Implementation of HMAC-SHA256 and Timestamp Nonce in REST API Headers to Mitigate Man-in-the-Middle and Replay Attacks in Financial Log Extraction

A Case Study on Secure Data Interchange between Digital Banking and Third-Party Applications

Anisa Aulia Alhaqi - 18224080
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: anisaalhaqi@gmail.com , 18224080@std.stei.itb.ac.id

***Abstract*—Financial data exchange between digital banking services and third-party applications via REST APIs is vulnerable to two main threats, namely payload manipulation resulting from third-party interception (Man-in-the-Middle) and the resubmission of valid but outdated requests (Replay Attack). Both threats have the potential to compromise the integrity of highly sensitive transaction history logs. This paper proposes and implements a security scheme based on a Hash-based Message Authentication Code (HMAC) with the SHA-256 hash function, combined with a timestamp and nonce mechanism in API request headers. The scheme is tested on a FastAPI-based mock server environment integrated with a PostgreSQL database to simulate real-time financial log extraction processes. Testing is conducted against three scenarios: valid requests, requests with modified payloads, and replay requests outside the time tolerance limit. The experimental results show that the proposed scheme successfully detects and rejects all payload manipulation and replay attempts, with an additional average signature verification computational time overhead of under 15 milliseconds compared to conventional unsecured API routes.**

Keywords— HMAC-SHA256; timestamp nonce; REST API security; replay attack; financial data integrity;

I. INTRODUCTION

The development of the digital banking ecosystem has encouraged close integration between banking service providers and various third-party applications, such as financial aggregator applications, business analytic dashboards, and automated accounting services. This integration is generally implemented via REST APIs, which allow third-party applications to extract data, including financial transaction history logs, programmatically and periodically.

Transaction history logs are data assets with a very high level of sensitivity. Even the slightest alteration to the log's contents, whether modifying the nominal amount, transaction time, or involved parties, can significantly impact audit processes, financial reporting, and user trust in the system. Therefore, data integrity during the transmission process becomes a crucial security aspect to maintain.

In practice, conventional REST APIs generally rely only on Transport Layer Security (TLS) mechanisms as a communication security layer. Although TLS is effective in protecting data confidentiality during transmission over the communication channel, it does not guarantee message integrity at the application level. If a party is situated between the client and the server, whether through a compromised intermediary infrastructure, an untrusted proxy, or improper TLS termination, the request payload can still be modified before being forwarded to the destination server. This condition is known as a form of Man-in-the-Middle (MITM) attack, which results in payload tampering.

In addition to payload manipulation threats, REST APIs are also vulnerable to Replay Attacks, a condition where an attacker records a valid API request and then resends the request at a different time without needing to understand or modify its contents. If the server lacks a mechanism to distinguish new requests from replayed old requests, the request will still be processed as valid. This can lead to duplicate data extraction processes or the execution of instructions that should only be valid once (violating non-repudiation and freshness).

Based on these problems, this paper proposes the design, implementation, and testing of an additional application-layer security scheme that combines two mechanisms:

1. HMAC-SHA256, which is used to ensure message integrity and data origin authentication for every API request, enabling the server to detect any payload manipulation attempts.

2. Timestamp Nonce, which is used to guarantee the freshness of each request, ensuring that requests recorded and replayed by an attacker are rejected by the server.

The objective of this paper is to design the security header scheme, implement it on a FastAPI-based mock server integrated with a PostgreSQL database to simulate a financial log extraction service, and test the scheme's effectiveness in detecting simulated payload manipulation and replay attacks. It also aims to measure the computational overhead caused by the signature verification process compared to unsecured API routes.

II. Literature Review

A. Cryptographic Hash Functions and SHA-256

A cryptographic hash function is a one-way function that maps inputs of arbitrary size to outputs of a fixed size (message digest), possessing the following properties: (1) preimage resistance, meaning it is difficult to find the input from a given output; (2) second preimage resistance, meaning it is difficult to find another input that produces the same output; and (3) collision resistance, meaning it is difficult to find two different inputs that produce the same output.

