SUNIMOD FIELD NOTES

The Webhook That Says “Paid” Can Be Forged: How Unsigned Events Turn Trusted Integrations Into Business Logic Attacks

An endpoint that trusts incoming JSON can be tricked into marking orders paid, granting access, or launching automations. Learn how forged webhooks work and how signatures, replay controls, idempotency, account binding, and state validation protect the business.

BY Christopher Jacobson

PUBLISHED July 21, 2026

TOPIC CISSP, Domain 1

https://sunimod.com/forged-webhooks-unsigned-events-business-logic-attacks/

The central lesson

A webhook payload is a **claim**, not proof. Before code acts on that claim, the receiver must establish who could have created it, whether the exact bytes were altered, whether the message is fresh, whether it has already been processed, whether it belongs to the expected provider account, and whether the requested state change is valid for the business record.

HTTPS protects the connection in transit and authenticates your server to the sender. It does not, by itself, prove that the client making a request to your public endpoint is the provider you intended to trust. A durable design combines transport security, provider-documented signature verification, replay resistance, idempotency, strict event and account binding, business-state validation, narrow downstream authority, and useful evidence.

Educational and defensive scope

The laboratory in this guide uses a fictitious payment provider, fictitious orders, a clearly fake signing secret, and a server bound only to `127.0.0.1`. It does not contact a payment processor, ecommerce platform, cloud account, customer system, or public endpoint. Test only applications and integrations you own or are explicitly authorized to assess.

Do not test a production webhook by sending fake “paid,” “refunded,” “shipped,” “approved,” “user created,” or similar events. Even a technically harmless request may trigger fulfillment, customer communication, accounting changes, access grants, or irreversible downstream automation. Use a local lab, an isolated test environment, or the provider’s supported sandbox and event-testing tools.

In this guide

1. What the vulnerability is

A webhook is an HTTP request one system sends to another when an event occurs. A payment platform may report that an invoice was paid. A source-control platform may report that code was pushed. A form service may report that a lead was created. The receiving application normally converts that message into a business action.

The vulnerability appears when the receiver treats the shape or content of the request as proof of origin. It may check that the method is `POST`, parse JSON, confirm that `type` equals `invoice.paid`, and then update the order. Those checks answer what the message *says*. They do not answer who was authorized to say it.

MITRE describes the broader weakness as CWE-345: Insufficient Verification of Data Authenticity: accepting data without adequately verifying its origin or authenticity. A related replay weakness is CWE-294: Authentication Bypass by Capture-replay, where a previously valid message can be reused to produce the same effect.

The separate questions a trustworthy webhook receiver must answer

Security property	Question	Failure when omitted
Transport security	Was the connection encrypted to the intended server?	Payloads or credentials may be exposed or modified in transit.
Authenticity and integrity	Could the expected sender have produced this exact message?	An arbitrary client can invent or alter a business event.
Freshness	Is the signed message recent enough to accept?	A captured valid request may remain reusable indefinitely.
Uniqueness	Has this provider event already been recorded?	Retries or replays can repeat fulfillment, credits, emails, or jobs.
Context binding	Does the event belong to the expected account, tenant, endpoint, and environment?	A valid event from another context may affect the wrong business data.
Semantic authorization	Is this event type allowed to cause this exact business transition?	A signed but unexpected field or event can overreach its intended purpose.
Durability and evidence	Can the system safely acknowledge, retry, reconcile, and investigate it?	Events are lost, duplicated, processed out of order, or impossible to reconstruct.

The severity is controlled by the action behind the endpoint. A webhook that adds a low-priority note has a different risk profile from one that releases inventory, grants a paid entitlement, changes DNS, triggers a deployment, approves a vendor, sends a password-reset link, or instructs another system to transfer data.

2. A solvable business opportunity: a Webhook and Integration Trust Review

This weakness can be turned into a focused, outcome-based project rather than an undefined “secure everything” engagement. A Webhook and Integration Trust Review connects the code-level receiver to the business workflow it controls.

Discover the integration boundaries

Inventory every inbound event endpoint across websites, WordPress plugins, ecommerce systems, custom applications, serverless functions, automation platforms, CRMs, payment systems, source-control tools, support platforms, identity systems, and internal services. Record the provider, environment, owner, event types, secrets or public keys, downstream permissions, and business actions.

Trace each event to its business effect

Follow the event beyond the controller. Determine whether it updates an order, creates an account, grants access, generates a shipping label, sends customer email, invokes a privileged API, modifies a file, launches a deployment, or queues a later job. A receiver can appear harmless while placing a trusted message into a queue that performs the high-impact action minutes later.

Harden and prove the controls

Implement the provider’s documented signature method, verify the exact raw request body, add freshness and duplicate controls, bind events to the expected account and environment, validate business state, narrow downstream authority, and create negative tests that demonstrate forged, stale, modified, duplicate, and cross-account events are rejected safely.

Operationalize the integration

Assign ownership, establish secret or key rotation, document provider changes, monitor rejection and retry patterns, preserve useful evidence without logging secrets, and create an incident runbook. The customer is not buying an HMAC function. The customer is buying automation that can explain why an event was trusted, limit what it can do, and recover when trust is questioned.

3. How webhook trust works

A typical integration crosses several trust boundaries. Security can fail at any one of them, so the receiver should be designed as a controlled pipeline rather than a single controller method.

1. **The provider creates an event.** It assigns an event identifier, type, account context, timestamp, and payload according to its current schema.
2. **The provider authenticates the message.** Depending on the provider, it may compute a shared-secret message authentication code or create an asymmetric digital signature.
3. **TLS protects delivery.** The provider establishes an HTTPS connection to the registered endpoint and sends the headers and exact body.

4. **The edge limits obvious abuse.** The reverse proxy enforces method, body-size, timeout, and reasonable rate controls without pretending those controls prove identity.
5. **The receiver verifies before parsing for action.** It retrieves the raw bytes, verifies the documented signature, validates freshness, and rejects failure before business logic runs.
6. **The receiver binds context.** It confirms the event, header, endpoint, provider account, tenant, environment, and allowed event subscription match the integration record.
7. **The receiver records uniqueness durably.** A provider-plus-event identifier is inserted under a uniqueness constraint so retries cannot repeat the same side effect.
8. **The business service validates the transition.** It compares amount, currency, object identity, current state, version, and other authoritative facts before changing data.
9. **Downstream work runs with limited authority.** A queue or worker receives only the information and permissions needed for the approved action.
10. **Monitoring and reconciliation close the loop.** The team can compare provider delivery records, internal event records, state changes, and failed or delayed processing.

