INSEGUS

CYBERSECURITY CONSULTING

Antonio Abad Correa
José Maria Solís
Pedro Lacarcel

Índice

- Portada
- Índice
- Resumen ejecutivo
- Core
- Pruebas
- Conclusión
- Bibliografía

Resumen ejecutivo

El presente informe desarrolla el Proyecto de Aseguramiento de la Información (PAI-1), cuyo objetivo es implementar un sistema cliente-servidor orientado a garantizar la integridad de las credenciales de usuario y de las transacciones electrónicas en una entidad financiera. Este trabajo se plantea como un ejercicio práctico de aplicación de los principios de seguridad en redes y sistemas, enmarcado en la asignatura de Ingeniería de Seguridad Informática.

En un entorno financiero, la transmisión y almacenamiento de datos sensibles —como credenciales de acceso o instrucciones de transferencia— representan un punto crítico de seguridad. El uso de redes públicas como Internet expone dicha información a riesgos como ataques *man-in-the-middle*, *replay*, derivación de claves y ataques de canal lateral. Por tanto, resulta indispensable diseñar mecanismos que preserven la integridad de la información y eviten la manipulación o acceso indebido.

Políticas y requisitos de seguridad

La entidad financiera ha definido dos políticas fundamentales:

1. Preservar la integridad en el almacenamiento de credenciales de usuario.

2. Mantener la integridad de las comunicaciones en todas las transacciones realizadas por medios electrónicos no seguros.

A partir de estas políticas, se establecen tres grupos de requisitos de seguridad:

- Credenciales de usuario: almacenamiento seguro mediante técnicas criptográficas, verificación confiable en el inicio de sesión y mecanismos contra ataques de fuerza bruta.

- Transacciones: aseguramiento de la integridad en la comunicación mediante el uso de MAC, NONCE, tamaños de clave adecuados y Secure-Comparator.

- Base de datos: protección de la integridad de la información almacenada, tanto de credenciales como de registros de transacciones.

Objetivos y alcance

El proyecto persigue:

- Implementar un sistema cliente-servidor que permita el registro, autenticación y gestión de usuarios.
- Diseñar un mecanismo de transmisión de transacciones que incorpore verificadores de integridad para prevenir ataques.
- Analizar alternativas para la compartición segura de claves, evaluando fortalezas y limitaciones.
- Realizar pruebas que permitan comprobar el cumplimiento de los requisitos definidos.

Metodología de implementación

El sistema se desarrolla utilizando sockets como medio de comunicación entre cliente y servidor. Las principales funcionalidades incluyen:

- Registro de usuarios: alta de nuevos usuarios con nombre y contraseña únicos.
- Inicio de sesión: validación de credenciales y establecimiento de una sesión activa.
- Gestión de sesiones: posibilidad de cierre seguro de sesión.
- Transacciones: envío de operaciones en formato *Cuenta origen, Cuenta destino, Cantidad transferida*, con almacenamiento persistente en el servidor.
- Verificación de integridad: aplicación de mecanismos criptográficos y de control para garantizar la confiabilidad de los mensajes transmitidos.

Resultados esperados

Se espera que el sistema cumpla con las políticas de seguridad definidas, ofreciendo un entorno capaz de:

- Mantener la integridad de la información en todas las fases del proceso (registro, autenticación, comunicación y almacenamiento).

- Prevenir ataques comunes asociados a la banca en línea, reforzando la confianza en la plataforma.

