Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Proyecto Fin de Carrera de Ingeniería en Informática

Análisis de rendimiento y niveles de protección de protectores de software

María Asunción Bazús Castán

Director: Ricardo J. Rodríguez Fernández

Ponente: José Javier Merseguer Hernáiz

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Marzo de 2014
Curso 2013/2014

Agradecimientos

A mi familia y amigos.
A Ricardo J. Rodríguez.
A José Merseguer.

Análisis de rendimiento y niveles de protección de protectores de software
RESUMEN

La ingeniería inversa es el proceso de analizar un programa a partir de su código binario. Puede usarse con diferentes propósitos, tanto legítimos (p. ej. encontrar errores en un programa) como ilegítimos (p. ej. copiar o analizar un software propietario). Los protectores de software son herramientas que permiten proteger un ejecutable frente a ataques de ingeniería inversa. Existen numerosas técnicas de protección que son implementadas por este tipo de herramientas. El funcionamiento básico de un protector de software consiste en encapsular el ejecutable a proteger dentro de otro ejecutable, que al ser iniciado descomprime el original en memoria y comienza su ejecución. Como se puede intuir, esta rutina adicional añade una sobrecarga tanto en tiempo de ejecución como en memoria. Sin embargo, no existe ningún trabajo en el que se haya medido esta sobrecarga. Tampoco existe ninguno donde se hayan categorizado las técnicas de protección más habituales, ni clasificado las herramientas protectoras más usadas.

El objetivo de este trabajo es realizar un estudio comparativo de diferentes herramientas protectoras en el entorno Windows. En primer lugar, se realiza una categorización de las técnicas de protección más comunes utilizadas en los protectores analizados. Después, se realiza un estudio comparativo a nivel cualitatativo y cuantitativo (fiabilidad y rendimiento). Para la evaluación cualitativa se lleva a cabo un análisis de las protecciones que implementa cada herramienta y se describen posibles técnicas de desprotección. Para la evaluación cuantitativa se ha creado un benchmark específico para realizar mediciones del tiempo de ejecución y consumo de memoria de un conjunto de aplicaciones protegidas con las distintas herramientas protectoras. Los resultados del benchmark permiten, por un lado, analizar la fiabilidad de las herramientas protectoras en la medida en que hayan podido o no ser protegidas con las distintas herramientas, y por otro lado, calcular el impacto en el rendimiento que tiene el uso de los protectores estudiados.

Los resultados del proyecto muestran que muchas de las protecciones no tienen la eficacia deseada ya que las ténicas de desprotección han sido ampliamente estudiadas y difundidas por la red. También muestran que el hecho de proteger un ejecutable provoca un mayor consumo de memoria y en algunos casos, también un mayor tiempo de ejecución. Esta sobrecarga será tanto mayor cuanto más técnicas de protección se hayan utilizado para proteger el ejecutable.

Índice

Índice de figuras

Índice de tablas

Índice de listados

Capítulo 1

Introducción

El término *ingeniería inversa*, en el campo de desarrollo de software, es el proceso de analizar un sistema existente con el objetivo de identificar sus componentes e interrelaciones y de producir una nueva representación del mismo en otra forma o con un mayor nivel de abstracción [CC90]. También puede verse como el proceso de ir hacia atrás en el ciclo de desarrollo de software, es decir, partiendo del binario llegar hasta el código fuente.

Se pueden distinguir dos tipos de técnicas utilizadas para el análisis del código binario: *análisis estático* y *análisis dinámico* [LTC13]. El análisis estático consiste en analizar el binario sin ejecutarlo. Para llevar a cabo este análisis, el primer paso es el desensamblado y el segundo el análisis del código ensamblador resultante. La principal desventaja de este tipo de análisis es que al no disponer de la información calculada en tiempo de ejecución se provoca una falta de exactitud en los resultados. Sin embargo tiene como ventaja la posibilidad de explorar todos los posibles caminos de ejecución. El análisis dinámico en cambio, requiere la ejecución del programa, consistiendo en monitorizar la ejecución y analizar un flujo de ejecución en concreto.

Existen diferentes tipos de herramientas utilizadas en la ingeniería inversa, tanto para el análisis estático (desensambladores, decompiladores) como para el análisis dinámico (depuradores, herramientas para el volcado de la imagen en memoria (*dumpers*), máquinas virtuales). Las técnicas y herramientas de ingeniería inversa pueden ser utilizadas con diferentes objetivos, tanto en el ámbito de desarrollo de software como en el ámbito de la seguridad informática [KLC+10, Eil05]. Algunas de las posibles aplicaciones son:

1. Analizar la funcionalidad de un programa que no está suficientemente documentado o encontrar errores.

2. Conseguir la interoperabilidad con otro sistema informático cuando no se dispone de documentación para la integración.

3. Eliminar la protección de un programa para hacerse con una copia ilegal del mismo (*cracking*) o utilizar partes del código de un programa de la competencia para copiarlas.

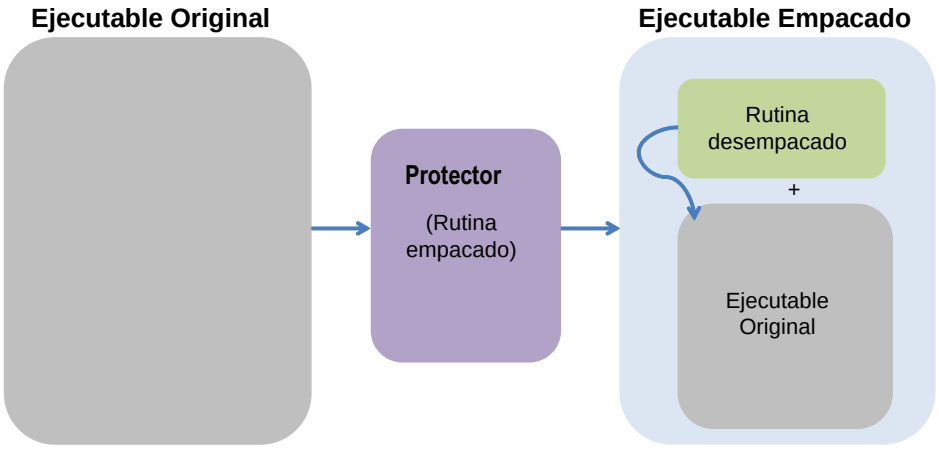

Figura 1.1: Proceso de protección de un ejecutable.

4. Detectar vulnerabilidades en un programa y explotarlas mediante la creación de software malicioso (en inglés, *malware*).

5. Analizar un programa con el objetivo de descubrir si puede tratarse de un software malicioso o no (sistemas anti-virus).

Dado que la ingeniería inversa puede utilizarse para conseguir copias ilegales de un programa, muchos distribuidores de software deciden incluir en sus programas protecciones anti-ingeniería inversa. Estas protecciones pueden ser implementadas tanto directamente en el código del programa como utilizando herramientas de terceros (por ejemplo, los denominados protectores de software (en inglés, *software packers*)). Un ejecutable protegido con una de estas herramientas es denominado ejecutable empacado.

El proceso de empacado consiste en, tomando como entrada un ejecutable (*ejecutable original*), generar otro ejecutable (*ejecutable empacado*) que contiene tanto el código y datos del programa original como la rutina de desempacado [VN06]. De modo que, en ejecución, primero se ejecuta la rutina de desempacado y una vez finalizada se transfiere el control a la primera instrucción del programa original, descomprimido en la memoria del programa (véase Figura 1.1). La diferencia entre los protectores de software y programas de compresión como gzip o bzip2, es que es el propio ejecutable el que contiene el código de descompresión, es decir, se asemeja a un fichero auto extraíble [Wik13a].

La técnica de empacado de software es considerada una de las técnicas anti-ingeniería inversa más potentes en el entorno de Microsoft Windows, ya que permite combinar diferentes técnicas de protección: anti-análisis estático y anti-análisis dinámico [KLC+10]. En este trabajo nos centramos en Windows ya que copa el mercado de los protectores.

Los protectores de software surgieron, en un inicio, como herramientas diseñadas para reducir el tamaño de un ejecutable. Al comprimir el ejecutable, se obtenía el beneficio añadido

de proteger el programa frente a ataques de ingeniería inversa al impedir que el código fuera analizado estáticamente. Sin embargo, fueron evolucionando con el paso del tiempo de tal manera que, mientras el tamaño del ejecutable dejaba de ser un aspecto crítico, la protección del software frente a ataques de ingeniería inversa era cada vez más importante, creándose así herramientas con métodos de protección más sofisticados y sacrificando, en algunos casos, el tamaño del ejecutable final y su tiempo de ejecución.

Actualmente, este tipo de herramientas protectoras son utilizadas a menudo por autores de malware para evitar que sus programas maliciosos sean detectados por los escáneres anti-virus [Tah07]. Esto tiene como consecuencia que programas benignos protegidos con alguna de estas herramientas puedan ser detectados como falsos positivos, además de obligar a los fabricantes de anti-virus a implementar algoritmos que desempaquen automáticamente estos programas antes de ser analizados para ver si se tratan o no de malware. Se calcula que aproximadamente el $80\,\%$ del malware actual se encuentra protegido con una o varias capas de protectores [GFC08].

El hecho de que un programa (benigno o no) esté protegido con una de estas herramientas no garantiza que nunca vaya a ser víctima de un ataque de ingeniería inversa. Es más una cuestión de tiempo que el programa original pueda ser restaurado a partir de la imagen en memoria: lo que se consigue con estas herramientas es dificultar este tipo de ataques.

En la actualidad existen múltiples herramientes de protección de software, tanto libres como comerciales [Wik13a]. Se calcula que existen aproximadamente unas 200 familias de packers diferentes [GFC08]. Sin embargo, no existe ninguna comparativa de ellas a nivel de calidad y cantidad de protecciones que ofrecen, ni tampoco a nivel de rendimiento en tiempo de ejecución y en consumo de memoria.

1.1. Objetivo

El objetivo de este PFC es realizar un análisis comparativo de diferentes herramientas de protección de software. Se llevará a cabo una evaluación tanto cualitativa como cuantitativa de 21 herramientas protectoras. Para la evaluación cuantitativa se describirán las protecciones que ofrecen así como la dificultad de eliminarlas. La evaluación cuantitativa se centrará en analizar tanto la fiabilidad de los protectores como su rendimiento a nivel de tiempo de ejecución y consumo de memoria.

La metodología utilizada para obtener esos resultados ha sido la siguiente:

- Profundizar en el conocimiento de las estructuras de binarios ejecutables de Windows.
- Estudio de las distintas ténicas de protección y desprotección de software (estudio de manuales y tutoriales y aprendizaje del uso de herramientas de depuración de ensamblador).
- Hacer una selección de los protectores de software interesantes para este estudio.
- Para cada protector seleccionado, hacer un estudio de las protecciones que implementa así como de la dificultad que supone eliminarlas.

- Creación de un benchmark para analizar la fiabilidad y el rendimiento (tiempo de ejecución y consumo de memoria) de los protectores seleccionados (un conjunto de programas en diferentes categorías como: cálculo entero, cálculo real, E/S de ficheros).

1.2. Motivación

La principal motivación para la elección de este PFC fue la oportunidad de profundizar mis conocimientos en el campo de la seguridad informática. La realización de este proyecto me ha permitido ampliar mis conocimientos sobre el funcionamiento interno del sistema operativo, la estructura de un ejecutable y las técnicas de protección y de desprotección de ejecutables más usadas en la actualidad en el entorno Windows.

1.3. Organización del documento

El presente documento está dividido en dos partes: la memoria, donde se explica el desarrollo del Proyecto; y los apéndices, donde se amplía la información de ciertos puntos relevantes.

El Capítulo 2 define los conceptos previos que sirven de ayuda para comprender el resto del documento, en concreto se describe la estructura de un ejecutable en Windows, el manejo de excepciones en Windows y las herramientas utilizadas. El Capítulo 3 describe el trabajo previo relacionado con el tema de este proyecto. El Capítulo 4 presenta una taxonomía de las protecciones de software. En el Capítulo 5 se describe el proceso de selección de los protectores para el estudio así como la creación de benchmark. El Capítulo 6 es la evaluación cualitativa y cuantitativa de los protectores. En él se describen las características de los protectores y las posibles técnicas de desprotección, después se analiza el rendimiento de los protectores en base a los resultados de las pruebas realizadas con el benchmark. Por último, en el Capítulo 7 se presentan las conclusiones y el trabajo futuro.

Respecto a los apéndices, el Apéndice A es donde se hace balance del esfuerzo temporal empleado en la realización del PFC. El Apéndice B describe las aplicaciones usadas en el benchmark con más detalle. Por último, el Apéndice C describe los protectores analizados en este trabajo, enumerando y clasificando las principales técnicas de protección que implementa cada uno así como el nivel de protección que proporcionan.

Capítulo 2

Conocimientos Previos

En este capítulo se definen algunos de los conceptos más importantes en los que se basa este proyecto. En primer lugar se describe la estructura de un fichero ejecutable en Windows y algunos conceptos fundamentales del funcionamiento del sistema operativo Windows. A continuación se enumeran los tipos de herramientas usadas en la ingeniería inversa y las herramientas utilizadas para la elaboración de este trabajo.

2.1. Formato Ejecutable Portable (PE) en Windows

El formato PE (*Portable Executable*) es un formato introducido por Microsoft junto con su sistema operativo Windows NT 3.1 en 1993. Se denominó *Portable* porque fue diseñado con la intención de convertirse en un estándar para todas las versiones del sistema operativo y distintos procesadores. A día de hoy sigue siendo utilizado en todas las versiones existentes de Windows aunque para versiones de 64 bits se ha definido el formato PE+ con algunas modificaciones. El formato PE es usado en archivos ejecutables (extensión .exe), librerías dinámicas (extensión .dll) y ficheros del sistema (extensión .sys) entre otros [Wik13c].

Un fichero en formato PE puede verse como un conjunto de datos organizados según una estructura predefinida. Como muestra la Figura 2.1, esta estructura se divide en dos partes principales, cabecera y secciones. La cabecera contiene toda aquella información necesaria para que el sistema operativo sea capaz de cargar el programa en memoria para empezar su ejecución correctamente, mientras que las secciones contienen todos los datos y el código del programa [Cor99, Pie94, Pie02].

La estructura de un fichero ejecutable está específicamente diseñada para permitir su carga y correcto funcionamiento en el sistema para el que fue diseñado. El *Windows Loader* es la parte el sistema operativo encargada de cargar el ejecutable y crear el proceso para que pueda ser ejecutado.

Una de las principales características de un ejecutable es que puede ser cargado en una dirección diferente de memoria cada vez que es ejecutado (en inglés, *relocatable*). Cuando el ejecutable es creado, se le asigna una dirección de carga preferente (el campo *ImageBase*, línea 11 en el Listado 2.4). Cuando el programa es ejecutado, el Windows Loader intenta en primer

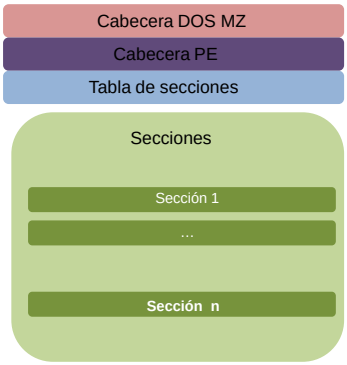

Figura 2.1: Estructura básica de un archivo PE en Windows.

lugar cargarlo a partir de esa dirección del espacio de direcciones virtual del proceso. Sólo en caso de no estar libre, lo carga en una diferente.

Otra característica importante es que cuando un fichero es ejecutado, el loader hace una copia del archivo en memoria (denominada *módulo*). La misma estructura del archivo en disco es replicada en memoria. La Figura 2.2 muestra como mayores offsets en disco se corresponden con direcciones de memoria más altas.

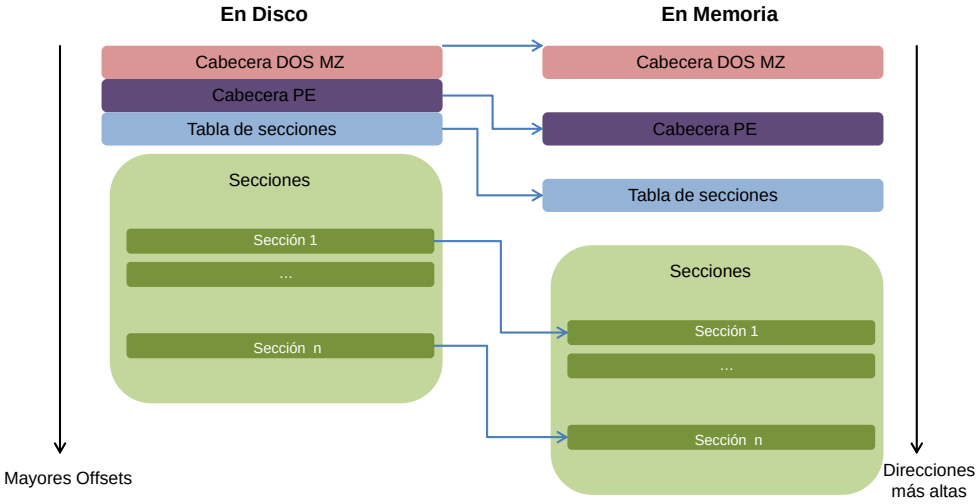

Figura 2.2: Estructura de un archivo PE en disco y en memoria.

