Capítulo 3
Ingeniería inversa

1. Introducción

1.1 Presentación

La técnica de ingeniería inversa (*reverse engineering* en inglés) consiste en estudiar un objeto (en nuestro caso un malware) para comprender su funcionamiento. En informática, eso se traduce por analizar el código de máquina de un programa; en nuestro caso, un malware. Dado que los malwares no se difunden con su código fuente y que no es posible encontrar códigos de malwares desarrollados en C o en C++, es necesario utilizar técnicas de ingeniería inversa para estudiar su funcionamiento interno. El analista estudiará el código en ensamblador del malware, función tras función.

Este código en ensamblador está disponible desensamblando el archivo binario.

El código en ensamblador no es tan fácil de leer como el código fuente. En efecto, se trata de un lenguaje de bajo nivel que manipula directamente instrucciones CPU y la memoria física.

Vamos a interesarnos principalmente en el ensamblador x86 (de 32 bits), aunque una pequeña sección presentará las principales diferencias entre el x86 y el x64 (de 64 bits) desde el punto de vista de la ingeniería inversa. En la actualidad, el 80 % de malwares están compilados en 32 bits, para poder impactar en el mayor número de máquinas posible (los sistemas Windows de 64 bits admiten los archivos binarios en 32 bits, pero no a la inversa).

Este capítulo explicará cómo leer e interpretar el ensamblador, las herramientas que se utilizan para ayudar en el análisis, así como trucos para facilitarlo.

1.2 Legislación

En muchos países, las técnicas de ingeniería inversa están reguladas por las leyes. Se pueden hacer muchos usos de esta disciplina:

– Espionaje industrial: ciertas empresas utilizan la ingeniería inversa para comprender los productos desarrollados por sus competidores y robar su conocimientos técnicos.
– Destrucción de protección anticopia: algunas personas utilizan estas técnicas para poder copiar videojuegos o copiar música bajo la protección de un sistema anticopia (DRM, *Digital Rights Management*).
– Interoperabilidad: los desarrolladores utilizan la ingeniería inversa para reescribir programas de forma que los productos puedan utilizarse en plataformas no compatibles con un fabricante (es el caso de muchos controladores de Linux).

Esta lista no es, evidentemente, exhaustiva, aunque permite comprender que esta técnica puede utilizarse para buenos o malos propósitos.

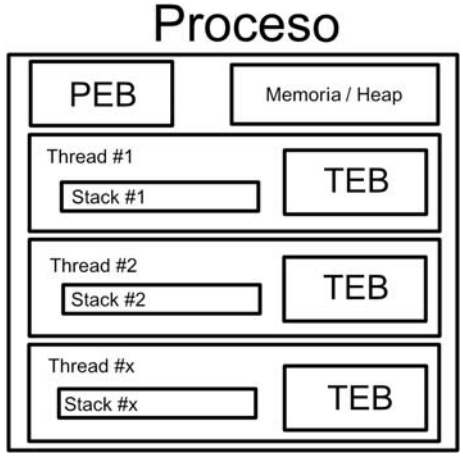
El uso de las técnicas de ingeniería inversa para analizar malwares se ha convertido en una práctica perfectamente legal. He aquí un fragmento del artículo en castellano que corresponde a nuestro caso (Artículo 96 de la ley de propiedad intelectual):

«La protección prevista en la presente Ley se aplicará a cualquier forma de expresión de un programa de ordenador (…) salvo aquellas creadas con el fin de ocasionar efectos nocivos a un sistema informático».

2. ¿Qué es un proceso de Windows?

