
Universidad
Carlos III de Madrid

# Criptografía y seguridad informática

## G24 - Entregable 2

**Javier Martín Pizarro: 100495861@alumnos.uc3m.es**

**Alberto Pascau Sáez: 100495775@alumnos.uc3m.es**

**Raúl Armas Seriña: 100495746@alumnos.uc3m.es**

**GitHub: Albrtito/CriptCript.git**

# Índice



# 1. Propósito de la aplicación. Estructura interna

## 1.1. Propósito de la aplicación

La aplicación simula una página web, levantada en local por el propio usuario, en la que se pueden crear desafíos criptográficos(*challenges*) con algoritmos de cifrado clásico.

A la hora de crear un desafío cada usuario podrá elegir el algoritmo con el que cifrar su desafío.El objetivo de otros usuarios será resolver que algoritmo de cifrado se ha utilizado y el valor de la clave del cifrado; para ayudar en el descifrado de desafíos la applicación implementará herramientas de criptoanálisis simples.
Cada desafío sera público o privado, dependiendo de la elección que haga su autor en su creación.

- **Desafío público:** cualquier usuario registrado tiene acceso a ellos.
- **Desafío privado:** solamente pueden ser compartidos con un único usuario. El creador del desafío selecciona qué usuario será capaz de verlo.

El propósito de esta aplicación es generar un sistema informático que cumpla unos requisitos mínimos. Nótese que a medida que la práctica avance, esta lista podrá verse modificada. Los requisitos para esta primera entregá incluyen las necesidades mínimas de la práctica. son:

1. El sistema debe de ser capaz de registrar y autenticar usuarios. Guardando su información confidencial correctamente cifrada(contraseñas).→**Requisito de confidencialidad**
2. El sistema debe ser capaz de permitir un inicio de sesión donde se sea capaz de obtener y comparar los datos cifrados de los usuarios de la base de datos con los proporcionados por el cliente.
3. El sistema debe de ser capaz de guardar y recuperar desafíos usando cifrado simétrico/asimétrico. → **Requisito de confidencialidad**
4. El sistema debe de ser capaz de permitir a un usuario $A$ leer el desafío creado por el usuario $B$ si este se lo ha compartido.
5. El sistema debe de autenticar que los mensajes mandados por usuarios a la base de datos no han sido alterados y son de quien dicen ser → **Requisito de Integridad y Autenticidad**

6. Para cumplir con el requisito 5 el sistema ha de implementar alguna forma de MAC o cifrado autenticado

7. El sistema debe de ser capaz de mostrar en pantalla todos los desafíos, privados y públicos, específicos para cada usuario.

8. Para cada usuario registrado, el sistema debe de crear un certificado emitido por una entidad $ADMIN$.

9. Para cada mensaje, debe de crearse una firma digital que verifique que el mensaje no ha sido modificado por terceros. → **Requisito de Integridad y Autenticidad**

## 1.2. Estructura interna

Esta aplicación consta de tres partes fundamentales:

- **Frontend:** esencial para la experiencia de usuario. Actua como interfaz entre el cliente y el backend.
- **Backend:** donde se encuentra la API, implementada usando Flask. Todos los mecanismos de cifrado, autenticación y *hasheo* se encuentran en este directorio.
- **Base de datos:** creada usando MariaDB SQL debido a su simplicidad. Actualmente la base de datos utiliza 3 bases de datos: mensajes, firma digital y certificados. De esta manera simulamos un entorno lo más independiente posible respecto al resto, aumentando la seguridad de nuestra aplicación. Puede ver un ejemplo en el primer anexo.

