Sistema de archivos remoto cifrado y de un solo archivo

Juan Antonio Nepormoseno Rosales

Resumen– El incremento en el uso de los servicios de almacenaje de datos en la nube, en los que el propietario de los datos guarda su información en los servidores de un tercero, resulta en un aumento del riesgo de que sus datos se vean comprometidos o manipulados. Almacenar datos en un medio no seguro es un problema que ya ha sido tratado con anterioridad en incontables ocasiones, e incluso si las partes principales necesarias para realizar este proyecto están al alcance de todos, no parece haber una solución disponible. El objetivo de este proyecto es el de diseñar y desarrollar una solución para mitigar este problema.

Palabras clave– Sistema de ficheros remoto, sistema de ficheros, FUSE, ciberseguridad, Python, PyCrypto.

Abstract– The increasing use of cloud storage services, in which the data owners upload their files a third party's server, results in an increasing risk of this data getting compromised or manipulated. Storing data in a non secure medium is a problem that has been treated time and again, and although the primary necessary parts needed to implement a solution for this problem are available to everyone, we could not find a good solution for it. This project's goal is to design and develop a solution to mitigate this problem.

Keywords– Remote file system, file system, FUSE, cyber security, Python, PyCrypto.

1 Introduction

La tecnología de comunicaciones ha evolucionado para permitir el trabajo descentralizado y la compartición de archivos en tiempo real y sin esfuerzo extra por parte del usuario. Esto es así hasta tal punto que contamos con servicios como Dropbox [1], que permite sincronizar el contenido de un directorio con un servidor en la nube al que pueden acceder varias personas de forma concurrente. A efectos prácticos, lo que nos ofrece este servicio es poder compartir un directorio a través de Internet con múltiples usuarios.

Este tipo de servicio es de inestimable valor para la sociedad actual, puesto que el intercambio de información en tiempo real y/o a gran escala es fundamental para el funcionamiento de miles de empresas que crean, consumen o incluso manejan estos datos. Sin embargo, parece ser que debido a la facilidad de uso y a la aparente seguridad de la que hacen alarde estos sistemas, se confían ciegamente datos confidenciales y de carácter privado cuyo filtrado supondría un grave problema para todas las partes implicadas. En la mayoría de los casos se garantiza la confidencialidad de los datos, pero se especifica que la empresa que controla el servicio puede acceder a ellos en cualquier momento debido a que esta se encarga de la parte criptográfica y, por lo tanto, tiene las claves usadas para cifrar y las necesarias para descifrar cualquier información que un usuario envíe a sus servidores [2]. Esto puede suponer una grave amenaza en el caso de que un atacante consiga desencriptar los datos, sea con ayuda interna o porque el propio atacante forme parte de esta empresa. En este caso, tanto la confidencialidad como la integridad de los datos se verán afectadas.

Para solucionar este problema necesitamos una base que permita obtener las mismas ventajas de estos servicios, la capacidad de compartir ficheros y sincronizar directorios, a la vez que se encargue de resolver el problema de la seguridad. Para ello se propone una solución basada en permitir la sincronización mediante la distribución de un archivo que guarda los cambios realizados sobre un sistema de ficheros propio. Esto quiere decir que tendremos que crear un fichero cifrado que los usuarios compartirán por un medio desconocido y que se presume que no será seguro, como por ejemplo uno de los servidores de terceros mencionados an-

• E-mail de contacte: juanantonio.nepormoseno@e-campus.uab.cat
• Mención realitzada: Tecnologies de la Informació
• Treball tutoritzat per: Ian Blanes Garcia (Departament d'Enginyeria de la Informació i de les Comunicacions)
• Curs 2015/16

teriormente. En este fichero se guardarán los cambios que los usuarios realizen sobre el directorio compartido. Para ello, en primer lugar, debemos detectar los cambios que se hacen sobre ese directorio, y con esa intención se creará un sistema de ficheros propio. Este deberá interactuar con nuestro programa cada vez que se realice una operación sobre él. Por ejemplo, al crear un directorio deberá comunicarle al programa que se está creando un directorio nuevo, junto con la dirección en la que se está creando y el resto de datos relevantes para poder repetir esta acción en otra máquina. El programa deberá entonces guardar todos estos datos en un fichero, que estará cifrado para impedir que una persona o máquina sin autorización para leer o modificar esos datos hagan eso mismo. Sin embargo, este fichero deberá ser descifrable e interpretable por otro computador siempre y cuando este esté autorizado a hacerlo. De esta forma conseguiremos que el cifrado sea de punto a punto, por lo que los propietarios del servidor usado para almacenar este archivo no tendrán manera de acceder a los datos en claro del fichero. Además, al cargar los datos de ese archivo en otra instancia del programa, el programa se encargará de repetir los cambios que no haya aplicado aún, por lo que conseguiremos la sincronización que buscamos.

1.1 Objetivos y metodología

El objetivo definido para este proyecto ha sido el de crear un programa capaz de realizar los pasos definidos anteriormente. Para lograr alcanzar esta meta, se ha propuesto una división en tareas del trabajo de diseño e implementación. Estas tareas definen la estructura de este documento y son las siguientes:

- **Diseño del formato del archivo:** este punto se refiere al estudio y al diseño del formato y los datos necesarios para crear el archivo y asegurar el correcto funcionamiento del programa. Para realizar esta tarea se ha subdividido el trabajo en objetivos más pequeños. En primer lugar, se estudiará el estado del arte para aprender de los éxitos y, más importante, de los errores que ya han cometido otras posibles soluciones al problema. A continuación se definirá la estructura que se ha diseñado para el fichero, junto con los datos que esta necesita. Para ello se estudiarán las distintas operaciones que se pueden realizar sobre el sistema de ficheros con el fin de decidir qué operaciones serán necesarias para repetir los cambios sobre este.

