HADRIAN

Application Penetration Test Report

Example LLC

REPORT DATE: 2026-06-01

Table of contents

Application Penetration Test Report

Client: Example LLC

Report date: 2026-06-01

Testing window: 2026-05-27 - 2026-05-29

Classification: Confidential

Reviewed by: Alex Morgan, Senior Security Engineer

Confidentiality Notice

This document contains confidential information about the security posture of the systems in scope. It is intended solely for the named recipient(s). Disclosure, copying, or redistribution to any party outside the intended audience without the prior written consent of Hadrian is prohibited. The findings in this report reflect the state of the application during the testing window only; subsequent changes may alter the applicability of any observation.

1. Executive Summary

1.1 Objective

This penetration test was commissioned to provide an independent assessment of the security posture of the Target-App web application at **example.com**. The objective was to identify risks that could (a) allow an unauthorized party to access tenant data, (b) permit an authenticated tenant user to cross tenant boundaries, or (c) disclose information that lowers the cost of a subsequent attack.

1.2 Findings Summary

A total of 7 risks were discovered along with 1 security hardening recommendation(s).

Severity Distribution:

Severity	Count
Critical	1
High	3
Medium	2
Low	1
Informational	1

Risks Overview:

ID	Risk Title	Severity	Category
R-01	Unauthenticated SQL Injection via JSON:API Filter Parameter	Critical	Injection
R-02	Cross-Tenant Insecure Direct Object Reference (IDOR)	High	Authorization & Authentication
R-03	Privilege Escalation via Mass Assignment on Account Update	High	Authorization & Authentication

R-04	Server-Side Request Forgery via Document Import	High	Server-Side Request Forgery
R-05	Broken Function-Level Authorization on Admin User Management	Medium	Authorization & Authentication
R-06	Stored Cross-Site Scripting in Document Title	Medium	Injection
R-07	Verbose Error and Stack Trace Disclosure	Low	Information Disclosure
R-08	Missing Security Headers and Insecure Cookie Attributes	Informational	Security Hardening

2. Scope and Rules of Engagement

2.1 In-Scope Targets

Primary target for this engagement: **example.com**

Credentials Tested:

Account	Login URL	Notes
Tenant A - Regular User	`https://example.com/login`	Low-privilege user account for testing authenticated areas and tenant boundaries.
Tenant A - Admin user	`https://example.com/login`	Administrative account for testing privilege boundaries and administrative features.
Tenant B - Regular User	`https://example.com/login`	
Tenant B - Admin user	`https://example.com/login`	

3. Hadrian Pentest Methodology

Testing follows Hadrian's AI-driven pentest methodology: a four-phase workflow run by autonomous specialist agents under human oversight. Coverage is aligned with OWASP Top 10 (2021) and maps to SOC 2, ISO 27001, NIS 2, and DORA control requirements.
Throughout every phase, agent activity is constrained by platform-enforced safeguards rather than the agent's own judgement: scope is enforced on every outbound request, tool calls and tool responses are screened independently of the agent, per-run budgets cap activity, and every reported finding must clear a separate validation step. Together these controls keep testing within the agreed rules of engagement, prevent runaway or hostile-input behaviour, and underpin Hadrian's commitment to scope-respecting, low-blast-radius, zero false-positive testing.

Phase 1: Reconnaissance

The engagement begins with informed discovery. Rather than starting blind, the agent is seeded with the target's existing asset intelligence and any documentation, credentials, or focus areas the customer provides. From there it crawls the application, performs directory and parameter discovery, and walks authenticated flows through a real browser. Discovered endpoints are normalized, deduplicated, and clustered into functional attack surfaces. Each surface is tagged with auth requirements, data sensitivity, and exclusion flags: destructive endpoints, third-party-owned services, and explicitly out-of-scope hosts are recorded here and excluded from all downstream phases.

