Grado en Ingeniería Informática

Criptografía y seguridad informática 25/26

Grupo 81

Práctica 1

«OmniMessenger: Envío de mensajes en masa»

Denis Loren Moldovan — 100522240

Jorge Adrian Saghin Dudulea — 100522257

Profesor

Lorena Gonzalez Manzano

Tabla de Contenidos

1. Propósito y estructura

La aplicación consiste en el envío masivo de mensajes, usando servicios como Telegram (Bots), Whatsapp (Business) y Gmail (Conexión al correo correspondiente), los cuales se almacenan para luego distribuirlos de forma ordenada.

Esta consiste en un backend escrito en Python, y un frontend en HTML, CSS, y JavaScript, los cuales se comunican mediante peticiones HTTP sobre TLS. Como no se va a desplegar la aplicación en un entorno de servidor real, se ha omitido el uso de «dotfiles» para la gestión de direcciones y de tokens. Por otro lado, el uso de la dirección «localhost» está codificado fuertemente dentro del código, entonces, si se quisiese usar un dominio personalizado, habría que implementarlo dentro del código.

Para el acceso a todos los servicios, se ha usado Caddy como proveedor de archivos, y proxy reversa. Además de asegurar un cifrado en la conexión, nos habilita el uso de subdominios y subdirectorios, donde se han usado para separar, por ejemplo, el visor de la base de datos. Por otro lado, se ha usado PostgreSQL para la base de datos y Adminer para la visualización de la base de datos. Finalmente, para asegurar el funcionamiento en la mayoría de máquinas, se ha contenerizado toda la aplicación usando Docker.

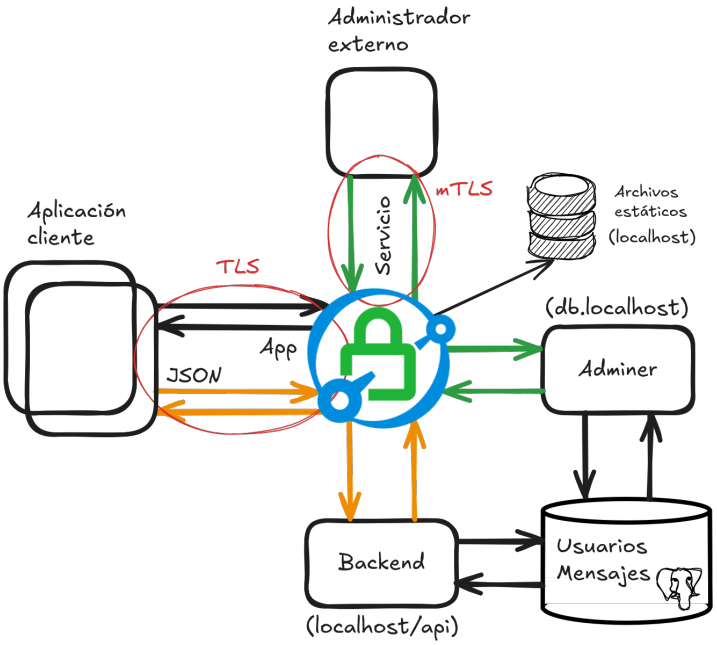

Figura 1: Diseño de la aplicación

1.1. Como ejecutar la aplicación

Para ejecutar la aplicación, primero hay que descargar y configurar Docker. Después, hay que construir la imagen personalizada usando `./setup.sh`, y finalmente ejecutar la aplicación usando `./run.sh`. Para detenerlo, hay que ejecutar `./down.sh`.[1]

La página principal se encontrará en `https://localhost`, y el gestor de la base de datos en `https://db.localhost`.[2] El certificado requerido para poder acceder al gestor se encuentra en `Docker/conf/SSL/Client/keystore.p12`, el cual hay que instalar en el navegador.[3]

[1] La aplicación ha sido desarrollada y probada en un entorno de Linux.
[2] Usuario predeterminado: «postgres» / Contraseña: «Cripto2526».
[3] Tiene una contraseña vacía

2. Operaciones criptográficas usadas

2.1. Autentificación de usuarios

La autenticación de usuarios se hace mediante usuario / contraseña, en la página de inicio de sesión `https://localhost/login`. En el caso de que el usuario no fuese autenticado con anterioridad, la raiz del dominio redirige al inicio de sesión. Toda la información de inicio de sesión se envía sobre TLS, por lo que nos aseguramos que los datos se transmiten de forma segura hasta el servidor.

La sesión del usuario se controla usando un token guardado en el almacenamiento local del navegador. Cuando este accede a la página, se envía el token de inicio de sesión al servidor, y este lo valida para que el resto de la comunicación no se rompa al realizar operaciones. Ahora bien, dicho token es generado al registrarse o al iniciar sesión, donde además se guarda la última IP de inicio de sesión, por lo que si se intenta usar el token fuera de la máquina inicial, el servidor requerirá al usuario volver a iniciar sesión.

Por otro lado, la contraseña no se guarda en texto plano. Al recibirla durante el registro, se «hashea» usando la librería de Python `bcrypt`, con la función `hashpw`, cuyo formato es:

```
$<version>$<coste>$<salt>.<hash>
```

Donde la versión indica la variante del algoritmo (e.g. $2b$), el coste, cuantas 2^{coste} veces se ejecuta el algoritmo (e.g. 12) y el salt generado de forma pseudoaleatoria. Luego para comprobar que la contraseña es correcta, se usa la función `checkpw`, la cual con la información descrita anteriormente, se «rehashea» la contraseña introducida y se comparan.