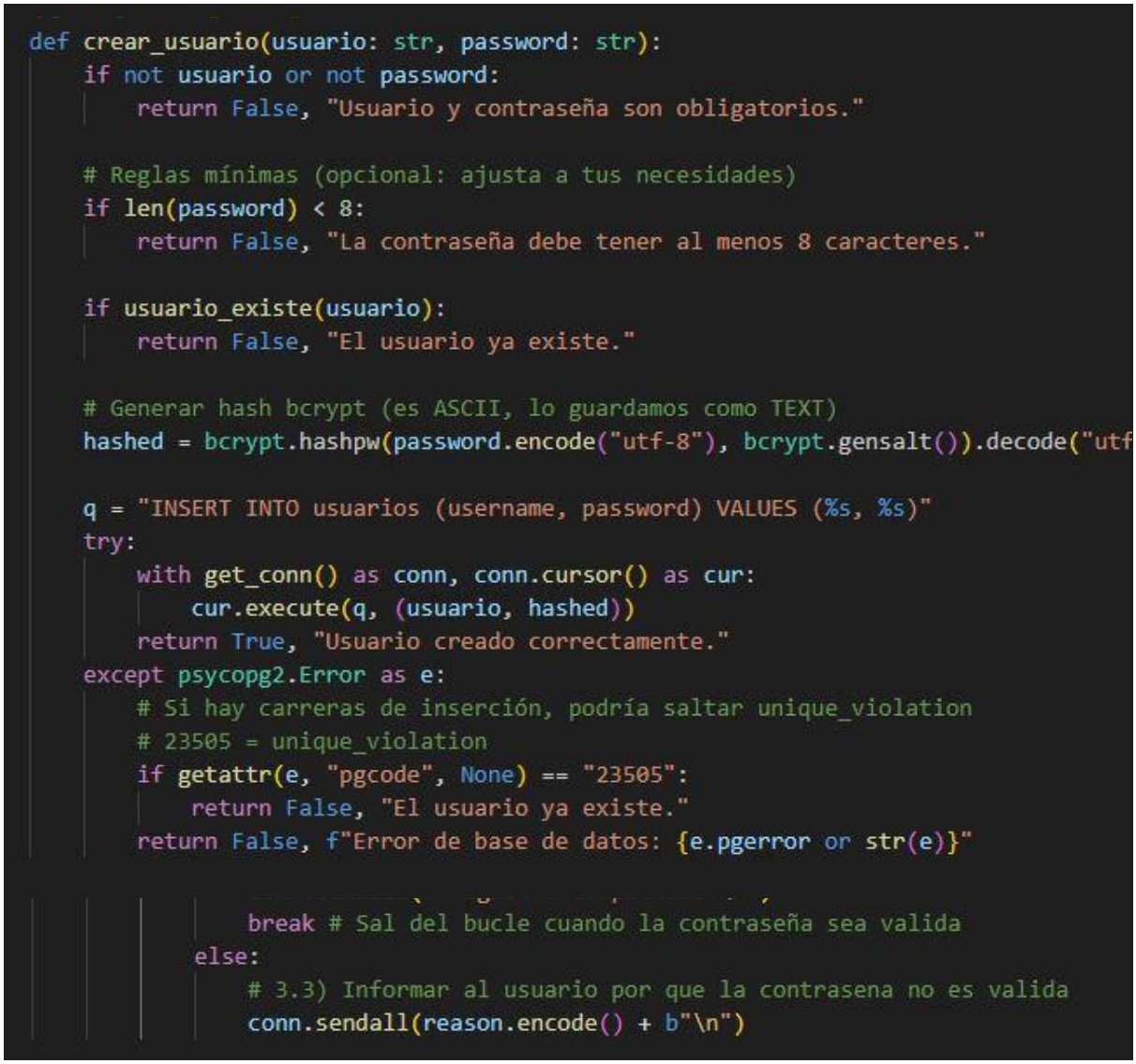
- Evaluar las técnicas de verificación de integridad, comparando su efectividad y su impacto en el rendimiento del sistema.

- Contribuir al aprendizaje académico mediante la integración de teoría y práctica en el campo de la seguridad informática.

Core

Requisitos Funcionales

1. Gestión de usuarios
    - Registro de nuevos usuarios (nombre de usuario + contraseña).
    - Validación de duplicidad en el registro.
    - Imposibilidad de modificar datos una vez creados.
    - Base de usuarios preexistentes cargada inicialmente.



```
def crear_usuario(usuario: str, password: str):
    if not usuario or not password:
        return False, "Usuario y contraseña son obligatorios."

    # Reglas mínimas (opcional: ajusta a tus necesidades)
    if len(password) < 8:
        return False, "La contraseña debe tener al menos 8 caracteres."

    if usuario_existe(usuario):
        return False, "El usuario ya existe."

    # Generar hash bcrypt (es ASCII, lo guardamos como TEXT)
    hashed = bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt()).decode("utf

    q = "INSERT INTO usuarios (username, password) VALUES (%s, %s)"
    try:
        with get_conn() as conn, conn.cursor() as cur:
            cur.execute(q, (usuario, hashed))
        return True, "Usuario creado correctamente."
    except psycopg2.Error as e:
        # Si hay carreras de inserción, podría saltar unique_violation
        # 23505 = unique_violation
        if getattr(e, "pgcode", None) == "23505":
            return False, "El usuario ya existe."
        return False, f"Error de base de datos: {e.pgerror or str(e)}"

                break # Sal del bucle cuando la contraseña sea valida
            else:
                # 3.3) Informar al usuario por que la contrasena no es valida
                conn.sendall(reason.encode() + b"\n")
```

En la gestión de usuario hemos implementado el registro de nuevos usuarios con la verificación en la base de datos. Esta verificación se hace con una llamada de comprobación en la base de datos para saber si están los datos ya incluidos. Esta función, también verifica el numero de caracteres una vez creamos la contraseña. También tenemos la base de datos precargada en formato txt.

2. Autenticación
    - Inicio de sesión con credenciales válidas.
    - Verificación de credenciales contra la base de datos.

