



Criptografía y Seguridad Informática
Grado de Ingeniería en Informática. Curso 2022-2023

Entregable 2. Desarrollo de una Aplicación que Utilice Criptografía.

Grupo reducido: 84
ID del grupo de prácticas: 01

Práctica realizada por los alumnos:

Carlos Iborra Llopis: 100451170@alumnos.uc3m.es
Miguel Castuera García: 100451285@alumnos.uc3m.es
Alejandro García Berrocal: 100451059@alumnos.uc3m.es

1. Propósito y funcionamiento de la aplicación

El propósito de la aplicación es permitir al usuario **ingresar dinero en su cuenta bancaria de manera segura**. Para ello se hará uso del cifrado híbrido (simétrico + asimétrico), la función hash, la firma digital, certificados digitales, un método de autenticación, y todas las mejoras posibles que se han podido aplicar.

El usuario podrá acceder a su cuenta bancaria utilizando su **token único personal** proporcionado por el banco, accediendo de esta manera a una página para las transacciones. En esta página el usuario podrá introducir la cantidad de dinero que desea ingresar y al pulsar enviar esta cantidad será añadida al dinero que contenga el usuario en su cuenta bancaria. Tras esto, el usuario será notificado con un mensaje confirmando su transacción.

```
database.json > ...
{
  "usuario": "José García Martín",
  "token": "e770708a8b682abd84de7851950e00479563c4edb57c2af0e77001b28c49887f",
  "dinero": 4870
},
{
  "usuario": "Carlos Suárez López",
  "token": "41da787004e0e2adc3a9499de0bc90ddbb5e487a115dad827582e01ca3cf012",
  "dinero": 50120
},
{
  "usuario": "Miguel Ángel García Cebrán",
  "token": "524aaddf27ebcebd61de1ebb099f20fd2a6ce161c53fc1520bfba2bd868f0b9",
  "dinero": 20
},
{
  "usuario": "Alberto Fernández Cuesta",
  "token": "d0e448d6a0fe0b4ba5781258b73c349bef455ebd287a05a182f5d400df8f643",
  "dinero": 11812
}
```

Para la prueba se han almacenado en la base de datos cuatro **usuarios** distintos con sus **respectivos nombres**, sus **tokens cifrados** con el hash SHA-256 y la **cantidad de dinero que contienen en su cuenta bancaria**. Para almacenar cada uno de los datos de los usuario se utiliza una base de datos de tipo .json, ya que es un formato muy parecido a lo que serían las **bases de datos NoSQL**, que era lo que teníamos pensado hacer en un principio debido a su comodidad y al tipo de datos que queríamos almacenar.

Al ingresar en la página el **usuario deberá introducir su token** el cual se recibe como input y se cifrará con un hash seguro (**SHA-256**) dando lugar al hash de su token. **Si este hash coincide** con el token de alguno de los usuario de la base de datos se le **redireccionará** a una página en la que se le pedirá que cantidad de dinero desea ingresar. En caso contrario se mostrará un error indicando que el token no corresponde al de ningún usuario y por tanto se le denegará el acceso.

La cantidad de dinero que desee ingresar el usuario será cifrada y descifrada a través del **cifrado simétrico AES, modo EAX**, que hace uso de una llave común. Esta llave común será compartida por un **canal seguro**, en nuestro caso a través del **cifrado asimétrico RSA**. Este cifrado asimétrico solo nos servirá para enviar la clave del cifrado simétrico y así establecer la conversación entre el usuario y el banco (hacemos esto ya que el cifrado simétrico es mucho más seguro y rápido que el asimétrico, por tanto solo utilizamos el asimétrico para pasar entre usuarios la previamente mencionada clave simétrica).

Después de recibir el mensaje, el banco **comprueba que la cantidad de dinero introducida es válida y correcta**, en otras palabras, son dígitos numéricos válidos y positivos. El **dinero se ingresará en la cuenta bancaria** y se **actualizará en la base de datos con la cantidad actual**.

Si por el contrario, la **cantidad es incorrecta**, es decir, contiene caracteres, dígitos negativos u otros errores posibles, se mostrará un mensaje de **error**.

