos encontramos con una nueva instrucción **JLE** (**J**ump if **L**ess than or **E**qual) saltar si es menor o igual, que junto a la instrucción **CMP** anterior se forma la estructura de

comparar el valor **[esp+28]** con **9**, si el resultado es menor o igual la instrucción **JLE** nos hará saltar a **(loc_4013E7),** de lo contrario continuará con la zona 4.

4. Epílogo de la función encargado de restaura la pila y retornamos a la función anterior.

De este modo, ya podremos identificar un bucle en código ensamblador.

Os dejo también el ***flowgraph*** que nos ofrece IDA PRO, el cual de forma gráfica nos ayuda a interpretar mucho más rápido que función está realizando el código analizado.



En la entrada anterior vimos algunos ejemplos de código en C y su equivalente en ensamblador. Hoy continuaremos con algunos ejemplos más que nos permitan entender más como funcionan los registros y ensamblador.

Pero antes de comenzar y ya que en el anterior post vimos un ejemplo de salto **(JLE)** debido a un bucle vamos a profundizar un poco más en el asunto.

Si recordáis la instrucción **JLE** (**J**ump if **L**ess than or **E**qual) realizaba el salto a la dirección indicada si la condición se cumplía. *¿Pero como sabe si esta se cumple o no?* Aquí es donde entra el registro de estado y sus ***FLAGS,*** el cual sirve para indicar el estado actual de la máquina y el resultado de algún procesamiento mediante bits de estado, entre los cuales podemos encontrar los siguientes:

- ***OF (overflow):*** Indica desbordamiento del bit de mayor orden después de una operación aritmética de números con signo (1=existe overflow; 0=no existe overflow).
- ***SF (Sign):*** Contiene el signo resultante de una operación aritmética (0=positivo; 1=negativo).
- ***PF (paridad):*** Indica si el número de bits 1, del byte menos significativos de una operación, es par (0=número de bits 1 es impar; 1=número de bits 1 es par).
- ***CF (Carry):*** Contiene el acarreo del bit de mayor orden después de una operación aritmética.
- ***ZF (Zero):*** Indica el resultado de una operación aritmética o de comparación (0=resultado diferente de cero; 1=resultado igual a cero).

Como ya habréis intuido, los flags **ZF, SF y OF** serán los más utilizados en comparadores y saltos. Retornamos al ejemplo del post anterior donde teníamos algo similar al siguiente código:

```
int main(int argc,char **argv)
{
    int i;
    for (i = 0; i < 10; i++ )
        printf("Hola!!\n");
}
```

```
.text:004013D8 loc_4013D8:
.text:004013D8 mov     [esp+20h+var_20], offset ; "Hola!!"
.text:004013DF call    puts
.text:004013E4 inc     [esp+20h+var_4]
.text:004013E8
.text:004013E8 loc_4013E8:
.text:004013E8 cmp     dword ptr [esp+28], 9
.text:004013ED jle     short loc_4013D8
.text:004013EF leave
```

Teniendo en cuenta que se está ejecutando la instrucción **JLE**, nos interesa saber con que registros de estado trabaja la misma (http://faydoc.tripod.com/cpu/jle.htm).

0F 8E	JLE	Jump near if less or equal (ZF=1 or SF<>OF)

Como se ha podido observar, la instrucción **JLE** trabaja con los registros **ZF, SF y OF**, por lo que mientras la condición se cumpla nos encontraremos dentro del bucle. Para verlo todo más claro, vamos a ver la secuencia en ensamblador y como afecta al registro de estado.

Esp+28	ZF	SF	OF	Condición
0	0	1	0	VERDADERA
1	0	1	0	VERDADERA

2	0	1	0	VERDADERA
..	..	..	..	VERDADERA
8	0	1	0	VERDADERA
9	1	0	0	VERDADERA
10	0	0	0	FALSA

Como veis, hasta que **NO** se ha cumplido la sentencia (**ZF=1 o SF <> OF**) no se ha salido del bucle. De este modo en la vuelta diez el bucle dejará de ejecutarse.

