Universidad Autónoma Metropolitana

Unidad Azcapotzalco

División de Ciencias Básicas e Ingeniería

Licenciatura en Ingeniería en Computación

Proyecto Tecnológico

# Análisis para la ayuda de la detección de Malware

Alumno: Bracho Garnica Jose Alberto

Matricula: 210301832

Correo Electrónico: al210301832@alumnos.azc.uam.mx

Asesor: M. en C. Arturo Zúñiga López

Correo Electrónico: azl@correo.azc.uam.mx

2018 - Primavera

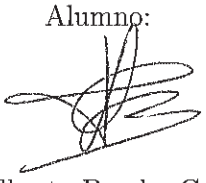
5 de Septiembre de 2018

# Declaratoria

Yo, M. en C. Arturo Zúñiga López, declaro que apróbe el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Yo, Jose Alberto Bracho Garnica, doy mi autorización a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital,así como en el Repositorio Institucional de UAM Azcapotzalco.

Alumno:



Jose Alberto Bracho Garnica

Asesor:

M. en C. Arturo Zúñiga López

# Resumen

Los *Malware* han sido las principales amenazas cibernéticas a nivel mundial de tal modo que no se pueda garantizar la seguridad al momento de navegar por Internet. A pesar de que existen un conjunto de programas antivirus que pueden garantizar que las computadoras estén libres de un ataque estos no son lo suficientemente potentes para eliminar los *Malware*, sobretodo porque los atacantes mejoran las características de los ejecutables para pasar desapercibidos por los antivirus. Muchos han logrado realizar clasificadores y herramientas que logran detectar cuando un archivo *PE* o ejecutable contiene *Malware* con resultados favorables, el único problema es que hacen uso de varias herramientas de detección que cumplen una única función causando que dichos clasificadores se vuelvan muy complejos lo que genera recursos. En este proyecto se logró identificar, en un único algoritmo de programación en *Python*, si un archivo PE o ejecutable es sospechoso de contener *Malware* o no realizando un análisis estático de 5 características de un archivo *PE* que nos indica dicha sospecha, además que muestra las llamada *API* y *URL* que realiza para su ejecución y muestra en código ensamblador el comportamiento del archivo *PE* con el fin de generar un informe para ayudar a elaborar mejores antivirus. Los resultados finales del proceso de análisis del algoritmo se encuentran dentro de lo aceptable debido que detecto correctamente de 49 archivos PE con *Malware* el 85.71 % y el 14.28 % restante detecto falsos negativos. Sin embargo el algoritmo también presenta limitantes debido a que solo pudo detectar de 50 archivos PE limpios el 72 % y el 28 % como falsos positivos, esto quiere decir que el algoritmo es útil para brindar información en la detección de *Malware* e información de las llamadas *API* y *URL* que el *Malware* solicita.

# Índice general

**Página**

# Índice de Figuras

# Índice de Tablas

**Tabla** **Página**

# Capítulo 1

# Introducción

Una de las mayores amenazas de seguridad en Internet es el *Malware* o programas maliciosos. *Malware* es un término colectivo usado para todo tipo de amenazas incluyendo virus, gusanos y troyanos. De acuerdo con las estadísticas de *Kaspersly Lab*, los usuarios en América Latina han recibido un total de 677,216,773 ataques de *Malware* durante los ocho primeros meses (1ero de enero a 31 de agosto) del año 2017 [1]

De acuerdo con el reporte del primer cuatrimestre del 2014 de los laboratorios de *Panda Security*[2], el 71.85 % de los programas descargados contienen *Malware*, afectando empresas u organizaciones así como países en todo el mundo, y se encuentran disfrazados en aplicaciones útiles y seguras.

La amenaza masiva que representa el *Malware* requiere de medidas de seguridad, por ejemplo, tener antivirus actualizados y mantener una alta precisión de detección para los*Malware* modernos. El problema es que el *Malware* está evolucionando con el tiempo, y además, los sistemas de detección a veces no se están actualizando, siendo estos inútiles para la detección y la eliminación de dichos programas, por lo que hace falta tener sistemas de detección más avanzados que puedan identificarlos sin complicaciones con el fin de obtener información del *Malware*.

El *Malware* tiene como objetivo infiltrarse en el sistema y dañar la computadora sin el consentimiento de su dueño, con distintas finalidades. En sus inicios los virus informáticos eran prácticamente la única forma de *Malware*. Actualmente, esta amenaza creció y ahora incluye otros vectores de infección y técnicas muy diversas de propagación, por ejemplo, el *Malware* puede comportarse como un *"browser exploits"*, es decir, aprovecha las fallas o vulnerabilidades de los sistemas operativos, enlaces punto a punto (*P2P*) y redes de datos. Hoy en día otras formas de *Malware* incluyen a los troyanos, gusanos (*worms*) que se propagan por la red de datos, agentes de *DDos*, *bots* controlados por canales *IRC*, *Spyware*, etc.

Para entender la amenaza que representa el *Malware*, es necesario conocer sus tipos y cómo funcionan. En general se puede dividir el *Malware* en las siguientes clases:

a) Virus clásicos

Programas que infectan a otros programas por añadir su código para tomar el control después de ejecución de los archivos infectados. El objetivo principal de un virus es infectar. La velocidad de propagación de los virus es algo menor que la de los gusanos.

b) Gusanos de red

Este tipo de *Malware* usa los recursos de red para distribuirse implicando que pueden penetrar de un equipo a otro como un gusano. Esto lo hacen por medio de correo electrónico, sistemas de mensajes instantáneos, redes de archivos compartidos, canales *IRC*, redes globales, etc. Su velocidad de propagación es muy alta y cuando logra entrar en el equipo, el gusano intenta obtener las direcciones de otros equipos en la red para empezar a enviarles sus copias. También suelen usar los datos del libro de contactos del cliente de correo electrónico. La mayoría de los gusanos se propagan en forma de archivos pero existe una pequeña cantidad de gusanos que se propagan en forma de paquetes de red y penetran directamente la memoria RAM del equipo víctima, donde ejecutan su código.

c) Caballos de Troya o Troyanos

Esta clase de programas maliciosos incluye una gran variedad de programas que efectúan acciones sin que el usuario se dé cuenta, y sin su consentimiento, recolectan datos y los envían a los criminales, destruyen o alteran datos con intenciones delictivas, causando desperfectos en el funcionamiento de la computadora o usan los recursos de la misma para fines criminales, como hacer envíos masivos de correo no solicitado. Ya que no infectan a otros programas o datos se suele decir que no corresponden a virus clásicos. Los troyanos no pueden penetrar a los equipos por sí mismo, sino se propagan por los criminales como un software “deseable o necesario” que es capaz de causar mucho más daño que los virus clásicos.

d) *Spyware*

*Software* que permite colectar la información sobre un usuario/organización de forma no autorizada y su presencia puede ser completamente invisible para el usuario. Pueden colectar los datos sobre las acciones del usuario, el contenido del disco duro, software instalado, calidad y velocidad de la conexión, etc. Pero no es su única función. Tambien son conocidos por lo menos dos programas (*Gator* y *eZula*) que permiten también controlar el equipo. Otro ejemplo de programas espías son los programas que instalan su código en el navegador de Internet para redireccionar el tráfico. Este tipo de *Malware* es el más conocido por los usuarios que navegan en Internet y se puede identificar al momento en que el navegador abre una página que no fue requerida por el usuario.

e) *Phishing*

Es una variedad de programas espías que se propaga a través de correo. Pretenden recibir los datos confidenciales del usuario, de carácter bancario preferente. Los *emails phishing* están diseñadas para parecer igual a la correspondencia legal enviada por organizaciones bancarias u otros mensajes importantes. Tales *emails* contienen un enlace que redirecciona al usuario a una página falsa que va a solicitar ingresar algunos datos confidenciales, como el número de la tarjeta de crédito.

f) *Adware*

Muestran publicidad al usuario en la interfaz. La mayoría de programas *adware* aparentan ser *softwares* efectivos y gratuitos. A veces pueden colectar y enviar los datos personales del usuario.

g) *Riskware*.

No son programas maliciosos pero contienen una amenaza potencial y en ciertas situaciones pone su información en riesgo. Incluyen programas de administración remota, marcadores, etc.

h) *Rootkits*

Un *rootkit* es una colección de programas usados por un *hacker* para evitar ser detectado mientras busca obtener acceso no autorizado a un ordenador. Esto se logra de dos formas: reemplazando archivos o bibliotecas del sistema o instalando un módulo de kernel. El *hacker* instala el *rootkit* después, obteniendo un acceso similar al del usuario, por lo general estos accesos pueden ser craqueando una contraseña o explotando una vulnerabilidad, lo que permite usar otras credenciales hasta conseguir el acceso administrativo.

i)*Spam*

Los mensajes no solicitados de un remitente desconocido enviados en cantidades masivas de carácter publicitario, político, de propaganda, solicitando ayuda, etc. Otra clase de *spam* hacen las propuestas relacionadas con varias operaciones ilegales con dinero o participación en algún negocio. También hay *emails* dedicados al robo de contraseñas o números de tarjetas de crédito, cartas de cadena, etc. El *Spam* genera una carga adicional a los servidores de correo y puede causar pérdidas de la información deseada.

El objetivo del análisis de *Malware* es poder determinar y comprender como funciona una determinada pieza de *Malware*, con el fin de realizar las medidas que permitan protegernos de esta. Hay dos preguntas fundamentales que se busca responder cuando se realiza un análisis de una determinada pieza de *Malware* en una computadora infectada por uno mismo. Primero, la razón por la cual la computadora fue infectada por la pieza de *Malware*. En segundo lugar, que es lo que hace exactamente este *Malware*.

Existen dos formas de analizar *Malware*: análisis estático y análisis dinámico.

a) Análisis Estático

Se realiza revisando los componentes del archivo binario sin la necesidad de ejecutar el mismo y estudiando cada componente. El archivo puede desmontarse usando un desensamblador como *IDA* [3]. El código máquina puede traducirse a código ensamblador para que los analistas puedan leer y comprender dicho código. La mayoría de los *Malware* modernos contiene técnicas evasivas para vencer este tipo de análisis, un claro ejemplo es incorporar errores de código sintáctico que confundirán a los desensambladores, pero que seguirán funcionando durante la ejecución real.

b) Análisis Dinámico

Se realiza observando el comportamiento del *Malware* mientras se está ejecutando en una computadora. Para que dicho análisis sea seguro es necesario realizarlo en un entorno protegido para evitar que el *Malware* infecte los sistemas de producción. El entorno protegido consiste en trabajar con máquinas virtuales que impidan que el *Malware* se propague a otras computadoras. El *Malware* moderno puede contener una gran variedad de técnicas evasivas, por ejemplo: el *Malware* puede saber que se está ejecutando en una maquina virtual, diseñadas para vencer el análisis dinámico que incluye pruebas para entornos virtuales o depuradores activos, retrasar la ejecución de cargas maliciosas o requerir alguna forma de entrada interactiva por parte del usuario. Hasta la fecha, existen diferentes formas de detección de *Malware* utilizando las siguientes herramientas:

- Antivirus: Detección de *Malware* con base en firmas.
- Búsqueda de cadenas: Utilidad para observar cadenas *ASCII* que se encuentren en el código.
- Explorador *PE*[1]: Detecta *Malware* escaneando archivos con el estándar *PE* (archivos ejecutables y de procesamiento).
- Ejecutables Portables (*PE*): Es un formato de archivo estándar que involucra archivos ejecutables de *Windows* (*.exe*) y sus librerías dinámicas (*.dll*). De acuerdo con el reporte de *Symantec Internet Security*[4] más del 50 % de archivos en Internet tienen contenido ejecutable, que no deberían tener, dando como resultado de un gran número de ataques.

El proyecto consistirá en realizar un programa computacional que ayude a reunir información de archivos que contengan *Malware*, para determinar si contienen archivos ejecutables (*.exe* ó *.msi*), librerías *.dll*(estos archivos son muy importantes para el sistema operativo *Windows*) y algún tipo de información que comprometa la seguridad de una computadora. Además, este proyecto utilizara el análisis estático debido a que con este método no será necesario ejecutar el *Malware* para su análisis haciendo su manejo más seguro, con el fin de reportar dicha información y brindar una ayuda para el diseño de antivirus modernos.

# Capítulo 2

# Antecedentes

Actualmente, existen varias organizaciones, universidades, entre otros, que han desarrollado sistemas de clasificación de *Malware*, hasta programas que además de clasificar, le dan solución al mismo que se tenga en una computadora y en dispositivos móviles.

Un ejemplo de un *software*fue el que se desarrolló en la Universidad de Icesi por el estudiante Christian Uscoque llamado *"Safe Candy"*.Este *software*es un sistema que analiza y detecta aplicaciones maliciosas para los sistemas móviles, específicamente aquellos que tienen el sistema operativo *Android*, utilizando técnicas de inteligencia artificial y algoritmos de "Aprendizaje Profundo" con resultados efectivos[5].

Los Investigadores del Instituto Politécnico Nacional (IPN) desarrollaron un servicio web llamado *"Garmdroid"*, el cual detecta aplicaciones que contienen código malicioso. El servicio web tiene una efectividad del 96.26 % y utiliza tecnología basado en "Aprendizaje Profundo" para analizar archivos de formato *APK* contenida en las aplicaciones *Android*, además de que muestra al usuario los componentes que tiene la aplicación analizada y los componentes de hardware a los cuales dicha aplicación da acceso, como cámara, micrófono, etc[6].