2.1 Introducción

Cuando se ejecuta un proceso en Windows, el sistema operativo va a crear automáticamente un espacio en la memoria para este y un primer subproceso (*thread*). Todos los procesos en ejecución disponen de una estructura llamada *Process Environment Block* (PEB) que los describe. Cada proceso dispone de uno o vario threads. Cada thread posee su propia pila (*stack*, ver capítulo Técnicas de ofuscación) y una estructura que lo define llamada TEB (*Thread Environment Block*). Los threads pueden acceder a la memoria del proceso. He aquí un esquema que resume un proceso:



2.2 Process Environment Block

Se puede leer el contenido del PEB de un proceso. He aquí un ejemplo de PEB de un proceso cmd.exe visto por el depurador de Microsoft: *WinDBG*. La dirección de memoria en la que se encontrará la estructura PEB esta almacenada en `$PEB`:

```
0:001> r $PEB
$peb=0000000301292000
```

A partir de esta dirección se puede comprobar el contenido de la estructura PEB:

```
0:001> dt _PEB 0000000301292000
ntdll!_PEB
   +0x000 InheritedAddressSpace : 0 ''
   +0x001 ReadImageFileExecOptions : 0 ''
   +0x002 BeingDebugged    : 0x1 ''
   +0x003 BitField         : 0x4 ''
   +0x003 ImageUsesLargePages : 0y0
   +0x003 IsProtectedProcess : 0y0
   +0x003 IsImageDynamicallyRelocated : 0y1
   +0x003 SkipPatchingUser32Forwarders : 0y0
   +0x003 IsPackagedProcess : 0y0
   +0x003 IsAppContainer   : 0y0
   +0x003 IsProtectedProcessLight : 0y0
   +0x003 IsLongPathAwareProcess : 0y0
   +0x004 Padding0         : [4]  ""
   +0x008 Mutant           : 0xffffffff`ffffffff Void
   +0x010 ImageBaseAddress : 0x00007ff7`da850000 Void
   +0x018 Ldr              : 0x00007ffe`98e753c0 _PEB_LDR_DATA
   +0x020 ProcessParameters : 0x000001e6`84de1be0 _RTL_USER_PROCESS_PARAMETERS
   +0x028 SubSystemData    : (null)
   +0x030 ProcessHeap      : 0x000001e6`84de0000 Void
   +0x038 FastPebLock      : 0x00007ffe`98e74fc0 _RTL_CRITICAL_SECTION
[...]
```

Por ejemplo, en el offset 0x020 se encuentra la estructura _RTL_USER_PROCESS_PARAMETERS, que contiene los parámetros del proceso. Esta se encuentra en la dirección 0x000001e6`84de1be0. He aquí su contenido:

```
0:001> dt _RTL_USER_PROCESS_PARAMETERS 0x000001e6`84de1be0
ntdll!_RTL_USER_PROCESS_PARAMETERS
   +0x000 MaximumLength    : 0x788
   +0x004 Length           : 0x788
   +0x008 Flags            : 0x6001
   +0x00c DebugFlags       : 0
   +0x010 ConsoleHandle    : 0x00000000`00000050 Void
   +0x018 ConsoleFlags     : 0
   +0x020 StandardInput    : 0x00000000`00000054 Void
   +0x028 StandardOutput   : 0x00000000`00000058 Void
   +0x030 StandardError    : 0x00000000`0000005c Void
   +0x038 CurrentDirectory : _CURDIR
   +0x050 DllPath          : _UNICODE_STRING ""
   +0x060 ImagePathName    : _UNICODE_STRING "C:\WINDOWS\system32\cmd.exe"
   +0x070 CommandLine      : _UNICODE_STRING ""C:\WINDOWS\system32\cmd.exe" "
   +0x080 Environment      : 0x000001e6`84df6380 Void
```

Como se puede comprobar, la línea de comandos está disponible en el desplazamiento (*offset*) 0x70 de esta estructura en el proceso que se está ejecutando.

WinDBG proporciona el comando `!PEB` que lee y formatea todos los elementos pertinentes del PEB y los muestra al usuario.