Previo a todo el proceso de envío del mensaje, tendrá lugar la **creación de la firma y el certificado**. Para ello, hemos decidido usar **ECDSA** (Elliptic Curve Digital Signature Algorithm) y así crear unas claves distintas a las usadas en el cifrado asimétrico RSA (esta decisión se explicará más adelante).

Por un lado, la **firma tomará el mensaje** inicial antes de cifrarse, hará un **hash** de este (no es necesario, pero siempre es mejor extremar la seguridad cuando se trata de un banco y de sus clientes, además de

Por otro lado, el **certificado será autofirmado usando el formato X.509**. Para este, usaremos los siguientes datos: el **nombre del usuario** (con **varios atributos**), la **clave pública del usuario**, un **número de serie aleatorio**, una fecha de **validez puesta en el mismo instante de su expedición** y una **de caducidad** (puesta a 30 minutos, al ser una transacción bancaria simple), y por último una **extensión**, **definida por el local host**.

Tras llevarse a cabo todo el intercambio de la clave con el asimétrico y el descifrado del mensaje usando el cifrado simétrico, se **verificará la firma usando el mensaje obtenido**, la firma anterior y la clave pública del usuario. Se comparan los hash de los dos mensajes y si todo es correcto la firma habrá sido verificada. Además, se verificará el certificado.

Todo esto nos sirve para asegurarnos de que **cumplimos con todos los objetivos de la seguridad** vistos en el *Tema 1: Introducción a la seguridad informática*. Estos son la **confidencialidad** (solo se da acceso a los usuarios autorizados), **disponibilidad**, **integridad** (se hacen comprobaciones de que el mensaje no ha sido modificado en ningún momento), **autenticación** (verificamos la identidad y que un usuario es quien dice ser) y por último, el **no repudio** (aseguramos mediante la firma de que no se pueden negar ciertos movimientos o hechos).

Además, como se verá más adelante, hemos llevado a cabo otras medidas de seguridad extra. Como la generación pseudoaleatoria de todas nuestras claves en cada conexión, lo que ayuda a ahorrar en espacio de servidores (al ser un banco con muchos clientes, nos ahorramos el guardar cientos de millones de claves de longitud nada desdeñable) y aporta una gran capa de seguridad extra.

Por constatar lo mencionado con un ejemplo, digamos que el usuario válido “José García Martín” accede con su token (1822312231):

[illegible]

Posteriormente ingresa una cantidad de 100€ en su cuenta bancaria; entonces la base de datos se actualiza con la nueva cantidad de dinero imprimiendo los siguientes mensajes del proceso por terminal:

[illegible]

2. Cifrado asimétrico

Se ha decidido utilizar el **cifrado asimétrico** o de clave pública que permite la distribución de claves de forma **fácil y segura**, ya que la clave que se distribuye es la pública manteniéndose la privada para el uso exclusivo del propietario.

Este cifrado a pesar de que es más lento, no requiere que se intercambien claves previamente y permite la autenticación del mensaje, el intercambio seguro de claves, el no repudio y detectar las manipulaciones sobre lo compartido.

Además, este cifrado aporta la seguridad que no aporta el cifrado simétrico pero **sería lento cifrar y descifrar todos los mensajes entre el usuario y el banco de esta forma**. Por ello, se ha decidido utilizar el **cifrado asimétrico únicamente para compartir la clave simétrica** y así aprovechar la velocidad que *AES modo EAX* proporciona para cifrar los mensajes a compartir entre el banco y el usuario.

Más específicamente, la aplicación hace uso de **RSA**, el sistema de cifrado asimétrico más utilizado rápido de todos ellos. Este sistema usa una fórmula matemática basada en el factor de integración para cifrar el mensaje. La clave pública es un producto de la multiplicación de dos números primos muy grandes y la única forma posible de descifrar los datos es conociendo los 2 factores primos del que se compone el enorme producto.

