Jack Richard

Code Debugging & Software Troubleshooting

I diagnose, troubleshoot and repair code issues in websites, e-commerce platforms, integrations and software systems.

MY SPECIALISATION

PHP and JavaScript debugging / WordPress and WooCommerce
Broken functionality / API errors / Database issues

ADDITIONAL PROJECT WORK

Custom automation, web scraping and software prototypes.

Available for remote freelance and contract projects.

Diagnose. Repair. Verify.

01 INITIAL ASSESSMENT

I review the reported symptoms and assess the likely scope.

02 TECHNICAL DIAGNOSIS

I reproduce the fault, inspect relevant code and identify its cause.

03 CODE REPAIR

I implement a focused fix and avoid unnecessary changes.

04 VERIFICATION

I test expected behaviour, edge cases and possible regressions.

05 HANDOVER

I explain the changes, evidence and any remaining limitations.

Selected projects & code fixes

Payment Guardian / FingerGate - Windows proof of concept

Six technical project examples - Before / After

ABOUT THE DEBUGGING EXAMPLES

Six examples of technical challenges in my areas of specialisation.

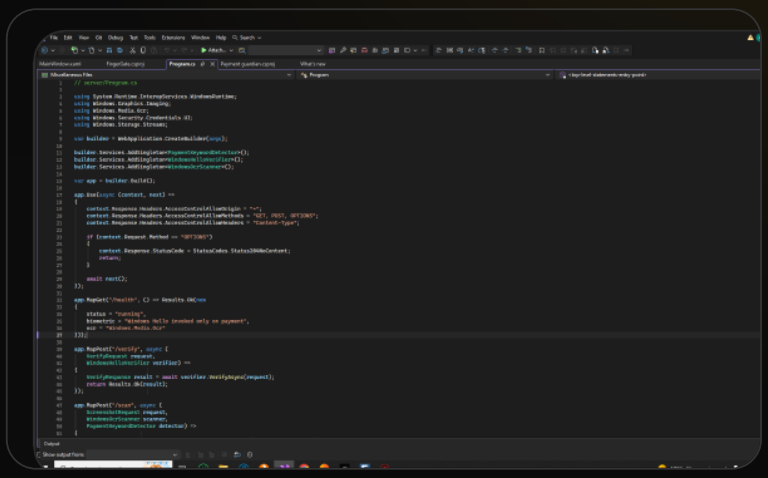
Each outlines a problem, code-level approach and verification plan.

Payment Guardian / FingerGate

Windows security proof-of-concept · Personal software project

THE PROBLEM

The project was inspired by recurring reports of children making purchases using saved family payment methods. The goal was to test whether an additional local verification layer could be triggered before sensitive payment-related actions.



Program.cs in Visual Studio: ASP.NET Core setup, Windows Hello verifier, OCR scanner, payment keyword detector and verification / scan endpoints.

1
Detect

Scan screen content for payment-related context using OCR and keyword detection.

2
Block

Display a full-screen verification gate with a countdown before continuation.

3
Verify

Invoke Windows Security / Windows Hello authentication as the extra confirmation step.

4
Test

Run and debug the prototype in Visual Studio and verify the flow against real purchase scenarios.

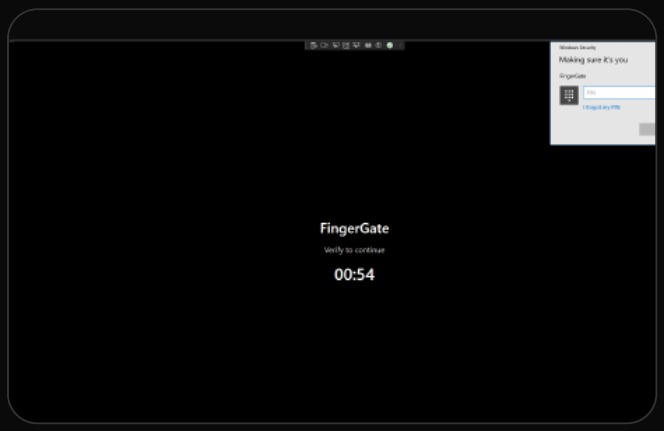
TECHNOLOGY USED

C# · .NET · Visual Studio · WPF / XAML · ASP.NET Core · Windows.Media.Ocr · Windows Hello · Screen Scanning · Debugging

