Universidad Carlos III

Ingeniería de la Ciberseguridad

Curso 2024-25

Práctica 3

Análisis de Malware

Ingeniería Informática, Cuarto curso

Adrián Fernández Galán (NIA: 100472182, e-mail: 100472182@alumnos.uc3m.es)
César López Mantecón (NIA: 100472092, e-mail: 100472092@alumnos.uc3m.es)
Manuel Gómez-Plana Rodríguez (NIA: 100472310, e-mail: 100472310@alumnos.uc3m.es)

Prof. Antonio Nappa
Grupo: 81

1. Introducción

Este documento recoge el desarrollo de la tercera práctica de la asignatura *Ingeniería de la Ciberseguridad*. El objetivo de esta práctica es el análisis y comprensión de dos programas maliciosos para extraer una *flag* de cada uno. Para ello trabajaremos con una máquina virtual de KaliLinux totalmente aislada con el fin de poder analizar los binarios tanto estáticamente como dinámicamente.

Los malwares son softwares con alguna intención maliciosa que puede poner a un usuario, sus datos o un dispositivo en riesgo. Para esta práctica, será fundamental identificar el comportamiento malicioso de cada uno de los softwares y comprender su funcionamiento. Además de extraer las *flags* con el formato adecuado.

Adicionalmente, es importante tener en cuenta que contamos con información acerca del autor de estos programas. Es por esto que centraremos gran parte de nuestros esfuerzos en la identificación de cadenas donde un prefijo y un sufijo se concatenen a una **semilla**, siendo esta típicamente una palabra seguida de algún caracter repetido sobre la que se ha aplicado alguna clase de cifrado de desplazamiento en un alfabeto, comúnmente el alfabeto de caracteres hexadecimales ("abcdef0123456789").

2. Parte 1

El primer malware se trata de un programa que se ejecuta sobre un directorio. Contamos con el código fuente, así como el binario ejecutable. Es importante destacar que este último no cuenta con marcas de depuración, lo que dificulta su análisis mediante decompiladores y entornos de *debugging*.

Durante su análisis se emplearán las herramientas Ghidra y gdb, además de otros métodos propios.

2.1. Análisis estático

Durante este proceso se ha tratado de comprender del funcionamiento del programa, extraer el flujo habitual del código e identificar secciones o funciones clave en su ejecución. Para ello se ha empleado *Ghidra* para decompilar y comparar el pseudocódigo resultante con el código C y verificar que ambos programas tienen el mismo comportamiento.

Finalmente se concluyó que el código l3.c y el ejecutable `runDir` son el mismo programa. Tras su análisis se extrajeron las siguientes conclusiones:

- El programa realiza una llamada `fork` con el fin de evitar el uso de herramientas como `gdb` u otros entornos de debugging.
- El programa genera una *flag* de longitud pseudoaleatoria y de caracteres al azar mediante la concatenación de la semilla rotada "srrqnn" con un prefijo y un sufijo, también generados aleatoriamente. Esta flag se transforma mediante las funciones `transform_flag` y `process_buffer`.
- El programa procesa archivos con las extensiones .txt, .pdf, .jpg, .png y .doc; creando una copia y aplicando una transformación descrita en la función `process_buffer`. A estos archivos transformados les inserta la `flag` transformada al final.
- Las funciones para transformar archivos o explorar directorios reciben una estructura donde se encuentra la *flag* sin transformar y transformada.

Con todo lo anterior es fácil concluir que se trata de un programa que cifra u ofusca archivos con extensiones concretas, creando una copia de seguridad previamente. Destaca que trabaja siempre con copias de los archivos, nunca con los originales. También, crea un archivo *processed_files.txt* donde registran todos los archivos que se han procesado y se incluye la *flag transformada*.

2.2. Análisis dinámico

El análisis dinámico ha permitido la extracción de las *flags* transformada y sin transformar. Para ello ha sido necesario sortear la llamada a la función `fork` para que el padre procese el directorio, al igual que el hijo, y permitir su análisis con la herramienta `gdb`. Esto se ha logrado mediante el siguiente código:

```
(gdb) break fork@plt
(gdb) run
(gdb) finish
(gdb) set $rax=0
```

Listing 1: Instrucciones en gdb para esquivar la llamada fork

De esta forma el proceso padre recibirá un 0 como resultado del fork, ejecutando el flujo habitual del proceso hijo. Con esto podemos extraer la *flag* a través de las siguientes instrucciones:

```
(gdb) break process_directory
(gdb) finish
(gdb) x/s $rsi
(gdb) x/s $rsi + 64
```

Listing 2: Instrucciones en gdb para extraer la flag

Con la primera instrucción `x/s` podemos extraer la semilla antes de ser transformada. Con la segunda, vemos el resultado de la transformación. Al repetir esto varias veces, hemos comprobado que la *flag* no cambia entre ejecuciones, lo que ha levantado sospechas sobre la aleatoriedad de la función `rand`. Al buscar información en el manual y realizar una prueba en 3 máquinas distintas con diferentes sistemas operativos se ha confirmado que la secuencia que genera esta función es siempre la misma, haciendo de la generación de la *flag* un proceso determinista. Para cualquier ejecución, la cadena de caracteres tendrá una longitud de 26, donde el prefijo y sufijo serán de longitud 10.

