

SEGURIDAD EN SISTEMAS INFORMÁTICOS Y EN INTERNET

PAI 1. INTEGRIDOS - VERIFICADORES DE INTEGRIDAD EN EL
ALMACENAMIENTO Y TRANSMISIÓN PARA ENTIDAD FINANCIERA



 Escuela Técnica Superior de
Ingeniería Informática

Ángela Lucía Morales Trujillano

Adrián Muriel Bautista

Alberto Molina Carballo

Sevilla, 24/09/2025

1.ÍNDICE

1.ÍNDICE.....	2
2.RESUMEN EJECUTIVO.....	2
3.CUERPO.....	4
3.1 Arquitectura.....	4
3.1.1 Diseño general.....	4
3.1.2 Protocolo de Comunicación.....	4
3.1.3 Flujo de operaciones.....	5
3.2 Decisiones de seguridad.....	5
3.2.1 Integridad en Almacenamiento.....	5
3.2.2 Integridad en Transmisión.....	6
3.2.3 Derivación de Claves (HKDF).....	7
3.2.4 Protección Anti-Brute-Force.....	7
3.2.5 Comparación Segura (Timing Attacks).....	8
4.PRUEBAS Y EVIDENCIAS.....	9
Script.....	9
Caso de prueba.....	9
Log Generado.....	9
5. CONCLUSIÓN.....	9

2. RESUMEN EJECUTIVO

El proyecto **PAI1 – Integridad Bancaria** tiene como objetivo desarrollar un sistema cliente-servidor que permita a los usuarios llevar a cabo operaciones bancarias básicas en un entorno simulado. A simple vista, podría parecer un programa básico de login, registro y transacciones, pero el verdadero reto ha sido conseguir usar mecanismos de seguridad parecidos a los que se emplean en sistemas financieros reales.

Durante el desarrollo, identificamos algunos de los problemas más comunes en este tipo de sistemas. Uno de los más evidentes es el de las **contraseñas**: si se almacenan en texto claro, cualquier intruso que tenga acceso a la base de datos podría sustraer todas las cuentas en cuestión de segundos. Para prevenir esto, decidimos implementar **Argon2**, que no es un desarrollo propio, sino el algoritmo que triunfó en la Password Hashing Competition en 2015 y que actualmente se recomienda incluso en entornos empresariales. Gracias a Argon2 y a la salt aleatoria generada para cada usuario, incluso si alguien accede a la base de datos, resultaría prácticamente imposible recuperar las contraseñas.

Otro desafío es la **integridad de los mensajes durante su transmisión**. Entre el cliente y el servidor podría haber un atacante interceptando la comunicación, y en teoría, podría modificarla antes de que llegue a su destino (el típico ataque Man-in-the-Middle). Para protegernos contra esto, utilizamos **HMAC con SHA-256**, que es un estándar establecido en el RFC 2104 y también se aplica en protocolos tan cruciales como TLS. De este modo, garantizamos que cualquier mensaje alterado sea rechazado automáticamente por el receptor, debido a que la firma no coincidiría.

Asimismo, relacionado con lo anterior, se encuentra el problema de los **ataques de repetición** (replay attacks). Si un atacante captura un mensaje válido, como por ejemplo, una orden de transferencia, podría reenviarlo varias veces con la intención de ejecutar la misma operación. Para acabar con este riesgo, añadimos a cada mensaje un **nonce**, un **timestamp** y un **número de secuencia**. Esta combinación asegura que cada solicitud sea única y no pueda ser reutilizada. No es una idea original nuestra; de hecho, sigue las recomendaciones del NIST en sus guías de criptografía sobre cómo resguardar los protocolos de comunicación.

Finalmente, pensamos en los **ataques de fuerza bruta**. Aunque contemos con Argon2, un atacante podría probar millones de contraseñas contra el servidor hasta acertar con alguna. La solución que implementamos fue un sistema de bloqueo: si un usuario introduce incorrectamente su contraseña cinco veces consecutivas, la cuenta queda bloqueada durante 120 segundos. Igualmente, esta idea no es una invención propia, sino que se basa en las pautas

proporcionadas por **OWASP**, una comunidad internacional que establece buenas prácticas de seguridad en aplicaciones.

