Práctica 2 - Mecanismos criptográficos

Ledicia Díaz Lago

19 de abril de 2020

1. Introducción

El objetivo de esta práctica es aplicar mecanismos criptográficos para asegurar información financiera. Para ello, se procede a desarrollar una aplicación en Python, que se va a lanzar desde la terminal escribiendo la siguiente instrucción:

- CriptoFinanciera -m [cs | ds | h | vh | s | vs | cert | ca | da | ts | tsv] [-p contraseña] -i chero.xml [-ad adicionales.txt] [-o salida.cpt]

A continuación, se explican los mecanismos criptográficos utilizados:

1.1. Algoritmo de clave simétrica: cifrado y descifrado

El cifrado simétrico es un tipo de mecanismo criptográfico que utiliza una misma clave para cifrar y descifrar el mensaje. Aunque existen diferentes tipos de algoritmos de cifrado, estos se pueden dividir en función de si el cifrado es byte a byte (cifrado por flujo), o si es por bloques (cifran por bloques de diferente número de bytes). El cifrado por bloques generalmente se usa junto con un modo de operación que permita cifrar mensajes de tamaño variable.

En este caso se va a implementar el algoritmo *AES* (*Advanced Encryption Standard*), que permite cifrar por bloques fijos de 128 bits (16 bytes). Los tamaños de la clave de cifrado pueden ser de 128, 192 o 256 bits. Para ello, se ha utilizado la librería Crypto de Python, y se ha creado una clase *Cipher*, con métodos de cifrado, descifrado, y un modo de operación.

Una vez implementada la clase se va a lanzar el script utilizando el modo cypher `cs' sobre el chero que se quiere cifrar (transaction.xml):

- python main.py -m cs -p secure_password -i .\DataFiles\transaction.xml -o .\DataFiles\transaction _encrypted.txt

En donde “transaction_encrypted.xml” es el resultado de cifrar el chero inicial, y “secure_password” es la contraseña para cifrar.

Para descifrar el chero se lanza el script en el siguiente modo:

- python main.py -m ds -p secure_password -i .\DataFiles\transaction_encrypted.txt -o .\DataFiles\transaction_plain.xml

En donde “transaction_plain.xml”" es el resultado de desencriptar el chero inicial, y “secure_password” contiene la contraseña utilizada para descifrar el mensaje, que es la misma que se utilizó para cifrarlo.

Si se compara el chero original “transaction.xml” con el fichero “transaction_plain.xml” se ve que son idénticos.

1.2. Cálculo y verificación de la función resumen

Para poder comprobar que un mensaje no ha sido manipulado se utilizan las funciones resumen o hash. Este tipo de funciones producen una cadena de bits de longitud fija a partir de una entrada de longitud variable, y son muy seguras ya que es muy fácil computar el hash de un mensaje, pero muy improbable obtener el mensaje a partir del hash.

Para implementar este mecanismo criptográfico en la aplicación se ha creado una clase *Hasher*, que contiene las funciones necesarias para calcular el hash del fichero utilizando el algoritmo SHA256 y comprobar que el hash calculado para el archivo coincide con el hash del archivo original.

Una vez programada la clase *Hasher*, las siguientes líneas de comando permiten calcular la función hash del archivo y comprobar si esta coincide con una función hash dada:

- **Cálculo del Hash:** python main.py -m h -i .\DataFiles\transaction.xml -o .\DataFiles\transaction_hash.hash
- **Verificación del Hash:** python main.py -m vh -i .\DataFiles\transaction.xml -ad .\DataFiles\transaction_hash.hash

Cuando se comprueba que el hash calculado (para el chero `transaction.xml') y guardado en "transaction_hash.hash" coincide con el hash del archivo `transaction.xml' se obtiene, evidentemente, que son iguales:

```
(base) C:\Users\HP\Documents\Master_FinTech\BlockChain\Tema3\Practica2Python>python main.py -m vh -i .\DataFiles\transaction.xml -ad .\DataFiles\transaction_hash.hash
shasum a1412661e256346f84407757ba9b0175b1ad8eb514d545442248499ccbecb1df
text hash a1412661e256346f84407757ba9b0175b1ad8eb514d545442248499ccbecb1df
Hash verificado
```

1.3. Firmar la información y verificar la firma

Las funciones hash son realmente útiles cuando se complementan con la firma electrónica, ya que de esta manera el receptor del mensaje puede comprobar el origen y la integridad de los datos.

