

# Tema 3a: Análisis estático



Análisis de Malware



Departamento de  
Sistemas de  
Comunicación  
y Control



ETS de  
Ingeniería  
Informática

# Índice

- Introducción
- Tipo de archivo
- Huella digital (*fingerprinting*)
- Escaneo con antivirus
- Extracción de strings
- Otras técnicas de ofuscación
- Información de cabecera
- Comparativa y clasificación de malware
- YARA

# Introducción (I)

- El análisis estático consiste en analizar un archivo sospechoso sin ejecutarlo.
- Es un análisis inicial que implica:
  - Extracción de información útil del archivo sospechoso.
  - Decisión informada de cómo clasificarlo
  - Dónde enfocar los siguientes esfuerzos de análisis

# Tipo de archivo (I)

- Nos permite identificar el Sistema Operativo o la arquitectura objetivos del *malware*.
- Aunque se suele ocultar la extensión para disimular la acción del *malware*, la firma de archivo (*file signature*) es útil para determinar dicho tipo de archivo.

# Tipo de archivo (II)

## ○ Método manual:

- Abrir el archivo con un editor hexadecimal.
- Archivos ejecutables en Windows presentan los bytes 4D 5A (MZ) al inicio.
- Ejemplos de editores:
  - HxD Hex editor
  - Comando xxd (Linux)

## ○ Otras herramientas:

- Utilidad “file” (Linux)
- CFF Explorer (Windows)

# Tipo de archivo (III)

## ○ Análisis con Python:

- Módulo *python-magic*: <https://github.com/ahupp/python-magic>

```
>>> import magic
>>> m = magic.open(magic.MAGIC_NONE)
>>> m.load()
>>> ftype = m.file(r'log.exe')
>>> print ftype
PE32 executable (GUI) Intel 80386, for MS
Windows
```

# Huella digital (*fingerprinting*) (I)

- El concepto *fingerprinting* implica la generación de valores *hash* criptográficos para un archivo sospechoso, a partir de su contenido.
  - Algoritmos como MD5, SHA1 o SHA256.
  - Sirven como identificador único de un archivo (y su contenido).
  - Detectar copias de archivos maliciosos durante el análisis dinámico.
  - Indicador a compartir con otros investigadores y analistas de seguridad.
  - Determinar si la huella digital está marcada como peligrosa en una base de datos de antivirus.

# Huella digital (*fingerprinting*) (II)

## ○ Generación de la huella:

- Linux: comandos *md5sum*, *sha256sum*, *sha1sum*
- Windows: utilidades como *HashMyFiles*
- Python: librería *hashlib*

```
>>> import hashlib
>>> content = open(r"log.exe", "rb").read()
>>> print hashlib.md5(content).hexdigest()
6e4e030fb2ee786e1b6b758d5897316
>>> print hashlib.sha256(content).hexdigest()
01636faaae739655bf88b39d21834b7dac923386d2b52efb41
42cb278061f97f
>>> print hashlib.sha1(content).hexdigest()
625644bacf83a889038e4a283d29204edc0e9b65
```

# Escaneo con Antivirus (I)

- Búsqueda de huellas digitales en bases de datos de Antivirus
  - VirusTotal:
    - Búsquedas por código *hash*, dominio, URL, o direcciones IP.
    - Visualización con *VirusTotal Graph*
    - API pública:
      - Generación de clave pública (al darse de alta en *VirusTotal*).
      - Uso programático (por ejemplo, Python)
      - Programas para el análisis de archivos *portable executable* (PE):
        - PEStudio, o *PPEE*

# Escaneo con Antivirus (II)

- Factores de riesgo:
  - Si el archivo sospechoso no es detectado por el antivirus, no implica necesariamente que no sea malicioso (modificación de código con técnicas de obfuscación).
  - Archivos con información sensible, personal o propietaria: no recomendable realizar un análisis en crudo, mejor analizar directamente su código *hash*.
  - Los atacantes también pueden comprobar si su archivo malicioso ya está almacenado como sospechoso para modificar sus técnicas y evitar que se detecte.

# Extracción de strings (I)

- Búsqueda de cadenas de caracteres sospechosas dentro de un archivo.
- Pistas sobre su funcionalidad (no necesariamente implican peligrosidad).
- Ejemplos:
  - Nombres de archivos creados por un ejecutable sospechoso.
  - Nombres de dominios a los que accede el programa.
  - URLs, direcciones IP, comandos de ataque, claves de registro, etc.

# Extracción de strings (II)

## ○ Herramientas:

- Comando *strings* en Linux: opción –a (strings ASCII), opción –el (Unicode).
- Herramienta PEStudio en Windows
- Decodificación de strings ofuscados:
  - *FireEye Labs Obfuscation String Solver (FLOSS)*
  - Para Linux y Windows
  - <https://www.fireeye.com/blog/threat-research/2016/06/automatically-extracting-obfuscated-strings.html>

# Otras técnicas de ofuscación (I)

- **Packers:**
  - Programas que utilizan técnicas de compresión para ocultar el contenido del ejecutable.
  - El resultado es un nuevo ejecutable que se autodescomprime al ser ejecutado.
  - Herramientas: UPX (<https://upx.github.io/>)
- **Cryptors:**
  - Similar al packer, pero utiliza encriptación en lugar de compresión.
  - Se desencripta al ser ejecutado.

