Tutorial de ensamblador para novatos

· Saltos Condicionales

Aquí...

Esta una lista de saltos condicionales divididos **sin signo** – y **con signo** Los valores sin signo solo pueden ser positivos, mientras trabajamos con valores sin signo, el BIT mas alto es el que dice si es positivo o no. Por lo tanto un valor hexadecimal como FFFF equivaldría a 65535 si el valor esta sin signo, y –1 si no lo esta. Hay también una sesión con saltos condicionales que no chequea si el valor tiene signo o no.

Saltos condicionales sin signo

JA	
Salta si superior	
JAE	Salta si superior o igual
JB	Salta si menor
JBE	Salta si menor o igual
JNA	Salta si no es superior, como JBE
JNAE	Salta si no superior o igual (como JB)
JNB	Salta si no es menor (Como JBE también)
JNBE	Salta si no es menor o igual (como JA)

Saltos condicionales con signo

JG	
Salta si es grande	
JGE	Salta si es grande o igual
JL	Salta si es menos
JLE	Salta si es menos o igual
JNG	Salta si no es grande, como JLE
JNGE	Salta si no es grande o igual como JGLE

JNL	Salta si no es inferior, como JGE
JNLE	Salta si no es inferior o igual, como JG

Saltos condicionales, no importa si tienen signo o no.

JZ Salta si es cero	
JE	Salta si es igual, como JZ
JNZ	Salta si no es 0
JNE	Salta si no es igual, como JNZ

Los mas interesantes, que son los que mas necesitaremos son **JZ, JNZ, JA** y **JB**.

Dependiendo de la instrucción previa (CMP en nuestro caso) el **flag 0** es colocado. Por lo tanto, lo que realmente hace JE (o JZ) verificar el flag 0. Si este esta puesto (zeroflag=1) saltara, de lo contrario no lo hará. Esto es bastante importante para los crackers porque ellos pueden modificar el valor del flag zero en caso de que lo necesiten.

· Una guía para la programación en ensamblador

· Registros ·

Registros.

Esta va a ser una palabra que oigas mucho a partir de ahora o uite a lot in the future. Los registros los podemos comparar a las variables de algún lenguaje de alto nivel. Son lugares dentro de la cpu donde se pueden almacenar y manipular números. Hay 4 tipos de registros diferentes, de propósito general, registros de la pila, registros de segmento y registros indexados

---

Registros de propósito general:

Estos registros tienen una capacidad de 16 bits, y son cuatro - **AX,BX,CX** y **DX**. Estos se dividen en dos registros de 8 bits, AX se divide en **AL** (low byte) y **AH** (high byte), y BX se divide en **BL** y **BH**, y así el resto,...un ejemplo:

Digamos que AX=1234. Luego AL=34 (los bytes bajos) y AH=12 (los bytes altos).

En lose 386 y superiores, Hay también registros de 32 bits, que tienen el mismo nombre que los de 16 bits, pero con una E al principio de los nombres (**EAX,EBX,ECX,EDX**).

Registros de la pila :

BP y SP son los dos registros de la pila. Describiremos que es la pila y para que sirve aqui.

Registros de segmentos:

Son 4, 6 en los 386 y superiores:

CS - Code segment. Segmento de Código. Este es el bloque de memoria donde se localiza el código
DS - Data segment. Segmento de Datos. Aquí es donde puedes acceder. Cuando trabajamos con cadenas este es el segmento de fuente a menudo

ES - Extra segment.Segmento extra. Simplemente es un segmento extra que puede ser usado del mismo modo que el anterior.
FS – Otro segmento (solo en 386+), se usa rara vez en cracking
GS – Y otro del 386+ que casi no se usa en cracking.

Registros Indexados:

Estos son los registros del puntero y a menudo son usado en funciones que tienen que ver con cadenas. Son solo 2, y tienen una longitud de 16 bits, o 32 en 386+, con la E, como en el primer caso:

SI - Source Index. Índice fuente. Usados como fuentes de las cadenas

DI - Destination Index.Indice destino, usados como las cadenas de destino

BX también puede ser usado como un registro indexado

Estos registros pueden ser usados junto con los de segmento como un **offset**. Por lo tanto , que quiere decir **DS:SI**? Bien simplemente, que Ds apunta al segmento de datos y SI es el offset del segmento de datos.