Hasta el momento, **no existe un método para descifrar un RSA con más de 768 bits de longitud** de clave; en lo que al banco implica, se ha decidido usar una clave de 2048 bits para minimizar al máximo las posibilidades de que un atacante pueda descifrar satisfactoriamente la información que se comparte en los sistemas de transacciones del banco. Este RSA es uno de los más seguros, utilizado actualmente por organizaciones gubernamentales y empresas.

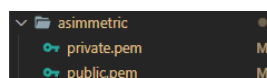
La **función que cumple** este sistema de cifrado en la aplicación es la de **distribuir la clave simétrica** que usarán el banco y el usuario **de la manera más segura posible de forma que permita al usuario ingresar dinero en su cuenta bancaria sin riesgos**. De esta manera un atacante solo puede acceder a la clave pública del banco y no a la privada. Esto hace prácticamente imposible que pueda obtener la simétrica con la que se comparten la información el banco y el usuario.

2.1. Generación y almacenamiento de las claves

Se genera una **clave pública de 2048 bits aleatoria de tipo RSA** que se usa después para **generar la privada con el *passphrase*** que es "23456".

Ambas **claves son almacenadas de forma que no estén en el código** y para el almacenamiento de ambas claves se utilizan **archivos .PEM**. La principal característica de este software es que **nos permite utilizar todos los núcleos de nuestra CPU** por lo que el **procesamiento** de las claves será lo más **rápido** posible.

```
llave = RSA.generate(2048)
module = "23456"
privada_banco = llave.export_key(passphrase=module)
```



Por último, es importante recalcar el hecho de que **solo se van a necesitar dos ficheros para las claves**, ya que tanto la clave simétrica compartida (que se crea implícitamente) como las claves asimétricas se generarán pseudoaleatoriamente y no será necesario almacenar más claves en lo

referente a cifrado simétrico + asimétrico. Por lo que reduce en gran medida la cantidad de datos que se han de almacenar y además añade una capa extra de seguridad al ser las claves válidas tan solo una vez. Esta idea se corroboró con el siguiente paper: [Symmetric key encryption using a simple pseudo-random generator](#)

2.2. Cifrado

Primero **se cifra la clave simétrica (AES modo EAX de 256 bits)** con la **clave pública del banco almacenada en *public.pem***. Para hacerlo se importa la clave pública, se crea el objeto RSA y se cifra la clave simétrica usando este objeto.

```
def cifrado_asimetrico(publica_banco, aes_key) -> bytes:
    """ Función para cifrar la clave simétrica """
    key = RSA.importKey(publica_banco)
    cipher_rsa = PKCS1_OAEP.new(key)
    key_cifrada = cipher_rsa.encrypt(aes_key)
    return key_cifrada
```

2.3. Descifrado

Se **descifra la clave simétrica (AES modo EAX) usando la clave privada del banco almacenada en el archivo *private.pem***. Para ello se importa la clave privada usando el *passphrase* con el que se ha generado la clave privada, se crea el objeto RSA y se descifra con el objeto la clave simétrica cifrada.

```
def descifrado_asimetrico(privada_banco, aes_key_cifrada, module="29456") -> bytes:
    """ Función para descifrar la clave simétrica """
    key = RSA.importKey(privada_banco, module)
    cipher_rsa = PKCS1_OAEP.new(key)
    key_descifrada = cipher_rsa.decrypt(aes_key_cifrada)
    return key_descifrada
```

Tras este proceso de distribución de la clave, tanto el usuario como el banco tienen la misma clave simétrica y pueden continuar con el proceso de ingresar dinero a través del cifrado simétrico.