En este caso se ha optado por utilizar el algoritmo RSA para generar una clave pública y su correspondiente clave privada.

Para generar la firma se ha creado la clase *Signer*, que contiene las funciones necesarias para firmar la función hash con la clave privada, y veri car comprobando el hash de mi documento coincide con el hash que se obtiene al descifrar con la clave pública.

Una vez implementada la clase *Signer*, la firma del chero y su posterior verificación se van a realizar mediante las siguientes instrucciones:

- **Firma:** python main.py -m s -p private_key.pem -i .\DataFiles\transaction.xml -o .\DataFiles\trasaction_signed.signature
- **Verifiación:** python main.py -m vs -p public_key.pem -i .\DataFiles\transaction.xml -ad .\DataFiles\trasaction_signed.signature

Para generar el chero de firma se introducen como argumentos el chero firmar y la clave privada de quien desea firmarlo. Esto genera un archivo "transaction_signed.signature" que contiene la firma de este documento con esa clave.

Para veri car que el fichero no ha sido modificado (integridad), y que el firmante es quien dice ser (autenticación) el chero de entrada será en este caso el que se quiere comprobar, y la clave será la calve pública necesaria para descifrar el chero firmado.

Cuando se veri ca la firma con los ficheros indicados, se obtiene un mensaje diciendo que la firma ha sido veri cada correctamente.

1.4. Certificado digital: cifrado y descifrado asimétricos

Un Certificado Digital consta de una pareja de claves criptográficas, una pública y una privada, creadas con un algoritmo matemático, de forma que aquello que se cifra con una de las claves sólo se puede descifrar con su clave pareja.

La clave privada se utiliza para realizar la firma electrónica, por lo tanto, el titular del certificado debe mantener bajo su poder la clave privada, ya que si ésta es sustraída, el sustractor podría suplantar la identidad del titular en la red. En este caso el titular debe revocar el certificado lo antes posible, igual que se anula una tarjeta de crédito sustraída.

La clave pública forma parte de lo que se denomina Certificado Digital en sí, que es un documento digital que contiene la clave pública junto con los datos del titular, todo ello firmado electrónicamente por una Autoridad de Certificación, que es una tercera entidad de confianza que asegura que la clave pública se corresponde con los datos del titular.

A continuación, se va a crear un certificado electrónico que contenga la clave pública asociada la clave privada que se utiliza para firmar, mediante el estándar X509.

- python main.py -m cert -o .\DataFiles\Cert_Priv_and_Pub_key_gen.cert
- python main.py -m cert -p .\DataFiles\private_key.pem -o .\DataFiles\Cert_from_PrivKey.cert

En el primer caso se ejecuta el programa sin aportar una clave privada, por lo que la clase se encarga de generar el par de claves pública y privada y de guardarlos en dos documentos *'cert_privKey.pem'* y *'cert_pubKey.pem'*. En el segundo caso se pasa como argumento el fichero que contiene la clave privada para generar la clave pública y meterla en el certificado.

Además, también es posible definir un cifrado asimétrico o híbrido en este caso, en el que se cifra el mensaje simétricamente, pero la contraseña que cifra y descifra el mensaje se cifra con una clave pública y se descifra con una privada:

- python main.py -m ca -p secure_password -i .\DataFiles\transaction.xml -o .\DataFiles\transaction_hybrid_encrypted.txr
- python main.py -m ds -p secure_password -i .\DataFiles\transaction_hybrid_encrypted.txt -o .\DataFiles\transaction_hybrid_decripted.xml

En donde 'secure_password' es la contraseña utilizada para el cifrado simétrico. La clase *Hybrid_Cipher* ya tiene incorporadas las rutas al par de clave pública/privada; no obstante, estas rutas se pueden cambiar llamando al constructor con las nuevas rutas.