CADI 5to. Congreso Argentino de Ingeniería
11vo. Congreso Argentino de Enseñanza de la Ingeniería

ANÁLISIS FORENSE DE SISTEMAS DE ARCHIVOS EN PYTHON

Constanzo, Bruno[a]; Nogueira, Enzo[b]; Di Iorio, Ana[a]
a InFo-Lab, Facultad de Ingeniería, Universidad FASTA
b Facultad de Ingenieria, Universidad Nacional de Mar del Plata
bconstanzo@ufasta.edu.ar

Resumen

Ante la ocurrencia de un hecho y en la búsqueda de rastros digitales que permitan reconstruir lo sucedido, los procesos de recuperación de información involucran la adquisición de la imagen forense, y la extracción y análisis de los datos. En este proceso, el parsing de las estructuras de sistemas de archivos es una tarea básica que se realiza sobre toda imagen de los medios de almacenamiento involucrados. De este análisis se desprenden metadatos de los archivos (fechas de creación, acceso y modificación, permisos de usuario, ubicación de los clusters en la partición, entre otros) extremadamente útiles.

Se presenta en este trabajo un proyecto que tuvo como resultado una librería desarrollada en Python que expone las estructuras básicas de un sistema de archivos de manera simple, y permite al informático forense acceder a metadatos y estructuras de bajo nivel.

Se ejemplifica esta potencialidad mediante casos de uso de pericias informáticas, así como la posibilidad de extender la librería para dar soporte a nuevos sistemas de archivos. La licencia LGPL elegida para su desarrollo garantiza el acceso al código, sin limitar posibilidades de uso. Además, la disponibilidad del código fuente permite un mejor entendimiento del funcionamiento interno de un sistema de archivos, de qué manera se accede a los datos y cómo los manipula una herramienta que utilice la librería, lo que permitiría auditar cualquier análisis forense realizado con la misma.

Abstract

When faced with an incident and the search of digital traces that allow to reconstruct what happened, data recovery processes require the acquisition of a forensic image, and the extraction and analysis of the data. In this process, parsing filesystem data structures is a basic task that is applied over every image of the storage drives involved. From this analysis file metadata is derived (Modified-Accesed-Created timestamps, user permissions, location of file clusters, among others) that is extremely useful.

We present in this work a project that resulted in a Python library that exposes the basic structures of a file system in a simple manner, and allows the digital forensic expert access metadata and low-level structures.

Examples of this potential are shown through examples of forensic analysis, as the possibility to extend the library to support new filesystems. The LGPL license chosen for development guarantees access to the code, without limiting users of it. The availability of the source code allows a better understanding of the inner workings of a filesystem, how data is accessed and how any tool that relies on this library handles it, allowing for easy audits of forensics analysis performed with it.

***Palabras clave**:* Informática Forense, Evidencia Digital, Sistemas Operativos, Python

INTRODUCCIÓN

Prácticamente todo análisis forense digital involucra el análisis de medios de almacenamiento masivo, ya sean HDD, SSD, pen drives, medios ópticos u otros. En estos, el perito informático encontrará sistemas de archivos, y de ellos deberá derivar información relevante para la causa: archivos vinculados con el hecho investigado, actividad del usuario, fechas de creación, modificación o último acceso de la información almacenada, o información de bajo nivel que el sistema operativo no expone [1].

Para organizar la información en un medio de almacenamiento los sistemas operativos utilizan sistemas de archivos o *filesystems*, conjuntos de estructuras de datos y reglas lógicas que permiten organizar contenido de manera entendible tanto para las personas como para las computadoras. El sistema operativo monopoliza el acceso a los archivos de manera que un proceso no pueda corromper la integridad del sistema de archivos, y también para brindar abstracciones que simplifican para los usuarios (ya sean personas o procesos) el almacenamiento persistente de la información [2].

A nivel sistema operativo el acceso a los *filesystems* se implementa necesariamente en modo kernel, ya que se debe operar en bajo nivel sobre los dispositivos de almacenamiento:

- En Windows nos encontramos con los *File System Drivers* (FSD) que permiten manejar tanto los sistemas de archivos locales al equipo como los de red [3].
- En Linux se pueden cargar módulos del kernel que manejen el acceso a un *filesystem* específico, o se puede utilizar el mecanismo FUSE para proveer un programa en espacio de usuario que interactúa con la librería libfuse integrada al kernel de Linux.