Phase 2: Test Planning

A planner agent translates the attack surface into an executable test plan: a matrix of scanner-class against endpoint-group, prioritized by exposure and likely impact. The planner is the single point at which scope, exclusions, customer constraints, and concurrency limits are applied to the engagement. Specialists inherit a pre-filtered worklist; they cannot widen it.

Phase 3: Active Scanning

The planner dispatches work to specialist vulnerability agents: each focused on a single attack class (injection, access control, authentication, business logic, etc.) and equipped with its own tooling. Specialists are intentionally narrow: a misbehaving agent has access only to the tools its attack class requires, keeping blast radius bounded and failures isolated. Agents issue authenticated and unauthenticated requests, observe behaviour, and confirm weaknesses. While they run, the platform enforces a layered set of safeguards independently of the agent loop:

- **Scope enforcement** on every outbound request, so agents cannot reach assets outside the agreed perimeter even by accident.
- **Pre-execution screening of tool calls** against the engagement's safety rules: no denial-of-service, no destructive actions, no data mutation beyond minimal proof of exploit.
- **Prompt-injection detection on tool responses**, so a compromised or hostile target cannot steer the agent.
- **Per-run tool-call and failure budgets**, preventing loops, retries, or runaway scanning, and ensuring scanner concurrency stays within levels that won't interfere with production.
- **Parameter guardrails** that pin engagement-specific inputs (target host, auth context) so an agent cannot substitute alternative values.

Phase 4: Triage and Reporting

Findings from all scanners are normalized, deduplicated, and re-scored against a consistent risk model. Each candidate finding is then re-validated by an independent checker against its supporting evidence, a structural defence against hallucinated or unreproducible vulnerabilities, before a Hadrian security analyst reviews and signs off the final report.

3.1 Test Coverage

The table below summarizes the sub-areas exercised under each control category during this engagement and whether any confirmed findings were produced.

Category	Test Coverage	Result	Findings
Authorization & Authentication	Credential handling, session management, access control, IDOR	Findings	R-02, R-03, R-05
Injection	SQL, XSS, command, template, XXE, and deserialization injection	Findings	R-01, R-06
Server-Side Request Forgery	Internal network access via server-side request vectors	Findings	R-04
File & Directory Exposure	Path traversal, file upload bypass, directory access	Pass	-
Information Disclosure	Verbose errors, config exposure, sensitive data leaks	Findings	R-07
Security Hardening	Security headers, TLS, CSP, CORS, cookie configuration	Findings	R-08
Unpatched Technologies	Known CVEs in detected technologies and components	Pass	-

3.2 Category Coverage in Detail

- **Authorization & Authentication:** This category covers the full lifecycle of how users prove identity and how sessions are managed. Tests look for ways an attacker could bypass login, hijack an active session, weaken or skip multi-factor authentication, abuse password reset flows, or escape from one user's context into another's. Coverage includes login-flow integrity (OAuth, SAML, MFA), session management across the post-authentication lifecycle (JWT, cookies, fixation, rotation, logout), credential handling (brute force, credential stuffing, weak or default credentials), and access control at two layers: function-level checks, which verify whether a role is allowed to invoke an endpoint at all, and object-level checks (IDOR), which verify that the caller actually owns or is authorized for the specific resource being referenced in path segments, query parameters, request bodies, or cookies.

- **Injection:** This category tests whether untrusted input flows into a backend interpreter or browser context with insufficient escaping. Coverage includes SQL injection across relational and NoSQL data layers, OS command injection, server-side template injection, XML and entity-based attacks (XXE), insecure deserialization in object payloads, and cross-site scripting (XSS) in reflected, stored, server-rendered, and DOM-based forms, exercised through a real browser to confirm execution. Each class is probed on every endpoint that accepts user-controlled data.

- **Server-Side Request Forgery:** This category checks whether the application can be coerced into making outbound requests on the attacker's behalf. Tests exercise URL-handling parameters, redirect targets, file or image fetchers, webhook configurations, and any feature that takes a remote address, looking for paths into the internal network, cloud metadata services, or arbitrary external endpoints.

