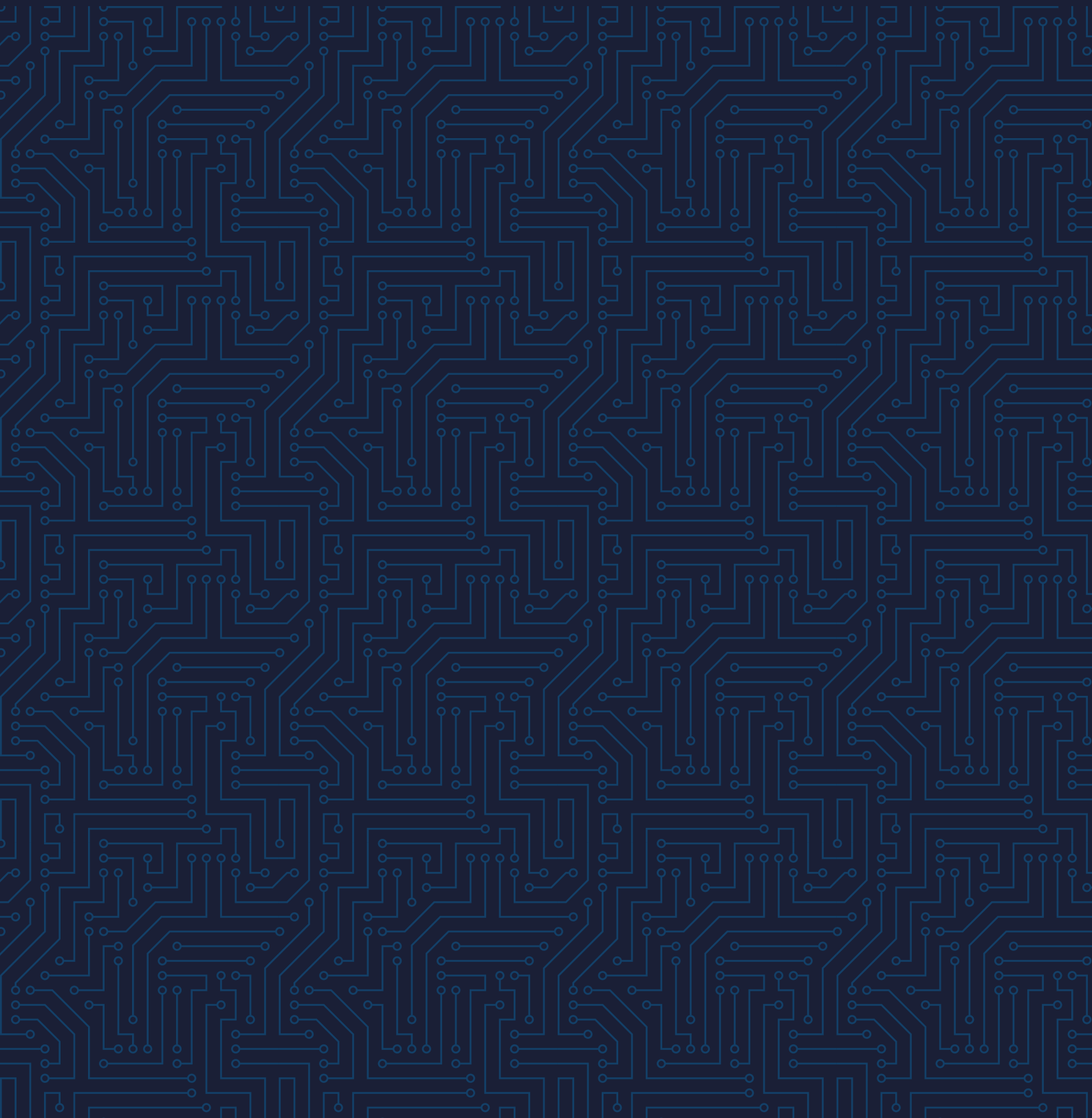
aisafe.io

adragos's Organization
Security Team

February 25, 2026

vinext

Source Code Audit



1 - Executive Summary

AISafe Labs conducted a Source Code Audit for adragos's Organization from February 24 to February 25, 2026.

During this engagement, AISafe performed an in-depth source code review of vinext, examining the codebase for security vulnerabilities, insecure coding patterns, design flaws, and general weaknesses in security posture. This static analysis approach allowed for comprehensive coverage of the application logic without runtime testing.