Antes de detallar las estructuras que componen un fichero ejecutable, conviene aclarar el término Dirección Relativa Virtual (RVA). Dado que en un ejecutable existen referencias cruzadas a otras posiciones dentro del mismo fichero, es necesario poder especificarlas de manera independiente a la dirección de inicio de carga (que puede variar de una ejecución a otra) para que no tengan que ser recalculadas por el Windows Loader cada vez que se carga. Es por esto

que muchos campos están especificados por su RVA. De este modo, una posición dentro de un ejecutable puede especificarse según su:

- Offset: especifica el desplazamiento de una determinada posición respecto al inicio del archivo almacenado en disco.
- *Virtual Address* (VA): especifica la dirección de memoria en el espacio de direcciones virtual del proceso.
- *Relative Virtual Address* (RVA): especifica el desplazamiento en el espacio de direcciones virtual del proceso relativo a la dirección de inicio de carga. Para convertir una RVA a VA, simplemente hay que sumarle la dirección de inicio de carga en cada ejecución (*ImageBase*).

A continuación se describe la estructura del formato PE detallando las partes que componen un ejecutable. Como ilustra la Figura 2.1 en primer lugar se encuentra la cabecera, compuesta a su vez por Cabecera DOS, Cabecera PE y tabla de secciones, y a continuación se encuentran las secciones, que contienen el código y datos del programa.

2.1.1. Cabecera de DOS

Todo fichero ejecutable con formato PE comienza con una cabecera DOS que contiene un pequeño programa MS-DOS. La funcionalidad de este programa es imprimir un mensaje de error (*"This program cannot be run in DOS mode"*) en caso de ser ejecutado en MS-DOS. La estructura de esta cabecera puede verse en el Listado 2.1. El campo *e_lfanew* en la línea 20, es el offset de la cabecera PE.

```
1  typedef struct _IMAGE_DOS_HEADER {
2      WORD e_magic;         // Magic number
3      WORD e_cblp;
4      WORD e_cp;
5      WORD e_crlc;
6      WORD e_cparhdr;
7      WORD e_minalloc;
8      WORD e_maxalloc;
9      WORD e_ss;
10     WORD e_sp;
11     WORD e_csum;          // Checksum
12     WORD e_ip;
13     WORD e_cs;
14     WORD e_lfarlc;
15     WORD e_ovno;
16     WORD e_res[4];
17     WORD e_oemid;
18     WORD e_oeminfo;
19     WORD e_res2[10];
20     DWORD   e_lfanew;        // offset de la cabecera PE
21   } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Listado 2.1: Estructura de la Cabecera DOS de un ejecutable en Windows.

2.1.2. Cabecera PE

En la cabecera PE es donde se almacenan las características principales del archivo, de tal manera que el sistema operativo Windows sepa cómo manejar el código del ejecutable. Por ejemplo, contiene la ubicación y tamaño de las áreas de código y datos. Esta estructura es idéntica en archivo y en memoria. Esta cabecera está compuesta a su vez por una cabecera de fichero (*FileHeader*) y una cabecera opcional (*OptionalHeader*) como muestra el Listado 2.2.

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

Listado 2.2: Estructura de la Cabecera PE de un ejecutable en Windows.

- **Cabecera.** (Campo *FileHeader* en el Listado 2.2). Esta cabecera almacena una estructura que contiene la información básica del fichero. El Listado 2.3 muestra todos los campos de esta estructura. Los más relevantes son el *NumberOfSections* en la línea 3, que indica el número de secciones que tiene el fichero, y el *SizeOfOptionalHeader* en la línea 7, que indica el tamaño de la cabecera opcional (en bytes).

```
1 typedef struct _IMAGE_FILE_HEADER {
2   WORD  Machine;
3   WORD  NumberOfSections;  //Numero de secciones en la tabla de secciones
4   DWORD TimeDateStamp;
5   DWORD PointerToSymbolTable;
6   DWORD NumberOfSymbols;
7   WORD  SizeOfOptionalHeader; //Tamanyo de la cabecera opcional en bytes
8   WORD  Characteristics;
9 } IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Listado 2.3: Estructura de la Cabecera de fichero de un ejecutable en Windows

- **Cabecera opcional.** (Campo *OptionalHeader* en el Listado 2.2). Esta cabecera no es, en realidad, una cabecera opcional ya que contiene toda aquella información importante que no tiene cabida en la anterior. El Listado 2.4 muestra todos los campos que contiene.

```
1  typedef struct _IMAGE_OPTIONAL_HEADER {
2    WORD                 Magic;
3    BYTE                 MajorLinkerVersion;
4    BYTE                 MinorLinkerVersion;
5    DWORD                SizeOfCode; //tamanyo de la seccion .text en bytes
6    DWORD                SizeOfInitializedData;
7    DWORD                SizeOfUninitializedData;
8    DWORD                AddressOfEntryPoint; //RVA de inicio de ejecucion
9    DWORD                BaseOfCode;
10   DWORD                BaseOfData;
11   DWORD                ImageBase;               //Direccion carga preferente
12   DWORD                SectionAlignment;        //Alineacion de las secciones en memoria
```

```
13    DWORD                FileAlignment;   //Alineacion de las secciones en disco
14    WORD                 MajorOperatingSystemVersion;
15    WORD                 MinorOperatingSystemVersion;
16    WORD                 MajorImageVersion;
17    WORD                 MinorImageVersion;
18    WORD                 MajorSubsystemVersion;
19    WORD                 MinorSubsystemVersion;
20    DWORD                Win32VersionValue;
21    DWORD                SizeOfImage;
22    DWORD                SizeOfHeaders;
23    DWORD                CheckSum;
24    WORD                 Subsystem;
25    WORD                 DllCharacteristics;
26    DWORD                SizeOfStackReserve;
27    DWORD                SizeOfStackCommit;
28    DWORD                SizeOfHeapReserve;
29    DWORD                SizeOfHeapCommit;
30    DWORD                LoaderFlags;
31    DWORD                NumberOfRvaAndSizes;
32    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
33  } IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
34  }
```

Listado 2.4: Estructura de la Cabecera Opcional de un ejecutable en Windows.

Dentro de la cabecera opcional, algunos de los campos más relevantes son los siguientes:

- *AddressOfEntryPoint* (Línea 8 en el Listado 2.4). Es la dirección de inicio de ejecución del programa.
- *ImageBase* (Línea 11 en el Listado 2.4). Es la dirección de carga preferente en memoria.
- *DataDirectory* (Línea 32 en el Listado 2.4). Es el directorio de datos y contiene un vector de punteros a estructuras como la que se muestra en el Listado 2.5. Cada elemento contiene la RVA y el tamaño de un determinado fragmento importante dentro del ejecutable que es necesario localizar de manera rápida. Por ejemplo, el primer y segundo elemento contienen la ubicación y tamaño de la tabla de exportaciones e importaciones respectivamente. La estructura de la Tabla de Importaciones se detalla en la sección 2.1.4.

```
1  typedef struct _IMAGE_DATA_DIRECTORY {
2      DWORD   VirtualAddress;     //RVA
3      DWORD   Size;
4  };
```

Listado 2.5: Estructura de un elemento del directorio de datos en la Cabecera Opcional de un ejecutable en Windows.

Por último, al final de la cabecera, se encuentra la **tabla de secciones**. Esta tabla contiene una entrada por cada sección existente en el fichero. En cada una de ellas se almacenan sus principales características (nombre, ubicación, tamaño).

2.1.3. Secciones

Después de la cabecera del ejecutable, se encuentran los datos y el código del programa divididos en secciones. Existen como mínimo dos secciones, una para datos y otra para código,

aunque típicamente la mayoría de los ejecutables contienen también la sección de importaciones. Con esta división se consigue que cada sección pueda tener unos permisos de acceso diferentes al ser cargado el fichero en memoria. Algunas de las secciones más comunes son:

- *.text*, contiene el código de la aplicación. Por defecto, tiene permiso de acceso de ejecución.
- *.data*, es donde se almacenan los datos inicializados en tiempo de compilación como las variables locales, globales, literales, etc. Por defecto, tiene permiso de acceso de lectura y escritura.
- *.rsrc*, es donde se almacenen los recursos necesarios de la aplicación. Por defecto, tiene permiso de acceso de lectura y escritura.
- *.idata*, contiene la tabla de importaciones. Por defecto, tiene permiso de acceso de lectura y escritura.
- *.edata*, contiene la tabla de exportaciones. Por defecto, tiene permiso de acceso de lectura.

2.1.4. Tablas de Importaciones

Entender cómo funcionan las importaciones en un ejecutable es de vital importancia para conocer el funcionamiento de muchos protectores de software, así como de diversas técnicas de desprotección.

Muchos programas necesitan hacer uso de APIs (*Application Programming Interface*) definidas en diferentes librerías (DLLs). Sin embargo, cuando el ejecutable es creado no se puede saber en qué direcciones estarán cargados dichos módulos en el momento de la ejecución. Además, en cada máquina y/o versión del sistema operativo se encontrarán en una posición diferente dentro de la librería.

El mecanismo del sistema operativo Windows para solventar este problema utiliza la denominada tabla de importaciones. El funcionamiento es el siguiente:

1. Al crear el ejecutable, el compilador crea la tabla de importaciones y la rellena con los nombres de todos las librerías que será necesario importar así como de las funciones que necesita de cada una.
2. Al cargar el programa en memoria para su ejecución, el *Windows loader*, leerá de esta tabla las librerías que necesita y las cargará en la memoria del ejecutable. A continuación buscará las direcciones de las funciones a importar y las escribirá en esta tabla.

Para poder localizar esta tabla dentro de un ejecutable, hay que comenzar por el directorio de datos de la cabecera opcional, línea 32 en el Listado 2.4. El segundo elemento de este vector (*DataDirectory[1]*) es donde da comienzo la tabla de importaciones. Este elemento es el directorio de importaciones y apunta a otro vector que contiene una entrada (descriptor) por cada DLL que el programa necesite importar. Cada descriptor contiene la estructura mostrada en el Listado 2.6. Nótese que no se almacena el tamaño de este vector de descriptores, por eso

para indicar el final, el último elemento será un descriptor con todos sus campos a 0.

```
1  typedef struct _IMAGE_IMPORT_DESCRIPTOR {
2    DWORD         OriginalFirstThunk;
3    DWORD         TimeDateStamp;
4    DWORD         ForwarderChain;
5    DWORD         Name;
6    DWORD         FirstThunk;
7  } IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

Listado 2.6: Estructura descriptor de importaciones en un ejecutable en Windows.

Los campos más importantes de esta estructura son:

- *Name* (Línea 5 en el Listado 2.6). Es un puntero al nombre de la DLL a importar.
- *OriginalFirstThunk* (Línea 2 en el Listado 2.6). Es un puntero a un vector de estructuras IMAGE_THUNK_DATA de tamaño DWORD (4 bytes). Contiene tantas entradas como funciones a importar, el final del vector se indica mediante una entrada con valor 0. Se suele referir a este vector con el nombre de Import Name Table (INT).
- *FirstThunk* (Línea 6 en el Listado 2.6). Es un puntero a un vector de estructuras IMAGE_THUNK_DATA de tamaño DWORD (4 bytes). Contiene tantas entradas como funciones a importar, el final del vector se indica mediante una entrada con valor 0. Se suele referir a este vector con el nombre de Import Address Table (IAT).

Es decir, existen dos vectores que son esencialmente idénticos, sin embargo los punteros IMAGE_THUNK_DATA del *FirstThunk* (IAT) tienen un doble propósito: cuando el ejecutable está en disco contienen (al igual que el *OriginalFirstThunk)* o bien el ordinal o el nombre de la función a importar (líneas 5 y 6 del listado 2.7), pero cuando el ejecutable es cargado en memoria, el loader sobreescribe este vector con la direcciones de las funciones importadas.