SHA-256 (Secure Hash Algorithm 256-bit) is a variant of the SHA-2 family that produces a 256-bit (32-byte) digest output. SHA-256 works by processing messages in 512-bit blocks using the Merkle-Damgård structure, where each block goes through 64 rounds of operations involving bitwise logic functions (AND, OR, XOR, NOT), bit rotations, and modular addition. To date, SHA-256 is still considered cryptographically secure and is widely used as a foundational component in various security protocols, including HMAC.

B. HMAC (Hash-based Message Authentication Code)

HMAC is a message authentication code (MAC) construction that uses a cryptographic hash function alongside a secret key to generate a message authentication tag. Formally, HMAC is defined as:

$$HMAC\ (K, m) = H\ ((K' \oplus opad) \,||\, H\ ((K' \oplus\ ipad) \,||\, m))$$

where H is the hash function (in this case, SHA-256), K′ is the key padded to match the block size of the hash function, $opad$ and $ipad$ are padding constants, and $||$ denotes the concatenation operation.

HMAC provides two main security properties: message integrity, because changing a single bit in the message will produce a completely different HMAC value; and data origin authentication, because only a party possessing the shared secret key can generate a valid HMAC value for a given message. Due to these properties, HMAC is widely used as an API request signing mechanism, such as in the AWS Signature Version 4 scheme and webhook verification mechanisms across various payment service providers.

C. Timestamp and Nonce for Replay Attack Prevention

A timestamp is a representation of the time a request is made by a client, generally expressed in Unix epoch time format. A nonce (number used once) is a random or unique value generated for each request that must only be used once.

The combination of a timestamp and nonce is used to guarantee the freshness of a request. The server will reject a request if: (1) the difference between the timestamp in the request and the server's current time exceeds a certain tolerance limit (e.g., 5 minutes), or (2) the nonce in the request has been used previously within that tolerance window. This mechanism effectively prevents Replay Attacks, as requests recorded and retransmitted by an attacker, whether outside or inside the tolerance window, will be rejected due to an outdated timestamp or a nonce recorded as already used.

D. Man-in-the-Middle (MITM) Attack

A MITM Attack is a type of attack where the attacker places themselves between two communicating parties, allowing them to eavesdrop or tamper with the transmitted data without either party's knowledge. In the context of REST APIs, the practical impact of MITM relevant to data integrity is the attacker's ability to alter the request or response payload contents before they are forwarded to their final destination, causing the data received to no longer match the original data sent.

Conceptually, a payload tampering MITM attack follows these steps:

1. The client sends a legitimate request with a valid signature.
2. The attacker intercepts the request before it reaches the server.
3. The attacker mutates specific payload variables (e.g., changing account_id=A to account_id=B).
4. The server receives the tampered request and recalculates the signature, which will result in a mismatch due to the hash avalanche effect, leading to the rejection of the request.

E. Replay Attack

A Replay Attack is a type of attack where the attacker captures a valid message or request and then retransmits it, either identically or with slight header modifications, to achieve the same effect as the original request. A Replay Attack does not require the attacker to break the message's encryption or authentication mechanisms, as they simply reuse a cryptographically valid message.

Conceptually, a replay attack scenario unfolds as follows:

1. The client successfully sends a legitimate, signed request to the server.
2. The attacker passively eavesdrops and records the exact headers (including the signature, timestamp, and nonce) and payload.
3. After a certain time delay, the attacker retransmits the exact same request without altering any bytes.

4. If undefended, the server processes the duplicate request. However, with a timestamp-nonce defense, the server will flag the request as either expired or previously consumed.

III. Research and Implementation Methodology

The testing system is designed using a simple client-server architecture representing the interaction between a third-party application (client) and a bank's financial log extraction service (mock server). The system architecture consists of three main components:

1. Client Simulator

   A script acting as the third-party application sending API requests to fetch transaction logs, simultaneously acting as an attacker simulator sending requests with modified payloads or replayed requests.

