UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Análisis de técnicas de evasión usadas por malware para su orquestación en entornos aislados de ejecución.

Trabajo Fin de Máster

Máster Universitario en Ciberseguridad y Ciberinteligencia

AUTOR/A: Planells García, Germán

Tutor/a: Esteve Domingo, Manuel

CURSO ACADÉMICO: 2022/2023

Agradecimientos

Me gustaría expresar mi más sincero agradecimiento a todas las personas que han mostrado su apoyo y contribuido a la realización de este Trabajo de Fin de Máster (TFM).

En primer lugar, me gustaría hacer una mención especial, agradeciendo a mi familia por su constante respaldo y compresión mostrado a lo largo de esta etapa, logrando así mantenerme motivado y enfocado en el proyecto.

A la empresa S2Grupo por la oportunidad brindada y experiencia adquirida en el programa Enigma University.

A Ana Nieto Jiménez, por sus conocimientos transmitidos en el transcurso de la realización de este trabajo, otorgando diferentes herramientas para dar forma al trabajo y sirviendo de guía para el progreso del mismo.

A mi tutor Manuel Esteve Domingo por su orientación y proporcionarme retroalimentación valiosa para la mejora de la propuesta.

Resum

Anàlisi i classificació de les tècniques anti-anàlisis emprades actualment pel malware per a la definició de nous mecanismes i eines que permeten als analistes enganyar el malware perquè desplegue tot el seu arsenal en les màquines víctima, millorant així els sistemes sandbox i facilitant l'anàlisi dinàmica.

Paraules clau: malware, anti-anàlisi, sandboxing, anàlisi dinàmica

Resumen

Análisis y clasificación de las técnicas anti-análisis empleadas actualmente por el malware para la definición de nuevos mecanismos y herramientas que permitan a los analistas engañar al malware para que despliegue todo su arsenal en las máquinas víctima, mejorando así los sistemas sandbox y facilitando el análisis dinámico.

Palabras clave: malware, anti-análisis, sandboxing, análisis dinámico

Abstract

Analysis and classification of the anti-analysis techniques currently used by malware for the definition of new mechanisms and tools that allow analysts to trick malware into deploying its entire arsenal on victim machines, thus improving sandbox systems and facilitating dynamic analysis.

Key words: malware, ant-analysis, sandboxing, dynamic analysis

Índice general

Índice de figuras

Índice de tablas

Listings

Glosario

- **Malware:** Software malintencionado cuyo principal objetivo es dañar o realizar acciones no deseadas en el sistema en el que se ejecuta.
- **Baipasear:** Evitar o sortear un elemento, fundamentalmente cuando se trata de un obstáculo o molestia, que perjudique la acción que se está llevando a cabo.
- **Sandbox:** Máquina virtual aislada, usualmente empleada como entorno de pruebas, que permite ejecutar código potencialmente peligroso o inseguro sin afectar a los recursos hardware o a la maquina local.
- **API:** Una API (Application Programming Interface) es una capa de abstracción implementada por un software, la cual expone determinadas funcionalidades del programa para que puedan ser utilizadas por terceros, sin la necesidad de conocer su implementación.
- **Framework:** Código software ya implementado que proporciona unas funcionalidades y una estructura que permiten desarrollar software más completo de forma más rápida.
- **Callback:** Función que es utilizada como parámetro de otra función, siendo ejecutada posteriormente por esta última.
- **Proxy:** Elemento que permite interceptar y redefinir operaciones fundamentales de un objeto.
- **Hipervisor:** Supervisor de máquina virtual que permite crear y ejecutar máquinas virtuales, además de aislaras del resto del sistema y gestionar los recursos hardware que demanda.
- **Depurador:** Herramienta que permite analizar el código de un programa en busca de fallos o comportamientos no deseados en el sistema.

CAPÍTULO 1
Introducción

En este trabajo final del Máster Universitario en Ciberseguridad y Ciberinteligencia se ha buscado crear una herramienta capaz de baipasear las técnicas de evasión implementadas por el *malware* moderno destinadas a evitar su análisis dinámico en entornos virtualizados.

En primer lugar, se ha realizado un estudio de las técnicas de evasión implementadas con mayor frecuencia por el *malware*, para posteriormente, analizarlas y encontrar una solución que evite que sean capaces de detectar un entorno virtualizado. Para ello, se ha hecho uso de un proceso llamado instrumentación dinámica de binarios, el cual basa su funcionamiento en modificar las instrucciones de un programa binario en ejecución.

Una vez definida la solución, se ha creado un entorno de pruebas virtualizado con el sistema operativo Windows 10 usando la aplicación VirtualBox. Para la implementación de la solución se ha usado Frida, que consta de un conjunto de herramientas de instrumentación dinámica. Y, para las pruebas se ha hecho uso de la herramienta Pafish, la cual implementa un elevado número de técnicas de evasión en entornos virtualizados o *sandbox*.

Por último, se ha redactado un capítulo con las conclusiones que se han obtenido durante la realización de todo el presente trabajo.

1.1 Motivación

Con el paso del tiempo, el *malware* se ha vuelto cada vez más avanzando, implementando técnicas progresivamente más complejas para infiltrarse y perpetrar acciones no deseadas en los distintos tipos de sistemas informáticos. Este aumento en la complejidad de su implementación no se enfoca solo en las fases relacionadas con el ataque al sistema informático, sino que, también lo han hecho en las técnicas usadas por los atacantes para evitar su posterior análisis y estudio. Dentro de las técnicas usadas por los atacantes para dificultar el trabajo de los analistas de *malware*, se encuentran las técnicas de evasión en entornos aislados de ejecución, usadas por el *malware* para no ejecutar su carga maliciosa si detecta que está siendo ejecutado en un entorno virtual o está siendo depurado por un programa para analizar su comportamiento.

Es importante poder analizar el *malware* para conocer cuál ha sido el alcance del ataque que se ha perpetrado en el sistema, y para posteriormente, poder crear medidas que prevengan un ataque con un *malware* de las mismas características. Por esta razón, me ha parecido una buena opción como trabajo final de máster desarrollar una herramienta que sea capaz de baipaser las técnicas de evasión implementadas por el *malware* para facilitar el trabajo de los analistas. Por último, me gustaría destacar que este trabajo está dentro

del proyecto final de la formación Enigma impartida por la empresa S2 Grupo, estando dicho proyecto final tutorizado por Ana Nieto Jiménez.

La motivación personal que me ha llevado a elegir realizar este trabajo ha sido la búsqueda de mejorar como profesional en el campo del análisis de *malware*. Desarrollar este trabajo me ha permitido estudiar el funcionamiento de un programa en Windows a bajo nivel, desensamblándolo y observando cómo se comunica con la API (Application Programming Interface) de Windows. También he podido analizar *malware* real y aprender de su funcionamiento para entender las técnicas usadas por los atacantes para realizar su actividad delictiva.

1.2 Objetivos

El objetivo principal de este trabajo es lograr que la solución implementada sea capaz de evitar que el *malware* advierta que se encuentra en un entorno virtualizado Windows mediante la supresión e interferencia en la ejecución de sus técnicas de evasión, sin alterar el comportamiento malicioso del mismo.

Para alcanzar el cumplimiento de este objetivo se derivan los siguientes subobjetivos:

- Recopilar información sobre el funcionamiento de las técnicas de evasión implementadas por el *malware* y las distintas formas existentes hasta la fecha para baipasearlas.
- Selección de técnicas adecuadas para el diseño de una solución que permita satisfacer los requisitos del trabajo planteado.
- Implementación de la solución y creación del entorno de pruebas virtualizado.

1.3 Metodología

La metodología empleada para la realización de este trabajo ha sido el desarrollo iterativo, para ello se ha planteado el trabajo en fases incrementales que al término de cada una se ha iterado sobre la totalidad del proyecto con los conocimientos adquiridos en la última fase realizada. Se ha elegido dicha metodología, debido a que era necesario desarrollar el trabajo a la par que se adquirían los conocimientos para realizar la implementación y las pruebas.

1.4 Tecnologías

Las tecnologías utilizadas para el desarrollo de este trabajo de fin de máster han sido las siguientes:

- **JavaScrypt:** Es un lenguaje de programación interpretado con posibilidad de se ser compilado Just-In-Time (JIT). Está basado en prototipos, es multiparadigma, trabaja con un solo hilo de ejecución, es dinámico y cuenta con soporte para los paradigmas de programación imperativos, declarativos y orientados a objetos.
- **Python:** Es un lenguaje de programación interpretado de alto nivel. Es multiparadigma ya que cuenta con soporte para los paradigmas de programación imperativos, funcionales y orientados a objetos.

- **Ensamblador:** Es un lenguaje de programación de bajo nivel que no es ni interpretado ni compilado, sino que, cuenta en un conjunto de códigos simbólicos llamados mnemónicos, los cuales representan cada uno una instrucción que es entendible por el procesador.
- **VirtualBox:** Es un software de escritorio, multiplataforma, gratuito y de código abierto, que permite la ejecución de máquinas virtuales con una configuración variable según las necesidades del proyecto, permitiendo la realización de pruebas o la ejecución de programas maliciosos sin afectar al dispositivo real en el que se ejecuta.
- **x64/x32dbg:** Es un depurador binario de código abierto, gratuito y para Windows, destinado al análisis de *malware* y a la realización de ingeniería inversa de programas de los que no se dispone del código fuente.
- **Ghidra:** es un *Framework* de ingeniería inversa de software, incluye un conjunto de herramientas de análisis de software de alto nivel, las cuales, permiten a los usuarios analizar el código compilado en una variedad de plataformas, incluidas Windows, macOS y Linux. Sus funciones incluyen desensamblaje, ensamblaje, decompilación, creación de gráficos y secuencias de comandos entre otras muchas.

1.5 Estructura de la memoria

La estructura del presente trabajo de final de máster es la siguiente:

- **Capítulo 1:** se realiza una breve introducción al trabajo, se habla sobre la motivación de su realización incluyendo la personal, se listan los objetivos, se describe la estructura de esta memoria y se detallan las distintas tecnologías usadas para la elaboración de este trabajo.
- **Capítulo 2:** se realiza un análisis comparativo de las herramientas de instrumentación dinámica binaria que existen a día de hoy, a la vez que, se enumeran las distintas herramientas destinadas a la automatización del análisis dinámico que integran dichas técnicas.
- **Capítulo 3:** se estudian las técnicas de evasión usadas por el *malware* para dificultar tanto el análisis estático como el dinámico.
- **Capítulo 4:** se propone una solución para el cumplimiento de los objetivos marcados.
- **Capítulo 5:** se presentan las conclusiones obtenidas, a la vez que se expone como se podría continuar el trabajo en un futuro.

CAPÍTULO 2
Estado del arte

El análisis de *malware* [1] ha ido evolucionando a la par que la complejidad con la que se implementan los programas maliciosos. A su vez, los atacantes han pasado de realizar ataques con el objetivo de obtener prestigio, a realizar ataques que buscan destruir o comprometer los sistemas informáticos de una organización. Esto, llevó a la aparición de dos tipos de análisis, el análisis estático [2], en el cual se analiza el código sin ejecutar el programa y se usan herramientas para desensamblarlo y decompilarlo y de esta forma poder leer el código y, el análisis dinámico [3] en el que se se ejecuta el programa en un entorno seguro llamado *sandbox*. Esto permite estudiar su comportamiento mediante el uso de herramientas con capacidad de depurar el código.

Los últimos reportes presentados por el Departamento de Seguridad Nacional del Gabinete de la Presidencia del Gobierno de España y ENISA (European Union Agency For Cybersecurity) para el año 2022, presentan un incremento de los ataques cibernéticos, así como, una mejora en sus técnicas [4][5]. Esta tendencia, ha llevado a implementar algoritmos de *machine learning* para automatizar la clasificación y el análisis del incremento de *malware* producido [6][7] [8].

El perfeccionamiento, tanto en las herramientas como en las propias técnicas de análisis previamente mencionadas ha llevado a los atacantes a implementar una serie de técnicas conocidas como técnicas anti-análisis [9][10] [11] [12], que dificultan en gran medida el trabajo de los analistas. Estas técnicas van, desde ofuscar o encriptar el código para evitar el análisis estático [13][14] a, implementar funciones que sean capaces de comprobar si se encuentran en un entorno virtualizado con el objetivo de abortar la ejecución antes de ejecutar la carga maliciosa y, de esta forma, dificultar el análisis dinámico [15][16] . Esto conduce a la necesidad de buscar nuevas soluciones que permitan baipasear dichas técnicas sin alterar el comportamiento del *malware* a analizar [17][18] [19].

2.1 Instrumentación dinámica binaria

La instrumentación dinámica binaria [20] [21] consiste en la inyección de código instrumental en un programa de forma que el comportamiento original de este no se vea afectado y así poder estudiar su funcionalidad en tiempo de ejecución. La creciente popularidad de esta técnica proviene de varios factores. En primer lugar, la instrumentación es fácil de utilizar para agregar secuencias de monitorización, otro factor importante es que el usuario no necesita preocuparse por guardar el contexto o manipular el código a bajo nivel, sino que, solo necesita registrar el retorno de las *callbacks* devueltas por el tipo de instrucción especificado en el código instrumental. Esta capa de abstracción es de gran utilidad en distintos ámbitos de aplicación, como por ejemplo, depuración de software, análisis de vulnerabilidades, criptoanálisis, entre otros.

En el ámbito del análisis del *malware*, esta técnica ha despertado un gran interés debido a que tiene la capacidad de alterar el flujo normal de ejecución de un programa dadas unas condiciones especificadas por el analista. En concreto, es capaz de añadir, modificar y eliminar instrucciones, así como, alterar parámetros de entrada y retorno de funciones. Esta serie de modificaciones en el programa original pueden emplearse para baipasear las técnicas de evasión previamente mencionadas sin alterar el comportamiento del *malware*, permitiendo así, realizar un análisis dinámico del mismo sin que aborte su ejecución cuando se ejecuta en un entorno virtualizado o está siendo inspeccionado mediante un depurador [22].