```
1  typedef struct _IMAGE_THUNK_DATA32 {
2      union {
3          DWORD ForwarderString;
4          DWORD Function;             // direccion de la funcion a importar
5          DWORD Ordinal;
6          DWORD AddressOfData;   // apunta a una estructura IMAGE_IMPORT_BY_NAME
7                          //con el nombre de la API
8      } u1;
9  } IMAGE_THUNK_DATA32;
```

Listado 2.7: Estructura elemento de la tabla de importación

La Figura 2.3 ilustra el funcionamiento de las tablas de importaciones con un ejemplo (*kernel32.ReadFile, kernel32.WriteFile*). Cuando el archivo está en disco, tanto la IAT como la INT contienen un vector de punteros a estructuras *IMAGE_IMPORT_BY_NAME*, cada una contiene el nombre de una de las funciones a importar. Cuando el archivo es cargado en memoria, el Windows loader sobreescribe el contenido de la IAT con las direcciones de memoria de las funciones.

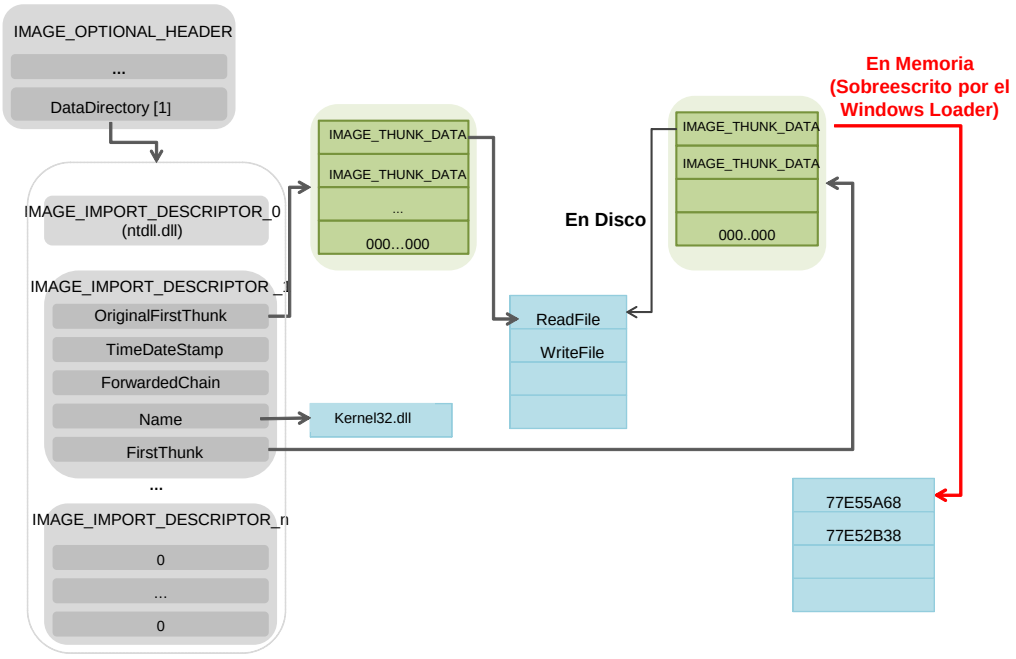

Figura 2.3: Funcionamiento de las tablas de importaciones en un archivo ejecutable (PE) en Windows.

Cuando el programa es cargado en memoria, primero se lee de la segunda posición del directorio de datos la lista de DLLs necesarias y se cargan en memoria, después se leen los nombres de las funciones a importar de cada DLL de la INT y, para cada una de ellas, se escribe la dirección real en la IAT.

De esta manera se consigue, por un lado, asegurar que el mismo ejecutable pueda funcionar correctamente en diferentes versiones de un mismo sistema operativo ya que las funciones estarán localizadas en diferentes direcciones de memoria. Y por otro lado, que todas las referencias a funciones importadas se hagan a través de la IAT. De esta manera el loader de Windows no necesita reparar una a una todas las referencias a funciones externas sino que sólo es necesario escribir la dirección en la IAT para que todo funcione correctamente.

Muchos protectores de software realizan modificaciones en esta estructura con el objetivo de impedir que el programa sea reconstruido a partir de la imagen en memoria.

2.2. Manejo de Excepciones en Windows

Las excepciones son eventos que provocan la ejecución de un código adicional fuera del flujo normal de ejecución denominado manejador de excepciones (en inglés *exception handler*). Existen dos tipos de excepciones: hardware y software. Una excepción hardware es iniciada por el procesador y se produce normalmente cuando éste ejecuta una operación no válida (e.g. acceso a memoria no válido, división entre 0, instrucción no válida). Una excepción software es iniciada por la aplicación o el sistema operativo.

En Microsoft Windows existe un manejo estructurado de excepciones (*structured exception handler*), esto significa que el sistema operativo distribuye las excepciones de manera organizada. Cada hilo de ejecución (*thread*) tiene una lista de manejadores de excepciones estructuradas (SEH). Cuando una excepción ocurre se llama al primer manejador de la lista, que puede manejarla o no. En caso de que no la maneje, se busca el siguiente manejador de la lista y así sucesivamente. Esto es posible ya que cada manejador incluye una referencia al manejador anterior [Eil05]. Esta lista enlazada de manejadores de excepciones es almacenada en la pila de ejecución y al primer elemento se puede acceder desde el bloque de información del hilo. Si no existe ningún SEH que maneje la excepción se ejecuta el *UnhandledExceptionFilter* y normalmente terminará la ejecución con un mensaje de error si este último manejador no trata la excepción. En las últimas versiones de Windows (Windows XP en adelante) se ha añadido otra lista de manejadores de excepciones *Vectored Exception Handler* que se ejecuta antes del SEH. La Figura 2.4 ilustra el manejo de excepciones en Windows.

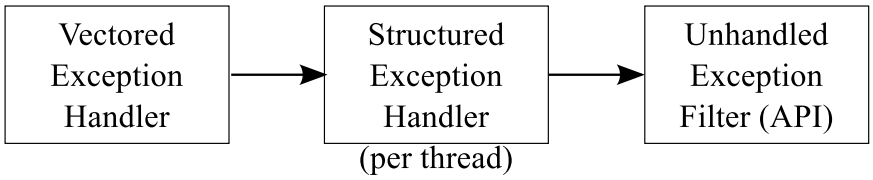

Figura 2.4: Manejo de excepciones en Windows.

2.3. Herramientas de Ingeniería Inversa

Existen diferentes herramientas en el ámbito de la ingeniería inversa, cada una con un propósito determinado. A continuación se describen brevemente los tipos de herramientas existentes tanto para análisis estático como para análisis dinámico [Eil05].

- **Desensambladores**. Este tipo de herramientas sirven para el análisis estático del código. Convierten el codigo máquina (binario) en lenguaje ensamblador más entendible por el ser humano. El conjunto de instrucciones máquina y su formato son específicos de cada plataforma, por lo tanto también existen desensambladores específicos para cada plataforma. Una de las herramientas de este tipo más potente y ampliamente utilizada es el IDAPro.
- **Decompiladores**. Este tipo de herramienta sirve para el análisis estático del código. El proceso de decompilación consiste en convertir el código de un lenguaje de bajo nivel a otro de alto nivel. Este tipo de herramientas funcionan especialmente bien con lenguajes como Java o .Net, ya que son lenguajes interpretados.
- **Depuradores** (en inglés *debuggers*). Son herramientas para el análisis dinámico del código. Permiten tracear la ejecución de un determinado proceso. Los depuradores fueron creados con el objetivo de encontrar fallos en un programa. Sin embargo, hoy en día existen depuradores creados específicamente para tareas de ingeniería inversa. Las principales funcionalidades de un debugger para poder ser utilizado en la ingeniería inversa son:

1. Incluir un desensamblador, esto hace posible visualizar tanto el código como los saltos de manera clara.
2. Puntos de ruptura software y hardware (en inglés *breakpoints*): todo debugger debe incluir esta funcionalidad. Los breakpoints software son instrucciones especiales que el debugger añade en el programa de modo que cuando se ejecuta esta instrucción se para la ejecución y se transfiere el control al debugger. Los hardware breakpoints, son una funcionalidad de la CPU que permiten parar la ejecución del programa cuando se accede a una determinada dirección transfiriendo el control al debugger.
3. Visualización de registros y memoria, un debugger debe permitir visualizar los registros importantes de la CPU así como el sistema de memoria. También es muy útil disponer de la visualización de la pila de ejecución.

Los debuggers más convencionales operan en modo usuario. Algunos de los más utilizados actualmente son OllyDbg, WinDbg, IDAPro. Existen también debuggers en modo kernel que son más potentes ya que se instalan como un componente más del núcleo del sistema operativo, por ejemplo, SoftICE.

El **OllyDbg** [Yus13] es una de las herramientas utilizadas durante el desarrollo de este trabajo. Es uno de los debuggers más ampliamente utilizados en la ingeniería inversa. Incluye un desensamblador son múltiples funcionalidades, así como numerosas vistas (código, registros CPU, pila de ejecución, memoria y lista de funciones importadas entre otras). También permite la modificación del programa mientras está siendo ejecutado (*hot patching*).

- **Dumpers**, son herramientas que permiten hacer una copia de la imagen en memoria del programa. Existen numerosas herramientas de este tipo, por ejemplo, Dumpbin, PEView, PEBrowse Professional. El OllyDbg también incluye una funcionalidad adicional que permite dumpear un programa.

- **Editores PE**, son editores para ficheros PE. Durante el desarrollo de este proyecto se ha utilizado el editor **PE Bear**, que permite visualizar varios ficheros PE en paralelo. Incluye también un desensamblador aunque no tan potente como el del OllyDbg. También permite visualizar la disposición de la distintas secciones.

- **Reconstructor de Tabla de Importaciones**, es un tipo de herramienta que permite reconstruir las Tablas de Importaciones de un ejecutable. Para este trabajo se ha utilizado el **Import Reconstructor**, que se puede utilizar para restaurar los descriptores de las importaciones en el directorio de datos y la IAT. También permite inyectar en el ejecutable las direcciones reales de las APIs.

Capítulo 3

Trabajo Relacionado

En la literatura no existe ningún trabajo previo en el que se evalúen distintos protectores de software a nivel de calidad, fiabilidad y rendimiento como se realiza en este trabajo. La mayoría de los trabajos relacionados con este tema se encuentran en el ámbito de la detección de malware.

Como se ha mencionado anteriormente, gran parte del malware que se difunde por la red hoy en día se encuentra empacado [BBN12, GFC08] para evitar ser detectado por los sistemas anti-virus, pero el hecho de que un programa esté empacado no significa necesariamente que sea maligno. Por tanto, los sistemas anti-virus se enfrentan al reto de tener que detectar si un programa empacado es maligno o no. Para evitar falsos positivos es necesario desempacar el programa antes de que pueda ser analizado por un escáner anti-virus. Realizar la tarea de desempacado manualmente puede resultar muy costoso, es por eso que existen numerosas investigaciones que tienen como objetivo encontrar un algoritmo efectivo de desempacado automático [RHD+06, KPY07, MCJ07, GFC08, JCL+10].

En [RHD+06] proponen un método que consiste en desensamblar el programa para construir un modelo estático del sistema y después tracear la ejecución hasta encontrar una instrucción ejecutada que no estaba en el modelo original. En [KPY07] proponen monitorizar la ejecución del código traceando aquellas instrucciones de escritura en memoria. Estas posiciones son marcadas y cuando una de ellas se ejecuta se considera que se ha llegado al punto de entrada del programa original. Sin embargo, con ninguno de estos dos métodos se consigue un buen rendimiento al realizarse el traceo a nivel de instrucción.

En [MCJ07] se tracean las páginas ejecutadas al mismo tiempo que se monitorizan las llamadas al sistema. Se considera que se ha llegado al punto de entrada original cuando se ejecuta una página que ha sido modificada y se realiza una llamada al sistema peligrosa (por ejemplo modificar o borrar una clave de registro, crear un proceso o escribir en memoria). En [GFC08] se propone un desempacado genérico llamado *Justin*. La técnica propuesta consiste en, basándose en heurísticas, detectar el fin de la rutina de desempacado y llamar al escáner anti-virus con el volcado de memoria en ese momento. En [Sun10] se desarrolla un framework para análisis de packers y desempacado automático denominado REFORM (*Reverse Engineering for Obfuscation Removal*). Para la detección del packer se mide la distribución del nivel de aleatoriedad que es comparado con un patrón predefinido para cada packer. En [JCL+10] proponen un método

para encontrar el punto de entrada del programa original basándose en el análisis de la entropía.

En [SIJ+11] proponen un método para detectar si es un programa está empacado y reconocer el packer. Este método está basado en generar un patrón dentro de la rutina de restauración que deba estar presente en todos los ejecutables protegidos con ese mismo packer. En [STF09] y [STMF09] utilizan información estructural del programa para detectar si está empacado o si puede ser maligno.

A diferencia de los anteriores, en [DP10] proponen un método para extraer y poder analizar la rutina de desempacado en un programa auto modificable. Para ello se hace un análisis offline de la traza de ejecución.

El análisis de malware es realizado normalmente dentro de un entorno controlado (Máquina Virtual o Sandbox) para evitar que el sistema sea dañado. Es por eso que es también habitual que los autores de malware incluyan en sus programas técnicas para detectar estos entornos y así modificar su comportamiento. En [RG14] se propone un nuevo método de análisis de malware que integra la instrumentación dinámica de ejecutables (DBI) con el análisis automático para evitar las técnicas anti-máquina virtual del malware. La ventaja de esta método es que, a día de hoy, todavía no existe malware que incluya técnicas anti-DBI.

El trabajo más cercano a este proyecto se ha encontrado en [KLC+10]. En él se implementan dos protectores de software para sistemas embebidos en el entorno Linux. Debido a que los sistemas embebidos tienen recursos más limitados, se analiza el rendimiento de estos protectores en términos de tamaño del código, tiempo de ejecución y consumo de energía.

Contribución. Con este trabajo se aporta una evaluación cualitativa y cuantitativa de los protectores de software más utilizados hoy en día, concretamente se han seleccionado 21 protectores para el estudio. Para la evaluación cualitativa se realiza un análisis de las protecciones que ofrecen y de la dificultad de eliminarlas. La evaluación cuantitativa muestra el impacto en el rendimiento (tiempo de ejecución y memoria consumida) de la ejecución de un programa protegido frente al mismo programa sin proteger y también evalúa la fiabilidad de las herramientas analizadas.

Capítulo 4

Taxonomía de las Protecciones de Software

Una vez introducidos los conceptos básicos, en este capítulo se describen y categorizan las distintas técnicas anti-ingeniería inversa que implementan las herramientas protectoras de software. Aunque existen numerosas técnicas y surgen nuevas continuamente, se detallan las más usadas por los protectores de software actuales. En función del tipo de ataque para el que sirven de protección, se pueden distinguir dos tipos de técnicas principales:

- **Anti-Análisis Estático**: protección de copia, compresión, cifrado y anti-desensamblado.
- **Anti-Análisis Dinámico**: anti-depuración (en inglés *anti-debugging*), anti-volcado (en inglés *anti-dumping*), anti-virtualizacion y anti-parcheo (en inglés *anti-patching*).

4.1. Protección Copia

El objetivo de las técnicas de protección de copia es impedir el uso de una copia del programa de manera ilegal. Una técnica es controlar el modo en que el programa es usado. Esto puede requerir controlar el número de veces, el tiempo durante el cual puede ejecutarse un programa o limitar el número de funcionalidades que tiene disponibles dependiendo de la licencia de la que se disponga [Eil05]. Para conseguir esto, muchos protectores implementan un sistema de gestión de licencias. Otra técnica es utilizar un número de registro de tal manera que debe introducirse al instalar la aplicación para poder ejecutarse. También se puede relacionar la copia con un determinado hardware de manera que el programa sólo funcione en aquella máquina donde el hardware esté disponible.

4.2. Compresión

La compresión del código es una de las principales técnicas anti-análisis estático. Algunos de los algoritmos de compresión más utilizados son aPLib [Sof13] y LZMA [Wik13b]. Además

de comprimir el código, se pueden usar otras ténicas de compresión adicionales, como comprimir los recursos o eliminar al sección *.reloc*. Aunque en la mayoría de los packers protectores y virtualizadores las técnicas de compresión no hacen que el tamaño del ejecutable sea menor que el del original, si no que todo el código de protección y virtualización añadido hace que normalmente sea mayor. La simple compresión del ejecutable ofrece una protección muy débil, ya que en cuanto la rutina de descompresión termina, el código original está íntegramente accesible en memoria sin ninguna protección adicional.

4.3. Cifrado

En muchas ocasiones las técnicas de compresión se utilizan conjuntamente con el cifrado del código. Los algoritmos de cifrado pueden ir desde un simple bucle XOR[1] a otros mucho más complejos. Algunos protectores cifran solamente partes del código que son especialmente sensibles a ataques. Otros descifran el código bajo demanda en tiempo de ejecución, con lo cual si se utiliza un desensamblador no se conseguirá obtener el código completo del programa. Se puede utilizar una contraseña para el cifrado que puede almacenarse o bien dentro de propio programa o bien solicitándosela al usuario al ejecutarse. También existen packers polimórficos que tienen diferentes algoritmos de cifrado para cada instancia creada [Yas07].