What an HMAC proves

RFC 2104 defines HMAC as a keyed message-authentication mechanism. In a common webhook design, the provider and receiver share a secret. The provider combines the exact message with that secret to generate a tag. The receiver independently computes the expected tag and compares it with the received tag.

Plaintext · conceptual signed-message construction used in the local lab

```
signed_message = timestamp + "." + event_id + "." + exact_raw_request_body
signature      = "sha256=" + HMAC-SHA-256(shared_secret, signed_message)
```

If the secret is known only to the intended parties, a matching HMAC provides evidence that someone possessing the secret created the tag and that the signed bytes were not changed afterward. It does **not** encrypt the payload, so HTTPS remains necessary. Because both parties know a shared secret, HMAC also does not provide public nonrepudiation.

Some providers use asymmetric signatures instead. The provider signs with a private key and the receiver verifies with a public key. Do not substitute the lab’s invented header names or message format for a real provider’s protocol. Use the provider’s current documentation and official verification library when one is available.

Why the raw body matters

The signature normally covers the body bytes that were sent. Parsing JSON and serializing it again can change whitespace, key order, escaping, number formatting, or Unicode representation while preserving the same logical object. Verification must therefore occur against the unmodified raw body before a framework, proxy, middleware, or controller transforms it.

What a valid signature does not prove

A valid signature is necessary, but it is not the entire authorization decision. It does not prove that the provider account itself was not compromised, that the secret was never exposed, that the event belongs to the correct connected account, that the message is new, that it arrived in order, that the object is still in the represented state, or that every payload field is safe to trust. High-consequence workflows need context and state validation after cryptographic verification.

4. How the attack chain works

No memory-corruption exploit is required. The attacker uses ordinary HTTP because the application failed to distinguish a trusted event source from an arbitrary client.

1. **A business adopts an event-driven integration.** A webhook is connected to payment, fulfillment, identity, support, deployment, content, or internal automation.
2. **The endpoint is exposed.** It must be reachable by the provider, so a public route, function URL, proxy path, plugin endpoint, or automation trigger accepts requests.
3. **The implementation trusts payload shape.** Code checks the event name and required fields but does not verify a provider signature—or verifies it after taking action.
4. **The endpoint or route becomes known.** It may appear in code, configuration, documentation, logs, tickets, screenshots, vendor dashboards, browser history, backups, or deployment output. Guessing is not required when operational data already contains it.
5. **A forged event is sent.** The request imitates the provider’s JSON and names a business object the receiver will accept.
6. **The receiver parses the message as legitimate.** A familiar event type such as `invoice.paid` reaches the same branch that handles a real provider delivery.
7. **The business action executes.** The application changes state, grants an entitlement, creates work, calls another API, or queues a downstream action.
8. **Retries or replays multiply the effect.** If event uniqueness and state-level idempotency are absent, one accepted message can be repeated.
9. **Normal automation obscures the cause.** Logs may show a successful webhook and a valid internal workflow, while lacking the evidence needed to show that the provider never sent the event.

Replay is a separate failure mode

An endpoint can verify a real signature and still be vulnerable to replay. If a valid request, its timestamp, and its signature can be captured from an unsafe log, debugging tool, compromised proxy, or provider dashboard, an attacker may resend the exact bytes. Signing a timestamp lets the receiver reject stale messages, while a durable event identifier prevents the same accepted event from producing the business effect twice within the allowed time window.

Conditions that determine severity

- Which event types are accepted, and can the provider configuration be narrowed?

- Which business objects can the payload name, and are identifiers predictable, disclosed, or cross-tenant?
- Does the handler trust amount, currency, role, recipient, destination, file path, URL, or status directly from the payload?
- What downstream account, API token, service role, queue worker, plugin, or automation receives the event?
- Can the event release goods, grant paid access, issue credits, change customer data, send messages, deploy code, or modify infrastructure?
- Are test and production endpoints, accounts, secrets, logs, and queues separated?
- Can a duplicate or out-of-order event repeat or reverse a state transition?
- Can responders compare the request with the provider’s authoritative event and delivery history?

5. An intentionally vulnerable endpoint

Do not deploy this example. It is intentionally unsafe and exists only to make the trust failure visible on localhost.

The lab models two fictitious orders in a local JSON file. The first order will be changed by a forged unsigned request. The second will be reserved for the signed receiver later in the guide.

JSON · fictitious local order state saved as state.json

```
{
  "orders": {
    "order-1042": {
      "expected_amount_minor": 12900,
      "currency": "USD",
      "status": "pending"
    },
    "order-2088": {
      "expected_amount_minor": 25900,
      "currency": "USD",
      "status": "pending"
    }
  },
  "processed_events": {}
}
```

The vulnerable endpoint checks the HTTP method, parses JSON, confirms the event type, and verifies that the named order exists. It then marks the order paid. Those checks may look responsible in a code review, but none establish that FictitiousPay sent the request.