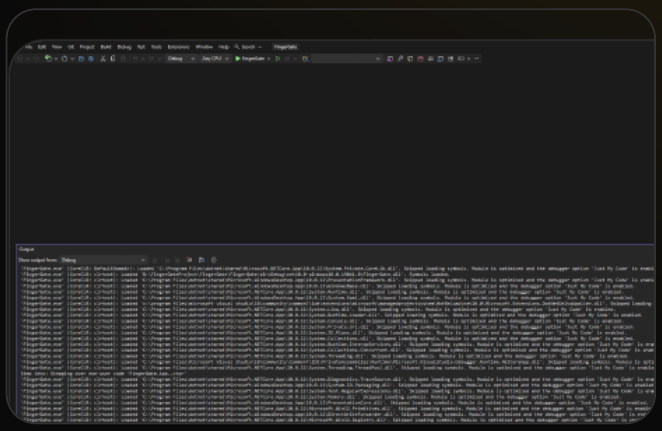
Status: working proof-of-concept tested locally. The project was intentionally left at prototype stage and was not packaged as a production installer.

FingerGate in action

From code to a running Windows verification flow



Running verification screen: FingerGate countdown with the Windows Security authentication prompt.

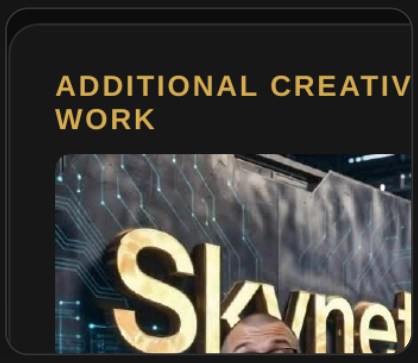


Visual Studio project and debug evidence showing the application running as FingerGate.exe.

WHAT THIS CASE STUDY DEMONSTRATES

- Translating a real-world problem into a working technical concept.
- Combining desktop UI, system authentication, screen analysis and application logic.
- Testing, debugging and iterating until the intended verification flow worked.
- Comfort working beyond website-only projects when a problem requires software.

ADDITIONAL CREATIVE WORK



Created a short branded AI promotional video with an AI presenter, visual branding, editing and final CTA as an additional content experiment.

“I work best when there is a real technical problem to solve: understand it, build a working solution, test it in practice and refine it until it does what it is supposed to do.”

PHP / ERROR HANDLING

SCENARIO: A customer-facing page crashes when an optional API response is missing.

BEFORE / PROBLEMATIC CODE

```
$response = fetchCustomer($id);
$name = $response['customer']['name'];
echo strtoupper($name);
```

AFTER / CORRECTED CODE

```
$response = fetchCustomer($id);
$name = $response['customer']['name'] ?? null;
if (!is_string($name) || $name === '') {
    error_log('Customer name unavailable');
    http_response_code(503);
    exit('Customer data temporarily unavailable');
}
echo htmlspecialchars(strtoupper($name), ENT_QUOTES, 'UTF-8');
```

ROOT CAUSE

The code assumes the API always returns a populated customer name. Missing or unexpected data can trigger warnings or a TypeError in strtoupper().

SOLUTION

Validate response shape before using the value, log the failure and return a controlled error.

VERIFICATION PLAN

Test valid, missing, null and non-string names. Confirm no fatal error and no internal details exposed.

WORDPRESS / PHP

SCENARIO: Two plugins declare the same global helper function, causing a fatal redeclaration error.

BEFORE / PROBLEMATIC CODE

```
// Plugin A
function sk_format_price($amount) {
    return number_format($amount, 2);
}
// Plugin B also defines sk_format_price()
```

AFTER / CORRECTED CODE

```
// Plugin B: use a unique namespace
namespace SkynetStudio\Pricing;
function format_price(float $amount): string {
    return number_format($amount, 2);
}
// Call as \SkynetStudio\Pricing\format_price(49.5);
```

ROOT CAUSE

Unscoped PHP function names share a global namespace. Loading two definitions with the same name causes a fatal error.

SOLUTION

Namespace the helper and update its callers. Audit other symbol collisions before deployment.

VERIFICATION PLAN

Activate both test plugins together; confirm no redeclaration and that formatting still works.