4.4. Anti Desensamblado

Son técnicas utilizadas con el objetivo de hacer más difícil el proceso de desensamblado del código o hacer que el código desensamblado no sea útil.

Existen numerosas técnicas de anti desensamblado, como por ejemplo, poner un salto que nunca se ejecuta a una instrucción basura (*junk code*), insertar código que nunca llega a ejecutarse (*dummy code*) o la permutación del código, es decir, transformar instrucciones simples en otras equivalentes pero más complejas [BBN12].

4.5. Anti Debugging

Son técnicas para impedir la utilización de herramientas de depuración. A continuación se describen brevemente las técnicas de protección frente a debuggers más comunes [Fer11, Shi09, BBN12].

4.5.1. Basada en APIs

Estas técnicas comprueban la existencia de un debugger utilizando información y funciones del sistema operativo.

[1] Cifrado basado en el operador binario XOR, fácil de descifrar por su propiedad conmutativa: $a \oplus b = c \Rightarrow b \oplus c = a, a \oplus c = b$.

- *isDebuggerPresent*. Es una funcion de la librería kernel32 que comprueba el flag *BeingDebugged* en el bloque de entorno del proceso (PEB: *Process Enviroment Block*). Si este valor es distinto de cero, existe debugger.

- *CheckRemoteDebuggerPresent*. Es una función de la librería kernel32 que comprueba si el proceso que se le pasa como parámetro está siendo debuggeado o no. Internamente, llama a la función *NTQueryInformationProcess* de la librería ntdll.

- *FindWindow*. Esta función se utiliza para detectar si hay una ventana que corresponde con el nombre de algún debugger conocido (por ejemplo OllyDbg, SoftICE).

- *NTQueryInformationProcess*. Es una función de la librería ntdll. Si se usa el parámetro *ProcessDebugPort*, devuelve el puerto asignado si el proceso está siendo debugeado o cero en caso contrario. También se puede llamar con el parámetro *ProcessDebugFlags*. En este caso, si el valor de retorno es cero indica que hay debugger. Por último, si se llama con el parámetro *ProcessDebugObjectHandle* y devuelve un valor distinto a cero indica que el proceso está siendo debuggeado.

- *NtSetInformationThread*. Es una función de la librería ntdll cuyo propósito es modificar un determinado hilo de ejecución. Se comporta de tal manera que si se le pasa como segundo parámetro el valor correspondiente a *ThreadHideFromDebugger* y existe un debugger, lo desconecta y termina el proceso.

- *SeDebugPrivilege*. Por defecto, un proceso tiene deshabilitado el token SeDebugPrivilege, pero si está siendo debuggeado está activado. Esta técnica consiste en intentar abrir otro proceso como csrss.exe con derechos de PROCESS_ALL_ACCESS. Si tiene el privilegio puede hacerlo y significa que hay debugger.

- *CreateTool32Snapshot32Next*. Es una función de la librería kernel32 que devuelve todos los procesos que se están ejecutando en ese momento. Esta técnica consiste en comprobar si alguno de los procesos que se están ejecutando corresponde a un debugger conocido (p.ej. “ollydbg.exe”). También se puede utilizar la función *QuerySystemInformation* de la librería ntdll.

- *OutputDebugString*. Es una función de la librería kernel32 que se comporta de manera diferente dependiendo de si hay debugger o no. En concreto, la función del kernel32 *GetLastError* devuelve 0 si hay un debugger y distinto de cero en caso contrario. Además de esto, se usa también para detectar la herramienta OllyDbg, ya que existe un bug que provoca que esta herramienta termine cuando se realiza una llamada a esta función con un cierto valor como parámetro.

4.5.2. Proceso e Hilos

Estas técnicas se basan en la comprobación de la información del proceso o hilos de ejecución. Las más comunes son:

- *NtGlobalFlag,ProcessHeapFlag* . Si un proceso está siendo debuggeado en el offset correspondiente de cada uno de estos flags en la estructura PEB ambos están activos.
- *Self Debugging*. Consiste en crear otro proceso que debugee al proceso padre de tal modo que como teóricamente un proceso sólo puede estar siendo debuggeado una vez, no puede existir un nuevo debugger en ese proceso.
- Multihilo. Consiste en crear múltiples hilos que cada cierto tiempo comprueban la existencia de debuggers.

4.5.3. Hardware y registros

A diferencia de las dos anteriores, este tipo de protecciones realizan comprobaciones directamente en los registros del procesador. Los Hardware Breakpoints son implementados en el procesador directamente. En la arquitectura x86 existen 8 registros dedicados DR0 - DR7: los 4 primeros contienen direcciones de memoria, cuando una de ellas es ejecutada se envía una señal (INT1) y la CPU para la ejecución, los siguientes 4 contienen un indicador de si el breakpoint está habilitado o no. Sólo pueden usarse, por tanto, 4 breakpoints hardware simultáneamente. Las funciones *GetCurrentThreadContext/SetCurrentThreadContext* permiten acceder a estos registros y por tanto comprobar si el programa está siendo debuggeado o no o inactivar los breakpoints.

4.5.4. Tiempo y Latencia

Otra manera de comprobar si hay un debugger es comprobar el tiempo o latencia entre instrucciones. La función *_rdstc* permite acceder al registro donde se almacena el número de ticks del procesador desde que se inició el sistema. Este registro existe en todos los procesadores Intel Pentium. Realizando dos llamadas a esta función y comparando el tiempo transcurrido con un valor fijo predefinido, se puede comprobar si el programa está siendo traceado. De manera similar se pueden utilizar las funciones *GetTickCount* o *timeGetTime* que devuelven el valor en milisegundos desde que el sistema fue iniciado en el primero caso, y el tiempo del sistema en el segundo.

4.5.5. Basada en Excepciones

Este tipo de técnicas se basan en el hecho de que el debugger atrapará determinadas excepciones en lugar de pasárselas a la aplicación debuggeada. Algunos debuggers pueden configurarse para pasar las excepciones al proceso, pero si esta opción no está activada, se podrá detectar la existencia de un debugger comprobando si las excepciones no son manejadas por la aplicación.

- *Software Breakpoints (0xCC - INT3)*. A diferencia de los Hardware Breakpoints, los Software Breakpoints utilizan la interrupción INT3. Cuando se pone un breakpoint el debugger sustituye el primer byte de la instrucción por 0xCC (*breakpoint opcode*). Cuando se ejecuta la instrucción con un breakpoint se restaura este byte con el valor original de manera que la ejecución puede continuar. Esta técnica consiste en comprobar la existencia de instrucciones 0xCC o sustituirlas por ceros.

- *UnhandledExceptionFilter()*. Cuando ocurre una excepción y no hay ningún manejador para ella en la cadena de manejadores de excepciones estructurada (SEH), en caso de no haber debugger se llama a la función *UnhandledExceptionFilter()*, y en caso de haber debugger se pasa la excepción al debugger. Esta técnica consiste en añadir una función mediante *SetUnhandledExceptionFilter()*, después generar una excepción y si esta función no se llega a ejecutar significa que hay debugger.

4.6. Anti Dumping

Son técnicas para impedir que el volcado del proceso desde memoria funcione correctamente. A continuación, se describen varias de las técnicas más utilizadas [Fer08].

- **Ofuscación de la Tabla de Importaciones.** Consiste en modificar algún elemento en la tabla de importaciones para evitar que el proceso de carga de Windows funcione correctamente. Por ejemplo, modificar la IAT de manera que la entrada correspondiente a una función no apunte a la DLL si no a una sección creada en tiempo de ejecución, y desde ahí saltar a la API real. A veces también se emulan las primeras instrucciones de la API en esta sección creada dinámicamente.

- **Nanomites.** Consiste en sustituir los saltos en el código original por la excepción INT3. Esta técnica requiere otro proceso que debugee. El proceso debugeador almacena en una tabla una entrada por cada salto, cuando se produce la INT3 se busca en la tabla con la dirección donde se ha producido la excepción, si se encuentra se compara el tipo de salto con la CPU flag para saber si el salto es tomado o no. En caso de que sea nanomite y el salto se tome, se toma de la tabla la dirección del salto y se continúa la ejecución desde allí. Si el salto no es tomado, se toma de la tabla el número de bytes de la instrucción de salto para saltárselos.

- **Robo de bytes.** Consiste en colocar parte del código fuera de las secciones del programa original junto con el código del protector o en algún otro lugar del programa creado dinámicamente. Si esta protección está implementada la ejecución del programa original comienza en un *falso* OEP, ya que cuando se llega allí los bytes robados ya se han ejecutado. A veces también se introduce código basura entre los bytes robados para que sea más difícil localizarlos y restaurarlos.

- **Tampering.** Consiste en realizar algún tipo de modificación para impedir que el programa pueda ser reconstruido, por ejemplo, borrar o modificar alguna parte de la cabecera.

- **Páginas Guarda.** Esta técnica se basa en el hecho de que las páginas marcadas como Páginas Guarda, generan una excepción *EXCEPTION_GUARD_PAGE* al ser accedidas. Este hecho se puede aprovechar para implementar, por ejemplo, un sistema de descompresión/descifrado bajo demanda. Cuando se carga el ejecutable en memoria en lugar de descomprimirse/descifrarse totalmente, se marcan las páginas que no se necesitan inicialmente. Cuando se accede a una de estas páginas marcadas, se genera una excepción y se procede a descomprimir o descifrar el código.

- **Virtualización.** Es una de las técnicas más potentes que existen actualmente. En condiciones normales, la CPU ejecuta instrucciones binarias que son traducidas del código nativo x86. La máquina virtual es una capa adicional que ejecuta bytecode por encima de la CPU, de tal manera que se comporta como una CPU manteniendo su estado interno (la pila puede ser implementada con un vector y los registros mediante variables). De esta manera, el código es reescrito para que sea ejecutado en la máquina virtual embebida. El uso de una máquina virtual tiene como ventaja que la lógica de la aplicación queda escondido en todo momento, ya que el lenguaje bytecode es un lenguaje diferente al ensamblador y además específico de cada protector. Adicionalmente, esta técnica puede combinarse con la ofuscación del código.

4.7. Anti Virtualización

Estas técnicas consisten en impedir que el programa pueda ser ejecutado en una máquina virtual. Las máquinas virtuales son utilizadas normalmente para analizar malware de manera que la ejecución se lleve a cabo en un entorno controlado sin riesgo de dañar el sistema. Existen diferentes tipos de máquinas virtuales:

- Basadas en hardware: utilizan la CPU para ejecutar determinadas instrucciones a velocidad nativa. Por ejemplo VMWare y VirtualBox.

- Basadas en software: ejecutan las instrucciones equivalentes a cada instrucción CPU en software, por ejemplo Hydra, Bochs, QEMU, Sandbox y Atlantis.

Existen técnicas para detectar máquinas virtuales específicas como VMWare. Por ejemplo, como es una máquina virtual basada en hardware tiene que replicar estructuras de datos como la Tabla Local Descriptora (*LDT*), que normalmente no es utilizada por Windows, con lo que si existe algún valor en ella se puede detectar. Por otro lado, en una ejecución normal no se puede tener acceso a instrucciones privilegiadas (IN/OUT) en modo usuario y esto causaría una excepción, pero con una máquina virtual VMWare no se produce esta excepción ya que las usa para comunicarse con su propio software (canal de comunicación host/guest). Existen muchos otros métodos de detección dependiendo del sistema operativo o del harware existente y también específicos de otras máquinas virtuales [Fer06, Fer08].

4.8. Anti Patching

Patching es el proceso de modificar el código binario con el objetivo de alterar su comportamiento de alguna manera [Eil05]. Este tipo de técnicas consisten en comprobar si ha habido algún tipo de modificación en la memoria del proceso. Para comprobar la integridad del fichero se puede utilizar el algoritmo CRC o un código hash. Normalmente se comprueba la integridad de una determinada función antes de ser llamada y si no coincide con el cálculo del programa original el programa deja de ejecutarse.

Estas técnicas son utilizadas con frecuencia para eliminar las protecciones anti copia. También puede utilizarse para detectar un debugger. Como ya se ha comentado, al colocar un software breakpoint el debugger sustituye los primeros bits de la instrucción por 0xCC, con lo cual al realizar una modificación, ésta puede ser detectada.

Capítulo 5

Selección de Protectores de Software y Benchmark Comparativo

En la primera parte de este capítulo se describe el proceso de selección de protectores que forman parte de la comparativa. Después, se detalla el proceso de creación del benchmark y las aplicaciones seleccionadas para el mismo.

5.1. Selección de protectores para el estudio

Es difícil medir con exactitud el número de protectores existentes. Fabricantes de anti-virus como Symantec han reportado la existencia de más de 200 familias de protectores de software [GFC08]. Dependiendo de sus características, los protectores de software se pueden clasificar en 3 tipos.

- **Compresores**: que comprimen el programa original para reducir el tamaño y dificultar el análisis estático.
- **Protectores**: que implementan una serie de protecciones contra ataques de ingeniería inversa. Normalmente las herramientas protectoras son también compresoras.
- **Virtualizadores**: que implementan una protección específica que consiste en reescribir el código original para que sea compatible solamente con la máquina virtual embebida.

También existen herramientas tipo **Bundler** que permiten construir un único ejecutable a partir de los archivos de la aplicación.

Los protectores que se han seleccionado para este trabajo son aquellos más utilizados actualmente en el sistema operativo Windows de los que se ha encontrado información disponible. Se han considerado solamente herramientas para Windows ya que la mayoría de ellas están destinadas a este sistema operativo. Se han seleccionado tanto herramientas gratuitas como de pago de todos los tipos arriba mencionados. La Tabla 5.1 muestra la lista de todos los protectores con lo que se ha trabajado, la versión utilizada, el precio de la licencia (en caso de ser una herramienta

	Versión	Última *Release*	Licencia	Tipo
ACProtect	2.1.0	09/2008	149$	Compresor&Protector
Armadillo	9.62	06/2013	aprox. 299$	Compresor&Protector
ASPack	2.32	08/2012	34€- 208€	Compresor&Protector
ASProtect	1.69	09/2013	117€- 347€	Compresor&Protector
Enigma	3.7	06/2013	149$- 299$	Protector&Virtualizador
EXE Stealth	4.19	06/2011	296,31€	Compresor&Protector
ExeCryptor	2.3.9	03/2006	135$- 749$	Compresor&Protector
ExPressor	1.8	01/2010	14$- 360$	Compresor&Protector
FSG	2.0	05/2004	Gratuita	Compresor
MEW	11	11/2004	Gratuita	Compresor
Obsidium	1.5	10/2013	139€- 289€	Protector&Virtualizador
PECompact	3.02.2	02/2010	89,95$499$	Compresor&Protector
PELock	1.06	01/2012	89$- 289$	Compresor&Protector
PESpin	1.33	05/2011	Gratuita	Compresor&Protector
