



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: Albrito/CriptCript.git

Índice

Índice	1
1. Propósito de la aplicación. Estructura interna	1
1.1. Propósito de la aplicación	1
1.2. Estructura interna	2
1.2.1. Frontend	2
1.2.2. Backend	3
1.3. Flujo de código	3
2. Autenticación de usuarios	4
2.1. Implementación con SHA256	4
3. Cifrado y descifrado de mensajes:AES256	4
3.1. Generación y uso de claves	5
4. Códigos de autenticación(MAC)	5
5. Anexos	6
5.1. Tablas SQL	6
5.2. Funciones Hash	6
5.3. Funciones KeyGen	7
5.4. Funciones AES	8
5.5. Funciones HMAC	9
5.6. Funciones MessageManager	11

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 qué algoritmo de cifrado se ha utilizado y el valor de la clave del cifrado; para ayudar en el descifrado de desafíos la aplicación 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 complementa a una interfaz de usuario. La estructura del código de este proyecto hereda del repositorio original.¹

El uso de cada parte y su inicialización 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.

¹Repositorio original: <https://github.com/jmartinpizarro/backend-builderplate>.

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’. La clase que conforma la firma digital y las funciones de certificados también se pueden encontrar aquí.
- **unittest** → Colección 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 dividido 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 básicos, firma digital y certificados**.

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

NOTA: Para la primera entrega solo se utilizan las clases de AESManger para cifrar y MACManager para autenticar, no obstante la estructura de clases está ideada para que otros algoritmos también puedan ser utilizados, incluso algoritmos de cifrado autenticado como fernet.

Haciéndose uso de esta estructura para la entrega final, el usuario será capaz de elegir con qué algoritmos cifrar

1.3. Flujo de código

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

- Se utiliza MessageManager para cifrar y autenticar desafíos públicos o privados que se van a guardar o a cargar desde la base de datos. Esta clase llama a AES y HMAC. La implementación de esta clase se puede encontrar en anexo 6.
- Se utiliza AES256 con modo CTR para cifrar y descifrar mensajes
- Se utiliza un código de HMAC para autenticar los mensajes
- Utilizamos una clase de hasheo implementando SHA256 para guardar usuarios y sus contraseñas.
- Las clases MessageManger y HashManager son las únicas que interactúan directamente con las rutas de la API en challenge_routes.py y user_routes.py

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 la contraseña y nombre del usuario fuera del alcance de atacantes.

Este método permite que comprobemos la autenticidad de un usuario o contraseña a través de una comparación de igualdad con el hash que tenemos guardado. Sabemos que dado el mismo texto en claro debemos siempre de obtener el mismo Hash.

$$H(M) = \text{Hash}$$

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.

En esta primera entrega se han utilizado las contraseñas hasheadas de los usuarios como claves para el cifrado de mensajes de forma asimétrica, más sobre esto y la generación de claves en el punto 3.1.

2.1. Implementación con SHA256

La implementación del hasheado de usuarios se ha hecho utilizando la librería *hashlib* de python. Esta librería 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 dirección *./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: AES256

Para el cifrado y decifrado de mensajes entre el backend y la base de datos se ha implementado un **cifrado simétrico**, específicamente **AES256** con el modo CTR.

Hemos elegido el modo CTR debido a que:

- No requiere generar padding para el mensaje a cifrar
- Solo hace falta una clave y un valor nonce
- El valor del nonce no ha de ser privado sino único, permitiendo que se almacene junto al mensaje cifrado y aumentando la seguridad del cifrado

Las funciones que implementan AES son dos *encript_AES* y *decript_AES* y se encuentran dentro de la clase *AESManager* bajo *./backend/src/utils/auth/AESManager.py*. Las capturas de estas funciones se pueden ver en el cuarto anexo. Ambas funciones requieren una clave y un valor para el nonce, la obtención de estos valores se explica en la siguiente sección

Por último, debemos de mencionar el **método que se utiliza para guardar el valor del nonce**. Sabiendo que este valor puede ser público lo únicos al texto cifrado una vez realizado el AES, al realizar el descifrado lo primero que hacemos es obtener los primeros 16 bytes del mensaje para usarlos como nonce.

3.1. Generación y uso de claves

La generación y comprobación de claves se realiza dentro de la clase KeyGen. Actualmente esta clase tiene un único método que genera las claves usando las contraseñas hasheadas de los usuarios. No obstante los hashes son demasiado grandes para ser usados en el AES256 pues son de 64bytes, debido a esto el método de generación de claves de KeyGen parte a la mitad el hash del usuario y lo devuelve como clave. La captura de este método se puede ver en el tercer anexo.

Diferenciamos entre dos tipos de claves que se generan ambas a partir del hash de la contraseña de un usuario.