La base del backend y de la base de datos han sido tomadas desde el repositorio Open Source **backend-builderplate** del alumno y participante en esta práctica Javier Martín, una iniciativa que permite agilizar y automatizar el proceso de levantar una API y una base de datos que complemente a una interfaz de usuario. La estructura del código de este proyecto hereda del repositorio original.[1]

**El uso de cada parte y su inizialización usando docker se recoge en el archivo $README.md$ del repositorio, así como una explicación similar a esta de la estructura del proyecto.**

### 1.2.1. Frontend

La carpeta del frontend contiene todo lo relacionado con la visualización de la web, ‘html‘, ‘css‘, ‘javascript‘. Los estilos (css) y scripts(Javascripts) tienen sus propias subcarpetas, el html se encuentra directamente bajo la carpeta frontend debido a que son pocos archivos y es ahí donde se inicializa el servidor del frontend.

Actualmente no hay protección frente a un ataque de búsqueda de urls, cualquiera puede acceder a todo el contenido de la carpeta frontend desde la web.

### 1.2.2. Backend

La carpeta del backend contiene los archivos de ‘app.py‘ y ‘requirements.txt‘ que recogen la creación de la api e instalación de los requisitos necesarios en el servidor de backend. El resto del código se encuentra bajo src. Aquí diferenciamos entre tres carpetas:

- **mariaDB** → Métodos relacionados con la conexión a la DB

[1] Repositorio original: `https://github.com/jmartinpizarro/backend-builderplate`.

- **utils** → Clases ocupadas de la autenticación, encriptación, generación de claves y hasheado, además del manager ocupado de utilizar todas esas clases para cifrar/descifrar y autenticar mensajes 'MessageManager.py'. La clase que conforma la firma digital y las funciones de certificados también se pueden encontrar aquí.

- **unittest** → Collección de test para comprobar que todas las clases funcionan correctamente. Actualmente solo se han implementado test para el HMAC pero se establecerá como objetivo para la segunda entrega crear test para el resto de clases.

- **routes** → colección de rutas estáticas para la API de Flask con la que realizaremos ciertas funciones dependiendo de las acciones que realice el usuario desde la interfaz.

De estas cuatro carpetas utils es la que interesa para evaluar los métodos criptográficos implementados. Esta carpeta se ha divido según el propósito del algoritmo que representa cada clase, creando secciones para algoritmos de **autenticación, encriptación, generación de claves, cifrado autenticado, generación y verificación de hashes y cifrados cásicos, firma digital y certificados**.

- **Todos estos algoritmos criptográficos han sido implementados usando la librería de python cryptography**

**NOTA:** Para esta entrega solo se utilizan algoritmos de cifrado simétrico como AES y cifrado asimétrico como RSA (ya que es el estándar frente a DSA). En general, el sistema usa un cifrado híbrido que añade un extra de seguridad a los protocolos usados.
También usaremos RSA para firma digital y certificación, además de HMACs para aportar integridad a los mensajes de los usuarios.

## 1.3. Flujo de código

**A fecha de la entrega final encontramos el siguiente flujo en el código de la aplicación:**

- Inicializamos las bases de datos con el certificado ADMIN (autoridad certificadora), usuario y contraseña y claves para la firma digital.

- Cuando el usuario se registra, se crea un usuario y una contraseña correctamente hasheada, además de sus claves de firma y un certificado firmado por la entidad ADMIN (autoridad certificadora).

- En la creación de mensajes, se cifra el contenido y también se firma para poder comprobar la integridad (y autenticidad) más adelante.

- A la hora de renderizar los mensajes, comprobamos que la clave pública del usuario que creó el mensaje se verifique con su certificado, que la firma del mensaje sea la correcta y por último desciframos (no sin antes comprobar si el HMAC es el correcto).

### 1.3.1. Flujo de código general

Asumimos que el usuario $A$ ya ha sido creado y ha iniciado sesión. Ha creado un mensaje privado a un usuario $B$. El siguiente diagrama describe el flujo del código ejecutado para cifrar el mensaje, generar una firma digital y el resto de lógica en lo que respecta a la criptografía:

## 2. Firma Digital

La firma digital es usada para verificar la integridad y autenticidad de los mensajes que los usuarios han creado y que se van a renderizar.

Inicializamos una base de datos separada a la anterior (donde almacenábamos la información de los usuarios y los mensajes), simulando un provedor externo e independiente.