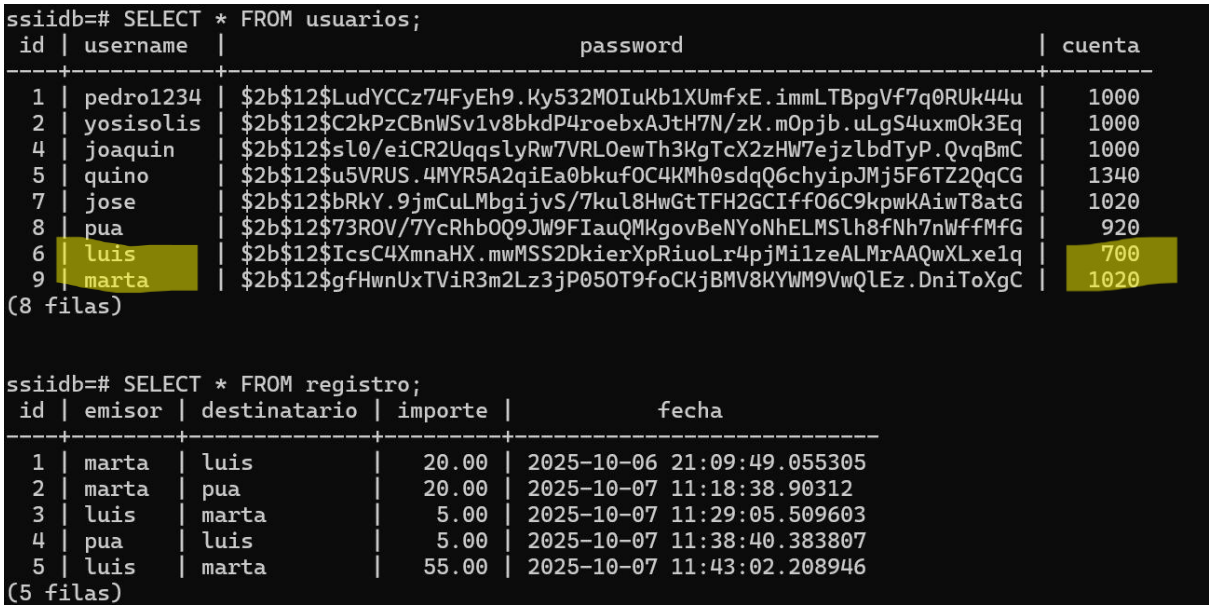
- Denegación de acceso en caso de error de autenticación.
- Cierre de sesión de usuarios conectados.

```python
# (Seguimos con el eco como hasta ahora y ya implementaremos la verificacion luego.)
elif opcion == "login":

    #pedir usuario
    conn.sendall(b"Introduce un nombre de usuario:\n")
    username = conn.recv(1024).decode().strip()

    #pedir contrasena
    conn.sendall(b"Introduce una contrasena:\n")
    password = conn.recv(1024).strip()

    #comprobar si existe y si la contrasena es correcta
    #comprueba si existe el nombre
    if username_exists(username):
        #comprueba si la contrasena es correcta
        with open(USERS_FILE, "rb") as f:
            for line in f:
                user_line, hashed_pwd_line = line.strip().split(b":", 1)

                # Comparamos el nombre de usuario
                if user_line.decode() == username:
                    # Usamos bcrypt.checkpw() para verificar la contraseña
                    if bcrypt.checkpw(password, hashed_pwd_line):
                        conn.sendall(b"Login exitoso.\n")
                        break
            else:
                #contrasena incorrecta
                conn.sendall(b"Usuario o contrasena incorrectos.\n")
```

En este apartado también tenemos la verificación de seguridad que en el anterior apartado ya que también se hace una llamada a la base de datos para comprobar si tanto el usuario y la contraseña están en para iniciar o no sesión. También tenemos implementado la posibilidad de cierre de sesión si en algún momento el usuario lo desea. La denegación de de acceso en caso de error en el inicio de sesión esta implementado de tal manera que tanto para el usuario como para la contraseña se comprueban que son correctos con el correspondiente mensaje de error que nos informa del fallo.

3. Transacciones
    - Envío de transferencias con el formato: *Cuenta origen, Cuenta destino, Cantidad transferida*.
    - Registro automático de transacciones sin validación de cuentas ni montos.
    - Persistencia de las transacciones en el servidor.

El primer paso de la función de transacción es pedir el usuario, una vez pedido si es erróneo pedirá de nuevo el usuario para proseguir con la cantidad a enviar de nuestra cuenta al usuario pedido. Como ultimo paso pide la contraseña de la cuenta para confirma la operación.



```
ssiidb=# SELECT * FROM usuarios;
 id | username  |                          password                          | cuenta
----+-----------+------------------------------------------------------------+--------
  1 | pedro1234 | $2b$12$LudYCCz74FyEh9.Ky532MOIuKb1XUmfxE.immLTBpgVf7q0RUk44u |   1000
  2 | yosisolis | $2b$12$C2kPzCBnWSv1v8bkdP4roebxAJtH7N/zK.mOpjb.uLgS4uxmOk3Eq |   1000
  4 | joaquin   | $2b$12$sl0/eiCR2UqqslyRw7VRLOewTh3KgTcX2zHW7ejzlbdTyP.QvqBmC |   1000
  5 | quino     | $2b$12$u5VRUS.4MYR5A2qiEa0bkufOC4KMh0sdqQ6chyipJMj5F6TZ2QqCG |   1340
  7 | jose      | $2b$12$bRkY.9jmCuLMbgijvS/7kul8HwGtTFH2GCIffO6C9kpwKAiwT8atG |   1020
  8 | pua       | $2b$12$73ROV/7YcRhbOQ9JW9FIauQMKgovBeNYoNhELMSlh8fNh7nWffMfG |    920
  6 | luis      | $2b$12$IcsC4XmnaHX.mwMSS2DkierXpRiuoLr4pjMi1zeALMrAAQwXLxe1q |    700
  9 | marta     | $2b$12$gfHwnUxTViR3m2Lz3jP05OT9foCKjBMV8KYWM9VwQlEz.DniToXgC |   1020
(8 filas)


ssiidb=# SELECT * FROM registro;
 id | emisor | destinatario | importe |           fecha
----+--------+--------------+---------+----------------------------
  1 | marta  | luis         |   20.00 | 2025-10-06 21:09:49.055305
  2 | marta  | pua          |   20.00 | 2025-10-07 11:18:38.90312
  3 | luis   | marta        |    5.00 | 2025-10-07 11:29:05.509603
  4 | pua    | luis         |    5.00 | 2025-10-07 11:38:40.383807
  5 | luis   | marta        |   55.00 | 2025-10-07 11:43:02.208946
(5 filas)
```