- La clave que se utiliza para cifrar desafíos públicos, que es siempre la misma y se obtiene usando el hash del admin.
- La clave que se utiliza para cifrar desafíos privados, que se obtendrá usando el hash del usuario que crea el desafío.

Valor del nonce: Como estamos usando el modo CTR de AES tenemos que generar un valor nonce que cumpla con que con la misma clave nunca se repita el mismo valor nonce.

Actualmente esto no esta pasando pues el valor se genera de forma aleatoria por el sistema operativo. Esto se debe a que no se ha podido implementar en el tiempo hasta la primera entrega.

La implementación usará el número de desafíos totales más uno como valor para el nonce de cada nuevo cifrado.

Futuras implementaciones:

La estructura de clases actual está pensada para que las clases que requieran una clave obtengan a través de la clase KeyGen independizando la creación de claves del resto de algoritmos. Para entregas futuras esta clase generará y almacenará claves en una nueva tabla de la base de datos para no depender de las contraseñas de los usuarios y hacer de todo el proceso algo más seguro.

4. Códigos de autenticación(MAC)

Los códigos de autenticacion se utilizan para asegurar la integridad y autenticación de los mensajes, para crearlos se utiliza HMAC pues es sencillo de utilizar e implementar. El código implementa la posibilidad de usar tanto ENCRYPT-THEN-MAC como MAC-THEN-ENCRYPT, no obstante nosotros hemos optado por utilizar la segunda

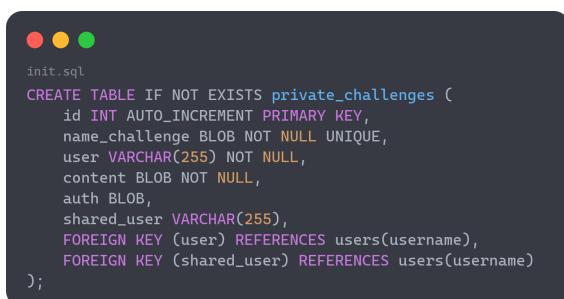
La implementacion del HMAC se realiza en la clase MACManager y se utiliza en el MessageManager. La clave utilizada es la misma que la del cifrado. Una vez creado el código se guarda junto al mensaje cifrado en la base de datos para luego poder autenticar el mensaje al traerlo de la base de datos.

El código de la clase MACManager se puede ver en el quinto anexo.

5. Anexos

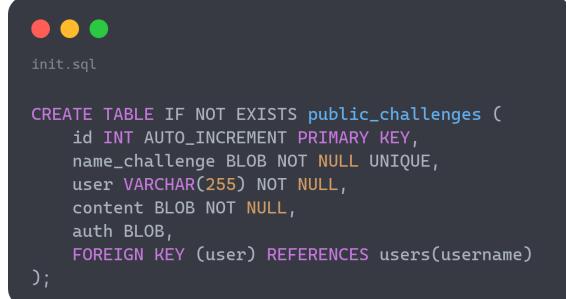
5.1. Tablas SQL

Código de inicialización del sql



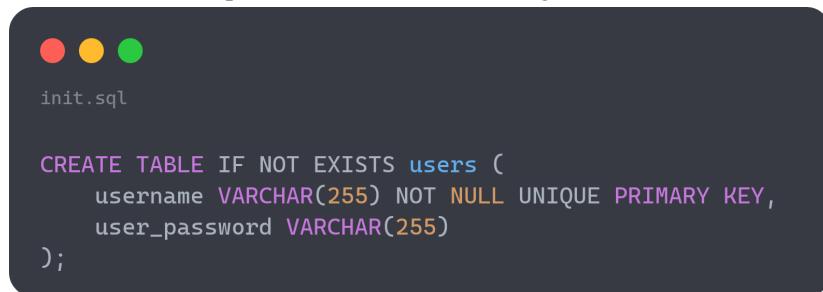
```
init.sql
CREATE TABLE IF NOT EXISTS private_challenges (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name_challenge BLOB NOT NULL UNIQUE,
    user VARCHAR(255) NOT NULL,
    content BLOB NOT NULL,
    auth BLOB,
    shared_user VARCHAR(255),
    FOREIGN KEY (user) REFERENCES users(username),
    FOREIGN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

Figura 1: Tabla de desafíos privados