Para la firma, se usa el algoritmo RSA. Se pensó en usar DSA, pero ya que RSA es el estándar de la industria, el código ha sido diseñado para ser lo más verídico posible respecto a la realidad.

En esta base de datos, llamada *digital_firm*, existen dos tablas distintas:

- secure_keys encargada de almacenar una clave pública y una clave privada cifrada para cada usuario (totalmente independientes a otras bases de datos)
- digital_signatures la firma digital generada por cada mensaje que ha creado un usuario

Para ello recogemos tres funciones en el archivo *DigitalSignManager.py*:

- generate_rsa_keys() generamos un par de claves relacionadas entre ellas: una clave pública y una clave privada en formato PEM binario.
- create_signature(): dada una clave privada PEM y un mensaje, firmamos dicho mensaje.
- verify_signature(): dada una clave pública PEM, un mensaje y una firma, comprobamos si dicha firma es verdadera o no.

De nuevo, mantenemos una estructura lo más modular posible, intentando seguir un estándar apropiado para el correcto entendimiento del código.

Si bien las funciones no tienen ningún *log* cuando son definidas, en otras partes del código donde son invocadas sí que se hace un *log* para enseñar los valores obtenidos.

Firma digital es usada especialmente en la renderización de mensajes, ya que al hacer una *query* a la base de datos, obtenemos una lista con todas las filas disponibles.

Para cada mensaje, comprobamos si la firma que se relaciona con ese mensaje es la misma que la que nosotros generamos. Si no es así, enviaremos un mensaje en la terminal diciendo que dicho mensaje no se verifica correctamente, y que por lo tanto ha sido eliminado de la lista de renderizado (el usuario al que iba dirigido no podrá verlo).

## 3. Certificados

Los certificados son usados ampliamente en la industria para comprobar datos y aportar autenticidad e integridad a mensajes o interacciones con el sistema.

En este proyecto, también se ha implementado una jerarquía de entidades certificadoras para poder realizar estas acciones correctamente. La jerarquía es la siguiente:

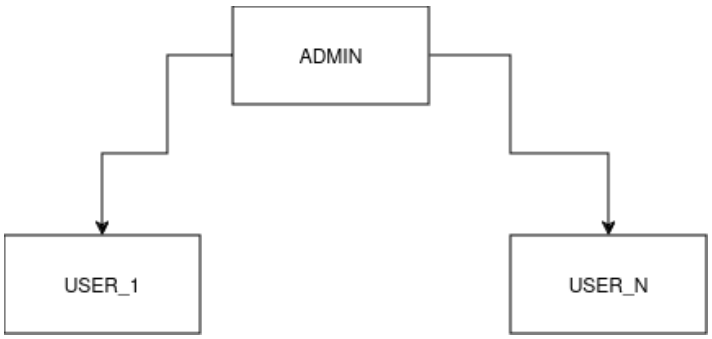


Figura 1: Jerarquía de entidades certificadoras

La entidad certificadora **ADMIN** es generada en la inicialización del proyecto. Para ello, tenemos una base de datos totalmente inpendiente que se encarga de guardar los certificados creados.

El certificado de esta entidad es firmado por sí misma, por lo que el usuario asume que es una entidad verífica y confiable.

Estos certificados son usados para verificar que la clave pública que se usa en firma digital es efectivamente la clave oficial de un usuario, y no una clave que pueda suponer un ataque a confidencialidad o veracidad de la información.

Para cada usuario que se registra en la aplicación se genera un certificado que es firmado por la entidad emisora **ADMIN**. Se guarda la clave pública generada para la firma como clave pública del certificado.

Antes de renderizar cada mensaje, comprobamos que el certificado es correcto y que la información es la correcta. Tras esto, procedemos con el flujo previamente mencionado (firma digital, descifrado, *response*).

Si la información del certificado obtenido no es correcta (es decir, que la verificación falla), eliminamos el mensaje de la lista de renderizado y emitimos un *log* en la terminal para que quede constancia del error que ha ocurrido en la verificación.