Pero esta situación no es conveniente para el análisis forense: en primer lugar, se pretende trabajar sobre una imágen forense y no sobre un dispositivo de almacenamiento [4]. Si bien es posible conectar una imagen forense por medio del *loopback device* en Linux, o por medio de drivers virtuales en Windows, tampoco es conveniente para el análisis forense exponer las imágenes de esta manera.

En segundo lugar, como ya se mencionó antes, los sistemas operativos no exponen toda la información del *filesystem* en las APIs de usuario, y esa información puede resultar útil para el análisis forense: cuándo fue la fecha de modificación de un registro de archivo, el *journal* del sistema de archivos, qué clusters fueron ocupados por un archivo (o eventualmente sobre-escritos por otro archivo luego de una eliminación), por citar algunos ejemplos, es toda información de gran utilidad para un forense, pero para accederla se necesita el acceso a más bajo nivel sobre el *filesystem*.

Por estas razones, es común utilizar en los análisis forenses *parsers* de sistemas de archivos, programas o funciones que tienen la capacidad de procesar los bytes de las estructuras de bajo nivel, y representar la información que estas contienen de manera entendible.

En este trabajo se presenta una librería escrita en Python que implementa parsers para dos *filesystems*, y también para otras estructuras asociadas como son tablas de particiones y encabezados VHD. El desarrollo de esta librería se enfocó en brindar acceso a los datos de bajo nivel de cada sistema de archivos, a través de código entendible y fácil de utilizar. Las abstracciones que brinda la librería permiten también implementar funciones de E/S sobre los *filesystems*, y de esta manera se puede enseñar cómo funcionan los sistemas de archivos reales y herramientas de recuperación de archivos.

En la siguiente sección se dará una breve definición teórica de los conceptos necesarios para entender el resto del trabajo. Luego se explicará en detalle el desarrollo, y finalmente se llegará a algunas conclusiones e ideas sobre el trabajo a futuro en este proyecto.

MARCO TEÓRICO

En esta sección haremos un repaso breve de conceptos generales de sistemas de archivos sin entrar en detalles técnicos. Salvo que se especifique explícitamente, los conceptos aquí expuestos refieren a las definiciones de Tanenbaum [2].

La mayoría de los dispositivos de almacenamiento tienen en sus primeros sectores una tabla de particiones. Aunque es infrecuente, eventualmente pueden encontrarse dispositivos de almacenamiento sin particionar, donde un sólo *filesystem* ocupa la totalidad del medio. Un medio particionado es un sólo dispositivo físico que se expone al sistema operativo como múltiples volúmenes lógicos, lo que permite organizar la información en el dispositivo. Por ejemplo, es común en sistemas operativos Linux contar con una partición para el sistema, una partición para

la memoria virtual, y (al menos) una partición para los datos de usuario.

Actualmente conviven dos mecanismos para la representación de las particiones en dispositivos de almacenamiento: *Master Boot Record* (MBR) y *GUID Partition Table* (GPT) [5].

MBR fue un mecanismo implementado por IBM en los años 80 y presenta múltiples limitaciones: sólo dispone de 64 bytes para representar hasta 4 particiones, y no puede direccionar más allá de 2 TB en el medio de almacenamiento. Se utilizó durante casi 30 años, extendiéndolo mediante mecanismos como *Extended Boot Record* (EBR, una forma de permitir más de 4 particiones dentro del esquema de particionado).

Las limitaciones de MBR llevaron a Intel, junto con el Foro UEFI, a definir un nuevo formato de tabla de particiones. GPT utiliza direcciones lógicas de bloque (LBA, del inglés *Logical Block Address*, a diferencia de MBR donde coexisten LBA y otro método para referir a las direcciones dentro del dispositivo de almacenamiento), y se organiza en un encabezado que define algunos metadatos de la tabla de particiones, y luego la tabla de particiones propiamente dicha en los siguientes 16KB. Con el tamaño estándar de registro de partición de 128 bytes se pueden definir 128 particiones, aunque es posible reducir el tamaño del registro para definir más particiones si fuera necesario. Dado que utiliza enteros de 64 bits para manejar los LBA, GPT puede direccionar dentro de un medio de almacenamiento hasta 9.4 zettabytes.

Cada partición en un medio de almacenamiento aloja un sistema de archivos. En este punto es importante definir que, cuando se habla de sistemas de archivos, podemos referirnos en sentido amplio a las reglas y estructuras de datos de un tipo de sistema de archivos (por ejemplo, FAT32 o ext2), o en particular a un sistema de archivos específico, una instancia particular con datos únicos.