Complementariamente, se han analizado los ficheros generados por el ejecutable. Esto ha permitido comprobar que, efectivamente, la flag escrita en el fichero *processed_files.txt*, la extraída durante la ejecución y la flag escrita al final de cada fichero es la misma. Se puede extraer la flag al final del fichero a través del siguiente comando:

```
tail -c 26 file.processed # 881531cc331308c2ca534a15c4
```

Listing 3: Obtención de la flag transformada concatenada al final del fichero

2.3. Descripción del Malware

El malware se trata de un cifrador de ficheros, lo que encaja con la descripción de un *ransomware*. Sin embargo, tiene ciertas particularidades destacables. Lo primero, crea una copia de cada fichero en el propio directorio, lo que no encaja del todo con el comportamiento típico de este tipo de malware. También, no trabaja sobre el archivo original, cifrando una copia. Esto se traduce en que, si una víctima ejecuta el binario en su máquina, no existirían daños reales sobre sus datos u archivos. No obstante, si se tiene en cuenta la naturaleza académica de este ejercicio, podemos ignorar estas particularidades y reconocer el potencial daño que puede tener un software de este tipo: la pérdida de acceso a ciertos ficheros en un directorio.

2.4. Descubrimiento de la flag

Para el descubrimiento de la *flag* ha sido necesario rotar la semilla descubierta. Dado que no cuenta con caracteres hexadecimales se ha empleado un *script* de `Python` (ver anexo) para rotarla en el alfabeto inglés buscando encontrar el patrón descrito en la introducción.

```
$ python rotate_seed.py
0   srrqnn
1   tssroo
2   uttspp
3   vuutqq
4   wvvurr
5   xwwvss
6   yxxwtt
7   zyyxuu
8   azzyvv
```

```
9   baazww
10  cbbaxx
11  dccbyy
12  eddczz
13  feedaa
14  gffebb
15  hggfcc
16  ihhgdd
17  jiihee
18  kjjiff
19  lkkjgg
20  mllkhh
21  nmmlii
22  onnmjj
23  poonkk
24  qppoll
25  rqqpmm
```

Listing 4: Resultado de rotar la semilla en el alfabeto inglés

De este resultado, la única semilla que cumple la condición de ser una palabra seguida de dos letras es *feedaa*. Adicionalmente cuenta exclusivamente con caracteres hexadecimales, lo que la convierte en la única candidata viable para ser la semilla. Adicionalmente, se han probado las rotaciones de ejercicios pasados sin resultado exitoso.

Finalmente, conociendo la longitud y ubicación de la semilla gracias al análisis estático se ha conseguido entregar existosamente la semilla: `6931FAC9DAFEEDAAB2B36C248B`. Esta semilla se ha generado replicando el código de la función `generate_flag`, eliminando la ofuscación y empleando la semilla en claro.

3. Parte 2

El segundo malware es una aplicación móvil desarrollada para sistemas *Android* a través de Kotlin. En el directorio raíz encontramos dos ficheros con la extensión *.kt*: *FileProcessor.kt* *MainActivity.kt*. Además encontramos un archivo *.apk*.

3.1. Análisis estático

Se ha descomprimido el fichero *.apk*, obteniendo varios ficheros compilados con la extensión *.dex*. Gracias a la herramienta Jadx hemos podido decompilar estos ficheros y analizar el código del programa.

Entre todos los binarios, destaca el fichero *classes3.dex*, donde están contenidas funciones como `generate_flag`. Adicionalmente, mirando el conjunto completo de los ficheros el programa tiene un comportamiento similar al de la parte anterior, destacando el uso del objeto `SecureRandom()` de `Java` para la generación de números aleatorios. Este método, al contrario que en la primera parte, si que genera números aleatorios para distinas ejecuciones.

El proceso de generación de la *flag* es similar al de la parte anterior. Una semilla *hardcoded* es concatenada a un prefijo y sufijo de longitudes aleatorias. La semilla encontrada en el código es "srrrss".

