Resumen Semana 14

El reto de esta semana consistirá en aplicar Ingeniería Inversa sobre el archivo subido al Discord.

Esta semana empezamos con reversing.

Reverse Engineering

La ingeniería inversa (en Software) es el proceso de obtener información acerca del funcionamiento interno de un programa a partir de un ejecutable.

.exe

Para poder ver cómo funciona la ingeniería inversa, vamos a entender primero cómo conseguimos un ejecutable en Windows (el proceso en Linux es parecido).

Imaginad que tenemos un fragmento de código en C como este:

```c
#include <stdio.h>

int main() {
    printf("Adios mundo :(");
    return 0;
}
```

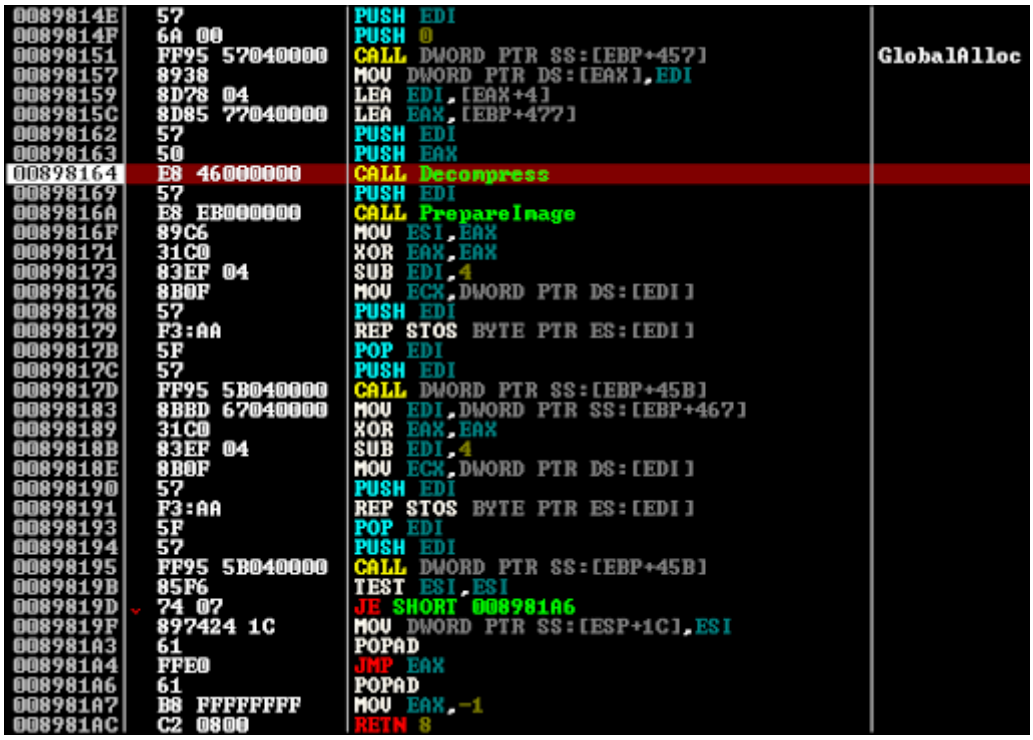
Todo lo que hace este trozo de código es imprimir "Adios mundo ☹" por consola. Para convertir este código en un ejecutable en Windows, hará falta **compilarlo.** Utilizando un compilador (como MinGW), se transforma este código en lo que llamamos "código máquina" (realmente pasan algunas cosas más, pero no hace falta que las conozcamos todavía). Y voila, tenemos un ejecutable.