PHP · intentionally vulnerable webhook receiver

```
<?php
declare(strict_types=1);

header('Content-Type: application/json; charset=utf-8');

if (($_SERVER['REQUEST_METHOD'] ?? '') !== 'POST') {
    http_response_code(405);
    echo json_encode(['accepted' => false, 'error' => 'POST required']);
    exit;
}
```

```php
$raw = file_get_contents('php://input');
$event = json_decode($raw ?: '', true);

if (!is_array($event)) {
    http_response_code(400);
    echo json_encode(['accepted' => false, 'error' => 'invalid JSON']);
    exit;
}

if (($event['type'] ?? '') !== 'invoice.paid') {
    http_response_code(202);
    echo json_encode(['accepted' => true, 'ignored' => true]);
    exit;
}

$orderId = (string) ($event['data']['order_id'] ?? '');
$statePath = __DIR__ . '/state.json';
$state = json_decode((string) file_get_contents($statePath), true);

if (!isset($state['orders'][$orderId])) {
    http_response_code(404);
    echo json_encode(['accepted' => false, 'error' => 'unknown order']);
    exit;
}

$state['orders'][$orderId]['status'] = 'paid';
file_put_contents(
    $statePath,
    json_encode($state, JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES) . PHP_EOL,
    LOCK_EX
);

echo json_encode([
    'accepted' => true,
    'order_id' => $orderId,
    'status' => 'paid',
]);
```

Several distinct weaknesses combine:

- No signature, message-authentication code, client certificate, or other provider authentication is checked.
- The endpoint trusts the request’s `type` field as authority to mark an order paid.
- The amount and currency in the request are ignored instead of being compared with the expected order.
- No provider account or tenant is bound to the event.
- No timestamp limits how long a captured message remains usable.
- No event identifier is stored, so retries and replays are not deduplicated.
- The flat-file update is not an appropriate production transaction or audit design.
- The endpoint reports success immediately after changing state, with no independent provider confirmation for a high-consequence action.

The exploit is not “breaking JSON.” The exploit is supplying data the application already knows how to trust.

6. Safe localhost demonstration

This demonstration requires PHP 8.1 or later and `curl`. Python 3 is used later to create a legitimate signed request. It creates files only in `~/sunimod-webhook-lab` and binds the development server to the loopback address.

Save `state.json` and `vulnerable-webhook.php` from the preceding section into the new directory. The setup refuses to overwrite an existing path.

Bash · create the local lab directory and start a loopback-only PHP server

```
LAB_ROOT="$HOME/sunimod-webhook-lab"

if [[ -e "$LAB_ROOT" ]]; then
  printf 'Refusing to overwrite existing path: %s\n' "$LAB_ROOT" >&2
  exit 2
fi

mkdir -m 700 "$LAB_ROOT"
cd "$LAB_ROOT"

# Save state.json and vulnerable-webhook.php from this guide here first.
php -S 127.0.0.1:8088
```

Leave that terminal running. In another terminal, send an unsigned request that claims the first invoice was paid.

Bash · forge a harmless payment event against the localhost-only vulnerable endpoint

```
curl --fail-with-body -sS \
  -X POST http://127.0.0.1:8088/vulnerable-webhook.php \
  -H 'Content-Type: application/json' \
  --data-binary '{"id":"evt_forged_0001","type":"invoice.paid","data":{"order_id":"order-1042","amount_minor":
12900,"currency":"USD"}}'
```

The endpoint accepts the claim because the JSON has the expected shape:

Plaintext · response from the intentionally vulnerable endpoint

```
{"accepted":true,"order_id":"order-1042","status":"paid"}
```

Inspect the two fictitious order states without installing a database client:

Bash · read the local lab state

```
php -r '
$state = json_decode(file_get_contents("state.json"), true);
foreach ($state["orders"] as $orderId => $order) {
    echo $orderId . "=" . $order["status"] . PHP_EOL;
}
'
```

Plaintext · abridged local state after the forged request

```
order-1042=paid
order-2088=pending
```

What the lab proves: an endpoint that trusts event content without authenticating the sender cannot distinguish the fictitious provider from a local `curl` command. The forged request reaches the normal business branch and changes the order.

What the lab does not prove: it does not discover a real endpoint, compromise a provider, steal a secret, bypass TLS, access customer data, process a payment, or reproduce a specific vendor’s webhook protocol. It isolates one authorization fact: payload shape is not sender identity.

7. Why common defenses fail

Several controls are useful layers, but they are frequently mistaken for proof that the provider sent the request.

Controls that help but do not replace provider authentication

Control	What it helps with	Why it is insufficient alone
HTTPS	Encrypts the connection and authenticates the server certificate to the client.	Any internet client can establish its own HTTPS connection to a public endpoint. TLS does not automatically identify the application-level sender.
A long or secret-looking URL	Reduces casual discovery and scanning.	URLs appear in configuration, logs, dashboards, tickets, screenshots, monitoring, and backups. A path is not a message-bound proof.
`User-Agent` or provider-looking headers	May support diagnostics.	Ordinary HTTP clients can set those headers.
IP allowlisting	Can reduce traffic to documented provider egress ranges.	Ranges can change, delivery may use shared infrastructure, and network origin does not validate the exact message. Use it only as a provider-supported supplementary control.
JSON schema validation	Rejects malformed structure and unexpected types.	An attacker can send perfectly valid JSON that makes a false claim.
CSRF protection	Protects browser sessions from cross-site request forgery.	Webhooks are server-to-server requests and often have no browser session or CSRF token. Signature verification serves a different purpose.
Rate limiting	Reduces accidental floods and simple denial-of-service pressure.	One forged high-impact event may be enough. A slow attacker can stay below the limit.
An event identifier without a signature	Can support duplicate detection.	An attacker can invent a new identifier unless the message containing it is authenticated.
A signature without freshness or uniqueness	Authenticates the signed bytes.	A captured valid message may be replayed unless old timestamps and duplicate identifiers are rejected.

The design goal is layered assurance. HTTPS, edge limits, optional provider IP controls, schema validation, signatures, timestamps, event IDs, account binding, state validation, and least privilege solve different parts of the problem.

8. A secure verification design

The following endpoint is still a teaching example, not a drop-in integration for a real provider. Its fictitious protocol signs a timestamp, event identifier, and exact raw body with HMAC-SHA-256. Real providers choose their own header names, canonicalization, algorithms, key formats, timestamp rules, and rotation behavior.

The receiver’s order of operations

1. Require the expected HTTP method and content type.
2. Limit body size before expensive parsing or downstream work.
3. Read the exact raw body once.
4. Validate required signature metadata and a narrow timestamp window.
5. Retrieve the signing secret from runtime configuration, not source code.
6. Recompute the expected tag over the provider-defined signed message.
7. Use a timing-safe comparison such as PHP’s `hash_equals`.
8. Only after verification, parse JSON and validate the event identifier and provider account.
9. Allow only required event types and strictly validate their fields.
10. Acquire a transactional state boundary, deduplicate the event, validate the business transition, record evidence, and commit the state change together.