- **Diseño del programa:** en esta parte se trata el diseño y la creación del sistema de ficheros y del programa, junto con la integración de todos los módulos del programa. Para ello se definirá cada módulo por separado y se explicará cómo interactua cada uno de los módulos con el resto. Se ha tenido en cuenta el funcionamiento que queremos lograr que tenga el programa para definir los requisitos de los distintos módulos y cómo deberán conectarse entre ellos. El programa tiene cinco módulos que son, por orden: el encargado de crear y gestionar el sistema de ficheros, el encargado de escribir en el archivo, el encargado de cifrar los datos y los encargados de leer el archivo y de descifrar los datos.

- **Criptografía:** en este punto se estudian las opciones disponibles a la hora de escoger un algoritmo de cifrado. El objetivo consiste en escoger un método de cifrado seguro, adecuado y que cumpla con los requisitos que se han definido. Para empezar se realizará un análisis de amenazas a las que puede ser sometido el sistema, con la intención de descubrir los puntos en los que la confidencialidad e integridad de los datos pueden quedar en peligro frente a un posible atacante. Estas amenazas se tendrán en cuenta a la hora de escoger los métodos de cifrado y durante el diseño del programa. En la planificación del proyecto este apartado está por separado, pero para mantener este documento lo más simple posible y ya que las partes relativas a la criptografía también están relacionadas con el diseño del archivo y del programa, se explicarán las decisiones tomadas sobre este tema en esos apartados. Esto se debe a que la parte criptográfica es una extensión a la funcionalidad tanto del fichero como del programa, por lo que se entiende mejor si se explica junto con estos. Además, debe quedar claro que en este trabajo no se tratará la forma de transmisión o almacenamiento en la nube del archivo, por lo que la seguridad de los datos debería ser independiente de ella.

En este documento se analizará, en primer lugar, el estado del arte de los servicios de almacenamiento y sincronización en la nube, para seguir con una explicación en detalle de las decisiones que se han tomado durante el diseño de los tres puntos anteriores: formato, programa y criptografía. Se seguirá con los resultados obtenidos tanto en la implementación como en las pruebas manuales y automáticas que se han realizado a lo largo del desarrollo sobre el sistema para poner a prueba su eficacia y robustez frente a datos anómalos o una carga de trabajo elevada. Para terminar, se presentarán las conclusiones obtenidas junto con unas posibles líneas de continuación.

2 Estado del arte

Disponemos de muchos servicios que pueden ser usados para compartir y sincronizar los datos, pero estos suelen descuidar la privacidad o no permitir al usuario control sobre ella [2]. Este es el caso de Google Drive o de el tan conocido Dropbox, que aplica medidas de seguridad como encriptar sus canales de comunicación (AES de 128 bits) y los ficheros que almacenan (AES de 256 bits), pero esto lo hacen por su parte, teniendo conocimiento de las claves usadas para cifrar y, por lo tanto, de las necesarias para descifrar. El usuario no tiene control sobre su privacidad, lo que se traduce en que no la tiene. Los administradores de Dropbox pueden acceder a sus datos en cualquier momento y hacer con ellos lo que deseen. Esta situación es la que se repite en la mayoría de soluciones actuales.

Lo ideal para mantener la privacidad sería que se aplicara un protocolo de conocimiento cero, en el que el servidor que guarda los datos no tiene conocimiento de las claves necesarias para descifrarlos. De esta forma, la tercera parte sólo se encarga de guardar los datos y no tiene responsabilidad sobre ellos. Este es el caso de Mega, el servicio de hosting de archivos sucesor de Megaupload, que aplica el cifrado y el descifrado en los clientes, por medio de la apli-

cación de escritorio o la aplicación web. Con este protocolo consiguen que la privacidad de los datos sea completa desde que estos salen de un usuario hasta que llegan al otro.

Además, está el problema de la sincronización. Mega puede tener el problema de la privacidad solucionado, pero no provee ningún mecanismo para modificar los datos y, por lo tanto, no existe la posibilidad de trabajar sobre ellos. La única solución que pone en práctica estas ideas a la vez es la de Tresorit, pero al ser software propietario y de pago no hay forma de acceder al código.

3 Diseño del fichero

Como se ha comentado en la introducción, el siguiente paso para la realización de este proyecto ha sido realizar el diseño del fichero que guarda todos los datos del directorio. En este apartado hablaremos de las decisiones que se han tomado y explicaremos porqué se han tomado.

3.1 Forma general del archivo

Para el formato del archivo se pensó que lo mejor sería tener una cabecera inicial con los datos imprescindibles para que funcione el programa. A continuación, se añadiría una lista de mensajes que guardaría cada llamada al sistema que se ha realizado en el directorio compartido, guardando también los datos necesarios para reproducir esa llamada en el directorio de otro cliente.

Respecto a los datos del programa, sólo necesitamos compartir un vector de 16 bytes que se genera al iniciar el programa y que es común en todos los clientes. Este vector está relacionado con el apartado criptográfico de la aplicación y hablaremos más adelante sobre él en la sección 3.5. La estructura del archivo, por lo tanto, es la definida en la figura 1.

Fig. 1: Forma general del archivo

Existen otros datos como la clave usada para cifrar y descifrar el archivo o un identificador del último mensaje interpretado, pero estos datos se guardan por separado debido a que son o bien de carácter secreto o bien de carácter local y único para cada cliente.

3.2 Cabeceras de un mensaje

Para empezar, tenemos que separar los datos que serán comunes en cada mensaje. Con este propósito nos inspiramos en los mensajes de protocolos de internet para definir los datos que necesitamos introducir en el mensaje. Por ejemplo, igual que el número de secuencia y el número de acknowledgment de un segmento de TCP [3], nuestros mensajes deberían estar numerados e indicar cuál fue el mensaje anterior al actual para poder ordenarlos si se recibieran de forma desordenada. Para cada uno de estos datos, el mensaje tendrá un header con una serie de rangos de bytes dentro del mensaje asignados. Esos bytes definen el valor real de ese dato.