- **File & Directory Exposure:** Path-handling and file-handling logic is tested for unintended file system access. Coverage includes path traversal and local file inclusion, where relative or absolute paths can reach files outside the intended directory; file upload bypass through executable types, content-type spoofing, or double extensions; and directory access through traversal sequences or exposed backup files that leak sensitive content such as configuration, source code, or credentials.

- **Information Disclosure:** The application is probed for data that should not be visible externally. This covers verbose errors and stack traces that leak framework internals, configuration exposure (configuration files, debug or staging endpoints reachable in production, version banners, tech-stack identifiers), and sensitive data leaks via endpoints that return more fields, records, or internal context than the client should see.

- **Security Hardening:** The application's defensive baseline is checked against current best practice. Tests evaluate security headers (CSP, X-Frame-Options, HSTS, Referrer-Policy, and similar), TLS configuration, cookie attributes (Secure, HttpOnly, SameSite), CORS configuration, and other settings that strengthen downstream defenses. Findings in this category surface as hardening items rather than verified risks, since they raise the security baseline rather than represent a directly exploitable issue on their own.

- **Unpatched Technologies:** Exposed services and unpatched technology components are tracked under this infrastructure verification scope, evaluating public footprint configurations and software daemon parameters against version vulnerability maps.

3.3 Safety and Guardrails

Every action the agent takes is checked before it touches the application. Three independent layers, defense in depth.

- **Tool-call safety check:** Before sensitive tools fire, an LLM safety judge inspects the call and the recent context. Destructive, state-changing, or out-of-scope actions are blocked, and the agent has to find another path.

- **Browser action firewall:** Inside the browser, every click, type, submit, and upload is screened for state-changing or destructive intent before it executes. Read-only navigation passes through.

- **Prompt-injection defense:** A dedicated classifier intercepts tool responses from the target application before the agent sees them. Adversarial text in the application cannot hijack the agent's behavior.

In addition to the three checks above, scope-level guardrails apply across the run: configurable rate limiting, per-tool method allowlists, scoped credentials, and user-defined out-of-scope areas that are passed as strong guidance to the agent.

4. Risk Rating Model

Findings are scored on a five-level scale (**Critical, High, Medium, Low, Informational**) derived from two axes: **impact** (what an attacker gains if the issue is successfully exploited) and **likelihood** (how feasible exploitation is given preconditions such as authentication, user interaction, and scope of the vulnerable population).

Severity	Typical Profile
Critical	Full compromise of the application, tenant takeover, unauthenticated remote code execution, or bulk exfiltration of customer data. Trivial to exploit: no specialized knowledge, user interaction, or privileged position required.
High	Significant privilege escalation, cross-tenant data access, or exposure of credentials/secrets that grant meaningful authority. Exploitation is practical but may require authenticated context or a specific tenant configuration.
Medium	Localized security-control failures: forced state changes against a victim user, cross-tenant information leakage of low-sensitivity data, or unauthenticated disclosure of third-party integration credentials with bounded blast radius. Exploitation typically requires user interaction or an authenticated foothold.
Low	Information disclosure useful only for reconnaissance, hardening gaps, or best-practice deviations with no direct path to compromise.
Informational	Observations, defense-in-depth recommendations, or deviations from guidance that do not represent an exploitable weakness.

5. Detailed Risks

R-01: Unauthenticated SQL Injection via JSON:API Filter Parameter

Category

Injection

Description

Drupal core's JSON:API read endpoints are exposed without authentication. When a collection is queried with a filter, the array keys of an `IN` condition's value set reach the generated SQL statement without being bound as parameters, so a request placing SQL syntax in those keys executes against the backend database. No session or credentials are required.

Severity

● Critical

Impact

An unauthenticated attacker can execute arbitrary SQL against the application database, exposing all tenant data including account records and password hashes. Depending on the database account's privileges, the same access permits modification or deletion of rows, or read and write access to the database host's filesystem.