2.3 Thread Environment Block

El mismo ejercicio que para el PEB se puede realizar con la estructura TEB:

```
0:001> r $TEB
$teb=0000000301299000
0:001> dt _TEB 0000000301299000
ntdll!_TEB
   +0x000 NtTib            : _NT_TIB
   +0x038 EnvironmentPointer : (null)
   +0x040 ClientId         : _CLIENT_ID
   +0x050 ActiveRpcHandle  : (null)
   +0x058 ThreadLocalStoragePointer : (null)
   +0x060 ProcessEnvironmentBlock : 0x00000003`01292000 _PEB
   +0x068 LastErrorValue   : 0
   +0x06c CountOfOwnedCriticalSections : 0
   +0x070 CsrClientThread  : (null)
   +0x078 Win32ThreadInfo  : (null)
   +0x080 User32Reserved   : [26] 0
```

Es interesante resaltar que en el offset 0x60 se encuentra la dirección del PEB vista anteriormente. En el caso de un proceso de 32 bits el offset es de 0x30. Así es como se puede obtener la dirección PEB mediante programación.

3. Ensamblador x86

3.1 Registros

El x86 es una arquitectura en la que el procesador utiliza principalmente registros de 32 bits para almacenar información. Cada registro contiene un número codificado en 32 bits, aunque este número puede verse como dos de 16 bits o 4 de 8 bits. Para mejorar su comprensión, ilustraremos este punto con un ejemplo.

El número hexadecimal 0xC0DEBA5E es un entero de 32 bits. En efecto, puede representarse por los siguientes 32 bits:

Hexadecimal	C	0	D	E	B	A	5	E
Binario	1100	0000	1101	1110	1011	1010	0101	1110

Puede verse como dos enteros de 16 bits, 0xC0DE y 0xBA5E, o como cuatro enteros de 8 bits, 0xC0, 0xDE, 0xBA y 0x5E. Es importante habituarse a realizar este pequeño ejercicio, pues el ensamblador hace a menudo un uso abusivo de estas distintas representaciones.

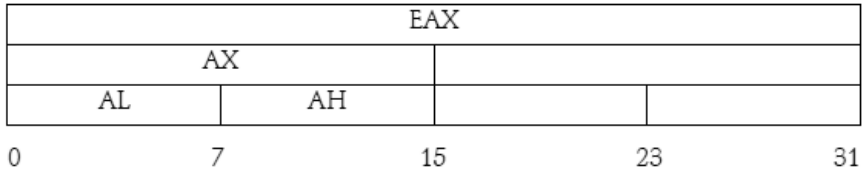
Para facilitar la explicación, se utilizan comúnmente términos específicos para distinguir estos números de distintos tamaños: un byte es un número de 8 bits; una palabra es un número de 16 bits, es decir, 2 bytes; un double es un número de 32 bits, es decir, dos palabras o 4 bytes.

Las arquitecturas x86 presentan principalmente 16 registros diferentes que se dividen en cinco tipos: registros generales, registros de índice, registros de punteros, registros de segmentos y por último los registros de estado o banderas (*flags* en inglés).

Registros generales

Existen cuatro registros de este tipo: EAX, EBX, ECX y EDX.

Estos registros suman 32 bits, pero pueden descomponerse en subregistros más pequeños. En este caso, su notación cambia. He aquí un ejemplo para EAX:



Las cifras debajo de la ruta corresponden a los bits del registro. La E presente al inicio de cada registro corresponde a *Extended* (Extendido). Los 32 bits son una especie de extensión de los 16 bits, de modo que los registros mantienen los mismos nombres agregando una E delante.

Como anécdota diremos que estas notaciones bárbaras provienen de las primeras arquitecturas de 8 bits, que incluían 4 registros generales: A, B, C y D. Con la aparición de las máquinas de 16 bits, se utilizaba la notación AX, BX, CX y DX, donde la X significaba *eXtended*. Cada uno de estos registros se descomponía en dos registros de 8 bits: Low para la parte inferior y High para la superior; de ahí las notaciones AL y AH. Cuando se pasó a una arquitectura de 32 bits, se utilizó el mismo mecanismo: se agregó una E de ***E****xtended* como prefijo de todos estos registros para mantener la coherencia con las notaciones anteriores.

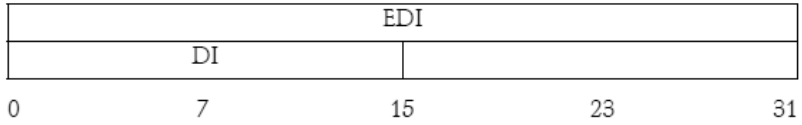
Generalmente, estos registros tienen un uso específico. Sin embargo, tenga en cuenta que su uso puede cambiar y, por lo tanto, no está garantizado.