The security assessment was conducted exclusively by AISafe's AI-powered penetration testing platform.

AISafe identified a total of 22 issues during the assessment. Each finding contains a detailed technical analysis, proof-of-concept demonstrations where applicable, and specific remediation steps addressing the underlying cause.

Severity classifications account for real-world exploitability and potential business impact, considering required attacker capabilities and realistic attack scenarios. The report concludes with our assessment of the overall security posture and recommended actions to enhance resilience, minimize exposure, and implement layered defenses.

1.1 Goals of the Assessment

The assessment objectives were automatically determined by our AI platform based on the target application's architecture, with any client-provided notes taken into consideration.

The source code analysis focused on addressing the following key security questions:

- Are there insecure coding patterns that could lead to vulnerabilities?
- Is authentication logic implemented correctly without bypass vectors?
- Are authorization checks consistently applied across all code paths?
- Is input validation and output encoding properly implemented?
- Are sensitive data (credentials, tokens) handled securely?
- Are there hardcoded secrets, debug code, or commented credentials?
- Does the code follow secure design principles and best practices?

1.2 Limitations

The following areas fell outside the scope of this assessment:

- Runtime behavior and deployed environment configuration
- Third-party dependencies and external libraries (unless explicitly included)
- Infrastructure, network, and cloud security
- Physical security controls
- Social engineering attacks
- Performance and availability testing

2 - Scope & Methodology

2.1 Source Code Delivery

The application source code was provided via a private Git repository at https://github.com/cloudflare/vinext. The assessment was conducted against the `main` branch.

2.2 Testing Scope

This was a source code audit. No web application targets were tested as part of this assessment. The analysis focused exclusively on static code review without runtime testing.

2.3 Testing Methodology

The source code audit was conducted following industry-standard secure code review frameworks:

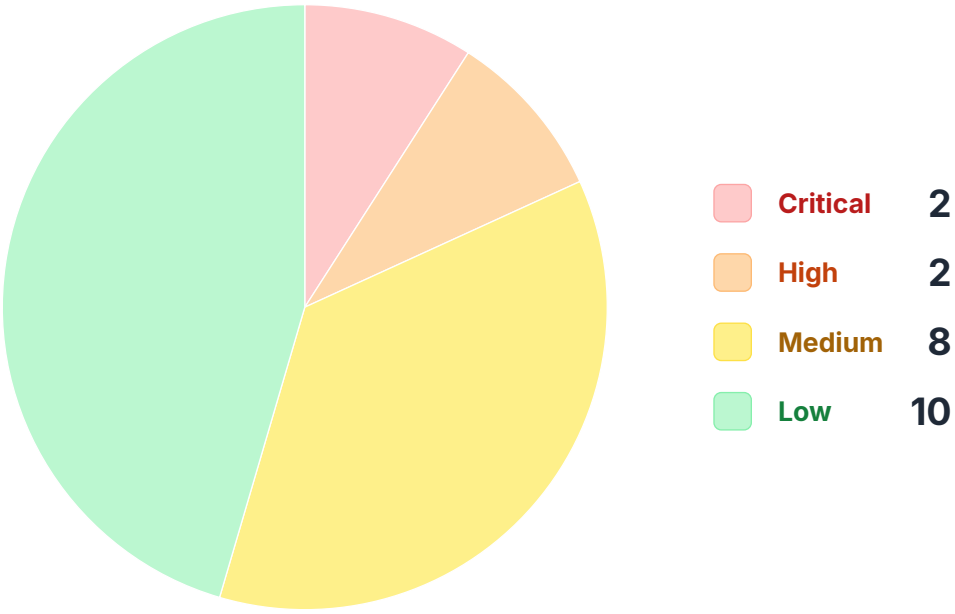
OWASP Code Review Guide v2.0	OWASP ASVS v4.0
CWE/SANS Top 25	SEI CERT Coding Standards

2.4 Assessment Timeline

Start Date	February 24, 2026 at 10:22 PM UTC
End Date	February 25, 2026 at 02:34 AM UTC
Duration	4h 12m

3 - Findings

A total of 22 vulnerabilities were identified during the security assessment. The chart below summarizes the findings by severity level.