Otro programa que se puede mencionar es el creado por la egresada de la universidad de Sevilla, María Valero Campaña, el cual consiste en un sistema que detecta los archivos o mensajes que entran en una red local, utilizando un sistema de almacenamiento (rb-S3), analizadores de trafico que colocó en distintos puntos de la red y una aplicación llamada *"rb-sequence-oozie"* la cual tomará los archivos y los enviará a un sistema de detección que hará uso de una serie de *frameworks* de análisis de ficheros, los cuales son: *YARA*, Virus total, Metas can, Clama, *Kaspersky* y *Cuckoo* [7].

El estudio y la práctica de diferentes técnicas de detección de *Malware* han llamado la atención en las universidades, tal es el caso de los estudiantes de la UNAM que presentaron una tesis explicando la construcción de un laboratorio de análisis de *Malware*, utilizando metodologías de análisis forense hacia equipos que tienen sistema operativo *Windows*, utilizando herramientas constituidas por canales de comunicación, accesos remotos y administración de *firewall* [8].

Otro ejemplo de aplicación para detectar Malware es el desarrollo de una herramienta llamada *"TTAnalyse"* para analizar ejecutables con *Malware* dinámicamente. Se crea un entorno basado en la emulación para revisar binarios. Estos binarios se ejecutan por medio de llamadas *API* y a otras funciones. Debido al entorno emulado, la metodología es invisible para el código de *Malware*. Ofrece un análisis rápido de un *Malware* desconocido, pero solo puede funcionar en una sola ruta de ejecución [9].

# Capítulo 3

# Justificación

En la actualidad, existe un gran número de *Malware* y los antivirus no tienen la capacidad de detectarlos por su forma de trabajar basado en firmas. Además, los autores del *Malware* mejoran las características de dichos programas dificultando así su detección. Aparte de la complejidad, para la detección a veces es necesario utilizar varias herramientas las cuales cada una cumple una función diferente, por ejemplo, *TriID utlity* [10], *UPX* [11], *Strings* [12], etc., y en conjunto realizan la detección pero no es practico utilizar distinto software para determinar la detección de *Malware*. Por lo tanto, el hecho de tener un único programa computacional que contenga las funciones de cada una de las herramientas y elabore un informe de las características y del comportamiento de un archivo o programa que contenga *Malware* será muy útil para tener información del comportamiento del software malicioso.

Con este proyecto se obtendrá un programa computacional que elabore un informe de la detección de los archivos que contienen *Malware*, reportando sus características y su comportamiento.

# Capítulo 4

# Objetivo General

Diseñar un programa computacional para detectar archivos o programas con contenido *Malware*.

## Objetivos Particulares

- Diseñar un módulo de Análisis PE para detectar la posibilidad de que un archivo examinado contenga otros archivos en su interior con el estándar PE.
- Diseñar un módulo donde se desempaquete los archivos con formato PE e indique en que parte del código del archivo se encontró *Malware* para posteriormente extraer dicho código conformado por cadenas *ASCII*.
- Diseñar un módulo que interprete las cadenas *ASCII* y los decodifique en caso de ser necesario.
- Diseñar un módulo que realice un informe de todas las características de las cadenas extraídas previamente.

# Capítulo 5

# Marco Teórico

Para el desarrollo de este proyecto se utilizó una máquina virtual con el sistema operativo *Linux*. Una máquina virtual no es más que un *software* capaz de cargar en su interior otro sistema operativo con el fin de hacer creer que es una computadora real.

Un punto importante a explicar es que existen dos tipos de máquinas virtuales: máquina virtual del sistema y máquinas virtuales de proceso.

Una máquina virtual de sistema es aquella que emula un CPU completo. En resumen es un *software* que puede hacerse pasar por otra computadora de tal modo que se puede ejecutar otro sistema operativo en su interior. Estas máquinas virtuales cuentan con su propio disco duro, memoria, tarjeta gráfica y demás componentes de textslhardware, con la excepción de que todo es virtual. Una máquina virtual por sí misma no puede leer o acceder a los archivos que se encuentran en la maquina anfitrión a pesar de que está funcionando en la misma, estas se encuentran funcionando de manera independiente.

Una máquina virtual de proceso es menos compleja que una máquina de sistema. Esta máquina ejecuta un proceso específico, como una aplicación basada en *Java* o en *.NET Framework*, en lugar de emular una *PC* por lo que su uso se limita exclusivamente la ejecución de aplicaciones.

## 5.1. Python

*Python* es un lenguaje de programación poderoso y fácil de aprender. Cuenta con estructuras de datos eficientes y de alto nivel y un enfoque simple pero efectivo a la programación orientada a objetos. La sintaxis de *Python* y su capacidad de que una misma variable puede tomar valores de distinto tipo en distintos momentos el cual se le conoce como “Tipado Dinámico”, junto con su naturaleza interpretada, hacen de este un lenguaje ideal para desarrollo rápido de aplicaciones en diversas áreas y sobre la mayoría de las plataformas [13].

El proyecto hace uso del lenguaje de programación *Python* debido a que maneja un conjunto de librerías que facilitan la lectura de un archivo con formato PE, contiene librerías las cuales son necesarios para obtener las características de un archivo malicioso, para desencriptar cadenas y acomodar la información con el fin de que se entienda, entre otros.

## 5.2. Archivo Portable Ejecutable (PE)

Los archivos ejecutables originalmente llamados Portable Ejecutable (PE) son programas los cuales se ejecutan o corren en sistemas operativos *Windows*. Estos archivos están formados de manera estructurada para una mayor flexibilidad y manejo. El archivo se divide en cabeceras y cuerpo. Las cabeceras proveen información al cargador de *Windows* para su correcta ejecución. Posterior a estas cabeceras se encuentra el cuerpo del archivo la cual contiene cada una de las secciones que conforma el archivo con sus componentes correspondientes.

La estructura que tiene el archivo PE es el mostrado en la figura 5.1:

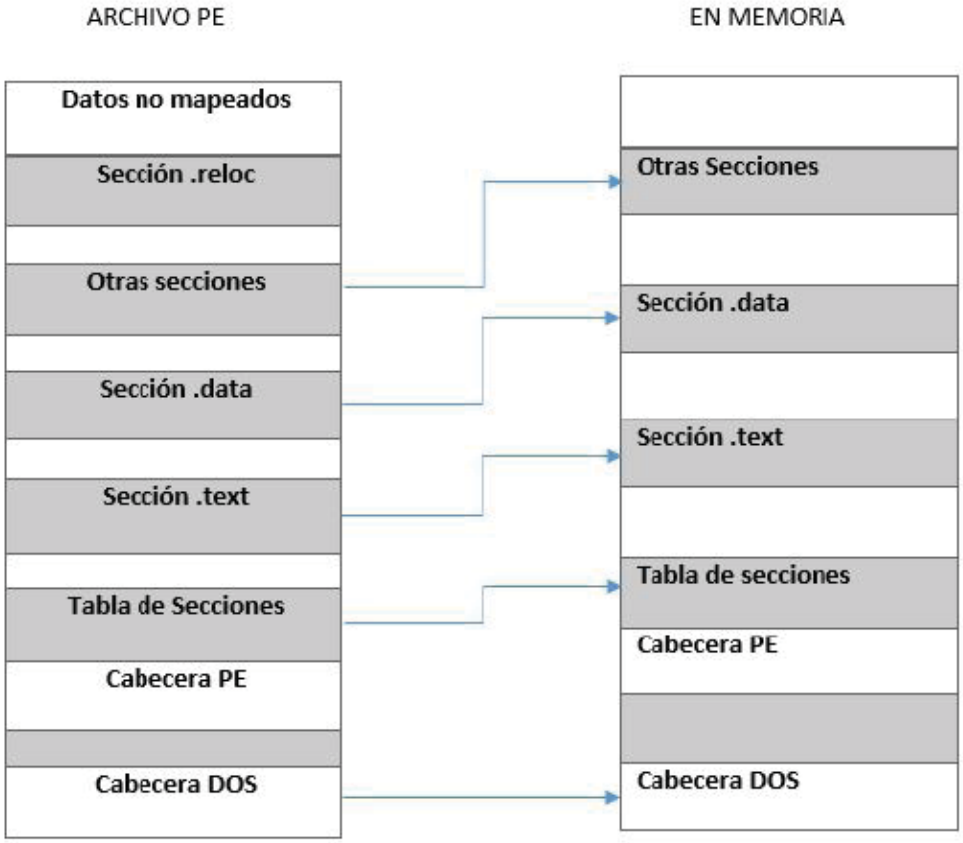



Figura 5.1: Modelo Machine Learning

- *DOS Header*

  *DOS* corresponde a las siglas en ingles *Disk Operating System* y representa a una familia de sistemas operativos de disco para las computadoras de Escritorio siendo este el primer sistema operativo creado en 1981. *DOS* cuenta con una interfaz de línea de comandos en modo texto o alfanumérico y este se encuentra en todas las computadoras actualmente.

  Esta cabecera que tienen los archivos PE tiene dos campos útiles:

  a) *e_magic*: este campo contiene las siglas 'MZ', en honor a Mark Zbikowky uno de los principales diseñadores de MS-DOS, que todo archivo ejecutable debe tener para que pueda ser ejecutado, es decir, que actúa como una firma de validación.

  b) *l_fanew*: es un apuntador hacia la cabecera *NT* el cual comienza por la firma PE00. Dicho valor suele encontrarse en el desplazamiento 0x3C

- *Dos-stub*

  Es una aplicación válida para *MS-DOS* la cual imprime en pantalla el mensaje *"This program cannot be run in DOS mode"*, esto quiere decir que indica que el archivo PE actualmente escaneado no puede ser ejecutado desde la línea de comandos de *MS-DOS* y existe en todos los archivos PE ejecutables.

  Las cabeceras *Dos-stub* y *Dos Header* nos indican si el archivo PE actualmente analizado también se puede ejecutar en la línea de comandos *DOS* o es exclusivo para *Windows*.

- Cabecera *NT*

  Esta cabecera inicia en el valor apuntado por el campo *I_fanew* debido a que en el inicio se encuentra una firma formada por los caracteres PEx0x0 que ubica la estructura del *IMAGE_FILE_HEADER*. La cabecera NT se encuentra tanto en archivos ejecutables como en librerías de enlace dinámico. En el caso de que el archivo PE no tenga dicha firma, el archivo no podrá ejecutarse al igual que con la firma MZ

- *FILE_HEADER*

  Esta cabecera contiene un conjunto de campos los cuales proporcionan información general acerca del archivo PE. Estos campos son los siguientes:

  - *Machine:* este campo define el tipo de arquitectura de la computadora que necesita para que el archivo PE se ejecute, dichos valores son los siguientes:
    - 0x14C: Este valor quiere decir que necesita de sistemas operativos *Windows* con procesador x86, es decir, para *Windows* de 32 bits.
    - 0x200: este valor quiere decir que necesita de sistemas operativos con *Intel Itanium* [14]
    - 0x8664: Este valor quiere decir que necesita de sistemas operativos Windows con procesador x64, es decir, para Windows de 64 bits

- *NumberOfSections:* Este campo contiene el número de todas las secciones que tiene el archivo PE. Existe un límite de 96 secciones que se pueden almacenar en conjunto dentro del archivo PE.
- *TimeDateStamp:* Indica el número de segundos a partir del 1 de Enero de 1970. Estos segundos nos indicaran la fecha de creación del archivo. Se toma como referencia la fecha antes mencionada debido a que fue el momento en que existieron los primeros archivos ejecutables.
- *PointerToSymbolTAble* y *NumberOfSymbols*: estos campos son empleados por los archivos .obj o archivos *COFF*. El valor que deben tener estos campos es 0.
- *SizeOfOptionalHeader:* este campo es de 2 bytes y contiene el valor del tamaño del *IMAGE_OPTIONAL_HEADER*.
- *Characteristics:* Este campo indica los distintos atributos del archivo mediante distintos valores. En el momento en que el archivo presenta varias características, el valor final se obtiene por medio de una suma de cada característica de acuerdo a la tabla 5.1:

| valores de las características | |
|---|---|
| Constante | Valor |
| IMAGE_FILE_RELOCS_STRIPPED | 0x0001 |
| IMAGE_FILE_EXECUTABLE_IMAGE | 0x0002 |
| IMAGE_FILE_LINE_NUMS_STRIPPED | 0x0004 |
| IMAGE_FILE_LOCAL_SYMS_STRIPPED | 0x0008 |
| IMAGE_FILE_AGGRESSIVE_WS_TRIM | 0x0010 |
| IMAGE_FILE_LARGE_ADDRESS_AWARE | 0x0020 |
| Unknown | 0x0040 |
| IMAGE_FILE_BYTES_REVERSED_LO | 0x0080 |
| IMAGE_FILE_32BIT_MACHINE | 0x0100 |
| IMAGE_FILE_DEBUG_STRIPPED | 0x0200 |
| IMAGE_FILE_REMOVABLE_RUN_FROM_SWAP | 0x040 |
| IMAGE_FILE_NET_RUN_FROM_SWAP | 0x080 |
| IMAGE_FILE_SYSTEM | 0x100 |
| IMAGE_FILE_DLL | 0x200 |

Tabla 5.1: Valores hexadecimales de las diferentes características de un archivo PE.