Dentro de esta técnica, existen actualmente dos aproximaciones para llevar a cabo esta tarea, dynamic binary translation (DBT) y dynamic probe injection (DPI), las cuales se explican con más detalle a continuación:

- **Dynamic binary translation** basa su funcionamiento en un compilador JIT (Just-In-Time), el cual recibe como parámetro de entrada el programa a ser instrumentalizado. Generalmente, intercepta la ejecución de la primera instrucción de dicho programa y genera mediante compilación en tiempo de ejecución un nuevo código ensamblador a partir de las instrucciones subsiguientes. El código resultante es idéntico al original, pero con modificaciones destinadas a garantizar que la herramienta usada obtenga el control sobre el flujo de ejecución y sea capaz de redirigir dicho flujo al código instrumentalizado, siendo este el que realmente se ejecuta. El nuevo código, puede ser llevado a una caché de código para reutilizarlo eventualmente como forma de reducir sus tiempos de acceso y así, aumentar el rendimiento.

- **Dynamic probe injection** funciona en base a editar el código del programa cargado en memoria en tiempo de ejecución. De esta forma, reemplaza instrucciones en los bloques de código que se desea instrumentar, generalmente con instrucciones `jump` o `trap`, que conducen el flujo de ejecución del programa al código de instrumentación previamente insertado. Este proceso permite insertar nuevas instrucciones o modificar y eliminar instrucciones del código original. Una vez el código de instrumentación ha sido ejecutado, el flujo de ejecución es devuelto al código original del programa.

A pesar de la utilidad que aportan, ambas aproximaciones se ven afectadas por desventajas, las cuales se deben tener en cuenta a la hora de elegir una aproximación u otra, poniendo especial atención a las necesidades específicas de la implementación que se busque poner en práctica. En concreto, dynamic binary translation puede incurrir en tiempos altos de ejecución al inicio, debido a la necesidad de crear y manejar las copias creadas. Mientras que, dynamic probe injection, puede acaecer de estos problemas durante toda la ejecución, sobre todo, cuando el reemplazo de las instrucciones se realiza con instrucciones `trap` y no `jump`, debido a que estas provocan transferencias entre el espacio de kernel y el de usuario y eso implica un alto coste computacional.

2.2 Frameworks de Instrumentación dinámica binaria

Actualmente, existen una gran variedad de *frameworks* que ofrecen una capa de abstracción a la hora de implementar la instrumentación dinámica binaria, añadiendo además, una serie de herramientas que facilitan la escritura del código, el manejo de errores y un aumento en el rendimiento de la ejecución. Algunos *frameworks* son Pin, DynamoRIO, FRIDA, libdetox, Valgrind, QuarkslaB DBI o DynInst, entre otros.

Para el presente trabajo se ha hecho un estudio superficial de los *frameworks* disponibles, para posteriormente, hacer un estudio en profundidad de dos de ellos con el objetivo de determinar cuál se ajustaría mejor a las necesidades del proyecto. Los criterios seguidos para acotar la selección se han realizado en base a la madurez de su implementación, su compatibilidad con distintas plataformas y las mejoras que aplica sobre la técnica para aumentar su rendimiento en ejecución.

A continuación se expone una breve introducción de los dos *frameworks* seleccionados:

- **Pin [23]** es un *framework* de instrumentación dinámica binaria diseñado para soportar los conjuntos de instrucciones de las arquitecturas IA-32, x86-64 y MIC.
- **Frida [24]** es un conjunto de herramientas de instrumentación dinámica binaria que permite inyectar bloques de código JavaScript u otros lenguajes en aplicaciones nativas en Windows, macOS, GNU/Linux, iOS, watchOS, tvOS, Android, FreeBSD y QNX.

2.2.1. Pin

Pin fue diseñado en 2005 por Intel, dando soporte a los sistemas operativos más usados en el mercado de esa época (Windows, MacOS y Linux). Permite la creación de herramientas de instrumentación dinámica de forma que sean sencillas de usar, transparentes y portables, a través de un conjunto de funciones API.

Permite analizar un ejecutable con distintos niveles de detalle:

- **Rutina:** En el nivel de detalle de rutina, se puede inspeccionar e instrumentar una rutina completa después de cargar la imagen en memoria por primera vez. En Pin, una rutina representa cualquier función de un lenguaje de programación procedimental.
- **Imagen:** A nivel de detalle de imagen, Pin permite la inspección e instrumentación del código completo de una imagen cuando se carga por primera vez en memoria. En ese momento, la imagen ya contiene todas las estructuras de datos del programa, así como, las bibliotecas compartidas cargadas. De esta forma, puede recorrer cada sección de la imagen, cada rutina de una sección y cada instrucción de una rutina.
- **Traza:** El nivel de detalle de traza, permite inspeccionar e instrumentar cada traza del ejecutable. Pin considera que una traza es un bloque básico de código que normalmente comienza al principio del objetivo de una rama y termina con una instrucción que causa una rama incondicional, incluyendo llamadas y retornos.
- **Instrucción:** A nivel de detalle de instrucción, Pin permite inspeccionar e instrumentar el ejecutable instrucción por instrucción.

Basa su funcionamiento en 3 componentes principales, el primero de ellos es la herramienta que contiene el código de instrumentalización y análisis, a la cual se hace referencia en la documentación como Pintool. El segundo componente es el motor del *framework*, que consta de una máquina virtual, una caché de código y una API de instrumentación invocada por Pintool. Por último, se encuentra la aplicación que se va a instrumentar. Pin, es capaz de trabajar en dos modos, el primero JIT mode (Dynamic binary translation) y el segundo probe mode (Dynamic probe injection).

Cuando se ejecuta en JIT mode, Pin utiliza un compilador JIT, el cual es el responsable de insertar el código de instrumentación. Sin embargo, la entrada a este compilador no

es `bytecode`, sino un ejecutable normal. Pin intercepta la ejecución de la primera instrucción del programa y genera un nuevo código compilado para las siguientes líneas de código hasta que encuentra una rama que interrumpe la secuencia de ejecución lineal. Luego transfiere el control a la secuencia generada, la cual es casi idéntica a la original. Cuando una rama sale de la secuencia se asegura de recuperar el control y, después de recuperarlo, Pin repite el proceso para el código objetivo de la rama y continúa con la ejecución. De esta forma, el único código que se ejecuta es el código generado, el código original solo se usa como referencia. Pin, es capaz de hacer este proceso eficiente ya que el código instrumentado resultante se guarda en la caché de código, para que pueda reutilizarse y ramificarse directamente de una secuencia a otra con tiempos cortos de acceso a memoria. El esquema de la arquitectura de dicho proceso, se puede observar en la imagen 2.1 extraída de la presentación realizada en 2009 por el creador de Pin, Robert Cohn.

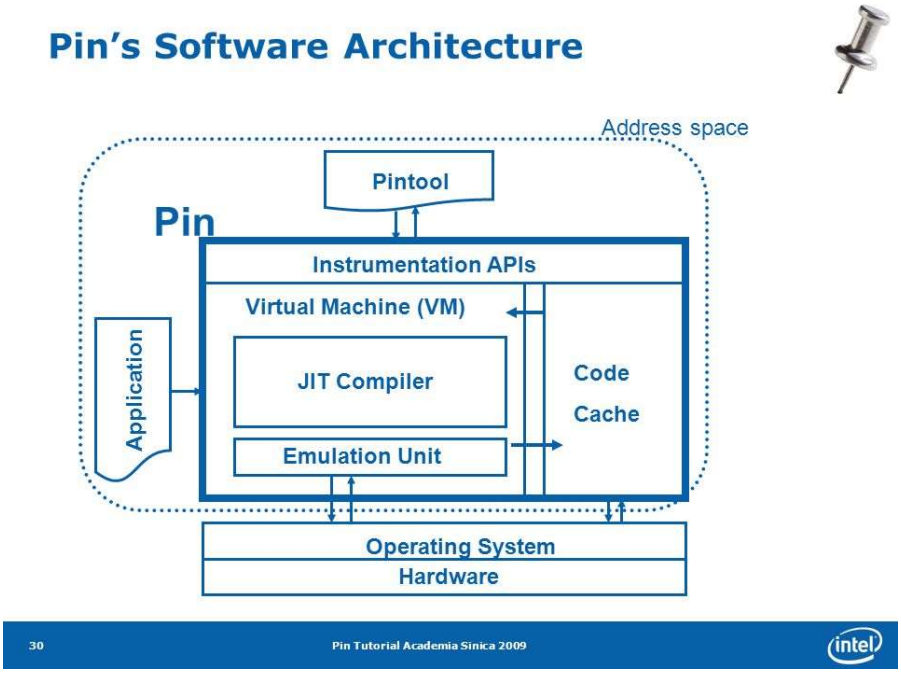

Figura 2.1: Arquitectura de Pin [25].

En probe mode, Pin inserta una instrucción `jump` al inicio de las rutinas que desea instrumentalizar. Dicha instrucción redirige el flujo al código de instrumentalización que el usuario ha implementado para esa rutina en concreto. Para que funcione, es necesario reubicar las instrucciones que se encontraban en la región de memoria donde se va a insertar el `jump`. De este modo, tiene mejor rendimiento que el JIT mode, pero pone más responsabilidad en el programador. Otra desventaja, es el hecho de que muchas de las funciones de la API del motor no están disponibles para este modo.

La técnica usada por Pin en probe mode para reemplazar instrucciones y redirigir el flujo es conocida como *trampoline hooking*. Como se puede observar en la imagen 2.2, se sustituyen las tres primeras instrucciones de la función que se desea *hookear* por una instrucción `jmp` que saltará a la función *proxy*, esta función contiene el código que en el caso de Pin, es el de instrumentación. Esto permite analizar los parámetros de entrada que ha recibido la función original, así como modificarlos, para provocar un comportamiento controlado distinto al original. El último paso, sería retornar el flujo a la función original, un *hook* normal reemplazaría los bytes modificados en el primer paso por los originales, y posteriormente, llamaría a la función con los parámetros modificados. Pero, esta forma de proceder requeriría instalar el *hook* cada vez que se quiera analizar la

ejecución de esa misma función. En cambio, *trampoline hooking* hace uso de un bloque de instrucciones al cual llama `trampoline`, que se encarga de ejecutar las instrucciones originales reemplazadas y retornar el flujo de ejecución a la instrucción de la función original que se encuentra inmediatamente después de la instrucción `jmp`, de esta forma, solo es necesario instalar el *hook* una única vez.

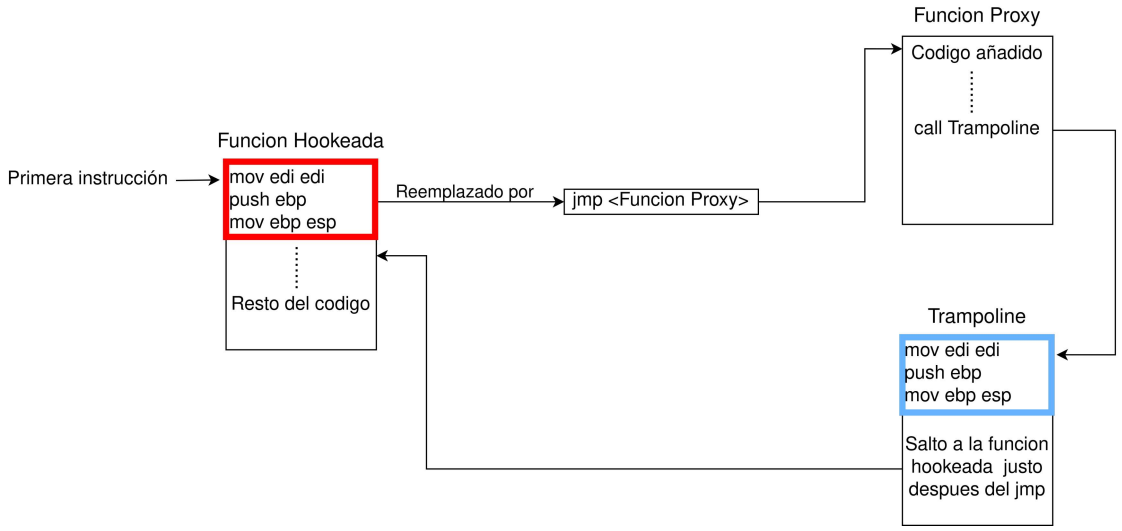

Figura 2.2: Trampoline hooking.

2.2.2. Frida

Frida fue desarrollado en 2013 por Ole Andre V. Ravnas, se creó con el objetivo de poder instrumentalizar un programa en ejecución, dando acceso al analista al flujo de ejecución y a los datos. Sus características posibilitan la inyección de código JavaScript, el cual, tiene la capacidad de modificar la ejecución original del programa.

La lista de sistemas y arquitecturas soportados es superior a la ofrecida por Pin, tal y como se puede observar a continuación:

- **Windows (x86, x64)**
- **Linux (x86, x64, arm, arm64, arm64e)**
- **FreeBSD**
- **MacOS (x86, x64, Apple Silicon M1)**
- **Android (incluido x86)**
- **iOS (arm64, arm64e, x64)**

La estructura seguida por Frida para instrumentalizar un proceso, es como la que se puede observar en la imagen 2.3. Consta de tres partes: el *script* de instrumentalización, el *script* de control y el proceso a instrumentalizar. El *script* de control se comunica con Frida a través de *bindings*, estos *bindings* son bibliotecas que dan soporte a la API de Frida en múltiples lenguajes (Python, Node, C#,...). Su rol principal es el de cargar el código de instrumentación e inyectarlo en el proceso que previamente ha sido creado suspendido, la creación del proceso en dicho estado es importante, ya que no debe ejecutarse antes de que el código sea inyectado. El *script* de control también es capaz de comunicarse con el

código de instrumentación inyectado en el proceso, enviando y recibiendo información relativa a la ejecución, permitiendo así la creación de archivos de *log* o mostrar información relativa al estado de ejecución. El *script* de instrumentación se encarga de interactuar con el proceso en ejecución, Frida permite implementar este *script* tanto en JavaScript como en TypeScript. El *script* de instrumentación es capaz entre otras muchas cosas de interceptar llamadas a funciones, monitorizar cualquier tipo de instrucción, modificar el flujo de ejecución del proceso y modificar parámetros de entrada y retorno de funciones.