2.2. Cifrado simétrico y asimétrico

Para el cifrado simétrico y asimétrico, nos hemos basado en el protocolo TLS, gestionado automaticamente por Caddy, al cual nosotros únicamente le teníamos que proveer con certificados generados y firmados en local.[4]

Además de asegurar una conexión cifrada con el servidor, también hemos usado mTLS para acceder a la página de gestión de la base de datos, accesible desde la dirección `https://db.localhost`. Como en un entorno empresarial, solo un número muy reducido de personas tendrían acceso a esta información, mediante el uso de certificados mTLS nos aseguramos tanto confidencialidad e integridad, como autenticación.

1. Generamos la CA.

```
openssl genrsa -out ca.key 4096
openssl req -x509 -new -nodes -key ca.key -sha256 -days 3650 -out ca.crt -subj
"/C=US/ST=Local/L=Local/O=Local CA/CN=LocalDevCA"
```

2. Generamos la llave del servidor, con su respectivo certificado.

```
openssl genrsa -out server.key 2048
openssl req -new -key server.key -out server.csr -subj "/C=US/ST=Local/L=Local/
O=Local Server/CN=localhost"
openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
server.crt -days 365 -sha256 -extfile server-ext.cnf
```

3. Generamos la llave del cliente con su respectivo certificado

```
openssl genrsa -out client.key 2048
openssl req -new -key client.key -out client.csr -subj "/C=US/ST=Local/L=Local/
```

[4] En una aplicación real, usaríamos Lets Encrypt como autoridad para firmar los certificado, pero Certbot no permite la generación de certificados para IPs locales (192.168.1.0/24).

```
O=Local Client/CN=client"
openssl x509 -req -in client.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out
client.crt -days 365 -sha256
```

4. Generamos el archivo `pkcs12` del certificado del cliente para importarlo al navegador.

```
openssl pkcs12 -export -in client.crt -inkey client.key -name 'client' -out
keystore.p12
```

2.3. Códigos de autenticación (MAC)

Las funciones de códigos de autenticación de mensajes (MAC) se utilizan en la aplicación para garantizar la integridad y autenticidad de los datos transmitidos entre el cliente y el servidor, principalmente durante el intercambio de tokens de sesión y en la comunicación cifrada bajo TLS. De esta forma, se evita que un atacante pueda modificar o falsificar los mensajes sin ser detectado.

En nuestro caso, el uso de MAC está integrado de forma implícita dentro del protocolo TLS, ya que las suites criptográficas negociadas por Caddy incluyen algoritmos de cifrado autenticado, como AES-GCM (Galois/Counter Mode) o CHACHA20-POLY1305, ambos reconocidos por ofrecer simultáneamente confidencialidad, integridad y autenticación del mensaje. Estos algoritmos no requieren una gestión separada del MAC, puesto que el cálculo del código de autenticación se realiza internamente como parte del proceso de cifrado, reduciendo el riesgo de errores de implementación.

3. Pruebas realizadas

ID	Descripción	Entrada	Resultado esperado
1	Registro de usuario correcto	• Usuario: `TestUser` • Contraseña: `Password@123`	• JSON: `{ "ok": true, "user": "TestUser", "token": "<16 caracteres hex>" }`
2	Registro con usuario ya existente	• Usuario: `TestUser` • Contraseña: `Password@123`	• JSON: `{ "ok": false, "error": "User already registered" }`
3	Registro con formato de usuario inválido	• Usuario: `usr` • Contraseña: `Password@123`	• JSON: `{ "ok": false, "error": "Formato usuario/ contraseña incorrecto" }`
4	Registro con contraseña inválida	• Usuario: `UsuarioValido` • Contraseña: `password123`	• JSON: `{ "ok": false, "error": "Formato usuario/ contraseña incorrecto" }`
5	Inicio de sesión correcto	• Usuario: `TestUser` • Contraseña: `Password@123`	• JSON: `{ "ok": true, "user": "TestUser", "token": "<nuevo token>" }`
6	Login con usuario inexistente	• Usuario: `NoExiste` • Contraseña: `Password@123`	• JSON: `{ "ok": false, "error": "User does not exist" }`
7	Login con contraseña incorrecta	• Usuario: `TestUser` • Contraseña: `MalaPass@123`	• JSON: `{ "ok": false, "error": "Wrong password" }`
8	Validación de token válido	• Token: `<token válido>`	• JSON: `{ "ok": true, "user": "TestUser" }`
9	Validación de token inexistente	• Token: `abc123fake`	• JSON: `{ "ok": false, "error": "Bad user" }`
10	Validación sin token	• Sin token en cabecera `Authorization`	• JSON: `{ "ok": false, "error": "Bad token (not received)" }`
11	Validación sin token	• Sin token en cabecera `Authorization`	• JSON: `{ "ok": false, "error": "Bad token (not received)" }`
12	Envío de mensaje válido	• Token: `<token válido>` • Mensaje: `"Hola mundo"`	• JSON: `{ "ok": true }`
13	Envío de mensaje demasiado largo	• Token: `<token válido>` • Mensaje: cadena de 2001 caracteres	• JSON: `{ "ok": false, "error": "Message too long" }`