Análisis de muestras maliciosas

En este laboratorio vamos a poner en práctica los conocimientos adquiridos en la parte de análisis básico de malware. Antes de entrar en la parte de análisis, encontrarás una descripción sobre el formato de archivos ejecutables de Windows y la herramienta `OllyDBG`, para detallar cómo funciona la parte de depuración de aplicaciones binarias. `OllyDBG` es un depurador de aplicaciones que puedes utilizar después de haber utilizado las herramientas básicas que hemos descrito en clase.

IMPORTANTE

Las muestras de malware que se te han proporcionado **son REALES**. Por tanto, ten mucho ojo cuando las ejecutes, dado que levarán a cabo su comportamiento malicioso en el sistema. Las encontrarás disponibles en la página web de la asignatura en un fichero comprimido con la contraseña "malware". **Se recomienda encarecidamente usar una máquina virtual o aislada para la realización de esta sesión de laboratorio**. El profesor no se hace responsable de los posibles daños que se puedan provocar por su ejecución.

1. El formato Portable Executable

El formato Windows *Portable Executable* (PE) es el formato estándar utilizado por Windows para representar archivos ejecutables [1]. Windows PE se introdujo desde WinNT 3.1 en adelante como un reemplazo para el formato ejecutable anterior, el formato de archivos COFF. Como comentario histórico de interés, el formato COFF también se usó en sistemas basados en Unix antes de ser reemplazado por el formato *Executable and Linkable Format* (ELF), formato actual de archivos ejecutables en estos sistemas.

El formato Windows PE es una estructura de datos definida en el archivo ≪`WinNT.h`≫ del Windows SDK, que se divide en diferentes partes como se muestra en la Figura 1. Primero, se encuentran los encabezados de MS-DOS (*MS-DOS headers*) para compatibilidad con versiones anteriores. Estos encabezados comprenden el propio encabezado de MS-DOS y el stub de MS-DOS, que es un fragmento de código para informar que el programa ejecutable no se puede ejecutar en modo DOS (mensaje de consola). Luego vienen los encabezados PE/COFF (*PE/COFF headers*), que incluyen los bytes mágicos de la firma PE como una cadena ASCII (caracteres "PE" seguidos de dos bytes nulos), el encabezado del archivo PE (*PE file header*), que define las características del programa binario (como la arquitectura de la máquina para la que fue compilado el archivo PE, su endianness, si tiene símbolos, etc.), y opcionalmente el encabezado opcional PE (*PE optional header*).

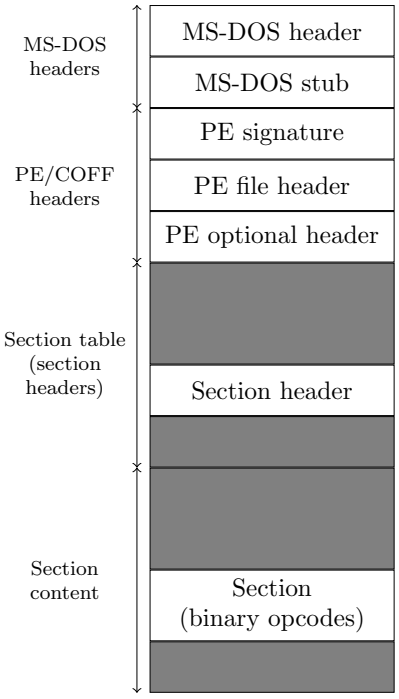


Figura 1: Formato de ejecutables de Windows: *Portable Executable* (PE).

En realidad, esta última parte de los encabezados PE/COFF sólo es opcional para archivos de objeto y siempre está presente para archivos ejecutables. El encabezado opcional incluye información importante sobre el programa binario, como su dirección de memoria virtual preferida y una estructura llamada *DataDirectory*, que contiene datos relevantes como los directorios de exportación e importación del programa binario, así como su tabla de reubicación.