```
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)
);
```

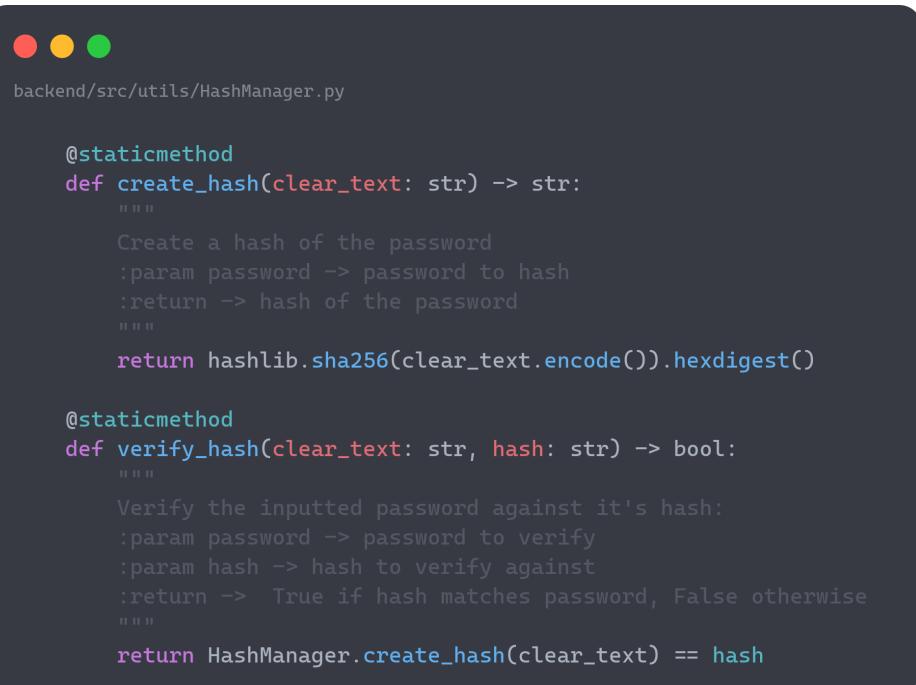
Figura 2: Tabla de desafíos públicos



```
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



```
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
```

5.3. Funciones KeyGen

```
backend/src/utils/keys/KeyGen.py

def key_from_user(user: str, length = 256) -> str:
    """
    :param user -> String with the username (in hash form)
    :param lenght -> Length of the key to generate in BITS
    :return -> Return the value of the key created
    """

    # Get the current password of the user:
    user_password = get_user_password(user)

    # The user password hash is too big to be an AES key: Get half of it
    len_password = len(user_password)
    half_password = user_password[len_password//2:]

    # Key equals to half the user password
    key = half_password

    # The key must be 32 bits long so it can later turn into a 256 bit key.
    # Check
    if len(key) == (length//8):
        return key
    else:
        raise TypeError("The key doesn't have a valid size")
```

5.4. Funciones AES

Funciones para el cifrado y descifrado con AES

```
backend/src/utils/encryption/AESManager.py

@staticmethod
def encrypt_AES(data: str, key: str, nonce = os.urandom(16)) -> bytes:
    """
    Encrypts the data with the key using AES
    :param data: The data to be encrypted
    :param key: The key to encrypt the data.
        - Either 128, 192 or 256 BITS
    :param nonce: Value for the nonce(CTR mode)
    :return: The encrypted data
    """

    # Convert the data and key to bytes
    data_bytes = data.encode()
    key_bytes = key.encode()

    # Create the AES cipher:
    cipher = Cipher(algorithms.AES(key_bytes), modes.CTR(nonce))
    # Create the encryptor. Messages going through it will be ciphered with
    # the cipher generated above.
    encryptor = cipher.encryptor()

    # Cipher the message:
    ciphered_data = encryptor.update(data_bytes) + encryptor.finalize()
    ciphered_data = nonce + ciphered_data

    # Return the ciphered message
    return ciphered_data
```

```

backend/src/utils/encryption/AESManager.py

@staticmethod
def decrypt_AES(encrypted_data: bytes, key: str) -> str:
    """
    Decrypts the data with the key using AES
    :param data: The data to be decrypted
    :param key: The key to decrypt the data
    :return: The decrypted data
    """

    # Convert the key to bytes
    key_bytes = key.encode()
    # Obtain the value for the nonce from the message
    nonce = encrypted_data[:16]
    # Subtract the nonce from the encrypted data
    encrypted_data = encrypted_data[16:]

    # Create the AES cipher:
    cipher = Cipher(algorithms.AES(key_bytes), modes.CTR(nonce))

    # Create the encryptor. Messages going through it will be ciphered with
    # the cipher generated above.
    decryptor = cipher.decryptor()

    # Cipher the message:
    data = decryptor.update(encrypted_data) + decryptor.finalize()

    # Return the ciphered message, decoded as a string

    return data.decode()

```

5.5. Funciones HMAC

Funciones de creación y verificación de claves de autenticación HMAC

```

backend/src/utils/auth/MACManager.py

@staticmethod
def create_ciphered_HMAC(ciphered_data:bytes, key: str) -> bytes:
    """
    Creates an HMAC based on the key, data and algorithm defined.
    + The algorithm is always sha256
    """

    # Convert the data and key to bytes
    key_bytes = key.encode()

    # Define the object that creates the HMAC - Object of type HashContext
    hmac_creator = hmac.HMAC(key_bytes, hashes.SHA256())

    # Create the HMAC hash:
    hmac_creator.update(ciphered_data)
    hmac_value = hmac_creator.finalize()

    return hmac_value