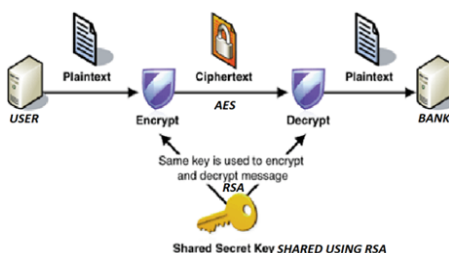
```
llave simétrica aleatoria: b'\xae\x87\x98\x95\xfb\x9c\x9a\xce,\xc1(\t\xff\xd7\x96i\x92\x08\xe6\xeaa\xcd8\xfb\x04-\nR\xea0'
```

```
llave simétrica cifrada: b'0\x3\x00\x3\x01\t\tb\x02\x82-\x922*\x04(kopI-Y \x15\x1\x87\xcd\xfb\x0f\x00\xce\x775@x9a\x00 mcT2\x98F\xfd)+cc\x12\x1\x7\xfd\xcc
a-\x982\x04\x03:\x8d\x1c\x17\x08\x03\x06\x05\x09\x15$|C\xff\x03\x02(d \x90\x18\x12\x05*8A1\x0e\t\x04\x98a8\x06\x06)\x01\x78\x03*p4q\x06\x09\x06\x1f)
\x08\x02\x00\x04\x05\x01\ng\x00\xfb\x0b\x0a\x05_\x04\x09\x08+\x06)\t\tb\x03\x01f\x0d\x01|\xf0\x03\x01\x00\x00\x01_\x0c87\x0d\x00\x0648p0\x1b\x06\xfd\x08\x06
\x0b\x02\x0e\x04\x0e3\x01\x09\x0d7\x0488v9a\x0f0\|\x0a-\x10\x04\x08\x05\x02\x01\x02\xff\x02\x06\x0f\x0576\x04\x02\x0c1\x077/$\x0ff1\x06-\x0by\x19\t\t\xcc\x09
\x022\x0b\x01\x00\x04\x05\xfb7\x02h\x0e3v\x079a1\x04\x0b'
```

```
llave simétrica descifrada: b'\xae\x87\x98\x95\xfb\x9c\x9a\xce,\xc1(\t\xff\xd7\x96i\x92\x08\xe6\xeaa\xcd8\xfb\x04-\nR\xea0'
```

2.4 Cifrado Híbrido

Es por todo ello que se dice que la aplicación usa un **cifrado híbrido (RSA + AES)** ya que el **intercambio de la clave simétrica se realiza mediante un cifrado asimétrico** y el **resto de los mensajes se cifran de manera simétrica**. Esto es especialmente útil cuando se desea enviar información a distintos destinatarios. Este cifrado proporciona a nuestra aplicación más seguridad y velocidad que los cifrados simétrico y asimétrico por separado, ya que se aprovecha de las ventajas que ambos nos ofrecen. Lo que lo hace **perfecto para un banco** que tiene un **gran número de usuarios**, y por tanto, un **gran flujo de mensajes** (transacciones) que además requieren de una **altísima seguridad**.

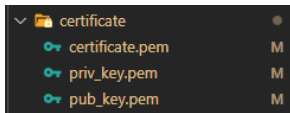


Es decir, la ventaja principal de esta combinación de métodos de encriptación es que se establece un canal de conexión entre los conjuntos de equipos de dos usuarios, y que su resultado es la seguridad adicional del proceso de transmisión junto con el rendimiento general mejorado del sistema.

3. Firma y certificado

3.1. Generación y almacenamiento de las claves

En lo referente a la implementación de firma y certificado, primero se generarán las **claves pública y privada** para la **firma digital** y el **certificado** del usuario. Estas **claves se generan** usando la curva **elíptica secp256k1** (la misma de la que hace uso **Bitcoin**), **primero se genera la clave privada y la pública se creará a partir de la privada recién creada**.



Estas claves, al igual que con el cifrado asimétrico RSA, **se almacenan en dos ficheros formato .PEM** (junto al certificado que se creará más adelante). Uno destinado a la clave privada y otro a la pública, por lo que sólo se necesitará crear dos ficheros, ya que las claves (y consecuentemente el certificado) se volverán a **generar pseudoaleatoriamente** con cada conexión entre un usuario y el banco. Esto **ahorra espacio en la base de datos** y además **añade seguridad**, como se ha mencionado previamente.