PHP · signed, freshness-checked, account-bound localhost receiver

```php
<?php
declare(strict_types=1);

const MAX_BODY_BYTES = 262144;
const MAX_CLOCK_SKEW_SECONDS = 300;
const PROVIDER_NAME = 'fictitiouspay';

function respond(int $status, array $payload): never
{
    http_response_code($status);
    header('Content-Type: application/json; charset=utf-8');
    echo json_encode($payload, JSON_UNESCAPED_SLASHES | JSON_THROW_ON_ERROR);
    exit;
}

function request_header(string $serverKey): string
{
    $value = $_SERVER[$serverKey] ?? '';
    return is_string($value) ? trim($value) : '';
}

if (($_SERVER['REQUEST_METHOD'] ?? '') !== 'POST') {
    respond(405, ['accepted' => false, 'error' => 'POST required']);
}

$contentType = strtolower(
    trim(explode(';', (string) ($_SERVER['CONTENT_TYPE'] ?? ''))[0])
);
if ($contentType !== 'application/json') {
    respond(415, ['accepted' => false, 'error' => 'application/json required']);
}

$raw = file_get_contents('php://input');
```

```
if (!is_string($raw) || $raw === '' || strlen($raw) > MAX_BODY_BYTES) {
    respond(400, ['accepted' => false, 'error' => 'invalid body size']);
}

$eventId = request_header('HTTP_X_FICTITIOUSPAY_EVENT_ID');
$timestampText = request_header('HTTP_X_FICTITIOUSPAY_TIMESTAMP');
$providedSignature = request_header('HTTP_X_FICTITIOUSPAY_SIGNATURE');

if (!preg_match('/^evt_[A-Za-z0-9_-]{8,80}$/D', $eventId)) {
    respond(401, ['accepted' => false, 'error' => 'invalid event metadata']);
}

if (!preg_match('/^[0-9]{10}$/D', $timestampText)) {
    respond(401, ['accepted' => false, 'error' => 'invalid event metadata']);
}

$timestamp = (int) $timestampText;
if (abs(time() - $timestamp) > MAX_CLOCK_SKEW_SECONDS) {
    respond(401, ['accepted' => false, 'error' => 'stale event']);
}

$secret = getenv('FICTITIOUSPAY_WEBHOOK_SECRET');
$expectedAccount = getenv('FICTITIOUSPAY_ACCOUNT_ID');
if (
    !is_string($secret) || $secret === '' ||
    !is_string($expectedAccount) || $expectedAccount === ''
) {
    respond(500, ['accepted' => false, 'error' => 'server configuration error']);
}

$signedMessage = $timestampText . '.' . $eventId . '.' . $raw;
$expectedSignature = 'sha256=' . hash_hmac('sha256', $signedMessage, $secret);
if (!hash_equals($expectedSignature, $providedSignature)) {
    respond(401, ['accepted' => false, 'error' => 'invalid signature']);
}

try {
    $event = json_decode($raw, true, 512, JSON_THROW_ON_ERROR);
} catch (JsonException) {
    respond(400, ['accepted' => false, 'error' => 'invalid JSON']);
}

if (!is_array($event) || ($event['id'] ?? null) !== $eventId) {
    respond(400, ['accepted' => false, 'error' => 'event identifier mismatch']);
}

if (($event['account_id'] ?? null) !== $expectedAccount) {
    respond(403, ['accepted' => false, 'error' => 'account mismatch']);
}

if (($event['type'] ?? null) !== 'invoice.paid') {
    respond(202, ['accepted' => true, 'ignored' => true]);
}

$data = $event['data'] ?? null;
if (!is_array($data)) {
    respond(422, ['accepted' => false, 'error' => 'event rejected']);
}

$orderId = $data['order_id'] ?? null;
$amountMinor = $data['amount_minor'] ?? null;
$currency = $data['currency'] ?? null;

if (!is_string($orderId) || !preg_match('/^order-[0-9]{4}$/D', $orderId)) {
    respond(422, ['accepted' => false, 'error' => 'event rejected']);
}

if (!is_int($amountMinor) || $amountMinor < 1) {
    respond(422, ['accepted' => false, 'error' => 'event rejected']);
}

if (!is_string($currency) || !preg_match('/^[A-Z]{3}$/D', $currency)) {
    respond(422, ['accepted' => false, 'error' => 'event rejected']);
}
```

```
$statePath = __DIR__ . '/state.json';
$handle = fopen($statePath, 'c+');
if ($handle === false || !flock($handle, LOCK_EX)) {
    respond(503, ['accepted' => false, 'error' => 'state store unavailable']);
}

try {
    rewind($handle);
    $stateText = stream_get_contents($handle);
    $state = json_decode($stateText ?: '{}', true, 512, JSON_THROW_ON_ERROR);
    $state['orders'] ??= [];
    $state['processed_events'] ??= [];

    if (isset($state['processed_events'][$eventId])) {
        flock($handle, LOCK_UN);
        fclose($handle);
        respond(200, ['accepted' => true, 'duplicate' => true]);
    }

    $order = $state['orders'][$orderId] ?? null;
    if (!is_array($order)) {
        throw new RuntimeException('unknown order');
    }

    if ((int) ($order['expected_amount_minor'] ?? -1) !== $amountMinor) {
        throw new RuntimeException('amount mismatch');
    }

    if (($order['currency'] ?? null) !== $currency) {
        throw new RuntimeException('currency mismatch');
    }

    if (($order['status'] ?? null) === 'cancelled') {
        throw new RuntimeException('invalid state transition');
    }

    $state['processed_events'][$eventId] = [
        'provider' => PROVIDER_NAME,
        'type' => 'invoice.paid',
        'payload_sha256' => hash('sha256', $raw),
        'received_at_utc' => gmdate(DATE_ATOM),
    ];
    $state['orders'][$orderId]['status'] = 'paid';

    $encoded = json_encode(
        $state,
        JSON_PRETTY_PRINT | JSON_UNESCAPED_SLASHES | JSON_THROW_ON_ERROR
    ) . PHP_EOL;

    rewind($handle);
    $written = ftruncate($handle, 0) ? fwrite($handle, $encoded) : false;
    if ($written !== strlen($encoded) || !fflush($handle)) {
        throw new RuntimeException('state write failed');
    }

    flock($handle, LOCK_UN);
    fclose($handle);

    respond(200, [
        'accepted' => true,
        'order_id' => $orderId,
        'status' => 'paid',
    ]);
} catch (RuntimeException $exception) {
    error_log('Webhook event rejected: ' . $exception->getMessage());
    flock($handle, LOCK_UN);
    fclose($handle);
    respond(422, ['accepted' => false, 'error' => 'event rejected']);
} catch (Throwable $exception) {
    error_log('Webhook handler failed: ' . $exception::class);
    flock($handle, LOCK_UN);
    fclose($handle);
```

```
    respond(500, ['accepted' => false, 'error' => 'internal error']);
}
```