Una vez entendido como funcionan algunos de los registros de estado, continuamos con alguna de las equivalencias entre lenguaje C y ensamblador.

4 - Operaciones matemáticas

Vamos a ver como realiza ensamblador algunos cálculos matemáticos básicos.

```
int main(int argc,char **argv)
{
    int edad1 = 8, edad2 = 2;

    int suma = edad1 + edad2;
    printf("La suma es: %i\n", suma);

    int resta = edad1 - edad2;
    printf("La resta es: %i\n", resta);

    int mult = edad1 * edad2;
    printf("La multiplicacion es: %i\n", mult);

    int div = edad1 / edad2;
    printf("La division es: %i\n", div);
}
```

Tal y como se puede observar en la captura anterior, se trata de un simple código que calcula la suma, resta, multiplicación y división de dos variables. Voy a dividir el código en cuatro partes para visualizarlo con más facilidad.

Empezamos con una simple suma:

```
.text:004013CE  mov     dword ptr [esp+44], 8
.text:004013D6  mov     dword ptr [esp+40], 2
.text:004013DE  mov     eax, [esp+40]
.text:004013E2  mov     edx, [esp+44]
.text:004013E6  lea     eax, [edx+eax]
.text:004013E9  mov     [esp+36], eax
.text:004013ED  mov     eax, [esp+36]
.text:004013F1  mov     [esp+4], eax
.text:004013F5  mov     dword ptr [esp], offset aLaSumaEsI ; "La suma es: %i\n"
.text:004013FC  call    printf
```

Utilizando la instrucción **LEA** carga el contenido de EAX+EDX en EAX, por lo que ya tenemos nuestro resultado en un registro para posteriormente imprimirlo por pantalla.

En el siguiente código se encuentra la equivalencia a la resta.

```
.text:00401401  mov     eax, [esp+40]
.text:00401405  mov     edx, [esp+44]
.text:00401409  mov     ecx, edx
.text:0040140B  sub     ecx, eax
.text:0040140D  mov     eax, ecx
.text:0040140F  mov     [esp+32], eax
.text:00401413  mov     eax, [esp+32]
.text:00401417  mov     [esp+30h+var_2C], eax
.text:0040141B  mov     [esp+30h+var_30], offset aLaRestaEsI ; "La resta es: %i\n"
.text:00401422  call    printf
```

En este caso se utiliza la instrucción **SUB** para restar el contenido de EAX a ECX, y almacenarlo nuevamente en ECX.

En el siguiente código se encuentra la equivalencia a la multiplicación.

```
.text:00401427 mov     eax, [esp+44]
.text:0040142B imul    eax, [esp+40]
.text:00401430 mov     [esp+28], eax
.text:00401434 mov     eax, [esp+28]
.text:00401438 mov     [esp+4], eax
.text:0040143C mov     dword ptr [esp], offset aLaMultiplicaci ; "La multiplicacion es: %i\n"
.text:00401443 call    printf
```

Para realizar la multiplicación de dos variables, se utiliza la instrucción **IMUL**, el cual multiplica el primer valor por el segundo, almacenando el resultado de la operación en el primer valor.

En el siguiente código se encuentra la equivalencia a la división.

```
.text:00401448 mov     eax, [esp+44]
.text:0040144C cdq
.text:0040144D idiv    dword ptr [esp+40]
.text:00401451 mov     [esp+24], eax
.text:00401455 mov     eax, [esp+24]
.text:00401459 mov     [esp+4], eax
.text:0040145D mov     dword ptr [esp], offset aLaDivisionEsI ; "La division es: %i\n"
.text:00401464 call    printf
```

Para realizar la división de dos variables son necesarias dos instrucciones:

- **CDQ** se encarga de convertir el valor de 32 bits almacenado en EAX, en uno de 64 bits almacenado en EDX:EAX.
- **IDIV** se encarga de dividir el valor almacenado en EAX con el valor que se le pasa a la instrucción.

INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE IV)

Tras aprender qué es y de que está compuesto el registro de estado y ver algunas operaciones matemáticas, hoy continuaremos con alguno de los códigos que habitualmente nos encontraremos al realizar ingeniería inversa sobre un binario.