El cargador de imágenes de Windows usa esta tabla de reubicación para cambiar cualquier instrucción o referencia de datos cuando el programa binario se asigna a una dirección virtual que no sea su dirección preferida. En los directorios de exportación se encontrarán las referencias de las funciones que exporta (se verán normalmente en los ficheros ejecutables de librerías compartidas, llamadas *dynamic link libraries* [DLLs] en Windows), mientras que en los directorios de importación se encontrarán las referencias a todas las funciones externas (de otras librerías) de las que depende el ejecutable para poder funcionar.

Este directorio de importación contiene, en concreto, una serie de entradas de la siguiente forma:

```
Offset     Size     Field
0          4        Import Lookup Table RVA
4          4        Time/Date Stamp
8          4        Forwarder Chain
12         4        Name RVA
16         4        Import Address Table RVA
```

El término RVA hace referencia a *dirección virtual relativa*. Esta dirección es relativa dado que un ejecutable se puede cargar en diferentes direcciones base, con lo que no puede existir una dirección absoluta. Esta dirección es relativa, por tanto, a la dirección de la imagen base (que

obtendrá cuando esté cargada en memoria).

Como puedes observar, uno de los campos hace referencia a la *Import Lookup Table* (ILT). Esta tabla contiene todas las direcciones que son importadas por el ejecutable de una determinada DLL, y contendrán punteros a cadenas (u ordinales) que identifican cómo se llama la API que se está importando (o la posición en la tabla de exportació, entre otros datos.

Por último, aparece un campo que hace referencia a la *Import Address Table* (IAT). Esta tabla contiene la misma información que la ILT, hasta que el ejecutable se carga en memoria para lanzarse en ejecución. En ese momento, se cargarán las DLLs de las que dependa el ejecutable (así como las DLLs de las que pudieran depender las DLLs que se cargan) y se resolverán las direcciones, apuntando de manera correcta cada función importada por el ejecutable a una dirección virtual, que será la implementación de la función importada de la DLL que corresponda. Este trabajo de resolución de direcciones de funciones importadas lo realiza el propio cargador de ejecutables de Windows.

Después de los encabezados PE/COFF aparecen los encabezados de sección (*section headers*). Para cada sección contenida en un programa binario existe un encabezado de sección que define el tamaño de la sección en el archivo binario, así como su tamaño en memoria, además de otras características (si los datos de esa sección son ejecutables, de sólo lectura, escribibles, etc.). Finalmente, se encuentra el contenido de cada sección como un flujo de bytes lineal. El comienzo y el final de una sección se definen en el encabezado de cada sección.

2. La herramienta `OllyDBG`

`OllyDBG` es un depurador de 32 bits que funciona sobre el sistema operativo Windows. Como se indica en `https://en.wikipedia.org/wiki/OllyDbg`, permite visualizar los registros de la CPU, reconoce procedimientos, llamadas API, tablas, constantes y cadenas, así como localiza rutinas a archivos y bibliotecas referenciados desde dentro del fichero binario que se está analizando. Además, tiene una interfaz amigable, y su funcionalidad puede ser extendida por *plugins* de terceros. Una lista de estos complementos adicionales y disponibles gratuitamente se puede consultar en `http://www.openrce.org/downloads/browse/OllyDbg_Plugins`.

Obtén este programa accediendo a la página web oficial `http://www.ollydbg.de/download.htm` (también se te ha proporcionado como material en el curso, en el fichero `odbg110.zip`). El proceso de instalación de `OllyDBG` es muy sencillo. Simplemente, descomprime el fichero comprimido que se descarga de la web y ya está listo para ser usado. La instalación de plugins también es extremadamente sencilla: por defecto, el `OllyDBG` busca cualquier fichero con extensión DLL que se encuentre en el mismo directorio que la herramienta e intenta cargarlos. Si son plugins del `OllyDBG`, estas librerías incorporarán ciertos métodos que son ejecutados por `OllyDBG` para completar la carga del plugin y la integración con el depurador.