El código máquina es un tipo de código que la CPU comprende, y sabe ejecutar. Genial, ahora tenemos un ejecutable (compuesto por código máquina y muchas otras cosas que ya veremos) en nuestro PC. Es una pena que seamos incapaces de comprender código máquina tal cual, ya que se ve de esta manera:

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Sin embargo, en algún momento en la historia se tuvo que programar únicamente usando código máquina, ya que no había otra cosa. Y aquí es donde nace el concepto de **lenguaje ensamblador.**

Voy a utilizar esta imagen que para nada he cogido de internet para explicar este nuevo concepto. Esta imagen muestra un "**debugger"**, un programa que nos deja ejecutar código ensamblador/código máquina instrucción por instrucción.



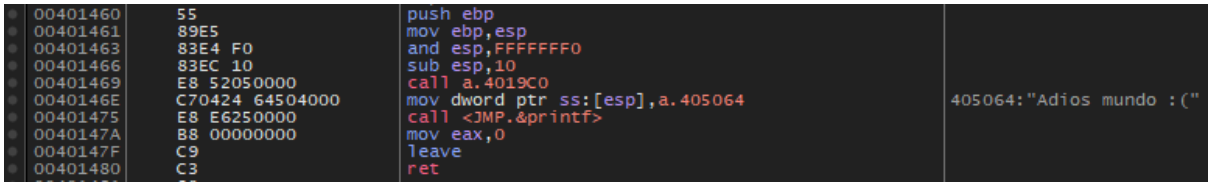
Lo que vemos en la columna de la izquierda del todo son **direcciones de memoria**. En este caso vemos desde 0x0089814E hasta 0x008981AC. Efectivamente, se visualizan en hexadecimal, y es conveniente que os vayáis adaptando a esto.

En la columna de su derecha, vemos los **opcodes**. Esto es… esencialmente el código máquina. El primer "57", es algo que la CPU entiende, y sabe que hacer con ello. Después le sigue un "6A 00", y la CPU de nuevo, sabe cómo ejecutar eso.

La tercera columna es la que más nos interesa. Esta contiene **código ensamblador**. El código ensamblador es EL MISMO CÓDIGO QUE EL CÓDIGO MÁQUINA PEEEEEERO interpretado para que lo entendamos los humanos 😊

Al igual que la CPU entiende "**57**" y "**6A 00**", nosotros entendemos "**PUSH EDI**" y "**PUSH 0**". Los opcodes que hemos visto se traducen en ensamblador para que lo entendamos.

Volvamos ahora al ejemplo del principio, el "Adios mundo ☹".



Una vez compilado, el código de la función “main” es el que se muestra arriba. Por ahora no necesitamos saber casi nada, porque no sabéis ensamblador ehehehe. Pero vemos que en algún punto sale “Adios mundo :(”, y a continuación una instrucción “**call <JMP.&printf>**”. Podemos asumir que el programa va a llamar a la función “printf”, y probablemente va a imprimir ese string.

Acabamos de hacer reversing de un hola mundo!

Ahora bien… Estamos utilizando un **debugger**, lo que significa que vamos al nivel más bajo de todos, donde podemos ejecutar instrucción tras instrucción. Vamos a dar un paso atrás antes de que nos explote la cabeza y vamos a conocer los **decompilers**.

Un decompiler es un programa que traduce (o lo intenta) un ejecutable a un lenguaje de alto nivel (como C o Java). Ejemplos de decompilers son:

- IDA Pro
- Ghidra (el que usaremos)
- Cutter

Una vez abierto el ejecutable en Ghidra, por ejemplo, si buscamos para encontrar la función **main** veremos algo así:

```
int __cdecl _main(int _Argc,char **_Argv,char **_Env)

{
  ___main();
  _printf("Adios mundo :(");
  return 0;
}
```

Y efectivamente… bastante acertado y similar al original.