Vamos a analizar el siguiente código:

```c
int main(int argc,char **argv)
{
    int valor = 10;
    int valor2 = 5;

    if (valor > 9 && valor2 < 6)
        printf("Todo correcto!!");
    else
        printf("Todo incorrecto!!");
}
```

```
.text:004013D6 mov     dword ptr [esp+24], 5
.text:004013DE cmp     dword ptr [esp+28], 9
.text:004013E3 jle     short loc 4013FA
.text:004013E5 cmp     dword ptr [esp+24], 5
.text:004013EA jg      short loc_4013FA
.text:004013EC mov     dword ptr [esp], offset ; "Todo correcto!!"
.text:004013F3 call    printf
.text:004013F8 jmp     short locret_401406
.text:004013FA ; ---------------------------------------------------------
.text:004013FA
.text:004013FA loc_4013FA:
.text:004013FA
.text:004013FA mov     dword ptr [esp], offset ; "Todo incorrecto!!"
.text:00401401 call    printf
.text:00401406
.text:00401406 locret_401406:
.text:00401406                 leave
.text:00401407                 retn
.text:00401407 sub_4013C0      endp
```

Si os fijáis este caso es similar al código que analizamos en la segunda entrega de la serie, una de las diferencias claves en este código es la nueva instrucción **JG** *(Jump if Greater)* que podemos ver en la dirección **0x004013EA**. Como se ha podido observar se trata de una instrucción de salto que se efectuará en el caso que la comparación anterior sea mayor al valor indicado.

Debido a que nos encontraremos una gran cantidad de instrucciones de salto cuando se está realizando ingeniería inversa sobre un binario vamos a recordar algunas de ellas y como funcionan.

Instrucción	Descripción	Flags
JA	Jump if above	CF=0 and ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX register is 0	CX=0
JECXZ	Jump if ECX register is 0	ECX=0
JE	Jump if equal	ZF=1
JG	Jump if greater	ZF=0 and SF=OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 or SF<>OF
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=0 and ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0

JNG	Jump if not greater	ZF=1 or SF<>OF
JNGE	Jump if not greater or equal	SF<>OF
JNL	Jump if not less	SF=OF
JNLE	Jump if not less or equal	ZF=0 and SF=OF
JNO	Jump if not overflow	OF=0
JNP	Jump if not parity	PF=0
JNS	Jump if not sign	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if parity odd	PF=0
JS	Jump if sign	SF=1
JZ	Jump if zero	ZF = 1

Como habréis podido observar los saltos no tienen ningún misterio ya que únicamente es necesario acordarse de su función su correspondiente valor en el registro de estado. Por lo que, cuando se cumpla la condición de los *flags* se realizara el salto a la dirección indicada en la instrucción.

Operaciones Lógicas

Ya que vimos en el anterior post como realizar cálculos matemáticos mediante las instrucciones **SUB, IMUL y DIV**, hoy veremos como se realizan cálculos lógicos a nivel de bit a través de las instrucciones **AND, NOT, OR y XOR**.

(AND dst, src) Devuelve un valor verdadero, siempre que los valores **dst** y **src** sean verdaderos.

Src	Operador	Dst		Result
1	**And**	1	=	**1**
1	**And**	0	=	**0**
0	**And**	1	=	**0**
0	**And**	0	=	**0**

(NOT dst) Niega el valor **dst**, y lo almacena en el mismo operador

Operador	Dst		Result
Not	1	=	**0**
Not	0	=	**1**

(OR dst, src) Devuelve un valor verdadero siempre que **dst** o **src** sean verdaderos.

Src	Operador	Dst		Result
1	**OR**	1	=	**1**
1	**OR**	0	=	**1**
0	**OR**	1	=	**1**
0	**OR**	0	=	**0**

(X0R dst, src) Devuelve un valor verdadero si únicamente uno de los valores **src** o **dst** es verdadero.