3.2. Firma

Para firma del mensaje se utiliza la clave privada del usuario, pero antes de firmar el mensaje, como se comentó en la introducción, se ha optado por hacer una **función resumen del mensaje**, es decir, un hash del mensaje. Así pues, **en caso de interceptación** de la firma, el **atacante tan solo obtendría el hash y no el mensaje**. Y **en caso de modificación**, al hacer la comprobación, **el hash del mensaje resultaría distinto** (lo que hace que en caso de robo o modificación, el hash del mensaje no servirá de nada a los atacantes debido a las propiedades de los hash seguros, **SHA-256**, como se comentó en el entregable 1) lo que mejora en gran medida la seguridad.

Asimismo, otra utilidad que brinda el hacer un hash del mensaje es que con ello nos aseguramos de que por muy largo que sea el mensaje, la **firma siempre** se hará sobre el hash de tan solo **8 palabras de 32 bits de longitud** (en el caso de **SHA-256** utilizado). Esto **reduce y agiliza en términos de eficiencia el proceso de la firma**, al tratar siempre firmas de un mismo tamaño relativamente bajo. Finalmente, tras hacer el hash del mensaje, se firma con la clave privada del usuario haciendo uso del algoritmo **ECDSA**.

```
# ? Usamos la clave privada del usuario para firmar el mensaje
signature = sign_msg_cryptography(msg_b, priv_key)
print(f"Firma del mensaje: {signature}\n")

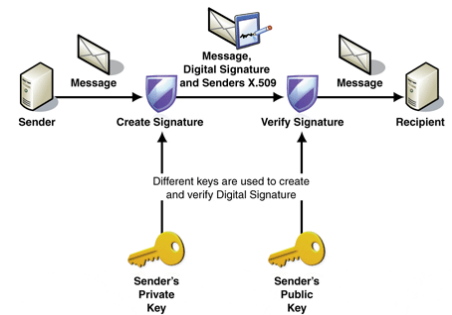
# Función para firmar el mensaje
def sign_msg_cryptography(msg, priv_key):
    """ Función para firmar el mensaje """
    # ? Hasheamos el mensaje - SHA3-256 (simplifica la longitud posible del mensaje)
    hashed_message = hash(msg)
    # ? Firmamos el mensaje con la llave privada del banco y el mensaje hasheado - ECDSA
    # Usamos secp256k1 como curva elíptica, la misma que usa Bitcoin
    signature = priv_key.sign(hashed_message.encode(
        'utf-8'), ec.ECDSA(hashes.SHA256()))
    return signature
```

3.3. Certificado

Tras la firma, se crea un **certificado con la clave privada del usuario**, dónde se usa el token introducido por el usuario al identificarse y así añadir más elementos al certificado (tanto el **nombre del usuario como el token identificativo** de este), esto es **obtener la información necesaria con la que se creará el certificado utilizando ECDSA y el formato X.509 como constructor**.

```
# ? Usamos la clave privada del usuario para certificar el mensaje
certificate_sign(priv_key, token_usuario)
```


El emisor y receptor han de ser el mismo usuario (al ser autofirmado), además, el **certificado contendrá** a parte de la **información del usuario** previamente mencionada, la clave pública de dicho usuario, el **número de serie del certificado** que se ha generado aleatoriamente, su **fecha de validez desde** el momento de expedición, **una fecha de validez hasta** pasados **30 minutos** desde la expedición y una **extensión** que está definida y configurada para su uso en el localhost. Finalmente se firma el certificado y se guarda en un fichero de formato **.PEM** destinado al guardado de certificados como mencionamos anteriormente.

[illegible]

A modo de inciso, la **idea detrás de poner una fecha de validez de 30 minutos** se debe a que **seguimos el patrón de la generación pseudoaleatoria de las claves con las ventajas que esto nos brinda**. Por ello, decidimos darle un tiempo de validez de 30 minutos ya que al tratarse de transacciones bancarias, la **comunicación** entre las partes será **breve** y no hará falta usar certificados longevos. Además en vez de guardar un solo certificado mucho tiempo, se **autogeneran unos certificados nuevos por cada comunicación debido a las nuevas claves pseudoaleatoriamente creadas** por lo que carece de sentido almacenar estos certificados. Así se asegura que no se darán **errores al renovar los datos** de un usuario del banco necesarios en nuestros certificados (lo cual sería un escenario plausible), **ni que estos certificados sin apenas uso** constante en las bases de datos, lo que de nuevo, ofrece un aumento en enorme de los costes de almacenamiento.