WOOCOMMERCE / VALIDATION

SCENARIO: A custom checkout field is treated as mandatory even when the field is absent from a particular checkout flow.

BEFORE / PROBLEMATIC CODE

```
add_action('woocommerce_checkout_process', function () {
    if (empty($_POST['delivery_note'])) {
        wc_add_notice('Delivery note required', 'error');
    }
});
```

AFTER / CORRECTED CODE

```
add_action('woocommerce_checkout_process', function () {
    // Optional field: validate only if supplied
    if (!isset($_POST['delivery_note'])) return;
    $note = sanitize_textarea_field(
        wp_unslash($_POST['delivery_note'])
    );
    if (mb_strlen($note) > 500) {
        wc_add_notice('Delivery note is too long', 'error');
    }
});
```

ROOT CAUSE

The custom validation blocks orders when an optional field is missing. It can break some checkout configurations.

SOLUTION

Match validation to the field requirement and sanitize the submitted text. For production, verify the field and hook work in the store's checkout implementation.

VERIFICATION PLAN

Test blank/filled field, long note, guest and logged-in checkout; test classic checkout and Blocks separately.

JAVASCRIPT / ASYNC

SCENARIO: A dashboard displays an empty screen after the API returns an HTTP error or invalid JSON.

BEFORE / PROBLEMATIC CODE

```
async function loadOrders() {
  const res = await fetch('/api/orders');
  const orders = await res.json();
  renderOrders(orders);
}
loadOrders();
```

AFTER / CORRECTED CODE

```
async function loadOrders() {
  try {
    const res = await fetch('/api/orders');
    if (!res.ok) throw new Error(`HTTP ${res.status}`);
    const orders = await res.json();
    if (!Array.isArray(orders)) throw new Error('Invalid data');
    renderOrders(orders);
  } catch (err) {
    console.error('Order load failed:', err);
    showError('Orders are temporarily unavailable.');
  }
}
loadOrders();
```

ROOT CAUSE

fetch() does not reject on HTTP 4xx/5xx. Unhandled parsing and rendering errors leave the interface without a useful fallback.

SOLUTION

Check HTTP status and response shape, catch failures and show a safe user-facing message.

VERIFICATION PLAN

Mock 200, 500, invalid JSON, wrong data shape and network failure; verify fallback UI.

SQL / PERFORMANCE

SCENARIO: An order history page slows down as the order table grows.

BEFORE / PROBLEMATIC CODE

```
SELECT * FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC;
```

AFTER / CORRECTED CODE

```
CREATE INDEX idx_orders_customer_created
  ON orders (customer_id, created_at);

SELECT id, created_at, total, status
FROM orders
WHERE customer_id = 42
ORDER BY created_at DESC
LIMIT 50;
```

ROOT CAUSE

A large result set and an unsuitable index can force expensive scans and sorting.

SOLUTION

Add an index aligned to filtering and ordering, select only required columns, and paginate. Check the actual execution plan before choosing an index.

VERIFICATION PLAN

Use EXPLAIN / EXPLAIN ANALYZE and representative data; compare latency and scanned rows before and after. No performance gain is claimed without measurement.

PHP / PDO / SECURITY

SCENARIO: A product search concatenates untrusted input directly into a SQL query.

BEFORE / PROBLEMATIC CODE

```
$term = $_GET['q'] ?? '';
$sql = "SELECT id, name FROM products WHERE name LIKE '%$term%'";
$rows = $pdo->query($sql)->fetchAll();
```

AFTER / CORRECTED CODE

```
$term = (string) ($_GET['q'] ?? '');
$term = str_replace(['!', '%', '_'], ['!!', '!%', '!_'], $term);
$stmt = $pdo->prepare(
    "SELECT id, name FROM products WHERE name LIKE :q ESCAPE '!'"
);
$stmt->execute(['q' => '%' . $term . '%']);
$rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

ROOT CAUSE

String concatenation allows untrusted input to change the SQL query. LIKE wildcards can also change search semantics.

SOLUTION

Use a prepared statement and escape LIKE wildcards when a literal substring search is intended. Example assumes a SQL dialect supporting ESCAPE and a configured PDO connection.

VERIFICATION PLAN

Test quotes, percent signs, underscores, empty searches and injection payloads against a disposable database.

Need a difficult code issue fixed?

I am available for focused debugging, troubleshooting, code repair and selected custom software projects.

CONTACT

info.skynetcustom@gmail.com

skynetstudio.com

Free initial assessment available for suitable enquiries.

Advanced Technical Project Examples

Twelve additional engineering scenarios covering e-commerce, backend development, APIs, performance and application security. Each shows an illustrative implementation pattern and verification strategy, not a claim of a completed client engagement.

07 | WooCommerce / Order Integrity — Idempotent payment webhook

Scenario: A gateway retries a payment callback and the store risks applying fulfilment twice.

BEFORE / RISKY APPROACH

