

SECURITY NOTES / 01

# A Valid Signature Is Not Replay Protection

Four decisions between a signed webhook and a safe business action.

## THE QUESTION

What does an authentic request actually prove?

## THE EXPERIMENT

Follow one event through verification, replay, and a local transaction.

## THE BOUNDARY

A local commit does not make a remote side effect exactly-once.

Amy V. | October 2026

Independent portfolio work. Not commissioned by, affiliated with, or endorsed by Cracken. The examples describe a local reference implementation.

A request arrives announcing that an invoice has been paid. Its signature is correct. The application records the event and performs an action. A moment later, the exact same request arrives again.

The signature is still correct.

Nothing about the message changed, so nothing about its HMAC should change. The failure is in asking authentication to answer a question about history: has this application already acted on this event?

Webhook Trust Lab makes that distinction visible in a small Python reference implementation. It uses fictional invoice events, deterministic timestamps, and a local SQLite database. No provider is contacted and no money moves. The purpose is to connect a security claim to something another developer can inspect and reproduce.

## Start with the attacker's capabilities

Assume an observer has captured one valid request. They can resend it or change its bytes, but they do not know the signing secret. A legitimate delivery service can also retry after a timeout. Those actors have different intentions, yet the receiver may see identical messages.

The protected asset is the application's business state. A signature is one control on the path to changing that state; it is not the state itself. The lab therefore separates four questions:

- Authenticity: does the message match the shared-key signing contract?
- Freshness: is the authenticated timestamp within the accepted interval?
- Identity: has this event ID already been completed, and does its content match?
- Execution: can the application commit the intended effect without an inconsistent intermediate state?

A compromised signing key is outside this model. Anyone holding the key can produce an authentic new message. The receiver still needs business authorization and validation; cryptography cannot determine whether an invoice should actually be marked paid.

## Authenticate the bytes that arrived

The lab's custom signing envelope contains a protocol label, integer timestamp, constrained event ID, and raw request body. Newlines separate the fields. Event IDs cannot contain newlines, so the boundary is unambiguous under this contract.

```
webhook-trust-v1\n
1700000000\n
evt_1\n
{"type":"invoice.paid","invoice_id":"inv_7","amount":50}
```

The displayed lines explain the envelope; the implementation concatenates the exact bytes. It calculates HMAC-SHA256 and uses `hmac.compare_digest` to compare the expected and supplied digest. Python documents this comparison as avoiding content-based short-circuit behavior. That is not a claim that every branch in the entire receiver takes constant time. [Python hmac documentation](#).

Parsing first would introduce an unnecessary ambiguity. Two JSON documents can mean the same thing while using different whitespace or key order. Reconstructing JSON after parsing does not reproduce the original bytes. The tests therefore change whitespace and key order while retaining the original signature; authentication rejects both changes.

There is also a test with malformed JSON and an invalid signature that replaces the JSON parser with a function that would fail if called. The rejection occurs without invoking it. This verifies ordering, rather than merely asserting that the source code looks sensible.

## A recent replay is still recent

The stateless verifier accepts the same authentic, fresh fixture twice. That is deliberate: a function with no history cannot know whether it has previously seen the event.

A timestamp reduces the interval during which a captured request is acceptable. In this lab, the signed timestamp must be within 300 seconds of the supplied clock, including both endpoints. The choice is a teaching assumption, not a recommendation for every provider. Clock skew, delivery delay, and the provider's signing and retry contract should determine a real integration's policy.

The time tests sign the old or future timestamp itself. This detail matters. Changing a timestamp while keeping its original signature creates a tampered request; rejecting it does not establish that the freshness control rejects an authentic old request.

The evidence distinguishes an invalid signature from an authenticated message outside the window. A replay inside the accepted window can pass both controls. OWASP likewise treats authenticated timestamps and event-ID deduplication as separate controls when the publisher's protocol supports them. [OWASP Webhook Security Cheat Sheet](#).

## Give event identity a durable meaning

The consumer stores the event ID and a digest of the authenticated body. A second delivery with the same ID and body can be classified as a duplicate. A newly signed message using the same ID with a different body is a conflict under this lab's contract.

That distinction is useful operationally. A routine retry should not look like a content-integrity incident, and a conflicting identity should not silently replace the original record. The test for conflicting reuse checks that the existing event and effect counts remain unchanged.

The database belongs to one authenticated sender. A shared multi-tenant inbox would need a sender namespace derived from trusted configuration or verified identity. Taking an unauthenticated sender label from the request would not establish that boundary.

Event identity is not business identity either. Two different event IDs can refer to the same invoice. This lab allows distinct event IDs; it does not enforce an invoice-level uniqueness rule. A payment application would need its own domain constraints. A narrowly stated guarantee is more useful than calling all repetition “solved.”

## The dangerous word is “processed”

Imagine a receiver that writes “seen” before calling a remote service. It crashes between those operations. The next delivery finds the marker and returns “already processed,” although the action never happened.

Reverse the order and a different failure appears: the remote action succeeds, but the process crashes before recording it. A retry may repeat the action.

The lab keeps its effect inside SQLite precisely so this boundary is inspectable. Its simulated action is a row in an `effects` table. The event row and effect row are written inside the same transaction. `BEGIN IMMEDIATE` obtains the write transaction before the check-and-insert sequence; competing attempts then inspect the committed result rather than independently claiming the same ID. See [SQLite transaction behavior](#).

```
db.execute("BEGIN IMMEDIATE")
# Inspect the event ID and authenticated body digest.
# If completed with identical content, return a duplicate result.
# Otherwise insert the event and its local simulated effect.
db.commit()
```

This abbreviated sketch explains the boundary; the executable implementation includes parameterized statements, validation, rollback, and explicit connection closing. The completed result is returned only after commit. A duplicate result says `duplicate_completed` because the matching completion and local effect were committed together.

## Test failures around the boundary

The suite injects an exception after the event claim and before the simulated effect. It expects neither row to remain and then verifies that a retry can complete. A second test inserts the local effect and raises before commit; it again expects both writes to roll back.

Another test sends sixteen deliveries through separate SQLite connections using eight worker threads. It checks for one applied local effect. Reopening the consumer verifies that duplicate state survives replacing the consumer object and its connections. These are specific local observations, not a distributed-systems benchmark or a power-loss certification.

The JSON evidence file includes nine scenarios. Run them with:

```
python -m unittest -v
python demo.py --output evidence/scenarios.json
```

The verification record contains the environment, result, and source hashes. It is preferable to a test count copied into several documents that can drift independently.

## What this design does not buy

SQLite cannot roll back a payment that another service already accepted. A real external effect may require an idempotency contract with that service, an outbox or durable worker, reconciliation, and an explicit handling policy for uncertain outcomes. Which design is appropriate depends on the effect and the remote system.

This sample also retains event records indefinitely. It does not implement cleanup, secret rotation, HTTP transport, rate limiting, or a provider adapter. Its body-size cap limits one request, but does not provide system-wide abuse protection. A valid signature does not grant business permission. The custom envelope is not an implementation of [RFC 9421 HTTP Message Signatures](#).

The practical result is a sharper review question. Instead of asking only whether a webhook is signed, ask which decision each control supports, where state becomes durable, and what happens when execution stops between steps. That produces a testable integration design, rather than a reassuring but incomplete authentication check.

## Reproduction and evidence

The [companion repository](#) contains `webhook_security.py`, `test_webhook_security.py`, `demo.py`, the threat model, and the machine-readable evidence. This article describes an independent local reference, not a vulnerability discovered in Cracken or a vendor integration.