3.4. Verificación

Por otro lado se compartirán las claves del cifrado simétrico usando *RSA* y se cifrará y se descifrará el mensaje con *AES modo EAX* como se comentó en la primera práctica. Tras obtener el banco el mensaje en claro, este **verificará la firma utilizando la clave pública y el hash del mensaje obtenido** tras el proceso de compartición mediante el cifrado híbrido, esto es, el cifrado simétrico más el asimétrico.

Finalmente, **se verifica que el certificado sea correcto y válido** con la **clave pública del usuario**. En caso de ser correcto se **comprobará que la cantidad indicada a transferir es válida** y se realizarán las acciones pertinentes para **ingresar el dinero en la cuenta del usuario**.

```
def función para verificar el certificado digital hecho en ECDSA
def verify_certificate(sign):
    """ Función para verificar el certificado digital """
    with open("certificate/certificate.pem", "rb") as f:
        certificate = x509.load_pem_x509_certificate(
            f.read(),)

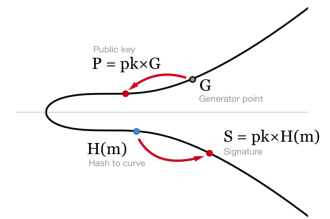
    with open("certificate/pub_key.pem", "rb") as f:
        public_key = serialization.load_pem_public_key(
            f.read(),)

    print("Verificando certificado digital con la llave pública del usuario...")
    try:
        public_key.verify(
            certificate.signature,
            certificate.tbs_certificate_bytes,
            ec.ECDSA(certificate.signature_hash_algorithm),
        )
    except Exception as e:
        print("Certificado digital verificado con éxito\n")
        return True
    except Exception as e:
        print(f"Certificado inválido: {e}")
        return False
```

```
# Función para verificar la firma
def verify_sign_cryptography(msg, signature, pub_key):
    """ Función para verificar la firma """
    # ? Hashemos el mensaje - SHA3-256 (simplifica la longitud posible del mensaje)
    hashed_message = hash_msg(msg)
    # ? Verificamos la firma con la llave pública del banco y el mensaje hashado - ECDSA
    try:
        pub_key.verify(signature, hashed_message.encode(
            'utf-8'), ec.ECDSA(hashes.SHA256()))
        return True
    except Exception:
        return False
```

3.5. Razones

ECDSA - Elliptic Curve Digital Signature Algorithm se usa para crear **firmas digitales** de datos y que nos permite **verificar la identidad de la persona que lo firma**. Es un mecanismo basado en **criptografía asimétrica**, en el que se generan dos claves, una privada y otra pública que están relacionadas por una compleja operación matemática realizada sobre una **función de curva elíptica (Secp256k1 en el caso de Bitcoin)**. Esta función selecciona un número de la curva y lo multiplica por otro originando un nuevo punto en la curva. Esto a nivel criptográfico es significativamente importante debido a que se deben tener altos conocimientos de curva elíptica logarítmica, y aún sabiendo el primer punto de la curva, hay escasa probabilidad que se averiguase el nuevo punto de la misma. Es un sistema tan complejo y completo que prácticamente **no se han llevado a cabo avances significativos en romperlo desde que se introdujo en 1985**.



Garantiza firmas únicas e irrepetibles para cada conjunto de claves públicas y privadas y la imposibilidad de falsificar las firmas digitales ya que la potencia computacional necesaria es inalcanzable. Por eso es un **estándar seguro de firmas digitales**.

Además, es aplicable en prácticamente todos los campos informáticos, por ejemplo, tal es su importancia que el precursor de la tecnología *blockchain*, el *Bitcoin* lo utiliza.