Petite	2.3	02/2005	170€	Compresor&Protector
Smart Packer	1.9	06/2013	59€- 399€	Compresor& Bundler
TeLock	0.98	08/2001	Gratuita	Compresor&Protector
Themida	2.2	10/2013	199€- 399€	Protector&Virtualizador
UPX	3.91	02/2013	Gratuita	Compresor
VMProtect	2.13	05/2013	99$- 399$	Protector&Virtualizador
Yodas Protector	1.03.3	08/2006	Gratuita	Compresor&Protector

Tabla 5.1: Protectores seleccionados para el estudio.

comercial, que puede variar en función de si es una licencia para uso personal o comercial), así como el tipo de protector.

Dado que la mayoría de las herramientas seleccionadas son comerciales y requiren una licencia para su uso, se ha utilizado la versión de prueba disponible en la página web del fabricante. En algunos casos, esta versión gratuita no tiene todas las funcionalidades disponibles y normalmente incluye un mensaje de aviso al ejecutar un programa protegido, lo que imposibilita la automatización de las pruebas de análisis de rendimiento. Para solucionar esto, se contactó con los fabricantes explicando la investigación que se estaba llevando a cabo y solicitando una versión de su producto para ser utilizada con fines meramente académicos. Afortunadamente, se obtuvo respuesta de muchos de ellos (aunque no de todos). Algunos proporcionaron una licencia para poder usar la version íntegra de la herramienta (ExPessor, SmartPacker), y otros (Obsidium, EXEStealth) se ofrecieron a empaquetar las aplicaciones para las pruebas (sombreados en gris la Tabla 5.1).

5.2. Creación del Benchmark

El objetivo de este benchmark es realizar una evaluación cuantitativa de los protectores de software. Los resultados del benchmark permitirán hacer una comparativa de las herramientas protectoras de software a nivel de fiabilidad y rendimiento (tiempo de ejecución y consumo de memoria).

Las aplicaciones utilizadas en el benchmark se muestran en la Tabla 5.2 y son todas de uso intensivo de CPU. La primera columna indica el nombre de la aplicación, después la versión utilizada para el benchmark, el lenguaje de programación en el que están desarrolladas, el tipo de aplicación y el tipo de cálculo que realiza la aplicación (real, entero o con gran demanda de E/S). Finalmente, se indica si esa aplicación ha sido utilizada en algún otro benchmark.

Nombre	Versión	Lenguaje	Tipo	Tipo de Cálculo	Origen
GNU go	3.8	C	IA - Juegos	Entero	SPEC CINT 2006
hmmer	3.0	C	Genética	Entero	SPEC CINT 2006
h264ref	18.5	C	Compresión video	Entero	SPEC CINT 2006
mcf	1.3	C	Combinatoria	Entero	SPEC CINT 2006
namd	2.8	C++	Biología, simulación de moléculas	Real	SPEC CFP 2006
povray	3.7	C++	Renderización	Real	SPEC CFP 2006
calculix	2.6	Fotran90 & C	Mecánica Estructural	Real	SPEC CFP 2006
palabos	1.4.	C++	Dinámica de fluidos (Método LBM)	Real	Propio
aescrypt	3.0.9	C	Criptografía, cifrado	E/S	Propio
bzip2	1.0.5	C	Compresión	E/S	SPEC CINT 2006
ffmpeg	0.10	C	Conversión de formatos video/audio	E/S	Phoronix Test Suite
md5	1.2	Borland Delphi 7	Criptografía, Hash	E/S	Propio

Tabla 5.2: Aplicaciones seleccionadas para el benchmark.

El benchmark realiza mediciones del tiempo de CPU y de la memoria consumida en la ejecución de cada una de estas aplicaciones tanto protegidas con cada uno de los protectores como no protegidas. En un principio se pensó realizar mediciones del tiempo de arranque (*start-up*) de las aplicaciones protegidas para poder calcular el tiempo de ejecución de la rutina de desempacado. No obstante, dado que muchas protecciones tienen lugar durante la ejecución de la aplicación (descompresión/descifrado bajo demanda y nanomites entre otras) se decidió hacer la medición del tiempo total de ejecución.

Para la medición de **consumo de memoria** por proceso, se ha utilizado la función *GetPro-*

cessMemoryInfo de la librería kernel32 de Windows. Esta función devuelve una estructura que contiene diferentes mediciones sobre la memoria consumida por el proceso. Los valores de los que se ha tomado medición en el benchmark son *PeakPagefileUsage* y *PeakWorkingSetSize*. Este último es el pico máximo de memoria RAM utilizada por el proceso y es el valor que se ha utilizado en la comparativa.

Para la medición del **tiempo de CPU** de un proceso se ha utilizado la función *GetProcessTimes* de la librería *kernel32* de Windows. Devuelve el tiempo de reloj (*Wall Clock*) de creación y fin del proceso, así como el tiempo de CPU en modo kernel y en modo usuario. Este último es el que se ha utilizado para la comparativa. La Figura 5.3 describe el hardware y el software en el entorno en el que se han llevado a cabo la ejecución del benchmark.

Nombre CPU	Intel®Core™2 Duo CPU P7450
Características	2.13 GHz, 1066 MHz bus
CPU(s)	2 cores
Memoria RAM	4 GiB (DDR2 SDRAM (2 x 2 GB))
Disco Duro	Serial ATA 500 GB 5400rpm
Sistema Operativo	Windows 7 Home Premium 64bit
Compilador C++	mingw32-gcc-g++ 4.8.1.4

Tabla 5.3: Hardware y software utilizado en las pruebas.

Los pasos seguidos para la creación del benchmark han sido:

1. Obtener las aplicaciones de prueba seleccionadas, o bien los binarios para Windows (en caso de estar disponibles) o bien el código fuente. Una vez obtenidas todas las aplicaciones, se han ajustado los ficheros de entrada para adecuar el tiempo de ejecución total de las pruebas. La ejecución de las aplicaciones de E/S se ha configurado de manera que leyeran y escribieran en disco uno o varios ficheros. En todas las ejecuciones de una misma aplicación se utiliza la misma carga de trabajo.

2. Para cada aplicación utilizada en el benchmark y para cada protector, empacar la aplicación con el máximo nivel de protección que ofrece la herramienta. Si el ejecutable generado no funciona correctamente, probar las protecciones una a una para ver cuál es la que puede estar causando problemas.

3. Una vez obtenido un ejecutable correcto (no siempre se ha obtenido, véase la Sección 6.2.1) incluirlo para su ejecución en el benchmark. Volver al paso 2 hasta haber protegido las 12 aplicaciones con los 21 protectores.

4. Ejecutar el benchmark y procesar los resultados.

Para aumentar la fiabiliad de los resultados, cada ejecución se ha llevado a cabo un total de **45 veces**. Además, al igual que en SPEC las aplicaciones se ejecutan de manera intercalada. El tiempo de ejecución total del benchmark es de aproximadamente **90 horas**.

Capítulo 6

Evaluación Cualitativa y Cuantitativa de Protectores Software

En este capítulo se realiza la evaluación cualitativa y cuantitativa de los protectores seleccionados. Para la evaluación cualitativa se ha hecho un análisis de las protecciones implementadas por cada protector y de la dificultad de eliminarlas. La evaluación cuantitativa ofrece un análisis de la fiabilidad y el rendimiento de los protectores, basándose en las pruebas realizadas con los protectores y en los resultados del benchmark.

6.1. Evaluación Cualitativa

En esta sección se estudia, para cada técnica de protección propuesta en el Capítulo 4, las técnicas de desprotección existentes y cuáles de los protectores analizados la implementan.

La Tabla 6.1 muestra una comparativa general de los protectores analizados en función de las protecciones que implementan. Se puede observar que la mayoría de ellos ofrecen la posibilidad de comprimir el ejecutable original, aunque no siempre se obtiene un ejecutable más pequeño de salida ya que las protecciones adicionales aumentan en muchas ocasiones el tamaño final. Existen 3 protectores (UPX, FSG y MEW) que solamente ofrecen funcionalidades de compresión del ejecutable, en estos casos el tamaño del ejecutable final es más pequeño que el original, sin embargo, la protección que tiene es mínima.

Algunos protectores incluyen también técnicas anti-desensamblado pero éstas no ofrecen ninguna protección más allá de complicar el análisis estático del código.

La mayoría de los protectores seleccionados implementan una o varias protecciones anti-debugging. Sin embargo, éstas son relativamente fáciles de vencer tanto de manera manual como automática. Por ejemplo, ayudándose de diferentes plug-ins para el OllyDbg.

Las protecciones anti-dumping son algo más complejas, aunque la mayoría de ellas son ampliamente conocidas y existen multitud de artículos, tutoriales, scripts y programas en la red que pueden usarse para vencerlas. Las técnicas anti-patching son también implementadas por muchos protectores, pero conseguir eliminarlas es relativamente sencillo.

	Compresión	Cifrado	Anti Desensamblado	Anti-Debugging	Anti-Dumping	Anti-VM	Anti-Patching	Virtualizador
ACProtect	✓	–	✓	✓	✓	–	–	–
Armadillo	✓	✓	✓	✓	✓	–	✓	–
ASPack	✓	✓	–	✓	–	–	–	–
ASProtect	✓	✓	✓	✓	✓	–	✓	–
Enigma	✓	✓	✓	✓	✓	✓	✓	✓
EXE Stealth	✓	–	–	✓	✓	✓	✓	–
ExeCryptor	✓	–	✓	✓	✓	✓	✓	✓
ExPressor	✓	–	–	✓	✓	✓	✓	–
FSG	✓	–	–	–	–	–	–	–
MEW	✓	✓	–	–	–	–	–	–
Obsidium	✓	✓	✓	✓	✓	✓	✓	✓
PECompact	✓	–	–	✓	✓	–	✓	–
PELock	✓	–	–	✓	✓	–	✓	–
PESpin	✓	✓	✓	✓	✓	–	✓	–
Petite	✓	–	–	✓	✓	–	–	–
Smart Packer	✓	✓	–	–	–	–	–	–
TeLock	✓	–	–	✓	✓	–	–	–
Themida	✓	✓	✓	✓	✓	✓	✓	✓
UPX	✓	–	–	–	–	–	–	–
VMProtect	✓	–	✓	✓	✓	✓	✓	✓
Yodas Protector	✓	–	✓	✓	✓	–	–	–

Tabla 6.1: Comparativa general de los protectores analizados.

Como se observa en la Tabla 6.1, de los 21 protectores analizados, 5 son del tipo virtualizador (EXECryptor, Enigma, Obsidium, Themida y VMProtect). Estos protectores son los más potentes que existen ya que esta es una técnica mucho más difícil de vencer que las anteriores y asegura un mayor nivel de protección dado que el código original no está presente en memoria en ningún momento de la ejecución. Sin embargo, tampoco garantiza que nunca vaya a ser vencida ya que se pueden crear herramientas desensambladoras que hagan la traducción inversa del código de la máquina virtual a código ensamblador.

En algunos casos (EXEStealth, EXPressor, Enigma, EXECryptor, Obsidium, Themida y VMProtect) también se implementan técnicas para detectar determinadas máquinas virtuales. Este tipo de técnicas son muy comunes también en los programas malware ya que así evitan ser analizados al no poder ser ejecutados en un entorno controlado [RG14].

A continuación se describe en detalle para cada técnica de protección descrita en el Capítulo 4 los diferentes packers que la implementan y se describen algunas de las técnicas de desprotección posibles. En el Apéndice C se detallan las características de cada uno de los protectores, así como las principales protecciones que implementan.

6.1.1. Protección Copia

Muchos de los packers analizados implementan un sistema de gestión de licencias que permite controlar el tiempo durante el cual la aplicación está disponible, el número de ejecuciones o las funcionalidades que estarán disponibles así como permitir la generación de claves de registro: ACProtect, Armadillo, Enigma, EXEStealth, EXECryptor, EXPressor, Obsidium, VMProtect, Yodas Protector.

Algunos permiten también relacionar el programa con un determinado hardware de manera que sólo puedan ser ejecutados en un determinado PC (EXEStealth, EXECryptor, Obsidium).

Hoy en día este tipo de protecciones no resultan demasiado efectivas ya que si alguien logra desproteger el programa puede difundir una copia rápidamente por la red.

6.1.2. Compresión

La mayoría de los protectores estudiados tienen la opción de comprimir el ejecutable. La mayoría también permite elegir el nivel de compresión deseado (a menor compresión, mayor rapidez y viceversa). Algunos tienen además la opción de comprimir recursos (ASPack, ASProtect, EXECryptor, EXPressor, PECompact, PESpin, UPX, MEW, FSG, PELock, Themida, TELock, VMProtect entre otros) o quitar la sección *.reloc* (EXECryptor, MEW, PECompact, PELock, PESpin, Petite , VMProtect, Yodas Protector entre otros).

La compresión de un ejecutable impide únicamente el análisis estático del código, ya que una vez termina la rutina de descompresión el código se encuentra disponible en memoria sin ningún tipo de protección adicional.

6.1.3. Cifrado

Algunos packers que la implementan son: Armadillo, ASPack, ASProtect, Enigma, MEW, Obsidium, PESpin, SmartPacker, Themida.

Una vez que la rutina de descifrado termina, el código ya está disponible en memoria sin ninguna protección adicional. Si la rutina de descifrado depende del número de registro, hay que obtenerlo primero para conseguir descifrar el programa.

6.1.4. Anti Desensamblado

Algunos protectores insertan código *dummy* en el código original (ASProtect, Enigma, Themida). Otros implementan la técnica de código metamórfico (ACProtect, EXECryptor, VMProtect y Themida). Esta técnica reescribe el código original con otras instrucciones, de manera que el código generado es diferente en cada ejecución. Otra técnica es la protección de las cadenas utilizadas en el programa (Obisidium, Enigma, VMProtect, Yodas Protector, Themida). El PESpin introduce saltos falsos en el código original y también implementa ofuscación del código como Obsidium.

Las técnicas anti-desensamblado consiguen hacer más difícil el análisis del programa complicando el código original para que sea más difícil tracearlo.

6.1.5. Anti Debugging

- **Basado en APIs**

  - *isDebuggerPresent*.

    Algunos packers que la implementan son: ASPack, ACProtect, ASProtect, Enigma, EXEStealth, EXECryptor, Obsidium, PECompact, PELock, Themida, TELock, VMProtect, Yodas Protector.

    Esta protección es muy sencilla de desproteger ya que hay diversos plug-ins de OllyDbg (p. ej. HideDebugger) que permiten hacerlo automáticamente, también se puede hacer de manera manual poniendo el flag PEB.BeingDebugged a cero.

  - *CheckRemoteDebuggerPresent*.

    Algunos packers que la implementan son: EXECryptor, Obsidium.

    Para evitar esta detección bastaría con modificar el valor devuelto por esta función.

  - *FindWindow*.

    Packer que la implementa: Obsidium.

    Se puede evitar modificando el valor devuelto por la API.

  - *CreateTool32Snapshot*.

    Algunos packers que la implementan son: ACProtect, Armadillo, Enigma, Obsidium, PELock, YodasProtector.

    Se puede evitar utilizando un plug-in de OllyDbg o de manera manual, parcheando la instrucción en la que se hace la comparación con el identificador de cada proceso para que no se ejecute.

  - *OutputDebugString*.

    Packer que la implementa: EXEStealth.

Se evita modificando el valor devuelto por la función en el caso de que haya debugger.

- **Proceso e Hilos**

  - NtGlobalFlag, ProcessHeapFlag.

    Algunos packers que la implementan son: EXEStealth, EXECryptor.

    La técnica de desprotección manual consiste en colocar los flags a 0, también se puede evitar de manera automática mediante un plug-in del OllyDbg (p.ej. HideOd).

  - Self Debugging.

    Algunos packers que la implementan son: Armadillo, EXPressor, PESpin.

    Para evitar que funcione la técnica *Self Debugging* puede utilizarse la la API CreateMutexA para no crear otro proceso si ya existe uno, se puede modificar el resultado para que el programa piense que ya hay un proceso debuggeando y así no se pase la ejecución al otro proceso.

  - Multihilo.

    Algunos packers que la implementan son: EXECryptor, Themida.

    En las protecciones multihilo, como existen diferentes hilos de ejecución, una posible técnica de desprotección consiste en modificar la API *CreateThread* para que los hilos no sean creados. Otra posibilidad es modificar esta misma API para que sean creados pero en modo suspendido, aunque la ejecución podría no funcionar.

- **Hardware y registros**.

  Algunos packers que la implementan son: ASProtect, Obsidium, PELock.

  Para evitar esta protección se puede utilizar el plug-in OllyDbg: Anti-Hardware Breakpoint, o utilizar en su lugar software breakpoints o memory breakpoints (en OllyDbg).