3.1 Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

Risk Level	Impact
Critical	Issues requiring immediate attention. These vulnerabilities pose the highest security risk and can lead to financial loss, service unavailability, or large-scale account compromise.
High	Issues that should be addressed promptly. These vulnerabilities warrant serious consideration and can result in account compromise with user interaction, or sensitive data exposure.
Medium	Issues to be addressed in the standard release cycle. These vulnerabilities typically affect individual users, require user interaction to exploit, or expose moderately sensitive information.
Low	Issues with minimal immediate risk. These vulnerabilities pose limited danger to confidentiality, integrity, and availability, and should be considered for remediation over time.
Info	Observations and best practice recommendations. These are valid findings that are either out of scope, have no direct security impact, or represent opportunities for security hardening.

4 - Vulnerability Findings

Here, we present a technical analysis of the vulnerabilities identified during our assessment. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

ID	Severity	Status	Description
AIS-ASO-VIN-001	Critical	Open	The fetch cache shim constructs cache keys using only the HTTP method and the request URL, deliberately omitting per-user headers such as Authorization and Cookie. When a Server Component issues an authenticated fetch() call with caching options (next.revalidate or cache: 'force-cache'), the first user's authenticated response — including their private data — is stored under a URL-only key and subsequently served to every other user who requests the same URL.
AIS-ASO-VIN-002	Critical	Open	The runWithHeadersContext() function in packages/vinext/src/shims/headers.ts is responsible for isolating per-request header and cookie state in Cloudflare Workers deployments. It relies on AsyncLocalStorage.enterWith() for context propagation; however, Cloudflare Workers explicitly does not support enterWith(). When the call fails, the function falls back to mutating a shared module-level variable, _fallbackState, and invokes the request handler function fn() without any als.run() wrapper. This makes the context visible to all concurrently executing requests within the same Worker isolate.
AIS-ASO-VIN-003	Medium	Open	The same broken AsyncLocalStorage.enterWith() pattern that affects the headers shim is also present in the navigation and router state modules. Both setServerContext() in packages/vinext/src/shims/navigation-state.ts and setSSRContext() in packages/vinext/src/shims/router-state.ts call _enterWith(), which fails silently on Cloudflare Workers, then fall back to mutating a shared module-

			level _fallbackState object with the current request's pathname, search parameters, and route parameters.
AIS-ASO-VIN-005	Low	Open	Three example Cloudflare Worker entry points expose raw internal error messages to HTTP clients in their top-level catch blocks. Because these files are presented as reference implementations intended to be copied directly by users, they can be deployed to production with insecure error handling.
AIS-ASO-VIN-007	Medium	Open	The catch-all redirect matching logic in packages/vinext/src/config/config-matchers.ts decodes percent-encoded characters from request path segments before substituting them into redirect destination templates. An attacker can encode a forward slash as %2F in the URL to bypass the protocol-relative URL guard that only inspects the raw, undecoded pathname.
AIS-ASO-VIN-008	Medium	Open	The default middleware matcher in vinext's Pages Router unconditionally skips middleware execution for any pathname that contains a dot character. This behaviour deviates silently from Next.js's actual default, which runs middleware against all paths. The condition is present in both the Node.js server and the inlined Cloudflare Worker entry.
AIS-ASO-VIN-009	Low	Open	The isSafeRegex() function in packages/vinext/src/config/config-matchers.ts is intended to prevent Regular Expression Denial of Service by rejecting dangerous patterns before they are compiled via new RegExp(). The function correctly identifies nested quantifier patterns such as (a+)+ or (.) by tracking quantifier depth, but it performs no analysis of alternation branches (|) inside repeated groups.
AIS-ASO-VIN-010	Medium	Open	The Pages Router production Cloudflare Worker entry generated by vinext imports next/head for its resetSSRHead and getSSRHeadHTML helpers but does not import vinext/head-state, the module that registers AsyncLocalStorage-backed per-request isolation

			for <Head> state. Without that import, head.ts falls back to a raw module-level _ssrHeadElements array shared across all concurrently executing requests in the same Worker isolate.
AIS-ASO-VIN-011	Medium	Open	The __validateCsrfOrigin() function used to protect Server Actions reads the x-forwarded-host header from the incoming request and treats it as a trusted source of the application's host identity, without first validating that the header originates from a trusted proxy. In Cloudflare Workers deployments, Cloudflare does not set or overwrite x-forwarded-host, meaning the value is entirely attacker-controlled.
AIS-ASO-VIN-012	Low	Open	The __isOriginAllowed() function that evaluates serverActionsAllowedOrigins wildcard patterns includes an unintended check that permits requests from the apex domain when a subdomain wildcard such as *.example.com is configured. The check uses pattern.slice(2) to extract the bare domain from the wildcard string and compares it directly against the request origin:
AIS-ASO-VIN-014	Medium	Open	The draftMode().enable() implementation in packages/vinext/src/shims/headers.ts constructs a Set-Cookie header that includes HttpOnly and SameSite=Lax attributes but omits the Secure flag. The cookie carries the raw draft secret value derived from the __VINEXT_DRAFT_SECRET environment variable.
AIS-ASO-VIN-016	High	Risk Accepted	The KVCacheHandler in packages/vinext/src/cloudflare/kv-cache-handler.ts stores and retrieves cache entries as plain JSON from Cloudflare KV without any cryptographic integrity protection. An attacker with write access to the KV namespace — obtained via a compromised CLOUDFLARE_API_TOKEN, a malicious build dependency, or a supply chain attack — can write crafted entries containing arbitrary HTML, RSC data, or API response bodies.