Los sistemas de archivos tienen en los primeros sectores un encabezado que describe al resto del sistema de archivos, define metadatos y punteros para facilitar la lectura del mismo. Dependiendo de cada tipo de sistema de archivos, estas estructuras variarán. En algunos *filesystems* además de estos datos, el encabezado contendrá código de arranque, instrucciones de assembler que guiarán el proceso de inicio de un sistema operativo.

Los archivos son una abstracción que permite almacenar información de manera persistente. Si queremos organizar archivos de acuerdo a una jerarquía o un sistema lógico, entonces utilizaremos directorios o carpetas. En muchos sistemas de archivos los directorios se implementan como un tipo especial de archivo capaz de contener a otros archivos en su interior (y por ende, otros directorios). Los sistemas de archivos que se organizan en directorios de manera jerárquica pueden representarse por medio de árboles, y tenemos un nodo de suma importancia: el directorio raíz, el directorio inicial del *filesystem* del cual se desprenden todos los demás archivos y directorios en él almacenados. Dado que es de suma importancia, el directorio raíz suele estar referido en el encabezado del filesystem, o siempre se almacena en una ubicación determinada.

Para asignar espacio de almacenamiento (sectores del dispositivo) a los archivos, los sistemas de archivo suelen separar su región de datos en unidades del mismo tamaño, denominadas *clusters* (algunos sistemas de archivos pueden definir mecanismos complementarios). El cluster es la unidad mínima de asignación de espacio a un archivo, y sólo puede asignarse de manera exclusiva a un archivo a la vez. Es de suma importancia para el *filesystem* llevar un registro de qué clusters componen cada archivo y llevar una mapa de asignación de clusters para saber rápidamente cuáles están libres y cuáles ocupados.

Para identificar a cada archivo en el sistema de archivos habrá una estructura de registro, que identifique unívocamente a cada archivo. A partir de este registro de archivo podremos encontrar el contenido del archivo en sí, y tener acceso a sus metadatos, ya sea embebidos dentro del registro o por referencia a alguna estructura auxiliar.

En general como metadatos nos encontraremos con *timestamps* de creación, modificación, y último acceso, el tamaño del archivo, los atributos de archivo (por ejemplo, si se trata de un directorio, un archivo oculto o de sistema, etc). Otros metadatos asociados a un archivo puede ser el usuario dueño del mismo, los permisos de acceso, el tamaño del archivo, o la cadena de clusters que compone el archivo, por nombrar algunos. Dependiendo del tipo de sistema de archivos, los metadatos que éste almacenará de cada archivo, y el tipo de datos o estructuras de datos utilizados para su almacenamiento.

DESARROLLO

Haruspex es una librería escrita en lenguaje de programación Python, utilizando exclusivamente módulos de la librería estándar de este lenguaje y aprovechando los mecanismos y estructuras de datos que éste brinda. Se eligió Python como lenguaje por su simplicidad, potencia y portabilidad, atributos que se trasladan también a la librería desarrollada. A través de Haruspex se proveen clases y funciones que permiten trabajar a nivel de bytes con las estructuras de datos de un *filesystem*. A través de la librería, entonces, es posible abrir un dispositivo de almacenamiento en modo dispositivo de bloques (a través del path /dev correspondiente en Linux, o el \\.\PhysicalDrive correspondiente en Windows), parsear las estructuras de datos de la tabla de particiones, y luego crear un objeto que accede al sistema de archivos en bajo nivel, exponiendo sus estructuras al programador.

Haruspex fue pensada desde un principio como una librería para ser utilizada tanto en el desarrollo de herramientas, o desde una consola interactiva (un *shell*) de Python. Las clases, funciones, métodos, argumentos y valores de retorno de las funciones están pensados para que la librería resulte cómoda de utilizar en estos dos modos, sin esconder datos o valores del usuario programador. El código de la librería se encuentra disponible bajo una licencia LGPL en GitHub en https://github.com/bconstanzo/haruspex/. También se encuentra disponible en PyPI, el repositorio de módulos de Python, bajo el nombre Haruspex, por lo que es posible instalarla ejecutando el comando “pip install haruspex”.