Una vez descargado, puedes proceder a ejecutar este programa y abrir un ejecutable cualquiera (menú File » Open). Puedes abrir el fichero de nombre "hola.exe" proporcionado. La vista que obtendrás al cargar un binario en `OllyDBG` es algo similar a lo que aparece en la Figura 2. Seguramente tendrás un color de fondo y destacado de las instrucciones diferente de lo que se muestra en esa figura; puedes cambiar el esquema de colores en el menú del click derecho Appearance – te recomiendo que al menos uses el esquema de destacado de las instrucciones de llamada y de saltos (Appearance » Highlighting » Jump'n'calls).

La ventana de la CPU (icono **C** en la barra de botones) es la ventana principal del `OllyDBG` (véase la Figura 2). Esta ventana se divide en 4 partes diferenciadas. La parte primera (parte superior izquierda) muestra el desensamblado del binario que se ha cargado en el depurador. Aparece la dirección de memoria de la instrucción (primera columna), junto con la codificación

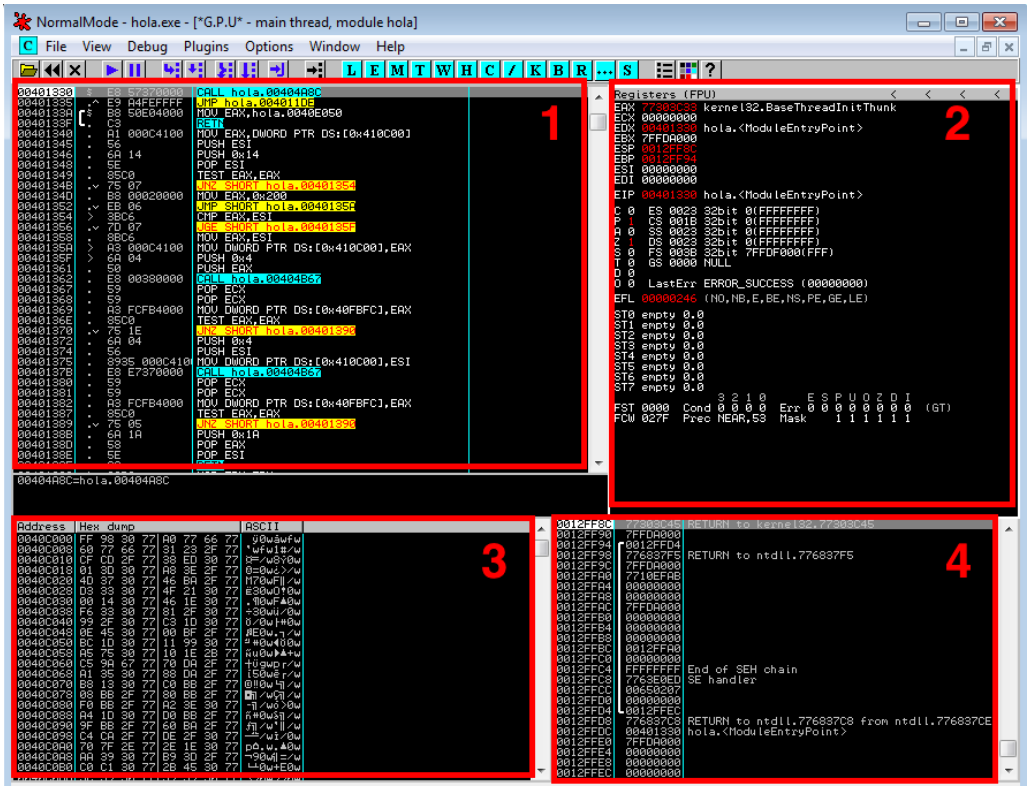


Figura 2: Ventana de CPU de la aplicación `OllyDBG`.