Como apunte adicional, la idea de basarse en protocolos de internet puede ser útil si en el futuro se decidiera implementar una función para enviar los datos a un servidor en tiempo de ejecución, pues ya tendrían un formato pensado para ser transmitido por internet. Aprovechando esta idea y para facilitarla aún más, se definió que el orden de escritura a memoria de los bytes, también conocido como endianness, fuera el usado en Internet.

A continuación se definen los datos que se introducirán en la cabecera:

- **ID del mensaje:** un entero de 4 bytes que actúa como identificador del mensaje. Cada mensaje está numerado para poder mantener un control de qué cambios se han realizado desde la última vez que se inició el programa. El programa guarda la última ID procesada y, una vez se vuelve a abrir con datos nuevos en el archivo, puede leer esta última ID para saber a partir de qué mensaje debe empezar a procesar.

- **ID del mensaje anterior:** al igual que el valor anterior, es un entero que guarda la ID de un mensaje, pero esta vez se trata del mensaje anterior al actual (o parent). Su presencia está pensada en que puede facilitar la identificación de problemas de incongruencias que podrían surgir en un futuro, suponiendo que se decide extender el programa para permitir la edición distribuida y en concurrente del fichero. De esta forma podemos recorrer la lista de cambios de forma ordenada para identificar estos conflictos y, potencialmente, resolverlos.

- **Fecha:** un dato de tipo entero de 8 bytes que guarda la fecha en "tiempo UNIX" en la que se ha realizado el cambio. Esto quiere decir decir que su valor es igual a la cantidad de segundos transcurrida desde el 1 de Enero de 1970 hasta el momento en que se guarda el mensaje en el archivo.

- **Longitud del payload:** una variable de tipo entero de 8 bytes que guarda un número que define la cantidad de bytes que ocupa el payload de este mensaje.

- **ID de operación:** este último dato es una variable que sólo ocupa 1 byte. Se usa para identificar qué operación se está leyendo, ya que cada uno necesita unos datos distintos. Para diferenciarlas asignamos un valor distinto en este campo para cada operación.

3.3 Tipos de operación

Para identificar los distintos tipos de operaciones se ha estudiado el módulo con el que se detectarán los cambios que se hacen sobre el directorio. Para ello crearemos nuestro propio sistema de ficheros, que detectará las llamadas al sistema y guardará cada una como un mensaje dentro del fichero. La implementación del sistema de ficheros se realizará con **FUSE**, o Filesystem in Userspace [4]. Este módulo tiene una serie de operaciones que podemos sobreescribir para hacer que nuestro sistema de ficheros realice acciones personalizadas cuando se llamen a esas funciones. Se han definido las que habría que tener en cuenta para transmitir los datos realizados por un cliente a otro y que este pueda realizar esos mismos cambios en su copia local, quedando así idéntico al de la máquina inicial. Estas son, de forma resumida, las siguientes:

- **mknod:** creación de un fichero.
- **mkdir:** creación de un directorio.
- **unlink:** elimina un fichero.
- **rmdir:** elimina un directorio.
- **rename:** cambia el nombre de un fichero o directorio.
- **chmod:** cambia los bits de permisos de un archivo.
- **truncate:** cambia el tamaño de un archivo.
- **open:** abre un archivo (para lectura o escritura). Necesaria para comprobar si un archivo puede ser abierto y porque el programa utiliza esta operación como aviso de que debe abrir el archivo mencionado y en qué modo.
- **write:** escribe en un archivo.
- **lock:** bloqueo POSIX de un archivo (restringe el acceso a un archivo al usuario o proceso que lo está accediendo en ese mismo momento).
- **release:** libera un archivo después de un lock.
- **releasedir:** libera un directorio después de un lock.

Además, hay una serie de operaciones que no se enviarán, y por lo tanto que no necesitaremos guardar en el fichero. Estas son las que sólo se usan a modo de lectura (por ejemplo, “read” para leer de un archivo o “getattr” para leer los datos de un archivo) o las que no tienen cabida en la descripción de nuestro sistema prototipo (por ejemplo, “symlink” para crear enlaces simbólicos o getxattr y setxattr, que manejan los atributos extendidos de ficheros).

3.4 Análisis de amenazas

En primer lugar se ha hecho un análisis de amenazas para definir de qué posibles ataques puede ser víctima nuestro programa. De esta forma, podemos ser conscientes de estos potenciales problemas de seguridad a la hora de desarrollar el software y tratar de mitigar los daños que pudiera causar la explotación de estas amenazas.

El análisis realizado se basa en los principios de seguridad **CIAA** (siglas de Confidentiality, Integrity, Availability, Authentication) [5], previendo los problemas que pueden haber al respecto de cada uno de estos principios. Cada uno de estos principios significan lo siguiente:

- **Confidentiality (confidencialidad):** es equivalente al concepto de privacidad, y se usa para hablar de la posibilidad de que la información pueda ser interpretada por individuos, entidades o procesos que no deberían tener acceso a ella. A la vez, debemos asegurar que esta información puede ser accedida por aquellos que tienen autorización para acceder a ella.
- **Integrity (integridad):** este concepto implica mantener la consistencia y fiabilidad de los datos durante todo su ciclo de vida. Debemos, por lo tanto, asegurar que los datos no se alteran durante los procesos que los manejan ni durante las comunicaciones entre máquinas. Además, también se refiere a evitar que se pierdan los datos por fallos humanos o debido a eventos exteriores como podría ser, por ejemplo, un fallo de comunicación.
- **Availability (disponibilidad):** este principio se refiere a que debe ser posible poder acceder a la información siempre que sea necesario. Es, por lo tanto, muy importante asegurarse del correcto funcionamiento del hardware y de que no haya problemas de compatibilidad con el software que usa la máquina.
- **Authentication (autenticación):** es el proceso de determinar que algo o alguien es quien dice ser. En el caso que nos ocupa, implica la seguridad de que la información la enviamos y la recibimos de quien está autorizado a recibirla y enviarla.