Por ejemplo, observando los valores del cuadro 5.1, si visualizamos el dato *Characteristics* del archivo PE que se está analizando y obtenemos el valor 0x0142 significa que las banderas que se encuentran habilitadas en el archivo PE son *Unknown* (0x0040), *IMAGE_FILE_32BIT_MACHINE* (0x0100), e *IMAGE_FILE_EXECUTABLE_IMAGE* (0X0002).

- *OPTIONAL_HEADER*

La cabecera opcional *(OPTIONAL_HEADER)* contiene información adicional para que el ejecutable se cargue de manera correcta. Esta información en los archivos ejecutables es necesaria y su tamaño es definido en el campo *SizeOfOptionalHeader* contenida como un parámetro más dentro del archivo PE. Esta cabecera está conformado por varios componentes importantes las cuales son:

- *Magic*: Este campo determina si el ejecutable analizado es para sistemas x86 o x64 en base a los siguientes valores:
  - 0x10b: El archivo se ejecuta en sistemas operativos *Windows* x86
  - 0x20b: El archivo se ejecuta en sistemas operativos *Windows* x64
- *MajorLinkerVersion:* Este campo proporciona la versión más alta del enlazador.
- *MinorLinkerVersion:* Este campo proporciona la versión más baja del enlazador.
- *SizeOfInitializedData* y *SizeOfUnitializedData*: Estos 3 campos están relacionados en cuanto a cómo obtener su valor y el único cambio es la procedencia de cada uno. Al igual que el ejecutable, sus secciones también presentan características y para relacionarlas con estos tres campos las características serian:

  - Valor: 0x00000020 = *IMAGE_SCN_CNT_CODE*
  - Valor: 0x00000040 = *IMAGE_SCN_CNT_INITIALIZED_DATA*
  - Valor: 0x00000080 = IMAGE_SCN_UNITIALIZED_DATA

- *AddressOfEntryPoint:* este campo nos proporciona la dirección virtual relativa (*RVA*) [15] hacia el punto de partida de ejecución del programa. Normalmente tiene su valor sumado al parámetro *ImageBase* el cual es 0x00400000.
- *BaseOfCode* y *BaseOfData*: Estos dos campos proporciona un *RVA* el cual indica o corresponde al inicio de las secciones de código y datos respectivamente.
- *ImageBase:* Este campo proporciona la dirección de preferencia donde se cargara el ejecutable. Este campo debe ser múltiplo de 0x10000. En los ejecutables es muy común ver este campo con el valor 0x10000000.
- *SectionAlignment:* Este campo proporciona un valor el cual será el alineamiento en bytes de las secciones del ejecutable en memoria. Este valor debe ser mayor o igual a *FileAlignment* y por lo general equivale al valor de una página en memoria: 4096 bytes.
- *FileAlignent:* Este campo nos proporciona un valor de alineamiento para los datos físicos de las secciones. Es indispensable que una sección física tenga como tamaño mínimo este valor y cualquier valor superior deberá ser alineado al mismo. Este valor puede ser potencia de 2 y su valor estará entre 0x200 y 0x10000.
- *MajorOperatingSystemVersion:* Este campo proporciona la versión principal del sistema operativo requerido.
- *MinorOperatingSystemVersion:* Este campo proporciona la versión mínima del sistema operativo requerido.
- *CheckSum:* Este es un campo que proporciona el resultado de un algoritmo de suma de comprobación del archivo proporcionado por la función *CheckSumMappedFile* o *MapFileAndCheckSum* de la librería *imagehlp.dll*. Este campo nos puede decir que el archivo PE contiene librerías dll o no, sin embargo es muy contradictorio que no tenga archivo.dll debido a que todo archivo ejecutable debe tener mínimo archivos .dll básicos para su correcto funcionamiento.

- *Subsystem*: Este campo nos dice el subsistema requerido para ejecutar el programa, por ejemplo, bajo consola, interfase gráfica, etc. Los valores que puede tener *Subsystem* se observan en la tabla 5.2:

| CONSTANTE | VALOR |
|---|---|
| IMAGE_SUBSYSTEM_UNKHOWN | 0 |
| IMAGE_SUBSYSTEM_NATIVE | 1 |
| IMAGE_SUBSYSTEM_WINDOWS_GUI | 2 |
| IMAGE_SUBSYSTEM_WINDOWS_CUI | 3 |
| IMAGE_SUBSYSTEM_OS2_CUI | 5 |
| IMAGE_SUBSYSTEM_POSIX_CUI | 7 |
| IMAGE_SUBSYSTEM_WINDOWS_CE_GUI | 9 |
| IMAGE_SUBSYSTEM_EFI_APPLICATION | 10 |
| IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER | 11 |
| IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER | 12 |
| IMAGE_SUBSYSTEM_EFI_ROM | 13 |
| IMAGE_SUBSYSTEM_XBOX | 14 |
| IMAGE_SUBSYSTEM_WINDOWS_BOOT_APLICATION | 16 |

Tabla 5.2: Conjunto de valores con su respectivo constante para el campo Subsystem.

- SECCIONES DEL ARCHIVO PE

  Un ejecutable, previamente mencionado, se divide en encabezados y cuerpo del archivo el cual se encuentra dividido en secciones. Este encabezado es el que nos brinda información acerca de dichas secciones, el número de secciones de un archivo es proporcionado por el campo *NumberOfSections* (*FILE_HEADER*) mencionado anteriormente. Los campos que definen las características de cada una de las secciones son las siguientes:

  - *Name:* Este campo representa una serie de bytes que contienen una cadena el cual representa el nombre de la sección terminada con un carácter nulo (0x00) en caso de que la cadena sea menor a 8 caracteres. Si la cadena tiene exactamente 8 caracteres entonces no se usa un carácter nulo al final. Este campo no puede tener más de 8 bytes
  - *VirtualSize*: El valor de este campo corresponde al tamaño que ocupara la sección una vez cargada en memoria. Si este valor es mayor al *SizeOfRawData* los bytes sobrantes serán rellenados por caracteres nulos (0x0).
  - *SizeOfRawData*: Este campo hace referencia al tamaño de la información inicializada en disco correspondiente a la sección. Si la sección solamente tuviese información no inicializada este campo tendrá el valor 0, además debe ser múltiplo de *FileAlignment*.
  - *PointerToRawData*: Este campo corresponde al desplazamiento del archivo donde se encuentra el primer byte de la sección en disco. En caso de que la sección únicamente tuviese información no inicializada el valor de este campo sería 0.
  - *PointerToRelocations*: El valor de este campo al comienzo de las entradas de reubicaciones de la sección. En archivos ejecutables este valor por defecto es 0, debido que esto lo hace el campo *"Reloc directory"*.
  - *NumberOfRelocations*: el valor de este campo corresponde al número de entradas de las reubicaciones en la sección. En los archivos ejecutables su valor es 0.

- *Characteristics*: Este campo expresa las distintas características que describen la sección. Aplica también múltiples características de manera que se suma el valor de cada característica presente.

Hay presentes en el archivo PE más campos que definen características, sin embargo dichos campos solo sirven para archivos *COFF* [16], por lo tanto el valor que tienen en un archivo PE es 0 y no hace falta mencionarlos para esta aplicación.

# Capítulo 6

# Desarrollo

Comenzamos este proyecto con la instalación y manejo de una máquina virtual. Existen varios programas que ayudan en la creación de máquinas virtuales algunos de ellos son:

- *VirtualBox* Puede emular maquinas con sistemas operativos *Windows*, *Linux* y *Mac*. Este software es gratis y es accesible para todo usuario en Internet
- *Parallels*: Puede crear únicamente maquinas con el sistema operativo *Mac*. Este software no es gratuito pero contiene componentes más complejos como portapapeles sincronizado, carpetas compartidas y capacidad para imprimir.
- *VMware*: Existen dos tipos de *VMware*: *VMware Workstation Player* y *VMware Workstation Pro*. *VMware Workstation Player* es gratuito dirigida a usuarios causales que necesitan crear y ejecutar máquinas virtuales, pero no necesitan soluciones avanzadas a nivel de empresa. *VMware Workstation Pro* incluye todas las características de *VMware Workstation Player* solo que añade la posibilidad de clonar maquinas, crear varias imágenes del sistema operativo y opciones para probar software y grabar los resultados dentro de la máquina virtual

En este proyecto se utilizó *VirtualBox* ya que es el más accesible para los usuarios y emula de manera eficiente una PC con el fin de elaborar un programa.

Una vez que se ha instalado *VirtualBox* se prosiguió a descargar una Imagen *ISO* el cual simula un CD original que contiene un sistema operativo ya sea *Windows*, *Linux* o *Mac*. Se decidió utilizar el sistema operativo *Linux* Debían debido a que es el más cómodo para programar y el más práctico para el estudio de *Linux*.

Para el proyecto nos enfocamos en tener la versión 3.0 de *Python* o superior debido a que las nuevas versiones de *Python* manejan módulos más complejos, entre ellas la más importante y esencial en este proyecto llamada *pefile*.

## 6.1. pefile

Es un módulo multiplataforma de *Python* que sirve para analizar y trabajar con archivos de tipo PE. La mayoría de la información contenida en las cabeceras PE es accesible así como detalles de las secciones y sus datos. Para poder utilizar *pefile* se debe importar dicho módulo de la siguiente manera:

***Import pefile***

*pefile* contiene una variedad de clases las cuales cumplen diferentes funciones. En este proyecto se enfoca en la clase PE, el cual es el que necesitamos para poder leer las características que contiene el archivo PE que se desea analizar.

Una vez que se ha importado el modulo hace falta declarar un objeto de dicha clase para tener acceso a los parámetros que maneja. Este objeto se declara como sigue:

***Pe = pefile.PE(ruta/hacia/ archivo.exe)***

La ruta hacia el archivo en *Linux* es la ruta entre carpetas que se debe tomar para llegar a la carpeta en donde se encuentran los archivos PE sean estos maliciosos o no. Otra forma de mandar la ruta del archivo es pasándo la ruta como argumento al momento de ejecutar el programa de *Python*. Para que el programa reconozca este argumento y lo utilice en el código, es necesario tener un módulo llamado *sys* [17] que contiene la función *sys.argv ( )* y este permite el paso de argumentos a *Python* desde la línea de comandos. Esta función actúa como un puente a la capacidad de comunicarse entre *Python* y otros lenguajes para poder interactuar.

Para poder utilizar la función *sys.argv( )* es necesario importar el módulo *sys*, de tal modo que la segunda forma de establecer las rutas de los archivos PE se hace de la siguiente manera:

***Import sys***
***file = sys.argv[1]***
***Pe = pefile.PE (file)***

De este modo el algoritmo puede estar analizando diferentes archivos PE y ver sus características.

Una vez que utilizamos la clase PE y le damos una ruta de un archivo PE, la variable que se asignó la función de la clase, en este caso la variable Pe, se le asigna todos los parámetros del archivo PE, datos de sus cabeceras y secciones, como atributos de la instancia PE, es decir que ya se pueden tener acceso a cada uno de los valores del archivo PE previamente analizado.

Aunque ya se tiene cada uno de los campos del archivo PE no está resuelto el objetivo principal de este proyecto el cual es identificar de un conjunto de archivos PE cuales pueden ser *Malware* o contener *Malware* y que archivos no. Además de que se debe explicar un asunto muy importante acerca de los archivos PE.

## 6.2. Archivos PE encriptados

Que los archivos PE se encuentren encriptados significa que el contenido de un archivo PE no está entendible para un usuario porque al utilizar programas para abrir archivos de texto, por ejemplo *WordPath*, block de notas, entre otros, se visualiza un conjunto de caracteres en *Unicode*, es decir alfanuméricos, que un usuario no puede entender, sin embargo no existe problemas o complicaciones para ejecutar el archivo lo que implica que el contenido del archivo PE si es importante para que dicho archivo se ejecute en una computadora.

Debido a que el archivo PE se encuentra encriptado, es necesario utilizar el módulo *pefile* que contiene la clase PE ya que este nos permite desencriptar los caracteres *Unicode* del archivo PE para que así sea entendible para los usuarios, cumpliendo así el objetivo de desencriptar cadenas *Unicode* que contienen *Malware*.

## 6.3. Parámetros para la detección de Malware

Ahora se procede a manejar los parámetros obtenidos con objeto Pe. De acuerdo con las investigaciones de *Yibin Liao* [18] se obtuvieron ciertos parámetros y algunas metodologías para averiguar que archivo PE que se esté analizando al momento puede contener *Malware*:

- Valor de *SizeOfInitializedData* igual a 0

  Definido anteriormente, el campo *SizeOfinitializedData* definen la dirección de inicio de la sección de datos, el cual se encuentra el código de programa que es necesario para su instalación. El hecho de que este campo tenga un valor de 0 está informando que no existe ningún código para que el archivo PE se logre ejecutar por lo tanto se tiene una contradicción a menos que un *hacker* haya colocado dicha característica con el objetivo de no llamar la atención.

  Las figuras 6.1 y 6.2 muestran la detección de este punto clave en un archivo sospechoso y otro archivo que no respectivamente. En el caso que el algoritmo detecta que el campo *SizeOfInitializedData* tiene el valor 0 escribe la palabra POSITIVO y en caso contrario escribe NEGATIVO.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: POSITIVO.
```

Figura 6.1: Resultado positivo del punto clave en un archivo PE.

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

 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: NEGATIVO
0x4400
```

Figura 6.2: Resultado negativo del punto clave de un archivo PE.