en hexadecimal de la instrucción y la codificación en ensamblador (segunda y tercera columna, respectivamente). En la segunda parte (parte superior derecha) aparecen los registros del procesador y los valores actuales. Fíjate que aparece también el registro de banderas, en forma vertical. Cada bandera está representada por una letra ('C' es la de acarreo, 'Z' la bandera de cero, etc.). Fíjate también que el valor del registro EIP, que marca la siguiente instrucción que va a ejecutar el procesador, se encuentra remarcado en la primera parte de la ventana (en la Figura 2, corresponde a la dirección `0x401330`). En la tercera parte (parte inferior izquierda), aparece un volcado de memoria. Ahí podemos visualizar el contenido de la memoria de cualquier zona de memoria que esté disponible en el binario que se está analizando. Por último, en la cuarta parte de esta ventana (parte inferior derecha), se encuentra la pila. La primera columna corresponde a la dirección de memoria de la pila, y a la derecha se muestra su contenido. Observa que el valor del registro ESP de la segunda parte se encuentra aquí remarcado, dado que ESP siempre apunta al tope de la pila.

Como ficheros adicionales, en esta práctica se te ha proporcionado diferentes plugins del depurador `OllyDBG`. Prueba a instalar tú mismo los plugins que se proporcionan en los ficheros `CommandBar_3.20.110.zip` y `HideDebugger124.zip`, y `ollydbgpedumper301.rar`.

Uno de los plugins instalados (concretamente, `CommandBar_3.20.110.zip`) te habrá habilitado en el `OllyDBG` una barra adicional en la ventana de la aplicación (parte inferior de la ventana). Esta barra permite que insertes comandos directamente, permitiendo hacer un uso del depurador menos dependiente del ratón y ahorrando mucho tiempo. Por ejemplo, si quisiéramos poner un punto de ruptura en la función de Windows `GetDlgItemTextA` (recuerda que se usa para leer un

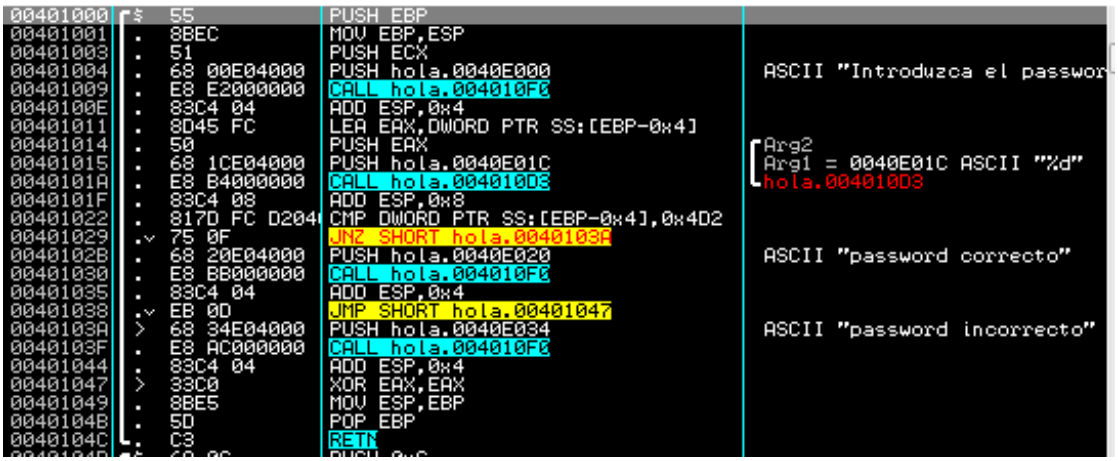


Figura 3: Código ensamblador generado del Código 1.

texto de un cuadro de diálogo), simplemente podemos escribir en la barra de comandos:

```
bp GetDlgItemTextA
```

Vamos a analizar el Código 1, programado con lenguaje C, a modo de ejemplo. Como puedes observar, el programa está solicitando un número al usuario, que almacena en la variable entera *numero* (línea 8). Después, se está comprobando si ese número es igual a 1234 para mostrar el mensaje de "password correcto". En caso contrario, se mostrará por pantalla el mensaje de "password incorrecto".

