Ingeniería de la Ciberseguridad G81

PRÁCTICA 3: Malware Analysis

22-Diciembre-2024

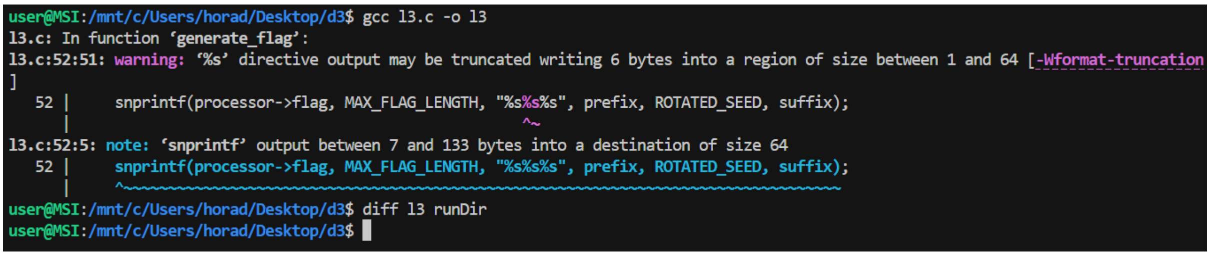
Prof: Antonio Nappa
Nombre del grupo: **GrupoAJA**

Alejandro Isla Álvarez (**100472228**)
József Iván Gafo(**100456709**)
Ángel Pérez Navas (**100472200**)

Flag type 1 3
Análisis de Código malicioso en la flag 1 4
Flag type 2 5
Análisis de Código malicioso en la flag 2 6

Flag type 1

Esta flag se puede obtener a partir del binario **runDir**. Para analizar su funcionalidad y su comportamiento podemos descompilar el binario usando herramientas como Binary Ninja. Sin embargo, dado que se proporciona el código fuente **l3.c*,*** podemos utilizarlo para un entendimiento más sencillo. Es importante comprobar que el código fuente corresponde con el binario, para lo que compilamos **l3.c** y lo comparamos con **runDir** usando el comando **diff**, comprobando así que son idénticos.

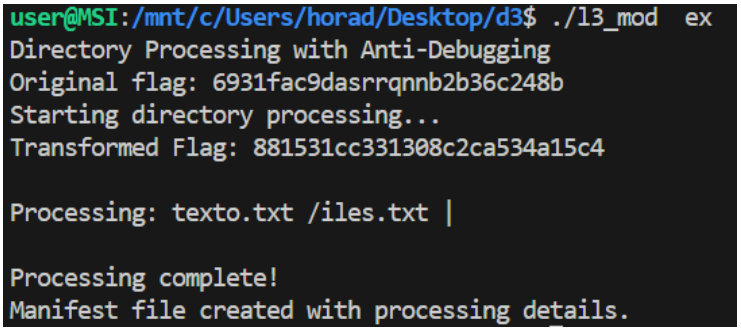


```
user@MSI:/mnt/c/Users/horad/Desktop/d3$ gcc l3.c -o l3
l3.c: In function ‘generate_flag’:
l3.c:52:51: warning: ‘%s’ directive output may be truncated writing 6 bytes into a region of size between 1 and 64 [-Wformat-truncation
]
   52 |     snprintf(processor->flag, MAX_FLAG_LENGTH, "%s%s%s", prefix, ROTATED_SEED, suffix);
      |                                                   ^~
l3.c:52:5: note: ‘snprintf’ output between 7 and 133 bytes into a destination of size 64
   52 |     snprintf(processor->flag, MAX_FLAG_LENGTH, "%s%s%s", prefix, ROTATED_SEED, suffix);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
user@MSI:/mnt/c/Users/horad/Desktop/d3$ diff l3 runDir
user@MSI:/mnt/c/Users/horad/Desktop/d3$
```

En términos generales el programa inicialmente crea un proceso hijo que será monitorizado por el padre usando **ptrace**, e impidiendo monitorizaciones externas por nuestra parte. El proceso hijo lleva a cabo dos tareas:

- Genera una flag de longitud aleatoria de entre 16 y 32 caracteres, con un prefijo y sufijo aleatorio, y una seed fija entre medias. Cabe destacar que el uso de la función **rand()** sin una semilla aleatoria produce siempre la misma longitud, prefijo y sufijo, por lo que la flag generada es siempre la misma. Posteriormente transforma la flag original mediante unas operaciones aritméticas de dos en dos bytes, y alguna ofuscación.
- La segunda tarea principal recorre el directorio dado como argumento al programa. Para cada fichero con una extensión determinada, crea una copia exacta (.backup) y una copia transformada (.processed) que contiene el contenido del fichero transformado de la misma forma que la flag y la flag transformada concatenada. También se crea un archivo **processed_files.txt** con información del procedimiento.