En este proyecto, solamente se ha considerado realmente crítico que sean seguras la parte de envío de los datos y la de la recepción de los mismos por la parte del cliente. Esto se debe a que la parte del sistema que desarrollamos está orientada a un uso más local, puesto que realizamos el tratamiento de los datos solamente en una máquina. Este tratamiento se hace únicamente antes de enviar los datos a otro usuario y justo después de recibir nuevos datos que otro usuario ha enviado. Estos datos podrían enviarse por Internet directamente o podrían alojarse en un servidor para que lo descargara el resto de usuarios, pero por lo que respecta a nuestro programa esto es irrelevante. Se deben asegurar los cuatro principios de seguridad desde que los datos se generan hasta que son leídos, pues esta es la única forma de asegurar que no serán vulnerables cuando se estén transmitiendo por cualquier medio.

Los resultados obtenidos tanto para el envío como para la recepción han sido los mismos, y, a modo de resumen, se reducen a lo siguiente:

- **Confidentiality (confidencialidad):** la principal causa de preocupación respecto a la confidencialidad es la posibilidad de que el canal de comunicación sea escuchado. También podría darse el caso, si el archivo se aloja en un servidor externo, que este servidor se vea comprometido y un atacante tenga acceso a él, pero este sigue siendo un problema que se traduce en que el canal de comunicación entre los distintos clientes no es seguro. Por lo tanto, la solución adecuada para esto

sería la de encriptar los datos antes de que salgan de la máquina y, preferiblemente, guardarlos en nuestra máquina ya cifrados.

- **Integrity (integridad):** respecto a la integridad de los datos, podemos encontrarnos con que un posible atacante puede cambiar el contenido del archivo y, por lo tanto, hacer que los clientes reciban datos incorrectos. Esto provocaría que el programa se comporte de una forma extraña y potencialmente peligrosa, pues un atacante podría corromper archivos o incluso modificarlos a voluntad. Para para asegurar que no han sido manipulados durante la comunicación o por el servidor que los guarda, usamos un código de autenticación de mensaje (MAC) que se basa en guardar el resultado de ejecutar una función de hash (HMAC) con el contenido cifrado del mensaje. No se ha añadido nada para evitar que se pierdan los datos por fallos humanos o cualquier otro motivo porque se considera que esto es algo que se aleja de los objetivos del proyecto. Estos problemas se pueden solucionar fácilmente usando, por ejemplo, una o varias copias de seguridad de las últimas versiones del archivo de datos.

- **Availability (disponibilidad):** para nuestro proyecto, la disponibilidad consiste en asegurar que se puede acceder a los datos del archivo. No tenemos forma de asegurarla, ya que esto es algo de lo que se encarga el servidor que guarda los datos y este se queda fuera del alcance del proyecto. Para este y para los casos en los que haya un error, generalmente sólo se rechazará la parte del archivo de datos que ha provocado ese error y se continuará la ejecución normalmente o se terminará con un mensaje de error que indique el problema.

- **Authentication (autenticación):** respecto a autenticar que los cambios introducidos en el archivo provienen de una fuente autorizada, podríamos tener un problema en el caso de que un atacante se hiciera pasar por un usuario y nos enviara datos aleatorios con el fin de provocar un funcionamiento errático en nuestro cliente. Sin embargo, gracias a la solución que seleccionamos para resolver el problema de integridad, el HMAC, no hace falta que añadamos nada más. En primer lugar, si recibimos un mensaje interpretable, es decir, que tenga sentido para el programa, y que además tiene un HMAC correcto, podemos asegurar que el mensaje proviene de la fuente de la que se supone que proviene. Además, como sabemos que el mensaje tiene sentido, quiere decir que esta fuente conoce nuestra clave y ha encriptado correctamente datos válidos, por lo que es un cliente que está autorizado a hacer cambios en el sistema de ficheros. En otras palabras, es un cliente con la clave de cifrado.

Por lo tanto, las decisiones tomadas después de este análisis son bastante lógicas. En primer lugar, el contenido del fichero debería estar cifrado. Todos los mensajes se guardarán ya cifrados en el archivo y deberán ser descifrados en el momento de interpretar el nuevo contenido. Además, para asegurar la integridad de los datos y la autenticidad de estos, es decir, que provienen de una fuente autorizada a hacerlos, se añadirá el HMAC al final del mensaje. Si el mensaje cifrado no corresponde con el HMAC guardado, descartaremos el mensaje por no ser fiable.

3.5 Criptografía

Como tenemos que encriptar los datos que genera el programa, debemos valorar las distintas posibilidades que podemos usar como algoritmo de cifrado. Es obvio que modos de cifrado como el Electronic Codebook (AES-ECB) no nos servirán, ya que este es un algoritmo de cifrado por bloques que encripta de forma idéntica dos bloques idénticos. Esto lo vuelve vulnerable a ataques de repetición, además de que hace posible detectar patrones que muy posiblemente se van a repetir en nuestro archivo, como puede ser realizar la misma operación en múltiples ocasiones. Un ejemplo sería el abrir el mismo archivo para editarlo en varias ocasiones. Este problema se puede apreciar mejor con el gráfico de la figura 2, que sirve para entender el funcionamiento de este algoritmo y en el que se ve que ningún bloque afecta a ningún otro. Como ya se ha comentado, esto hace que dos bloques con los mismos datos generen el mismo bloque cifrado.