Con todo lo anterior se concluye que se trata de un software con un comportamiento similar al de la parte anterior, adaptado a la plataforma de android.

3.2. Análisis dinámico

Se ha utilizado un móvil aislado de la red para la ejecución del software y la observación de su comportamiento. Se observa que se obtienen *flags* distintas en cada ejecución (ver figura 1). Sin embargo, no se perciben efectos nocivos a los datos almancenados en el dispositivo. Destaca que se ha analizado el programa con un antivirus, sin levantar ninguna alarma.

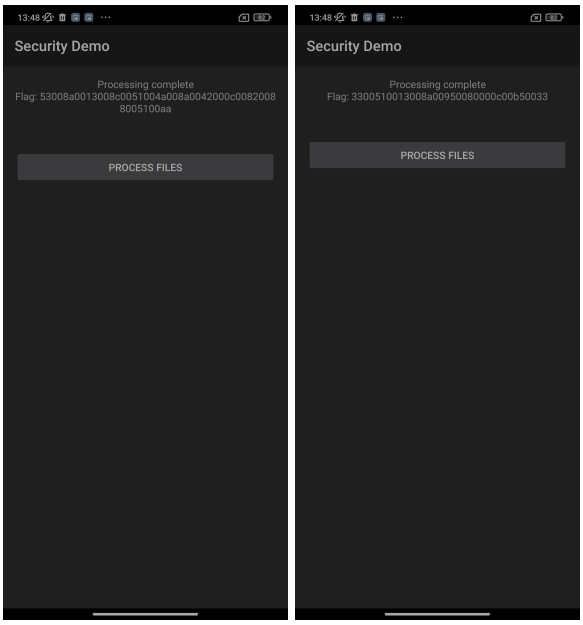

Figura 1: Dos ejecuciones sobre un dispositivo android

Destaca que, además de la instalación, el usuario debe darle permisos sobre el sistema de ficheros para que el software pueda ejecutarse. Es decir, se necesita interacción del usuario para la ejecución del malware.

3.3. Descripción del Malware

De la misma forma que en la parte anterior, el análisis dinámico apunta a que se trata de un *ransomware*. No obstante, la ejecución sobre un dispositivo android no tiene como consecuencia ningún efecto observable.

3.4. Descubrimiento de la flag

Para el descubrimiento de la *flag* ha sido necesario tanto las conclusiones extraídas del análisis estástico como el análisis del código fuente incluído el archivo *FileProcessor.kt*. En este último aparece un comentario donde se incluye la forma en la que se ha obtenido la semilla incluída en el binario, afirmando que se trata de un desplazamiento de 13 posiciones sobre la palabra "feedff". No obstante, si se aplica este proceso a la inversa sobre la cadena que se usa como semilla se obtiene la palabra "feeeff". Ante esta situación, se ha seguido el mismo proceso de la parte anterior obteniendo dos *flags* candidatas: `3850413feedffb6b28ff359c466459e` y `3850413feeeffb6b28ff359c466459e`.

Tras probar ambas, la *flag* que ha sido aceptada en el envío ha sido la primera: `3850413feedffb6b28ff359c466459e`.

Es importante destacar que esta semilla es la única de las dos que sigue el patrón de prácticas anteriores. Sin embargo, en el ejecutable la *flag* resultante proviene de rotar la cadena "feeeff". Esto resalta la importancia de contar con el código fuente a la hora de analizar software malicioso, siendo un elemento que facilita en gran medida el trabajo.

4. Conclusiones

El análisis estático y dinámico de archivos binarios es una de las principales actividades relacionadas con la ciberseguridad. Esta práctica nos ha permitido explorar el análisis estático mediante el decompilado de archivos binarios, estudiando en el proceso las heerramientas más famosas de este campo, como `ghidra` o `jadx`.

Este análisis nos sirvé también para mejorar nuestra capacidad de análisis de código, permitíendonos discernir si un código es malicioso o no, algo altamente crucial en la ciberseguridad. Así,

valoramos está práctica ya que nos ha brindado la oportunidad de conocer herramientas de decompilado y la oportunidad de practicar el análisis de código mediante la búsqueda de la intención del malware así como la *flag* en claro.

Anexo

Código para la rotación de la flag en el alfabeto inglés

```
seed = "srrqnn"
prefix = "6931fac9da"
sufix = "b2b36c248b"

for i in range(26):
    b = ""
    for j in seed:
        b += chr(((ord(j) - 97 + i) % 26) + 97)
    print(f'{i}\t{prefix + b + sufix}')
    #print(f'{i}\t{b}')
```