# Otras técnicas de ofuscación (II)

- **Detección de ofuscación con Exeinfo PE:**
  - Detección de packers con Windows.
  - Más de 4,500 huellas digitales para detectar compiladores, packers y cryptors utilizados para ofuscar un programa.
  - Información y referencias para deshacer la ofuscación.
- **Otras herramientas:**
  - TrID
  - TRIDNet
  - Detect It Easy
  - RDG Packer Detector
  - packerid.py
  - PEiD

# Información de cabecera (I)

- Los ejecutables en Windows siguen un formato *PE/COFF* (*Portable Executable/Common Object File Format*).
- Las cabeceras de los PE contienen información sobre dónde cargar el ejecutable en memoria, los recursos de los que depende o las librerías y funciones que utiliza.
- Herramientas para su análisis:
  - PEStudio
  - PPEE
  - CFF Explorer
  - PE Internals
  - PEBrowse Professional

# Información de cabecera (II)

## ○ Dependencias e *Imports*:

- Los archivos ejecutables importan librerías DLL (en Windows) que proporcionan funcionalidad determinada.
- Inspeccionar estas DLLs puede darnos información sobre la peligrosidad de un ejecutable.
  - Si un archivo tiene muy pocos *imports*, hay muchas probabilidades de que esté tratado con un *packer*.
  - Si el DLL es llamado durante el tiempo de ejecución no aparecerá en la cabecera y por tanto no podrá ser detectado.
- Herramientas como *PEstudio* o librerías python como *pefile*.

# Información de cabecera (III)

## ○ Dependencias e Imports (ejemplo PEStudio):

symbol (110)	blacklisted (110)	anony...	anti-debug (110)	library (7)
connect	x	-	-	wsock32.dll
gethostbyaddr	x	-	-	wsock32.dll
gethostbyname	x	-	-	wsock32.dll
getpeername	x	-	-	wsock32.dll
getsockname	x	-	-	wsock32.dll
htonl	x	-	-	wsock32.dll
htons	x	-	-	wsock32.dll
inet_addr	x	-	-	wsock32.dll
inet_ntoa	x	-	-	wsock32.dll
ioctlsocket	x	-	-	wsock32.dll
listen	x	-	-	wsock32.dll
ntohs	x	-	-	wsock32.dll
recv	x	-	-	wsock32.dll
select	x	-	-	wsock32.dll
send	x	-	-	wsock32.dll
socket	x	-	-	wsock32.dll

# Información de cabecera (IV)

## ○ *Exports:*

- Los ejecutables y las DLLs también pueden exportar funciones.
- Los atacantes suelen crear DLLs que exportan funciones maliciosas, para ser cargadas por un proceso externo.
- La inspección de *exports* puede dar una idea de las posibilidades que proporciona una DLL.
- Herramientas como *PEstudio* o librerías python como *pefile*.

# Información de cabecera (V)

## ○ Tabla de secciones:

Sección	Descripción
.text o CODE	Código ejecutable
.data o DATA	Datos de lectura/escritura y variables globales
.rdata	Datos de sólo lectura. A veces también información de importación y exportación.
.idata	Información de <i>imports</i> (si no existe la sección, la tabla está en .rdata)
.edata	Información de <i>exports</i> (si no existe la sección, la tabla está en .rdata)
.rsrc	Recursos del ejecutable (iconos, menús, cuadros de diálogo, strings...)

# Información de cabecera (VI)

## ○ Campos de una sección:

Campo	Descripción
Names	Nombre de las secciones
Virtual-size	Tamaño de la sección cargada en memoria
Virtual-address	Dirección relativa ( <i>offset</i> desde la dirección base) de la sección en memoria
Raw-size	Tamaño de la sección en disco
Raw-data	Dirección absoluta ( <i>offset</i> en el archivo donde se encuentra la sección)
Entry-point	Dirección relativa virtual ( <i>Relative Virtual Address, RVA</i> ) donde comienza la ejecución de código. Suele ser la sección .text

# Información de cabecera (VII)

## ○ **Discrepancias en secciones:**

- Los nombres de secciones no son los comunes que añade un compilador.
- El punto de entrada (Entry-point) es una sección distinta de .text, lo cuál suele indicar una rutina de descompresión (packers).
- Los tamaños Raw-size y Virtual-size suelen ser similares, con pequeñas diferencias debidas a la alineación de secciones. Una diferencia importante (Virtual-size elevado) suele indicar que el archivo ha sido tratado con un packer.

# Información de cabecera (VIII)

- ***Timestamp de compilación:***
  - Información de cuándo fue compilado el ejecutable.
  - Puede ser útil para analizar la línea de tiempo de una campaña de ataques.
  - Puede ser modificada por un atacante (por ejemplo, a una fecha futura): aunque la fecha real no puede ser detectada, indica un comportamiento anómalo.