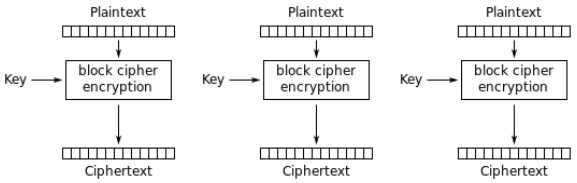

Fig. 2: Funcionamiento del cifrado por bloques (AES-ECB) (Imagen extraída de Wikipedia)

Por lo tanto, no nos sirve un algoritmo de cifrado por bloques normal y corriente, lo que necesitamos es un cifrador de flujo (cipher stream) [6]. Un cifrador de flujo es un cifrado en el que se combina el texto en claro con partes del mensaje cifrado anteriormente. De esta forma, conseguimos añadir aleatoriedad al resultado final y ofuscamos este resultado para que no sea posible detectar patrones. Dentro de este tipo de cifrados hemos encontrado varias soluciones, pero nos hemos decantado por Cipher Feedback (AES-CFB). A continuación sigue una lista de las otras opciones junto con los motivos por los que no han sido escogidas:

- **Cipher Block Chaining (AES-CBC):** este iba a ser el algoritmo escogido inicialmente, ya que aporta los mismos beneficios que el escogido y tiene un funcionamiento muy parecido, como se puede ver en el gráfico de la figura 3. En resumen, con este modo se hace una operación XOR entre el texto a encriptar y el anterior bloque cifrado para hacer a cada bloque único. Sin embargo, se descubrió que tiene una vulnerabilidad que permite modificar el contenido de un bloque si se conoce el texto en claro antes de realizar el ataque [7]. Por lo tanto, para evitar que se diera este caso, se terminó descartando.

- **Output Feedback (AES-OFB):** como se puede apreciar en la figura 4, es similar al escogido, pero no permite un acceso aleatorio a los datos, sino que se debe desencriptar el archivo desde el inicio. Esto puede ser

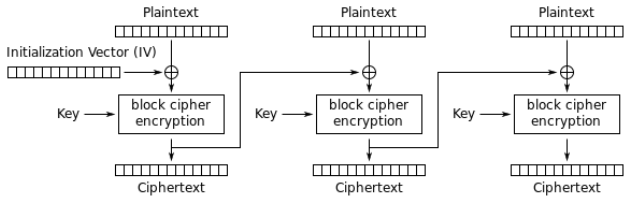

Fig. 3: Funcionamiento del cifrado CBC (Imagen extraída de Wikipedia)

problemático si se realiza mucho trabajo sobre el sistema de archivos, haciendo crecer el archivo de cambios y, por lo tanto, haciendo que la tarea de leerlo de inicio a fin sea más difícil. Con el método escogido tenemos la oportunidad de guardar la posición del último byte leído para empezar a leer desde nos interesa.

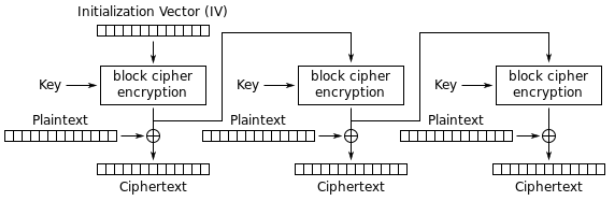

Fig. 4: Funcionamiento del cifrado OFB (Imagen extraída de Wikipedia)

- **Propagating Cipher Block Chaining (PCBC):** una modificación del modo de cifrado CBC para incluir la propagación de cambios tanto en encriptado como en el desencriptado. Por lo tanto, tiene el mismo problema que OFB: no dispone de la posibilidad de acceder a los datos aleatoriamente, con los problemas o falta de oportunidades que eso conlleva.

El modo de cifrado escogido, **Cipher Feedback (AES-CFB)**, tiene una manera de funcionar muy parecido a la del CBC. Este también usa el último bloque cifrado anterior, pero lo usa como vector de inicialización (IV) del proceso de cifrado en lugar de realizar con él una simple operación XOR. De esta forma conseguimos que este pase a formar parte del proceso de encriptado del bloque y no es sólo una operación que se realiza al final con el resultado, con lo que solucionamos la vulnerabilidad que presenta el modo CBC. Esto se puede ver mejor comparando el gráfico de la figura 5 con el propio gráfico del CBC, el de la figura 3.

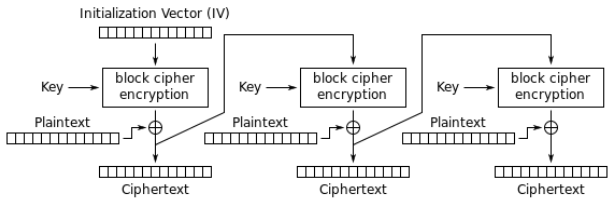

Fig. 5: Funcionamiento del cifrado CFB (Imagen extraída de Wikipedia)

3.6 Resultado del diseño del fichero

Como resultado, tenemos que el formato que tendrán los mensajes consiste en una cabecera de 24 bytes de datos comunes a todos los mensajes, seguidos de un byte que indica el tipo de operación. A continuación, le sigue el payload con los datos de la operación, que puede contener sus propias cabeceras para localizar los valores que guarda. Este payload siempre vendrá dado en bloques de 16 bytes por motivos relativos al funcionamiento del modo de cifrado y todos estos datos se guardarán cifrados en el archivo. Por último, un bloque de 16 bytes contiene el HMAC usado para validar el mensaje y decidir si es fiable o no. Este HMAC se calcula con los datos cifrados del resto del mensaje, tanto header como payload. Se ha realizado el gráfico de la figura 6 para ayudar a entender la localización de cada elemento en el resultado final:

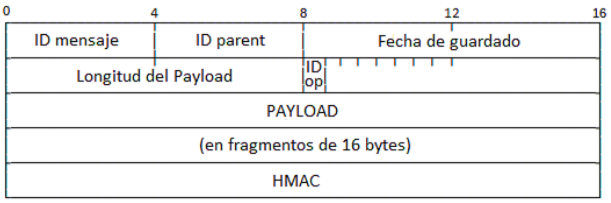

Fig. 6: Formato final de un mensaje del fichero de datos

4 Diseño del programa

Basándonos en los datos extraídos de las conclusiones anteriores, al diseñar el formato del fichero, podemos imaginar el funcionamiento del programa separándolo en distintos módulos. De esta forma, separamos las funcionalidades de cada uno y facilitamos la comprensión del programa como un todo. A continuación sigue una explicación de los módulos propuestos y las funciones de las que deberían encargarse cada uno de ellos.