Aquí podemos ver una captura de la tabla de los usuarios con los importes previos a la transacción y de la tabla de registros de transacciones. Una vez hagamos la transacción, en nuestro caso, lo haremos siendo el usuario Luis, que mandara al usuario marta 50 euros.

```
PS C:\Users\PC1\Desktop\SSII\PAI1\Ejemplos client-server\python> python client
socket.py
Eres nuevo usuario o quieres loggearte? nuevo/login
> login
Implementando la funcionalidad de login

Introduce un nombre de usuario:
> luis
Introduce una contrasena:
> Plapino123?
Login exitoso.
Que deseas hacer?
1. Transaccion
1
Iniciando transaccion.
Introduce el nombre del destinatario:
> marta
marta
Su saldo en cuenta es:
700
Introduce la cantidad a transferir:
> 50
Por seguridad, introduce tu contrasena para confirmar la transaccion:
> Plapino123?
ack de la contrasena
ack de los datos
ack del nonce del cliente
✅ Transaccion exitosa:
   Destinatario: marta
   Cantidad: 50
```

```
ssiidb=# SELECT * FROM usuarios;
 id | username  |                          password                          | cuenta
----+-----------+------------------------------------------------------------+--------
  1 | pedro1234 | $2b$12$LudYCCz74FyEh9.Ky532MOIuKb1XUmfxE.immLTBpgVf7q0RUk44u |   1000
  2 | yosisolis | $2b$12$C2kPzCBnWSv1v8bkdP4roebxAJtH7N/zK.mOpjb.uLgS4uxmOk3Eq |   1000
  4 | joaquin   | $2b$12$sl0/eiCR2UqqslyRw7VRLOewTh3KgTcX2zHW7ejzlbdTyP.QvqBmC |   1000
  5 | quino     | $2b$12$u5VRUS.4MYR5A2qiEa0bkufOC4KMh0sdqQ6chyipJMj5F6TZ2QqCG |   1340
  7 | jose      | $2b$12$bRkY.9jmCuLMbgijvS/7kul8HwGtTFH2GCIffO6C9kpwKAiwT8atG |   1020
  8 | pua       | $2b$12$73ROV/7YcRhbOQ9JW9FIauQMKgovBeNYoNhELMSlh8fNh7nWffMfG |    920
  6 | luis      | $2b$12$IcsC4XmnaHX.mwMSS2DkierXpRiuoLr4pjMi1zeALMrAAQwXLxe1q |    650
  9 | marta     | $2b$12$gfHwnUxTViR3m2Lz3jP05OT9foCKjBMV8KYWM9VwQlEz.DniToXgC |   1070
(8 filas)


ssiidb=# SELECT * FROM registro;
 id | emisor | destinatario | importe |           fecha
----+--------+--------------+---------+----------------------------
  1 | marta  | luis         |   20.00 | 2025-10-06 21:09:49.055305
  2 | marta  | pua          |   20.00 | 2025-10-07 11:18:38.90312
  3 | luis   | marta        |    5.00 | 2025-10-07 11:29:05.509603
  4 | pua    | luis         |    5.00 | 2025-10-07 11:38:40.383807
  5 | luis   | marta        |   55.00 | 2025-10-07 11:43:02.208946
  6 | luis   | marta        |   50.00 | 2025-10-07 11:46:47.98574
(6 filas)
```

4. Persistencia de datos
    - Almacenamiento permanente de credenciales de usuario.

Usamos la aplicación PostgreSQL

    - Registro histórico de las transacciones realizadas.

Anteriormente demostramos que tenemos un registro histórico de las transacciones.

```python
# Inicializar la base de datos
def init_db():
    conn = sqlite3.connect("usuarios.db")
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS usuarios (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT UNIQUE NOT NULL,
            password TEXT NOT NULL
        )
    ''')
    conn.commit()
    conn.close()

# Función para crear cuenta y guardar contraseña encriptada
def crear_cuenta(username, password):
    hashed = bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt())
    try:
        conn = sqlite3.connect("usuarios.db")
        cursor = conn.cursor()
        cursor.execute("INSERT INTO usuarios (username, password) VALUES (?, ?)", (username, hashed))
        conn.commit()
        conn.close()
        return "✅ Usuario registrado con éxito"
    except sqlite3.IntegrityError:
        return "⚠️ El usuario ya existe"
```

5. Interfaz de comunicación
    - Conexión cliente-servidor a través de sockets.

```python
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
```

    - Interacción para registro, login y operaciones de transferencia.