El resultado de todo esto es un sistema bastante sólido para ser un proyecto académico. El cliente permite registrarse, loguearse, realizar transacciones y cerrar sesión, mientras que el servidor verifica que cada paso se lleve a cabo de manera segura. Además, todas estas funcionalidades han sido evaluadas en diversos escenarios: tanto en situaciones de uso normal como en intentos de ataque (repetición, manipulación de mensajes, fuerza bruta). Para cada prueba se han almacenado **ficheros de log** que sirven como evidencia de que las medidas de seguridad efectivamente funcionan.

En conclusión, consideramos que el proyecto no solo satisface los requisitos del PAI1, sino que refleja cómo se abordan los problemas de seguridad en la práctica. Cada técnica que empleamos —desde Argon2 hasta los HMAC, pasando por nonces y bloqueos— está respaldada por estándares y por las mismas recomendaciones que siguen sistemas reales. Esto nos proporciona la confianza de que, aunque se trate de un entorno académico, lo que hemos desarrollado es un sistema que podría establecer las bases de una aplicación bancaria auténtica.

3.CUERPO

3.1 Arquitectura

3.1.1 Diseño general

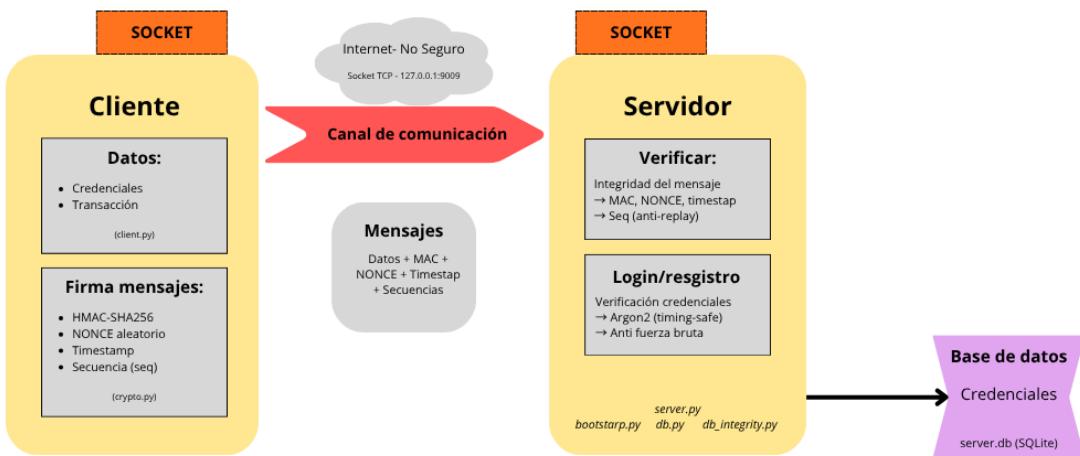
El sistema sigue una arquitectura cliente-servidor clásica con comunicación por sockets TCP en localhost: 9009.

La comunicación se estructura mediante JSON para facilitar el parsing y la depuración.

Componentes principales:

- **Cliente** (client/client.py): Genera peticiones firmadas con MAC
- **Servidor** (server/server.py): Valida la integridad y ejecuta operaciones
- **Módulo criptográfico** (common/crypto.py): Funciones de seguridad compartidas
- **Base de datos** (server.db): SQLite con persistencia de usuarios y transacciones

En la siguiente imagen observamos un esquema visual de nuestra arquitectura:



3.1.2 Protocolo de Comunicación

Los mensajes entre cliente y servidor siguen este formato JSON:

- Mensajes sin protección (registro/login inicial):

```
{
  "type": "register" | "login",
  "user": "nombre_usuario",
  "password": "contraseña"
}
```

- Mensajes protegidos (transacciones/logout):

```
{
  "type": "transfer" | "logout",
  "user": "nombre_usuario",
  "payload": [...],
  "ts": timestamp_unix,
```

```
"seq": numero_secuencia,  
"nonce": valor_aleatorio,  
"mac": firma_hmac  
}
```

La firma MAC se calcula sobre todos los campos EXCEPTO el propio campo "mac", usando JSON canónico (ordenado y sin espacios) para garantizar consistencia.