The code intentionally returns similar rejection messages for business validation failures while writing a limited internal reason to the server log. It records a SHA-256 payload digest rather than the signing secret. Whether a production system may retain raw payloads depends on the payload’s sensitivity, legal obligations, operational need, and retention policy.

The JSON file is a lab convenience. File locking demonstrates one atomic boundary on a single local process, but it is not a production event store. A real application should use a transactional database or durable queue, a uniqueness constraint on provider and event identifier, safe retry semantics, backups, monitoring, and tested failure handling.

A production event-record pattern

The exact schema depends on the database and privacy requirements. The important property is a durable uniqueness rule that is committed with the processing decision, not an in-memory “seen” list that disappears on restart.

SQL · illustrative durable webhook event record

```
CREATE TABLE webhook_events (
    provider_name VARCHAR(80) NOT NULL,
    event_id VARCHAR(160) NOT NULL,
    event_type VARCHAR(160) NOT NULL,
    account_id VARCHAR(160) NOT NULL,
    payload_sha256 CHAR(64) NOT NULL,
    received_at_utc TIMESTAMP NOT NULL,
    processed_at_utc TIMESTAMP NULL,
    processing_result VARCHAR(40) NOT NULL,
    PRIMARY KEY (provider_name, event_id)
);

CREATE INDEX webhook_events_account_received_idx
    ON webhook_events (account_id, received_at_utc);
```

A uniqueness violation should be handled as an expected duplicate outcome, not a server crash. For high-value effects, the event record and business state change should share a transaction or an outbox pattern so a process failure cannot acknowledge one while losing the other.

Send a legitimate signed event

Save the secure receiver as `secure-webhook.php` and the following sender as `send-signed-event.py` in the lab directory. The sender signs the exact compact JSON bytes that it transmits.

Python · create and send a signed fictitious event to localhost

```
#!/usr/bin/env python3
"""Send a signed, fictitious webhook event to the localhost lab."""

from __future__ import annotations

import hashlib
import hmac
import json
import os
import time
import urllib.error
```

```
import urllib.request

URL = "http://127.0.0.1:8088/secure-webhook.php"
EVENT_ID = os.environ.get("EVENT_ID", "evt_demo_00000001")
SECRET = os.environ["FICTITIOUSPAY_WEBHOOK_SECRET"].encode("utf-8")

payload = {
    "id": EVENT_ID,
    "type": "invoice.paid",
    "account_id": "acct-demo-001",
    "data": {
        "order_id": "order-2088",
        "amount_minor": 25900,
        "currency": "USD",
    },
}

body = json.dumps(
    payload,
    separators=(",", ":"),
    ensure_ascii=False,
).encode("utf-8")
timestamp = str(int(time.time()))
signed_message = b".".join(
    [timestamp.encode("ascii"), EVENT_ID.encode("ascii"), body]
)
signature = "sha256=" + hmac.new(
    SECRET,
    signed_message,
    hashlib.sha256,
).hexdigest()

request = urllib.request.Request(
    URL,
    data=body,
    method="POST",
    headers={
        "Content-Type": "application/json",
        "X-FictitiousPay-Event-Id": EVENT_ID,
        "X-FictitiousPay-Timestamp": timestamp,
        "X-FictitiousPay-Signature": signature,
    },
)

try:
    with urllib.request.urlopen(request, timeout=5) as response:
        print(f"status={response.status}")
        print(response.read().decode("utf-8"))
except urllib.error.HTTPError as error:
    print(f"status={error.code}")
    print(error.read().decode("utf-8"))
    raise SystemExit(1) from error
```

Start the loopback-only server with the fictitious secret and expected account in its environment:

Bash · start the secure localhost receiver with fictitious runtime values

```
export FICTITIOUSPAY_WEBHOOK_SECRET='whsec_demo_only_7bf4a21d9c6e'
export FICTITIOUSPAY_ACCOUNT_ID='acct-demo-001'
php -S 127.0.0.1:8088
```

In another terminal, run the sender twice with the same event identifier:

Bash · send one valid event and then a duplicate

```
cd "$HOME/sunimod-webhook-lab"
export FICTITIOUSPAY_WEBHOOK_SECRET='whsec_demo_only_7bf4a21d9c6e'
```

```
python3 send-signed-event.py
python3 send-signed-event.py
```

Plaintext · expected shape of the two signed responses

```
status=200
{"accepted":true,"order_id":"order-2088","status":"paid"}
status=200
{"accepted":true,"duplicate":true}
```

The first request passes signature, freshness, account, field, and state checks. The second request carries a new valid signature but the same event identifier, so the receiver recognizes it as a duplicate and does not repeat the state transition.

An unsigned request to the secure endpoint is rejected before order logic runs:

Bash · send the same kind of unsigned claim to the secure endpoint

```
curl -sS \
  -X POST http://127.0.0.1:8088/secure-webhook.php \
  -H 'Content-Type: application/json' \
  --data-binary '{"id":"evt_forged_0002","type":"invoice.paid","account_id":"acct-demo-001","data":
{"order_id":"order-2088","amount_minor":25900,"currency":"USD"}}'
```

Plaintext · rejection before business processing

```
{"accepted":false,"error":"invalid event metadata"}
```

Implementation details that matter

- **Follow the provider exactly.** Do not invent a canonicalization rule, copy another vendor’s header format, or assume every provider uses HMAC.
- **Verify before parsing for action.** Middleware that consumes or changes the body can make correct verification impossible.
- **Compare safely.** PHP documents `hash_equals` as a timing-attack-safe comparison; the known expected value belongs in the first argument and the untrusted received value in the second.
- **Keep clocks reliable.** Timestamp checks require monitored time synchronization and an intentionally chosen tolerance that accounts for legitimate delivery delay.
- **Rotate without blind spots.** When a provider supports an overlap period, accept the documented active keys only for that controlled window, identify which key verified, and remove the retired key on schedule.
- **Separate environments.** Test and production should have different endpoints, accounts, secrets or key pairs, queues, data, and monitoring.
- **Do not log secrets or signature bases.** Log the provider, event identifier, account, verification outcome, payload digest, handler version, state decision, and timestamps according to policy.