4.1 Sistema de ficheros

Es la base del proyecto. Gracias a FUSE [4], un módulo para sistemas operativos basados en Unix, podemos crear nuestro propio sistema de ficheros incluso usando un usuario sin privilegios de administrador. En nuestro sistema de ficheros sobreescribimos las operaciones del sistema para que, cada vez que se realice una llamada a una de las operaciones que lo modifican, esta se reconozca y se envíe al módulo de guardado en el fichero. Así pues, este módulo genera lo que para el sistema operativo y para el usuario es un dispositivo virtual. Este es el directorio que compartimos con otros usuarios; todo el trabajo que se haga sobre ese dispositivo será guardado en el fichero para posteriormente ser replicado en otra máquina.

Como ya se ha comentado en el apartado 3.3, se ha realizado un estudio de qué operaciones afectan realmente al sistema y, por lo tanto, qué operaciones debemos guardar en el fichero y qué operaciones no. Por ejemplo, hay operaciones que sólo leen metadatos de ficheros y que son llamadas en multitud de ocasiones, pero estas operaciones no aportan nada, simplemente son usadas por otros programas o rutinas para conocer los datos que guarda el sistema. Estas son las

operaciones que hemos llamada “operaciones de lectura”, entre las que se encuentran las que acceden a metadatos de ficheros y directorios, las de lectura de archivos. Después encontramos operaciones con un uso poco común o que no nos interesaba implementar debido a la falta de tiempo disponible para realizar este proyecto o a que su implementación podría suponer un problema. Este puede ser el caso de operaciones para trabajar con los atributos extendidos de ficheros y directorios, que no se estimaron necesarias para la primera versión del proyecto, o la creación de enlaces simbólicos (symlinks).

4.2 Módulo de guardado de cambios

La tarea de este módulo consiste en guardar los datos que recibe del sistema de ficheros. Para cada tipo de operación deberá generar el mensaje con su cabecera y los datos en el formato adecuado que ya definimos en la sección 2. Para ello usamos el módulo de Python llamado Struct [8], que permite convertir datos en cadenas de bytes con un formato específico. Para ello definimos una cadena de texto que indica este formato junto con el tipo de los datos. Este formato será idéntico para las cabeceras, pero cada operación guarda unos datos distintos y, por lo tanto, habrá una parte específica para cada una. Además, en este formato definimos el endianness de los datos como el mismo que la red. De esta forma nos evitamos conflictos que pudieran surgir si un cliente guarda los datos en little endian y otro intenta leerlos en big endian.

Con esta transformación de los datos nos olvidamos de tener que definir procedimientos independientes que parsean el contenido del payload si decidieramos, por ejemplo, usar el formato XML, que fue el escogido en un principio. Con este método sólo tenemos que realizar el paso inverso, es decir, convertir los datos leídos de bytes a variables del tipo correspondiente según el formato que ya hemos definido a la hora de guardar los datos. Sin embargo, un inconveniente que encontramos es que tenemos que añadir la longitud de las cadenas de texto como un campo del mensaje antes de guardarlas para saber cuántos caracteres tendremos que leer. Esto implica que para cada cadena de caracteres añadimos un entero, es decir 4 bytes extra, que definen su longitud. El tener que añadir datos extra podría llegar a considerarse un problema, pero la alternativa del XML también añade datos en principio irrelevantes al obligar a separar todo mediante etiquetas.

Como apunte final, algunas operaciones requieren del file handler del archivo que modifican para funcionar. No podemos usar el mismo handler que tiene un cliente al modificar su copia local del archivo en otro cliente que está realizando el cargado de cambios, ya que no sería válido y provocaría errores. Por lo tanto, tenemos que guardar los handlers que nos da el sistema operativo al realizar las llamadas a funciones como create u open a la hora de ejecutar los cambios del fichero de cambios. Además, debemos actualizar el valor del file handler si realizamos llamadas a funciones que puedan modificar el path del fichero o el valor de este handler. Este es el caso, por ejemplo, de las operaciones rename o release, que modifican el path del fichero y liberan el handler, respectivamente.

Gracias a este módulo, todos los cambios realizados sobre el sistema de ficheros pasarán por este módulo, por lo que una vez convertidos a los datos que se pueden guardar en el fichero y antes de guardarlos, debemos pasarlos al módulo que se encarga de encriptarlos.

4.3 Módulo de encriptado

Este módulo se encarga de preparar el fichero para ser enviado por un canal no seguro. Su función es la de encriptar los datos en claro y devolver el resultado al módulo de guardado de cambios. Además, también se encarga de calcular el código de validación HMAC. Esto se realiza aplicando una función hash al contenido cifrado del archivo, lo que nos devuelve el código calculado que en nuestro caso será una cadena de 16 bytes. Entonces, el módulo de guardado se encarga de añadir el mensaje ya cifrado y el HMAC al final del fichero de datos.

Todo lo relativo a la criptografía se hace con ayuda del módulo de Python PyCrypto [9]. Para realizar el encriptado simplemente empaquetamos los datos de una operación con el módulo Struct como ya hemos comentado anteriormente y ciframos esa cadena de bytes con una simple llamada al módulo PyCrypto. A continuación hay un código de ejemplo:

```
from Crypto.Cipher import AES

cipher = AES.new(key, AES.MODE_CFB, iv)
result = cipher.encrypt(text)
```

Cabe tener en cuenta que “key” e “iv” son unas variables que contienen un número de bytes igual al tamaño de bloque y que “text” es la cadena de bytes a encriptar. En este pequeño ejemplo “cipher” es el objeto que PyCrypto nos devuelve para realizar el encriptado y “result” sería la cadena de bytes resultado, obviamente.

Para realizar la generación del HMAC del mensaje. Es un proceso muy sencillo, como se puede ver en el siguiente ejemplo:

```
from Crypto.Hash import HMAC, SHA256

hash = HMAC.new(key, digestmod=SHA256)
hash.update(text)
result = hash.digest()
```

Como en los casos anteriores, “key” debe estar correctamente inicializada, pero esa es la mayor complicación de esta parte. Esta función hash no requiere que el texto introducido sea múltiplo de ningún tamaño de bloque, además de que el valor que retorna es de tamaño fijo, por lo que es muy sencilla su escritura y su lectura en y del archivo.