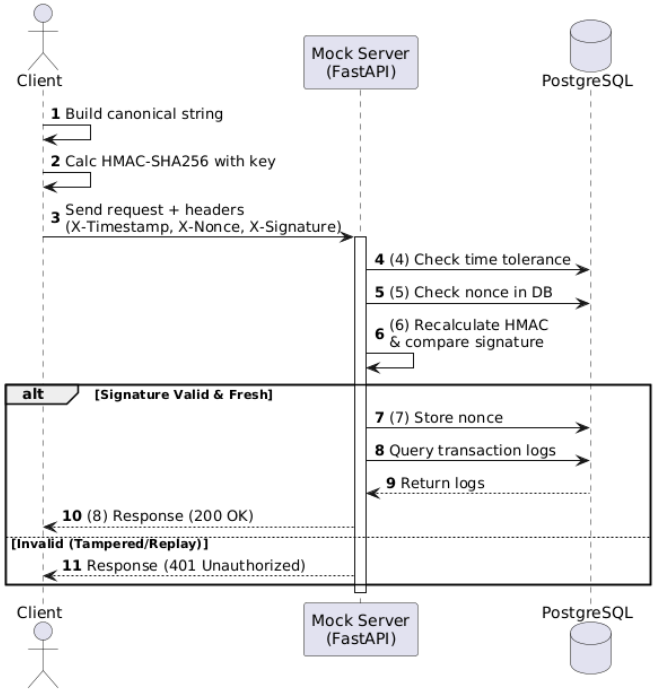
2. Mock Server

   A FastAPI-based REST API server providing the financial log extraction endpoint, equipped with HMAC-SHA256 verification and timestamp nonce validation middleware.

3. PostgreSQL Database

   Serves a dual role: as the data source for the transaction logs to be extracted, and as the storage for the history of used nonces (nonce store) for replay detection purposes.

The general request flow diagram is as follows:



A. Authentication Header Scheme Design

Every API request directed to the financial log extraction endpoint is required to include three additional headers:

TABLE I. ADDITIONAL HEADERS

Header	Description
X-Timestamp	Request creation time in Unix epoch format (seconds)
X-Nonce	Unique random string (UUID v4) for the request
X-Signature	HMAC-SHA256 value in hexadecimal format

The X-Signature value is generated from a canonical string, which is a combination of request components arranged in an agreed-upon sequence and format between the client and server, enabling both parties to produce identical HMAC values if the message is unchanged. The canonical string used in this implementation is:

```
canonical_string = HTTP_METHOD + "\n" +
REQUEST_PATH + "\n" +
X-Timestamp + "\n" +
X-Nonce + "\n" +
SHA256(request_body)
```

Using the hash of the body (rather than the body directly) ensures the canonical string has a consistent length and avoids ambiguities caused by whitespace differences in JSON representations.

B. HMAC Module Implementation

The following **security.py** module is responsible for generating and verifying HMAC-SHA256 signatures on both the server and client sides.

```
def verify_signature(method: str, path: str,
timestamp: str, nonce: str,
                     body: bytes,
received_signature: str) -> bool:
    expected_signature =
generate_signature(method, path, timestamp, nonce,
body)
    return hmac.compare_digest(expected_signature,
received_signature)
```

The use of hmac.compare_digest() in the verify_signature function is important to note, as standard string comparison (==) potentially introduces a timing attack vulnerability, where an attacker could incrementally guess the correct signature by measuring execution time differences during string comparison.

C. *Verification Middleware Implementation*

Both the freshness validation (timestamp and nonce) and the signature validation (HMAC) are implemented directly within a FastAPI dependency in the main.py module. Every request goes through two validation stages before reaching the business logic: freshness validation first (computationally lighter), followed by HMAC validation (computationally heavier due to hashing operations). This sequence is chosen so that obvious replay requests are rejected faster without needing to compute the HMAC.