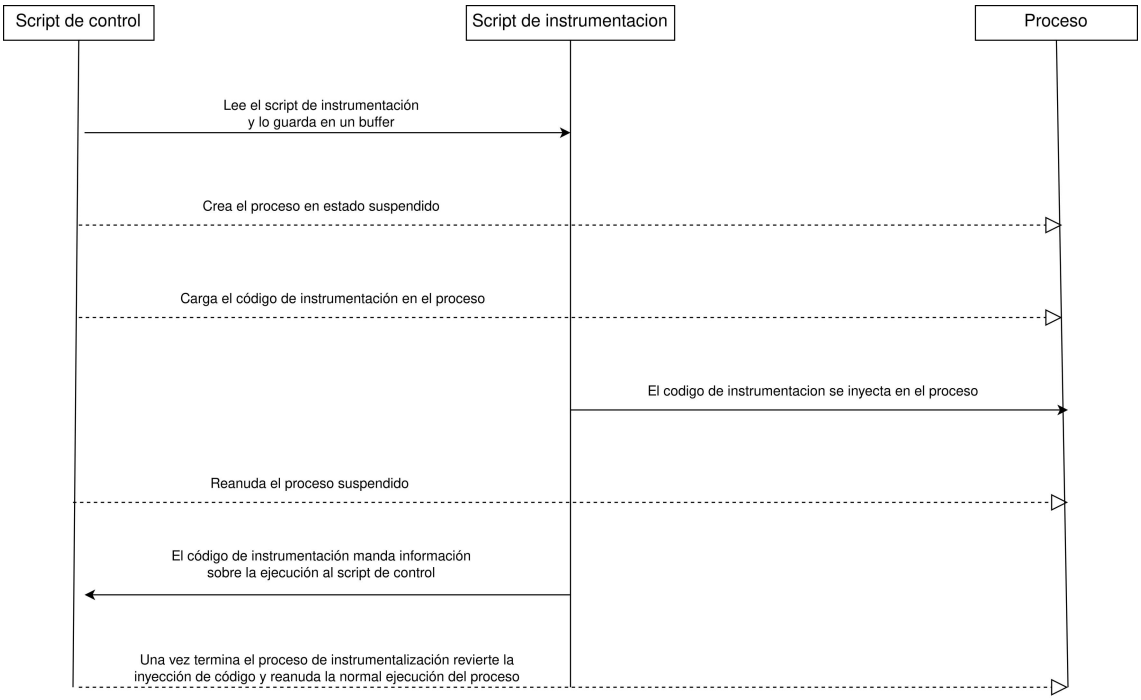

Figura 2.3: Estructura de instrumentalización de un proceso en Frida.

Existen dos herramientas principales que Frida provee para la instrumentación de código. La primera de ellas es Interceptor, la cual se basa en Dynamic probe injection, y la otra, Stalker, la cual está basada en Dynamic binary translation.

Stalker es un motor de rastreo de código que permite realizar un seguimiento transparente de cualquier instrucción o llamada a función realizada en el programa. Para ello, recompila el programa mientras este se encuentra en ejecución, el código máquina generado a través de este proceso, es una copia local capaz de ejecutarse en la CPU (Central Processing Unit). Este nuevo código se guarda en una caché de código que puede leerse, escribirse o ejecutarse según las necesidades del programa. De esta forma, el código modificado y ejecutado es el generado a partir del original, manteniendo este último intacto.

Interceptor ofrece una API que permite inspeccionar o modificar el flujo de ejecución en cada llamada a función. Para ello, hace uso de distintas técnicas, siendo la principal, *trampoline hooking*, técnica explicada en el *framework* anterior. A través de la API es posible hacer *hooking* fácilmente a funciones y secciones de código, para lo cual, hace uso de los `NativePointers`, un tipo definido por Frida que permite trabajar con punteros de memoria en los *scripts* de instrumentalización escritos en JavaScript. Gracias a esto, posibilita la invocación de dos importantes `callbacks`:

- **onEnter():** Se ejecuta cuando se llama a la función, permitiendo analizar y modificar los parámetros con los que ha sido llamada antes de que sean manipulados por el código. Además, facilita la posibilidad de conocer el estado de determinadas regiones de memoria antes de la ejecución de la función.

- **onLeave():** Se ejecuta cuando la función va a finalizar, por lo que se pueden analizar y modificar los valores de retorno, así como observar los cambios realizados por la función en los parámetros de entrada y otras regiones de memoria.

2.2.3. Conclusiones sobre los frameworks

Tanto Intel Pin como Frida son dos potentes *frameworks* de instrumentalización dinámica binaria que ofrecen herramientas tanto para analizar un programa, como modificar su flujo. En la Tabla 2.1, se puede ver una comparación de sus características básicas. Siendo similares en rendimiento, la elección de Frida sobre Intel Pin se ha debido a los siguientes motivos:

1. A diferencia de Pin, Frida es *open-source*, lo que posibilita que se pueda observar y entender el funcionamiento interno del *framework*. Esto es de gran utilidad para corregir errores o comportamientos no deseados de la herramienta implementada. A su vez, es posible añadir nuevas funciones en caso de ser necesarias.
2. En comparación con Pin, el desarrollo del código de instrumentación es más fácil y rápido, esto es debido a que la API de Frida está implementada con un enfoque más directo, por lo que reduce las llamadas necesarias para obtener un mismo resultado.
3. El soporte para móviles que ofrece Frida, permitiría reutilizar una parte del conocimiento ofrecido en este trabajo para desarrollar una herramienta similar destinada a analizar *malware* en dispositivos móviles.

	Pin	Frida
Soporte para distintos lenguajes	NO	SI
Soporte para instrumentalización en móviles	NO	SI
Multiplataforma	SI	SI
Open source	NO	SI

Tabla 2.1: Características básicas de los *frameworks* seleccionados.

CAPÍTULO 3
Técnicas anti-análisis

Las técnicas anti-análisis (también llamadas técnicas de evasión), son técnicas implementadas por el *malware*, orientadas a dificultar el análisis del comportamiento que mantienen en un sistema informático. Este análisis se puede realizar de dos formas, a través del estudio de su código (análisis estático) o mediante el estudio de los cambios realizados en el sistema durante su ejecución (análisis dinámico). De cara a evitar ambas formas de análisis, los desarrolladores de *malware* implementan, en primer lugar, una serie de técnicas destinadas a que el código no sea legible y, en segundo lugar, funcionalidades con el objetivo de detectar elementos del sistema que puedan indicar que se encuentran en un entorno virtualizado o están siendo depurados, para posteriormente, abortar la ejecución antes de ejecutar la carga maliciosa.

3.1 Análisis estático

En el contexto de la seguridad informática, la ofuscación consiste en la modificación del código a través de un conjunto de técnicas con el fin de que no sea legible, pero manteniendo su comportamiento inalterado en ejecución. En este apartado, se va a realizar un análisis de las técnicas más populares que se emplean con dicho propósito, así como, las variantes de *malware* que son capaces de evitar la detección durante el análisis estático.

Malware Cifrado

Un *malware* cifrado suele estar compuesto por un *payload* cifrado, un *decryptor* y una clave de cifrado. El *decryptor* tiene la función de descifrar el *payload* cada vez que se ejecuta el *malware*. Si la clave es modificada, la firma del *payload* será distinta, por lo que es imposible identificar a qué tipo de *malware* pertenece dicho código encriptado, aunque haya sido descifrado y analizado con anterioridad. Sin embargo, el código del *decryptor* permanece legible, por lo que es posible conocer el tipo de *malware* observando el patrón del código que emplea el *decryptor*.

Malware Oligomórfico

El *malware* oligomórfico es la evolución del *malware* cifrado. A diferencia de éste, es capaz de realizar en el *decryptor* pequeñas modificaciones que provocan un cambio en su firma y, por tanto, ya no sería un identificador único para determinar el tipo de *malware*. El problema de esta variante reside en que los pequeños cambios realizados, sólo permiten crear un número limitado de variantes (del orden de cientos). En consecuencia,

una herramienta de análisis estático automatizado puede detectar el *malware* una vez se hayan relacionado con el todas las posibles variantes que implementa.

Malware Polimórfico

El *malware* polimórfico, a diferencia del oligomórfico, consigue crear un número ilimitado de distintos *decryptors* empleando diversos métodos de ofuscación para crear las variantes. Este hecho hace que todas las partes que lo componen tengan una firma distinta en cada ejecución, por lo cual, puede evitar fácilmente la detección basada en firmas. A pesar de esto, el *payload* una vez descifrado es constante, por lo que puede utilizarse como una fuente de detección.

Malware Metamórfico

El *malware* metamórfico hace uso de un gran número de técnicas de ofuscación que modifican íntegramente su código creando una nueva versión tras cada ejecución, la cual parece diferente, pero funciona esencialmente igual. Para ello emplea un motor de generación de código que debe ser capaz de reconocer, analizar y mutar su propio cuerpo. A diferencia de las variantes de *malware* vistas anteriormente, el metamórfico nunca revela el *payload* constante en memoria debido a que no utiliza cifrado. Esto provoca que esta variante de *malware* sea muy difícil de detectar con analizadores estáticos.

Dead-Code Insertion

Dead-Code Insertion, es una técnica en la que se le añade a un programa instrucciones sin funcionalidad con el objetivo de cambiar la apariencia del código, pero sin que repercuta en su comportamiento durante la ejecución. Una de las formas más comunes de implementar esta técnica es a través de las instrucciones `nop`. Un ejemplo de esto se puede ver en los Listings 3.1 y 3.2, el primero consiste en un código en lenguaje C desensamblado que realiza una suma y una multiplicación, en el segundo, se han insertado bloques de instrucciones `nop` para cambiar su apariencia. A día de hoy, existen herramientas que permiten eliminar del programa este tipo de bloques de código de forma automática, por lo que esta técnica ha ido evolucionando hacia la inserción de bloques de código que realizan funcionalidad sin modificar la del programa, como por ejemplo, incrementar en uno el valor de un registro, guardarlo en pila, recuperarlo y decrementar su valor en uno.

```
1    1213: e8 98 fe ff ff        call   10b0 <__isoc99_scanf@plt>
2    1218: 8b 55 e8              mov    -0x18(%rbp),%edx
3    121b: 8b 45 ec              mov    -0x14(%rbp),%eax
4    121e: 01 d0                 add    %edx,%eax
5    1220: 89 45 f0              mov    %eax,-0x10(%rbp)
6    1223: 8b 55 e8              mov    -0x18(%rbp),%edx
7    1226: 8b 45 ec              mov    -0x14(%rbp),%eax
8    1229: 0f af c2              imul   %edx,%eax
9    122c: 89 45 f4              mov    %eax,-0xc(%rbp)
10   122f: 8b 45 f0              mov    -0x10(%rbp),%eax
11   1232: 89 c6                 mov    %eax,%esi
```

Listing 3.1: Ejemplo de codigo.

```
1213: e8 98 fe ff ff        call   10b0 <__isoc99_scanf@plt>
1218: 90                    nop
1219: 90                    nop
121a: 90                    nop
121b: 8b 55 e8              mov    -0x18(%rbp),%edx
121e: 8b 45 ec              mov    -0x14(%rbp),%eax
1221: 01 d0                 add    %edx,%eax
1223: 89 45 f0              mov    %eax,-0x10(%rbp)
1226: 90                    nop
1227: 90                    nop
1228: 90                    nop
1229: 8b 55 e8              mov    -0x18(%rbp),%edx
122c: 8b 45 ec              mov    -0x14(%rbp),%eax
122f: 0f af c2              imul   %edx,%eax
1232: 89 45 f4              mov    %eax,-0xc(%rbp)
1235: 90                    nop
1236: 90                    nop
1237: 90                    nop
1238: 8b 45 f0              mov    -0x10(%rbp),%eax
123b: 89 c6                 mov    %eax,%esi
```

Listing 3.2: Dead-Code Insertion con instrucciones nop.

Register Reassignment

Register Reassignment cambia los registros que usan las instrucciones del programa de una ejecución a otra manteniendo el código y su comportamiento exactamente igual. Un ejemplo de esta técnica se puede ver en el Listing 3.3, que basándose en el Listing 3.1 modifica los registros `edx` y `eax` usados por las instrucciones por los registros `edi` y `ecx`.

```
1213: e8 98 fe ff ff        call   10b0 <__isoc99_scanf@plt>
1218: 8b 7d e8              mov    -0x18(%rbp),%edi
121b: 8b 4d ec              mov    -0x14(%rbp),%ecx
121e: 01 f9                 add    %edi,%ecx
1220: 89 4d f0              mov    %ecx,-0x10(%rbp)
1223: 8b 7d e8              mov    -0x18(%rbp),%edi
1226: 8b 4d ec              mov    -0x14(%rbp),%ecx
1229: 0f af cf              imul   %edi,%ecx
122c: 89 4d f4              mov    %ecx,-0xc(%rbp)
```

Listing 3.3: Ejemplo de Register Reassignment.

Subroutine Reordering

Subroutine Reordering es capaz de ofuscar el código original de un programa a través de la reordenación aleatoria de sus subrutinas. Como resultado, es posible generar n! diferentes variantes, donde n es la cantidad de subrutinas. Poniendo el caso de un programa con veinte subrutinas, obtendremos un total de 20! = 2.432902e+18 posibles variantes del mismo programa.

Instruction Substitution

Instruction Substitution modifica el código original sustituyendo algunas instrucciones por una o varias que realizan una función equivalente. Por ejemplo, la instrucción `mov %edx, %ecx` puede ser sustituida por el conjunto de instrucciones formado por la instrucción `push %edx` seguido de una instrucción `pop %ecx`.

Code Transposition

Code Transposition reordena las secuencias de código del programa original sin que se vea alterado el flujo de ejecución. Para ello, uno de los métodos más utilizados es mezclar aleatoriamente las instrucciones del programa y reconstruir el flujo original mediante la inserción de saltos incondicionales o instrucciones `jmp`. Un ejemplo de esto se puede ver en el Listing 3.4, que basándose en el Listing 3.1, cambia el orden en que se ejecuta la suma y la multiplicación.

```
1     1213: e8 98 fe ff ff          call   10b0 <__isoc99_scanf@plt>
2     1218: eb 0e                   jmp    1228 <sum>
3 000000000000121a <mult>:
4     121a: 8b 7d e8                mov    -0x18(%rbp),%edi
5     121d: 8b 4d ec                mov    -0x14(%rbp),%ecx
6     1220: 0f af cf                imul   %edi,%ecx
7     1223: 89 4d f4                mov    %ecx,-0xc(%rbp)
8     1226: eb 0d                   jmp    1235 <cont>
9 0000000000001228 <sum>:
10    1228: 8b 7d e8                mov    -0x18(%rbp),%edi
11    122b: 8b 4d ec                mov    -0x14(%rbp),%ecx
12    122e: 01 f9                   add    %edi,%ecx
13    1230: 89 4d f0                mov    %ecx,-0x10(%rbp)
14    1233: eb e5                   jmp    121a <mult>
15 0000000000001235 <cont>:
16    1235: 8b 45 f0                mov    -0x10(%rbp),%eax
```

Listing 3.4: Ejemplo de Code Transposition.