Bitcoin la ha sido **inspiración** para usar este tipo de cifrado, al ser este un proyecto que ha demostrado ampliamente su **robustez en cuanto a seguridad** y aguante como activo; y al ser esto un banco, se ha visto como una **buena referencia** a seguir en cuanto a medidas de seguridad; es por esta misma razón que se ha decidido **usar su propia curva elíptica, Secp256k1**.

Security (In Bits)	RSA Key Length Required (In Bits)	ECC Key Length Required (In Bits)
80	1024	160-223
112	2048	224-255
128	3072	256-383
192	7680	384-511
256	15360	512+

La gran ventaja que ofrece **ECDSA** frente a RSA es que sus **claves** ofrecen un **mismo nivel de seguridad** siendo estas mucho **más pequeñas**. Esto es muy útil ya que **mejora el rendimiento y la seguridad** al tener que tratar con claves de menor tamaño.

En otras palabras, **las llaves más pequeñas permiten generar firmas digitales más rápidas, certificados de menor tamaño y se necesitan menos datos para realizar conexiones TLS** que resulta en **tiempos de carga y de conexiones más rápidas**.

Comparando ambos, **ECDSA es más seguro, tiene procesos de verificación y firma un 40% más rápidos que RSA, certificados de menor tamaño, carga más rápida de sitios web y apoya los estándares gubernamentales para la protección de datos de la información**.

Acerca de los certificados, se ha mencionado que en la implementación se utiliza **X.509 como constructor**, este es un **formato estándar para certificados de clave pública** que asegura pares de claves criptográficas en entidades como el banco de esta aplicación o el usuario. Se introdujo en 1988 y ha sido adaptado para uso de Internet por la PKI de IETF (Grupo de Trabajo de Ingeniería de Internet). Este hecho distingue una clave criptográfica de un certificado digital. La **importancia de este formato reside en establecer parámetros para identificar al propietario del certificado** como puede ser el **emisor, fecha de emisión y caducidad**, entre otros. Además, considera los conceptos de validación, rutas de certificación y revocación. Es por esto que sin la definición de este estándar sería imposible establecer y operar una PKI adecuadamente.

4. Autenticación

En lo referente al método de autenticación elegido, se ha decantado por usar un **método basado en algo que tenemos**, particularmente un **token**, ya que con él podemos simular la última capa de un 2FA (Two Factor Authentication), dónde el usuario tras haber metido su contraseña, recibe (o por correo electrónico o por SMS) una clave (token) que es la que se introduce en nuestra app.

Se ha decidido así ya que de esta forma, pese a la mala elección de contraseña que pueda hacer el usuario, el banco se asegura de proporcionarle un código lo suficientemente largo y aleatorio para que sea lo suficientemente difícil de romper mediante fuerza bruta.

En este caso, el banco usa **tokens de longitud 10**, compuesto por cifras numéricas que van del 0 al 9 (como en el ejemplo de la introducción, dónde José García Martín accedió con el token 1822312231 que se le proporcionó), este token identifica indistinguiblemente a cada usuario, y por seguridad, se **almacenará en la base de datos** habiéndose hecho previamente un hash seguro **SHA-256** del token.

5. Mejoras

5.1. Almacenamiento de claves de en base de datos

Como se mencionó en la introducción y en el entregable 1, se hace uso de un archivo *.json* dónde se almacena el hash de los tokens e información relevante de los usuarios como su dinero y nombre completo. Por otro lado, las **claves no se almacenan** al utilizar la generación pseudoaleatoria de claves, esto carecería de sentido ya que dichas **claves** solo tendrán **un uso** y tras este, serán revocadas. Lo que nos ofrece ciertas ventajas como se ha ido explicando a lo largo de este documento.

5.2. Uso de distintos modos de operación

Para el sistema simétrico se usa *AES modo EAX* como se explicó en el entregable 1 debido a todas las propiedades que aporta a la aplicación bancaria como el uso de memoria constante para procesar flujos de datos, el procesamiento de datos asociados estáticos, mínima expansión del mensaje, uso de CTR, ...