9. Idempotency, ordering, and state validation

A cryptographically valid event can still produce an incorrect business result when delivery behavior and state transitions are treated casually. Provider retries, manual redelivery, network timeouts, concurrent delivery, and out-of-order events are normal integration conditions—not exceptional attacker behavior.

Event-level idempotency

Record the provider namespace and event identifier under a database uniqueness constraint. The provider name matters because two providers may use the same identifier format. The endpoint should return a successful duplicate outcome only after confirming the previously recorded event represents the same expected context.

Business-level idempotency

Different event identifiers may describe the same object or state. A provider may legitimately create two events, an operator may resend an event, or a workflow may produce related notifications. The business action itself must therefore be safe to repeat. “Set order to paid if it is pending and the authoritative amount matches” is safer than “increment paid count and issue a credit every time this event type arrives.”

Out-of-order delivery

Do not assume delivery order equals creation order. Stripe’s current webhook guidance, for example, states that event order is not guaranteed and recommends retrieving missing objects when necessary. A robust integration uses provider object versions, event creation time, monotonic sequence data when offered, or a read-only provider API call to determine current authoritative state.

Examples of state-aware decisions

Incoming claim	Required local checks	Safer outcome
Invoice paid	Expected provider account, invoice mapping, amount, currency, current status, and authoritative payment state for high-value fulfillment.	Transition only the matching pending order; otherwise hold for reconciliation.
Subscription canceled	Customer and subscription binding, effective date, current plan, newer event/version, and contractual grace period.	Schedule the permitted entitlement change rather than deleting access immediately from one field.
User created	Tenant, approved identity source, email/domain policy, role allowlist, and invitation state.	Create the minimum account state without granting an administrator role from payload data.
Deployment approved	Repository, branch, environment, commit, approver policy, artifact identity, and current release window.	Queue a separately authorized deployment rather than executing production changes inside the webhook process.
File ready	Provider object ownership, expected MIME/type, size, storage location, malware scanning, and filename handling.	Retrieve through a constrained worker and quarantine until validation completes.

Independent confirmation is selective, not universal. Calling the provider API for every low-risk event may add cost, latency, and a new availability dependency. Reserve authoritative callbacks or

reconciliation for actions whose business impact justifies the additional assurance, and use a least-privilege read credential that cannot create payments or change account configuration.

Acknowledgment and queues

Providers often impose short response deadlines and retry unsuccessful deliveries. Verify the request, validate the minimum required context, record the event durably, and then acknowledge according to the provider’s documented behavior. Move slower downstream work to a queue with bounded retries and a dead-letter or reconciliation path. Never return success before the system has retained enough state to avoid losing the event.

10. How to audit your integrations

Begin with a business-service inventory, not a search for one header name. The same organization may receive webhooks through custom PHP, WordPress REST routes, ecommerce plugins, serverless functions, workflow platforms, low-code automations, API gateways, vendor connectors, and internal message relays.

Build an integration record

The following provider-neutral JSON is an inventory example, not a configuration file. It captures the ownership, trust method, business actions, and evidence needed to operate the integration.

JSON · fictitious webhook and business-control inventory

```
{
  "integration": "FictitiousPay production events",
  "business_owner": "Revenue Operations",
  "technical_owner": "Web Platform",
  "endpoint": "https://hooks.northwind.example/fictitiouspay",
  "environment": "production",
  "provider_account_id": "acct-example-001",
  "accepted_event_types": [
    "invoice.paid",
    "invoice.payment_failed"
  ],
  "business_actions": [
    "release-order-for-fulfillment",
    "update-customer-balance"
  ],
  "signature_method": "provider-documented HMAC-SHA-256",
  "raw_body_verified": true,
  "freshness_window_seconds": 300,
  "deduplication_key": "provider_name + event_id",
  "authoritative_state_check": "required before high-value fulfillment",
  "secret_location": "managed secret store reference only",
  "rotation_owner": "Web Platform",
  "last_rotation_test": "2026-06-15",
  "last_negative_test": "2026-07-01",
  "incident_runbook": "IR-WEBHOOK-003"
}
```

Perform a read-only code search

From a repository you are authorized to inspect, the following searches can help locate likely inbound routes and nearby verification controls. They do not evaluate runtime configuration, vendor SDK

behavior, generated routes, proxies, serverless settings, plugin code outside the repository, or whether the controls are correct.

Bash · inventory likely webhook routes and verification-related code

```
rg -n --hidden \
  --glob '!vendor/**' \
  --glob '!node_modules/**' \
  --glob '!storage/**' \
  '(webhook|callback|php://input|register_rest_route|admin_post_nopriv)' .

rg -n --hidden \
  --glob '!vendor/**' \
  --glob '!node_modules/**' \
  '(hash_hmac|hash_equals|signature|webhook[_-]?secret|event[_-]?id|idempot)' .
```

A route match with no nearby signature code is a review lead, not proof of a vulnerability. Verification may live in middleware, a gateway, a provider SDK, or another service. Conversely, a match for `hash_hmac` does not prove the correct body, key, algorithm, timestamp, or comparison is used.

Review the edge without confusing it for authentication

Method restrictions, body limits, timeouts, request-rate controls, and a dedicated host can reduce operational risk. The following Nginx example is structural only. Confirm syntax, placement, TLS configuration, provider delivery behavior, proxy trust, and capacity in a test environment before adapting it.