A continuación detallaremos las clases y funciones que expone la librería al usuario, enfocándonos en los dos sistemas de archivo que hoy se encuentran implementados: FAT32 y ext2. Cabe mencionar que, en pos de la facilidad de uso, las dos clases que definen a cada *filesystem* respectivamente respetan la misma interfaz para operaciones comunes, además de definir métodos y atributos propios de cada implementación.

Particiones y VHD

En primer lugar, Haruspex ofrece clases que permiten leer la tabla de particiones, ya sea en formato MBR y GPT, y cada registro de partición dentro de ellas. En ambas implementaciones, se parsean todos los metadatos que la implementación define para cada partición, y se arma una lista con todas las particiones definidas en la tabla. Para procesar una tabla de particiones, es suficiente con leer los bytes correspondientes desde el dispositivo o imágen forense, e instanciar un objeto de la clase adecuada pasando estos bytes como parámetro al constructor.

Además de las particiones MBR y GPT, Haruspex brinda soporte para el formato de imágen de disco virtual VHD. Dado que el formato se compone, en su forma más básica, de un bloque de metadatos que se escriben al final de una imagen de dispositivo, simplemente alcanza con leer la información de este bloque, y escribir en el mismo los valores adecuados si estamos haciendo una operación de escritura sobre la imagen de disco virtual. De esta manera Haruspex puede tanto leer como escribir VHD que luego se podrán conectar a una máquina virtual o directamente al sistema operativo.

Sistemas de archivos: FAT32

FAT32 es un sistema de archivos simple, que se cimenta sobre dos estructuras fundamentales: la *File Allocation Table* (FAT), estructura que le da su nombre, y los *Directory Entry*, los registros de archivo que almacena los metadatos para cada archivo en el el sistema de archivos.

La FAT es un arreglo lineal indizado por la posición de cada cluster: la posición 0 en la FAT se corresponde con el cluster 2, la posición 1 con el cluster 3 y así sucesivamente (los clusters 0 y 1 no existen en FAT32). Cuando un cluster se encuentra libre, en su posición se almacena un 0, cuando se encuentra ocupado por un archivo en su posición se almacena un valor mayor que 0, que puede ser tanto la posición del siguiente cluster del archivo, o un valor mayor a 0x0FFFFFF0 si se trata del último clúster del archivo.

Por otro lado, el *Directory Entry* es un registro de 32 bytes que almacena el nombre, extensión, atributos, timestamps de creación, modificación y último acceso (con 5, 4 y 2 bytes respectivamente, y por ende distintos niveles de precisión), ubicación del primer cluster, y tamaño del archivo.

Los *Directory Entry* se almacenan en los clusters asignados a cada directorio, y cuando se desea acceder a un directorio es necesario leer todos sus clusters, en la búsqueda del último *Directory Entry* válido (en general es el último que no tiene sus bytes en valor 0).

En el encabezado del *filesystem*, los primeros 512 bytes de la partición, nos encontraremos con el primer cluster del directorio raíz, y algunos

valores que son necesarios para calcular dónde comienza el área de datos del sistema de archivos: cuántas FATs tiene la partición (en general 2, por redundancia), cuánto espacio ocupa cada FAT, la cantidad de sectores reservados, y el tamaño de cluster (en sectores de 512 bytes). Con esta información, ya estamos en condiciones de leer en su totalidad el sistema de archivos y sus contenidos.

Haruspex brinda 3 mecanismos que facilitan esta tarea: en primer lugar, una clase FileHandle que implementa la interfaz de archivos de Python (es decir, FileHandle es un *file-like object*). Por otro lado, se brinda una clase Directory que maneja la lógica para leer e interpretar el contenido de los mismos (es decir, instancia un registro de archivo por cada *Directory Entry* válido que encuentra en el directorio que está abriendo). Por último, al momento de instanciar el sistema de archivos, además de parsear el encabezado y las FATs, carga el directorio raíz y lo asigna al atributo "root" del objeto que representa al *filesystem*. Es decir, una vez instanciado el *filesystem* se cuenta con el directorio raíz a disposición, del cual se pueden ver los contenidos, abrir los archivos o directorios que sean de interés, o explorar las estructuras que los representan.

Para simplificar aún más el acceso a los datos, la clase del sistema de archivos brinda un método "open", que recibe un path absoluto dentro del sistema de archivos, y lo abre utilizando un objeto de tipo FileHandle o Directory, según corresponda.