Src	Operador	Dst		Result
1	**Xor**	1	=	**0**
1	**Xor**	0	=	**1**
0	**Xor**	1	=	**1**
0	**Xor**	0	=	**0**

INTRODUCCIÓN A LA INGENIERÍA INVERSA X86 (PARTE V)

Hoy, para finalizar la serie de entregas de introducción a la ingeniería inversa realizaremos un sencillo ejercicio o *crackme*, en el cual podremos aplicar los conocimientos adquiridos en las anteriores entregas y así poder resolver el ejercicio.

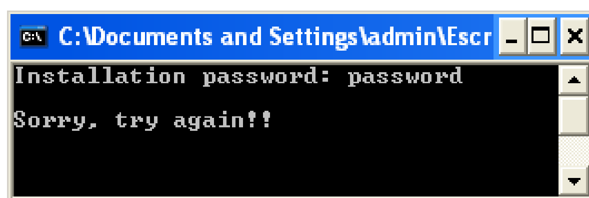
Os dejo las cuatro entradas anteriores de la serie por si queréis repasarlas:

- Introducción a la ingeniería inversa x86 (Parte I)
- Introducción a la ingeniería inversa x86 (Parte II)
- Introducción a la ingeniería inversa x86 (Parte III)
- Introducción a la ingeniería inversa x86 (Parte IV)

El fichero que pretendemos analizar se encuentra disponible en las siguiente url: Ejercicio ingeniería inversa

MD5 (fichero .exe): *c12ca49829f6a340e944936ffb1f2cde*
SHA1 (fichero .exe): *d3f9b9a2c0e8624aae672c3e9a4f1a00e740659b*

Una vez descargado el binario lo ejecutamos y visualizamos como nos pide un password de instalación, el cual mediante ingeniería inversa tendremos que averiguar. (Recalco averiguar porque obviaremos la parte de modificar el ejecutable para saltarnos la comprobación)



Si desensamblamos el binario nos encontrarnos dos funciones importantes (**sub_4013C0** y **sub_401424**), que serán las encargadas de realizar los cálculos y de informar de si el password es el correcto respectivamente.

Analizaremos cada una de las funciones en busca como se genera el password, detallando las instrucciones más importantes y las que nos llevarán a solucionar el reto.

Función **sub_4013C0**: (Voy a dividir en dos la función para facilitar su visualización)

```
.text:004013CE mov     byte ptr [esp+31], 64h
.text:004013D3 mov     dword ptr [esp+24], 5
.text:004013DB mov     dword ptr [esp], "Installation password: "
.text:004013E2 call    printf
.text:004013E7 call    getchar
.text:004013EC mov     [esp+23], al
.text:004013F0 mov     al, [esp+23]
.text:004013F4 mov     dl, [esp+31]
```

- **0x004013CE:** Se almacena en la dirección [esp+31] el valor hexadecimal 64 correspondiente a la letra “d” del abecedario.
- **0x004013D3:** Se almacena en la dirección [esp+24] el valor entero “5”.

- **0x004013E2:** Se llama a la función *printf* con el texto “Installation password: “ en ESP.
- **0x004013E7:** La función getchar() almacena en AL (los primero 8 bits del registro EAX) el carácter introducido por teclado. Esto ya nos da una pista de que datos se están guardando para posteriormente trabajar con ellos. En este caso se almacena el primer byte introducido por teclado, el cual se puede asociar al primer número, carácter o símbolo introducido. Por lo que si asignamos como password “hola” la función getchar() únicamente se quedará con el carácter “h”.
- **0x004013F0:** Se almacena el valor introducido por teclado en AL (EAX).
- **0x004013F4:** Se almacena el inicializado anteriormente “d” en DL (EDX).

```
.text:004013F8 xor     eax, edx
.text:004013FA mov     [esp+22], al
.text:004013FE movsx   eax, byte ptr [esp+22]
.text:00401403 mov     [esp+16], eax
.text:00401407 mov     edx, offset sub_401424
.text:0040140C mov     eax, [esp+16]
.text:00401410 mov     [esp+4], eax
.text:00401414 mov     eax, [esp+24]
.text:00401418 mov     [esp], eax
.text:0040141B call    edx ; sub_401424
.text:0040141D call    _getch
.text:00401422 leave
.text:00401423 retn
```

- **0x004013F8:** Se realiza la operación lógica XOR sobre **EAX** y **EDX.**
- **0x00401410:** Se almacena el resultado del XOR en ESP+4
- **0x00401418:** Se almacena la variable inicializada anteriormente “5” en ESP**.**
- **0x0040141B:** Esta instrucción es la última “útil” de la función, en la cual se realiza una llamada a una nueva función, hay que tener en cuenta que en las funciones anteriores se han almacenado valores en la ESP para utilizarlos en esta función.