4.4 Módulo de cargado de cambios

Este es el módulo que lee y parsea el contenido el fichero de datos para después aplicar los nuevos cambios en nuestra copia local. El funcionamiento es el mismo que el del módulo de guardado pero a la inversa. En primer lugar, recibimos el archivo cifrado y se lo pasamos al módulo de desencriptado para obtener los datos en claro. Una vez

tenemos los datos en claro, recorremos la lista de mensajes hasta encontrar el último mensaje interpretado por este módulo. Como contamos con la longitud del payload en la cabecera y la longitud del HMAC es fija, sólo nos hace falta ir leyendo las cabeceras de los mensajes hasta encontrar uno con una ID mayor que la del último mensaje interpretado. Una vez hemos encontrado el primer mensaje nuevo, debemos ir recogiendo mensaje a mensaje hasta llegar al final de la lista. Para cada uno de estos mensajes tenemos que guardar sus datos y hacer las llamadas adecuadas al sistema de ficheros. Esto tan simple como que, si el mensaje dice que se han escrito 8 bytes en un fichero, deberemos llamar a la operación de escritura de nuestro sistema de ficheros usando como parámetros el path del fichero y los bytes escritos que encontramos en el mensaje. De esta forma, se repiten todas las acciones que se han realizado en otra instancia del programa, por lo que ambas instancias acabarán conteniendo la misma estructura de ficheros y directorios y estos tendrán los mismos datos en ambos sistemas.

4.5 Módulo de desencriptado

Este es el encargado de, una vez recibido el fichero con los cambios realizados por otro usuario, descifrar el archivo para que puedan tratarse los datos. Guarda la misma relación con el módulo de encriptado que el módulo de cargado con el módulo de guardado, es decir, realiza el proceso inverso. No obstante, el funcionamiento no es tan simple, pues necesitamos recopilar el vector de inicialización de este mensaje, que está formado por los últimos bytes del mensaje anterior. Esto significa que requiere de un método para recoger esos datos del fichero, por lo que nos obliga a tener un método en el módulo de cargado que controle los bloques del fichero y que nos permita seleccionar el bloque adecuado para recuperar el vector de inicialización.

Al final, el código resultado usando el módulo PyCrypto es el siguiente:

```
from Crypto.Cipher import AES

cipher = AES.new(key, AES.MODE_CBC, iv)
result = cipher.decrypt(ciphertext)
```

Como en caso del encriptado, es un código muy simple. No obstante, también como en el anterior caso, hay que asegurarse de inicializar correctamente las variables “key” e “iv” (siendo esta última leída del archivo encriptado).

5 Resultados

Como resultado final del proyecto, tenemos un programa funcional que cumple los requisitos y objetivos planteados al inicio de este documento. El programa funciona en Linux y es capaz de crear un sistema de ficheros con el que los usuarios pueden interactuar. Una vez se han realizado cambios sobre este sistema de ficheros, por ejemplo creando una estructura de carpetas y ficheros, modificando esos ficheros y borrando los que ya no son necesarios. El usuario puede entonces enviar un archivo llamado “commits” que contiene todos estos cambios ya cifrados. En otra instancia del progama, un segundo usuario recibe la clave y el

Fig. 7: Esquema del funcionamiento del programa y de las comunicaciones entre los distintos módulos

fichero “commits”. Al abrir el programa, aparece una lista de los cambios que se han realizado y puede comprobar como, evidentemente, se han reproducido los cambios en su copia del sistema de ficheros.

Respecto a las pruebas, se han realizado de múltiples tipos, manuales y automáticas. Gracias a estas pruebas se han ido descubriendo errores a medida que se desarrollaba. Por ejemplo, uno de los primeros errores encontrados fue un problema con el encoding de los datos a la hora de guardarlos. El programa dejaba de funcionar cuando intentábamos trabajar con caracteres con tilde o con la letra ‘ñ’. La solución fue pasar todas las cadenas de texto a UTF-8 y guardar los datos como bytes. Tal y como este problema se solucionó, ha habido otros problemas similares que también se han arreglado. El resultado final, por lo tanto, tiene un mínimo de calidad que garantiza que el programa está funcional.

También se han realizado tests de rendimiento con la herramienta de benchmarking Bonnie++, que nos permite conocer, para un sistema de ficheros: la velocidad de lectura y escritura de datos, la cantidad de seeks por segundo y la cantidad de operaciones de metadatos por segundo que se pueden realizar en él. En nuestro caso el único valor que nos interesa es el de escritura, pues las operaciones de lectura no las gestiona nuestro sistema de ficheros, simplemente las relega al sistema operativo. Desde el primer vistazo a estos valores, advertimos que era negativos. Obviamente, trabajar con una capa extra entre el sistema de ficheros real y las acciones del usuario conlleva un overhead importante. Debemos tener en cuenta que estamos añadiendo a cada una de estas operaciones la carga adicional de tener que recoger los datos, tratarlos y cifrarlos, para después guardarlos en el disco. Al final, esto acaba penalizando mucho el rendimiento del sistema, sobre todo cuando se realizan muchas

llamadas concurrentes. En el caso de Bonnie++ el problema es aún más acuciante, pues realiza millones de escrituras de un sólo carácter en un archivo. Se ha comprobado, a la hora de reproducir los cambios realizados en el sistema por Bonnie++, que la mayoría de mensajes tienen un payload de 30 bytes. A esto tenemos que sumarle los 24 bytes del header, lo que da un total de 54 bytes para guardar un cambio de 1 solo byte. Esto se traduce a que, respecto a las pruebas realizadas en el sistema operativo, las operaciones de escritura en el sistema de ficheros han sido 200 veces más lentas en promedio. Aunque es un problema importante, este se ve exagerado por las operaciones que realiza Bonnie++.