El registro EAX se utiliza para realizar cálculos. El registro EBX se utiliza a menudo para acceder a matrices (*arrays*). El registro ECX se usa frecuentemente como contador de bucles. EDX se utiliza como extensión de EAX para almacenar más información virtualmente.

Registros de índice

Existen dos registros de índice: EDI y ESI.

Estos registros utilizan 32 bits, aunque pueden igualmente descomponerse en subregistros. He aquí un ejemplo para EDI:



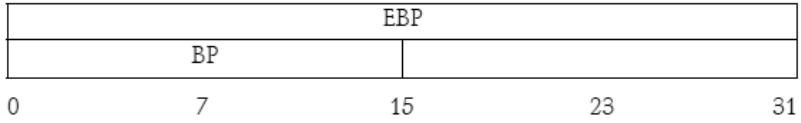
Las cifras por debajo representan el número de bits.

Estos registros se utilizan por lo general para manipular cadenas de caracteres o copias en memoria; la D de EDI significa "Destino" (*Destination*) y la "S" de ESI significa Origen o fuente (*Source*). De este modo, cuando se realiza una copia en memoria podemos observar a menudo que EDI representa el puntero hacia la zona de memoria de destino y ESI representa el puntero hacia la zona de memoria de origen. Cabe destacar que, si bien los compiladores respetan a menudo esta convención, no siempre ocurre así.

Registros de punteros

Existen tres registros de punteros: EBP, ESP y EIP.

Estos registros pueden igualmente descomponerse en subregistros. He aquí un ejemplo para el registro EBP:



Las cifras representan los bits del registro. Estos registros son registros particulares. He aquí el uso de cada uno de ellos:

– El registro EBP contiene una dirección que apunta a la base de la pila, es decir, el límite entre los argumentos y las variables locales. Este registro permite acceder, generalmente, a los datos apilados en memoria en la pila (*stack*). EBP permite recuperar datos en memoria.
– El registro ESP contiene la dirección actual de la parte superior de la pila. Este puntero designa la zona de la pila donde se copiarán los datos para ponerlos en la memoria de pila. ESP permite almacenar datos en memoria.
– El registro EIP contiene la dirección de la siguiente instrucción para ejecutar.

Estos registros se utilizan de manera implícita y generalmente no se pueden modificar por las funciones del programa.

Registros de segmentos

Existen seis registros de segmentos: CS, DS, ES, FS, GS, SS.

Estos registros son de 16 bits utilizados para almacenar los valores.

Registro de estado o banderas

Este registro de 32 bits se llama EFLAGS.

Algunas banderas están reservadas por los fabricantes y no pueden utilizarse; tienen un valor predefinido. A continuación, se muestra un diagrama del contenido de este registro con los nombres de las banderas o sus valores reservados:

Bit	Nombre de la bandera
0	CF (*Carry Flag*; acarreo)
1	1 (reservada)
2	PF (*Parity Flag*; paridad)
3	0 (reservada)
4	AF (*Auxiliary carry Flag*; acarreo auxiliar)
5	0 (reservada)
6	ZF (*Zero Flag*; cero)
7	SF (*Sign Flag*; signo)
8	TF (*Task Flag*; tarea)
9	IF (*Interrupt Flag*; interrupción)
10	DF (*Direction Flag*; dirección)
11	OF (*Overflow Flag*; desbordamiento)
12	IOPL (*I/O Privilege Level*; nivel de privilegio de E/S)
13	(nivel de privilegio de E/S)
14	NT (*Nested Task*; tarea anidada)
15	0 (reservada)
16	RF (*Resume Flag*; reiniciar tarea)
17	VM (*Virtual 8086 Mode*; modo virtual)
18	AC (*Alignement Check*; control de alineación)
19	VIF (*Virtual Interrupt Flag*; interrupción virtual)
20	VIP (*Virtual Interrupt Pending*; interrupción virtual pendiente)