```
$order = wc_get_order($order_id);
$order->payment_complete($transaction_id);
send_to_warehouse($order);
```

AFTER / ENGINEERING APPROACH

```
// Payment handler: transaction ID must be unique.
// Acquire a DB-level lock or unique event row first.
if (!claim_event_once($gateway_event_id)) {
    return new WP_REST_Response(['ok' => true], 200);
}
$order = wc_get_order($order_id);
if (!$order || !hash_equals(
    (string) $order->get_transaction_id(),
    (string) $transaction_id
)) {
    // Validate gateway signature + transaction separately.
    release_event_claim($gateway_event_id);
    return new WP_Error('invalid_payment', 'Mismatch', ['status' => 400]);
}
queue_fulfilment_once($order_id);
```

Root cause: Retries are normal; a callback without durable deduplication can repeat side effects.

Solution: Verify the provider signature and payment state before this handler; persist unique event claims and queue fulfilment using a unique order key. Helpers shown are application-specific.

Verification: Replay the same signed event concurrently, then simulate a crash and retry; verify one fulfilment and correct recovery.

08 | Shopify / Liquid — Variant-aware pricing and availability

Scenario: A theme shows the first variant price even after customers select a different variant.

BEFORE / RISKY APPROACH

```
<span>{{ product.price | money }}</span>
<button>Add to cart</button>
```

AFTER / ENGINEERING APPROACH

```
<span id="variant-price">
  {{ product.selected_or_first_available_variant.price | money }}
</span>
<button id="add-to-cart"
  {% unless product.selected_or_first_available_variant.available
%}disabled{% endunless %}>
  Add to cart
</button>
<script>
// In theme variant-change handler:
function syncVariant(variant) {
  document.querySelector('#variant-price').textContent =
    Shopify.formatMoney(variant.price, window.themeMoneyFormat);
  document.querySelector('#add-to-cart').disabled = !variant.available;
}
```

```
</script>
```

Root cause: Liquid is server-rendered; client-side selection requires the theme to update displayed state.

Solution: Use the selected variant on initial render and synchronise UI in the existing theme variant-change event. Shopify.formatMoney and themeMoneyFormat require compatible theme setup.

Verification: Test price changes, sold-out variants, deep links, currency formatting and add-to-cart payloads.

09 | Shopify / GraphQL — Rate-limit-aware bulk product sync

Scenario: A catalogue importer gets throttled and restarts from the beginning after interruptions.

BEFORE / RISKY APPROACH

```
for (const item of products) {
  await graphql(mutation, { item });
}
```

AFTER / ENGINEERING APPROACH

```
// Pseudocode: adapt to Admin GraphQL API version.
for (const batch of remainingBatches(cursor)) {
  const response = await shopifyGraphQL(batch);
  if (response.errors?.some(isThrottleError)) {
    await sleep(retryDelay(response));
    retrySameBatch();
    continue;
  }
  assertNoUserErrors(response.data);
  persistCheckpoint(batch.cursor);
}
```

Root cause: Unbounded mutation loops ignore API cost limits and do not retain a durable checkpoint.

Solution: Use API-supported bulk operations where appropriate, otherwise implement cost-aware retries and resumable checkpoints. Helper functions are integration-specific.

Verification: Inject 429/throttle responses, partial userErrors, interrupted workers and duplicate batches; compare resulting catalogue IDs.

10 | WordPress / REST Security — Permission-controlled admin endpoint

Scenario: A custom REST route exposes an administrative action to unauthorised visitors.

BEFORE / RISKY APPROACH

```
register_rest_route('studio/v1', '/rebuild', [
  'methods' => 'POST',
  'callback' => 'studio_rebuild_index',
]);
```

AFTER / ENGINEERING APPROACH