AIS-ASO-VIN-017	High	Open	During client-side navigation, packages/vinext/src/shims/router.ts fetches the target page's HTML from the server, extracts a JavaScript module URL from the __ field of the embedded JSON, and passes it directly to import() without any path validation.
AIS-ASO-VIN-019	Medium	Open	In the Pages Router, vinext's default middleware matcher unconditionally excludes all paths starting with /api when no explicit config.matcher is configured. This deviates silently from Next.js's documented behaviour, where middleware with no matcher runs on every path including /api/*.
AIS-ASO-VIN-022	Medium	Open	The resolveHost() function in packages/vinext/src/server/prod-server.ts validates the x-forwarded-host header against the VINEXT_TRUSTED_HOSTS allowlist but returns req.headers.host — the plain HTTP Host header — without any corresponding validation when x-forwarded-host is absent. In HTTP/1.1, the client sets the Host header freely, making it attacker-controlled.

5 - Miscellaneous Observations

The following observations are informational findings, best practices, and recommendations that do not represent immediate security vulnerabilities but may be useful for improving the overall security posture.

ID	Severity	Status	Description
AIS-ASO-VIN-004	Low	Open	The draft mode secret comparison in packages/vinext/src/shims/headers.ts uses the standard JavaScript === operator, which short-circuits on the first differing character and is therefore not constant-time. An attacker who can measure HTTP response latency across many repeated requests could exploit this timing side-channel to enumerate the __VINEXT_DRAFT_SECRET value character-by-character.
AIS-ASO-VIN-006	Low	Open	The development-mode middleware runner in packages/vinext/src/server/middleware.ts concatenates the raw exception message into the body of HTTP 500 responses returned to clients. Any exception thrown from a user's middleware function is caught and its .message property is embedded directly in the response.
AIS-ASO-VIN-013	Low	Open	The startLocalServer() function in packages/vinext/src/cloudflare/tpr.ts constructs a JavaScript code string that is evaluated by a spawned Node.js child process via the -e flag. Filesystem paths are embedded into the string with only backslash escaping applied, leaving double-quote characters, newlines, and other JavaScript metacharacters unhandled.
AIS-ASO-VIN-015	Low	Open	The CI failure notification script at .github/scripts/send-email.mjs sends POST requests to https://api.example.com/send-email — a non-functional placeholder URL accompanied by an inline TODO comment acknowledging that it must be replaced. Every invocation of this script will receive a connection error or 404 response, log the failure to stderr, and exit with a non-zero code.
AIS-ASO-VIN-018	Low	Open	When a Pages Router API route module exists but does not export a default handler function, the Vite development server responds with an HTTP 500 message that includes the full absolute filesystem path to the module file.

AIS-ASO-VIN-020	Low	Open	The sitemapToXml() function in packages/vinext/src/server/metadata-routes.ts applies escapeXml() to all string fields in a sitemap entry (url, lastModified, changeFrequency, images) but directly interpolates the priority field into the XML output without any escaping or type validation.
AIS-ASO-VIN-021	Low	Open	The robotsToText() function in packages/vinext/src/server/metadata-routes.ts constructs robots.txt output by directly interpolating string fields — including userAgent, allow, disallow, sitemap, and host — without stripping embedded newline characters. Because robots.txt uses newlines as directive and record separators per RFC 9309, a field value containing ` or ` will inject additional directives into the generated output.

Fetch Cache Cross-User Data Leakage Critical AIS-ASO-VIN-001

Description

The fetch cache shim constructs cache keys using only the HTTP method and the request URL, deliberately omitting per-user headers such as `Authorization` and `Cookie`. When a Server Component issues an authenticated `fetch()` call with caching options (`next.revalidate` or `cache: 'force-cache'`), the first user's authenticated response — including their private data — is stored under a URL-only key and subsequently served to every other user who requests the same URL.