Tenemos implementado las correspondientes interacción en el registro, login y transferencia. Se podrá comprobar con la correspondiente ejecución del código e interactuando con el mismo.

6. Mensajes del sistema

- Confirmación de registro de usuario.

```
if is_valid:
    # 3.1) Genera una sal y hashea la contrasena
    hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())

    # 3.2) Guardar en fichero
    with open(USERS_FILE, "ab") as f:
        f.write(username.encode() + b":" + hashed_password + b"\n")

    conn.sendall(b"Registro completado.\n")
    break # Sal del bucle cuando la contraseña sea valida
else:
    # 3.3) Informar al usuario por que la contrasena no es valida
    conn.sendall(reason.encode() + b"\n")
```

- Notificación de errores (usuario ya existente, credenciales incorrectas, etc.).

En cada una de las funciones o pasos que da el usuario el sistema tiene multitud de comentario si se produce cualquier tipo de fallo en cualquiera de los posibles casos que se pudieran dar para saber que es lo que esta fallando.

```
# Comprobar si existe
if username_exists(username):
    conn.sendall(b"Usuario ya existe. Prueba con otro.\n")
```

```
    else:
        #contrasena incorrecta
        conn.sendall(b"Usuario o contrasena incorrectos.\n")
```

- Mensajes de validación de transacciones exitosas.

```
        return True, "✅ Transacción válida"
    except Exception as e:
        return False, f" Error en verificación: {e}"
```

Requisitos de Seguridad

1. Credenciales de usuario
   - Almacenamiento seguro: las contraseñas no deben guardarse en texto plano, sino con algoritmos hash seguros (ej. SHA-256, bcrypt, Argon2).

```
# 3.1) Genera una sal y hashea la contrasena
hashed_password = bcrypt.hashpw(password, bcrypt.gensalt())
```

   - Verificación de credenciales: validación en el inicio de sesión de manera segura, sin exponer datos sensibles.

```python
def verificar_usuario(usuario: str, password: str) -> bool:
    stored_hash = _get_user_hash(usuario)
    if not stored_hash:
        return False
    try:
        return bcrypt.checkpw(password.encode("utf-8"), stored_hash.encode("utf-8"))
    except ValueError:
        # Por si el hash en DB tuviera formato inesperado
        return False
```

- Protección contra fuerza bruta: implementación de límites de intentos fallidos o retardo progresivo en accesos.

```python
# Define los requisitos para una contraseña fuerte
def is_strong_password(password):
    # La contraseña debe ser un objeto de bytes para usar len()
    if len(password) < 8:
        return False, "La contrasena debe tener al menos 8 caracteres."
    if not any(char.isdigit() for char in password.decode()):
        return False, "La contrasena debe contener al menos un numero."
    if not any(char.isalpha() for char in password.decode()):
        return False, "La contrasena debe contener al menos una letra."
    if not any(char in "!@#$%^&*()_+-=[]{}|;':\",.<>/?`~" for char in password.decode()):
        return False, "La contrasena debe contener al menos un simbolo."
    return True, ""
```

```python
def bloqueado(usuario):

    until = LOCKED_USERS.get(usuario)
    if until is None:
        return False, 0
    if now() > until:
        del LOCKED_USERS[usuario]
        FAILED_LOGINS[usuario] = 0
        return False, 0
    return True, int(until - now())


def registrar_fallo(usuario):
    """Registra un fallo de login para el usuario. Devuelve True si el usuario queda bloqueado."""
    if usuario not in FAILED_LOGINS:
        FAILED_LOGINS[usuario] = 0
    FAILED_LOGINS[usuario] += 1
    intentos_restantes = MAX_ATTEMPTS - FAILED_LOGINS[usuario]
    if FAILED_LOGINS[usuario] >= MAX_ATTEMPTS:
        LOCKED_USERS[usuario] = now() + LOCK_SECONDS
        return True, 0
    return False, intentos_restantes

def reset_fallos(usuario):
    """Resetea el contador de fallos de login para el usuario."""
    if usuario in FAILED_LOGINS:
        del FAILED_LOGINS[usuario]
```

2. Transacciones
    - Integridad en la comunicación: uso de mecanismos como MAC y NONCE para garantizar que los mensajes no sean alterados.

```python
# 1) Verificar timestamp
now = int(time.time())
if abs(now - datos["timestamp"]) > ALLOWED_DRIFT:
    return False, "Timestamp fuera de ventana"
```

```python
def datos_transaccion(valor, destinatario, nonce_server):

    nonce_client = secrets.token_hex(16)
    ts = int(time.time())

    datos={
        "cantidad": valor,
        "destinatario": destinatario,
        "nonce_client": nonce_client,
        "nonce_server": nonce_server,
        "timestamp": ts
```

```
63      # Generar hash bcrypt (es ASCII, lo guardamos como TEXT)
64      hashed = bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt()).decode("utf-8")
65
```

- Protección frente a ataques comunes: prevención de ataques de *man-in-the-middle*, *replay* y de derivación de claves.