```
add_action('rest_api_init', function () {
  register_rest_route('studio/v1', '/rebuild', [
    'methods' => WP_REST_Server::CREATABLE,
    'callback' => 'studio_rebuild_index',
    'permission_callback' => function () {
      return current_user_can('manage_options');
    },
  ]);
});
```

Root cause: An endpoint without an explicit permission callback can expose privileged operations.

Solution: Require a capability check; authenticated cookie-based requests also need the standard REST nonce. Validate all input inside the callback.

Verification: Test logged-out, subscriber, editor and administrator requests, with and without valid nonces.

11 | PHP / Concurrency — Atomic stock reservation

Scenario: Two customers buy the last item at the same moment, producing overselling.

BEFORE / RISKY APPROACH

```
$stock = getStock($sku);
if ($stock > 0) {
  setStock($sku, $stock - 1);
}
```

AFTER / ENGINEERING APPROACH

```
// InnoDB / PDO; inside an existing transaction:
$pdo->beginTransaction();
$stmt = $pdo->prepare(
  'UPDATE inventory SET stock = stock - 1 '.
  'WHERE sku = ? AND stock >= 1'
);
$stmt->execute([$sku]);
if ($stmt->rowCount() !== 1) {
  $pdo->rollBack();
  throw new RuntimeException('Out of stock');
}
$pdo->commit();
```

Root cause: A read-then-write sequence is not atomic under concurrent requests.

Solution: Use a conditional atomic UPDATE in a transaction. For multi-item carts, establish consistent locking order and rollback on any failure.

Verification: Run parallel requests for one unit; verify exactly one successful reservation and no negative stock.

12 | JavaScript / Web Security — Server-side checkout price authority

Scenario: A browser submits a discounted total that can be altered in developer tools.

BEFORE / RISKY APPROACH

```
const total = cart.reduce((s, x) => s + x.price, 0);
await fetch('/checkout', {
  method: 'POST', body: JSON.stringify({ cart, total })
});
```

AFTER / ENGINEERING APPROACH

```
// Browser sends only product IDs and quantities.
await fetch('/checkout', {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify({
    lines: cart.map(x => ({ id: x.id, qty: x.qty }))
  })
});
// Server: authenticate, validate quantities,
// reprice from trusted catalogue and verify discounts.
```

Root cause: Client-calculated amounts are untrusted input.

Solution: Compute totals and discounts on the server from trusted product data; bind the final charge to the authenticated order and use gateway idempotency.

Verification: Tamper with price, quantity, discounts and currency; confirm server-side totals remain authoritative.

13 | Python / ETL — Resumable incremental data extraction

Scenario: A scheduled extraction duplicates records after a worker restarts.

BEFORE / RISKY APPROACH

```
for row in fetch_all():
    insert(row)
```

AFTER / ENGINEERING APPROACH

```
# SQLite illustration: upsert stable external IDs.
with sqlite3.connect('items.db') as db:
    db.executemany(
        "INSERT INTO items(id, payload) VALUES(?, ?) "
        "ON CONFLICT(id) DO UPDATE SET payload=excluded.payload",
        [(r['id'], json.dumps(r)) for r in page]
    )
# Save source cursor only after successful commit.
# Resume from the last committed cursor.
```

Root cause: Append-only writes and early cursor updates cause duplicates or missed records.

Solution: Use stable keys, upserts and durable checkpoints. For production, store the cursor in the same transactional database where possible.

Verification: Interrupt after each page, replay pages, change records and verify counts and final cursor.

14 | API / Distributed Systems — Retry-safe outbound requests

Scenario: A service retries a POST after timeout and creates duplicate invoices.

BEFORE / RISKY APPROACH

```
await fetch('/api/invoices', {
  method: 'POST', body: JSON.stringify(invoice)
});
```

AFTER / ENGINEERING APPROACH

```
const key = invoice.requestId; // stable across retries
const res = await fetch('/api/invoices', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Idempotency-Key': key
  },
  body: JSON.stringify(invoice)
});
if (!res.ok) throw new Error(`HTTP ${res.status}`);
// Server must enforce uniqueness of key + payload hash.
```

Root cause: A network timeout does not tell the caller whether the server committed the invoice.

Solution: Use a stable idempotency key and server-side durable deduplication. Only retry transient errors with bounded backoff.

Verification: Simulate response loss after commit, simultaneous retries and same-key/different-payload conflicts.

15 | SQL / Reporting — Pagination without missing records

Scenario: A high-traffic orders report skips or repeats rows while new orders arrive.

BEFORE / RISKY APPROACH