Internamente las clases manejan los atributos y metadatos que leen del sistema de archivos con *properties*, un mecanismo de python que permite implementar setters y getters de manera transparente al usuario. De esta manera se verifican que los valores que se quieren asignar a los distintos valores sean coherentes con los tipos de datos de cada atributo. También se preservan los bytes de cada estructura como fueron leídos del disco por si el usuario quisiera hacer un examen a bajo nivel de estos valores.

Sistemas de archivos: ext2

Ext2 forma parte de la familia de sistemas de archivos de Linux 'Ext' (*extended filesystem*), siendo este la segunda versión. Hoy en día la mayoría de sistemas operativos basados en Linux (comúnmente llamados *distribuciones*), por defecto usan el sistema de archivos Ext4, que es la última versión al día de hoy, pero de todas formas se puede seguir encontrando con computadoras no tan modernas, o dispositivos extraíbles, que utilicen Ext2. Por otro lado, los sistemas de archivos 'Ext' se caracterizan por estar basados en *features* o características, esto quiere decir que una versión posterior toma como base la versión anterior y le agrega una serie de funcionalidades extra, logrando así la compatibilidad entre las distintas versiones. Por ende, el hecho de tener un parser de Ext2 facilita en gran medida a desarrollar un parser de versiones posteriores del *filesystem*.

Respecto a la estructura en disco de Ext2, una partición con este sistema de archivos se organiza de la siguiente manera: los primeros 1024 bytes están destinados al área de boot (el cual no es gestionado como tal por el *filesystem*), y el resto de la partición se divide en grupos de bloques del mismo tamaño. Cada grupo de bloques se compone de: una copia del superbloque, una copia de la tabla de descriptores de grupo, un bitmap de bloques, un bitmap de inodos, una tabla de inodos, y el resto de espacio se destina a bloques de datos, es decir, donde se encontrarán los datos de todo archivo que pertenezca al *filesystem*. Respecto a esto último, en Ext2 un archivo puede ser considerado como un archivo regular (de datos), como un directorio, etc. Y por más que alguno de esos tipos de archivos no almacenen datos como tal, sino metadatos (como los directorios), sus bloques asignados se encuentran en el área de bloques de datos de un determinado grupo de bloques.

Por lo dicho anteriormente, las siguientes estructuras dan forma a un sistema de archivos Ext2 [6]:

- *Superbloque*. Almacena información general de todo el *filesystem*, como por ejemplo el tamaño de bloque, la cantidad de bloques por grupo, etc.
- *Descriptor de grupo*. Almacena información sobre un determinado grupo de bloques, como por ejemplo la cantidad de bloques libres, el n° de bloque donde comienza la tabla de inodos, etc.
- *inodo*. Almacena los metadatos de cualquier tipo de archivo, y es el medio para acceder a sus datos; existe un único inodo para cada archivo del *filesystem*, identificado con un número.
- *Entrada de directorio*. Es la estructura que compone a un directorio, y es donde se

almacena el nombre de un archivo junto con el número del inodo que lo representa.

Teniendo en cuenta las estructuras de Ext2, el parser de este sistema de archivos se desarrolló de la siguiente manera.

Por un lado, se cuenta con cuatro clases de Python que representan a las cuatro estructuras de datos: *Superblock*, *GroupDescriptor*, *Inode* y *DirectoryEntry*. La función de cada una es analizar una secuencia de bytes dada (previamente leída del disco o imagen forense), y traducir su significado, para dar origen a la estructura de datos como tal. De esta manera, podremos ver cada uno de los campos que conforman a una determinada estructura, sin ninguna limitación: tendremos acceso total a la información del *filesystem* y sus estructuras.

Por otro lado, se cuenta con las clases de Python encargadas de representar y manejar los directorios y archivos del *filesystem*. En el caso de los directorios (clase *Directory*), se parsea su estructura interna, es decir, las entradas de directorio (haciendo uso de la clase *DirectoryEntry*). Para esto, primero se leen los bloques del *filesystem* que estén asignados al directorio en cuestión, donde efectivamente se encuentran las entradas de directorio. Quien sabe cuáles son dichos bloques, es el inodo que representa al directorio (todo inodo tiene la referencia a cada uno de los bloques asignados al archivo o directorio que está representando). Y en el caso de los archivos (clase *FileHandle*), se implementa la lectura de sus datos (*read*), el desplazamiento dentro de él (*seek*), la posición actual de su puntero a los bytes (*tell*) y la clausura del mismo (*close*). Al igual que con los directorios, primero se deben ubicar los bloques asociados al archivo (que en este caso serán de datos 'puros'), mediante el inodo que lo representa. Pero como no hay nada que parsear dentro de esos bloques (los archivos de datos son un conjunto de bytes sin estructura), se van leyendo los bloques a medida que se hace un 'read', obteniéndose así los bytes 'en crudo'.