Code Integration

Code Integration, es una técnica que consiste en la fusión de un proceso malicioso con un programa benigno. Con ese objetivo, el *malware* decompila el código del programa objetivo en objetos manipulables para, posteriormente, añadir su código entre dichos objetos y reensamblar el código combinado en un nuevo ejecutable.

Encoded Code

Encoded Code es una técnica de ofuscación por la cual, el código de un programa se modifica mediante un algoritmo de codificación con el objetivo de que este no sea legible. El algoritmo más conocido para este propósito es Base64. Es esencialmente un esquema de codificación de 64 caracteres, siendo el carácter de relleno el signo = (igual). Su alfabeto incluye las letras [a-z, A-Z], los caracteres [+ y /] y los números [0-9]. El algoritmo de codificación funciona encadenando tres caracteres para generar una cadena de 24 bits que, posteriormente, se divide en cuatro fragmentos de 6 bits, cada uno de los cuales se traduce a uno de los caracteres del alfabeto de Base64.

Packing the code

Packing the code es una técnica de ofuscación en la que el código del *malware* se comprime con el objetivo de no ser legible hasta que se encuentre en ejecución. El proceso de empaquetado genera un nuevo ejecutable como se puede ver en la Figura 3.1, el cual se compone principalmente de dos elementos: el código binario del *malware* empaquetado y un código de desempaquetado. El nuevo *entry point* (OEP) del ejecutable será el código de desempaquetado. De tal forma, que cuando es ejecutado desempaqueta el código

en memoria durante la ejecución, maneja los *imports* del *malware* y redirige el flujo de ejecución al OEP del *malware*, iniciando así su ejecución.

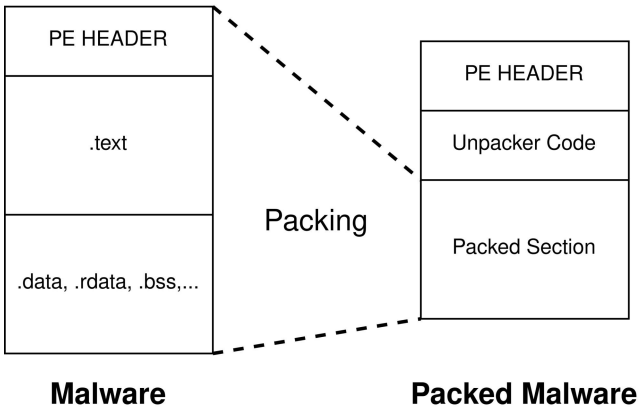

Figura 3.1: Packed Malware.

3.2 Análisis dinámico

Las técnicas de anti-análisis orientadas a impedir el análisis dinámico del *malware*, hacen uso de una gran cantidad de estrategias para detectar si el *malware* está siendo ejecutado en un entorno virtual o está siendo estudiado mediante un software de depuración y, en caso de obtener alguna evidencia de uno de estos dos hechos, abortar su ejecución. Las estrategias empleadas con este fin se pueden agrupar en dos grupos: detección observable y detección deductiva.

3.2.1. Detección observable

El software de virtualización crea una serie de rasgos distintivos únicos en el sistema virtualizado como pueden ser archivos, procesos en ejecución, registros, etc. De igual modo, al virtualizar el hardware se suele crear con la misma configuración, por lo que se convierte en un identificador único más. Estos rasgos distintivos son comúnmente llamados artefactos, y son generados también por el software de depuración. Las estrategias de anti-análisis agrupadas bajo el nombre de detección observable son todas aquellas que basan su detección en la observación de estos identificadores únicos, por lo que no necesitan de una interpretación para determinar si se encuentran en un sistema virtualizado. A continuación, se va a exponer una recopilación de algunas de las técnicas que implementan dichas estrategias tomando a VirtualBox como objetivo a detectar.

Artefactos creados por el software de virtualización

El software de virtualización crea en el entorno virtualizado Windows una serie de subclaves de registro. Para detectar si están presentes en el sistema, el *malware* hace uso de la función `RegOpenKeyExA()`, la cual devuelve ERROR_SUCCESS (valor = 0x0) si la función se ha ejecutado correctamente y la subclave está presente en el sistema. De lo contrario, el valor devuelto es un código de error distinto de cero. Las subclaves generadas por Virtualbox se pueden ver en la Tabla 3.1.

A su vez, existen distintas subclaves de registro que se encontrarían en un sistema real, pero que Virtualbox crea estableciendo unos valores que son identificativos de la presencia de un entorno virtualizado. El *malware* hace uso de la función `RegQueryValueExA()`

Elemento al que hace referencia	Nombre de la subclave del registro
VirtualBox Guest Additions	SOFTWARE\\Oracle\\VirtualBox Guest Additions
ACPI	HARDWARE\\ACPI\\DSDT\\VBOX__
FADT ACPI	HARDWARE\\ACPI\\FADT\\VBOX__
RSDT ACPI	HARDWARE\\ACPI\\RSDT\\VBOX__
VirtualBox Services	SYSTEM\\ControlSet001\\Services\\VBoxGuest
	SYSTEM\\ControlSet001\\Services\\VBoxService
	SYSTEM\\ControlSet001\\Services\\VBoxMouse
	SYSTEM\\ControlSet001\\Services\\VBoxSF
	SYSTEM\\ControlSet001\\Services\\VBoxVideo

Tabla 3.1: Subclaves del registro creadas por Virtualbox.

para conocer el valor de dichas subclaves. En la Tabla 3.2 se pueden observar las subclaves cuyo valor es establecido por Virtualbox de forma que actúa como un identificador único.

Elemento al que hace referencia	Nombre de la subclave de registro	Nombre del valor del registro	Datos del valor
SCSI	HARDWARE\\DEVICEMAP\\Scsi\\Scsi Port 0 \\Scsi Bus 0\\Target Id 0\\Logical Unit Id 0"	Identifier	VBOX
SystemBiosVersion	HARDWARE\\Description\\System	SystemBiosVersion	VBOX
VideoBiosVersion	HARDWARE\\Description\\System	VideoBiosVersion	VIRTUALBOX
SystemBiosDate	HARDWARE\\DESCRIPTION\\System	SystemBiosDate	06/23/99

Tabla 3.2: Datos de valor de subclaves del registro creadas por Virtualbox.

Por otro lado, es conocido que las máquinas virtuales contienen funciones y controladores especiales que ayudan al sistema operativo virtualizado a funcionar como si estuviera cargado en la placa base. Esta funcionalidad se encuentra normalmente en archivos que en su mayoría se encuentran en las rutas C:\windows\System32\Drivers\ y C:\windows\System32\. Los desarrolladores de *malware* pueden realizar una simple búsqueda de alguno de estos archivos conocidos para determinar si se encuentran en un entorno virtualizado. A continuación se puede ver una lista con los archivos generados por VirtualBox, que son buscados por el *malware* con mayor frecuencia:

- **C:\windows\System32\Drivers\VBoxMouse.sys**
- **C:\windows\System32\Drivers\VBoxGuest.sys**
- **C:\windows\System32\Drivers\VBoxSF.sys**
- **C:\windows\System32\Drivers\VBoxVideo.sys**
- **C:\windows\System32\vboxservice.exe**
- **C:\windows\System32\vboxtray.exe**
- **C:\windows\System32\VBoxControl.exe**
- **C:\windows\System32\vboxdisp.dll**
- **C:\windows\System32\vboxhook.dll**
- **C:\windows\System32\vboxmrxnp.dll**

- **C:\windows\System32\vboxogl.dll**
- **C:\windows\System32\vboxoglarrayspu.dll**
- **C:\windows\System32\vboxoglcrutil.dll**
- **C:\windows\System32\vboxoglerrorspu.dll**
- **C:\windows\System32\vboxoglfeedbackspu.dll**
- **C:\windows\System32\vboxoglpackspu.dll**
- **C:\windows\System32\vboxoglpassthroughspu.dll**

Por último, existen numerosos procesos en ejecución que están estrechamente relacionados con la existencia de un entorno virtualizado. Un desarrollador de *malware* dispone de múltiples formas de obtener la lista de procesos en ejecución, por lo que solo tendría que comparar los nombres de estos procesos conocidos con la lista obtenida para detectar que se encuentra en un sistema virtualizado. Algunos de los procesos en ejecución más conocidos en sistemas virtualizados por Virtualbox son `vboxservice.exe` y `vboxtray.exe`.

Detección basada en direcciones MAC (Media Access Control) conocidas

La mayoría del software de virtualización incluye la función de virtualizar una tarjeta de red. Si el entorno virtualizado está funcionando con una tarjeta que no ha sido personalizada, el *malware* podrá detectarlo, ya que la tarjeta por defecto es creada con la misma dirección MAC o cuenta con muy pocas variantes tal y como se puede ver en la Tabla 3.3. Como ejemplo de este hecho, un programa malicioso sería capaz de recuperar la información de las direcciones asociadas a los adaptadores mediante la función `GetAdaptersAddresses()` y comparar el valor de la dirección MAC devuelto con los de la Tabla 3.3.

Software de virtualizacion	Inicio de la dirección MAC
VirtualBox	08:00:27
VMware	00:05:69
VMware	00:0C:29
VMware	00:1C:14
VMware	00:50:56

Tabla 3.3: Direcciones MAC conocidas de software de virtualización.

Detección basada en la información de la CPU

En la arquitectura x86, `cpuid` es una instrucción complementaria del procesador que permite que el software obtenga distinta información de este según el valor de entrada. Cuando el *malware* hace uso de dicha instrucción, al analizar y comparar el valor que se devuelve, puede concluir en qué tipo de sistema se ejecuta. Si se está ejecutando en un entorno virtualizado, la instrucción `cpuid` devuelve un valor conocido según el software de virtualización que se esté empleando.

Principalmente, el *malware* obtiene la información necesaria a través de la instrucción `cpuid` de dos formas distintas. La primera, consiste en proporcionar a `cpuid` un valor

de entrada en el registro `EAX` igual a 0x40000000, esto hará que la instrucción escriba en los registros `EBX`, `ECX` y `EDX` los valores de la marca del hipervisor. Una vez ha obtenido estos valores, solo tiene que darles formato tal y como se puede ver en el Listing 3.5, para posteriormente compararlos con los valores conocidos de las distintas marcas de hipervisor recopiladas en la Tabla 3.4.

```
int ebx = 0, ecx = 0, edx = 0;
__asm__ volatile("cpuid" \
    : "=b"(ebx), \
    "=c"(ecx), \
    "=d"(edx) \
    : "a"(0x40000000));
sprintf(vendor  , "%c%c%c%c", ebx, (ebx >> 8), (ebx >> 16), (ebx >> 24));
sprintf(vendor+4, "%c%c%c%c", ecx, (ecx >> 8), (ecx >> 16), (ecx >> 24));
sprintf(vendor+8, "%c%c%c%c", edx, (edx >> 8), (edx >> 16), (edx >> 24));
```

Listing 3.5: Uso de la instruccion CPUID para conocer el proveedor del hipervisor.

Marca del hipervisor	Valor
VirtualBox	VBoxVBoxVBox
VMware	VMwareVMware
KVM	KVMKVMKVM\0\0\0
Microsoft Hyper-V	Microsoft Hv
Xen	XenVMMXenVMM
Parallels	prl hyperv

Tabla 3.4: Valores de las distintas marcas de hypervisor conocidas.

El otro método empleado por el *malware* para obtener esta información consiste en ejecutar la instrucción `CPUID`, pero esta vez con un valor en el registro `EAX` de 1. Esta llamada devuelve una respuesta de 31 bits, los cuales contienen distinta información del procesador. En una máquina real, el último bit de esta respuesta es igual a 0, en cambio, en una máquina virtual es igual a 1. En el Listing 3.6 se puede ver un ejemplo de código que devuelve `True` o `False` dependiendo del valor de este último bit.

```
int ecx;
__asm__ volatile("cpuid" \
    : "=c"(ecx) \
    : "a"(0x01));
return (ecx >> 31) & 0x1;
```

Listing 3.6: Comprobación del bit del hipervisor a traves de la instrucción cpuid.

Detección basada en la lectura del Process Environment Block

El Process Environment Block (PEB) es una estructura de datos que existe por cada proceso en el sistema y que contiene información relativa al proceso. El PEB contiene varios campos que pueden ser leídos por un *malware* para detectar si están siendo depurados por un programa. El campo más utilizado para este propósito es el `BeingDebugged`, el cual puede ser leído directamente o mediante las funciones de la API de windows que retornan el valor de dicho campo, como por ejemplo `isDebuggerPresent()` o `CheckRemoteDebuggerPresent()`.

3.2.2. Detección deductiva

Los entornos virtuales y los depuradores exponen una serie de indicadores que, a pesar de no ser un hecho explícito de su presencia, permiten deducirla. Las estrategias de esta categoría se basan en la observación de estos indicadores para comprobar si están dentro de un rango. Dicho rango ha sido previamente calculado en base a diferentes parámetros como por ejemplo, tiempos de ejecución o número de interacciones del usuario con el sistema.

Detección basada en tiempos

Normalmente, los entornos virtuales cuentan con unos recursos hardware limitados, esto produce que su rendimiento sea menor que el de un entorno real. Las técnicas basadas en detección de tiempos aprovechan este hecho asumiendo que una determinada función o un conjunto de instrucciones, se debería ejecutar en una cantidad máxima de tiempo, y una vez superado ese umbral establecer que se encuentran ante un entorno virtual. Para medir los tiempos, el *malware* puede utilizar funciones de la API de Windows como por ejemplo `GetTickCount()` pero, de cara a obtener una mayor precisión pueden usar la instrucción `rdtsc` del procesador, la cual devuelve el tiempo en ciclos de CPU. En el Listing 3.7 se puede observar la medición del tiempo de ejecución en ciclos de la instrucción `cpuid`. Ejecutando este código cien veces y obteniendo la media de todos los tiempos, se puede obtener un valor fiable del tiempo de ejecución del sistema, que comparado con la ejecución previa del mismo código en un entorno real puede determinar si es un entorno virtual o no.

```
1 unsigned long long tiempo, tiempo2, tiempoTotal;
2 unsigned eax, edx;
3 asm volatile("rdtsc" : "=a" (eax), "=d" (edx));
4 tiempo = ((unsigned long long)eax) | (((unsigned long long)edx) << 32);
5   asm volatile("cpuid" :: "a"(0x00));
6 asm volatile("rdtsc" : "=a" (eax), "=d" (edx));
7 tiempo2 = ((unsigned long long)eax) | (((unsigned long long)edx) << 32);
8 tiempoTotal = tiempo2 - tiempo;
```

Listing 3.7: Medición de tiempo de ejecucion de la instruccion cpuid.