En este apartado están las capturas de los correspondiente código en otros apartado para no repetir información.

```
# 1) Verificar timestamp
now = int(time.time())
if abs(now - datos["timestamp"]) > ALLOWED_DRIFT:
    return False, "Timestamp fuera de ventana"
```

- Tamaños de clave adecuados: selección de claves seguras según estándares actuales.


```
63      # Generar hash bcrypt (es ASCII, lo guardamos como TEXT)
64      hashed = bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt()).decode("utf-8")
65
```


```
def is_strong_password(password):
    # La contraseña debe ser un objeto de bytes para usar len()
    if len(password) < 8:
        return False, "La contrasena debe tener al menos 8 caracteres."
    if not any(char.isdigit() for char in password.decode()):
        return False, "La contrasena debe contener al menos un numero."
    if not any(char.isalpha() for char in password.decode()):
        return False, "La contrasena debe contener al menos una letra."
    if not any(char in "!@#$%^&*()_+-=[]{}|;':\",.<>/?`~" for char in password.deco
        return False, "La contrasena debe contener al menos un simbolo."
    return True, ""
```

- Uso de comparadores seguros: implementación de Secure-Comparator para verificar integridad de manera robusta.

```
9           if not hmac.compare_digest(expected, firma):
0               return False, " Firma inválida"
1
```

Estos requisitos lo tenemos implementados en el proyecto haciendo uso de la MAC, NONCE, hmac y tamaño clave 256. Hacemos uso del hash bcrypt criptográficamente seguro. Hash Bcrypt usa una cadena de texto generada por una función criptográfica unidireccional que incluye sal. Es la forma segura de guardar contraseñas.

3. Base de datos

- Integridad de la información: protección frente a alteración no autorizada de credenciales y registros de transacciones.
- Persistencia confiable: asegurar que los datos no se corrompan durante el almacenamiento o recuperación.
- Acceso restringido: limitar el acceso a la base de datos únicamente a procesos autorizados.

Para nuestra base de datos hemos utilizado postgreSQL, que cumple con los requisitos de seguridad. Implementa transacciones ACID (Atomicidad, Consistencia, Aislamiento y Durabilidad), garantizando que las operaciones sobre los datos se realicen de forma coherente y segura. Usa controles de acceso por roles y privilegios, impidiendo modificaciones no autorizadas. Permite firmas digitales y logs de auditoría (por ejemplo, con pgAudit) para detectar cambios no autorizados. Las credenciales se almacenan en archivos protegidos y se autentican mediante métodos seguros (MD5, SCRAM-SHA-256, Kerberos, LDAP, etc.).

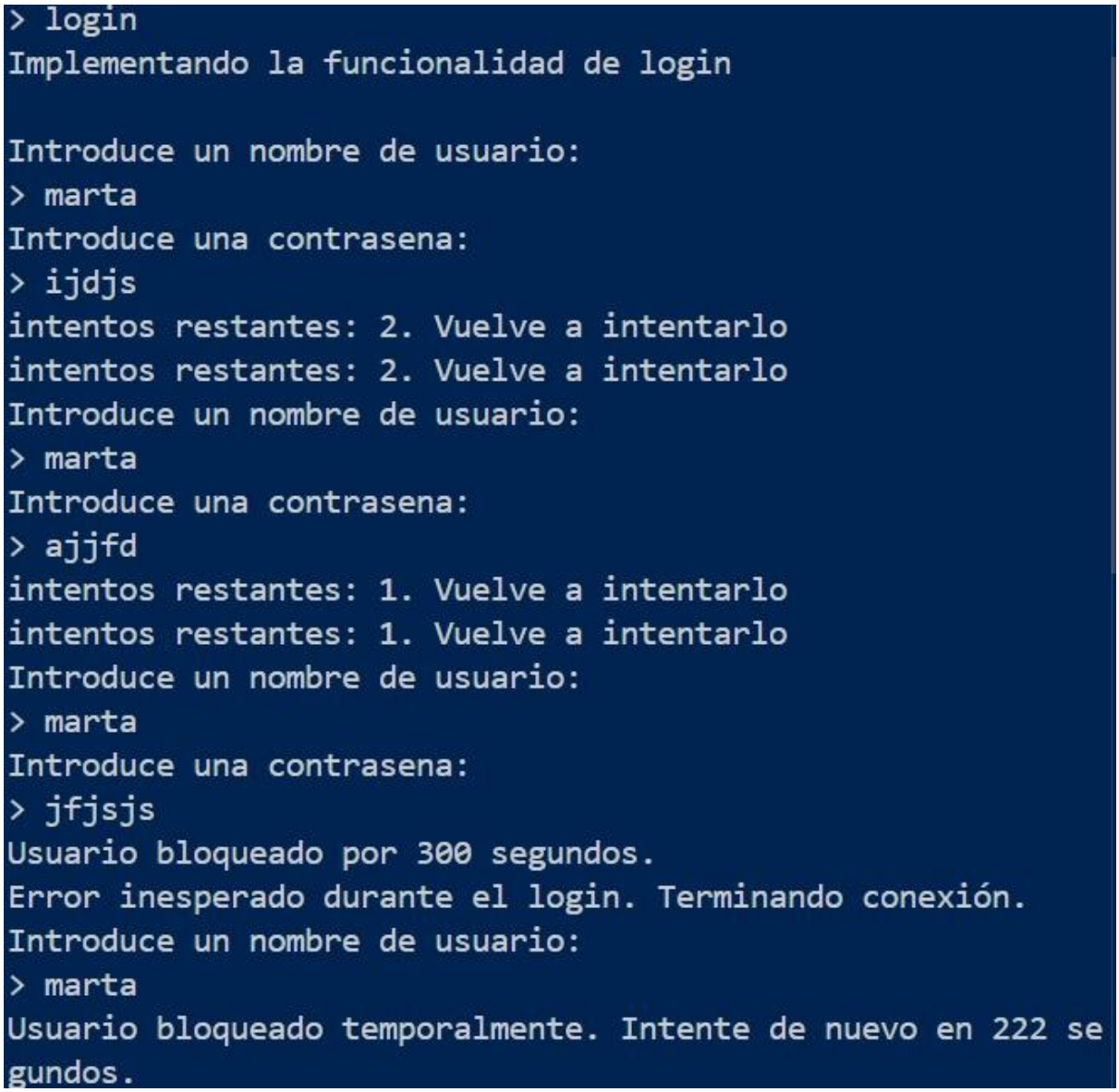
4. Comunicación cliente-servidor
    - Protección de la sesión: garantizar que las credenciales transmitidas no puedan ser interceptadas o reutilizadas.
    - Mensajes validados: todos los datos intercambiados deben estar sujetos a verificación de integridad antes de ser aceptados.