# Información de cabecera (IX)

- **Recursos del PE:**
  - Información almacenada en la sección .rsrc (iconos, menús, diálogos, strings...)
  - Información del origen, nombre de la compañía, autor del programa, información de copyright...
  - A veces se almacena información como binarios adicionales o señuelos.
  - Herramientas como *Resource Hacker* (<http://www.angusj.com/resourcehacker>)

# Comparativa y clasificación de malware (I)

## ○ **Fuzzy hashing:**

- Técnica para comparar ejecutables sospechosos y extraer porcentajes de similitud con malware previamente detectado.
- Herramienta ssdeep (<http://ssdeep.sourceforge.net>). También con librería python (ssdeep).
- Identifica similitud aunque los códigos hash exactos de cada muestra sean distintos.

# Comparativa y clasificación de malware (II)

## ○ **Import hashing:**

- Técnica en la que se extrae el código hash a partir de las librerías importadas por el ejecutable y su orden.
- Si dos archivos presentan el mismo *imphash* (*import hash*), es muy probable que estén relacionados (hayan sido compilados de la misma fuente).
- Herramientas como *PEstudio* o librerías python como *pefile*.

# Comparativa y clasificación de malware (III)

## ○ **Section Hashing:**

- Similar a *import hashing*.
- Se utiliza el código hash (MD5) calculado para cada sección.
- Herramientas como *PEstudio* o librerías python como *pefile*.

# YARA (I)

- Herramienta proporcionada por VirusTotal: <http://virustotal.github.io/yara>
- Reglas basadas en información textual o binaria de muestras malware.
- Las reglas consisten en strings + expresión booleana.
- Librería python: yara-python

# YARA (II)

- **Conceptos básicos de reglas:**
  - Identificador de la regla: nombre de la regla.
  - Definición de strings: se puede omitir si la regla no se basa en strings. Cada string se compone de un identificador (nombre) y un valor alfanumérico.
  - Sección de condiciones: especifica qué condiciones se tienen que cumplir para que la regla se cumpla o no.

# YARA (III)

## ○ Ejemplos (I):

```
rule suspicious_strings
{
    strings:
        $a = "Synflooding" ascii wide nocase
        $b = "Portscanner" ascii wide nocase
        $c = "Keylogger"   ascii wide nocase
    condition:
        ($a or $b or $c)
}
```

# YARA (IV)

## ○ Ejemplos (I)

- La regla “suspicious\_strings” busca los strings “Synflooding”, “Portscanner” y “Keylogger” en el binario, y si encuentra al menos uno de ellos se considera como cumplida.
- Busca tanto strings ASCII como Unicode (opción “wide”), y no tiene en cuenta las mayúsculas (opción “nocase”).

# YARA (V)

## ○ Ejemplos (II):

```
rule suspicious_strings
{
    strings:
        $mz = { 4D 5A}
        $a = "Synflooding" ascii wide nocase
        $b = "Portscanner" ascii wide nocase
        $c = "Keylogger" ascii wide nocase
    condition:
        ($mz at 0) and ($a or $b or $c)
}
```

# YARA (VI)

- Ejemplos (II):
  - Añadimos  $\$mz = \{4D\ 5A\}$  para asegurar que la regla se aplica únicamente a archivos ejecutables (PE).
  - El operador booleano obliga a que se cumpla dicha condición (en la dirección 0) y además se encuentre alguno de los strings especificados.

# YARA (VII)

## ○ Ejemplos (III):

```
rule embedded_office_document
{
meta:
description = "Detects embedded office document"

strings:
    $mz = { 4D 5A }
    $a = { D0 CF 11 E0 A1 B1 1A E1 }
condition:
    ($mz at 0) and $a in (1024..filesize)
}
```

# YARA (VIII)

## ○ Ejemplos (III):

- Detecta un documento Excel embebido.
- Observa si el archivo es ejecutable, y busca un string hexadecimal que se corresponde con Excel.
- La búsqueda se realiza entre el *offset* (1024), para saltarse la cabecera, y el final del documento (*filesize*).

# YARA (IX)

## ○ Ejemplos (IV):

```
rule mal_digital_cert_9002_rat
{
meta:
    description = "Detects malicious digital
certificates used by RAT 9002"
    ref = "http://blog.cylance.com/another-9002-
trojan-variant"

strings:
    $mz = { 4D 5A }
    $a = { 45 6e 96 7a 81 5a a5 cb b9 9f b8 6a ca
8f 7f 69 }

condition:
    ($mz at 0) and ($a in (1024..filesize))
}
```

# YARA (X)

## ○ Ejemplos (IV):

- Detecta un malware específico denominado RAT9002 a partir de su número de serie asociado a su certificado digital.
- La búsqueda se realiza entre el *offset* (1024), para saltarse la cabecera, y el final del documento (*filesize*).

# Tema 3a: Análisis estático



Análisis de Malware



Departamento de  
Sistemas de  
Comunicación  
y Control



ETS de  
Ingeniería  
Informática