Hasta aquí ya sabemos que:

- Se almacena el primer byte introducido por teclado
- Se almacena el valor “d”
- Se realiza un XOR de los valores anteriores
- Se llama a la función sub_401424 pasándole el valor resultante del XOR y el valor “5”.

Función **sub_401424**:

```
.text:00401424 push    ebp
.text:00401425 mov     ebp, esp
.text:00401427 sub     esp, 18h
.text:0040142A mov     eax, [ebp+12]
.text:0040142D mov     edx, [ebp+8]
.text:00401430 lea     eax, [edx+eax]
.text:00401433 cmp     eax, 1Bh
.text:00401436 jnz     short loc_401446
.text:00401438 mov     dword ptr [esp], "\nWell Done!!"
.text:0040143F call    printf
.text:00401444 jmp     short locret_401452
.text:00401446 ; ---------------------------------------------
.text:00401446
.text:00401446 loc_401446:
.text:00401446 mov     dword ptr [esp], "\nSorry, try again!!" ;
.text:0040144D call    printf
.text:00401452
.text:00401452 locret_401452:
.text:00401452 leave
.text:00401453 retn
```

- **0x0040142A y 0x0040142D:** Se almacenan los valores pasados anteriormente en EAX y EDX.
- **0x0040141B:** Sumamos EAX y EDX y lo almacenamos en EAX.
- **0x00401433:** Comparamos el resultado (EAX) con el valor hexadecimal 1B, en decimal 27.
- **0x00401436:** Se realiza un salto JNZ a una dirección (mensaje "sorry, try again!!") si la comparación es falsa. Como habréis podido imaginar es justamente en esta comprobación donde nos interesa que la comparación sea cierta.

Hasta aquí sabemos:

- Los valores pasados a esta función sumados deben dar 27 para poder resolver el reto.

Con todo lo recopilado hasta ahora podemos decir que: debemos encontrar un carácter, número o símbolo que al realizar un XOR con el carácter "d" y sumándole el valor 5 el resultado sea 27.

Resumido en una fórmula:

((primer_byte_introducido_por_teclado) **XOR** (carácter_d)) **+** 5 **=** 27

o

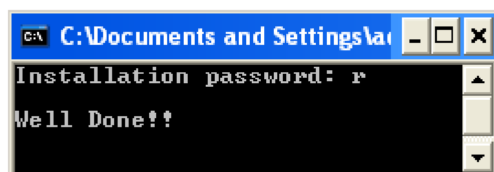
((primer_byte_introducido_por_teclado) **XOR** (carácter_d)) **=** 27 - 5 = 22

Si os fijáis, es necesario trabajar en binario para poder identificar el valor correcto. Al tener un único valor al otro lado de la igualdad, es posible realizar la operación XOR entre los dos valores conocidos "22" y "d" de este modo obtendremos el valor correcto.

Valor	Binario
22	0 0 0 1 0 1 1 0
Carácter "d"	0 1 1 0 0 1 0 0
Resultado	0 1 1 1 0 0 1 0

Y el valor en binario "0 1 1 1 0 0 1 0" corresponde con el valor hexadecimal 72 que a su vez corresponde a la letra "r" del abecedario, la cual será la solución a nuestro retro.

Si realizamos la prueba sobre el ejecutable conseguimos que se nos muestre el mensaje "Well Done!!"



Aquí finaliza la serie de entradas sobre la introducción a la ingeniería inversa en x86, seguiremos añadiendo nuevos contenidos en el blog (http://unlearningsecurity.blogspot.com).

**Os podéis descargar el código en C del reto desde el siguiente enlace:* Código_C_reto