- **Tiempo y Latencia** (GetTickCount, timeGetTime).

  Algunos packers que la implementan son: Armadillo (GetTickCount), EXEStealth (rdstc).

  Es una protección sencilla, la técnica para evitarla consiste en escribir un 0 en el lugar donde guarda la comparación entre las dos llamadas a la API.

- **Basadas en Excepciones**

  Algunos packers que la implementan son: Armadillo, PECompact, ASPack, ASProtect, Obsidium, PELock, PESpin.

  Para evitar esta protección se pueden pasar automáticamente las excepciones al programa, por ejemplo, utilizando la funcionalidad disponible en el OllyDbg.

6.1.6. Anti Dumping

- **Ofuscación de la IAT.**

  Algunos packers que la implementan son: ACProtect, ASProtect, Enigma (APIs emuladas), EXEStealth, ExPressor (con APIs emuladas), Obsidium (APIs emuladas),PELock , PESpin (APIs emuladas), Petite, Themida, TELock, VMProtect, Yodas Protector.

  Para arreglar las entradas erróneas de la IAT existen varias posibilidades: si son pocas se puede solucionar a mano, buscando cuál es la API correspondiente a cada entrada defectuosa, si no se pueden utilizar herramientas como el Import Reconstructor que también tiene plug-ins para packers específicos. Esta técnica automática tampoco asegura que se arreglen el $100\,\%$, si no que puede quedar alguna que haya que arreglar a mano. Otro método es encontrar el punto donde se guardan los valores erróneos en la IAT y hacer que no se ejecute para que no se sobreescriban los correctos.

- **Nanomites**.

  Algunos packers que la implementan son: Armadillo, PESpin.

  Esta técnica fue introducida por el Armadillo [Fer08]. La herramienta *ArmInLine* permite restaurar los Nanomites en el Armadillo de manera automática, de igual manera existe el programa *SpiNano* para el PESpin.

- **Robo de bytes**.

  Algunos packers que la implementan son: ACProtect (*Code Replace*), ASProtect, Enigma, Obsidium, PELock, PESPin, Armadillo (*Code Splicing*), Themida.

  Esta técnica fue introducida por el ASProtect [Fer08]. Para conseguir restaurar el programa original hay que identificar cuales son los bytes robados y continuar la ejecución hasta llegar al falso OEP. Los bytes robados hay que añadirlos en una nueva sección al final del ejecutable. Al añadir una nueva sección es necesario ajustar el tamaño de la imagen en la cabecera (SizeOfImage). Por último, hay que modificar el OEP del programa para que inicie la ejecución desde estos bytes robados que hemos añadido y al final incluir la llamada al falso OEP para que continúe la ejecución. En el caso del Armadillo y el Themida, el robo de bytes no se produce en el OEP sino en posiciones aleatorias de memoria.

- **Tampering**.

  Packer que la implementa: Armadillo.

  La cabecera modificada por el Armadillo se puede reconstruir con el programa LordPE.

- **Páginas Guarda**.

  Algunos packers que la implementan son: EXEStealth, Armadillo.

  El Armadillo en su funcionalidad *Copy-Mem II* implementa una variación de este método que utiliza un proceso hijo, de tal manera que el proceso hijo está vacío al inicio y el padre es el encargado de copiar las páginas descifradas y controlar la ejecución del proceso hijo. Una vez se ha ejecutado las borrará nuevamente del proceso hijo. Una

técnica para evitar esta protección consiste en engañar al proceso padre para que descifre todo el código al inicio, modificando el programa manualmente para simular las excepciones. Este es un método de protección muy potente pero puede ralentizar mucho la ejecución del programa. Además, también puede ser incompatible con algunos programas.

- **Virtualización.**

  Algunos packers que la implementan son: EXECryptor, Enigma, Themida, Obsidium, VMProtect.

  Estas técnicas son de las más potentes que existen en los protectores de software actuales. Parte del código original se reescribe de tal manera que sólo es compatible con la máquina virtual embebida, con lo cual hay que entender en primer lugar la lógica de esta traducción, para después implementar la conversión inversa desde el *bytecode* de la máquina virtual a lenguaje ensamblador.

  Dentro de los protectores analizados existen varios que implementan diversas técnicas de virtualización:

  - EXECryptor implementa en su técnica de *Code Morphing* una transformación de comandos en instrucciones de máquina virtual.
  - Enigma permite seleccionar qué funciones del programa quieren virtualizarse, así como seleccionar la virtualización del *Entry Point*.
  - Themida permite seleccionar las funciones a virtualizar así como la virtualización del *Entry Point*. Se puede especificar el procesador en el que se quiere virtualizar (CISC, RISC). Ambas tienen la misma lógica pero la RISC es más compleja. Las instrucciones generadas pueden ser metamórficas, de tal manera que dos empaquetados de un mismo programa resultan en un código diferente. También permite ofuscar el código de la máquina virtual para dificultar su análisis, introduciendo múltiples saltos condicionales.
  - VMProtect también genera *bytecodes* metamórficos.
  - En Obsidium se reescriben ciertas partes del código para que sólo sean entendibles por la máquina virtual embebida.

  Tanto para el Themida como para el EXECryptor existen diversas herramientas y scripts que traducen automáticamente los *bytecodes* a ensamblador.

6.1.7. Anti Virtualización

Algunos packers que la implementan son: EXEStealth, EXPressor, Enigma (VMWare, VirtualPC y Sandboxie) , EXECryptor (VMWare y Virtual PC), Obsidium,Themida (VMWare, VirtualPC), VMProtect (VMWare, VirtualPC, VirtualBox, Sandboxie).

6.1.8. Anti Patching

Algunos packers que la implementan son: Armadillo, ASProtect, Enigma, EXEStealth, EXECryptor, EXPressor, Obsidium, PECompact, PELock, PESpin, Themida, VMProtect.

Estas protecciones se evitan buscando el lugar donde se realizan la comprobación y evitar la llamada a esa función o modificando el valor devuelto por la función de comprobación. Los packers Obsidium, Enigma comprueban en tiempo de ejecución cada cierto tiempo si ha habido modificaciones, en estos casos hay algo más de complejidad ya que es necesario evitar que se creen otro hilos de ejecución.

6.2. Evaluación Cuantitativa

A continuación se realiza un análisis cuantitativo de las 21 herramientas de software seleccionadas para el estudio. Este análisis incluye el estudio de la fiabilidad y del rendimiento a nivel de tiempo de ejecución y consumo de memoria.

6.2.1. Análisis Fiabilidad

En esta sección se hace un análisis de la fiabilidad de las herramientas protectoras estudiadas. Por fiabilidad se entiende la ratio de ejecutables correctos generados por una determinada herramienta protectora. Para sacar estos resultados, primero se ha intentado proteger la aplicación con todas las protecciones que ofrece la herramienta activadas. En caso de no funcionar, se han ido probando las protecciones una a una para ver cuál era la protección que corrompía el ejecutable final. En algunos casos, incluso quitando todas las protecciones, el ejecutable generado no ha funcionado correctamente.

La Tabla 6.2 muestra los resultados observados en términos de fiabilidad, tanto por tipo de aplicación como la ratio total. Obsidium y EXEStealth tienen una ratio del $100\,\%$: en estos casos se han considerado los ejecutables empacados por los propios fabricantes que se prestaron a colaborar en el proyecto.

En algunos casos el ejecutable empacado con FSG, MEW, PESPin, Petite, TELock y Yodas Protector daba un error específico: *"Runtime error: R6002 floating point support not loaded"*. Se encontró la descripción de este problema en varios foros [KPN13, MrG13, CGS13, bla13]. Es un fallo que tienen determinados protectores y ocurre con ejecutables que han sido compilados con MSVC++5 o superior.

Hay que mencionar también, que con algunas aplicaciones, para poder generar un ejecutable correcto no se han podido utilizar todas las protecciones disponibles.

- Con ACProtect se han encontrado problemas con la opción de *CodeReplace* y *API Redirection*.
- Con Armadillo en algunos casos ha funcionado el ejecutable final pero sin utilizar la protección más fuerte denominada *Copy-MemII*.
- PELock ha dado problemas en algunos casos son su funcionalidad *API Redirection*.

- Con PESPin sólo se ha podido generar un ejecutable correcto en algunos casos sin utilizar las protecciones *Remove OEP* ni *Debug Blocker*.

También hay que tener en cuenta a la hora de valorar estos resultados que en el caso de VMProtect, Themida, EXECryptor, Armadillo y PELock sólo se han podido utilizar las protecciones disponibles en la versión de prueba.

	Cálculo entero	Cálculo real	Gran demanda E/S	Total
ACProtect	25 %	50 %	75 %	50 %
Armadillo	50 %	100 %	75 %	75 %
ASPack	100 %	100 %	75 %	91 %
ASProtect	100 %	100 %	75 %	91 %
Enigma	75 %	75 %	100 %	83 %
EXE Stealth	100 %	75 %	100 %	91 %
ExeCryptor	25 %	50 %	0 %	25 %
ExPressor	100 %	100 %	100 %	100 %
FSG	0 %	50 %	100 %	50 %
MEW	100 %	50 %	100 %	50 %
Obsidium	100 %	100 %	100 %	100 %
PECompact	25 %	100 %	100 %	75 %
PELock	25 %	100 %	75 %	66 %
PESpin	0 %	50 %	75 %	41 %
Petite	50 %	25 %	50 %	41 %
Smart Packer	50 %	75 %	100 %	75 %
TeLock	25 %	50 %	50 %	41 %
Themida	25 %	100 %	100 %	75 %
UPX	100 %	100 %	100 %	100 %
VMProtect	100 %	100 %	100 %	100 %
Yodas Protector	25 %	25 %	50 %	33 %

Tabla 6.2: Fiabilidad de los protectores analizados.

6.2.2. Análisis Rendimiento

Para analizar el rendimiento de los protectores se han utilizado las mediciones realizadas en el benchmark. En él se ha ejecutado tanto la aplicación original como la aplicación protegida con cada herramienta. Con estos datos se puede calcular el tiempo de más ($overhead_{cpu}$) y la memoria de más consumida ($overhead_{mem}$) al ejecutar las aplicaciones protegidas con cada uno de los protectores.

En el benchmark se han realizado mediciones de 17 de las 21 herramienas seleccionadas. Las 4 que se han quedado fuera han sido Themida, Armadillo, EXE Stealth, Smart Packer. Los motivos de exclusión han sido:

- **Ventana Aviso de licencia** (Themida y Armadillo). Como se ha comentado en la sección 5.1, en algunos casos, a pesar de haber solicitado al fabricante una versión completa de la herramienta con fines académicos, no ha sido posible obtenerla. En estos casos, no se ha podido utilizar todas las protecciones que la herramienta ofrece. Por otro lado, la ventana de aviso de licencia no permitía ejecutar el benchmark completo de manera automática. Esto se solventó en algunos casos (EXECryptor, VMProtect y ACProtect) utilizando un script de AutoHotKeys [Hal13], el cual debía estar ejecutándose a la vez que el benchmark y se encargaba de cerrar auomáticamente las ventanas de diálogo emergentes. En otros casos (Themida y Armadillo) no ha servido esta solución ya que incluyen una *splash screen*. Como el script funciona buscando por el nombre de la ventana y en estos casos no tiene nombre, no se pudieron incluir en el benchmark.

- **Proceso adicional** (EXEStealth, SmartPacker). Algunas herramientas implementan protecciones que crean un proceso hijo que es el que ejecuta el programa original, por ejemplo ExPressor, PESpin, SmartPacker y EXEStealth. En el caso de ExPressor y PESpin se ha podido adaptar el benchmark para que haga las mediciones del nuevo proceso creado. EXEStealth y SmartPacker son más complejos ya que se crean y destruyen varios procesos adicionales a lo largo de la ejecución.

La Figura 6.1 muestra el $overhead_{cpu}$ medio por protector. El $overhead_{cpu}$ calculado es el % de más que tardan en ejecutarse las aplicaciones protegidas con cada protector respecto a la ejecución de las aplicaciones originales.

Se observa que algunos protectores no provocan $overhead_{cpu}$ en las aplicaciones del benchmark. Esto es debido en algunos casos (FSG, MEW, UPX) a que se trata de herramientas únicamente compresoras, con lo cual la diferencia entre la ejecución original y la protegida es mínima al ser necesario solamente la ejecución de la rutina descompresora.

Como se ha comentado en la Sección 6.2.1, hay que tener en cuenta que no todos los ejecutables han sido generados con todas las protecciones disponibles en la herramienta.

Los mayores porcentajes de $overhead_{cpu}$ se observan en los protectores ASProtect, EXPressor, Obsidium y PELock. Todos ellos están entre un $6\,\%$ y un $8\,\%$.

La Figura 6.2 muestra el $overhead_{mem}$ medio por protector. El $overhead_{mem}$ calculado es el % de memoria consumida de más al ejecutar las aplicaciones protegidas con cada protector respecto a la ejecución de las aplicaciones originales.

En ella se oberva que todas las herramientas protectoras provocan un mayor consumo de memoria que la ejecución de la aplicación original. Las herramientas compresoras (FSG, MEW y UPX) son también las que tienen un menor overhead de memoria. El ASPack es compresor y tiene alguna protección pero mínima, con lo cual el overhead de memoria y de cpu que provoca son también pequeños y están en torno al $2\,\%$. Por otro lado, el Yodas es la que mayor $overhead_{mem}$ tiene con un $20\,\%$, a pesar de que en tiempo de ejecución costaba lo mismo que la ejecución original. En el caso del Enigma, el $overhead_{mem}$ del $17\,\%$ se puede explicar ya que se ha utilizado la funcionalidad de virtualizar algunas funciones de la aplicación. Esto demuestra que a pesar de ser una protección muy potente, la virtualización de la aplicación también provoca una mayor ralentización de la ejecución, así como un mayor consumo de memoria.

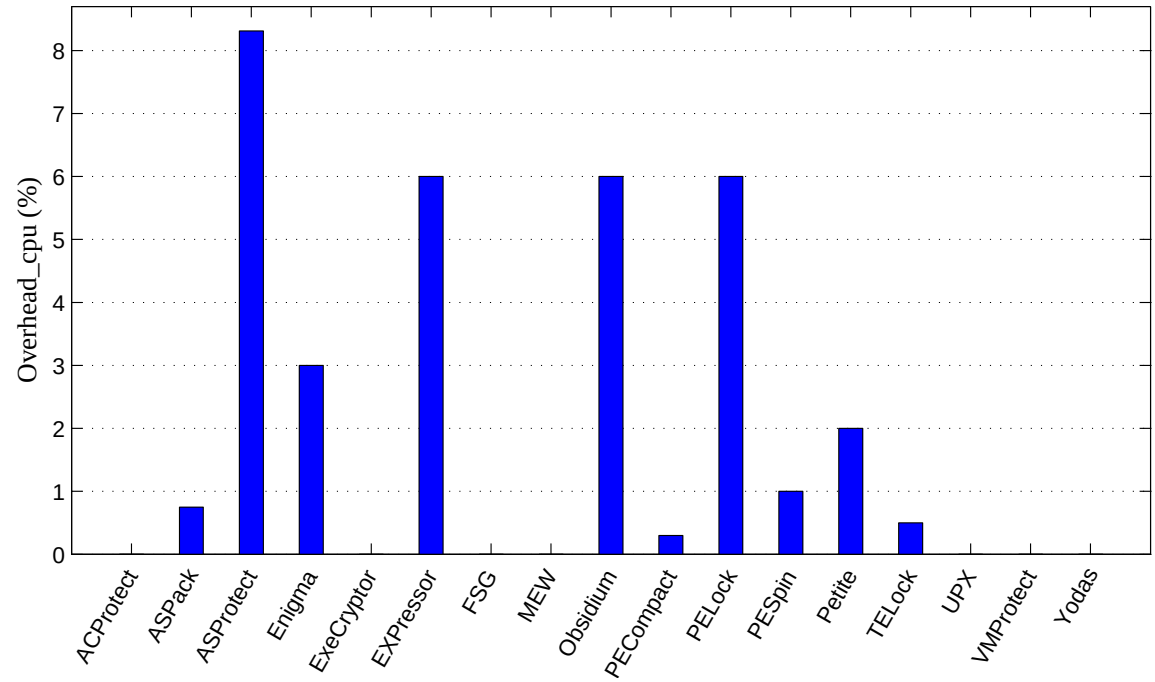

Figura 6.1: Overhead de cpu medio de los protectores.

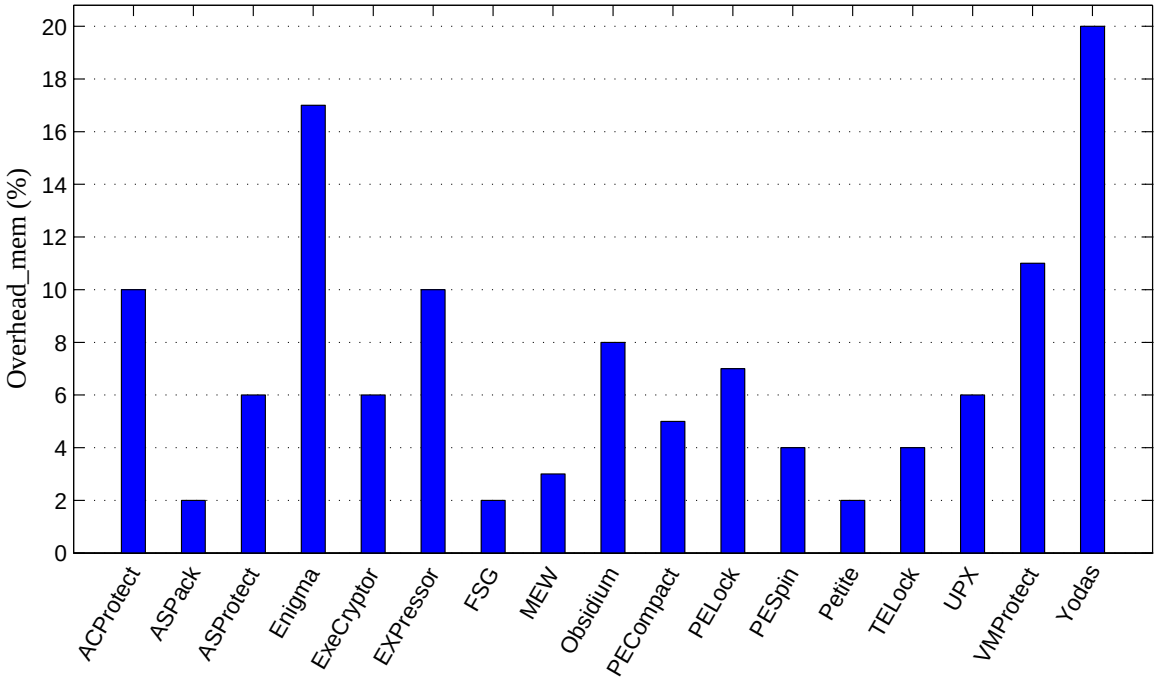

Figura 6.2: Overhead de memoria medio de los protectores.

Capítulo 7

Conclusiones y Trabajo Futuro

Este capítulo presenta algunas conclusiones obtenidas de la elaboración de este PFC. Además, plantea el trabajo futuro.

Las herramientas protectoras de software implementan diferentes técnicas de protección con el objetivo de evitar que un programa sea víctima de un ataque de ingeniería inversa. Estas mismas herramientas y técnicas son utilizadas por creadores de malware para evitar que sus programas sean detectados por los anti-virus.

En este proyecto se ha hecho un análisis cualitativo y cuantitativo de un total de 21 herramientas protectoras, tanto gratuitas como comerciales.

En primer lugar, se han analizado las protecciones implementadas por cada herramienta. En función de las posibilidades que ofrece y de la configuración elegida a la hora de proteger un programa, se puede obtener un mayor o menor nivel de protección. Sin embargo, casi ninguna de ellas puede evitar que el código esté accesible en memoria en algún momento de la ejecución. También se han descrito posibles técnicas de desprotección para las protecciones. La mayoría de ellas son ampliamente conocidas y han sido descritas y difundidas por la red. Esto provoca que, en muchos casos, aunque un programa haya sido protegido antes de su distribución no deja de ser vulnerable a ataques de ingeniería inversa.

Después, se ha hecho un estudio de la fiabilidad y el rendimiento de las herramientas. Para ello, se ha intentado proteger un total de 12 aplicaciones con todas ellas detectando algunos problemas con determinadas protecciones. Así pues, a la hora de proteger un ejecutable no siempre se puede hacer uso de todas las funcionalidades de las que dispone la herramienta protectora. Por otro lado, se ha comprobado que el hecho de proteger una aplicación provoca una sobrecarga en la memoria consumida y además, en algunos casos, una sobrecarga en el tiempo de ejecución. Una de las protecciones más fuertes de las estudiadas es la virtualización de determinadas partes del código ya que evita que el código virtualizado esté visible en memoria. Sin embargo, esta técnica debe usarse con cuidado ya que tiene un impacto directo en el rendimiento.