Nginx · supplementary method, body-size, and rate controls

```
# http context
limit_req_zone $binary_remote_addr zone=webhook_per_ip:10m rate=10r/s;

server {
    listen 443 ssl;
    server_name hooks.northwind.example;

    location = /fictitiouspay {
        limit_except POST {
            deny all;
        }

        client_max_body_size 256k;
        limit_req zone=webhook_per_ip burst=20 nodelay;
        proxy_pass http://127.0.0.1:9000;
    }
}
```

Rate limiting by source address can behave poorly when a provider uses a shared or changing egress fleet. It must not become the sole authentication mechanism, and limits must account for legitimate bursts, retries, and disaster recovery.

Authorized audit checklist

- List every inbound webhook endpoint, including disabled, legacy, test, plugin, low-code, and vendor-managed routes.
- Record the business owner, technical owner, provider account, tenant, environment, event types, and downstream actions.

- Confirm the provider’s current documented verification method and official SDK version.
- Verify the exact raw request body is available before parsing or mutation.
- Confirm failed signatures stop processing before queues, database updates, emails, or API calls.
- Confirm timestamp or sequence validation matches the provider protocol and monitored clock behavior.
- Confirm durable duplicate detection uses provider plus event identifier and survives restarts and concurrency.
- Test business-level idempotency with two distinct events representing the same state.
- Test out-of-order delivery and stale state transitions.
- Bind the event to the expected provider account, tenant, endpoint, and environment.
- Validate amount, currency, object mapping, role, destination, URL, filename, and other high-risk fields against local expectations.
- Subscribe only to event types the integration requires.
- Separate test and production secrets, accounts, data, queues, logs, and alerts.
- Document secret or public-key rotation, emergency revocation, and provider ownership changes.
- Review logs for sensitive payloads, full authorization headers, signing secrets, and long-lived raw bodies.
- Confirm monitoring can distinguish invalid signature, stale request, duplicate event, rejected state, processing failure, and provider delivery outage.
- Run a provider-supported negative test and a reconciliation exercise without touching real customer transactions.

Prioritize by business effect

Review first the endpoints that can release goods, grant paid access, change roles, send money-related instructions, create refunds or credits, export data, deploy code, modify infrastructure, write customer records, or invoke a privileged downstream service. A low-volume webhook can be critical when one accepted event has a high consequence.

11. Potential business repercussions

The actual impact depends on the endpoint’s authority, the fields it trusts, downstream automation, detection time, and the organization’s ability to reconcile with the provider. The same missing signature can cause a minor data-quality issue in one integration and a major operational incident in another.

How a forged or replayed event can affect a business

Impact path	Possible repercussions	Evidence needed
Fraudulent fulfillment	Goods, digital downloads, appointments, services, or licenses may be released without an authoritative payment or approval.	Provider event history, order timeline, fulfillment records, inventory changes, delivery logs, and handler decisions.

Impact path	Possible repercussions	Evidence needed
Unauthorized entitlement	Accounts may receive paid features, higher limits, membership, credits, or roles they did not earn.	Event IDs, account mapping, role changes, session history, downstream access logs, and current provider state.
Duplicate business action	Repeated emails, shipments, credits, tickets, notifications, invoices, or background jobs may increase cost and confuse customers.	Duplicate identifiers, queue records, idempotency keys, job attempts, and business-object history.
Data integrity failure	Orders, subscriptions, customer status, accounting records, CRM fields, or operational dashboards may no longer match the authoritative system.	Before-and-after records, database audit history, provider exports, reconciliation reports, and application logs.
Privacy or confidentiality impact	A forged event may trigger an export, notification, account link, file retrieval, or workflow that exposes data to an unintended recipient.	Payload fields, data-access logs, message delivery records, storage access, recipient lists, and applicable data classification.
Operational interruption	Teams may pause checkout, fulfillment, account provisioning, deployments, or integrations while they determine which events are trustworthy.	Endpoint configuration, delivery failures, queue depth, outage timeline, manual workarounds, and recovery decisions.
Financial and reporting error	Revenue recognition, balances, refunds, commissions, inventory, tax records, or customer statements may require reconciliation and correction.	Accounting source records, provider settlement data, internal ledgers, correction entries, and approval history.
Contractual and reputational harm	Customers and partners may question the reliability of automated transactions, access decisions, and incident communication.	Confirmed scope, affected records, control history, response timeline, remediation evidence, and accurate communications.

Do not assume every unsigned endpoint has already been exploited. A finding establishes that the application lacks a dependable source-authentication control. Incident conclusions require logs, provider records, business-state comparison, and other evidence.

12. How to respond to suspected forgery

If a webhook accepted an event the provider did not send, a signing secret may be exposed, or event records cannot be reconciled, treat the receiver and every downstream action as an incident boundary.

1. **Stop new high-impact effects.** Disable the affected endpoint, pause the queue, place fulfillment or entitlement changes into review, or switch the integration to a safe fail-closed mode. Avoid destroying pending evidence.
2. **Preserve delivery and application evidence.** Retain reverse-proxy logs, request identifiers, provider delivery records, event IDs, timestamps, payload digests, handler versions, queue messages, job attempts, database audit history, and downstream API logs according to legal and privacy requirements.
3. **Establish the verification state.** Determine whether signatures were absent, optional, checked after processing, computed over modified bytes, compared unsafely, or accepted with an overly broad timestamp window.
4. **Use the provider’s authoritative records.** Compare internal event IDs and object state with the provider’s event and delivery history. Account for manual redelivery, retries, multiple endpoints, test mode, connected accounts, and event-version differences.

5. **Rotate or revoke trust material.** Follow the provider’s documented signing-secret or key rotation process. Update every legitimate receiver, remove retired material, and confirm that secrets were not copied into source, tickets, logs, or shared configuration.
6. **Review the provider account.** Check administrators, API credentials, endpoint registrations, connected accounts, apps, audit logs, multifactor protection, support access, and unexpected configuration changes. A valid signature can result from provider-account compromise.
7. **Define the exposure window.** Identify when the weak endpoint became reachable, when the secret or key may have been exposed, which handler versions ran, and which events produced downstream effects.
8. **Reconcile and reverse carefully.** Compare payments, orders, entitlements, refunds, credits, messages, exports, deployments, and customer records with authoritative sources. Reverse only with business-owner approval and evidence; automatic rollback can create a second incident.
9. **Rebuild the trust path.** Implement provider-specific verification, timestamp and duplicate controls, account binding, transactional processing, least privilege, monitoring, rotation, and negative tests before resuming normal automation.
10. **Coordinate the business response.** Involve leadership, legal counsel, privacy, finance, customer support, vendors, insurers, and affected customers according to the facts, contracts, jurisdictions, and records involved.