El código que ha generado el compilador de C de Microsoft se muestra en la Figura 3. En esta práctica, se te ha proporcionado adicionalmente el fichero "hola.exe", con firma MD5 `1301fd982ed15d9aa1394ce8d23a663a`. Puedes abrirlo con `OllyDBG` y llegar hasta el código mostrado dirigiéndote a la dirección de memoria `0x401000` en la ventana de la CPU (sube hasta arriba del todo con el ratón, o introduce esa dirección en la opción de Go to, accesible desde el botón derecho). En concreto, la comparación de ese número introducido por el usuario con 1234 se está produciendo en la instrucción:

```
00401022  817D CF D2040 CMP DWORD PTR SS:[EBP-0x4], 0x4D2
```

Código 1: Ejemplo de "Hola, mundo" en código C (fichero *hola.c*).

```
1  #include <stdio.h>
2
3  int main(int argc, char *argv[])
4  {
5      int numero;
6
7      printf("Introduzca el password: ");
8      scanf("%d", &numero);
9      if(numero == 1234)
10         printf("password correcto");
11     else
12         printf("password incorrecto");
```

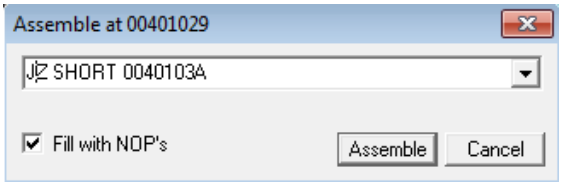


Figura 4: Cambio de la instrucción de salto condicional (dirección `0x00401029`).

```
13
14      return 0;
15  }
```

La instrucción `CMP` es una instrucción que sirve para comparar dos valores en lenguaje ensamblador. En la parte izquierda de esta instrucción aparece una referencia a una zona de memoria (por estar entre corchetes sabemos que se refiere a una dirección de memoria), en concreto, a `EBP - 4`. En la parte derecha aparece el valor `0x4D2`. Este valor es el valor 1234 pero en hexadecimal. Después de esta operación, aparece un salto condicional:

```
00401029     75 0F          JNE SHORT hola.0040103A
```

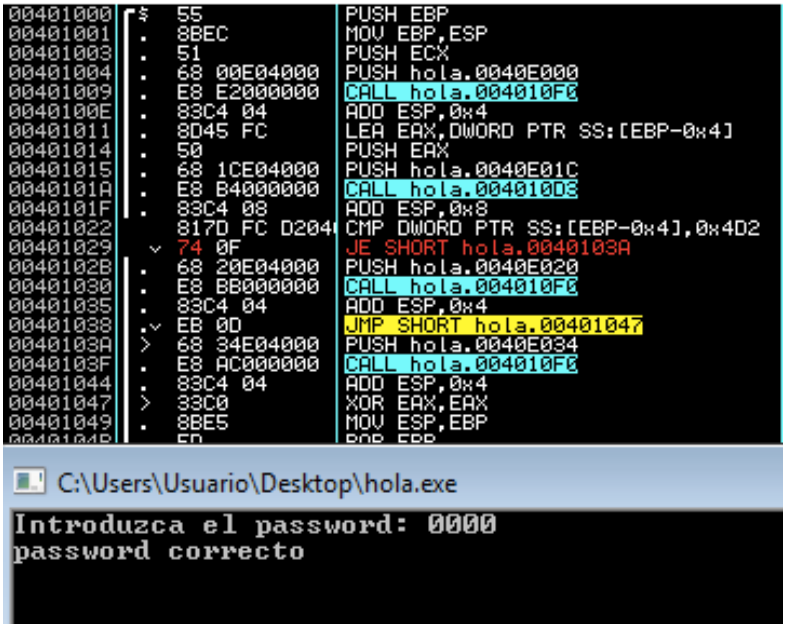
Este salto condicional evalúa lo que ha sucedido con la instrucción anterior y en función del resultado, llevará el flujo de ejecución a otra dirección, o continuará linealmente. Es decir, si los números que se han comparado con la instrucción `CMP` anterior son iguales, este salto condicional no se ejecutará, con lo que el procesador pasará a ejecutar las instrucciones siguientes a este salto. Es decir, se ejecutará la instrucción que aparece en la dirección `0x0040102B` (instrucción `PUSH hola.0040E020`). Observa en la Figura 3 que en estas instrucciones aparecen referencias a esa cadena de "password correcto". En caso de que los números comparados sean diferentes, entonces se ejecutará el salto cambiando la ejecución a `0x0040103A`. Observa en la Figura 3 que en estas instrucciones aparecen referencias a la cadena de "password incorrecto".