· La pila ·

¿Que demonios es esto?

La pila es un lugar en el que puedes almacenar temporalmente los datos. Para almacenarlos se usa el comando **push**,y para acceder el comando **pop** . La pila función del modo LIFO, last input, first output, o lo que es lo mismo ultima entrada, primera salida.

Ejemplos:

```
Push ax
push bx
push cx
...
pop cx
pop bx
pop ax
```

Por lo tanto los registros son sacados en el orden inverso al que los has metido.

El registro **SS** es el registro de la pila y el registro **SP** es el del offset. (Mira aquí para entender que son y como funcionan los registros registros)

---

guía de ensamblador para novatos ·

· Instrucciones mas comunes ·

Aquí...

Describiremos las instrucciones que debes saber para entender lo que se esta haciendo. Esto solo son las instrucciones básicas y si quieres llegar a ser un cracker serio (espero que tu quieras también) lo sea, mejor que aprendas mas sobre ellas

.

MOV destino, origen
Esta instrucción simplemente mueve el valor en una localización de memoria (registro o variable).

```
Ej  : MOV AX, 1234h   ; AX = 1234h
      MOV BX, AX      ; BX = AX
```

Esto movería el valor 1234 hex (4660 dec) al registro AX. Despues el valor de AX seria movido al registro BX. En un lenguaje de alto nivel se haría de este modo (pascal) AX:=$1234; BX:=AX; (pascal notation).

ADD dest, valor
Esto simplemente suma algo al valor del destino.

```
EG : MOV AX, 10h     ; Ax ahora vale 10h
     ADD AX, 10h     ; Ax ahora vale 20h
     ADD AX, 5h      ; Ax ahora vale 25h
```

SUB dest, valor
Resta algo al valor del destino

```
EG : MOV AX, 10h     ; AX vale 10h
     SUB AX, 2h      ; Ax vale 8h
```

INC dest
Incrementa algo (registro, variable o cualquier cosa).

```
EG : MOV AX, 10h     ; AX es 10h
     INC AX          ; Ax es ahora 11h
```

DEC dest
Decrementa algo (registro, variable o cualquiercosa).

```
EG : MOV AX, 10h     ; AX es 10h
     dec AX          ; Ax es ahora Fh
```

CMP origen, destino
Compara el origen con el destino.

```
EG : MOV AX, 10h     ; AX es 10h
     MOV BX, 11h     ; BX es 11h
     CMP AX, BX      ; Compara AX con BX
```

La linea siguiente a CMP AX,BX probablemente sera un salto condicional. Si quisieramos saltar si AX fuera igual a BX, pondriamos un JE (jump if equal) localización (seria un offset) despues de la comparacion.will problaby be a **conditional jump**. Si quisieramos saltar si no lo fuera, pondriamos un JNE despues del CMP.

JMP localización
Salta a otro lugar del código.

```
EG : JMP 200h        ; El programa saltara al offset 200h
```

MOVSB o MOVSW
Mueve (mejor dicho, copia) cualquier byte (MOVSB) o plabras (MOVSW) desde DS:SI a ES:DI. Incrementa SI

```
EG : Vemos que DS:SI apunta a un byte que tiene de valor 5h
     MOVSB           ;  Lleva el byte que apunta DS:SI y lo situa en
                        ES:DI

El byte que apunta ahora ES:DI es 5h
```

Esta instrucción es muy comun en el cracking, cuando una cadena es copiada a otra localización. Las instrucciones son usadas junto con la instrucción REP

LODSB o LODSW
Carga cualquier byte o palabra desde DS:SI y lo pone en AL (LODSB) o AX (LODSW). Incrementa SI.

```
EG : Vamos a ver que DS:SI apunta a una palabra que points to a word
     tiene el valor de EBh
     LODSW       ; Copia la palabra que apunta DS:SI y la coloca en AX

     AX contendra ahora EBh
```

Estas instrucciones se usan normalmente junto a REP

STOSB o STOSW

Coge el valor de AL (STDSB) o AX (STDSW) y lo situa en DS:SI. Incrementa SI.

```
EG : Vamos a ver como AX contiene el valor EBh
     STOSW          ; Copia el valor de AX y lo situa en la palabra que
                      apunta DS:SI.
                    ; DS:SI ahora apunta a ua palabra que contiene EBh
```

también se usa junto con REP

REP

Repite una instrucción el numero de veces que CX señala. Un REP delante de un MOVSB,LODSB o STOSB haria que se repitese esa instruccion.

```
EG : MOV AL,Bh      ; AL contiene bh ahora
     MOV CX,5h      ; CX contaene 5h ahora
     REP STOSB      ; Esto copiaria el valor de AL (5h) en donde apunte
                      DS:SI 5 veces
                    ; e incremenaria SI todas las veces.
```