Y por último, se cuenta con la clase que efectivamente nos permite analizar una partición con Ext2 (clase *Ext2*), y maneja la lógica de cómo recorrer el *filesystem*. Esta clase es la encargada de leer los bytes del *filesystem* y comunicarse con las demás clases para parsear las estructuras adecuadamente. Primero parsea el superbloque, la tabla de descriptores de grupo y el directorio raíz, y luego nos permitirá tanto abrir archivos y directorios (*open*), como leer en crudo cualquier bloque o inodo del *filesystem*.

CONCLUSIONES Y TRABAJO FUTURO

Se presentó en este trabajo la librería Haruspex, escrita en Python que permite parsear las estructuras de sistemas de archivos y extraer información por medio de una interfaz de programación simple y extensible.

La elección de una licencia LGPL permite que sea utilizada por terceros, pero aún así protegiendo al código de la librería y requiriendo que se publiquen modificaciones sobre el código de la librería propiamente dicha. De todos modos, los autores estamos atentos a sugerencias en este aspecto y abiertos a considerar alternativas.

Al haber implementado dos sistemas de archivos reales, FAT32 y ext2, y verificado que las implementaciones son correctas, se demostró la capacidad de la librería para dar soporte a *filesystems* reales. Esto es particularmente útil porque en ocasiones al momento de realizar pericias informáticas los expertos se enfrentan a dispositivos que implementan sistemas de archivos propios, sobre los cuales es necesario realizar un trabajo de ingeniería inversa para poder dilucidar su funcionamiento [7]. Por medio de Haruspex es posible implementar una capa de abstracción del *filesystem* en cuestión, que permita acceder a las estructuras, metadatos y la información almacenada en el dispositivo de manera transparente. Actualmente no existen herramientas forenses que permitan realizar esto de manera simple como lo permite nuestro desarrollo.

Además, dado que la librería presentada está programada utilizando únicamente funciones de la librería estándar de Python, es portable a cualquier sistema operativo donde haya una distribución de Python disponible. Se han realizado pruebas instalando Haruspex en Windows, Linux y Android.

La librería se ha utilizado también en el ámbito de clases de la materia Sistemas Operativos, brindando a los alumnos la posibilidad de trabajar con, explorar y analizar el comportamiento de sistemas operativos reales. También se ha utilizado en cursos de postgrado para mostrar los mecanismos de recuperación de archivos que implementan algunas herramientas forenses.

A futuro, se planea ampliar el soporte de sistemas de archivos incorporando NTFS y exFAT. Se están incorporando mecanismos para dar soporte a la escritura de datos, lo que

permitiría realizar experiencias prácticas más ricas en clases, y automatizar la creación de imágenes forenses de prueba, por nombrar algunas posibilidades.

AGRADECIMIENTOS

Agradecemos a la Universidad FASTA y el Ministerio Público de la Provincia de Buenos Aires por brindar un espacio de trabajo único como es el InFo-Lab donde podemos realizar este tipo de proyectos de investigación y desarrollo.

Queremos dedicar este trabajo a la memoria de Felipe Evans, un gran ingeniero, profesor y amigo.

REFERENCIAS

1. Roussev, V. (2017). *Digital Forensic Science: Issues, Methods, and Challenges.* Morgan & Claypool.
2. Tanenbaum, A. (2009). *Sistemas Operativos Modernos (3ra Edición)*. Pearson Educación. México.
3. Solomon, D., Russinovich, M. (2012). *Windows Internals, Sixth Edition, Part 2.* Microsoft Press. Redmond.
4. Nikkel, B. (2016). *Practical Forensic Imaging: Securing Digital Evidence with Linux Tools.* No Starch Press. San Francisco.
5. Microsoft. (2009). *How Basic Disks and Volumes Work*. Documentación online MSD. Disponible en: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc739412(v=ws.10).
6. Bovet, D., Cesati, M. (2002). *Understanding the Linux Kernel, 2nd Edition.* O'Reilly Media.
7. SIlva, G. (2018). Ingeniería inversa del sistema de archivos de DVRs PCBox. *Simposio de Informática y Derecho SID 2021, 47 JAIIO Jornadas Argentinas de Informática.*