Vamos a intentar modificar este programa que con cualquier número muestre el mensaje de "password correcto", excepto para el número 1234, que mostrará "password incorrecto". Es decir, vamos a cambiar el comportamiento normal de este ejecutable.

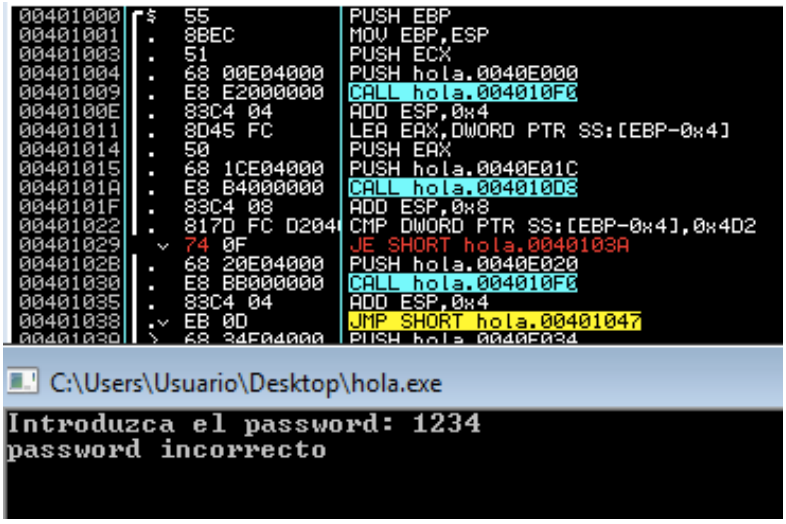
Para ello, lo único que tenemos que hacer es cambiar ese salto condicional de la línea `0x00401029` por su inverso: tenemos que transformar esa instrucción de ensamblador `JNE` en la instrucción `JE`. Si hacemos doble click sobre esta instrucción en la ventana del `OllyDBG` se nos va a abrir la ventana de ensamblado, y verás algo similar a la ventana que se muestra en la Figura 4. Ahora, podemos modificar la instrucción simplemente cambiando ese `JNE` por `JE` (o `JNZ` por `JZ`, ya que estos saltos son equivalentes), tal y como se muestra en la Figura 4.

Una vez realizada esta modificación, si ahora ejecutamos el programa (botón similar al "Play" del vídeo, ▶, o menú Debug > Run) e introducimos un valor cualquiera al programa, como el número 0000, veremos que el programa nos lo acepta como correcto. Del mismo modo, si le introducimos el número 1234, nos mostrará el mensaje de "password incorrecto". Ambos resultados de ejecución se muestran en la Figura 5.

Esto que acabas de realizar en esta práctica guiada se conoce como *parcheo* de la aplicación, ya que has modificado las instrucciones en ensamblador del binario. Habrás observado que si cierras la aplicación `OllyDBG` y la vuelves a abrir, volviendo además a cargar el binario "hola.exe". Si ahora lo ejecutas (o si lo ejecutas de manera normal, desde una línea de comandos de Windows), observarás que únicamente acepta como válido el número original. Es decir, el cambio que has realizado se ha perdido. Esto es así porque el parcheo que habías realizado es un parcheo *en*



(a) Aceptando cualquier número como correcto



(b) Rechazando el número 1234 como correcto

Figura 5: Resultados de ejecución de la aplicación modificada.

caliente, es decir, es un parcheo realizado en la memoria del proceso que se está ejecutando: **el cambio de instrucciones realizado no se refleja en el ejecutable que se almacena en el disco duro**.