Affected Area

`example.com/jsonapi/node/article`

Steps to Reproduce

1. Confirm the input reaches the SQL layer. Inject a single quote into the value array key and observe a `500` response whose body contains a database syntax error such as `SQLSTATE[42000]: Syntax error or access violation`:

```
GET
/jsonapi/node/article?filter[f][condition][path]=title&filter[f][condition]
[operator]=IN&filter[f][condition][value][x']=1
```

2. Confirm arbitrary execution with a time-based payload. Inject a conditional delay in the key and observe that the response is held for approximately five seconds, which does not occur for the baseline request:

```
GET
/jsonapi/node/article?filter[f][condition][path]=title&filter[f][condition]
[operator]=IN&filter[f][condition][value][x) AND SLEEP(5)-- -]=1
```

3. Data extraction was confirmed against the `users` table (account name and password hash columns for a single user) using a boolean-based technique. Extracted values are redacted from this report.

Remediation steps

1. Upgrade Drupal core to the current security release for your branch so the JSON:API filter handling is patched.
2. Build all database queries with bound parameters rather than string concatenation, and validate non-parameterisable identifiers such as table or column names against an allowlist. Input filtering and escaping are not a sufficient fix.

References

- https://cheatsheetseries.owasp.org/cheatsheets/SQL_Injection_Prevention_Cheat_Sheet.html
- https://cwe.mitre.org/data/definitions/89.html

R-02: Cross-Tenant Insecure Direct Object Reference (IDOR)

Category

Authorization & Authentication

Description

The document endpoint resolves the object from the identifier in the path and returns or updates it based only on the caller being authenticated, with no check that the caller's tenant owns the object. Document identifiers are sequential integers, so valid identifiers belonging to other tenants are trivially enumerable.

Severity

- High

Impact

A standard authenticated user can read and modify documents belonging to any other tenant, defeating tenant isolation. Iterating the identifier lets an attacker harvest or tamper with records across every tenant on the platform.

Affected Area

`example.com/api/v1/documents/{document_id}`

Steps to Reproduce

1. Authenticate as the Tenant B user, list documents, and note an identifier owned by Tenant B:

```
GET /api/v1/documents
```

The response includes a document with `id` 4011.

2. Authenticate as the Tenant A user. Establish a baseline against a document you own:

```
GET /api/v1/documents/4012
```

Response is `200 OK` with Tenant A's document.

3. As the Tenant A user, request the Tenant B identifier from step 1:

```
GET /api/v1/documents/4011
```

Response is `200 OK` and returns Tenant B's document, confirming cross-tenant read with no ownership relationship.

4. As the Tenant A user, modify the same Tenant B document to confirm write access. Tenant B is a provisioned test tenant, so the change is controlled and reversible:

```
PUT /api/v1/documents/4011

{"title": "modified-by-tenant-A"}
```

Response is `200 OK` and Tenant B's record is updated.

5. Identifiers are sequential. Read access was additionally confirmed against identifiers outside the provisioned tenants; no modification was performed against records belonging to tenants not provisioned for the engagement.

Remediation steps

1. Enforce object-level authorization on every read and write request. Before returning or modifying an object, verify server-side that the authenticated session's tenant owns or is explicitly granted access to the referenced object, and return `404` or `403` otherwise.

2. Replace sequential integer identifiers with non-sequential values such as UUIDs. This removes trivial enumeration but is defense in depth and does not replace the ownership check.

References

- https://owasp.org/API-Security/editions/2023/en/0xa1-broken-object-level-authorization/
- https://cwe.mitre.org/data/definitions/639.html

R-03: Privilege Escalation via Mass Assignment on Account Update

Category

Authorization & Authentication

Description

The account update endpoint binds fields from the request body directly onto the user record without an allowlist. Privilege-controlling attributes such as `role` are writable from the client, although they should be set only by server-side logic.

Severity

- High

Impact

A standard user can elevate their own account to tenant administrator by adding a role field to a routine profile update, gaining full administrative control over their tenant's users and data without any approval step.