Para la comunicación cliente servidor en la aplicación del postgres usaremos el modo sslmode=reqire y ssl=true.

Pruebas

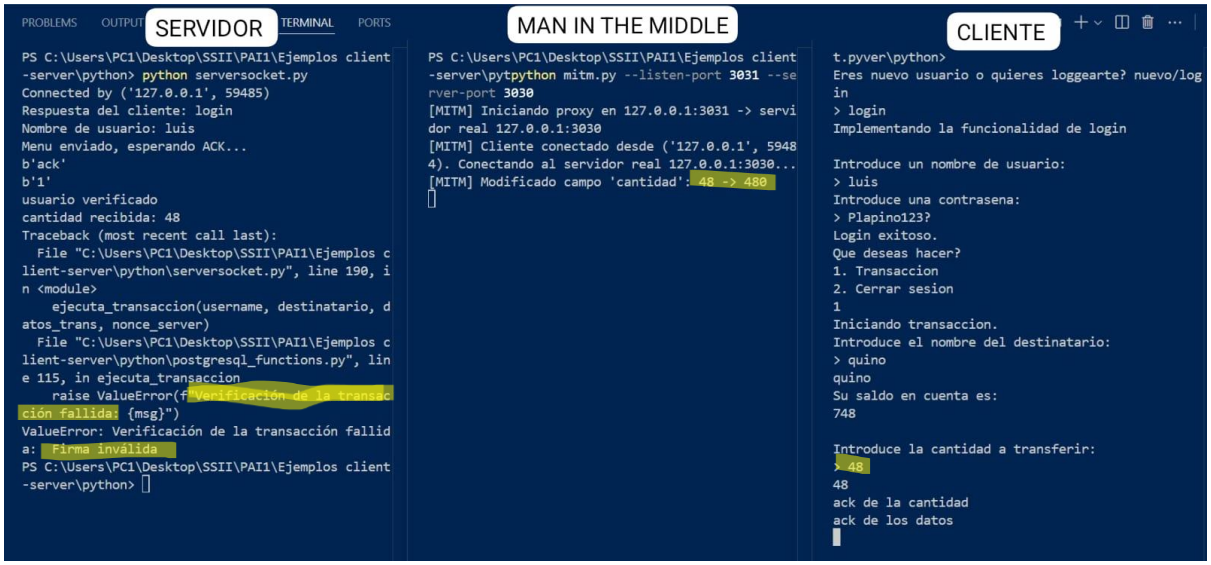
Para comprobar que nuestros requisitos de seguridad cumplen con su objetivo se harán las siguientes pruebas.