- Valor de *DllCharacteristics* igual a 0

  Este valor define no solo la posición de inicio de los archivos *.dll* que maneja el archivo PE sino que también indica cuales archivos *dll* específicos utiliza para su correcto funcionamiento. Que este campo tenga el valor de 0 quiere decir que este archivo PE no tiene archivos *dll* lo cual no puede ser posible porque cualquier programa o ejecutable necesita de los archivos *dll* para su correcta ejecución, por lo tanto ha sido modificado por *hacker* colocando en dicho archivo *Malware*.

  Las figuras 6.3 y 6.4 muestran los casos de detección de este campo en un archivo sospechoso y en otro limpio respectivamente.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: POSITIVO.

valor de SizeOfInitializedData: 0
caracteristicas DLL: POSITIVO
```

Figura 6.3: Resultado positivo de *DllCharacteristics* en un archivo PE.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: NEGATIVO
0x1800

caracteristicas DLL: NEGATIVO
0x8140
```

Figura 6.4: Resultado negativo de *DllCharacteristics*en un archivo PE.

- Valor de *MayorImageVersion* igual a 0

  Este valor define el tipo de versión en el cual el archivo PE actualmente analizado se ejecuta, estos pueden ser *Windows, Linux, Mac, IOS*, entre otros. Debido a que debe trabajar a lo mas en un sistema operativo, dicho valor no debe ser 0 y tener el valor correspondiente a la versión que se ejecutara, en este caso *Windows*, por lo tanto el hecho de que este campo tenga un valor a 0 quiere decir que no necesita de ningún sistema operativo, lo cual resulta contradictorio ya que si no los necesita entonces este archivo no se podrá ejecutar, así que es buen parámetro de búsqueda para *Malware*.

  Este caso se ve reflejado en las figuras 6.5 y 6.6 para un archivo sospechoso y otro archivo limpio respectivamente.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: POSITIVO.

valor de SizeOfInitializedData: 0
caracteristicas DLL: POSITIVO

version de imagen mayor: POSITIVO
```

Figura 6.5: Resultado positivo de *MayorImageVersion* en un archivo PE.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: NEGATIVO
0x1c000

caracteristicas DLL: NEGATIVO
0x8000

version de imagen mayor: NEGATIVO
0x5
```

Figura 6.6: Resultado negativo de *MayorImageVersion*en un archivo PE.

- Valor del campo *Checksum* igual a 0

  Este valor define el número de archivos incluidos en el archivo PE, pueden ser archivos *.dll* al iniciar el programa o controladores, lo cual hace que su valor dependa del número de archivos *.dll* que logro cargar para su ejecución. Cuando el valor de *Checksum* es igual a 0 quiere decir que no cargo ningún archivo *.dll* para su ejecución, de echo este campo está ligado directamente con el campo *DllCharacteristics* debido a lo que se explicó previamente, lo cual no puede ser porque una aplicación necesita de dichos archivos para poder ejecutarse correctamente, entonces dicho archivo ha sido modificado por un *hacker* para no levantar sospechas.

  este caso se puede observar en las figuras 6.7 y 6.8 para un archivo sospechos y uno limpio respectivamente.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: POSITIVO.

valor de SizeOfInitializedData: 0
caracteristicas DLL: POSITIVO

version de imagen mayor: POSITIVO

checksum: POSITIVO

Verificacion de Checksum:

['CRC:     Claimed: 0x0, Actual: 0xe6fb [sospechoso] ']
```

Figura 6.7: Resultado positivo del campo *Checksum* en un archivo PE.

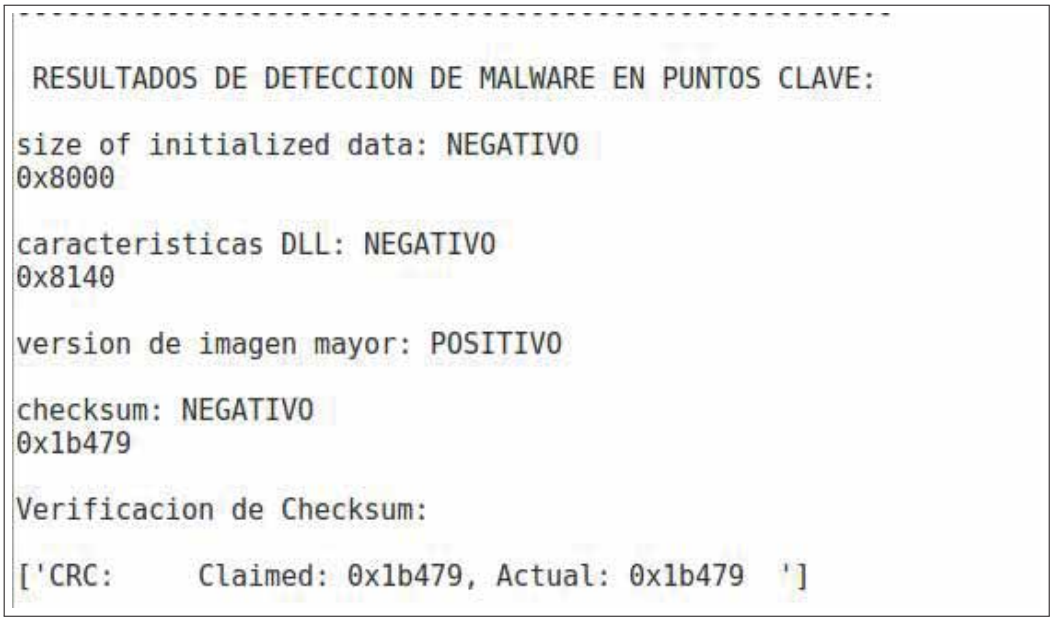



Figura 6.8: Resultado negativo de *Checksum* en un archivo PE.

- Nombres de las secciones del archivo PE extraños

  Como ya se ha mencionado cada archivo PE se compone de varias secciones los cuales cada uno cumple una función diferente y contiene un conjunto de campos que sus valores son también diferentes de cada sección. Algunos de los nombres de las secciones validas que tienen un archivo PE son los siguientes:

  - *.text*: El código del programa o aplicación
  - *.bss*: Las variables no inicializadas del archivo PE
  - *.reloc*: La tabla de localizaciones
  - *.data*: Las variables inicializadas
  - *.rsrc*: Los recursos del archivo PE (cursores, sonidos, menús,. . . .)
  - *.rdata*: Los datos de solo lectura
  - *.idat*a: La tabla de importación

  Las siguientes secciones pueden estar o no dentro del archivo PE:

  - *.upx*: firma de una compresión *UPX*, propio de *software UPX*.
  - *.aspack*: firma de paquetes *ASPACK*, propio de *software ASPACK*.
  - *.adata*: firma de un paquete *ASPACK*, propio de *software ASPACK*

La figura 6.9 muestra un caso de nombres de secciones sospechosas en un archivo PE y la figura 6.10 nombres de secciones validos que debe de tener un archivo PE.

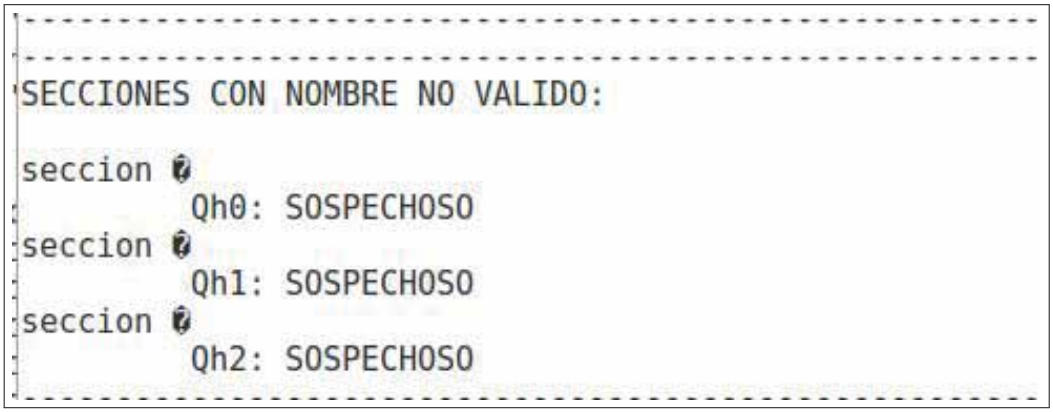



Figura 6.9: Nombres de secciones sospechosas.

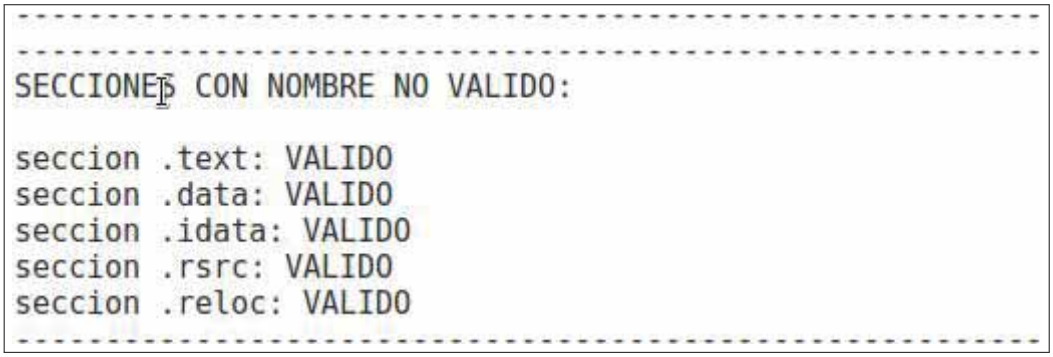



Figura 6.10: Nombres de secciones validas de un archivo PE.

El primer conjunto de secciones son las secciones base de un archivo PE para su correcta ejecución lo cual será lo más visto al momento de obtener las características de un conjunto de archivos PE.

El segundo conjunto de secciones siguen siendo secciones base como las anteriores con la excepción de que se utilizó un algoritmo o herramienta de compresión o empaquetamiento como lo son *UPX* o *ASPACK* respectivamente.

A pesar de que pueden existir unas secciones empaquetadas y otras que no, en este proyecto se utilizaron un conjunto de muestras de *Malware* que no contienen secciones empaquetadas, por lo tanto solo será posible encontrar nombres de secciones base para el análisis de *Malware*.

Otra pista que nos puede indicar que el archivo PE analizado contiene *Malware* es que los nombres de sus secciones tengan nombres desconocidos, por ejemplo .6dnn4fha, .Bga1m3ar, entre otros.

Estos han sido los principales puntos por los cuales este proyecto está basado para diferenciar entre los archivos PE que contienen *Malware* y cuáles no, sin embargo, se han encontrado otras metodologías que nos pueden dar más pistas para diferenciar los archivos PE con *Malware* de los que no. Dichas metodologías son:

- Valor de *Checksum* diferente al valor cuando fue creado

  Como se mencionó previamente, el valor de *Checksum* es uno de los principales valores en la cual puede existir presencia de *Malware* al observar que dicho valor sea 0. A pesar de que esta característica que se busca indica que el archivo PE no contiene archivos *.dll*, es incorrecto, lo que quiere decir que fue modificado por el *hacker*. La clase *PE* incluida en el módulo *pefile* contiene un método llamado *generate_checksum( )* el cual realiza un cálculo del número de archivos *.dll* que contiene el archivo PE. Ya que se obtuvo el valor de la función *generate_checksum( )*, se realiza una comparación del valor calculado junto con el valor que se obtiene con el campo *pe.OPTIONAL_HEADER.CheckSum* previamente explicado. Los resultados pueden variar, por ejemplo siendo estos iguales a 0 o diferentes de 0 pero lo que se busca es que tengan el mismo valor, y si estos valores son diferentes ayuda a verificar que este campo ha sido modificado para fines maliciosos.

  El modo de utilizar la función *generate_checksum()* es asignando el resultado a una variable de la siguiente forma:

  $$\boldsymbol{crc\_actual = pe.generate\_checksum( )}$$

  De esta manera ya se obtiene dicho valor para la comprobación. La figura 6.11 visualiza el caso en que hubo una modificación en el campo *Checksum*.

```
 RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:

size of initialized data: NEGATIVO
0x1800

caracteristicas DLL: NEGATIVO
0x8140

version de imagen mayor: POSITIVO

checksum: NEGATIVO
0x67cd

Verificacion de Checksum:

['CRC:     Claimed: 0x67cd, Actual: 0x6573 [sospechoso] ']
------------------------------------------------------------
```

Figura 6.11: Valores diferentes del campo Checksum en el archivo PE.

- Secciones en base al tamaño de su contenido

  Este método ayudará a identificar que secciones pueden contener *Malware* basándose en el tamaño de su contenido. Esto será útil para brindar información porque a pesar que ya se puede identificar nombres extraños en algunas secciones en el archivo PE, no garantiza que las secciones que tienen nombres validos estén libres de *Malware*. Este método necesita de dos valores contenidos en el archivo PE: entropía y el tamaño de la sección.

  - Entropía: se considera como la cantidad de información promedio que contienen las secciones. En resumen proporciona la cantidad de información que contiene la sección de una forma comprimida tomando como base variables especificas pertenecientes a la misma sección. Este valor se puede obtener con la función get_entropy( )
  - Tamaño: la cantidad en bytes del contenido de la sección. Este valor se obtiene del campo *SizeOfRawData* del archivo PE.

  La entropía no puede pasarse de 7 debido a que dicho valor indica que la sección tiene mucho contenido. Cuando la entropía tiene un valor menor a 7 se considera normal para el contenido de la sección pero también se debe verificar que dicho valor no se encuentre entre 0 y 1 porque eso indica que no hay contenido dentro de la sección lo cual resulta contradictorio porque si la sección se encuentra en el archivo ejecutable entonces debe tener archivos o código en su interior.

  En la figura 6.12 se muestra una sección del código fuente [19] perteneciente a *Nullege* [20] que explica el párrafo antes mencionado.