Affected Area

`example.com/api/v1/users/{user_id}`

Steps to Reproduce

1. Authenticate as the Tenant A standard user (user ID 105) and confirm the account is non-administrative:

```
GET /api/v1/users/105
```

Response shows `"role": "member"`.

2. Establish a baseline with a routine profile update and confirm it succeeds:

```
PUT /api/v1/users/105

{"displayName": "qa-test"}
```

Response is `200 OK`.

3. Repeat the update including a privilege field that should be server-controlled:

```
PUT /api/v1/users/105

{"role": "tenant_admin"}
```

Response is `200 OK`.

4. Confirm the escalation. Re-fetch the profile and exercise an administrative endpoint with the same session:

```
GET /api/v1/users/105
```

Response now shows `"role": "tenant_admin"`, and administrative endpoints return `200` for the session.

Remediation steps

1. Bind only an explicit allowlist of client-modifiable fields on the update endpoint, and reject or ignore any field outside it, including `role` and other privilege or ownership attributes.

2. Set privilege-controlling fields exclusively through server-side logic guarded by an authorization check, never from the request body.

References

- https://owasp.org/API-Security/editions/2023/en/0xa3-broken-object-property-level-authorization/
- https://cwe.mitre.org/data/definitions/915.html

R-04: Server-Side Request Forgery via Document Import

Category

Server-Side Request Forgery

Description

The document import feature fetches a user-supplied URL server-side without validating the destination. Because the request originates from the application server, it can reach internal addresses that are not exposed to the internet, including the link-local cloud metadata endpoint at `169.254.169.254`.

Severity

High

Impact

An attacker can make the application issue requests to arbitrary internal services from its trusted network position, and the cloud metadata endpoint is reachable. The instance enforces IMDSv2, so IAM credentials could not be retrieved through the GET-only fetcher, and the demonstrated impact is internal service reachability. On an instance running IMDSv1 the same flaw would expose temporary IAM credentials and escalate to full cloud account compromise.

Affected Area

`example.com/api/v1/documents/import`

Steps to Reproduce

1. Authenticate as the Tenant A regular user. From the Documents page, submit an import that points at a listener you control:

```
POST /api/v1/documents/import

{"sourceUrl": "https://<listener-you-control>/probe"}
```

An inbound request arrives at the listener from the application's egress address, confirming the server fetches attacker-supplied URLs with no destination restriction. Record the egress address.

2. Establish reachability. Submit imports for a spread of internal targets and compare the responses the endpoint surfaces:

```
http://169.254.169.254/latest/meta-data/   -> upstream 401, returned in
well under a second

http://10.0.0.1:1/                          -> connection refused

http://192.0.2.1/                           -> timeout
```

The differing outcomes show the server reaches the internal network and that a live service answers on `169.254.169.254:80`, which is distinct from a uniform proxy error.

3. Attribute the responder. The upstream response surfaced for the `169.254.169.254` request carries the header `Server: EC2ws` and the IMDSv2 token-required body, identifying the responder as the EC2 instance metadata service rather than an egress proxy.

Testing stopped here. Credential retrieval requires an IMDSv2 token obtained via `PUT /latest/api/token`, and the fetcher issues `GET` only, so no token could be obtained and no credentials were retrieved. Probing was limited to confirming reachability; no discovered internal service was interacted with further.

Remediation steps

1. Validate the resolved destination IP rather than the hostname, and reject private, loopback, link-local, and other reserved ranges, including `169.254.0.0/16` and the IPv6 equivalents. Pin the resolved address for the connection and re-check it on every redirect hop to prevent DNS rebinding and redirect-based bypass. Restrict schemes to `http` and `https`.

2. Route outbound fetches through an isolated egress path, such as a forward proxy or a network segment with no route to the metadata service or internal subnets, so that a bypass of the address check still cannot reach internal targets. Return a generic failure on fetch errors rather than the upstream status or body.