5.3. Validación de datos del usuario

En la aplicación bancaria, se validan todos los datos que introducen los usuarios. Antes de establecer una comunicación, **se valida el token** que ha introducido el usuario con nuestra base de datos de los usuarios (con el hash SHA-256 del token). Luego, tras haber recibido el mensaje y descifrarlo, se **valida** que el usuario ha introducido un **mensaje válido** (esto es, que sea un número positivo y aceptable). Además, si durante el proceso el usuario introduce cualquier dato no válido o esto deriva en **excepciones**, la **aplicación se bloqueará y devolverá un error** como medida de seguridad.

5.4. Otras mejoras

Durante la realización del proyecto siempre se han buscado distintas mejoras que se podrían llevar a cabo para mejorar la seguridad, velocidad y eficiencia de nuestra aplicación. Por nombrar algunas, hacemos **hash del mensaje dentro de la firma**, todos los hash son seguros del tipo **SHA-256**, usamos **modelos asimétricos con claves distintas** tanto para el envío de la clave simétrica, dónde usamos **RSA** como para las **firmas y los certificados**, dónde usamos **ECDSA**; siempre hacemos hash de los elementos privados a proteger (tokens o mensajes), primamos la seguridad de los activos de nuestros clientes sobre la velocidad de nuestra aplicación (como por ejemplo al utilizar *AES modo EAX*), creamos **pseudoaleatoriamente todas nuestras claves para cada conexión** entre el usuario y el banco, e.t.c.

5. Anexos

Anexos de la primera entrega que nos han sido de ayuda para la realización de esta segunda entrega:

- [EAX Mode - Whitepaper](#)
- [The EAX Mode of Operation - iacr.org](#)
- [EAX Mode - Crypto++](#)
- [EAX mode - Crypto Wiki](#)
- [EAX Mode - Wikipedia](#)
- [Web App using Flask - Digital Ocean](#)
- [Cifrado AES-128 vs AES256 - redeszone.net](#)
- [AES256 - profesionalreview.com](#)
- [AES256 bits como funciona - hardzone.es](#)
- [Funciones Hash - economia3](#)
- [¿Qué es y para qué sirve un hash? - Grupo Atico34](#)

Anexos añadidos a partir de esta segunda entrega:

- [Understanding How ECDSA Protects Your Data - Instructables](#)
- [Digital Signatures, ECDSA - cryptobook.nakov](#)
- [ECDSA vs RSA: Everything You Need to Know - sectigostore.com](#)
- [¿Qué es el algoritmo de firma ECDSA? - bit2me.com](#)
- [starkbank/ecdsa-python: A lightweight and fast pure Python ECDSA library - github.com](#)
- [Cryptography 39.0.0.dev1 documentation](#)
- [Symmetric key encryption using a simple pseudo-random generator - researchgate.net](#)
- [ECC vs RSA: Comparing SSL/TLS Algorithms - Cheap SSL Security](#)
- [Criptografía asimétrica - wikipedia.org](#)
- [RSA - wikipedia.org](#)
- [Encryption - boxcryptor.com](#)
- [Claves RSA - ionos.es](#)
- [Presentación sobre criptografía - Medialab Prado](#)
- [X.509 Reference - Cryptography 39.0.0.dev1 documentation](#)
- [RSA - Cryptography 39.0.0.dev1 documentation](#)
- [cryptography.x509.name - Cryptography 3.2 documentation](#)
- [Elliptic curve cryptography - Cryptography 39.0.0.dev1 documentation](#)
- [Un mayor nivel de seguridad con certificados SSL ECDSA - plusplushosting.net](#)
- [Qué es ECDSA curva elíptica - academy.bit2me.com](#)
- [Curva Elíptica Bitcoin, por que utilizar ECDSA - oroyfinanzas.com](#)
- [Fundamentos sobre Certificados Digitales, el estándar X.509 y estructura de certificados - Security Art Work](#)
- [Qué es un certificado x509 - ssl.com](#)
- [What Are The Different Types Of Certificate Formats? - The Sec Master](#)
- [¿Qué es el cifrado híbrido? - Techopedia](#)
- [5 ejemplos de sistemas híbridos - Diego Gascón Méndez](#)
- [Funcionamiento de RSA - juncotic](#)