- Fuerza bruta



- Man-in-the-middle

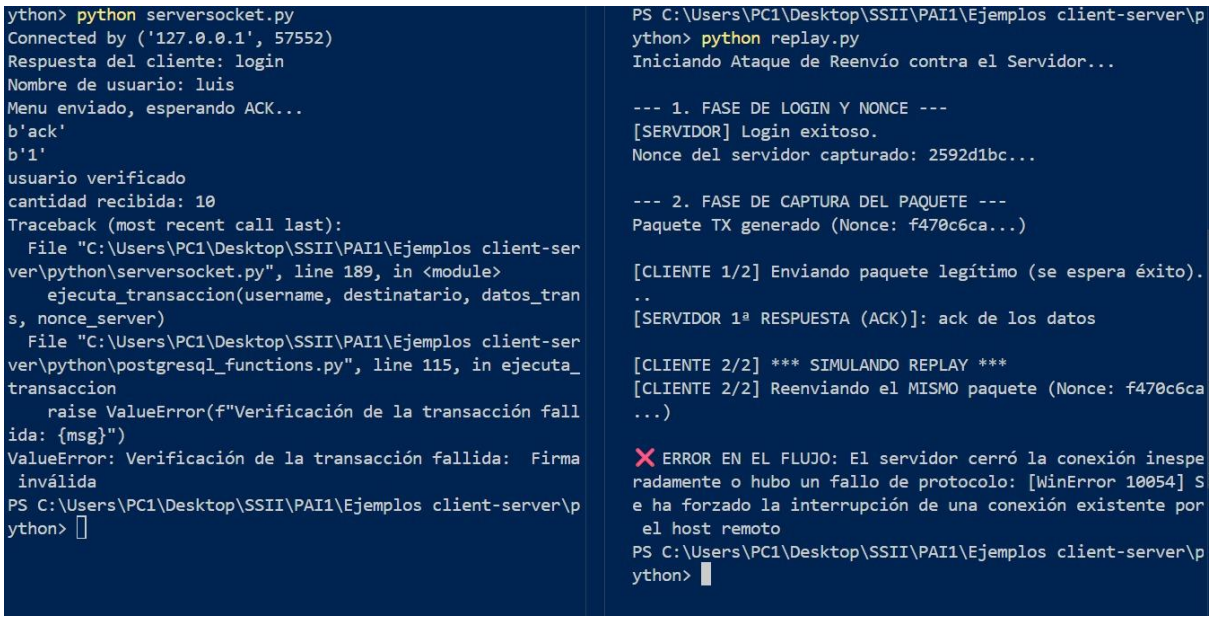
Codigo script Man in the middle -> mitm.py



Se desarrolló un script de pruebas que actúa como agente intermedio (Man-in-the-Middle) en un entorno de laboratorio autorizado. El script establece una conexión con el servidor haciéndose pasar por el cliente legítimo, intercepta los mensajes de la operación transaccional y permite la modificación del contenido del mensaje (incluido el objeto JSON, la firma HMAC y el nonce). No obstante, la rutina de verificación del servidor denominada verify_transaction, que compara el nonce, la HMAC y la marca temporal (timestamp) del mensaje, detecta la manipulación y provoca el cierre inmediato de la conexión. En consecuencia, la operación no se completa y el servidor rechaza la transacción alterada.

- Replay

Codigo script replay -> replay.py



Se intentó un ataque de reenvío (replay) en un entorno de pruebas. El agente atacante interceptó y reenvió un paquete de transacción válido; sin embargo, el servidor detectó la manipulación/reutilización y rechazó la operación, cerrando la conexión. El

rechazo se produjo porque la rutina de verificación de la transacción (verify_transaction) detectó una discrepancia en la firma (HMAC) –o en parámetros asociados como el nonce o el timestamp– y lanzó una excepción que ocasionó la terminación de la sesión.

Conclusión

El proyecto desarrollado demuestra una implementación sólida y coherente de principios de seguridad y confiabilidad en el tratamiento de la información. A nivel de persistencia, el uso de PostgreSQL como sistema gestor de base de datos garantiza la integridad, disponibilidad y confidencialidad de los datos mediante mecanismos nativos de control de acceso, autenticación segura (scram-sha-256), transacciones ACID y registro de operaciones, asegurando así una gestión confiable y trazable de la información. En cuanto a la comunicación y autenticación, el sistema incorpora medidas avanzadas de protección criptográfica, empleando NONCE, HMAC y timestamp para prevenir la alteración o repetición de mensajes y comparadores seguros que mitigan ataques por tiempo o canal lateral. Estas estrategias permiten mantener la integridad de las transacciones y la autenticidad de las entidades comunicantes.
Las pruebas de seguridad realizadas –incluyendo la simulación controlada de un ataque de tipo *Man-in-the-Middle*– confirmaron la eficacia de estos mecanismos. A pesar de la interceptación y modificación deliberada de los mensajes de transacción, la función de verificación verify_transaction detectó las alteraciones mediante la comparación de NONCE, HMAC y marca temporal, provocando el cierre inmediato de la conexión y evitando cualquier modificación no autorizada de los datos. Esto valida la robustez del sistema frente a ataques de interceptación, *replay* o manipulación en tránsito.
De forma complementaria, se establecieron políticas y recomendaciones para reforzar la protección frente a ataques comunes, control de sesiones seguras, gestión adecuada de claves y secretos y validación estricta de entradas. En conjunto, la solución implementada cumple satisfactoriamente con los requisitos de integridad de la información, persistencia confiable, acceso restringido y comunicación segura, demostrando una arquitectura capaz de resistir intentos de manipulación, suplantación o explotación de vulnerabilidades.

Bibliografía

- PostgreSQL Global Development Group. *Password authentication* — https://www.postgresql.org/docs/current/auth-password.html. PostgreSQL
- IETF. *RFC 8446 — The Transport Layer Security (TLS) Protocol Version 1.3.* IETF Datatracker
- IRTF/CFRG. *RFC 9106 — Argon2 Memory-Hard Function.* IETF Datatracker
- Jones, A., et al. *RFC 5869 — HKDF.* IETF Datatracker
- OWASP. *Web Security Testing Guide.* OWASP Foundation