```

```
backend/src/utils/auth/MACManager.py
    raise ValueError("The HMAC is not correct")

@staticmethod
def verify_ciphered_HMAC(ciphered_data:bytes , key:str, hmac_value:bytes) -> bool:
    """
    Verifies the HMAC value for some key and some data
    """

    # Convert the data and key to bytes
    key_bytes = key.encode()

    # Define the object that verifies the HMAC: Object of type HashContext
    hmac_verifier = hmac.HMAC(key_bytes, hashes.SHA256())

    # Verify the HMAC hash:
    hmac_verifier.update(ciphered_data)
    try:
        hmac_verifier.verify(hmac_value)
        return True
    except:
        raise ValueError("The HMAC is not correct")
```

5.6. Funciones MessageManager

Funciones de cifrado y autenticación de mensajes

```
backend/src/utils/MessageManager.py

@staticmethod
def decipher_message(ciphered_message:bytes, key:str, ENCRYPTION_TYPE = "AES") ->str:
    """
    Decipher a given message, a key and an algorithm
    NOTE: See comments in KeyGen for future changes

    :param ciphered_message -> Ciphered message in bytes
    :param key -> String value of the key to cipher
    :param ENCRYPTION_TYPE -> Type of algorithm used to cipher
    :return -> Cleartext string
    """

    try:
        logging.info(
            f"""
            _____
            Message being deciphered: {ciphered_message}
            Key used: {key}
            _____
            """
        )
        message = AESManager.decrypt_AES(ciphered_message, key)
        logging.info(
            f"""
            _____
            Message deciphered: {message}
            Key used: {key}
            _____
            """
        )
        return message
    except:
        logging.warning("Problem while deciphering a message")
        raise Exception("Could not decipher message")
```

```
backend/src/utils/MessageManager.py
@staticmethod
def cipher_message(message:str, key:str, ENCRYPTION_TYPE = "AES") -> bytes:
    """
    Cipher a given a message, a key and an algorithm
    NOTE: See comments in KeyGen for future changes

    :param message -> Cleartext string message
    :param key -> String value of the key to cipher
    :param ENCRYPTION_TYPE -> Type of algorithm used to cipher
    :return -> Ciphered bytes
    """

    try:
        logging.info(
            f"""
            _____
            Message being ciphered: {message}
            Key used: {key}
            _____
            """
        )
        ciphered_message = AESManager.encrypt_AES(message, key)
        logging.info(
            f"""
            _____
            Message ciphered: {ciphered_message}
            Key used: {key}
            _____
            """
        )
    except:
        logging.warning("Problem while ciphering a message")
        raise Exception("Could not cipher message")
```

backend/src/utils/MessageManager.py

```
@staticmethod
def auth_create(ciphered_message:bytes,key:str,AUTH_TYPE = "HMAC") -> bytes:
    """
    Creates an authentication code for ciphered messages.
    NOTE: Using the ENCRYPT-THEN-MAC approach

    :param ciphered_message -> Ciphered message in bytes
    :param key -> String value of the key to cipher
    :param AUTH_TYPE -> Type of algorithm used to authenticate
    """

    try:
        logging.info(
            f"""
            _____
            Auth bein created: {ciphered_message}
            Key used: {key}
            _____
            """
        )
        auth = MACManager.create_ciphered_HMAC(ciphered_message, key)
        logging.info(
            f"""
            _____
            Auth created: {auth}
            Key used: {key}
            _____
            """
        )
        return auth
    except:
        logging.warning("Problem while creating auth")
        raise Exception("Could not create auth for message")
```

backend/src/utils/MessageManager.py

```
@staticmethod
def auth_verify(MAC:bytes, ciphered_message:bytes, key:str, AUTH_TYPE= "HMAC") -> bool:
    """
    Checks the authenticity of a ciphered message
    NOTE: Using the ENCRYPT-THEN-MAC approach

    :param MAC -> MAC value to verify
    :param ciphered_message -> Ciphered message in bytes
    :param key -> String value of the key to cipher
    :param AUTH_TYPE -> Type of algorithm used to authenticate
    """

    try:
        logging.info(
            f"""
            _____
            Auth being verified: {MAC}
            Cipherd message: {ciphered_message}
            Key used: {key}
            _____
            """
        )
        verify = MACManager.verify_ciphered_HMAC(ciphered_message, key, MAC)
        logging.info(
            f"""
            _____
            Verified mac: {MAC}
            Key used: {key}
            result: {verify}
            _____
            """
        )
    except:
        logging.warning("Problem while verifying auth")
        raise Exception("Could not verify auth for message")
```