Universidad
Carlos III de Madrid

Criptografía y seguridad informática

G24 - Entregable 1

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 application 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.

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 tablas *users* ,*private_challenges* y *public_challenges*.Estas tablas se recogen 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 ‘requirementes.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
- **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‘.

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

- **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.

Finalmente, bajo la carpeta src se encuentran también los archivos *name_routes.py* que contienen el routeado para la comunicación de la aplicación con el frontend

De estas tres 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**.
En sta carpeta también podemos encontrar la clase MessageManager; encargada de cifrar y autenticar los mensajes con el algoritmo correcto.

NOTA: Para la primera entrega solo se utilizan las clases de AESManger para cifrar y MACManager para autenticar, no obstante la el flujo información entre clases está ideado para que otros algoritmos puedan se utilizados, incluso algoritmos de cifrado autenticado como fernet.
Haciendouso de esta estructura para la entrega final, el usuario será capaz de elegir con que algoritmos cifrar

2. Autenticación de usuarios

Durante el registro de los usuarios, tanto el usuario como la contraseña de este son *hasheados* e introducidos en la base de datos.Esto asegura que nunca se guardarán contraseñas o nombres de usuario en texto plano, manteniendo la confidencialidad de dicha contraseña y usuario fuera del alcance de atacantes.
Este método de guardado permite que comprobemos la autenticidad de un usuario o contraseña a traves de una comparación de igual con el hash que tenemos guardado. Sabemos que dado el mismo texto en claro debemos siempre de obtener el mismo Hash.

$$H(M) = Hash$$

Entonces, a la hora de iniciar sesión o trabajar con los usuarios en la base de datos, deberemos hacer un *hash* para más tarde compararlo. Si dicho *hash* es igual al esperado, podremos trabajar con él, autenticando que es el usuario introducido es el correcto.

Por supuesto, siempre existe el riesgo de las **colisiones** tal que $H(M) = H'(M) =$ colisión, pero es algo que es complicado que ocurra, ya que las funciones resumen están diseñadas para minimizar estos posibles errores.

2.1. Implementación con SHA256

La implementación del hasheado de usuarios se ha hecho utilizando la librería *hashlib* de python. Esta libreria ofrece multitud de funciones hash, de las que hemos elegido SHA256 debido a que es una de las más seguras actualmente.

El código que permite tanto crear como verificar hashes se encuentra en la clase *HashManager* con direccion *./backend/src/utils/HashManger.py*.Capturas de las funciones de creación y verificación se encuentran en el segundo anexo

3. Cifrado y descifrado de mensajes

Para el cifrado de los mensajes y otros datos importantes se ha decantado por un cifrado simétrico, específicamente *AES128*. Para mantener una estructura de código simple y eficiente, se ha desarrollado

una clase *CipherManager* que incorpora varios métodos, de especial importancia:

- hkdf_expand: usada para expandir *hashes*.
- cipherChallengeAES: usada para cifrar en AES
- decipherChallengeAES: usada para descifrar un mensaje previamente cifrado en AES

Como ya se mencionó en la sección 2, los *hashes* que se usarán para generar la clave son de 64 bits, por lo que se necesitará expandirlos hasta los 128 bits para trabajar con ellos.

Es decir, que el cifrado es dependiente de la contraseña del usuario, aportando una capa de confidencialidad bastante robusta y eficiente. De la misma manera, el descifrado ocurre usando la misma metodología, que puede verse resumida tal que:

1. Tras comprobar en el frontend que el usuario exista como usuario que ha iniciado sesión, hacemos una *request* al *backend*.
2. Desde el backend, se *hashea* el usuario que ha enviado la petición y buscamos su contraseña en la base de datos.
3. Expandimos al contraseña hasta obtener una longitud de 128 bits.
4. Aplicamos el cifrado/descifrado *AES128* y obtenemos un mensaje cifrado/descifrado.
5. Devolvemos la correspondiente *response* al frontend.

También se ha planteado una **estructura de cifrado asimétrica** que mantenga como claves privadas las claves generadas a partir de las contraseñas expandidas de los usuarios y que se utilice como clave pública una clave general para todos. Con esto conseguiríamos un nuevo grado de confidencialidad en nuestro software.

Nótese que esto último no se encuentra implementado actualmente, pero sí se espera hacerlo como una mejora en la siguiente iteración.

4. Autenticación usando MAC

5. Anexos

5.1. Tablas SQL

Tablas:

```sql
init.sql
CREATE TABLE IF NOT EXISTS private_challenges (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name_challenge BLOB NOT NULL UNIQUE,
    user VARCHAR(255) NOT NULL,
    content BLOB NOT NULL,
    auth BLOB,
    shared_user VARCHAR(255),
    FOREIGN KEY (user) REFERENCES users(username),
    FOREIGN KEY (shared_user) REFERENCES users(username)
);
```

Figura 1: Tabla de desafíos privados

```sql
init.sql

CREATE TABLE IF NOT EXISTS public_challenges (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name_challenge BLOB NOT NULL UNIQUE,
    user VARCHAR(255) NOT NULL,
    content BLOB NOT NULL,
    auth BLOB,
    FOREIGN KEY (user) REFERENCES users(username)
);
```

Figura 2: Tabla de desafíos públicos

```sql
init.sql

CREATE TABLE IF NOT EXISTS users (
    username VARCHAR(255) NOT NULL UNIQUE PRIMARY KEY,
    user_password VARCHAR(255)
);
```

Figura 3: Tabla de usuarios

5.2. Funciones Hash

```python
backend/src/utils/HashManager.py

    @staticmethod
    def create_hash(clear_text: str) -> str:
        """
        Create a hash of the password
        :param password -> password to hash
        :return -> hash of the password
        """
        return hashlib.sha256(clear_text.encode()).hexdigest()

    @staticmethod
    def verify_hash(clear_text: str, hash: str) -> bool:
        """
        Verify the inputted password against it's hash:
        :param password -> password to verify
        :param hash -> hash to verify against
        :return ->  True if hash matches password, False otherwise
        """
        return HashManager.create_hash(clear_text) == hash
```