```
SELECT id, created_at FROM orders
ORDER BY created_at DESC
LIMIT 50 OFFSET 5000;
```

AFTER / ENGINEERING APPROACH

```
-- Keyset pagination: cursor is (created_at, id).
SELECT id, created_at FROM orders
WHERE created_at < :cursor_time
   OR (created_at = :cursor_time AND id < :cursor_id)
ORDER BY created_at DESC, id DESC
LIMIT 50;
-- Index: (created_at DESC, id DESC)
```

Root cause: Large offsets are expensive and shifting rows make page boundaries unstable.

Solution: Use a deterministic tie-breaker and keyset cursor; consider a snapshot/high-water mark for stable exports.

Verification: Insert rows between pages and compare duplicates, omissions and query plans.

16 | WordPress / Performance — Avoiding repeated remote API calls

Scenario: A product page calls an external service on every request, slowing page rendering.

BEFORE / RISKY APPROACH

```
$rates = wp_remote_get($api_url);
$price = json_decode(wp_remote_retrieve_body($rates), true);
```

AFTER / ENGINEERING APPROACH

```
$rates = get_transient('studio_rates_v1');
if ($rates === false) {
  $res = wp_remote_get($api_url, ['timeout' => 3]);
  if (!is_wp_error($res) &&
      wp_remote_retrieve_response_code($res) === 200) {
    $rates = json_decode(wp_remote_retrieve_body($res), true);
    if (is_array($rates)) {
      set_transient('studio_rates_v1', $rates, 300);
    }
  }
}
// On failure, use a defined safe fallback policy.
```

Root cause: Blocking remote calls increase latency and amplify third-party outages.

Solution: Cache validated responses with a suitable TTL, and define explicit stale-data or unavailable-state behaviour. For heavy traffic, add stampede protection.

Verification: Test timeout, malformed JSON, cache expiry and concurrent cache misses.

17 | WooCommerce / Background Jobs — Reliable asynchronous fulfilment

Scenario: A checkout handler performs a slow warehouse call and times out after payment.

BEFORE / RISKY APPROACH

```
add_action('woocommerce_payment_complete', function ($id) {
  warehouse_api_ship($id);
});
```

AFTER / ENGINEERING APPROACH

```
add_action('woocommerce_payment_complete', function ($id) {
```

```
    if (!as_next_scheduled_action(
        'studio_send_warehouse', ['order_id' => $id], 'studio')) {
      as_enqueue_async_action(
        'studio_send_warehouse', ['order_id' => $id], 'studio', true);
    }
  });
  // Worker must verify order payment and use a
  // durable fulfilment idempotency key.
  add_action('studio_send_warehouse', 'studio_worker', 10, 1);
```

Root cause: Synchronous external I/O blocks checkout; repeated events can enqueue repeated work.

Solution: Use Action Scheduler with correct installed API version, durable idempotency in the worker and observable retries. The example uses an application-specific worker.

Verification: Simulate API downtime, retries, duplicate payment events and interrupted workers.

18 | Security / Webhooks — HMAC verification and replay protection

Scenario: A public webhook endpoint accepts forged or replayed events.

BEFORE / RISKY APPROACH

```
const event = JSON.parse(req.body);
processEvent(event);
```

AFTER / ENGINEERING APPROACH

```
// Node.js: req.rawBody must be the exact received bytes.
const expected = crypto.createHmac('sha256', secret)
  .update(req.rawBody).digest('hex');
const received = req.headers['x-signature'];
if (!/^[a-f0-9]{64}$/i.test(received || ''))
  return res.sendStatus(401);
if (!crypto.timingSafeEqual(
  Buffer.from(expected, 'hex'), Buffer.from(received, 'hex')))
  return res.sendStatus(401);
// Also enforce timestamp window + unique event ID.
processVerifiedEvent(req.rawBody);
```

Root cause: Unauthenticated payloads can be forged; valid signatures alone do not stop replay.

Solution: Validate the provider-specific signature format and timestamp using raw bytes, then deduplicate event IDs before side effects.

Verification: Test altered payload, wrong key, stale timestamp, repeated ID and parallel deliveries.