Detección basada en características hardware

Como se ha mencionado previamente, los entornos virtuales cuentan con unos recursos hardware limitados. Esto hace posible que el *malware* pueda deducir que se encuentra en un entorno virtual debido a unas características inferiores a lo usual. Las principales características hardware que busca un *malware* son:

- El espacio de memoria en disco que se suele asignar a las máquinas virtuales no supera los 100GB, por lo que si un *malware* comprueba el tamaño de disco verá que el valor devuelto es anómalo.
- La memoria principal es un recurso escaso en los ordenadores, por lo que la cantidad que se le puede asignar a una máquina virtual se suele limitar a un valor entorno a los 2GB, que de ser comprobado es un indicativo de la presencia de un entorno virtualizado.
- Al igual que con la memoria principal, el número de procesadores es un recurso limitado en un ordenador, por lo que el número que se le asigna normalmente a las

máquinas virtuales no supera el par, lo que lo convierte en otro campo que buscará comprobar el *malware*.

Detección mediante pruebas de Turing inversas

Las pruebas de turing inversas dentro del mundo de las técnicas anti-análisis consisten en la capacidad de detectar si existe interacción humana con el sistema. Se basan en que muchas máquinas virtuales destinadas al análisis dinámico son *sandbox* automatizadas que no tienen ningún tipo de interacción humana en su funcionamiento, por lo que si un *malware* no observa interacción humana, puede determinar que se encuentra en un entorno virtualizado y por lo tanto, abortar su ejecución. Existen múltiples formas de llevar a cabo estas comprobaciones. Por ejemplo, una forma muy empleada es instalar un *hook* mediante la función `SetWindowsHookExA()` al tipo de procedimiento `WH_MOUSE_LL`, esto instala un procedimiento de enlace que supervisa los mensajes del ratón. De esta forma, el *malware* es capaz de esperar a que se realice un *click* en el ratón para ejecutar su carga maliciosa. Este proceso se puede utilizar de la misma forma pero con el tipo de procedimiento `WH_KEYBOARD_LL` para detectar si existen pulsaciones en el teclado. Otra forma bastante extendida, consiste en buscar las últimas entradas del usuario. Para ello, el *malware* suele aprovechar la combinación de funciones `GetTickCount()` y `GetLastInputInfo()` para calcular el tiempo de inactividad del usuario.

Debido a que para contrarrestar este tipo de técnicas se han creado herramientas que simulan el comportamiento humano, dichas técnicas han evolucionado para tener la capacidad de detectar el comportamiento humano generado digitalmente, ya que en su mayoría suelen estar basados en patrones fáciles de detectar. Un ejemplo de este tipo de técnicas es la que utiliza la función `GetCursorPos()`. Esta función devuelve la posición del cursor del ratón en coordenadas de pantalla. Esto, permite en primer lugar, detectar interacción humana si entre dos llamadas a la función existe un cambio en las coordenadas devueltas. Pero, también es posible establecer un valor umbral el cual determina la velocidad máxima a la que un ser humano puede mover el ratón, y de ser superado, implicaría que el movimiento es demasiado rápido para ser generado por un humano y el *malware* abortaría su ejecución.

CAPÍTULO 4
Solución propuesta

En este capítulo se va a exponer la solución propuesta que se ha construido en base a todo lo anteriormente mostrado en este trabajo. En primer lugar, se ha decidido emplear Frida para la implementación de la herramienta, ya que como se puede ver en el apartado 2.2.2, permite realizar la implementación de los *scripts* en lenguaje JavaScript y, a través de la API que ofrece, instrumentalizar fácilmente un programa. En concreto, se va a hacer uso de la capacidad que tiene para introducir *hooks* mediante la técnica de *trampoline hooking*, ya que en su mayoría, las técnicas de evasión basan su funcionamiento en llamadas a las funciones de las librerías dinámicas que forman la API de Windows. Del mismo modo, también facilita el uso del lenguaje de programación Python para desarrollar el *script* de control. Ambos lenguajes empleados son ampliamente conocidos y cuentan con una extensa documentación que facilita tanto la implementación de dicho trabajo, como la comprensión futura del código por terceras personas.

Como software de virtualización se ha optado por usar VirtualBox, los motivos han sido los conocimientos previos a este trabajo que se tenían de la herramienta, así como, que es también la única solución profesional que está disponible gratuitamente como software de código abierto según los términos de la licencia pública general GNU (GPL) versión 3. El sistema operativo elegido para ser virtualizado ha sido Windows, en concreto, su versión Windows 10 Pro. Debido a esto, una gran parte del trabajo ha consistido en analizar y estudiar tanto las técnicas de evasión diseñadas para este sistema en particular como, el propio funcionamiento de Windows de cara a poder diseñar una mejor solución. Se ha elegido Windows debido a las necesidades del trabajo final de enigma, en el que este trabajo está enmarcado. A pesar de esto, Windows es el sistema operativo de escritorio más utilizado en el mundo según datos de statcounter, a fecha de Julio de 2023 cuenta con una cuota de mercado de 69,52 %. Por consecuencia, una cantidad importante del *malware* que se desarrolla, lo hace con la intención de afectar a este sistema, por lo que es necesario enfocar más recursos en su protección.

Por último, para la fase de realización de pruebas se ha usado Pafish, la cual es una herramienta de pruebas gratuita y de código abierto que utiliza diferentes técnicas para detectar máquinas virtuales y entornos de análisis de *malware*. Para ello, recopila técnicas de evasión empleadas por *malware* real agrupadas según el software o elemento que buscan detectar y que delata la presencia de un entorno virtual. Las agrupaciones que realiza se pueden ver a continuación:

- **Técnicas de detección de depuradores**
- **Técnicas de detección basadas en la información de la CPU**
- **Pruebas genéricas de turing inversas**

- **Técnicas genéricas de detección de sandbox**
- **Técnicas de detección de hooks**
- **Técnicas de detección de Sandboxie**
- **Técnicas de detección de wine**
- **Técnicas de detección de VirtualBox**
- **Técnicas de detección de VMware**
- **Técnicas de detección de Qemu**
- **Técnicas de detección de Bochs**
- **Técnicas de detección de cuco**

4.1 Arquitectura e implementación propuesta

La herramienta desarrollada en base al esquema de funcionamiento de Frida se constituye de dos componentes, el primero de ellos es un archivo escrito en lenguaje Python que actúa como *script* de control. El segundo componente, son los distintos archivos que contienen el código de instrumentación. Para facilitar la comprensión del código y que estos archivos no fueran excesivamente extensos, se ha decidido crear uno por cada agrupación de técnicas de evasión que se busca baipasear. Como se puede ver en la Figura 4.1, el funcionamiento de la herramienta se puede dividir en varias fases claramente diferenciadas, las cuales se van a explicar en detalle a continuación junto al código del *script* de control que las realiza.

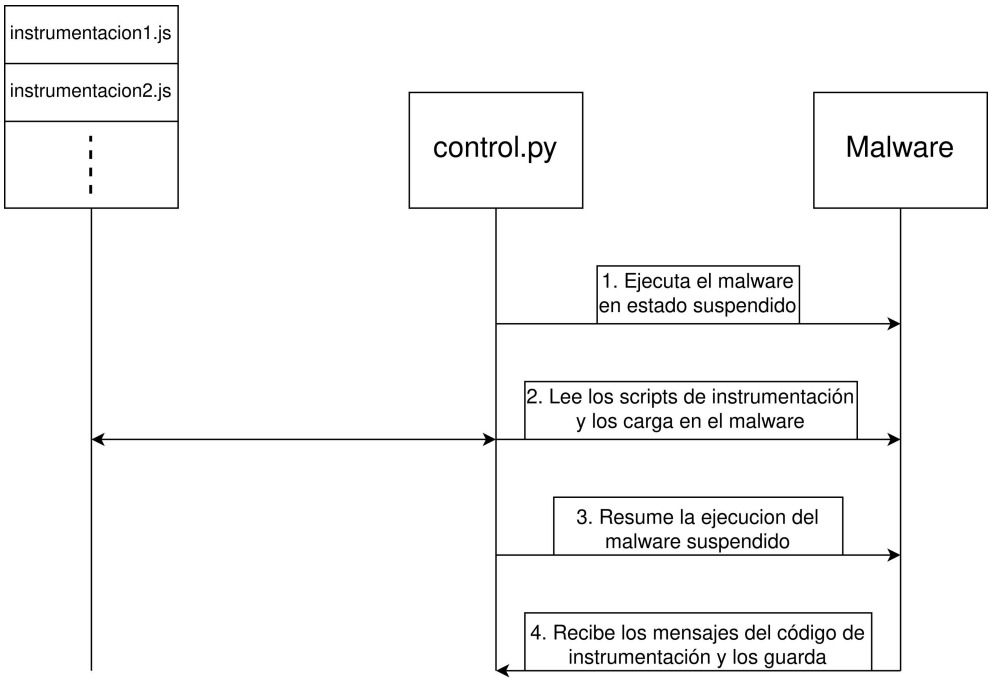

Figura 4.1: Esquema del funcionamiento de la herramienta.

En la fase 1 el archivo de control se encarga de crear el proceso suspendido asociado al *malware* que se desea instrumentalizar. El código que realiza dicha funcionalidad se puede observar en el Listing 4.1, en la primera línea se obtiene el dispositivo local, que a

lo largo de este trabajo siempre va a ser el sistema operativo virtualizado. Una vez se ha obtenido el `device`, se crea el proceso suspendido mediante la función `spawn()`. Para ello, es necesario mandarle como parámetro la ruta donde se encuentra el ejecutable. Esta ruta se recibe a través de la ejecución por línea de comandos de la instrucción `python control.py process_name`. Como resultado de la ejecución de la función, se obtiene el PID (Process ID) del proceso, el cual se muestra por pantalla. Por último, se crea un objeto de sesión a través de la función `attach()`, lo que nos permite interactuar con el proceso cuyo PID ha sido enviado por parámetro.

```
1    device = frida.get_local_device()
2    pid = device.spawn(process_name)
3    print('pid: %d' % pid)
4    session = device.attach(pid)
```

Listing 4.1: Código del script de control correspondiente a la Fase 1.

A continuación, en la fase 2 el *script* de control lee los *scripts* de instrumentalización y los carga en el *malware*. El código que realiza este proceso se puede examinar en el Listing 4.2. En primer lugar, como se puede ver en las líneas [1-3], se recorre mediante un bucle un `array` que contiene la ruta de todos los archivos de instrumentalización. Cada archivo es abierto y su código se concatena en una variable que al finalizar el bucle contiene todo el código de los distintos archivos. Esta variable es pasada como parámetro a la función `create_script()`. Como resultado de la ejecución de la función, se retorna un objeto `script` que es el que permite posteriormente cargar el código instrumentalizado mediante la función `load()`, tal y como se puede observar en la línea 7. Previamente a la ejecución de `load()`, en la línea 6, se asigna qué función se ha de ejecutar cuando se reciba la *callback* cuyo identificador es `message`. De esta forma, se habilita la recepción de mensajes que se explicará en detalle en la Fase 4.

```
1    for file in files:
2        with open(file, 'r') as script_file:
3            code = code + script_file.read()
4
5    script = session.create_script(code)
6    script.on('message', on_message)
7    script.load()
```

Listing 4.2: Código del script de control correspondiente a la Fase 2.

En la fase 3, se reanuda la ejecución del proceso suspendido, para ello se ejecuta la línea de código `device.resume(pid)`, la cual indica al sistema operativo el PID del proceso a reanudar. La creación del proceso suspendido y su posterior reanudación es debido a que la carga del código de instrumentalización se tiene que realizar antes de que el *malware* se esté ejecutando. De otra forma, la carga del código no sería posible y la ejecución terminaría con una excepción.

Por último, en la fase 4 el *script* de control recibe los mensajes del código instrumentalizado que se encuentra cargado en el *malware*. En la fase 2 se ha visto que mediante la línea de código `script.on('message', on_message)`, se establece que la función `on_message()` debe ser ejecutada cada vez que llegue una *callback* de tipo `message`. El código de dicha función se puede observar en el Listing 4.3, y consiste en la creación de un fichero de *log*. La implementación se ha realizado de tal forma que para cada ejecución se cree un archivo distinto para poder llevar un registro de todas las ejecuciones, esto se consigue mediante la creación del nombre del archivo del *log* usando como formato la fecha y hora del momento de ejecución del *script* de control y, almacenándolo en

una variable global (_LOG_FILENAME) para que la función pueda tener acceso, de esta forma nunca existirán dos ficheros iguales. Entrando más en detalle, la función recibe dos parámetros, de los cuales, el primero contiene el mensaje que ha enviado el código en el campo payload, siendo este el contenido que acaba escribiéndose en el *log* tal y como se puede observar en la línea 3.

```
1     def on_message(message, data):
2         file1 = open(_LOG_FILENAME, "a")
3         file1.write(message['payload']+"\n")
4         file1.close()
```

Listing 4.3: Código del script de control correspondiente a la Fase 4.

Una vez se ha explicado la implementación del script de control y, cómo funciona el flujo de ejecución de la herramienta, se va a mostrar la implementación que se ha realizado de los *scripts* de instrumentación, los cuales, contienen la lógica que permite baipasear las técnicas de evasión. Para ello es necesario en primer lugar, exponer como Frida realiza la instrumentación a nivel de código. Con este propósito se ha creado un ejemplo usando la función wcsstr(), el cual se va a explicar a continuación y cuyo código se puede observar en el Listing 4.4.