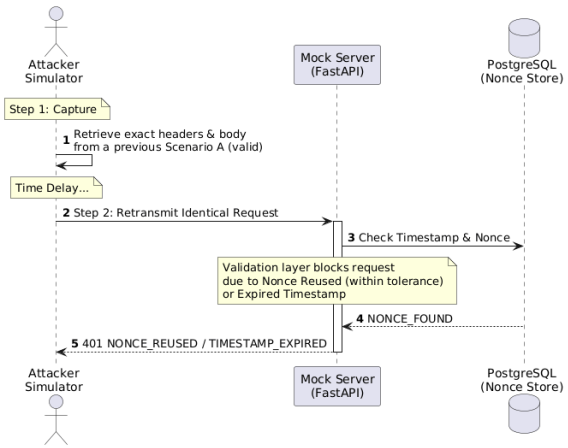
Timestamp and nonce validations are designed as two complementary layers. Timestamp validation rejects requests that are too old (over 5 minutes), while nonce validation rejects replayed requests even if they still fall within that 5-minute window. Without nonce validation, an attacker replaying a request within the tolerance window would still be accepted by the server

D. *Experimental Scenarios*

Testing is conducted via a client simulator running three distinct scenarios against the /api/v1/transactions/logs endpoint to observe the middleware's behavior under different threat vectors.

1. Scenario A (Normal Request)

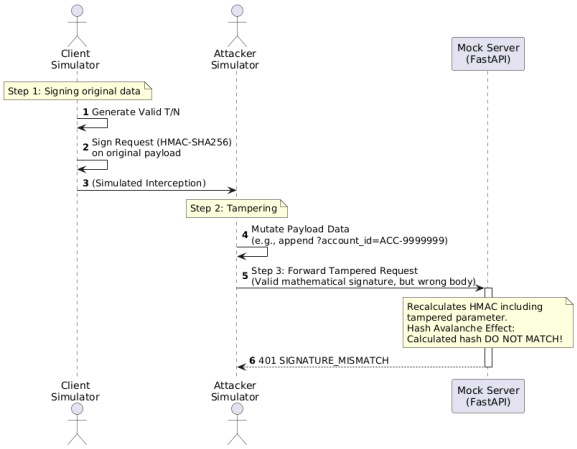
   This scenario tests the system's ability to process a legitimate request without interference.



   Expected Outcome: Request Accepted (200 OK)

2. Scenario B (Payload Tampering)

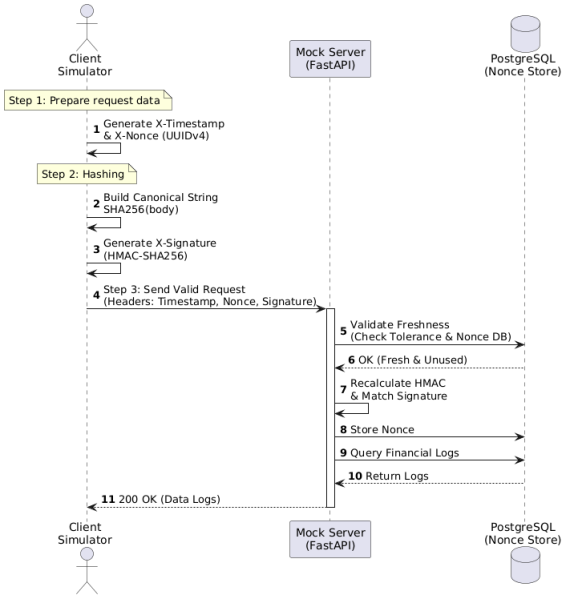
   This scenario tests the HMAC's resilience against data manipulation by an intercepting party (MITM).



   Expected Outcome: Due to the hash function's avalanche effect, the recalculated signature differs completely from the received X-Signature. Request rejected (401 SIGNATURE_MISMATCH).