Las herramientas protectoras de software, necesitan evolucionar muy rápidamente ya que con el paso del tiempo su efectividad disminuye drámaticamente. Protecciones que hace

unos años eran consideras complejas, hoy en día son muy sencillas de evitar, ya que las técnicas utilizadas para vencerlas se han difundido por la red o incluso se han creado scripts o herramientas automáticas para realizar complejas restauraciones del ejecutable en poco tiempo e incluso por alguien inexperimentado.

Algunas líneas de trabajo futuro que pueden completar o ampliar este trabajo son:

- **Utilizar versiones completas de los protectores**. Como se ha comentado anteriormente, con 5 de los protectores analizados sólo se ha podido trabajar con una versión demo (Armadillo, EXECryptor, Themida, PELock y VMProtect). A pesar de haber solicitado al fabricante una versión completa, no ha sido posible obtenerla. Por ello, una posible línea de trabajo futuro sería realizar el análisis utilizando versiones completas de todos los productos.
- **Ampliar el número de protectores**. Se podrían aumentar el número de herramientas protectoras analizadas, sobre todo aquellas que implementen alguna técnica de protección que no se haya visto en este estudio.
- **Ampliar el número de aplicaciones del benchmark**. Aumentar el número de aplicaciones del benchmark para evaluar el rendimiento y la fiabilidad de los protectores.

Bibliografía

[BBN12] Rodrigo Rubira Branco, Gabriel Negreira Barbosa, and Pedro Drimel Neto. Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and AntiVM Technologies. In *BlackHat USA'12*, 2012.

[bla13] blackd0t. Pe packer causing runtime msvc++ runtime r6002 error. https://www.openrce.org/forums/posts/1002, Dic 2013. Online Accessed.

[CC90] Elliot J. Chikofsky and James H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, Volume 7, Issue 1:Pages 13–17, January 1990.

[CGS13] CGSoftLabs. About runtime error! r6002 floating point support not loaded. http://www.makephpbb.com/phpbb/viewtopic.php?mforum=cgsoftlabs&p=148, Dic 2013. Online accessed.

[Cor99] Microsoft Corporation. Microsoft portable executable and common object file format specification. http://msdn.microsoft.com/enus/windows/hardware/gg463119, 1999.

[DP10] S. Debray and J. Patel. Reverse Engineering Self-Modifying Code: Unpacker Extraction. pages 131–140, Oct 2010.

[Eil05] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wilev publishing Canada, 2005.

[Fer06] Peter Ferrie. Attacks on More Virtual Machine Emulators. Symantec Security Response, Dec 2006.

[Fer08] Peter Ferrie. Anti Unpacker Tricks. http://pferrie.tripod.com/papers/unpackers.pdf, 2008.

[Fer11] Peter Ferrie. *The Ultimate Anti-Debugging Reference*, Apr 2011.

[GFC08] Fanglu Guo, Peter Ferrie, and Tzi-Cker Chiueh. A Study of the Packer Problem and Its Solutions. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, September 2008.

[Hal13] Halweg. Fensterpflege. http://www.autohotkey.com/board/topic/6812-close-annoying-dialog-boxes-automatically/, Dic 2013. Online accessed.

[JCL+10] Guhyeon Jeong, Euijin Choo, Joosuk Lee, Munkhbayar Bat-Erdene, and Heejo Lee. Generic Unpacking using Entropy Analysis. In *IEEE: 5th International Conference on Malicious and Unwanted Software*, 2010.

[KLC+10] Min-Jae Kim, Jin-Young Lee, Hye-Young Chang, SeongJe Cho, Minkyu Park, Yongsu Park, and Philip A. Wilsey. Design and Performance Evaluation of Binary Code Packing for Protecting Embedded Software against Reverse Engineering. *IEEE International Svmposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages Pages 80–86, May 2010.

[KPN13] KPNC. Msvc challenge for pe packers. http://nezumi-lab.org/blog/?p=73#comments, Dic 2013. Online Accessed.

[KPY07] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A Hidden Code Extractor for Packed Executables. Technical report, Carnegie Mellon University, 2007.

[LTC13] Kaiping Liu, Hee Beng Kuan Tan, and Xu Chen. Binary Code Analysis. *Computer*, vol. 46(8):60–68, Aug 2013.

[MCJ07] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic, and Safe Unpacking of Malware. In *In Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07). IEEE Computer Society*, pages 431–441, 2007.

[MrG13] MrGneissGuy's. Runtime error r6002 - floating point not loaded. http://forum.exetools.com/printthread.php?t=12459, Dic 2013. Online accessed.

[Pie94] Matt Pietrek. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. *Microsoft Systems*, Mar 1994.

[Pie02] Matt Pietrek. An In-Depth Look into the Win32 Portable Executable File Format. *MSDN Magazine*, Feb 2002.

[RG14] Ricardo J. Rodríguez and Iñaki Rodríguez Gastón. Hardering Sandbox Environments with Dynamic Binary Instrumentation Techniques. 2014.

[RHD+06] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack-Executing Malware. Technical report, College of Computing, Georgia Institute of Technology, 2006.

[Shi09] Tyler Shields. Anti-Debugging : A Developers View. Technical report, Veracode Inc., USA, 2009.

[SIJ+11] Donghwi Shin, Chaetae Im, Hyuncheol Jeong, Seungjoo Kim, and Dongho Won. The new signature generation method based on an unpacking algorithm and procedure for a packer detection. *International Journal of Advanced Science and Technology*, 27, 2011.

[Sof13] Ibsen Software. Aplib compression library. http://ibsensoftware.com/products_aPLib.html, Dic 2013. Online Accessed.

[STF09] M. Zubair Shafiq, S. Momina Tabish, and Muddassar Farooq. PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. Technical report, Next Generation Intelligent Networks Research Center (nexGIN RC) National University of Computer & Emerging Sciences (NUCES-FAST), 2009.

[STMF09] M.Z. Shafiq, S.M. Tabish, F. Mirza, and M. Farooq. PE-Miner: Realtime Mining of 'Structural Information' to Detect Zero-Day Malicious Portable Executables. In *Lecture Notes in Computer Science*, 2009.

[Sun10] Li Sun. *REFORM: A Framework for Malware Packer Analysis Using Information Theory and Statistical Methods*. PhD thesis, School of Mathematical and Geospatial Sciences,College of Science, Engineering and Health,RMIT University, 2010.

[Tah07] Gaith Taha. Counterattacking the packers. McAfee Avert Labs, 2007.

[VN06] Miroslav Vnuk and Pavol Navrat. Decompression of run-time compressed PE-files. 2006.

[Wik13a] Wikipedia. Executable compression. http://en.wikipedia.org/wiki/Executable_compression, Nov 2013. Online accessed.

[Wik13b] Wikipedia. Lempel-Ziv-Markov chain algorithm, Dic 2013. Online accessed.

[Wik13c] Wikipedia. Portable executable. http://en.wikipedia.org/wiki/Portable_Executable, Nov 2013. Online accessed.

[Yas07] Mark Vincent Yason. *The art of unpacking*. IBM Internet Security Systems, 2007.

[Yus13] Oleh Yuschuk. Olly Debugger. http://www.ollydbg.de/, Nov 2013. Online accessed.

Apéndice A

Fases de Desarrollo

A.1. Planificación temporal y esfuerzo

El primer paso para la realización de este proyecto fue leer información sobre lo que son las herramientas protectoras de software, así como el estudio de la estructura de un fichero ejecutable en Windows. Después se procedió a estudiar las diferentes técnicas de protección y desprotección de ejecutables en Windows.

A continuación se realizó una selección de herramientas protectoras para el estudio y se analizaron las protecciones implementadas por cada uno de ellos. Para ello se estudiaron diferentes tutoriales específicos para cada herramienta.

Después se procedió a crear un benchmark para poder realizar la evaluación de la fiabilidad y del rendimiento de los protectores seleccionados. Se buscaron las aplicaciones para incluir en él, se realizaron pruebas con cada una de ellas y las distintas herramientas y después se programó la herramienta que realizara las mediciones de tiempo de ejecución y consumo de memoria.

El proyecto se ha ido documentando poco a poco desde el principio, dedicando las últimas semanas a estructurar la memoria. Durante todo el desarrollo del proyecto se han mantenido reuniones semanales o bisemanales de seguimiento con el director de proyecto.

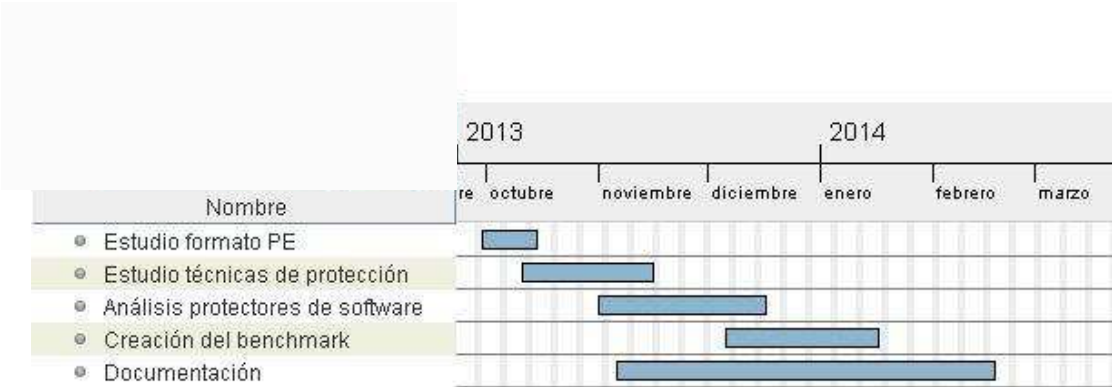

Figura A.1: Planificación Temporal.

La Figura A.2 muestra el porcentaje de horas dedicadas a cada tarea de proyecto y la Tabla A.1 muestra el detalle de horas dedicadas y las horas totales.

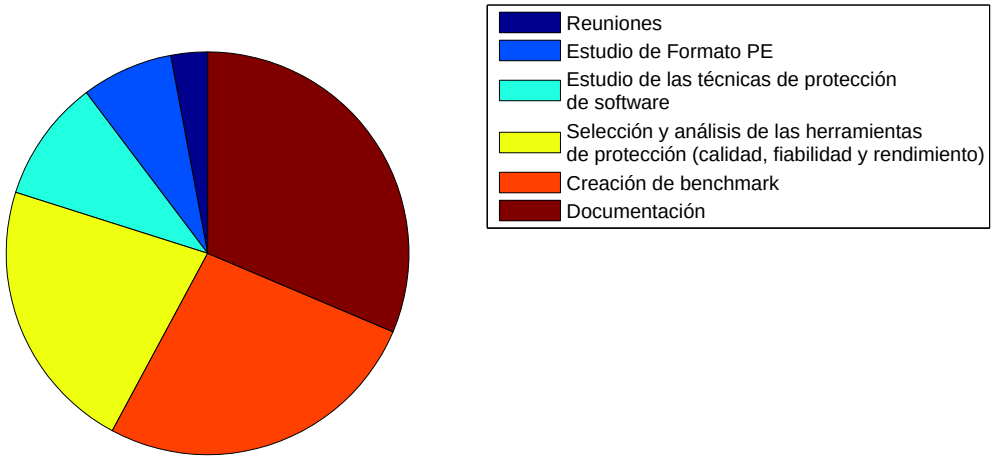

Figura A.2: Horas dedicadas.

Tareas	Horas
Reuniones	10
Estudio de Formato PE	32
Estudio de las técnicas de protección de software	45
Análisis de las herramientas de protección (calidad, fiabilidad y rendimiento)	98
Creación del benchmark	117
Documentación memoria	159
Total	**461**

Tabla A.1: Horas dedicadas.

Apéndice B

Aplicaciones Usadas en el Benchmark

En este apéndice se muestra todo el software que ha sido utilizado para la creación del benchmark, junto con detalles de cada uno así como los ficheros de entrada y de salida generados.

B.1. aescrypt

Versión: 3.0.9
Categoría: Criptografía, cifrado.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: *Advanced Encryption Standard* (AES) es un algoritmo de cifrado simétrico. Fue adoptado por el gobierno de los Estados Unidos de América en 2001, después de un proceso en el que se evaluaron diferentes algoritmos de cifrado que duró 5 años. Esta evaluación se hizo para sustituir al algoritmo *Data Encryption Standard* (DES).
Entrada: Se indica la contraseña para cifrar y el fichero a utilizar es `test800.mkv`, que tiene un tamaño de 823MB.
Salida: Se obtiene el fichero cifrado `test800.mkv.aes`.
Página web: `http://www.aescrypt.com/`
Usada en: Origen propio, no es usado en benchmarks conocidos.

B.2. bzip2

Versión: 1.0.5
Categoría: Compresión de datos
Tipo de cálculo realizado: Entero
Descripción: Es un compresor de datos de alta calidad, libre de patentes y libremente disponible. Típicamente comprime ficheros entre un $10\,\%$ y un $15\,\%$ más que con otros tipos de compresores, y es alrededor de dos veces más rápido en compresión y seis veces más rápido descomprimiendo.

Entrada: Se comprimirán de una única ejecución los siguientes ficheros: texto.doc, con texto en plano; test.mp4, fichero de vídeo y Crisantemo.jpg.
Salida: Se han comprimido los ficheros y los resultados se han guardado en disco.
Página web: bzip.org
Autor: Julian Seward
Usada en: SPEC CINT 2006

B.3. calculix

Versión: 2.6
Categoría: Mecánica estructural.
Tipo de cálculo realizado: Real
Lenguaje de programación: Fortan90 y C
Descripción: Es un software de elementos finitos para aplicaciones estructurales tridimensionales. Utiliza la teoría clásica de elementos finitos. Puede ser utilizado para resolver gran variedad de problemas, estáticos y dinámicos.
Entrada: Se resuelve la ecuación de elementos finitos a partir del fichero de entrada *axrad2.inp*.
Salida: Se genera un fichero de salida con los resultados *axrad2.frd*.
Página web: `http://www.calculix.de`
Autor: Guido D.C. Dhondt
Usada en: SPEC CFP 2006

B.4. GNU go

Versión: 3.8
Categoría: Inteligencia Artificial - Juegos
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Programa que analiza jugadas del juego Go.
Entrada: se utiliza la opción - benchmark 100 de tal manera que GNU Go juegue contra si mismo repetidamente (hasta 100 veces) empezando el juego con una serie de movimientos aleatorios.
Página web: `http://www.gnu.org/software/gnugo/gnugo.html`
Autor: Man Lung Li et Al.
Usada en: SPEC CINT 2006

B.5. ffmpeg

Versión: 0.10
Categoría: Conversión de formatos de video/audio.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Es un framework multimedia que permite codificar, decodificar, transcodificar, multiplexar, demultiplexar, filtrar y reproducir la mayoría de los formatos de audio y video. Genera una librería para poder ser utilizada por cualquier software para reproducción o conversión de formatos.
Entrada: Se usa el fichero `input.mkv` de formato YUV420, a una resolución de 1280x688, con el códec de audio mp3 a 48Khz.
Salida: Se genera el ficher `output.avi` con el formato MPEG-4, a una resolución de 640x480 con el codec de audio Dolby AC3 en stereo a 48Khz.
Página web: `http://ffmpeg.org`
Autor: Fabrice Bellard et al.
Usada en: Phoronix Test Suite (PTS)

B.6. h264ref

Versión: 18.5
Categoría: Compresión y conversión de video y audio.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Software de conversión de formatos de video y audio. Utiliza los *códecs* H.264/AVC.
Entrada: Se codifica el fichero `foreman_part_qcif_422.yuv`, una secuencia de imágenes en formato YUV420, a una resolución de 176x144.
Salida: Se genera el fichero `foreman_cif.264` en formato H.264.
Página web: `http://iphome.hhi.de/suehring/tml/`
Autor: Karsten Sühring et Al.
Usada en: SPEC CINT 2006

B.7. hmmer

Versión: 3.0
Categoría: Genética
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Busca una proteína en una base de datos. Usa el modelo de cadenas de Markov ocultas (HMMs) como modelos estadísticos, que es usado en biología computacional para

buscar patrones en secuencias de ADN.
Entrada: Se indica la proteína a buscar, goblins4 en el fichero `globins4.hmm` y la base de datos donde tiene que hacerlo `uniprot_sprot.fasta`
Salida: Genera cuatro ficheros donde se indican las coincidencias encontradas.
Página web: `http://hmmer.janelia.org/`
Autor: Sean Eddy et Al.
Usada en: SPEC CINT 2006

B.8. mcf

Versión: 1.3
Categoría: Combinatoria.
Tipo de cálculo realizado: Entero
Lenguaje de programación: C
Descripción: Es un programa diseñado para resolver problemas de planificación de vehículos en compañías de transporte públicas.
Entrada: Se utiliza el fichero de entrada *big8.net* que contiene los horarios de un número determinado de viajes con su horario de llega y de salida.
Salida: Genera un fichero de salida *mcf.out* que contiene la planificación óptima calculada por el programa.
Página web: `http://www.zib.de/loebel/`
Autor: Andreas Loebel
Usada en: SPEC CINT 2006

B.9. md5

Versión: 1.2
Categoría: Criptografía
Tipo de cálculo realizado: Entero
Lenguaje de programación: Borland Delphi 7
Descripción: Es una aplicación para generar y verificar md5 checksum.
Entrada: Como entrada se ha utilizado el fichero de la base de datos de la aplicación hmmer *uniprot_sprot.fasta*.
Salida: Se genera el checksum del fichero de entrada.
Página web: `http://www.md5summer.org/`
Autor: Luke Pascoe
Usada en: Origen proprio.

B.10. namd

Versión: 2.8
Categoría: Biología, simulación de moléculas
Tipo de cálculo realizado: Real
Lenguaje de programación: C++
Descripción: Es un software de dinámica molecular paralela diseñado para simulación de alto rendimiento de grandes sistemas biomoleculares.
Entrada: Se utiliza un fichero de prueba que se llama `apoa1.namd` preparado para un benchmark, ya que es la carga significativa para un único procesador en una gran simulación.
Salida: Genera 3 ficheros `apoa1.coor`, `apoa1.vel` y `apoa1.xsc` con los resultados de la simulación.
Página web: `http://www.ks.uiuc.edu/Research/namd/`
Autor: Jim Phillips et al.
Usada en: SPEC CFP 2006

B.11. palabos

Versión: 1.4
Categoría: Dinámica de fluidos.
Tipo de cálculo realizado: Real
Lenguaje de programación: C++
Descripción: Es un software para simulación computacional de dinámica de fluidos. Se basa en el método Lattice Boltzmann.
Entrada: Se utiliza una simulación denominada *cavity2d*.