6 Conclusiones

En este trabajo se ha visto cómo se desarrolla un programa capaz de crear su propio sistema de ficheros y trabajar con datos cifrados. En primer lugar, se ha diseñado el formato de un fichero que contiene los cambios realizados sobre un sistema de ficheros. Para ello se ha hecho un estudio de los datos que serían necesarios para recrear estos cambios, tanto datos específicos del sistema de ficheros como datos generales usados para entender qué hay en cada mensaje. Se ha hecho un análisis de amenazas y se ha completado el diseño del formato del archivo basándose en las conclusiones obtenidas en él. Además, se ha diseñado e implementado un programa dividido en módulos, que están interconectados y que trabajan de manera cooperativa para conseguir el resultado que deseamos. Para ello se ha creado un sistema de ficheros propio, aprendiendo así a trabajar sobre herramientas tan potentes e interesantes como FUSE. La libertad que ofrece a la hora de desarrollar y la cantidad de aplicaciones que tiene este módulo son increíbles. También se ha aprendido a usar un módulo de criptografía llamado PyCrypto y se han aplicado las conclusiones obtenidas en el análisis de amenazas para implementar un sistema criptográfico que cumple con los requisitos de seguridad y privacidad que se habían definido.

Además, creemos que el análisis de seguridad realizado puede ser útil para los desarrolladores de aplicaciones que comunican datos por Internet, en general. Obviamente a quien esté diseñando o implementando una aplicación similar le resultará más útil, pero nunca está de más comprobar qué preocupa a otros desarrolladores. Es es muy posible que alguna vez se dé el caso de que alguien descubra así algo en lo que no había pensado antes, si no es por las conclusiones obtenidas en este trabajo, podría ser porque estas hayan inspirado una búsqueda o investigación sobre el tema en cuestión.

Esperamos que este trabajo incite la curiosidad de los lectores y que, con suerte, alguien proponga mejoras para este trabajo y continúe avanzando en él, pues es una idea muy interesante que no parece estar aprovechándose mucho. Quizá el ejemplo más notable por su popularidad sea el de Mega, pero esa misma filosofía se puede y quizá se debería aplicar a muchos otros servicios.

6.1 Líneas de continuación

En primer lugar, este trabajo deja claramente abierta una puerta a implementar una aplicación basada en el paradigma cliente-servidor y que use el programa de este proyecto. Con esto se podría tener una aplicación que envíe los cambios al fichero en tiempo real. Además, se pueden producir conflictos si varios usuarios trabajan en paralelo y después envían los cambios a un mismo servidor. El primero en enviar los datos usaría el vector de inicialización de la última versión del servidor y sería correcto, pero el segundo también estaría usando ese vector de inicialización y debería usar el del mensaje del otro usuario. Por lo tanto, sería muy útil un servidor que otorgara turnos o tokens para que cada usuario envíe sus datos de forma ordenada. Mejor aún sería que la aplicación pudiera resolver los posibles conflictos automáticamente, usando los IDs del parent, pero en este caso se necesitaría que un nodo central de confianza procesara estos conflictos y, al ser externo a la máquina local, deberíamos realizar algunas comprobaciones para saber que es de confianza.

También habría que seguir mejorando el proyecto, pues en el estado actual se han encontrado problemas que no han podido ser solucionados en el tiempo del que se disponía. Entre otros, está el problema de la sincronización que se comenta en el párrafo anterior y el preocupante problema de rendimiento provocado por los cálculos necesarios para generar y añadir el contenido de los mensajes al fichero que ya se comentó en el apartado número 5, el de resultados.

Por último, sería muy buena idea dedicarle tiempo a implementar el resto de operaciones que se salían del alcance del proyecto. Quizá el uso de enlaces simbólicos o de atributos extendidos pueda ser muy útil para un posible usuario de la aplicación, por lo que sería aconsejable desarrollar y testear esas operaciones también.

Agradecimientos

Antes de terminar este artículo, me gustaría expresar mi agradecimiento a mi tutor, Ian Blanes, por los consejos y la guía que me ha ofrecido, sin los cuales este proyecto no habría podido sortear algunos de los obstáculos con los que se ha encontrado.

Referències

[1] Dropbopx, Wikipedia, 2016. [Online]. Disponible en: https://en.wikipedia.org/wiki/Dropbox_(service) [Fecha de acceso: 21-Jun-2016].

[2] Comparison of file synchronization software, Wikipedia, 2016. [Online]. Disponible en: https://en.wikipedia.org/wiki/Comparison_of_file_synchronization_software [Fecha de acceso: 21-Jun-2016].

[3] V. G. Cerf and R. E. Kahn, Protocol for Packet Network Intercommunication. IEEE, 1974.

[4] The reference implementation of the Linux FUSE (Filesystem in Userspace) interface, GitHub, 2016. [Online]. Disponible en: https://github.com/libfuse/libfuse [Fecha de acceso: 21-Jun-2016].

[5] R. Hasan, S. Myagmar, A. J. Lee, W. Yurcik, Toward a Threat Model for Storage Systems. University of Illinois, 2005.

[6] A. J. Menezes, P. C. van Oorschot and S. A. Vanstone, Handbook of Applied Cryptography. CRC Press, 1996, pp. 228–233.

[7] Practical malleability attack against CBC-Encrypted LUKS partitions, J. Lell, 2016. [Online]. Disponible en: http://www.jakoblell.com/blog/2013/12/22/practical-malleability-attack-against-cbc-encrypted-luks-partitions/ [Fecha de acceso: 21-Jun-2016].

[8] Python's Struct module documentation, Python Software Foundation, 2016. [Online]. Disponible en: https://docs.python.org/2/library/struct.html [Fecha de acceso: 21-Jun-2016].

[9] PyCrypto documentation, D. Litzenberger, 2016. [Online]. Disponible en: https://www.dlitz.net/software/pycrypto/api/2.6/ [Fecha de acceso: 21-Jun-2016].