3. Metodología de análisis de malware

Recuerda que las preguntas que hay que responder cuando uno realiza un análisis de malware son las siguientes:

- *¿Qué actividad maliciosa está llevando a cabo en el sistema?*
- *¿Cómo se inicia esta actividad?* (es decir, localizar el método de persistencia que utiliza)
- *¿Cuándo infectó el sistema, y cómo consiguió entrar?*
- *¿Cómo se puede eliminar la amenaza y limpiar el sistema infectado, y cómo puedo prevenir que vuelva a ocurrir?*

Para responder a estas preguntas, la metodología del análisis de malware a seguir es la siguiente:

1. **Fase de análisis estático**. Esta fase, también llamada fase de análisis de código muerto, comprende el análisis del fichero binario sin llegar a ejecutarlo. Es decir, en esta fase se leerá el contenido binario (los bytes) del fichero. En esta fase ayudará conocer en profundidad el ensamblador de la arquitectura donde se ejecuta el binario (es aquí donde entran las habilidades de ingeniería inversa que veíamos en la sección anterior). Esta fase se divide en tres aspectos básicos:
   - *Cálculo de firmas MD5/SHA1/SHA256*. En esta subfase se calculará la firma del binario usando algún algoritmo criptográfico como MD5, SHA-1, o SHA-256. Estos algoritmos permiten, dada una entrada, calcular un número que identifica de manera "exclusiva" al binario. Una vez calculada la firma, se puede proceder a buscarla en Internet con el objetivo de localizar otros análisis de la muestra de malware y facilitar así el trabajo propio de análisis. Una buena herramienta en Windows para esta tarea es `HashTab` (`http://implbits.com/products/hashtab/`). En sistemas MacOS o GNU/Linux, se pueden usar los comandos nativos `md5sum` o `sha1sum`, por nombrar algunos.
   - *Cadenas del fichero binario*. Esta segunda subfase consiste en localizar todas las cadenas de texto que estén contenidas dentro del binario. Las cadenas de texto, como variables definidas dentro del código fuente original del programa y que éste usa durante su ejecución, se encuentran dentro del fichero ejecutable. De hecho, si el ejecutable no tiene ningún mecanismo de protección (por ejemplo., ofuscación de código o cifrado), estas cadenas se encontrarán en el fichero completamente visibles. Es decir, si viéramos el fichero ejecutable con un procesador de textos, se verían las cadenas de texto claramente. La presencia de ciertas cadenas es un indicio de la actividad que estará realizando la muestra maliciosa. Por ejemplo, si vemos una dirección IP de Internet, o un dominio web y partes de cadenas relativas a una petición HTTP POST o GET, podemos intuir que la muestra maliciosa seguramente se está conectando a dicha IP/dirección web mediante peticiones HTTP POST o GET. Para esta fase, se puede usar una herramienta como `strings` (herramienta nativa en sistemas basados en Unix, y disponible para Windows en `https://docs.microsoft.com/en-us/sysinternals/downloads/strings`).

- *Funciones de importación*. Por último, la última subfase del análisis estático consiste en analizar las funciones APIs de Windows que está utilizando el fichero ejecutable. Una función API es una función proporcionada por el sistema operativo que facilita la interacción con el mismo por parte de otros programas. Es decir, permite al desarrollador de aplicaciones utilizar los recursos del sistema como ficheros, sockets de red, claves de registro, crear procesos, etc. Por ejemplo, la presencia de funciones relativas a ficheros (por ejemplo, *FindFirstFileA*) llevará a concluir que supuestamente el malware realiza alguna acción con ficheros. Para conocer más sobre las funciones que usa el binario, se recomienda consultar la web del MSDN (`https://msdn.microsoft.com/en-us/library/ms310241`) para consultar una función específica. En concreto, el MSDN proporciona información acerca de qué hace, qué parámetros usa, y qué devuelve cada una de las funciones del sistema. En Windows, para consultar las funciones de importación se pueden usar herramientas como `CFF Explorer` (`http://www.ntcore.com/exsuite.php`).