CALL procedimiento

Llama a un procedimiento, y despues de que el procedimiento termina regresa.

```
EG : CALL 4020      ; Salta al offset 4020 y continua la ejecucion
desde
                      Alli hasta que se llega a un RET.
                    ; Despues continuara  en la siguiente linea.
```

Esto es una llamada cercana. Cuando una llamada cercana es ejecutada solo saltas a un offset diferente. también estan las llamadas lejanas. Se salta a un segmento y a un offset diferentes.

```
EG : CALL 013f:2310     ; Salta al segmento 013f, y el offset apunta
a 2310.
```

guía de ensamblador para novatos ·

· A simple Patcher ·

Bien, aquí hay...

Un parche hecho en ASM muy simple. Espero que puedas aprender algo de aquí.

```
assume cs:code,ds:code
code    segment
        org 100h
start:

mov ah,09h          ; escribe algo en la pantalla
lea dx,text1
int 21h

mov ah,09h          ; escribe algo mas
lea dx,text2
int 21h

mov ah,03dh          ; abre el archivo
mov al,02
lea dx,file
int 21h

jnc filefound        ; encuentra el archivo?
mov ah,09h           ; como no lo hace escribe algo en la pantalla
lea dx,text3
int 21h
jmp ready            ; y sale del programa

filefound:           ; Si, lo ha encontrado
mov handle,ax         ; Vamos a salvar el filehandle, como manejar el
                    archivo
mov ah,42h           ; Necesitamos mover el puntero
xor al,al
xor cx,cx
mov bx,handle
mov dx,02F9Ch         ; Mira en still confused para ver como
                    obtengo este valor, al final
```

```
int 21h

mov ah,40h          ; Vamos a parchearlo
mov bx,handle
mov cx,1            ; Solo queremos escribir un byte
lea dx,value        ; este es el valor que queremos escribir
int 21h

mov ah,42h          ; Ok, mueve el puntero de nuevo
mov al,0
mov bx,handle
xor cx,cx
mov dx,030EDh
int 21h

mov ah,40h          ; Y parchea este valor
mov bx,handle
mov cx,1
lea dx,value
int 21h

mov ah,42h          ; Reconoces esto???
mov al,0
mov bx,handle
xor cx,cx
mov dx,03482h
int 21h

mov ah,40h          ; Y esto???
mov bx,handle
mov cx,1
lea dx,value
int 21h

mov ah,3eh          ; Ya esta hecho, cerremos el archivo
mov bx,handle
int 21h

mov ah,09h          ; Escribe algo mas en pantalla
lea dx,text4
int 21h

ready:
mov ax,4c00h        ; Regresemos a dos
int 21h             ; estamos listos!

handle dw 0
text1 db 'Crack for CGI-star pro 3.1',13,10,'$'
```

```
text2 db 'Made by Cruehead / MIB',13,10,'$'
text3 db Tienes que tener CSPRO.EXE en el mismo directorio de
CRACK.COM',13,10,'$'
text4 db 'Hecho! Disfrutalo!',13,10,'$'
value db 235                    ;el valor hex para 235 es EB y EB para JMP
file db 'CSPRO.EXE',0           ;

code ends
end start
```

Todavia confuso?

Pienso que el fuente habla por si mismo. Solo piensa para saber como supe donde mover el puntero:

```
mov ah,42h          ; Necesitamos mover el puntero
xor al,al
xor cx,cx
mov bx,handle
mov dx,02F9Ch       ; Este valor...
int 21h
```

He usado nuestro querido **Softice** para crackear esto, cuando uso el soft ice rapidamente veo donde tenemos que parchear el programa. Por lo que cambio los valores en un editor hexadecimal, luego uso la utilidad de dos **FC (File Compare)** para comparar el archivo original con el crackeado. Y así obtengo los valores, por lo tanto conociendo esto, el resto del código deberia ser facil de seguir **reversed.**