Salida: Se genera como salida una serie de imágenes en determinados puntos de la simulación en formato .ppm.
Página web: `http://www.palabos.org`
Autor: Flowkit Ltd.& University of Geneva.
Usada en: Origen propio. En SPEC CFP2006 se utiliza otra implementación de mismo método (LB).

B.12. povray

Versión: 3.7
Categoría: Renderización
Tipo de cálculo realizado: Real
Lenguaje de programación: C++
Descripción: Es una herramienta que produce gráficos por ordenador de muy alta calidad. Utiliza el algoritmo de trazado de rayos para generar imágenes tridimensionales.
Entrada: Se utiliza la opción de renderizar uno de los ficheros de ejemplo que incluye povray:

`chess2.pov`.
Salida: Se obtiene el fichero `chess2.png` con la imagen.
Página web: `http://www.povray.org/`
Autor: David Buck et al.
Usada en: SPEC CFP 2006

Apéndice C

Protectores Considerados en el Estudio

C.1. ACProtect

En este trabajo se ha utilizado la versión demo 2.1.0, que es la última disponible y data del año 2008. Actualmente la página web ya no se encuentra activa.

La Tabla C.1 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Desensamblado	**Bajo**
Dummy code	
Anti Debugging	**Bajo**
isDebuggerPresent	
Process32Next	
Anti Dumping	**Medio**
Stolen Bytes (Code Replace)	
Ofuscación IAT (API Random Redirection)	

Tabla C.1: Protecciones en ACProtect 2.1.0.

C.2. Armadillo

Armadillo Software Passport (`http://www.siliconrealms.com/armadillo.php`) es una herramienta propietaria que permite proteger archivos .exe, .dll, .scr o .ocx. Existen versiones tanto para 32 como para 64 bits (SoftwarePassport 32-bit y SoftwarePassport 64-bit). En este trabajo se ha utilizado la versión demo 9.6.2 disponible en la web. Este es un protector bastante potente que ofrece 4 niveles de protección, Protección Mínima, Protección Standard, Protección Standard + Debug Blocker y Copy-Mem II + Debug Blocker. Las dos primeras ofre-

cen una protección muy débil y sólo es recomendable su uso cuando las otras no funcionan. La protección Debug Blocker evita el uso de debuggers mediante la técnica de self-debugging y Copy-Mem II es una protección anti-dumping que descifra las páginas dinámicamente durante la ejecución del programa. Este es un método de protección muy potente pero puede ralentizar mucho la ejecución del programa, también puede ser incompatible con algunos programas. También ofrece varias protecciones anti dumping: Nanomites, eliminación de la Import Table (IT), técnicas anti-análisis como permutación del código. Implementa también una protección anti-patching, para evitar que el código pueda ser modificado en memoria (existe un bug en esta protección y es que si se trata de un programa auto-modificabable entiende esto como un ataque, aunque la modificación venga del mismo programa).

La Tabla C.2 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Medio**
Anti Debugging	**Medio**
isDebuggerPresent	
selfDebugging	
anti SoftICE	
GetTickCount	
Anti Dumping	**Alto**
Nanomites	
Ofuscación IAT	
Stolen Bytes (Code Splicing)	
Páginas Guarda	
Borrar Cabecera PE	
Anti Patching	**Medio**

Tabla C.2: Protecciones en Armadillo 9.62.

C.3. ASPack

ASPack (`http://www.aspack.com/`) es una herramienta comercial desarrollada para comprimir el ejecutable y apenas tiene capacidad de protección del programa. Permite comprimir ejecutables de 32bits y es compatible con las versiones de Windows XP/7/8 y Windows Server 2003/2008/2012. Se ha utilizado la versión demo 2.32 disponible en la web del fabricante.

La Tabla C.3 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Bajo**
Anti Debugging	**Bajo**
isDebuggerPresent	
Debugger interruptions (INT3)	

Tabla C.3: Protecciones en ASPack 2.32.

C.4. ASProtect

El ASProtect es un protector más potente que el anterior desarrollado por las misma compañía (`http://www.aspack.com/`). Permite comprimir ejecutables de 32bits y es compatible con las versiones de Windows XP/7/8 y Windows Server 2003/2008/2012. También existe una versión para 64bits. Para este trabajo se ha utilizado la versión demo 1.69 disponible en la web del fabricante.

La Tabla C.4 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Bajo**
Anti Debugging	**Medio**
isDebuggerPresent	
Debugger interruptions (INT3)	
Anti Dumping	**Medio**
Ofuscación IAT (Advanced Import Protection)	
Ofuscación IAT (Emulate Standard System Functions)	
Stolen Bytes	
Anti Patching	**Bajo**
Checksum Protection	

Tabla C.4: Protecciones en ASProtect 1.69.

C.5. Enigma

Es una herramienta comercial (`http://www.enigmaprotector.com/en/about.html`), que permite proteger ejecutables de 32 y de 64bits. Permite proteger archivos .exe, .dll, .ocx y .scr entre otros, y también archivos .NET. Es compatible con todas las versiones de Windows desde Windows95 hasta Windows8. En este trabajo se ha utilizado la versión 3.7. Imple-

menta muchas técnicas anti-ingeniería inversa, incluyendo la opción de virtualizar funciones del programa.

La Tabla C.5 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Bajo**
Anti Desensamblado	**Medio**
Dummy Code	
Protección de Strings	
Anti Debugging	**Medio**
isDebuggerPresent	
Process32Next	
Anti Dumping	**Alto**
Ofuscación IAT	
Stolen Bytes	
Virtualizador	
Anti VM	**Medio**
Anti VMWare, VirtualPC, Sandboxie	
Anti Patching	**Medio**
Multihilo	

Tabla C.5: Protecciones en Enigma 3.7.

C.6. EXECryptor

Es un software propietario (`http://www.strongbit.com/execryptor.asp`) que permite proteger cualquier aplicación Win32, excepto programas que contienen código interpretado (p.ej .NET). Es compatible con cualquier versión de Windows desde Win95 hasta WinXP. En este trabajo se ha utilizado la versión demo 2.3.9 disponible en la web del fabricante, esta versión no tiene disponibles las funcionalidades de anti debugging ni protección del Entry Point.

Como particularidad el EXECryptor, además de las técnicas anti debugging, crea varios hilos de ejecución que llevan a cabo tareas anti debugging. También implementa una técnica denominada *CodeMorphing* que consiste en la ofuscación de código pero a nivel de instrucción CPU, de tal modo que se transforman las instrucciones en otras equivalentes pero diferentes. Además, esta técnica también transforma otros comandos en instrucciones de máquina virtual (P Codes).

La Tabla C.6 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Desensamblado	**Medio**
Code Morphing	
Anti Debugging	**Alto**
isDebuggerPresent	
checkRemoteDebuggerPresent	
outputDebugString	
NtGlobalFlag	
ProcessHeapFlag	
Multihilo	
Anti Dumping	**Alto**
Ofuscación IAT (API Redirection)	
Stolen Bytes	
Virtualización	
Anti VM	**Bajo**
Anti Patching	**Bajo**

Tabla C.6: Protecciones en EXECryptor 2.3.9.

C.7. EXE Stealth

Es una herramienta comercial (`http://www.webtoolmaster.com/exestealth.htm`) que permite comprimir y proteger cualquier ejecutable en formato PE. En este trabajo se ha utilizado la versión demo 4.19. Entre las protecciones que ofrece, destaca la implementación de las Páginas Guarda como método Anti Dumping.

La Tabla C.7 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

C.8. EXPressor

El EXPressor es una herramienta comercial (`http://www.cgsoftlabs.ro/express.html`) que permite comprimir y proteger cualquier aplicación 32/64 bits (.exe, .dll, .net. ocx, .scr). Es compatible con Win98, Win2000, XP y Vista. En este trabajo se ha utilizado la versión 1.8 con una licencia de uso obtenida de propio fabricante. Ofrece muy buenos niveles de compresión (Algoritmo LZMA [Wik13b]).

La Tabla C.8 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Desensamblado	**Bajo**
Anti Debugging	**Medio**
isDebuggerPresent	
outputDebugString	
NtGlobalFlag	
rdstc	
Anti Dumping	**Medio**
API Redirection (Ofuscación IAT)	
Páginas Guarda	
Modificación Cabecera PE	
Anti VM	**Bajo**
Anti Patching	**Bajo**
Checksum Protection	

Tabla C.7: Protecciones en EXEStealth 4.19.

C.9. FSG

Es una herramienta gratuita (`http://www.xtreeme.prv.pl`), permite comprimir ejecutables Win32 y es compatible con las versiones de Windows desde Windows 95 hasta Windows7. Se trata solamente de una herramienta compresora, utiliza la librería aPLib para la compresión. En este trabajo se ha utilizado la versión 2.0.

C.10. MEW

Es una herramienta gratuita (`http://web.archive.org/web/20070831063728/http://northfox.uw.hu/index.php?lang=eng&id=dev`) que permite comprimir ejecutables de Windows y es compatible con las versiones de Windows desde Windows 95 hasta Windows7. Se trata solamente de una herramienta compresora y utiliza el algoritmo de compresión LZMA [Wik13b]. En este trabajo se ha utilizado la versión 11.

C.11. Obsidium

Es una herramienta comercial (`http://www.obsidium.de/show/details/en`) que permite comprimir y proteger ejecutables de Windows, existe una version de 32 bits compatible con cualquier version de Windows desde Windows2000 a Windows 8 y otra version de 64 bits compatible con cualquier versión de Windows de 64 bits. Permite comprimir también ejecutables .NET. En este trabajo se ha utilizado la versión 1.5 disponible en la web del fabricante.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Debugging	**Medio**
Process32Next	
Self Debugging	
Protect Memory	
Anti Dumping	**Medio**
Ofuscación IAT (API Emuladas)	
Anti VM	**Bajo**
Anti Patching	**Bajo**

Tabla C.8: Protecciones en EXPressor 1.8.

Permite comprimir, cifrar (incluyendo descifrado dinámico en tiempo de ejecución de partes del código), ofuscar y virtualizar el código original del programa.

La Tabla C.9 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Medio**
Anti Desensamblado	**Medio**
Anti Debugging	**Alto**
isDebuggerPresent	
checkRemoteDebuggerPresent	
Detecta Hardware Breakpoints	
UnhandledExceptionFilter	
FindWindowA	
Process32Next	
Anti Dumping	**Alto**
Stolen Bytes	
Ofuscación IAT (APIs emuladas)	
Virtualizador	
Anti VM	**Bajo**
Anti Patching	**Bajo**

Tabla C.9: Protecciones en Obisidium 1.5.

C.12. PECompact

Es una herramienta comercial (`http://bitsum.com/pecompact/`), permite comprimir ejecutables, no se creó en un inicio para proteger ejecutables, es por eso que todos los mecanismos de protección están disponibles mediante plug-ins. Los plug-ins para este estudio son pec2ldr_default.dll y pec2ldr_debug.dll (que sólo tiene protecciones básicas anti debugging) y pec2ld_ead.dll que tiene protecciones adicionales. En este trabajo se ha utilizado la versión 3.02.2 disponible en la web del fabricante.

La Tabla C.10 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Debugging	**Medio**
isDebuggerPresent	
checkRemoteDebuggerPresent	
pec2ld_ead.dll	
Anti Dumping	**Bajo**
Ofuscación IAT (API redirection)	
Anti Patching	**Bajo**

Tabla C.10: Protecciones en PECompact 3.02.2.

C.13. PELock

Es una herramienta comercial (`http://www.pelock.com/`), permite comprimir y proteger ejecutables y es compatible con las versiones de Windows desde Windows95 a Windows XP/Vista. En este trabajo se ha considerado la versión demo 1.0.6 que no tiene disponibles las funcionalidades de anti dump ni redirección del código.

La Tabla C.11 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

C.14. PESpin

Es una herramienta gratuita (`http://pespin.w.interia.pl/`) que permite comprimir y proteger ejecutables. Es compatible con las versiones de Windows XP/Vista/7. Tiene una versión para 32-bit y otra para 64-bit. Para este trabajo se ha utilizado la versión 1.33.

La Tabla C.12 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Debugging	**Medio**
isDebuggerPresent	
Process32Next (OllyDbg, SoftICE)	
INT3	
Hardware Breakpoints	
Anti Dumping	**Medio**
Stolen Bytes	
Ofuscación IAT (API redirection)	
Code Redirection	

Tabla C.11: Protecciones en PELock 1.0.6.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Bajo**
Anti Desensamblado	**Medio**
Saltos falsos	
Anti Debugging	**Medio**
self Debugging	
Debugger INT	
Anti Dumping	**Alto**
Stolen Bytes	
Ofuscación IAT (API emulation)	
Nanomites	
Code Redirection	

Tabla C.12: Protecciones en PESpin 1.33.

C.15. Petite

Es una herramienta commercial (`http://www.un4seen.com/petite/`) que permite comprimir ejecutables Win32. Es compatible con versiones de Windows desde Windows95 hasta Windows7. También implementa una técnica de protección básica de ofuscación de la tabla de importaciones. En este trabajo se ha utilizado la versión 2.3.

C.16. Smart Packer

Es una herramienta comercial (`http://www.smartpacker.nl/`) de tipo Bundler, permite construir un solo ejecutable a partir de varios ficheros. También permite comprimir y cifrar el programa. Es compatible con versiones de Windows2000 o superiores. En este trabajo se ha utilizado la versión 1.9 con una licencia obtenida del propio fabricante.

C.17. TELock

Es una herramienta gratuita que permite comprimir y proteger ejecutables. No tiene página web disponible. En este trabajo se ha utilizado la última versión disponible que es la 0.98.

La Tabla C.13 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Debugging	**Bajo**
isDebuggerPresent	
Anti Dumping	**Bajo**
Ofuscación IAT	

Tabla C.13: Protecciones en TELock 0.98.

C.18. Themida

Es un protector y virtualizador comercial fabricado por Oreans Technology (`http://www.oreans.com/themida.php`), compatible con todas las versiones de Windows desde Windows95. En este trabajo se ha utilizado la versión demo 2.2.3, que tiene algunas funcionalidades limitadas. Es una herramienta protectora de las más potentes que existen en el mercado actualmente. Utiliza una tecnologia propia denominada *SecureEngine* que añade diversas protecciones al ejecutable, además de permitir la virtualización de partes del código.

La Tabla C.14 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Medio**
Anti Desensamblado	**Medio**
Garbage Code	
Código metamórfico	
Anti Debugging	**Alto**
isDebuggerPresent	
Anti Breakpoints	
Multihilo (Debugger Monitors)	
Anti Dumping	**Alto**
API wrapping (Ofuscación IAT)	
Anti API Spyer (Ofuscación IAT)	
Stolen Bytes	
Code Replace	
Anti VM	**Medio**
Anti VMWare, VirtualPC	
Anti Patching	**Medio**

Tabla C.14: Protecciones en Themida 2.2.3.

C.19. UPX

Es un compresor gratuito (`http://upx.sourceforge.net/`) disponible para Windows y Linux. Permite comprimir recursos, iconos y exports. La versión utilizada en este trabajo es la 3.91.

C.20. VMProtect

Es una herramienta comercial (`http://vmpsoft.com/`) que también permite virtualizar diversas partes del código. Es compatible con todas las versiones de Windows desde Windows 95 en adelante. En este trabajo se ha utilizado la versión demo 2.13, esta versión no tiene las funcionalidades de ofuscación de la IAT, ni anti patching.

La Tabla C.15 muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

C.21. Yodas Protector

Es una herramienta de uso libre (`http://yodap.sourceforge.net/`) permite comprimir y proteger un ejecutable. En este trabajo se ha utilizado la versión 1.03.3. La Tabla C.16

Protección	Nivel de protección
Compresión	**Bajo**
Cifrado	**Bajo**
Anti Desensamblado	**Medio**
Code Replace (Mutation)	
Anti Debugging	**Medio**
isDebuggerPresent	
Process32Next	
Anti Dumping	**Alto**
Ofuscación IAT	
Virtualización	
Anti VM	**Medio**
Anti VMWare, VirtualPC, Sandbox y VirtualBox	
Anti Patching	**Bajo**

Tabla C.15: Protecciones en VMProtect 2.13.

muestra las principales protecciones que implementa junto con el nivel de dificultad de desprotección.

Protección	Nivel de protección
Compresión	**Bajo**
Anti Desensamblado	**Bajo**
Protección de Strings	
Anti Debugging	**Bajo**
isDebuggerPresent	
anti SoftICE	
Anti Dumping	**Medio**
Ofuscación IAT (API Redirection)	

Tabla C.16: Protecciones en YodasProtector 1.03.3.