```
for sec in pe.sections:
    s = "%-10s %-12s %-12s %-12s %-12f" % (
        ''.join([c for c in sec.Name if c in string.printable]),
        hex(sec.VirtualAddress),
        hex(sec.Misc_VirtualSize),
        hex(sec.SizeOfRawData),
        sec.get_entropy())
    if sec.SizeOfRawData == 0 or (sec.get_entropy() > 0 and sec.get_entropy() < 1) or sec.get_entropy() > 7:
        s += "[SUSPICIOUS]"
    out.append(s)
```

Figura 6.12: Código fuente de *Nullege* referente a la entropía.

Los valores de estos campos son diferentes por cada sección que tiene el archivo PE, por lo cual es recomendable utilizar un ciclo for para acceder a cada una de la secciones del ejecutable y ahí calcular los respectivos valores del tamaño y la entropía.

El contenido de cada sección se encuentra encriptado con *MD5* y *SHA1* los cuales son métodos de encriptación por lo cual será necesario ver posibles módulos de *Python* que nos pueden ayudar a realizar dicha desencriptación.

## 6.4. Otras metodologías de detección

Estos métodos no nos ayudaran a identificar que archivos PE contienen *Malware* y que archivos no, de manera programable, pero serán útiles para los especialistas en diseñar antivirus ya que estos se basan en el código ensamblador, llamadas *API*, entre otros. A continuación se explicara acerca de estos métodos.

- Desensamblador

  El proceso de programación hace uso de diferentes lenguajes, por ejemplo *Python, C, C++, Java*, entre otros. Para una computadora el único lenguaje que puede entender es el llamado código maquina [21] y está formado por conjuntos de ceros y unos, es decir código binario. Sin embargo un programador o usuario no es capaz de entender dicho lenguaje lo que hizo que fuera necesario crear lenguajes de programación que fueran entendibles y controlables para los mismos. El lenguaje ensamblador fue el primero de estos lenguajes y sigue usándose actualmente. Este lenguaje expresa las instrucciones de una forma más natural al usuario y a la vez muy cercana al microprocesador de una computadora ya que cada una de sus instrucciones puede convertirse en su respectivo en código máquina, es decir al momento de ejecutar el programa en código ensamblador, este código se convierte en código máquina para que sea entendido por la computadora.

  Debido a que el código ensamblador es el lenguaje más cercano a los microprocesadores, su sintaxis involucran registros, direcciones de memoria, asignaciones, etc., lo que puede ayudar que un especialista en microprocesadores a identificar en que parte del código se puede encontrar la parte en donde el *Malware* comienza a ejecutarse.

  *Python* tiene un módulo que nos puede ayudar a identificar el contenido de un archivo PE y convertirlo en lenguaje ensamblador llamado *Pydasm* [22] el cual nos puede ayudar a obtener dicho lenguaje.

  En la figura 6.13 se visualiza un ejemplo de un código en ensamblador de un archivo PE calculado por el algoritmo de este proyecto.

```
DESENSAMBLADOR:

push dword 0x401c34
call 0x401612
add [eax],al
inc eax
add [eax],al
add [eax],dh
add [eax],al
add [eax],bh
add [eax],al
add [eax],al
add [eax],al
add ch,al
mov cl,0xa0
cmp byte [ebx+0x45],0xc2
dec ebx
scasd
mov ds,[esi]
aad 0x92
push edi
and eax,0x1
add [eax],al
add [ecx],al
add [eax],al
```

Figura 6.13: Código ensamblador de un archivo PE.

- Llamadas a interfaces de programación de aplicaciones (*API*)

  Una *API* es un conjunto de código (o reglas) y especificaciones que las aplicaciones pueden seguir para comunicarse entre ellas: sirviendo de interfaz entre programas diferentes de la misma manera en que la interfaz de usuario facilita la interacción humano – computadora.

Las *APIs* pueden servir para comunicarse con el sistema operativo (*Windows, Linux, etc*), con bases de datos o con protocolos de comunicaciones. Permiten hacer uso de funciones ya existentes en otro software (o de la infraestructura que ya existe en otras plataformas) para ahorrar tiempo y recursos en creación de funciones ya existentes, reutilizando así código que se sabe que está probado y que funciona correctamente. En el caso de herramientas propietarias, es decir que no sean de código abierto, son un modo de hacer saber a los programadores de otras aplicaciones como incorporar una funcionalidad concreta sin por ello tener que proporcionar información acerca de cómo se realiza internamente el proceso.

Debido a que las funciones *API* se encuentran ya existentes y la mayoría de aplicaciones o programas hacen uso de las mismas, es muy probable que los diseñadores de *Malware* o *hackers* incluyan en el archivo PE llamadas a funciones *API* sospechosas, como dar acceso en la computadora a un servidor en Internet o que solicite una dirección *IP* ya sea del equipo que se esté usando en el momento, de una página web o incluso ambos, es por eso que será de gran ayuda informar que tipos de llamadas *API* necesita el archivo PE analizado para poder identificar el tipo de *Malware* con el que se está tratando y darle posiblemente clasificarlo.

La figura 6.14 muestra las llamadas *API* que un archivo PE solicita para su funcionamiento:

```
lista de llamadas API:

['SearchPathW', 'SetEnvironmentVariableW', 'ExpandEnvironmentStringsW', 'GetEnvironmentVariableW',
 'LoadStringW', 'LoadLibraryExW', 'GetProcAddress', 'FreeLibrary', 'GetModuleHandleA', 'CreateProc
essW', 'GetCurrentThread', 'GetCurrentProcessId', 'GetCurrentThreadId', 'GetCurrentProcess', 'Crea
teThread', 'SetThreadPriority', 'TerminateProcess', 'GetFileAttributesExW', 'CompareFileTime', 'Wa
itForSingleObject', 'OpenEventW', 'Sleep', 'GetTickCount', 'GetSystemTimeAsFileTime', 'GetVersionE
xW', 'GetSystemDirectoryW', 'RegQueryValueExW', 'RegCloseKey', 'RegQueryInfoKeyW', 'RegOpenKeyExW'
, 'RegEnumValueW', 'SetLastError', 'SetUnhandledExceptionFilter', 'UnhandledExceptionFilter', 'Get
LastError', 'HeapSetInformation', 'CharNextW', 'CloseHandle', 'lstrlenW', 'LocalFree', 'LocalAlloc
', '__p__commode', '_XcptFilter', '_wtoi', 'memmove', '_except_handler4_common', '_controlfp', '__
getmainargs', '_acmdln', '_initterm', '__setusermatherr', '_ismbblead', '__p__fmode', '_cexit', '_
exit', 'exit', '_amsg_exit', '__set_app_type', '?terminate@@YAXXZ', '_wcsicmp', 'memcmp', 'memset'
, 'QueryPerformanceCounter', 'GetStartupInfoA', 'ApiSetQueryApiSetPresence', 'ResolveDelayLoadedAP
I', 'DelayLoadFailureHook']
```

Figura 6.14: Conjunto de llamadas API en un archivo PE.

- Llamadas a localizadores Uniformes de Recursos (*URL*)

  Una *URL* es una secuencia de caracteres que sigue un estándar y que permite denominar recursos dentro del entorno de Internet para que puedan ser localizados. Los documentos de texto, fotografías y los audios, entre otros tipos de contenidos digitales, tienen un *URL* cuando se publican en Internet. Estos localizadores permiten crear enlaces o *links* en la *World Wide Web (WWW)*, lo que facilita la navegación.

  Estas *URL* son las que definen que servicios o servidores necesita el archivo PE para que pueda ejecutarse correctamente ya sea una instalación u obtener recursos que ayudan a completar su instalación.

  Estas llamadas están relacionadas con las llamadas *API* ya que es posible que algunas funciones *API* necesiten de un servidor en Internet por lo que hacen uso de la *URL* que garantice el enlace hacia dicho servidor. Es por esta razón que es importante presentar en el reporte los tipos de *URL* que se enlaza el archivo PE, en el caso que tenga direcciones *URL*, para poder identificar el comportamiento del *Malware* el dicho archivo.

Las versiones actuales de *Python*, de la versión 3 en adelante, tienen una librería llamada *re* [23] la cual necesitamos debido a que los patrones del archivo PE se encuentran encriptadas, como antes se mencionó, siendo cadenas *Unicode* y patrones de 8 bits. Sin embargo, no se puede hacer coincidir una cadena *Unicode* con un patrón de bytes o viceversa. Es por esa razón que se hace uso del módulo *re* ya que este emplea expresiones regulares las cuales nos servirán para encontrar las coincidencias entre cadenas *Unicode* y los patrones de 8 bits.

Las expresiones regulares especifican un conjunto de cadenas que coincidan con dichas expresiones, por lo tanto las funciones del módulo *re* permiten verificar si una cadena en particular coincide con una expresión regular dada.

Este proceso de coincidencias comienza haciendo un ciclo en la cual vayamos haciendo coincidir cadenas *Unicode* contenidas en el archivo PE con una expresión regular general, una cadena que contenga letras, números e incluso símbolos que se puedan imprimir en pantalla, e ir registrando dichas coincidencias. Este proceso lo hacemos con la función contenida en el módulo *re* llamada *findall( )* la cual recibe 3 parámetros: la expresión regular, la cadena *Unicode* y un valor perteneciente al conjunto de parámetros del módulo *re* que define lo que la función se enfoque en encontrar, en este caso se usó los parámetros *re.IGNORECASE* y *re.MULTILINE* los cuales realizan la coincidencia de mayúsculas y minúsculas y verifican la posición de las cadenas en la expresión regular por medio de los símbolos específicos, por ejemplo $ . El propósito de este primer proceso es encontrar todas las cadenas en el archivo PE que se pueden imprimir en pantalla y evitar símbolos como comillas (“”), apostrofes (‘), entre otros.

Finalmente se vuelve a realizar un ciclo de coincidencias con expresiones regulares pero en este caso ya se buscan cadenas específicas que tienen que ver con direcciones *URL*, es decir cadenas que empiecen con *www, http, ftp*, entre otros los cuales son esquemas que indican el protocolo de red usado para recuperar la información del recurso identificado. Este proceso se realiza con la función *search( )* que también recibe 3 parámetros: la expresión regular especificando los protocolos de red que utilizan las *URLs*, la cadena del archivo PE, y el valor de búsqueda de coincidencias el cual es *re.IGNORECASE*. La salida de esta función será una lista con todas las *URLs* que el archivo se enlaza para su funcionamiento.

- Firmas del archivo PE.

  Anteriormente, estas firmas solo podían ser escaneadas del archivo PE utilizando una herramienta llamada *PEiD* [24] que se usa para detectar empaquetadores, encriptadores y compiladores encontrados en dichos archivos, sin embargo desde la versión 1.2.6, *pefile* admite el análisis de las firmas de *PEiD* gracias a un módulo incluido en *pefile* llamado *peutils*. Este módulo realiza algunas características de la herramienta PE, es decir, nos puede indicar si el archivo PE esta empaquetado o tiene secciones empaquetadas, nos puede dar una aproximación de las firmas o identificadores del archivo y nos puede dar un indicativo de que el archivo PE que se esté analizando es sospechoso o no.

Para obtener la aproximación del identificador o firma del archivo necesitamos un archivo extra que contiene un conjunto de firmas ya clasificadas llamado *UserDB.txt*. Este archivo es abierto para los usuario interesados y se puede descargar en [25].

Como *peutils* es un módulo más de *Python*, se debe importar de la siguiente forma:

***Import peutils***

Una vez que este módulo se ha importado, tenemos que cargar en memoria el archivo *UserDB.txt* usando la siguiente función:

***signatures = peutils.SignatureDatabase('ruta/hacia/archivo.txt')***

Ya que escribimos esta sentencia, la variable *signatures* ya se vuelve un objeto del módulo *peutils* y ya contiene las funciones que ofrece.

Ahora procedemos a buscar todas las posibles coincidencias que identifiquen la firma o identificador del archivo PE, esto lo hacemos con la función *match_all( )*, la cual necesita 3 parámetros: el contenido del archivo PE, y dos variables llamados *ep_only* y *section_start_only*. Su sintaxis es la siguiente:

***coincidencias = signatures.match_all(pe, ep_only=True,***
***section_start_only=False)***

Si *ep_only* es verdadero, el resultado será una cadena con el nombre del empacador. De lo contrario, será una lista de la forma (*file_ofsset*, nombre del empaquetador). Especificando en que parte del archivo se encontró la firma [26].

Después de que se escribió esta sentencia, la variable coincidencias ya contiene una lista de posibles firmas que puede identificar al archivo PE, siendo información útil para los especialistas en antivirus.

La figura 6.15 muestra un diagrama de flujo del proceso de detección de *Malware* en los archivos PE.

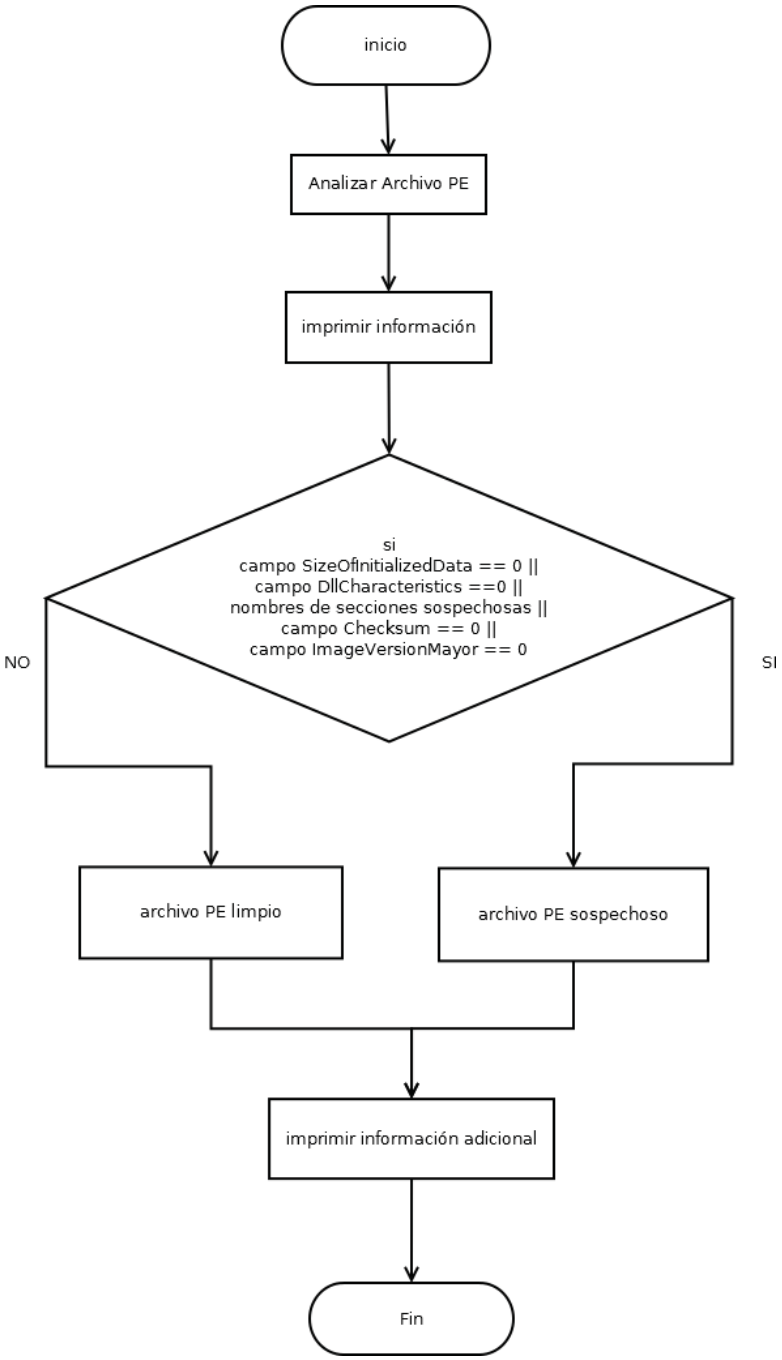



Figura 6.15: Diagrama de flujo para la detección de *Malware* en Archivos PE.

# Capítulo 7

# Resultados

Los resultados se tomaron en cuenta en base a los 5 puntos clave que se explicaron en la parte de desarrollo de este reporte y se analizaron 99 archivos PE los cuales 49 archivos contienen *Malware* y 50 archivos no. Estos resultados se dividirán en tablas y cada una nos dirán que puntos clave nos servirán más para poder detectar que archivos PE contienen *Malware* y que archivos no.

En la tabla 7.1 se calcularon los porcentajes de cada uno de los puntos de manera individual. El porcentaje se obtuvo analizando un total de 99 archivos PE, siendo este conjunto una unión entre archivos PE con *Malware* y archivos PE limpios.

| característica definida | malware | normal | porcentaje malware | porcentaje normal | diferencia % |
|---|---|---|---|---|---|
| 1) Size Of Initialized Data == 0 | 15 | 84 | 15.15151515 | 84.84848485 | -69.6969697 |
| 2) nombres de secciones desconocidas | 24 | 75 | 24.24242424 | 75.75757576 | -51.51515152 |
| 3) características DLL == 0 | 52 | 47 | 52.52525253 | 47.47474747 | 5.050505051 |
| 4) imagen de versión mayor == 0 | 65 | 34 | 65.65656566 | 34.34343434 | 31.31313131 |
| 5) Checksum == 0 | 55 | 44 | 55.55555556 | 44.44444444 | 11.11111111 |

Tabla 7.1: Porcentaje de resultados del análisis de archivos PE.

La tabla 7.1, tal como se mencionó previamente, se obtuvieron los resultados tomando los puntos clave de manera individual, por lo que el siguiente paso consistirá en realizar varios conjuntos de dichos puntos y observar los resultados.

Para realizar la segunda tabla procedimos a asignar un número a cada punto clave que nos ayudara a identificar más rápido el mismo. Dicha asignación se muestra en la tabla 7.2.

| característica definida | valor |
|---|---|
| Size Of Initialized Data == 0 | 1 |
| nombres de secciones desconocidas | 2 |
| características DLL == 0 | 3 |
| imagen de versión mayor == 0 | 4 |
| Checksum == 0 | 5 |

Tabla 7.2: Asignación de un número a cada punto clave.

En la tabla 7.3 se muestran los resultados de varios conjuntos de los puntos clave tomando las asignaciones que se muestran en la tabla 7.2.

| características combinadas | Malware | Normal | % malware | % normal | % diferencia |
|---|---|---|---|---|---|
| 1,2 | 23 | 76 | 23.23232323 | 76.76767677 | -53.53535354 |
| 1,2,3 | 60 | 39 | 60.60606061 | 39.39393939 | 21.21212121 |
| 1,2,4 | 61 | 38 | 61.61616162 | 38.38383838 | 23.23232323 |
| 1,2,5 | 56 | 43 | 56.56565657 | 43.43434343 | 13.13131313 |
| 1,2,3,4 | 72 | 27 | 72.72727273 | 27.27272727 | 45.45454545 |
| 1,2,3,5 | 72 | 27 | 72.72727273 | 27.27272727 | 45.45454545 |
| 1,2,4,5 | 71 | 28 | 71.71717172 | 28.28282828 | 43.43434343 |
| 1,2,3,4,5 | 77 | 22 | 77.77777778 | 22.22222222 | 55.55555556 |
| 3,4,5 | 75 | 24 | 75.75757576 | 24.24242424 | 51.51515152 |

Tabla 7.3: Resultados de detección en base a varios conjuntos de los puntos clave.

# Capítulo 8

# Análisis y discusión de resultados

El objetivo de la tabla 7.1 es mostrar que tan frecuente el algoritmo detecta un resultado positivo en cada punto clave de forma individual en un archivo PE para poder observar la precisión del algoritmo al momento de realizar un análisis de un programa o archivo ejecutable.

Es por esta razón que no solo se registraron los resultados positivos (columna **malware** de la tabla 7.1) de los puntos clave sino que además los resultados negativos (columna **normal** de la tabla 7.1), es decir cuando los puntos clave no presentaban sospechas de *Malware*, se obtuvieron sus porcentajes respectivos y además se estableció en una columna llamada **Diferencia %** el margen de error en cuanto a la precisión del algoritmo.

Por ejemplo: si observamos el punto clave *"Size Of Initialized Data"* podemos ver que de un conjunto de 99 archivos PE analizados, el 15.15 % se detecto una sospecha positiva de *Malware* en este punto clave y el 84.84 % se obtuvieron resultados negativos, lo que dio un margen de error del 69.69 % (columna **Diferencia %** de la tabla 7.1) que se obtuvo haciendo una resta de su porcentaje de detección positiva (**porcentaje *malware***) con su porcentaje de detección negativa (**porcentaje normal**).

Los resultados han sido aceptables, considerando que en este proceso de detección se tomó los puntos clave de manera independiente. Sin embargo, durante el registro de resultados se observaron varios detalles:

- Los nombres de las secciones, en la mayoría de los archivos PE analizados, no fueron desconocidos, ya que estos nombres coincidieron con los nombres validos mencionados previamente en el desarrollo de este proyecto. Se pudo notar que hubo archivos PE los cuales varios de sus secciones estaban empaquetados y tenían el nombre de la herramienta que utilizaron para dicho proceso, en estos casos, se vio que la herramienta más utilizada fue *UPX*.

- El valor del campo *SizeOfInicializedData* fue distinto de 0 en la mayoría de los archivos PE analizados obteniendo así un porcentaje de detección de *Malware* del 15.15 %.
- En los campos donde se obtuvo un mayor porcentaje de detección de *Malware* fueron *"características DLL"*, "imagen de versión mayor" y *"Checksum"* con 52.52 %, 65.65 % y 55.55 % respectivamente.
- El campo "Imagen de versión mayor" obtuvo el porcentaje de detección de *Malware* más alto, sin embargo, en el desarrollo de este proyecto se explicó que se tomaron 99 muestras en total para el cálculo de resultados, los cuales 55 muestras están limpias y 49 muestras contienen *Malware*. Este campo tuvo un resultado positivo en 65 archivos PE considerando que en el conjunto solo hay 49 archivos PE con *Malware*.

En la tabla 7.3 se busco identificar en que puntos clave, tomándolos de manera conjunta, es decir ya no de forma independiente, hay mayor porcentaje de identificar que archivos PE contienen *Malware* y que archivos PE estan limpios.

Para realizar esta búsqueda se tomo la metodología representada en el diagrama de flujo de la figura 6.15, es decir, tomando un conjunto de puntos clave como los seleccionados en la tabla 7.3 y basándonos en la asignación de los identificadores de la tabla 7.2, en el cual almenos en un punto clave se detecto sospechas positivas.

Por ejemplo: si tomamos la tercera fila de la tabla 7.3 vemos los números 1,2 y 4, y observando la tabla 7.2 notamos que los numero hacen referencia a los puntos clave *"Size Of Initialized Data"*, "nombres de secciones desconocidas.ºimagen de version mayorrespectivamente. Una vez que entendemos las referencias, las demás columnas son parecidas a la tabla 7.1: tomando como base el diagrama de flujo identificamos, de 99 archivos PE, el 61.61 % se detectaron sospechas positivas en almenos uno de los puntos clave seleccionados y el 38.38 % dieron resultados negativos, lo que dio un margen de error (columna **%diferencia** de la tabla 7.3) del 23.23 % obtenido haciendo la misma resta del porcentaje de Malware ( **%malware**) con la resta de porcentaje normal ( **%normal**).

Aunque los resultados fueron mejores que los de la tabla 7.3 también se encontraron varios detalles importantes durante el proceso de detección:

- En la primera fila de la tabla 7.3, vemos que al tomar en cuenta los campos *"Size of Initialized Data"* y "nombres de secciones desconocidas", el porcentaje de detección fue muy bajo y tiene relación con los porcentajes de estos campos en la tabla 7.1.
- En algunos archivos PE, al principio se dedujo que son archivos limpios debido a que en los campos 1, 3, 4, y 5 (para más detalles ver la tabla 7.2) dieron resultados negativos para *Malware*, sin embargo estos contenían nombres extraños en sus secciones, por lo que se consideró como *Malware* al momento de estar realizando los cálculos de porcentajes.
- Los archivos PE más sospechosos variaban entre nombres de secciones no válidos y el valor de la entropía en sus secciones, sin embargo todos coincidían en que tenían los campos 3, 4 y 5 con detecciones positivas de presencia de *Malware*.

- Previamente se mencionó que el campo “imagen de versión mayor” tuvo una detección positiva en la mayoría de archivos PE, incluso los que no tienen *Malware*, lo que este campo no garantiza la efectividad en cuanto a la detección de *Malware*, pero al incluirlo con otros puntos clave se obtiene un mejor resultado, específicamente con los campos *“Checksum”* y *“Caracteristicas DLL”* explicado en el inciso anterior.

Para finalizar las pruebas, se realizo nuevamente el analisis de los 99 archivos PE, de los cuales 49 archivos contienen *Malware* y 50 archivos no, con el propósito de comprobar que puede realmente identificar los 49 archivos con *Malware* y los 50 archivos que no.

Los resultados fueron satisfactorios para los archivos PE con *Malware* porque el algoritmo logro detectar exitosamente el 85.71 % de dichos archivos. Sin embargo, el 14.28 % de los archivos PE dieron falsos negativos, lo que quiere decir que estos archivos necesitan examinarse con mas detalle debido a que el *hacker* logro esconder el *Malware* con el fin de evadir la detección.

Los resultados de los archivos PE sin *Malware* también han sido efectivos debido a que logro detectar el 72 % de 50 archivos, es decir que identifico correctamente que estos archivos no tienen *Malware*, y el 28 % restante dio falsos positivos, es decir que los identifico con *Malware* considerando que forman parte del conjunto de archivos PE limpios.

# Conclusiones

Como se puede ver en los resultados, el algoritmo de este proyecto obtuvo porcentajes satisfactorios pero también con un margen de error decepcionante para brindar información de detección de *Malware* en archivos PE.

Los resultados indican que es más efectivo tomar conjuntos de las características clave del archivo PE analizado para obtener mayores probabilidades de identificar si contiene *Malware* o no, especialmente que los campos *DllCharacteristics*, imagen de versión mayor y *Checksum* sean positivos, es decir que sean sospechosos, para así proceder a la clasificación del tipo de *Malware* o al análisis dinámico para ver su comportamiento.

El algoritmo puede informar que secciones se encuentran empaquetadas y muestra su contenido por medio de una cadena encriptada en MD5, pero tiene la limitante de que no consigue desempaquetar por falta de recursos y no puede desencriptar debido a que aunque los metodos para desencriptar si existen, el creador configuro un conjunto de contraseñas que no se encuentran accesibles para todo usuario y requiere de permisos privados.

Un aspecto positivo hablando de la encriptación es que solo se encuentra limitada en las secciones, no logramos visualizar el contenido de las mismas pero por el módulo pefile de Python se pudo visualizar de manera clara las características generales de los archivos PE los cuales fueron suficientes para brindar información de los mismos.

Para dar más complejidad a este proyecto, se anexaron otras metodologías al algoritmo como el desensamblador el cual ayuda a identificar sospechas de *Malware* a nivel de microprocesadores y las llamadas URL y API que nos ayudan a descubrir si hace peticiones en Internet sospechosas aparte de las peticiones que necesita para instalar complementos pertenecientes a la aplicación misma.

Por último, y debido a todo lo que proporciona el algoritmo, este proyecto cumple de una forma satisfactoria el brindar información importante para la detección de *Malware* debido a que el algoritmo detecto exitosamente el 85.71 % de archivos PE con *Malware* de un conjunto de 49 archivos maliciosos, pero tiene dificultades para detectar archivos PE limpios ya que detecto de manera exitosa el 72 % de un conjunto de 50 archivos PE limpios, esto quiere decir que el algoritmo es útil para futuras investigaciones e incluso mejorar su efectividad si es manejado por personas con mas experiencia.

# Bibliografía


[1] K. Lab. 33 ataques por segundo: Kaspersky lab registra un aumento del 59 % en ataques de malware en américa latina. [Online]. Available: https://latam.kaspersky.com/about/press-releases/2017_33-attacks-per-second-increase-inmalware-attacks-in-latin-america

[2] "Pandalabs quarterly report january-march 2014," 2014.

[3] N. D. COD3. Desensambladores para ingeniería inversa. [Online]. Available: http://nochesdecode.com.ar/2012/01/desensambladores-para-ingenieria-inversa/

[4] "Internet security threat report 2014," 2014.

[5] C. buenas noticias. Icesiste recibe premio por investigación sobre análisis de software malicioso. [Online]. Available: http://calibuenasnoticias.com/2016/10/14/icesistarecibe-premio-por-investigacion-sobre-analisis-de-software-malicioso/

[6] I. P. Nacional. Garmdroid, sistema politécnico que detecta apps con malware. [Online]. Available: https://www.la-prensa.com.mx/ciencia-y-tecnologia/154309-garmdroid-sistema-politecnico-que-detecta-apps-con-malware

[7] M. V. Campaña, "Detección de malware usando herramientas de big data," pp. 1–91. [Online]. Available: http://bibing.us.es/proyectos/abreproy/90424/fichero/PFG+-+Mar%C3%ADa+Valero.pdf+

[8] R. F. O. A. Ramírez Gutiérrez Rubén Omar, "Implementación de un laboratorio de análisis de malware," pp. 1–150, 2009. [Online]. Available: http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/1150/Tesis.pdf?%20s%C3%A9quense=1

[9] C. K. y. E. K. U. Bayer, "Ttanalize: A tool for analyzing malware."

[10] M. P. H. Page. Triid - file identifier. [Online]. Available: http://mark0.net/soft-trid-e.html

[11] GitHub. Upx: the ultimate packer for executables. [Online]. Available: https://upx.github.io/

[12] Alfon. Esas pequeñas utilidades. análisis forense. strings. [Online]. Available: https://seguridadyredes.wordpress.com/2009/10/14/esas-pequeaas-utilidades-analisis-forense-strings/

[13] P. S. Foundation. Tutorial de python. [Online]. Available: http://docs.python.org.ar/tutorial/3/real-index.html

[14] W. L. enciclopedia libre. Intel itanium. [Online]. Available: https://es.wikipedia.org/wiki/Intel_Itanium

[15] RedinSkala. El formato pe (portable executable) parte i. [Online]. Available: http://www.redinskala.com/2013/10/09/formato-pe-parte-i/

[16] W. L. enciclopedia libre. Common object file format. [Online]. Available: https://es.wikipedia.org/wiki/Common_Object_File_Format

[17] Python. Sys module python tutorial. [Online]. Available: https://pythonprogramming.net/sys-module-python-3/

[18] Y. Liao, "Pe-header-based malware study and detection," pp. 1–23. [Online]. Available: https://pdfs.semanticscholar.org/presentation/6eee/55df088005b94fb91cb7725136a785820dcc.pdf

[19] N. P. C. Research. Pefile.pe. [Online]. Available: http://nullege.com/codes/show/src@m@a@mart-HEAD@bin@pescanner.py/271/pefile.PE

[20] Nullege. Nullege. about us. [Online]. Available: http://nullege.com/pages/about

[21] U. de Castilla La Mancha, "Lenguaje maquina: la interfaz entre el hardware y el software," pp. 1–942, 2015. [Online]. Available: http://www.esi.uclm.es/www/isanchez/eco0910/maquina.pdf

[22] GitHub. libdasm. [Online]. Available: https://github.com/jtpereyda/libdasm/tree/master/pydasm

[23] Python. re. regular expressions operations. [Online]. Available: https://docs.python.org/2/library/re.html

[24] GitHub. pefile.peutils. [Online]. Available: https://github.com/erocarrera/pefile/blob/wiki/PEiDSignatures.md

[25] W. MACHINE. Download userdb.txt. [Online]. Available: http://web.archive.org/web/20160507191641/http://woodmann.com/BobSoft/Download.php?file=Files%2FOther%2FUserDB.zip

[26] pefilenew. Peutils. [Online]. Available: https://www.pydoc.io/pypi/pefilenew-2.2/autoapi/peutils/index.html

[27] VirtualBox. Donwload virtualbox. [Online]. Available: https://www.virtualbox.org/wiki/Downloads

[28] U. de Barcelona, "Tutoriales de instalación y uso de programas de preservación digital. virtualbox, versión 4.1.x. instalacion para windows 7," pp. 1–28. [Online]. Available: http://bd.ub.edu/preservadigital/sites/bd.ub.edu.preservadigital/files/Tutoriales_VirtualBox.pdf

[29] SOFTZone. Cómo crear una máquina virtual con virtualbox paso a paso. [Online]. Available: https://www.softzone.es/2014/10/30/como-crear-una-maquina-virtual-con-virtualbox-paso-paso/

[30] Github. pefile - download. [Online]. Available: https://github.com/erocarrera/pefile

[31] ——. pydasm - python module wrapping libdasm. [Online]. Available: https://github.com/jtpereyda/libdasm/tree/master/pydasm

# Apéndice A

# Instalación de *VirtualBox*

*VirtualBox* puede instalarse en sistemas operativos *Windows* y *Linux*. En este proyecto se instaló en *Windows* y se puede buscar en [27] el archivo instalador el cual existe una descarga para cada sistema operativo y la descarga depende de lo que el usuario desee. Una vez descargado se recomienda seleccionar el icono con el botón derecho del *mouse* y seleccionar la opción "Ejecutar como administrador" del menú que se despliega, esto con el fin de evitar problemas en su instalación, y seguir los pasos mostrados en el manual que se descarga junto con el programa o buscarlo en [28]. Para más detalle para abrir *VirtualBox* como Administrador se puede observar la figura A.1.

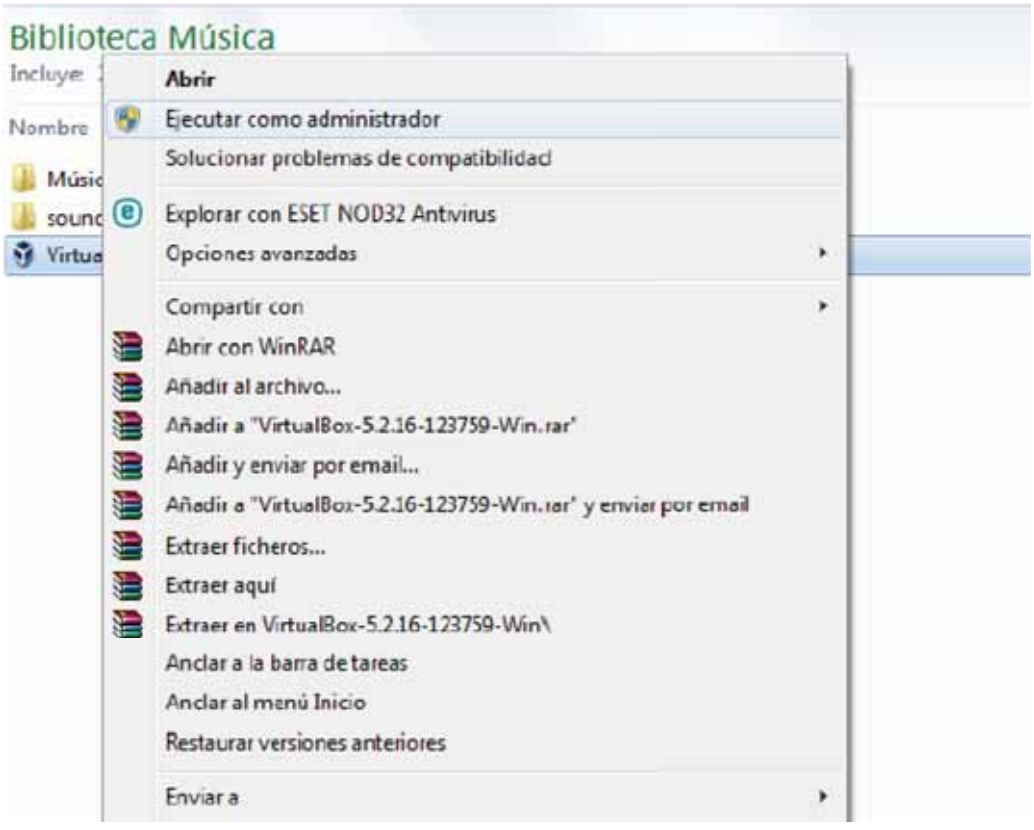



Figura A.1: Recomendación al abrir *VirtualBox*.

# Apéndice A

*VirtualBox*, como se mencionó en el desarrollo, realiza la visualización de un sistema operativo como si fuera una computadora real, esto quiere decir que también necesitara un tiempo para instalar el sistema operativo deseado. Para realizar dicha instalación es necesario descargar una imagen *ISO* el cual contiene una copia exacta de un sistema operativo como *Windows, Linux* entre otros. Estos se pueden encontrar fácilmente en Internet solo que se debe asegurar que la imagen *ISO* que se vaya a descargar sea compatible con el tipo de sistema operativo que la computadora del usuario tenga (32 bits o 64 bits), esto se puede saber fácilmente accediendo a las propiedades de la computadora y una vez que se conozca el tipo de sistema operativo, se sabrá que archivo *ISO* descargar. Para más detalles de cómo crear una máquina virtual consultar [29]

# Apéndice B

# Instalación de *Python*.

La mayoría de sistemas operativos *Linux* actuales ya tienen incluido la versión de *Python* 2.4 o 2.4.5 y muy pocos la versión 3. Este proyecto necesita mínimo tener instalado la versión 3 de *Python* o mayor debido que estas versiones conocen el módulo *pefile* el cual es el principal que se necesita para la elaboración de este proyecto. Este proyecto hizo uso de *Linux Debían* por lo tanto el comando de instalación será para *Debían* el cual es el siguiente:

- ***apt-get install python3***

Con este comando se iniciara el proceso de instalación para este lenguaje de programación. Ya que la instalación termino, verificamos que *Python* se instaló correctamente, esto lo hacemos con el comando ***python*** y obtendremos lo parecido a la figura B.1.

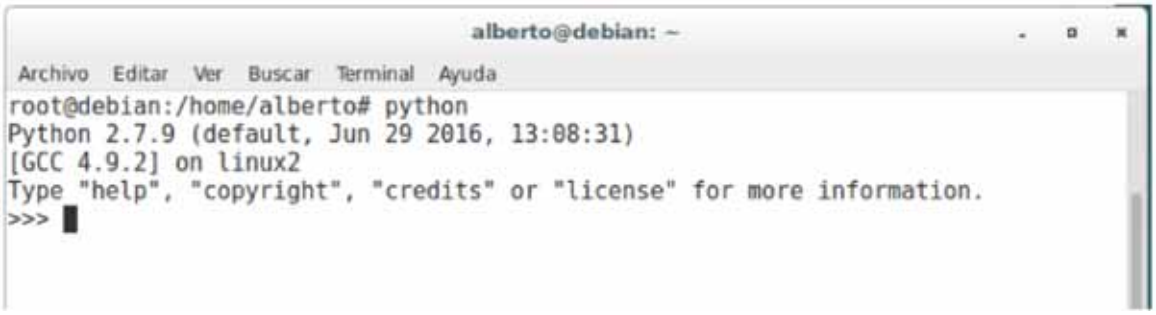



Figura B.1: Verificación de una correcta instalación de *Python*.

*Python* es un lenguaje de programación que tiene su propia línea de comandos, al igual que *Linux*, por lo que los tres símbolos que se observan al último de la figura B.1 es el intérprete de *Python* el cual está esperando a que el usuario escriba un comando y este se ejecute en el momento que se ingresa. Con esto verificamos que la instalación se realizó de manera exitosa.

# B.1. Modulos de *Python*

Procedemos con la instalación de los diferentes módulos de *Python* que se utilizaron en este proyecto los cuales son de dos tipos: los módulos que ya estaban incluidos en *Python* y los que necesitan instalarse aparte.

## B.1.1. Módulos ya incluidos en *Python*

Como el titulo lo indica, estos módulos están permitidos en el momento en que *Python* se instala. Las versiones de *Python* de 3 o superiores ya tienen incluidos estos módulos por lo que no es necesario instalarlos uno por uno. Estos módulos son:

- *time*
- *os*
- *sys*
- *string*
- *re*

Estos módulos son básicos y necesitan importarse al programa para ser utilizados, la importación se hace con el siguiente comando:

***import* "nombre del módulo"**

Remplazando **"nombre del módulo"** por uno de los nombres previamente mencionados en los incisos.

### B.1.2. Módulos que necesitan instalarse aparte.

Estos módulos son más complejos y no se encuentran disponibles al momento de instalar *Python*, esto es debido a que no tiene mucho tiempo en que fueron creados, pero fueron creados con el fin de ser compatibles para *Python*. Los módulos utilizados fueron los siguientes:

- *pefile*
- *pydasm*

Estos módulos, a pesar de que son independientes uno del otro, su instalación es la misma. Se inicia descargando *pefile* y *pydasm* en [30] y [31] respectivamente, ambos se encontraran comprimidos en *.Zip* por lo que solo será cuestión de descomprimirlos para proceder a usarlos.

Estos módulos, una vez ya descomprimidos, contienen un conjunto de archivos necesarios para su correcta instalación y manejo. Entre los archivos se encuentra uno en especial llamado *setup.py* (contenido en cada módulo, es decir hay un archivo *setup.py* por modulo) y ese archivo es el instalador del módulo. Estos módulos solo se pueden instalar desde la línea de comandos de *Linux* por lo que la siguiente sentencia realiza dicha instalación:

***python ruta/del/archivo/setup.pyïnstall***

Con este comando *pefile* y *pydasm* se agregaron a la lista de módulos de *Python* por lo que lo último a realizar son las importaciones respectivas de los mismos y así ya podrán utilizarse en el programa:

***import pefile***
***import pydasm***

# Apéndice C

# Código fuente del algoritmo.