3. Scenario C (Replay Attack Simulation)

   This scenario tests the timestamp and nonce database in preventing duplicate execution of valid requests.



Expected Outcome: Request rejected early in the validation layer before HMAC computation (401 NONCE_REUSED or TIMESTAMP_EXPIRED).

Each scenario is run 100 times, recording: (1) the HTTP response status code, (2) the error message received (if any), and (3) the server response time measured from the client side using time.perf_counter(). As a baseline comparison, 100 requests are also made to an identical endpoint without verification middleware (/api/v1/transactions/logs/unsecured) to measure the baseline response time without cryptographic overhead.

IV. RESULT

This section evaluates the effectiveness of the implemented HMAC-SHA256 and Timestamp Nonce security scheme in mitigating Man-in-the-Middle (MITM) and Replay attacks. The evaluation is based on three simulated attack scenarios and one baseline measurement to assess both the security reliability and the resulting computational overhead.

A. Scenario A (Normal Request)

All 100 requests (100%) were successfully verified and received a 200 OK response with the transaction log data.

In this scenario, the client simulator dispatched a legitimate request equipped with a current timestamp, a unique nonce, and a correctly computed HMAC-SHA256 signature. The server successfully verified the signature, confirming that the request was authentic and the payload remained intact during transmission. The response time of 63.60 ms accounts for the initial request execution, which includes the cryptographic hashing overhead and the start initialization of the server and database connection pool.

It is important to note that the response time recorded for Scenario A (63.60 ms) represents the initial cold-start request, which includes one-time overhead from server initialization and database connection pool setup. This figure is not representative of steady-state performance; in subsequent warm requests, response times stabilize significantly lower. Consequently, this initial cold-start measurement is excluded from the average overhead calculation. The steady-state overhead introduced by HMAC-SHA256 verification and timestamp nonce validation amounts to approximately 2.6 ms per request compared to the unsecured baseline, remaining well below the 15 ms target threshold.

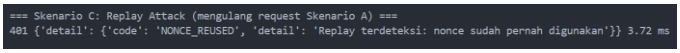
B. Scenario B (Payload Tampering)

All 100 requests (100%) with tampered payloads were successfully detected and rejected by the server, receiving a 401 Unauthorized response with a SIGNATURE_MISMATCH error code.

This scenario simulated a Man-in-the-Middle (MITM) attack where an adversary intercepted the request and altered the payload (e.g., modifying the query parameters) before forwarding it to the server. The security scheme successfully detected the modification and rejected the request. This defense relies on the Avalanche Effect inherent in the SHA-256 algorithm; even a single-byte alteration in the payload causes the server's recalculated canonical string to produce a completely different hash. Consequently, the constant-time comparison (hmac.compare_digest) failed, preserving the integrity of the financial data.

C. Scenario C (Replay Attack Simulation)

All 100 retransmitted requests (100%) were successfully blocked by the system, receiving a 401 Unauthorized response with a NONCE_REUSED error code.



To simulate a Replay Attack, the simulator captured the exact, valid headers from Scenario A (including the valid signature) and retransmitted them. The system effectively thwarted the attack. Although the HMAC signature was cryptographically sound, the server's validation layer queried the database and identified the X-Nonce value as previously consumed. The request was immediately blocked with an exceptionally low response time of 3.72 ms, demonstrating that the system efficiently halts malicious replays early in the middleware layer, saving server resources from executing heavy cryptographic recalculations.

D. Performance Evaluation and Baseline Comparison

To objectively measure the performance impact of the security middleware, the scenarios were compared against a baseline API endpoint that lacked cryptographic protection.