La flag transformada que se guarda en los ficheros transformados y que se muestra por consola no es la que buscamos, sino que será, probablemente, la flag original sin transformar. Una opción podría ser revertir el proceso de transformación mediante operaciones inversas. Sin embargo, sabiendo que el código fuente corresponde al binario podemos simplemente modificarlo para incluir un **printf** de la flag original y ejecutarlo con un directorio de prueba en un entorno seguro. También se podrían considerar otras alternativas como eliminar la monitorización con ptrace y hacer un análisis dinámico para encontrar la flag, si bien no es necesario. Una vez que tenemos la flag original solo es necesario deshacer la rotación ROT13 a la seed, y comprobar que la flag resultante es, en efecto, válida.



Ejecución del binario modificado para que muestre la flag original

Análisis de Código malicioso en la flag 1

Vamos a analizar en profundidad el programa **l3.c** y vamos a observar qué cualidades pueden hacer que este programa pueda llegar a ser sospechoso.

1. **Evita la depuración con técnicas avanzadas**.
   Vemos que se usa *ptrace* y se monitorean registros de depuración (*u_debugreg*). El programa es capaz de detectar si está siendo depurado mediante el uso de *ptrace* y puede verificar posibles registros de depuración del sistema.
   Si se detecta depuración, el programa mata el proceso con **SIGKILL**, esta técnica es utilizada por el **malware para dificultar el análisis por parte de los investigadores**.
   Además, se verifican cambios en el PID del padre (*check_parent_status* para comprobar el PPid), se comprueba si el proceso padre, es el mismo que al inicio para así comprobar si hay posibles herramientas de depuración.

2. **Altera el contenido mediante transformaciones.**
   Este programa procesa archivos como **.txt**, **.doc**, **.pdf**, **.jpg**, **.png**, realizando copias de seguridad (*.backup*) en un directorio y luego aplica una serie de transformaciones al contenido con la función *Mixed Boolean Arithmetic*. Después de esto, escribe una flag transformada al final de cada archivo, esto puede ser usado para **corromper datos legítimos y poder ocultar posible información maliciosa**.

3. **Encripta y genera datos aleatorios para modificar archivos.**
   Cuando el programa usa la función *mba_transform()*, es capaz de encriptar contenido de manera bastante complicada para luego revertir. El programa genera datos que parecen aleatorios y así modifica los archivos (añade contenido que antes no estaba en el archivo). Esta técnica es muy usada por los atacantes para **implantar identificadores o malware en los archivos que han sido procesados**.

4. **Evita el análisis y la protección contra el debugging**.
   El programa usa técnicas como el proceso *raise(SIGCONT)* para necesitar un *tracer* para poder continuar. Además, configura el *ptrace* para terminar el proceso si el *tracer* ejecuta *PTRACE_O_EXITKILL*. Además, puede llegar a protegerse del debugging con la constante *ROTATED_SEED*, ya que esta puede ser **utilizada como marcador para evitar las modificaciones**.

5. **Puede corromper datos.**
   El programa es capaz de procesar archivos como fotos y documentos como hemos mencionado anteriormente, pudiendo así **encriptarlos bloqueando así su acceso**.

6. **Usa técnicas de seguimiento sospechosas**. se puede observar que se implementa un flujo donde el padre actúa como *tracer* y el hijo como proceso que monitoriza. Esto es raro en un programa y es típico de un **comportamiento encubierto**.

Flag type 2

Esta flag se obtiene a partir del apk de Android **ransandr.apk**. En este caso resulta más complicado verificar que los archivos .kt proporcionados corresponden al código fuente de la aplicación. Sin embargo, usando la herramienta **jadx** obtenemos algunos ficheros .jar en java a partir de **classes3.dex** que muestran una funcionalidad aparentemente idéntica, por lo que asumimos como válidos los ficheros proporcionados.

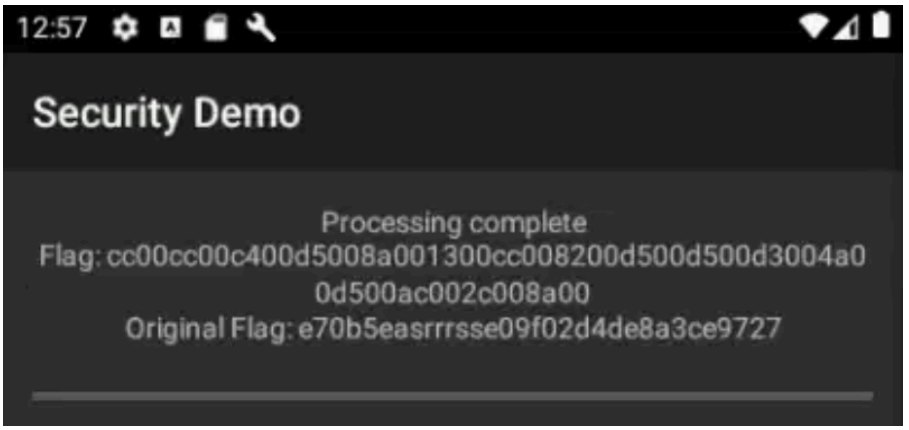
El funcionamiento de la aplicación es casi idéntico al primer caso:

- Se genera una flag de longitud, prefijo y sufijo aleatorios (esta vez sí son verdaderamente aleatorias) con una seed fija entre medias (se da la pista como comentario que esta ha sido rotada con ROT13). Después se transforma la flag igual que antes.
- El proceso de los archivos del directorio es el mismo, pero en este caso se toma como directorio el del almacenamiento externo, que en versiones modernas de Android suele corresponder al de **/storage/emulated/0/**.

La flag transformada que se muestra tras procesar el directorio no es la queremos, sino que vuelve a ser la flag original sin transformar con ROT13 de la seed. Para obtenerla se ha seguido el procedimiento de reconstruir la aplicación y modificarla:

- Creamos un nuevo proyecto en Android Studio y utilizamos como base los ficheros **MainActivity.kt** y **FileProcessor.kt** proporcionados.
- Surgen errores al faltar los recursos de la carpeta **res** y al no tener el **AndroidManifest.xml** requerido. Los ficheros que se encuentran comprimidos en el apk no parecen servir, pues siguen produciéndose errores y el manifest es ilegible.
- Optamos por utilizar la herramienta **jadx-gui** para obtener el contenido del apk. El manifest obtenido es legible y se copia al proyecto junto al contenido de la carpeta **res**.
- Surgen algunos errores menores durante la compilación, que son resueltos modificando manualmente los xml de la carpeta **res** que dan problemas de dependencias.
- Compilamos la aplicación y se genera el apk, el cual ejecutamos en un entorno seguro, y comprobamos que el funcionamiento es idéntico al del apk proporcionado.

- Finalmente, solo es necesario modificar el código para que muestre la flag original, deshacer la rotación y obtener la flag válida para el challenge.



Ejecución del apk modificado para que muestre la flag original

Análisis de Código malicioso en la flag 2

En los siguientes archivos (***FileProcessor.kt*** **y** ***MainActivity.kt***) también es posible encontrar posible código malicioso. En general, podemos observar que se produce una modificación de archivos no autorizada por el usuario (mediante la copia *.backup* y el archivo que es procesado *.processed*).
También vemos que se incluye la flag transformada al final de cada archivo procesado (técnica normalmente usada para **filtrar información** y/o incrustar marcadores) y la ofuscación de los datos. Vamos a analizar los archivos por separado.

En ***FileProcessor.kt*** vemos lo siguiente:

- La función *processFile(file:File)* lee el contenido original y lo transforma usando la función *transformFlag()* (esta función transforma la flag con operaciones como XOR y luego es la ofuscación que se mete en los archivos) y *mbaTransform()* (la cual aplica una transformación bit a bit que luego es difícil de rastrear y también puede ser una forma de esconder información) y luego escribe los datos en un archivo *.processed*, esto se usa para modificar de forma “silenciosa” archivos e inyectar datos. Por lo que, como sabemos, corromper los archivos puede ser visto como un posible comportamiento **ransomware**.

- También podemos ver que con las funciones *generateFlag()* y *generateRandomHex()*, somos capaces de generar una flag ofuscada que combinada con un prefijo y sufijo aleatorios, se puede usar para **rastrear a las víctimas** (la flag no es trivial).

En ***MainActivity.kt*** vemos lo siguiente:

- Con la función *processDirectory(directory: File)*, iteramos sobre los archivos en el directorio externo del dispositivo, entiendo que sin el permiso del usuario, puede llegar a procesar muchos archivos (posible **exfiltrado de datos**). (Esta línea del código *Environment.getExternalStorageDirectory()*, básicamente accede al almacenamiento externo donde pueden estar los datos personales de la persona.)

- Considero que también puede ser un problema la función *isTargetFile(file: File)* puesto que al solo seleccionar unos archivos con una extensión en específico (*.txt*, *.doc*…), es más fácil difundir los datos insertados o la transformación aplicada ("*payload*") a muchos documentos del usuario. Esto puede causar que se **aumente el alcance del daño** o la manipulación que puede llegar a hacer el programa.