En primer lugar, es necesario obtener el puntero a la dirección de memoria de la función wcsstr(), la cual se encuentra en la librería MSVCRT.DLL. Frida es capaz de obtener dicho puntero a través de la función Module.getExportByName() pasándole como parámetros el nombre de la función y el nombre de la librería donde la función se encuentra, como se puede ver en la línea 1. El siguiente paso, es definir una clase que contenga las *callbacks* predefinidas por Frida, las cuales son onEnter() y onLeave(). La primera de ellas es lanzada cuando la función wcsstr() es llamada y, a través de su parámetro de entrada args, el cual contiene los argumentos de la función representados como una matriz de punteros, es posible tanto visualizar como modificar los argumentos según las necesidades del analista. En cambio, la segunda *callback* es lanzada cuando se va a ejecutar el return de la función wcsstr(), permitiendo acceder a su valor de retorno, tanto para visualizarlo, como para modificarlo. Mediante la modificación del parámetro de retorno, se es capaz de influir en el flujo de ejecución del programa analizado, logrando así comportamientos distintos a los implementados por el desarrollador del *malware*. Por último, es preciso interceptar la llamada a wcsstr() para añadir toda la funcionalidad que permite que las *callbacks* funcionen. Esto se consigue mediante la función Interceptor.attach(), que como se puede ver en la línea 8, recibe como parámetros el puntero a la dirección de memoria de la función wcsstr() y la clase que contiene las *callbacks* y el código implementado en ellas. Con esta función, Frida se encarga de realizar toda la implementación necesaria para que mediante la técnica de *trampoline hooking*, se intercepte la función wcsstr() añadiendo el código de instrumentalización de las *callbacks*, y posteriormente el flujo de ejecución retorne al código del programa.

```
1 const wcsstrPtr = Module.getExportByName("MSVCRT.DLL", "wcsstr");
2 class wcsstr {
3     onEnter(args) {
4     }
5     onLeave(retval){
6     }
7 }
8 Interceptor.attach(wcsstrPtr, new wcsstr);
```

Listing 4.4: Código de ejemplo que expone el funcionamiento de la instrumentación con Frida.

Una vez expuesto el funcionamiento de la instrumentación que Frida realiza, se puede explicar en detalle la implementación realizada para baipasear los distintos tipos de técnicas de evasión. Con este propósito, se va a mostrar a continuación el código de instrumentación diseñado para cada técnica, acompañado de su correspondiente explicación.

RegOpenKeyEx

La función `RegOpenKeyEx()` es usada por el *malware* para descubrir subclaves de registro que indiquen la presencia de un entorno virtual. Para evadir dichas comprobaciones se ha implementado el código del Listing 4.5. En este, se accede al segundo argumento de la función `RegOpenKeyEx()`, que como puede observar en la documentación de Windows corresponde al "Nombre de la subclave del registro que se va a abrir". Este argumento, se almacena en una variable global con el objetivo de que sea accesible cuando la función va a terminar su ejecución. La razón de que deba ser accesible al finalizar reside en la necesidad de comprobar a que subclave se está accediendo para no alterar el comportamiento de la carga maliciosa del *malware*, ya que la instrumentalización se aplica a todas las llamadas a esta función, y es posible que el *malware* las realice con otros fines distintos a aplicar una técnica de anti-análisis y por tanto, no deban ser alteradas. Por ello, se realiza la comprobación de la presencia de las cadenas `vbox` y `virtualbox` en el nombre de la subclave, y de ser así se modifica el valor de retorno `ERROR_SUCCESS (0x0)` por `ERROR_INVALID_FUNCTION (0x1)`. Indicando de esta forma al programa malicioso que la subclave no está presente en el sistema, aunque realmente si lo esté. Como añadido, la línea 10 ejecuta la función `send()`, la cual se encarga de enviar información al *script* de control tal y como se ha mencionado previamente, en este caso se envía el nombre del registro que ha sido comprobado por el *malware* en relación con una técnica de evasión.

```
1  class RegOpenKeyEx {
2
3      firstParam = null;
4
5      onEnter(args) {
6          this.firstParam = args[1].readAnsiString()
7      }
8      onLeave(retval){
9          if((this.firstParam.toLowerCase().indexOf("vbox") != -1) || (this.
               firstParam.toLowerCase().indexOf("virtualbox") != -1)){
10             send("Registro comprobado por el malware: "+this.firstParam)
11             retval.replace(0x1)
12         }
13     }
14 }
```

Listing 4.5: Código de instrumentación para la función RegOpenKeyEx.

RegQueryValueExA

La función `RegQueryValueExA()` es empleada por el *malware* para conocer los datos del valor de una clave de registro. Para baipasear la técnica anti-análisis que emplea dicha función se ha implementado el código del Listing 4.6. A diferencia de la implementación anterior, el argumento que es necesario almacenar en una variable global es un puntero a un búfer que recibe los datos del valor durante la ejecución, por lo que la información a la que se busca acceder no va a estar disponible durante la ejecución de la

callback `onEnter()`. A pesar de esto, es posible almacenar en la variable global el puntero a memoria, para posteriormente leer el valor que contiene cuando la función vaya a finalizar, tal y como se puede ver en la línea 10. Las siguiente líneas, comprueban que el valor que el *malware* buscaba tiene relación con una técnica de evasión, y de ser así, reemplazar el valor devuelto `ERROR_SUCCESS (0x0)` por el de `ERROR_INVALID_FUNCTION (0x1)`.

```
1  class RegQueryValueExA {
2
3      firstParam = null
4      text = null
5
6      onEnter(args) {
7          this.firstParam = args[4]
8      }
9      onLeave(retval){
10         this.text = this.firstParam.readAnsiString()
11         if((this.text.toLowerCase().indexOf("vbox") != -1) || (this.text.
               toLowerCase().indexOf("virtualbox") != -1) || (this.text.
               toLowerCase().indexOf("06/23/99") != -1)){
12             retval.replace(0x1)
13         }
14     }
15 }
```

Listing 4.6: Código de instrumentación para la función RegQueryValueExA.

GetFileAttributesA

Dentro de las técnicas de evasión, la función `GetFileAttributesA()` es utilizada para descubrir la presencia de archivos en el sistema que sean indicativo de que dicho sistema sea un entorno virtual. Para evitar que la comprobación se lleve a cabo con éxito se ha implementado el código del Listing 4.7. En el cual, se accede al primer argumento de la función al momento de ser llamada. Este argumento, contiene el nombre del archivo o directorio del cual se busca recuperar los atributos del sistema de archivos. Una vez accedido, se almacena en una variable global para poder tener acceso al contenido de dicho argumento cuando la ejecución de la función vaya a finalizar. En dicho momento, se comprueba que el nombre del fichero, contenga las cadenas de caracteres relacionadas con los ficheros generados por un entorno virtual y, de ser así, se envía la información al *script* de control y se reemplaza el valor devuelto por el de `INVALID_FILE_ATTRIBUTES -1 ($ffffffff)`.

GetAdaptersAddresses

Las máquinas virtuales cuentan con adaptadores de red, que si no son modificados traen la configuración por defecto que les proporciona el software de virtualización, siendo esta configuración, siempre la misma. Con el objetivo de detectar el entorno virtual, el *malware* emplea la función `GetAdaptersAddresses()` para visualizar si la dirección física del adaptador empieza con 08:00:27 (Virtualbox). Con el fin de evitar esta comprobación, es necesario acceder al cuarto argumento de la función, el cual consiste en un puntero a un búfer que contendrá al finalizar la ejecución de la función una lista enlazada de estructuras `IP_ADAPTER_ADDRESSES`. En el Listing 4.8, se puede ver que al final de la ejecución, se comprueba que el puntero contenga información, y de ser así, se desplaza el puntero a la dirección de memoria que representa al parámetro `PhysicalAddress`

```
class GetFileAttributesA {

    firstParam = null;

    onEnter(args) {
        this.firstParam = args[0].readAnsiString()
    }
    onLeave(retval){
        if((this.firstParam.indexOf("VBox") != -1) || (this.firstParam.
            toLowerCase().indexOf("vbox") != -1) || (this.firstParam.
            toLowerCase().indexOf("virtualbox") != -1) ){
            send("Archivo comprobado por el malware: "+this.firstParam)
            retval.replace(0xffffffff)
        }
    }
}
```

Listing 4.7: Código de instrumentación para la función GetFileAttributesA.

dentro de la estructura `IP_ADAPTER_ADDRESSES`. Este desplazamiento ha sido calculado previamente, según la información que proporciona la documentación de Windows sobre esta estructura y el tamaño de los parámetros que la componen. Una vez se ha realizado el desplazamiento, se modifica el contenido de la dirección mediante la escritura de `0x1` para que no coincida con un valor conocido de los asignados por un software virtual.

```
class GetAdaptersAddresses {

    firstParam = null;

    onEnter(args) {
        this.firstParam = args[3];
    }
    onLeave(retval){
        if(this.firstParam != 0){
            this.firstParam = this.firstParam.add(80)
            this.firstParam.writeInt(1);
        }
    }
}
```

Listing 4.8: Código de instrumentación para la función GetAdaptersAddresses.

FindWindowA

El código implementado que se puede observar en el Listing 4.9, intercepta la llamada a la función `FindWindowA()` con la intención de evitar la búsqueda por parte del *malware* de la existencia de una ventana, cuyo nombre de clase o nombre de ventana coincidan con el nombre de un artefacto conocido que pertenezca a un entorno virtual. Para ello, se accede al primer y segundo argumento de la función, los cuales contienen el nombre de clase y nombre de ventana respectivamente. Posteriormente, si el valor accedido no es `null` y es uno de los valores que tienen relación con un entorno virtual, se procede a modificar el valor de retorno de la función por un cero en vez de retornar el identificador de la ventana.

```
class FindWindowA {

    firstParam = null;
    secondParam = null;

    onEnter(args) {
        this.firstParam = args[0];
        this.secondParam = args[1];
    }
    onLeave(retval){
        if((this.firstParam != 0)){
            this.text = this.firstParam.readAnsiString();
            if((this.text.toLowerCase().indexOf("vbox") != -1)){
                retval.replace(0x0)
            }
        }else if((this.secondParam != 0)){
            this.text = this.secondParam.readAnsiString();
            if((this.text.toLowerCase().indexOf("vbox") != -1)){
                retval.replace(0x0)
            }
        }
    }
}
```

Listing 4.9: Código de instrumentación para la función FindWindowA.

CreateFileA

`CreateFileA()` es otra función empleada por el *malware* para comprobar la existencia de ficheros que revelan la presencia de un entorno virtual. La implementación de la solución que se puede observar en el Listing 4.10 es muy similar a la planteada para evadir la comprobación de la función `GetFileAttributesA()`. La única diferencia, radica en que en el caso de que se compruebe de que se está intentando crear o abrir un fichero que sea un artefacto conocido, se modifica el valor de retorno normal por un menos uno, lo que equivale a `INVALID_HANDLE_VALUE`.

```
class CreateFileA {

    firstParam = null;

    onEnter(args) {
        this.firstParam = args[0].readAnsiString()
    }
    onLeave(retval){
        if(this.firstParam.indexOf("VBox") != -1){
            send("Archivo comprobado por el malware: "+this.firstParam)
            retval.replace(-1)
        }
    }
}
```

Listing 4.10: Código de instrumentación para la función CreateFileA.

WNetGetProviderNameA

El nombre del proveedor de red en un entorno virtual creado por Virtualbox es un valor conocido, este hecho es aprovechado por los desarrolladores de *malware* para crear

una técnica anti-análisis basándose en la función `WNetGetProviderNameA()`. Para evitar dicha técnica, se ha implementado el código del Listing 4.11. En el cual, se almacena en una variable global el segundo argumento de la función cuyo valor es un puntero a un búfer que recibe el nombre del proveedor de red durante la ejecución de la función. De esta forma, al finalizar la función se comprueba si el nombre del proveedor de red hace referencia al que asigna Virtualbox y, de ser así, se devuelve un `ERROR_INVALID_FUNCTION` en vez de un `NO_ERROR`.

```
class WNetGetProviderNameA {

    firstParam = null;

    onEnter(args) {
        this.firstParam = args[1];
    }
    onLeave(retval){
        this.text = this.firstParam.readAnsiString()
        if(this.text.toLowerCase().indexOf("virtualbox") != -1){
            retval.replace(1)
        }
    }
}
```

Listing 4.11: Código de instrumentación para la función WNetGetProviderNameA.

Process32Next

Las máquinas virtuales ejecutadas por Virtualbox, tienen varios procesos activos que a través de su detección se puede determinar la existencia de un entorno virtual. El *malware* hace uso de la función `Process32Next()` para este propósito, ya que permite recorrer todos los procesos del sistema y visualizar el nombre de su archivo ejecutable. Con el propósito de evitar la detección mediante esta técnica, se ha implementado el código del Listing 4.12. En el, se puede observar el acceso al segundo argumento de la función, el cual es un puntero a la estructura `PROCESSENTRY32`, la cual contiene una descripción de una entrada de la lista de los procesos que residen en el espacio de direcciones del sistema cuando se toma una instantánea a través de la función `CreateToolhelp32Snapshot()`. Una vez se ha accedido al argumento, se realiza un desplazamiento de memoria dentro de la estructura para leer el parámetro `szExeFile` y, en caso de que sea alguno de los que tiene relación con VirtualBox modificarlo por `prueba.exe`.

```
class Process32Next {

    firstParam = null;

    onEnter(args) {
        this.firstParam = args[1];
    }
    onLeave(retval){
        this.text = this.firstParam.add(44).readAnsiString()
        if((this.text.toLowerCase().indexOf("vbox") != -1)){
            this.firstParam.add(44).writeAnsiString("prueba.exe")
        }
    }
}
```

Listing 4.12: Código de instrumentación para la función Process32Next.

GetDiskFreeSpaceExA

Los entornos virtuales cuentan con recursos hardware limitados en comparación con los entornos reales, esto es aprovechado por el *malware* para detectar su existencia. Una de las funciones más utilizadas con dicho propósito es `GetDiskFreeSpaceExA()`. La cual es capaz de recuperar la información sobre el número total de bytes en un disco a través de su tercer argumento, `lpTotalNumberOfBytes`. Para evadir la detección mediante esta técnica, se ha sobrescrito la dirección de memoria correspondiente al argumento antes de que el flujo de ejecución retorne al usuario, tal y como se puede observar en el Listing 4.13. De forma que, cuando el *malware* compruebe el tamaño de disco mediante esta función, obtendrá que su capacidad es de 483183820800 bytes = 450 gigabytes.

```
1  class GetDiskFreeSpaceExA {
2
3      firstParam = null;
4
5      onEnter(args) {
6          this.firstParam = args[2];
7      }
8      onLeave(retval){
9          this.firstParam.writeU64(483183820800)
10     }
11 }
```

Listing 4.13: Código de instrumentación para la función GetDiskFreeSpaceExA.