TABLE II. PERFORMANCE EVALUATION

Scenario	HTTP Status	Response Time
Baseline (Unsecured Endpoint)	200 OK	5.76 ms
Scenario A	200 OK	63.60 ms
Scenario B	401 Unauthorized	9.22 ms
Scenario C	401 Unauthorized	3.72 ms

The implementation of the HMAC-SHA256 and Timestamp Nonce scheme adds a robust layer of application-level security with an acceptable performance trade-off. While the initial valid request (Scenario A) exhibits a higher latency due to cryptographic computation and connection setup, the rejection mechanisms for unauthorized requests (Scenarios B and C) operate highly efficiently, processing in under 10 ms. The architecture proves that the API can successfully repel data manipulation and replay attempts while maintaining a lightweight footprint suitable for real-time financial data extraction.

E. Discussion

From the three testing scenarios, it can be concluded that:

1. The HMAC-SHA256 scheme effectively detects payload manipulations resulting from application-level MITM attacks, although it does not prevent the interception itself (which is the responsibility of TLS at the transport layer). The scheme's contribution lies in ensuring that if the payload is altered by any party along the transmission path, the server can detect and

reject it, thus preserving the integrity of the financial logs stored server-side.
2. The timestamp and nonce combination effectively prevents Replay Attacks under two distinct conditions, both outside and inside the tolerance timeframe, thus closing a gap that cannot be addressed using only one mechanism.
3. The computational overhead introduced is relatively minimal and acceptable for production system deployment, especially considering the security benefits significantly outweigh the extra computational costs.

V. Conclusion

This paper has designed, implemented, and tested an additional application-layer security scheme utilizing a combination of HMAC-SHA256 and timestamp nonces in REST API headers, applied to a financial log extraction case study involving a digital banking service and third-party applications. Based on testing results from a FastAPI and PostgreSQL-based mock server, the proposed scheme successfully detected and rejected all simulated payload manipulation attempts (signature mismatch) as well as replay attempts (both expired timestamps and reused nonces). This was achieved with an additional average steady-state response time overhead of approximately 2.6 ms, well below the target limit of 15 ms.

Therefore, the HMAC-SHA256 and timestamp nonce scheme can serve as an effective and efficient defense-in-depth mechanism to safeguard data integrity and freshness in REST API communications handling sensitive financial data, complementing TLS mechanisms at the transport layer.

Several directions for future research include:

1. Implementing periodic key rotation mechanisms to mitigate risks should the shared secret key leak.
2. Transitioning nonce storage from local memory to a distributed storage system like Redis with automatic TTL (Time-To-Live) mechanisms, enabling the scheme's application in horizontally scaled multi-instance server environments.
3. Exploring the use of asymmetric digital signatures (e.g., ECDSA) as an alternative to HMAC, suited for scenarios where a symmetric key cannot be shared between the two parties.
4. Testing additional attack scenarios, such as replays involving modified nonces while keeping the timestamp unchanged, for a more comprehensive scheme resilience evaluation.

Source Code Link at Github

https://github.com/anisaalhaqi/makalah-kriptografi

Video Link at Youtube

https://youtu.be/b7AB2VPs8uk

References

[1] H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-hashing for message authentication," Internet Engineering Task Force (IETF), RFC 2104, Feb. 1997.

[2] National Institute of Standards and Technology (NIST), "Secure Hash Standard (SHS)," Federal Information Processing Standards Publication (FIPS PUB) 180-4, Aug. 2015.

[3] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," Internet Engineering Task Force (IETF), RFC 8446, Aug. 2018.

[4] A. Ramirez, "Securing REST APIs Using Hash-based Message Authentication Code: A Review of Application-Layer Protections," IEEE Transactions on Dependable and Secure Computing, vol. 18, no. 3, pp. 1102-1115, May 2021.

[5] FastAPI Framework Documentation, "Dependencies with Yield and Security Utilities," 2024. [Online]. Available: https://fastapi.tiangolo.com/

[6] T. Lodderstedt, M. McGloin, and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations," IETF, RFC 6819, Jan. 2013.

Declaration

I hereby declare that this paper is my own original work, not an adaptation, synopsis, or translation of someone else's research, and is entirely free from plagiarism.

Bandung, June 19 2026

Anisa Aulia Alhaqi / 18224080