```
1
2  import time
3  import string
4  import os, sys
5  import pydasm
6  import re
7
8  try:
9      import peutils
10     import pefile
11
12 #Asegurarse que tienen las librerias previamente mencionadas instaladas
13 except ImportError:
14     print "pefile no esta instalado, ver http://code.google.com/p/pefile/"
15
16 try:
17      import magic
18 except ImportError:
19     print "python-magic no esta instalado, los tipos de archivos
20     no estaran disponibles"
21
22 try:
23      import yara
24 except ImportError:
25      print "yara-python no esta instalado,
26             ver http://code.google.com/p/yara-proyect"
27
28 ##############################################################################
29 #APIs sospechosos para alertar
30 os.system("clear")
31 alerts = ['OpenProcess', 'VirtualAllocEx', 'WriteProcessMemory',
32           'CreateRemoteThread', 'ReadProcessMemory', 'CreateProcess',
33           'WinExec', 'ShellExecute', 'HttpSendRequest', 'InternetReadFile',
34           'InternetConnect', 'CreateService', 'StartService' ]
35
36
37 print("\n PROGRAMA DE PROYECTO TERMINAL\n")
38 print("analizando archivo PE....\n")
39 file = sys.argv[1]
40 listw = '/home/alberto/Documentos/proyectos_mal/PT/Wordlist.txt'
41 pe = pefile.PE(file)
42
```