3. Enforce IMDSv2 and set the instance metadata hop limit to 1.

References

- https://cheatsheetseries.owasp.org/cheatsheets/Server_Side_Request_Forgery_Prevention_Cheat_Sheet.html
- https://cwe.mitre.org/data/definitions/918.html

R-05: Broken Function-Level Authorization on Admin User Management

Category

Authorization & Authentication

Description

The administrative user-management endpoints under `/api/v1/admin/` perform no role check. Any authenticated session can call them regardless of role, so a standard user reaches functions intended for tenant administrators.

Severity

● Medium

Impact

A standard user can retrieve the full user roster for their tenant, including roles and email addresses, and can create new user invitations. Both are administrative functions. The actions are confined to the caller's own tenant and do not by themselves grant elevated privileges, so the blast radius is bounded to that tenant.

Affected Area

`example.com/api/v1/admin/users`

Steps to Reproduce

1. Authenticate as the Tenant A regular user:

2. Call the admin listing endpoint with the standard-user session:

```
GET /api/v1/admin/users
```

Response is `200 OK` and returns every user in the tenant with roles and email addresses, with no administrative role required.

3. Confirm a state-changing admin action is equally unprotected by creating an invitation:

```
POST /api/v1/admin/users

{"email": "qa+invite@tenant-a.example", "role": "member"}
```

Response is `201 Created` and the invitation is issued, confirming a non-admin can invoke administrative user management.

Remediation steps

1. Enforce a server-side role check on every endpoint in the administrative namespace, verifying the session holds the required administrative role for the tenant before the handler runs. Apply it through shared middleware so new endpoints inherit the control rather than each implementing its own.

2. Deny by default. An endpoint with no explicit authorization rule should reject the request, not allow it.

References

- https://owasp.org/API-Security/editions/2023/en/0xa5-broken-function-level-authorization/
- https://cwe.mitre.org/data/definitions/862.html

R-06: Stored Cross-Site Scripting in Document Title

Category

Injection

Description

The document title is stored as submitted and rendered into the documents list view without contextual output encoding. Markup placed in a title is returned unescaped and executes in the browser of any user in the tenant who opens the list.

Severity

● Medium

Impact

Script supplied in a document title runs in the authenticated session of any tenant user who views the list, and can perform actions that user is permitted to perform through the application, such as reading or modifying documents in the tenant. Session cookies are set `HttpOnly` and `Secure`, so the script cannot read the session token and the issue does not yield a one-click account takeover through cookie theft. Impact is bound to actions taken within the victim's active session and requires the victim to view the malicious document.

Affected Area

`example.com/app/documents`

Steps to Reproduce

1. Authenticate as any regular user and go to the Documents page

2. Create a new document and set its title to the following payload, then save:

```
<img src=x onerror=alert(document.domain)>
```

3. Sign in as a second Tenant A user and open the documents list. The title renders as markup and the `onerror` handler runs in the viewer's browser, shown by an alert displaying the application origin. This confirms the stored title executes as script.

Remediation steps

1. Apply contextual output encoding when rendering user-controlled values such as the title, encoding for the HTML context at the point of output rather than sanitizing on input.

2. Add a Content-Security-Policy that disallows inline script as defense in depth, so an encoding gap does not immediately result in script execution

References

- https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html
- https://cwe.mitre.org/data/definitions/79.html

R-07: Verbose Error and Stack Trace Disclosure

Category

Information Disclosure

Description

The application returns unhandled exceptions to the client with verbose error output enabled. A request that triggers an exception receives a full stack trace in the response body rather than a generic error.

Severity

- Low

Impact

The trace discloses the application's internal layout, including absolute filesystem paths, module and class names, and the Drupal and PHP versions. This lowers the cost of further attacks by identifying the framework, version, and code structure to target, but provides no access or direct path to compromise on its own.