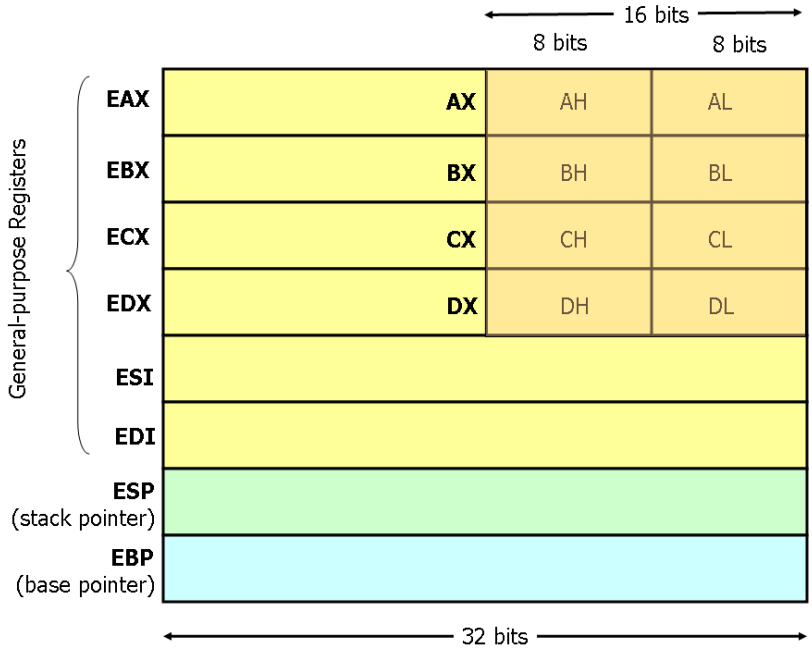
Lo que estamos haciendo ahora es **análisis estático** del ejecutable. Se llama así porque no lo estamos ejecutando. Cuando analicemos el ejecutable ejecutándolo (con el **debugger**), estaremos realizando **análisis dinámico**.

Lo único malo del análisis estático es que el programa en cuestión puede tener ciertas protecciones que hacen que sea un infierno intentar discernir algo en el código al hacer reversing.

Y por esto, en las siguientes clases empezaremos a utilizar y entender el análisis dinámico. Por ahora vamos a ir aprendiendo algo de lenguaje ensamblador para que luego se haga todo más sencillo.

x86

Vamos a aprender lenguaje ensamblador para la arquitectura x86 (arquitectura de 32 bits). Lo primero que hay que entender es que, en este lenguaje no existen “variables” como tal. Existen una serie de registros donde podemos guardar información.



Son estos, y por ahora todo lo que debemos saber es que en ellos podemos guardar valores numéricos. Por ejemplo, si tenemos el registro “eax”, podemos usar la instrucción:

“mov eax, 100”

para colocar el valor 100 en “eax”. Como habéis podido notar, la sintaxis es “mov dst, src”, lo que quiere decir que primero se coloca el destino, y después el valor que se quiere guardar. Esta instrucción hace lo mismo que en un lenguaje de nivel alto esto:

“int a = 100;”

Existen muchísimas instrucciones en este lenguaje, pero realmente hay un conjunto de ellas que se usan el 90% del tiempo. Estas instrucciones, además, pueden tener **operandos** o no. Por ejemplo, la instrucción mov que acabamos de ver tiene dos operandos “src” y “dst”.

Veamos algunas instrucciones:

- nop
    - Literalmente no hace nada; es como vosotros
- inc eax
    - Incrementa el valor de eax en 1
- dec eax
    - Decrementa el valor de eax en 1
- call “funcion”
    - Llama a la función “funcion” (ya hablaremos más de esto)

- add eax, valor
    - Incrementa el valor de eax en “valor”
- sub eax, valor
    - Decrementa el valor de eax en “valor”
- mov eax, 1337
    - Coloca el valor 1337 en eax

Y por ahora vale con eso…

Reto

Toca ensuciarse las manos, descargarse Ghidra (ghidra-sre.org) e intentar resolver el reto (quizá es un poco difícil ahora pero quiero que probéis y me preguntéis) que estará en discord.

Por ahora eso es todo, cualquier duda no dudéis en preguntar a cualquier compañero o a Llamas y Merk!

Hasta la semana que viene!