Todos estos pasos descritos corresponden a un análisis estático básico de la muestra. Un análisis más avanzado usaría herramientas desensambladoras u otras técnicas basadas en grafos de control de flujo o ejecución simbólica, entre otras. En cualquier caso, una de las mayores limitaciones del análisis estático es que el código del binario a analizar no tiene que estar ofuscado o protegido. Además, un análisis estático nos muestra **todos los posibles caminos de ejecución** de ese fichero ejecutable. Es decir, el espacio de estados de todos estos caminos puede ser muy grande y difícil de gestionar.

2. **Fase de análisis dinámico**. Esta fase, también llamada fase de análisis de código vivo o en caliente, comprende el estudio del fichero binario durante su ejecución. En concreto, en esta fase se estudia la interacción de la muestra de malware con su entorno. Esta fase se puede dividir en dos subfases, según lo que se entiende por entorno:
   - *Interacción con el Sistema Operativo*. Cuando entendemos por entorno la interacción que realiza con el sistema operativo, hay que atender a tres extremos particulares: ficheros (*¿qué archivos está creando, modificando, o eliminando?*, claves de registro (*¿qué claves de registro está creando, modificando, o eliminando?*) y procesos (*¿qué procesos está creando, modificando, o eliminando?*). Si se observa que la muestra de malware está creando nuevos ficheros, habrá que analizar de qué tipo de ficheros se trata y en caso de ser ficheros ejecutables, realizar con ellos un nuevo análisis de malware.
   - *Interacción con el exterior (Internet)*. Por último, hay que fijarse en la interacción que realiza el malware con Internet. En concreto, habrá que localizar si la muestra de malware realiza alguna conexión a Internet (*¿a qué direcciones IP o dominios de Internet trata de conectarse?*), y en caso de que haga alguna conexión, hay que mirar qué tipo de información le está mandando, cómo lo recibe y qué tipo de información recibe. Estos servidores a los que se conecta para mandarles información y posiblemente recibir órdenes se denominan *servidores de mando y contro*l (*C&C servers*, del inglés).

Esta fase de análisis dinámico que acabamos de describir nos servirá para responder a (casi) todas las cuestiones relativas al malware que se comentaban al inicio de esta sección. Un análisis dinámico más avanzado requeriría usar un depurador para analizar durante tiempo de ejecución cómo van evolucionando los registros de la CPU, los valores que devuelven las funciones de Windows que llama el malware, y demás.

Por último, cabe destacar que **a diferencia del análisis estático, en el análisis dinámico únicamente se analiza uno de los posibles caminos de ejecución**, el cual además puede depender de las condiciones actuales de ejecución.

4. Tareas a realizar

Como tareas, en este laboratorio se te solicita que respondas a las siguientes preguntas de las muestras de malware seleccionadas y proporcionadas en esta sesión como material adicional.

Las muestras de malware de interés para este laboratorio son, en concreto, las siguientes:

- *lab_malware01* (firma MD5: `2de244d4bf9851367108fa5d80729aaf`).
- *malware02* (firma MD5: `0de9765c9c40c2c2f372bf92e0ce7b68`).

Ejercicio 1.

¿Se está creando algún fichero en el sistema? En caso afirmativo, ¿cuántos ficheros y de qué tipo son? Detalla su nombre y extensión, así como una breve explicación de su contenido.

Ejercicio 2.

¿La muestra realiza algún tipo de persistencia? Si es así, detalla qué tipo de persistencia está realizando. Si existe alguna otra interacción con el registro de Windows, descríbela.

Ejercicio 3.

¿Interacciona con algún otro proceso? Si es así, describe con cuál (o cuáles) y qué interacción realiza.

Ejercicio 4.

¿Conecta con alguna dirección o dominio de Internet? Si es así, detalla con qué dominio o IP está intentando conectar. En caso de que sea un nombre de dominio, descubre la dirección IP a la que resuelve el nombre de dominio encontrado. Te puede servir para esto páginas web como `http://whois.domaintools.com/`.

Ejercicio 5.

Basado en el análisis del comportamiento que has observado, **¿cómo categorizarías esta muestra de malware?**

Referencias

[1] Microsoft Corporation. PE Format. [Online; `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`], November 2021. Accessed on November 29, 2021.