3.1.3 Flujo de operaciones

1. REGISTRO:

Cliente -> {"type":"register", "user":"X", "password":"Y"}

Servidor -> Hashea con Argon2, guarda en BD, responde OK/ERROR

2. LOGIN:

Cliente -> {"type":"login", "user":"X", "password":"Y"}

Servidor -> Verifica Argon2, reinicia contador de secuencia, responde OK/ERROR

3. TRANSACCIÓN (requiere login previo):

Cliente -> Genera mensaje con seq++, nonce único, calcula MAC

Servidor -> Valida MAC, nonce no repetido, seq > último, guarda transacción

4. LOGOUT:

Cliente -> Mensaje firmado tipo "logout"

Servidor -> Limpia sesión, responde OK

3.2 Decisiones de seguridad

3.2.1 Integridad en Almacenamiento

Requisito: "Preservar la integridad de credenciales almacenadas"

Decisión: Argon2 (librería argon2-cffi)

¿Por qué Argon2?

- Es el ganador de la Password Hashing Competition (2015)
- Resistente a ataques GPU/ASIC por su alto consumo de memoria
- Incorpora salt automáticamente (único por usuario)
- Key stretching nativo: hace inviable fuerza bruta

Alternativas descartadas:

- SHA-256 simple: vulnerable a tablas rainbow
- bcrypt: menor resistencia a hardware especializado que Argon2
- PBKDF2: más lento en software, menos robusto que Argon2

Implementación:

```
ph = PasswordHasher() # Parámetros por defecto (time_cost=2, memory_cost=102400)
```

```
hash = ph.hash(password) # Genera: $argon2id$v=19$m=102400,t=2,p=8$salt$hash
```

La verificación usa ph.verify(hash, password) que internamente hace comparación resistente a timing attacks.

3.2.2 Integridad en Transmisión

Requisito: "Conservar integridad en comunicaciones no seguras"

Decisión: HMAC-SHA256 + nonce + timestamp + secuencia

¿Por qué HMAC-SHA256?

- Estándar RFC 2104, ampliamente auditado
- Resistente a ataques de extensión (a diferencia de SHA-256(key||msg))
- Computacionalmente eficiente
- Soportado nativamente en Python (librería hmac)

Capa 1 - MAC: Detecta modificaciones

Capa 2 - Nonce: Evita replay de mensajes idénticos (añadimos función *cleanup_old_nonces* para no gastar excesiva memoria, eliminando así los nonces cada 60 segundos)

Capa 3 - Timestamp: Ventana de validez (± 30 segundos)

Capa 4 - Secuencia: Evita replay con nonce diferente pero mismo seq

Ejemplo de ataque mitigado:

Un atacante captura un mensaje válido y lo reenvía:

- Si lo reenvía exactamente igual → rechazado por nonce repetido
- Si cambia el nonce → rechazado por MAC inválido
- Si recalcula MAC con nuevo nonce → rechazado por seq antigua
- Si pasa mucho tiempo → rechazado por timestamp expirado

3.2.3 Derivación de Claves (HKDF)

Problema: Usar la misma clave maestra para MAC y almacenamiento es mala práctica

Solución: HKDF-SHA256 (RFC 5869)

K_master = b"demo-psk-change-me"

K_mac = hkdf_sha256(K_master, b"mac")

K_storage = hkdf_sha256(K_master, b"storage") # Si se necesitara

¿Por qué HKDF?

- Deriva claves criptográficamente independientes de una clave maestra
- Garantiza que comprometer K_mac no compromete K_storage
- Estándar NIST SP 800-108

Alternativas descartadas:

1. Diffie-Hellman / ECDH: Necesitaría certificados X.509, lo cual excede el alcance del PAI
2. TLS/SSL nativo (Python ‘ssl’ module): Oculta los detalles de implementación

Implementación simplificada (suficiente para la práctica):

```
def hkdf_sha256(ikm: bytes, info: bytes, length=32) -> bytes:
```

```
prk = hmac.new(b"\x00"*32, ikm, sha256).digest()
```

```
t = hmac.new(prk, info + b"\x01", sha256).digest()
```

```
return t[:length]
```

3.2.4 Protección Anti-Brute-Force

Requisito implícito: Proteger login contra ataques de fuerza bruta

Decisión: Límite de intentos por usuario con backoff temporal

Parámetros:

- WINDOW = 60 segundos
- MAX_FAILS = 5 intentos
- BACKOFF = 120 segundos de bloqueo

Funcionamiento:

1. Tras 5 intentos fallidos en 60 segundos → usuario bloqueado 120 segundos
2. Durante el bloqueo, incluso credenciales correctas son rechazadas
3. Tras el bloqueo, contador se reinicia