GetTickCount

`GetTickCount()` es una función que recupera el número de milisegundos transcurridos desde que se inició el sistema. Un programa malicioso puede beneficiarse de esta función bajo la premisa de que un sistema que acaba de iniciarse, y ejecuta un archivo malicioso, es con toda probabilidad un entorno virtual empleado para analizar *malware*. Para evitar que un entorno virtual sea detectado mediante esta técnica se ha implementado un código que intercepta la función y modifica su valor de retorno, dicho código se puede observar en el Listing 4.14. En la línea 6, se recupera el valor que la función iba a devolver originalmente y, se almacena en una variable a la cual se le va a sumar `0x124F80`, número que equivale a veinte minutos. Esta cifra, se ha calculado en base a la observación de esta técnica empleada en distintos archivos de *malware*, no superando ninguna el valor umbral de veinte minutos en su detección.

```
1  class GetTickCount {
2
3      onEnter(args) {
4      }
5      onLeave(retval){
6          var time = retval;
7          retval.replace(time.add(0x124F80));
8      }
9  }
```

Listing 4.14: Código de instrumentación para la función GetTickCount.

GetCursorPos

La posición del cursor es un factor importante dentro de las técnicas de evasión, ya que existen múltiples entornos virtuales que realizan análisis automatizado y que no cuentan con interacción humana. Esto, es aprovechado por el *malware* para diferenciar un entorno real de uno virtual a través del uso de la función `GetCursorPos()`. Para evitar la identificación a través de esta función, se ha implementado el código del Listing 4.15. En el, se intercepta la función para modificar el único argumento del cual dispone, `lpPoint`, el cual consiste en un puntero a una estructura `POINT` que recibe durante la ejecución de la función las coordenadas de pantalla del cursor. Para simular interacción humana con el sistema, es necesario modificar los parámetros x e y de la estructura `POINT` con unos valores aleatorios que se modifiquen con cada ejecución distinta de la función. El código de las líneas [11-14] realiza esta función, acotando unos valores para los números generados de forma aleatoria, que se correspondan con los que podría realizar un humano con un dispositivo que permita mover el cursor.

```
1  class GetCursorPos {
2
3      firstParam = null;
4      x = null;
5      y = null;
6
7      onEnter(args) {
8          this.firstParam = args[0]
9      }
10     onLeave(retval){
11         this.x = Math.floor(Math.random() * 1000);
12         this.firstParam.writeLong(this.x)
13         this.y = Math.floor(Math.random() * 1000);
14         this.firstParam.add(8).writeLong(this.y)
15     }
16 }
```

Listing 4.15: Código de instrumentación para la función GetCursorPos.

4.2 Prueba de concepto

En esta sección, se van a mostrar tanto las pruebas realizadas como la configuración del entorno virtual creado para la realización de las mismas. Como se ha mencionado anteriormente, para la realización de las pruebas se va a usar Pafish, esta herramienta de *testing* utiliza diferentes técnicas para detectar máquinas virtuales y entornos de análisis de *malware* del mismo modo que lo hacen las familias de *malware*. La elección de dicha herramienta ha permitido desarrollar más rápido la solución debido a que, al estar todas las pruebas centralizadas en una herramienta, los fallos en los resultados de las pruebas eran más sencillos de interpretar y subsanar. Por otro lado, para el entorno de pruebas, se ha empleado Virtualbox junto con una imagen de disco Windows 10 Pro. La configuración utilizada para la creación e instalación de la imagen ha sido la estándar sin realizar ninguna modificación a los componentes hardware virtualizados, salvo por la asignación de recursos, la cual se puede observar en la Tabla 4.1. El aumento de los recursos asignados para la realización de este proyecto frente a los que Virtualbox asigna por defecto, es debido a la necesidad de aumentar el rendimiento del entorno virtual para la implementación de la solución y la ejecución de las pruebas presentadas.

Componente Hardware	Valor asignado
Número de procesadores	4
Memoria física	8192
Almacenamiento	50GB
Memoria de vídeo	128MB

Tabla 4.1: Recursos asignados al entorno de pruebas.

La etapa de pruebas ha consistido en una fase inicial donde se ha ejecutado una batería de técnicas ofrecida por Pafish con el objetivo de observar qué técnicas eran las que detectaban la presencia de un entorno virtual sin añadir ninguna herramienta de instrumentación. Posteriormente, se han agrupado las técnicas de evasión que lograban detectar el entorno virtual según la lógica de su implementación de cara a probar la solución que se estaba realizando. A continuación, se van a exponer las distintas agrupaciones de técnicas, así como las pruebas realizadas y su efectividad para cada una de ellas.

Técnicas de detección de VirtualBox

Técnicas de detección de VirtualBox	Detección preherramienta	Detección postherramienta
Reg key Scsi port->bus->target id-> logical unit id->0 identifier	X	
Reg key (HKLM\\HARDWARE\\Description \\System \"SystemBiosVersion\	X	
Reg key (HKLM\\SOFTWARE\\Oracle\\ VirtualBox Guest Additions	X	
Reg key (HKLM\\HARDWARE\\Description\\ System \"VideoBiosVersion\	X	
Reg key (HKLM\\HARDWARE\\ACPI\\DSDT\\ VBOX__	X	
Reg key (HKLM\\HARDWARE\\ACPI\\FADT\\ VBOX__	X	
Reg key (HKLM\\HARDWARE\\ACPI\\RSDT\\ VBOX__	X	
Reg key (HKLM\\SYSTEM\\ControlSet001\\ Services\\VBox*	X	
Reg key (HKLM\\HARDWARE\\DESCRIPTION\\ System \"SystemBiosDate\	X	
Driver files in C:\\WINDOWS\\system32\\ drivers\\VBox*	X	
Archivos de sistema adicionales	X	
Buscando una dirección MAC que empiece con 08:00:27	X	
Buscando pseudo devices	X	
Buscando VBoxTray windows	X	
Buscando VBox network share	X	
Buscando VBox processes	X	
Buscando VBox devices usando WM	X	

Tabla 4.2: Técnicas de detección de VirtualBox.

Las técnicas que se pueden ver en la Tabla 4.2 son todas las técnicas de evasión implementadas por Pafish cuyo objetivo es detectar la presencia de un entorno virtual creado por el software de virtualización VirtualBox. En ella, se puede observar que todas las técnicas son capaces de detectar el entorno virtual, pero una vez se ha aplicado la solución implementada se consiguen inhibir en su totalidad. Esto es debido a que la detección que realizan estas técnicas se basa en artefactos conocidos cuya búsqueda en el sistema se realiza a través de la API de Windows, la cual puede ser instrumentalizada mediante las utilidades que ofrece Frida y, mediante el código de instrumentación diseñado en este trabajo, es posible modificar los parámetros de entrada y salida anulando de esta forma a las técnicas anti-análisis de esta agrupación.

Técnicas genéricas de detección de sandbox

Técnicas genéricas de detección de sandbox	Detección preherramienta	Detección postherramienta
Comprobando username		
Comprobando file path		
Comprobando nombres comunes de muestras en el drives root		
Comprobando si el tamaño de disco <= 60GB via DeviceIoControl()		
Comprobando si el tamaño de disco<= 60GB via GetDiskFreeSpaceExA()	X	
Comprobando si Sleep() esta parcheado usando GetTickCount()		
Comprobando si el numero de procesadores es < 2 vía acceso PEB		
Comprobando si el numero de procesadores es < 2 via GetSystemInfo()		
Comprobando si la memoria física es < 1Gb		
Comprobación del tiempo de actividad del sistema operativo usando GetTickCount()	X	
Comprobando si el sistema operativo IsNativeVhdBoot()		

Tabla 4.3: Técnicas genéricas de detección de sandbox.

En la Tabla 4.3 se pueden observar las técnicas de evasión que implementa Pafish para detectar *sandbox* genéricas. Estas técnicas, comprueban nombres de usuario y rutas empleadas de forma genérica por distintas *sandbox* comerciales, así como los recursos hardware de los que dispone el sistema. Inicialmente, el entorno virtual solo es detectado mediante el tamaño de disco y el tiempo que lleva activo el sistema. Para ello, las técnicas emplean de nuevo llamadas a las funciones de la API de Windows, lo que permite evadir la detección mediante la instrumentalización de dichas funciones y la modificación de sus valores a través del código de instrumentación. Como apunte, aunque el resto de técnicas no hayan detectado el entorno virtual debido a la configuración empleada en la creación de la máquina, todas las técnicas salvo el caso de la detección mediante el acceso al PEB que utiliza código ensamblador, podrían ser anuladas empleando el mismo sistema de instrumentalización.

Técnicas de detección basadas en la información de la CPU

La tabla 4.4 contiene las técnicas anti-análisis que realiza Pafish para detectar un entorno virtual basándose en la información que proporciona la CPU. A diferencia de las técnicas vistas con anterioridad, las cuales están basadas en llamadas a la API de Windows, estas técnicas basan su funcionamiento en la ejecución de un conjunto de instrucciones en lenguaje ensamblador el cual está embebido en un lenguaje de más alto nivel, en el caso de las pruebas realizadas este lenguaje de alto nivel es C. Esta implementación impide la instrumentalización empleada por la solución desarrollada debido a que la detección no recae en funciones externas que se puedan instrumentalizar, sino en el propio código del programa. Por todo esto, ninguna de las técnicas de este grupo ha podido ser evadida en las pruebas realizadas, comprobando la necesidad de emplear otro enfoque de instrumentalización o modificar la configuración de la CPU en la máquina virtual para poder inhibirlas.

Técnicas de detección basadas en la información de la CPU	Detección preherramienta	Detección postherramienta
Comprobando la diferencia entre los tiempos de la CPU(rdtsc)		
Comprobando la diferencia entre los tiempos de la CPU(rdtsc) forzando VM-Exit	X	X
Comprobando el bit del hipervisor usando cpuid	X	X
Comprobando el proveedor del hipervisor usando cpuid	X	X

Tabla 4.4: Técnicas de detección basadas en la información de la CPU

Pruebas genéricas de turing inversas

Pruebas genéricas de turing inversas	Detección preherramienta	Detección postherramienta
Comprobando la presencia del ratón		
Comprobando la existencia de movimiento del cursor	X	
Comprobando la velocidad del movimiento del cursor	X	
Comprobando la actividad de clicks del ratón	X	X
Comprobando la actividad de doble clicks en el ratón	X	X
Comprobando la actividad a través de una alerta de dialogo	X	X
Comprobando la actividad a través de una alerta de dialogo plausible	X	X

Tabla 4.5: Pruebas genéricas de turing inversas

Las pruebas de turing inversas implementadas por Pafish para detectar un entorno virtual se pueden observar en la Tabla 4.5. Este tipo de técnicas basan la detección del entorno virtual en la existencia de interacción humana con el sistema, en el caso de las

pruebas realizadas comprueban tanto el movimiento del ratón como su velocidad, la pulsación de *clicks* y la interacción del usuario con alertas generadas por las técnicas de evasión. Para hacer estas comprobaciones en el sistema, las técnicas hacen uso de las funciones de la API de Windows, pero a diferencia de los dos primeros casos planteados en esta sección, la necesidad de la intervención de un ser humano hace más complejo la creación de un código de instrumentación que sea capaz de inhibir las técnicas de evasión sin afectar a la carga maliciosa ejecutada por el *malware*. Esto, se puede ver reflejado en el número de técnicas de evasión inhibidas en la tabla, teniendo éxito únicamente en las que tienen relación con la posición y la velocidad del cursor. Por otro lado, en las pruebas realizadas se ha comprobado que las técnicas que dependen de la pulsación de *clicks* en el ratón se podrían anular mediante la instrumentalización de la función `SetWindowsHookEx()` para evitar que instale un procedimiento de enlace en `WH_MOUSE_LL` pero esto, afectaría a las funciones de la carga maliciosa del *malware* que empleen la instalación de este procedimiento para lograr su objetivo. Del mismo modo, las alertas de diálogo generadas podrían ser interceptadas mediante instrumentalización, pero debido a la necesidad de que la solución sea genérica sin enfocarse en un *malware* en concreto, hace imposible determinar si la alerta generada es parte de una técnica de evasión o de un mecanismo empleado por la carga maliciosa, lo que impide inhibir este tipo de técnicas.

Conclusión de las pruebas de concepto

La tabla 4.6 aporta una visión global de las técnicas de evasión implementadas por Pafish orientadas a detectar el software de virtualización VirtualBox así como, de su eficacia antes y después de la aplicación de la herramienta desarrollada. A partir de esta visión, es posible extraer la siguiente información:

- Hay un número de técnicas implementadas que debido a la configuración inicial de la máquina no son capaces de detectar el entorno virtual sin necesidad de aplicar una medida adicional.
- El porcentaje de éxito global de las técnicas de evasión sin aplicar la herramienta desarrollada es de $28/43 = 65{,}11\,\%$.
- El porcentaje de éxito global de las técnicas de evasión se reduce a $7/43 = 16{,}28\,\%$ cuando se aplica la solución desarrollada.

Agrupación de técnicas	Técnica	Detección preherramienta	Detección postherramienta
Técnicas de detección de depuradores	Usando IsDebuggerPresent()		
	Usando BeingDebugged vía acceso PEB		
Técnicas de detección basadas en la información de la CPU	Comprobando la diferencia entre los tiempos de la CPU(rdtsc)		
	Comprobando la diferencia entre los tiempos de la CPU(rdtsc) forzando VM-Exit	X	X
	Comprobando el bit del hipervisor usando cpuid	X	X
	Comprobando el proveedor del hipervisor usando cpuid	X	X
Pruebas genéricas de turing inversas	Comprobando la presencia del ratón		
	Comprobando la existencia de movimiento del cursor	X	
	Comprobando la velocidad del movimiento del cursor	X	
	Comprobando la actividad de clicks del ratón	X	X
	Comprobando la actividad de doble clicks en el raton	X	X
	Comprobando la actividad a través de una alerta de dialogo	X	X
	Comprobando la actividad a través de una alerta de dialogo plausible	X	X
Técnicas genéricas de detección de sandbox	Comprobando username		
	Comprobando file path		
	Comprobando nombres comunes de muestras en el drives root		
	Comprobando si el tamaño de disco <= 60GB vía DeviceIoControl()		
	Comprobando si el tamaño de disco<= 60GB vía GetDiskFreeSpaceExA()	X	
	Comprobando si Sleep() esta parcheado usando GetTickCount()		
	Comprobando si el numero de procesadores es <2 via acceso PEB		
	Comprobando si el numero de procesadores es <2 via GetSystemInfo()		
	Comprobando si la memoria física es <1Gb		
	Comprobación del tiempo de actividad del sistema operativo usando GetTickCount()	X	
	Comprobando si el sistema operativo IsNativeVhdBoot()		
Técnicas de detección de hooks	Comprobando la función ShellExecuteExW()		
	Comprobando la función CreateProcessA()		
Técnicas de detección de VirtualBox	Reg key Scsi port->bus->target id-> logical unit id->0 identifier	X	
	Reg key (HKLM\\HARDWARE\\Description \\System \"SystemBiosVersion\	X	
	Reg key (HKLM\\SOFTWARE\\Oracle \\VirtualBox Guest Additions	X	
	Reg key (HKLM\\HARDWARE\\Description \\System \"VideoBiosVersion\	X	
	Reg key (HKLM\\HARDWARE \\ACPI\\DSDT\\VBOX__	X	
	Reg key (HKLM\\HARDWARE \\ACPI\\FADT\\VBOX__	X	
	Reg key (HKLM\\HARDWARE \\ACPI\\RSDT\\VBOX__	X	
	Reg key (HKLM\\SYSTEM\\ControlSet001 \\Services\\VBox*	X	
	Reg key (HKLM\\HARDWARE\\DESCRIPTION \\System \"SystemBiosDate\	X	
	Driver files in C:\\WINDOWS\\system32 \\drivers\\VBox*	X	
	Archivos de sistema adicionales	X	
	Buscando una dirección MAC que empiece con 08:00:27	X	
	Buscando pseudo devices	X	
	Buscando VBoxTray windows	X	
	Buscando VBox network share	X	
	Buscando VBox processes	X	
	Buscando VBox devices usando WMI	X	

Tabla 4.6: Entorno virtual detectado por las técnicas de Pafish antes de la implementación de la herramientas y posterior a ella.