```
44 fn = os.path.basename(file)
45 fs = os.path.getsize(file)
46
47 print "Nombre del archivo: " + str(fn)
48 print "Tamanio del archivo: " + str(fs) + " bytes"
49
50
51 sizedata = pe.OPTIONAL_HEADER.SizeOfInitializedData
52 dll = pe.OPTIONAL_HEADER.DllCharacteristics
53 majorversion = pe.OPTIONAL_HEADER.MajorImageVersion
54 checksum = pe.OPTIONAL_HEADER.CheckSum
55 RVA = pe.OPTIONAL_HEADER.NumberOfRvaAndSizes
56
57 print "-----------------------------------------------------------------"
58 ep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
59 ep_ava = ep+pe.OPTIONAL_HEADER.ImageBase
60 data = pe.get_memory_mapped_image()[ep:ep+100]
61 offset = 0
62
63 while offset < len(data):
64       i= pydasm.get_instruction(data[offset:], pydasm.MODE_32)
65       print pydasm.get_instruction_string(i, pydasm.FORMAT_INTEL, ep_ava+offset)
66       offset += i.length
67
68 print "-----------------------------------------------------------------"
69
70
71
72 print "-----------------------------------------------------------------"
73 print "\n RESULTADOS DE DETECCION DE MALWARE EN PUNTOS CLAVE:\n"
74 if (sizedata == 0):
75    print "size of initialized data: POSITIVO.\n"
76    print "valor de SizeOfInitializedData: " + str(sizedata)
77 else:
78    print "size of initialized data: NEGATIVO"
79    print str(hex(sizedata)) + "\n"
80
81 if (dll == 0):
82    print "caracteristicas DLL: POSITIVO\n"
83 else:
84    print "caracteristicas DLL: NEGATIVO"
85    print str(hex(dll)) + "\n"
86
87 if (majorversion == 0):
88    print "version de imagen mayor: POSITIVO\n"
89 else:
90    print "version de imagen mayor: NEGATIVO"
91    print str(hex(majorversion)) + "\n"
92
93 if (checksum == 0):
94    print "checksum: POSITIVO\n"
95 else:
96    print "checksum: NEGATIVO"
97    print str(hex(checksum)) + "\n"
98
99 # segunda comprobacion de checksum
100 print "Verificacion de Checksum:\n"
101 crc_claimed = checksum
102 crc_actual = pe.generate_checksum()
103 resultado = []
```

```
                    crc_claimed, crc_actual, "[sospechoso]"
                    if crc_actual != crc_claimed else ""))
print resultado
print "-------------------------------------------------------------"
###################################
out = []
print "-------------------------------------------------------------"
print "\nSECCIONES:\n"
for sec in pe.sections:
    print "Nombre: " + str(sec.Name)
    print "Entropia: " + str(sec.get_entropy())
    print "SizeOfRawData: " + str(sec.SizeOfRawData)
    print "\n"

print "-------------------------------------------------------------"


#### determinar si un punto de entrada PE es sospechoso ######################
print "-------------------------------------------------------------------------"
name = ""
ep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
pos = 0
array = []
for sec in pe.sections:
    sec.get_entropy()
    if sec.SizeOfRawData == 0 or (sec.get_entropy() > 0 and sec.get_entropy() < 1)
       or sec.get_entropy() > 7:
        estado = "sospechoso"
        sc = str(sec.Name)
        md5 = sec.get_hash_md5()
        sha1 = sec.get_hash_sha1()
        array.append([sc,"md5: " + str(md5),"sha1: " + str(sha1), estado])

if array:
   print "\nsecciones sospechosas: \n"
   print array
else:
   print "\nSECCIONES NO SOSPECHOSAS"
print "\n"
print "-------------------------------------------------------------------------"

print "-------------------------------------------------------------------------"
print "SECCIONES CON NOMBRE NO VALIDO: \n"
seccionesValidas = ['.text', '.code', 'CODE', 'INIT', 'PAGE','.bss', '.reloc',
                    '.data', '.rsrc', '.rdata', '.idata', 'UPX', '.adata',
                    '.aspack', 'DATA']
pos = 0
for sec in pe.sections:
    name = sec.Name.replace('\x00','')
    if (name not in seccionesValidas)or pos == len(pe.sections):
        print "seccion " + str(name) + ": SOSPECHOSO"
    else:
        print "seccion " + str(name) + ": VALIDO"

print "-------------------------------------------------------------------------"

##############################################################
printable = set(string.printable)
def get_process(stream):
    found_str = ""
```

```
        data = stream.read(1024 * 4)
        if not data:
            break
        for char in data:
            if char in printable:
                found_str += char
            elif len(found_str) >= 4:
                yield found_str
                found_str = ""
            else:
                found_str = ""


########################
### checar el analisis de los directorios de datos

try:
    pe.parse_data_directories(directories=[
          pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_IMPORT'],
          pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_EXPORT'],
          pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_TLS'],
          pefile.DIRECTORY_ENTRY['IMAGE_DIRECTORY_ENTRY_RESORUCE']])

except:
    out.append("no puede analisar el archivo (talves no es un PE?)")
    out.append("")

print "-----------------------------------------------------------------"

##CHECAR URL EN EL ARCHIVO PE
print "PETICIONES URL"

arc = open(file, 'rb')
array2 = []
arrayURL = []
arrayFILE = []
arrayFileNames = []

for found_str in get_process(arc):
    fname = re.findall("(.+\.([a-z]{2,3}$))+", found_str, re.IGNORECASE |
            re.MULTILINE)
    if fname:
        word = fname[0][0]
        array2.append(word)

for elem in sorted(set(array2)):
    match = re.search("^http:|^ftp:|^sftp:|^ssh:|^www|.com$|.org$|.it$|
                              .co.uk$|.ru$|.jp$|.net$|.ly$|
                              .gl$|^([0-9]{1,3})(?:\.[0-9]{1,3}){3}$",
                               elem, re.IGNORECASE)


    if match and len(elem) > 6:
        arrayURL.append(elem)
    else:
        arrayFILE.append(elem)

for elem in sorted(set(arrayFILE)):
    file_type = {
        "Video":".3gp",
```

```
        "Video":".asf",
        "Web Page":".asp",
        "Web Page":".aspx",
        "Video":".asx",
        "Video":".avi",
        "Backup":".bak",
        "Binary":".bin",
        "Image":".bmp",
        "Cabinet":".cab",
        "Data":".dat",
        "Database":".db",
        "Word":".doc",
        "Word":".docx",
        "Library":".dll",
        "Autocad":".dwg",
        "Executable":".exe",
        "Email":".eml",
        "Video":".flv",
        "FTP Config":".ftp",
        "Image":".gif",
        "Compressed":".gz",
        "Web Page":".htm",
        "Web Page":".html",
        "Disc Image":".iso",
        "Log":".log",
        "Archive Java":".jar",
        "Image":".jpg",
        "Image":".jepg",
        "Audio":".mp3",
        "Video":".mp4",
        "Video":".mpg",
        "Video":".mpeg",
        "Video":".mov",
        "Installer":".msi",
        "Object":".oca",
        "Object":".ocx",
        "Autogen":".olb",
        "Backup":".old",
        "Registry":".reg",
        "Portable":".pdf",
        "Web Page":".php",
        "Image":".png",
        "Email":".psd",
        "Document":".pub",
        "Compressed":".rar",
        "Text":".rtf",
        "Query DB":".sql",
        "Adobe Flash":".swf",
        "Image":".tif",
        "Temporary":".tmp",
        "Text":".txt",
        "compressed":".tgz",
        "Audio":".wav",
        "Audio":".wma",
        "Video":".wmv",
        "Excel":".xls",
        "Excel":".xlsx",
        "Compressed":".zip"
        }

```

```
288             match = re.search(file_type[descr]+"$", elem, re.IGNORECASE)
289             if match:
290                 arrayFileNames.append([descr, elem])
291 
292 arc.close()
293 
294 if (len(arrayURL) == 0):
295    print "El archivo PE no contiene URLs"
296 else:
297    print "URLs: " + str(arrayURL)
298 print "\n" + str(arrayFileNames) + "\n"
299 
300 print "-----------------------------------------------------------------"
301 
302 ####################################
303 # TIMING
304 def timing(f):
305     def wrap(*args):
306         time1 = time.time()
307         ret = f(*args)
308         time2 = time.time()
309         print '%s Time: %0.3f s' %(f.func_name,float(time2 - time1))
310         return ret
311 
312     return wrap
313 
314 print "-----------------------------------------------------------------"
315 
316 #verificar compresion de archivos PE
317 
318 signatures = peutils.SignatureDatabase('/home/alberto/Documentos/proyectos_mal/
319                                         PT/UserDB.TXT')
320 coincidencias = signatures.match(pe, ep_only = True, section_start_only=False)
321 print coincidencias
322 print "-----------------------------------------------------------------"
323 
324 print "-----------------------------------------------------------------"
325 
326 print "LISTA DE LLAMADAS API"
327 
328 API_LIST = []
329 TPF_LIST = []
330 
331 for entry in pe.DIRECTORY_ENTRY_IMPORT:
332     for API in entry.imports:
333      API_LIST.append(API.name)
334 
335 TPF_LIST.append("CheckSum"+str(pe.OPTIONAL_HEADER.CheckSum))
336 TPF_LIST.append("LoaderFlags"+str(pe.OPTIONAL_HEADER.LoaderFlags))
337 
338 print "lista de llamadas API:\n"
339 print API_LIST
340 
341 print "-----------------------------------------------------------------"
```

Listing C.1: Programa de reporte de información en la detección de Malware

## C.1. Entregables

En el CD se incluira lo siguiente.

- Código fuente del programa.
- Un archivo que contiene un conjunto de identificadores para buscar las firmas del archivo PE llamado *UserDB.txt*.