Implementación: Tabla login_attempts en SQLite(username, last_fail_ts, fails)

Alternativas más avanzadas no implementadas (fuera del alcance):

- CAPTCHA tras 3 intentos
- Bloqueo IP además de usuario
- Rate limiting global del servidor

3.2.5 Comparación Segura (Timing Attacks)

Problema: Comparaciones byte-a-byte normales (==) revelan información temporal

Ejemplo vulnerable:

```
if computed_mac == received_mac: # ¡MAL!  
  
    return True
```

Un atacante mide tiempos de respuesta:

- MAC correcta hasta byte 10 → tarda X ms
- MAC correcta hasta byte 15 → tarda Y ms ($Y > X$) → Puede derivar el MAC carácter a carácter

Solución: hmac.compare_digest()

Compara TODOS los bytes en tiempo constante, independiente de dónde difieren.

```
return hmac.compare_digest(computed_mac, received_mac) # ✓ BIEN
```

Esta función está en la librería estándar de Python (hmac) y cumple con estándares de seguridad para comparaciones criptográficas.

3.2.6 Integridad de la Base de Datos

Requisito adicional: Garantizar que la base de datos no sea modificada externamente

Decisión: Verificador de integridad con HMAC-SHA256 sobre SHA-256 del archivo

¿Por qué este enfoque?

- Detecta modificaciones offline a la base de datos
- Usa clave derivada independiente (K_DB_INTEGRITY) mediante HKDF
- Verificación automática al iniciar el servidor
- Registro automático al cerrar (atexit)

Implementación: server/db_integrity.py

- compute_db_hash(): Calcula SHA-256 del archivo completo
- compute_db_mac(): HMAC sobre el hash
- verify_db_integrity(): Comparación segura con compare_digest()
- save_integrity_record(): Guarda {timestamp, hash, mac, tamaño}

Funcionamiento:

1. Al cerrar servidor → Guarda registro en server.db.integrity
2. Al iniciar servidor → Recalcula y compara con registro previo
3. Si no coincide → Alerta de seguridad y opción de detener

Limitación: Solo detecta modificaciones cuando el servidor está apagado (protección en caliente requeriría WAL journaling, fuera del alcance).

4.PRUEBAS Y EVIDENCIAS

Se han desarrollado scripts de prueba automatizados que generan logs de evidencia documentados en la carpeta logs/.

Tabla resumen:

Script	Caso de prueba	Log Generado
test_04_messages.py test_05_integrity.py test_05_preexistentes test_06_blocking.py test_07_logout.py test_07.persistencia	Registro/login básico Ataques MITM/replay/seq Usuarios semilla Anti-brute-force Cierre de sesión Verificación BD	04_messages.txt 05_integrity.txt 05_preexistentes 06_blocking.txt 07_logout.txt 07.persistencia.txt

Estas evidencias demuestran empíricamente que las medidas de seguridad funcionan.

5. CONCLUSIÓN

Se ha implementado exitosamente un sistema cliente-servidor que cumple todos los requisitos funcionales y de seguridad del enunciado. Las decisiones técnicas tomadas (Argon2, HMAC-SHA256, HKDF) están alineadas con estándares actuales de la industria.

Logros principales:

- Resistencia demostrada contra 4 vectores de ataque principales
- Persistencia con SQLite funcional y testeada
- Usuarios preexistentes correctamente inicializados
- Evidencias exhaustivas generadas automáticamente

Limitaciones y trabajo futuro:

- Clave maestra hardcodeada (en producción: Diffie-Hellman o certificados)
- Sin cifrado de payload (solo integridad, no confidencialidad)
- Anti-brute-force básico (podría mejorarse con CAPTCHA o rate limiting IP)

El proyecto cumple su objetivo educativo de implementar verificadores de integridad en un contexto realista de entidad financiera.

6. REFERENCIAS

RFC 2104 - HMAC: Keyed-Hashing for Message Authentication

RFC 5869 - HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

Argon2: Password Hashing Competition Winner (2015)

NIST FIPS 180-4 - Secure Hash Standard (SHA-256)

OWASP - Password Storage Cheat Sheet [6] Python hmac library documentation

Claude - Consulta y corrección de código

Chatgpt - Asistencia diseño de la arquitectura