CAPÍTULO 5
Conclusiones y trabajos futuros

Las técnicas de evasión recopiladas en este trabajo muestran las dificultades a las que se enfrentan las técnicas de análisis de *malware* más utilizadas actualmente. Estos hechos evidencian la necesidad de introducir nuevas técnicas o crear nuevas herramientas que apoyen el análisis realizado por las técnicas ya existentes. Dentro de este contexto, la instrumentación dinámica de binarios ha demostrado ser una potente herramienta que permite baipasear un elevado número de técnicas de evasión analizadas en el presente trabajo, permitiendo de esta forma, aumentar el porcentaje de éxito de los análisis dinámicos realizados. A pesar de esto, existe un número reducido de técnicas que debido a su implementación no han podido ser instrumentalizadas por la solución propuesta ya que, según las pruebas realizadas, sería necesario instrumentalizar el programa entero y analizarlo instrucción a instrucción a nivel de código ensamblador. Por esta razón, sería necesario introducir junto a la herramienta desarrollada cambios en la configuración del entorno virtual como, por ejemplo, modificar el valor de `cpuid` manualmente en los archivos de configuración, tal y como permite la mayoría del software de virtualización.

Por otro lado, cabe destacar que la solución está basada en la implementación que Frida realiza de la aproximación dynamic probe injection, para la que usa principalmente la técnica *trampoline hooking*. Sin embargo, Frida también cuenta con una implementación que es capaz de crear una herramienta basada en dynamic binary translation, la cual tendría la capacidad de analizar cada instrucción en lenguaje ensamblador perteneciente al programa, por lo que sería interesante estudiarlo de cara a implementar otra posible solución al problema planteado que contará con un mayor porcentaje de técnicas de evasión baipaseadas.

En conclusión, queda demostrada la utilidad de la instrumentación dinámica de binarios en el campo del análisis de *malware*, pese a ello es necesario dedicar más tiempo de investigación para explorar todas las posibilidades que puede aportar. En esta misma línea se encuentra la solución expuesta en el presente documento, la cual se puede considerar la primera piedra de un trabajo en evolución.

5.1 Trabajos futuros

De cara a una continuación de este trabajo en el futuro, sería interesante cambiar el planteamiento de basar la solución en una sola herramienta y, estudiar la aplicación de otras medidas junto con la ya planteada como, por ejemplo, ampliar el número de técnicas de evasión inhibidas mediante la modificación de los archivos de configuración de los entornos virtuales. Tanto VirtualBox como VMware, siendo estos los softwares de virtualización más utilizados actualmente, permiten realizar cambios en la configuración de las máquinas virtuales. Tomando de ejemplo la necesidad de modificar los valores devuel-

tos por la instrucción `cpuid` cuando es ejecutada dentro de un entorno virtual, es posible realizar esto en VirtualBox mediante la ejecución del comando `VBoxManage modifyvm <uuid|vmname>-cpuid-set <leaf[:subleaf]><eax><ebx><ecx><edx>` con los valores deseados mientras que, para VMware sería necesario añadir las líneas del Listing 5.1 en el archivo VMX de la máquina.

```
cpuid.40000000.ecx="0000:0000:0000:0000:0000:0000:0000:0000"
cpuid.40000000.edx="0000:0000:0000:0000:0000:0000:0000:0000"
cpuid.1.ecx="0---:----:----:----:----:----:----:----"
```

Listing 5.1: Modificación de los valores devueltos por cpuid en VMware.

Otra posible continuación del trabajo pasaría por emplear lo aprendido para desarrollar una segunda solución basada en la implementación que Frida realiza de dynamic binary translation, la cual recibe el nombre de Stalker. A grandes rasgos, Stalker es un *code tracing engine* que permite seguir hilos y capturar cada función, bloque e instrucción que es ejecutada por un programa. Permitiendo con cada captura insertar, modificar o eliminar instrucciones a nivel de código ensamblador obteniendo, además, información sobre la instrucción previa y posterior a la que va a ser ejecutada. Esta información, ofrece un amplio contexto de la acción que puede estar desarrollando un *malware* en cada instante de su ejecución, y por tanto, actuar en consecuencia para inhibir una técnica de evasión o simplemente interpretar dicha información y mostrarla por pantalla para informar al analista.

Por último, la utilización de VirtualBox es una buena opción para analizar *malware* complejo de forma exhaustiva interaccionando directamente con él. Pero, sería interesante ampliar la solución con el objetivo de que se pueda implantar en un sistema automatizado de análisis de *malware*. La principal ventaja con la que cuentan estos entornos es que son capaces de recibir archivos sospechosos y en cuestión de poco tiempo, proporcionar un informe detallado que describe el comportamiento del archivo en base a su ejecución dentro de un entorno virtual aislado. De esta forma, una persona sin conocimiento específico de las herramientas empleadas en el análisis de *malware*, pero que sea capaz de interpretar los datos, podría reconocer las acciones que efectúa un programa malicioso cuando se encuentra en ejecución. Esto, posibilita aumentar el número de *malware* analizado en una cantidad determinada de tiempo además de, realizar la función de filtrado ante los casos más difíciles que necesiten de las herramientas convencionales utilizadas por los analistas de *malware*.

Bibliografía

1. SIHWAIL, Rami; OMAR, Khairuddin y ARIFFIN, KA Zainol. A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *Int. J. Adv. Sci. Eng. Inf. Technol*. 2018, vol. 8, n.º 4-2, págs. 1662-1671.
2. YOUSUF, Muhammad Irfan; ANWER, Izza; RIASAT, Ayesha; ZIA, Khawaja Tahir y KIM, Suhyun. Windows malware detection based on static analysis with multiple features. *PeerJ Computer Science*. 2023, vol. 9, e1319.
3. OR-MEIR, Ori; NISSIM, Nir; ELOVICI, Yuval y ROKACH, Lior. Dynamic malware analysis in the modern era—A state of the art survey. *ACM Computing Surveys (CSUR)*. 2019, vol. 52, n.º 5, págs. 1-48.
4. ENISA. *ENISA Threat Landscape 2022* [online]. [visitado 2023-08-07]. Disp. desde: https://www.enisa.europa.eu/publications/enisa-threat-landscape-2022.
5. SEGURIDAD NACIONAL DEL GABINETE DE LA PRESIDENCIA DEL GOBIERNO, Departamento de. *Informe Anual de Seguridad Nacional 2022* [online]. [visitado 2023-08-07]. Disp. desde: https://www.dsn.gob.es/es/documento/informe-anual-seguridad-nacional-2022.
6. AKHTAR, Muhammad Shoaib y FENG, Tao. Malware Analysis and Detection Using Machine Learning Algorithms. *Symmetry*. 2022, vol. 14, n.º 11, pág. 2304.
7. GRŽINIĆ, Toni y GONZÁLEZ, Eduardo Blázquez. Methods for automatic malware analysis and classification: a survey. *International Journal of Information and Computer Security*. 2022, vol. 17, n.º 1-2, págs. 179-203.
8. IJAZ, Muhammad; DURAD, Muhammad Hanif e ISMAIL, Maliha. Static and dynamic malware analysis using machine learning. En: *2019 16th International bhurban conference on applied sciences and technology (IBCAST)*. IEEE, 2019, págs. 687-691.
9. KIM, Minho; CHO, Haehyun y YI, Jeong Hyun. Large-Scale Analysis on Anti-Analysis Techniques in Real-World Malware. *IEEE Access*. 2022, vol. 10, págs. 75802-75815.
10. OLAIMAT, Mohammad N; MAAROF, Mohd Aizaini y AL-RIMY, Bander Ali S. Ransomware anti-analysis and evasion techniques: A survey and research directions. En: *2021 3rd international cyber resilience conference (CRC)*. IEEE, 2021, págs. 1-6.
11. BIONDI, Fabrizio; GIVEN-WILSON, Thomas; LEGAY, Axel; PUODZIUS, Cassius y QUILBEUF, Jean. Tutorial: An overview of malware detection and evasion techniques. En: *Leveraging Applications of Formal Methods, Verification and Validation. Modeling: 8th International Symposium, ISoLA 2018, Limassol, Cyprus, November 5-9, 2018, Proceedings, Part I 8*. Springer, 2018, págs. 565-586.
12. OYAMA, Yoshihiro. Trends of anti-analysis operations of malwares observed in API call logs. *Journal of Computer Virology and Hacking Techniques*. 2018, vol. 14, n.º 1, págs. 69-85.

13. YOU, Ilsun y YIM, Kangbin. Malware obfuscation techniques: A brief survey. En: *2010 International conference on broadband, wireless computing, communication and applications*. IEEE, 2010, págs. 297-300.

14. SINGH, Jagsir y SINGH, Jaswinder. Challenge of malware analysis: malware obfuscation techniques. *International Journal of Information Security Science*. 2018, vol. 7, n.º 3, págs. 100-110.

15. AFIANIAN, Amir; NIKSEFAT, Salman; SADEGHIYAN, Babak y BAPTISTE, David. Malware dynamic analysis evasion techniques: A survey. *ACM Computing Surveys (CSUR)*. 2019, vol. 52, n.º 6, págs. 1-28.

16. BULAZEL, Alexei y YENER, Bülent. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. En: *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium*. 2017, págs. 1-21.

17. VEERAPPAN, Chandra Sekar; KEONG, Peter Loh Kok; TANG, Zhaohui y TAN, Forest. Taxonomy on malware evasion countermeasures techniques. En: *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*. IEEE, 2018, págs. 558-563.

18. KIM, Jong-Wouk; BANG, Jiwon y CHOI, Mi-Jung. Defeating anti-debugging techniques for malware analysis using a debugger. *Advances in Science, Technology and Engineering Systems Journal*. 2020, vol. 5, n.º 6, págs. 1178-1189.

19. NEP, Pham Ri y CAM, Nguyen Tan. A Research on Countering Virtual Machine Evasion Techniques of Malware in Dynamic Analysis. En: *International Conference on Intelligent Computing & Optimization*. Springer, 2022, págs. 585-596.

20. PRIYADARSHAN, Soumyakant. A study of Binary Instrumentation techniques. 2019.

21. ZHAO, Valerie. Evaluation of dynamic binary instrumentation approaches: Dynamic binary translation vs. dynamic probe injection. 2018.

22. LEE, Young Bi; SUK, Jae Hyuk y LEE, Dong Hoon. Bypassing anti-analysis of commercial protector methods using DBI tools. *IEEE Access*. 2021, vol. 9, págs. 7655-7673.

23. INTEL. *Pin - User's Manual* [online]. [visitado 2023-08-07]. Disp. desde: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html.

24. DEVELOPERS, The Frida. *Frida Documentation* [online]. [visitado 2023-08-01]. Disp. desde: https://frida.re/docs/home/.

25. COHN, Robert. *Pin Tutorial* [online]. [visitado 2023-08-07]. Disp. desde: https://www.intel.com/content/dam/develop/external/us/en/documents/pintutorial-academiasinica-1.ppt.

APÉNDICE A

Anexo I: OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**			X	
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**			X	
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**			X	
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**	X			
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

A continuación, se va a exponer un listado de los objetivos de desarrollo sostenible que tienen relación con el presente trabajo de final master junto, con una reflexión que aporta los motivos de dicha relación:

- **ODS 3. Salud y bienestar:** Actualmente, un gran número de grupos de ciberdelincuentes están tomando como objetivo de sus ataques a hospitales y otros organismos relacionados con la salud. Dado que el trabajo realizado busca apoyar el análisis de los programas empleados por los atacantes, reduce el tiempo que los servicios se ven interrumpidos, así como ayuda a crear nuevos mecanismos para defender este sector.
- **ODS 6. Agua limpia y saneamiento:** Existen precedentes de sabotajes al suministro de agua mediante ataques cibernéticos. Al igual que en el caso anteriormente expuesto, el trabajo realizado puede ayudar a que el suministro de agua sea restablecido en un menor tiempo y posteriormente, crear mecanismos para su defensa a través de la identificación de las técnicas empleadas por los programas maliciosos.
- **ODS 8. Trabajo decente y crecimiento económico:** Los ataques cibernéticos destinados a cortar la cadena de suministro de las empresas están al alza, llegando incluso algunos de ellos a provocar el cierre de la empresa con la destrucción de puestos de trabajo que ello conlleva. Por ello, es necesario aumentar los mecanismos de defensa de cara a evitar estas situaciones, lo cual es uno de los objetivos de la herramienta desarrollada.
- **ODS 9. Industria, innovación e infraestructuras:** Tanto la industria como las infraestructuras consideradas críticas, son objetivo de numerosos intentos de ciberataques cada año. La necesidad de proteger ambos sectores se ve reflejada en el Esquema Nacional de Seguridad. A través del análisis de las técnicas que emplean los atacantes es posible crear métodos de seguridad específicos para este sector.
- **ODS 16. Paz, justicia e instituciones sólidas:** La identificación de ciertos componentes dentro de los programas maliciosos, puede llevar a revelar la autoría de los atacantes, pudiendo así aplicar justicia sobre los hechos realizados y favoreciendo un mundo más pacífico.