Two common mistakes: rotating the webhook secret and assuming prior effects are undone, or deleting suspicious queue messages and logs before responders can determine what happened. Rotation protects future verification; it does not reconcile actions already accepted.

13. Build a durable webhook governance model

The weakness returns when remediation is treated as one code patch. Webhook trust changes over time as providers add event types, schemas evolve, endpoints move, plugins update, secrets rotate, business workflows gain authority, owners leave, and test integrations become permanent.

Webhook lifecycle decisions and evidence

Lifecycle stage	Required decision	Evidence to retain
Adopt	Is a webhook the right mechanism, which provider account is trusted, and what is the maximum acceptable business effect?	Business owner, data flow, provider documentation, threat model, event allowlist, and approval.
Build	How are raw bytes verified, freshness enforced, duplicates rejected, fields validated, and downstream authority constrained?	Design review, code, tests, schema, permissions, queue behavior, and failure cases.
Deploy	Are endpoint, account, secret or key, DNS, TLS, proxy, environment, and monitoring correctly bound?	Deployment record, configuration references, test results, ownership, and rollback plan.
Operate	Are invalid signatures, retries, duplicates, delays, queue failures, and reconciliation differences visible and owned?	Dashboards, alerts, sampled reviews, provider delivery reports, and incident tickets.
Rotate	Can trust material change without accepting unknown keys, losing events, or leaving a retired secret active?	Rotation schedule, overlap window, validation results, retirement evidence, and named owner.
Change	Do new event types, fields, accounts, plugins, or downstream actions expand the original risk decision?	Change review, updated threat model, regression tests, and revised runbook.

Lifecycle stage	Required decision	Evidence to retain
Retire	Has the provider endpoint, route, secret, queue, DNS, automation, and documentation been removed together?	Offboarding checklist, revocation record, route removal, retained evidence, and owner sign-off.
Respond	Can the organization stop effects, verify provider history, reconcile state, rotate trust, and communicate accurately?	Runbook, exercise results, contacts, recovery steps, and post-incident improvements.

Useful policy is specific enough to test: “Every production webhook that changes business state must verify the provider’s documented signature over the raw request body before processing, reject stale messages, durably deduplicate provider event IDs, bind the provider account, validate the business transition, and have an assigned owner and rotation procedure.”

14. How Sunimod can help

Sunimod can turn webhook risk into a practical website, application, or integration project with a defined boundary and verifiable deliverables. The work can focus on one high-value payment or fulfillment path, a WordPress site and its plugins, a custom web application, or a broader set of business automations.

A useful engagement may include:

- An inventory of inbound webhook and callback routes, providers, accounts, environments, event types, owners, and downstream effects.
- A trust-boundary map from public endpoint through proxy, application, queue, database, vendor API, and business workflow.
- Provider-specific signature verification using the current official documentation or supported SDK.
- Raw-body handling, freshness checks, durable duplicate controls, account binding, schema validation, and state-transition rules.
- Least-privilege service accounts and separation between event receipt and privileged downstream work.
- Secret or public-key storage and rotation procedures that separate test from production.
- Negative tests for unsigned, modified, stale, duplicate, out-of-order, cross-account, amount-mismatch, and invalid-state events.
- Monitoring, reconciliation, incident-response guidance, and documentation the team can operate after delivery.
- Implementation support for custom software, APIs, WordPress integrations, ecommerce workflows, queues, and continued maintenance where appropriate.

The result should answer the business questions as clearly as the technical ones: Which automated claims can release value? Which system is authoritative? What stops a false event? How is a duplicate made harmless? Who rotates trust? What evidence would let the company explain an incident?

15. Key takeaways

- A webhook payload is data supplied to a public receiver; its familiar event name is not proof of origin.
- HTTPS protects transport but does not, by itself, authenticate the application-level sender.
- Provider-documented signature verification must use the exact raw body and occur before business action.
- A timestamp limits replay age, while a durable provider-plus-event identifier prevents duplicate processing.
- A valid signature does not replace provider-account binding, strict field validation, least privilege, or state-aware authorization.
- Retries, duplicate deliveries, manual redelivery, concurrency, and out-of-order events must be normal design cases.
- High-value actions may justify an independent read from the provider’s authoritative API before fulfillment or access is released.
- Logs should preserve useful event and decision evidence without exposing signing secrets or unnecessary sensitive payloads.
- The durable solution is an owned integration lifecycle: discover, design, test, deploy, monitor, rotate, change, retire, and respond.

16. Sources and further reading

- MITRE CWE-345: Insufficient Verification of Data Authenticity
- MITRE CWE-294: Authentication Bypass by Capture-replay
- RFC 2104: HMAC—Keyed-Hashing for Message Authentication
- OWASP API10:2023—Unsafe Consumption of APIs
- GitHub Docs: Validating webhook deliveries
- GitHub Docs: Best practices for using webhooks
- Stripe Docs: Receive events in your webhook endpoint
- PHP Manual: `hash_hmac`
- PHP Manual: `hash_equals`
- NIST SP 800-218: Secure Software Development Framework Version 1.1

Sources accessed July 21, 2026. Provider algorithms, header formats, retry behavior, SDKs, endpoint settings, and rotation features can change. Verify the current documentation for the exact provider and account before implementation.

Make automated events prove their identity before they change your business

Hire Sunimod to review the webhook paths behind payments, fulfillment, memberships, customer records, deployments, and custom automations—then replace implicit trust with verified signatures, replay controls, idempotent processing, accountable ownership, and tested recovery procedures.

Request a webhook and integration security project quote

Describe the provider, website or application, affected workflow, and desired business outcome. Do not submit passwords, API keys, webhook signing secrets, access tokens, payment-card data, or other sensitive credentials through the form.