Affected Area

`example.com/api/v1/documents`

Steps to Reproduce

1. Send a request that supplies a parameter of an unexpected type to trigger an unhandled exception. For example, request the documents collection with a non-numeric page limit:

```
GET /api/v1/documents?page[limit]=abc
```

2. Observe an `HTTP 500` response whose body contains a full PHP backtrace, including absolute server paths such as `/var/www/html/modules/custom/app_core/src/Controller/DocumentController.php`, the chain of class and method calls, and the Drupal and PHP version strings.

Remediation steps

1. Disable verbose error output in production. Set the site error-handling level so messages and backtraces are not rendered to the client, and set PHP

`display_errors = Off`. Error detail should be written to server-side logs only.

2. Map unhandled exceptions to a generic error response with no internal detail, so a triggered error does not reveal framework or code structure.

References

- https://cwe.mitre.org/data/definitions/209.html
- https://owasp.org/Top10/A05_2021-Security_Misconfiguration/

R-08: Missing Security Headers and Insecure Cookie Attributes

Category

Security Hardening

Description

Application responses do not set several recommended security headers: `Content-Security-Policy`, `Strict-Transport-Security`, `X-Frame-Options` (or an equivalent CSP `frame-ancestors`), `X-Content-Type-Options`, and `Referrer-Policy`. The session cookie is set with `HttpOnly` and `Secure` but without a `SameSite` attribute.

Severity

- Informational

Impact

The missing headers leave the application without baseline protection against clickjacking, MIME-type sniffing, and protocol downgrade, and the absent `Content-Security-Policy` removes a control that would have limited the stored XSS in R-06. The session cookie's missing `SameSite` attribute allows it to be sent with some cross-site requests. None of these are directly exploitable on their own. They raise the security baseline and reduce the impact of other issues.

Affected Area

`example.com`

Steps to Reproduce

1. Request an application page and capture the full set of response headers.

2. Confirm the response does not include `Content-Security-Policy`, `Strict-Transport-Security`, `X-Frame-Options`, `X-Content-Type-Options`, or `Referrer-Policy`.

3. Authenticate and inspect the `Set-Cookie` header for the session cookie. It includes `HttpOnly` and `Secure` but no `SameSite` attribute.

Remediation steps

1. Set the security headers across all responses: `Content-Security-Policy` restricting script sources and disallowing inline script, `Strict-Transport-Security` with a long max-age and `includeSubDomains`, `X-Content-Type-Options: nosniff`, a `Referrer-Policy`, and a frame-ancestors restriction through CSP or `X-Frame-Options`.

2. Add `SameSite` (`Lax` or `Strict`, per the application's cross-site requirements) to the session cookie while keeping `HttpOnly` and `Secure`.

References

- https://owasp.org/www-project-secure-headers/
- https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html

Appendix A: Agentic Testing

This appendix describes the agentic execution model that produced the findings in §5 and the touchpoints where a human reviewer is involved. It is included for transparency on how the testing was performed; the body of the report is intended to stand on its own without reference to this appendix.

Execution Model

The pentest is run by a planner agent that orchestrates a population of specialist agents. The planner never performs scanning itself; it owns scope, prioritization, and follow-up decisions. Specialists are narrow by design: each is an expert in one vulnerability class, with its own toolset, payload library, and validation logic. This separation gives failure isolation, a bounded blast radius per agent, and full attribution from action to finding.

Per-Task Behavior

Each task is executed by a dedicated agent that issues authenticated and unauthenticated requests, observes responses, adapts its payload strategy to the observed behavior, and, when it suspects a weakness, attempts to confirm it through a minimal reproduction. Agents are instructed to operate within the scope and safety limits; destructive routes are explicitly excluded from task generation.

Human-in-the-Loop

Humans collaborate with AI agents at various phases during a test, helping to escalate findings, and perform follow-up investigations. A human reviewer validates each candidate finding before it is included in the report: the reproduction is re-run from a clean context, severity is reviewed against the impact / likelihood rubric, and the risk narrative, evidence, and remediation guidance are enriched if needed. No risk was reported without this review step.

End of report.