The cache key is built in `packages/vinext/src/shims/fetch-cache.ts` as follows:

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
const parts = [`fetch:${method}:${url}`];  // Authorization/Cookie NOT included
```

The original fetch call is executed with the full request init — including the user's `Authorization` header — and the resulting authenticated response is captured along with all headers:

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
const response = await originalFetch(input, cleanInit); // includes user's
Authorization header
```

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
cloned.headers.forEach((v, k) => { headers[k] = v; }); // ALL headers cached
including Set-Cookie
```

User A's response body and headers are then stored in the shared cache under the URL-only key:

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
handler.set(cacheKey, cacheValue, { fetchCache: true, ... }); // User A's data
stored under shared URL key
```

When User B issues the same request, the identical cache key is resolved and the stored entry is returned:

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
const cached = await handler.get(cacheKey, { kind: 'FETCH', tags }); // User B
hits User A's cache entry
```

packages/vinext/src/shims/fetch-cache.ts TypeScript

```
return new Response(cachedData.body, { status: cachedData.status ?? 200, headers:
cachedData.headers }); // User A's data returned to User B
```

This affects both the in-process `MemoryCacheHandler` (sharing within an isolate) and the `KVCacheHandler`, which persists the poisoned entry globally, causing every subsequent request — including unauthenticated ones — to receive User A's private response until the TTL expires or a tag invalidation occurs.

Impact

Any authenticated data fetched via Server Components with caching enabled — including user profiles, account details, and private API payloads — can be served verbatim to other users. With the `KVCacheHandler` backend, the first user's response is stored globally and replayed to

all subsequent visitors, including unauthenticated ones, until the cache entry is evicted. This constitutes a severe cross-user data leakage that could violate GDPR and other data-protection regulations, expose sensitive personal and financial information, and enable account takeover if session tokens or secrets are included in the cached response.

Remediation

To remediate this issue, AISafe recommends to include a keyed hash of the `Authorization` and `Cookie` request headers in the cache key whenever those headers are present, and to emit an automatic warning — or default to `cache: 'no-store'` — when an authenticated fetch is detected alongside caching options.

Status

Open No action has been taken to address this finding yet.

Session Cookie Leakage via Unsafe ALS Fa... Critical AIS-ASO-VIN-002

Description

The `runWithHeadersContext()` function in `packages/vinext/src/shims/headers.ts` is responsible for isolating per-request header and cookie state in Cloudflare Workers deployments. It relies on `AsyncLocalStorage.enterWith()` for context propagation; however, Cloudflare Workers explicitly does not support `enterWith()`. When the call fails, the function falls back to mutating a shared module-level variable, `_fallbackState`, and invokes the request handler function `fn()` without any `als.run()` wrapper. This makes the context visible to all concurrently executing requests within the same Worker isolate.

packages/vinext/src/shims/headers.ts — TypeScript

```
_fallbackState.headersContext = ctx;  // User A's cookies written to global state
```

While User A's `fn()` is suspended at an `await` point, a second request from User B arrives and overwrites the same `_fallbackState`:

packages/vinext/src/shims/headers.ts — TypeScript

```
_fallbackState.headersContext = ctx;  // User B's cookies OVERWRITE User A's
context
```

When User A's execution resumes and calls `cookies()`, the `_getState()` helper attempts to retrieve the `AsyncLocalStorage` store. Because `enterWith()` failed and no `als.run()` scope was established, `getStore()` returns `null` and the function falls back to `_fallbackState`, which now holds User B's context:

packages/vinext/src/shims/headers.ts — TypeScript

```
function _getState(): VinextHeadersShimState {
  const state = _als.getStore();
  return state ?? _fallbackState;  // getStore() returns null (enterWith failed)
→ returns User B's fallback
```

As a result, the `cookies()` shim returns the wrong user's session cookies to User A's Server Component:

packages/vinext/src/shims/headers.ts — TypeScript

```
export async function cookies(): Promise<RequestCookies> {
  const state = _getState();  // returns User B's state while in User A's
request!
  return new RequestCookies(state.headersContext.cookies);
```

The code itself acknowledges that the fallback is "not concurrency-safe but works for dev." Because Cloudflare Workers isolates process multiple requests concurrently via the event loop, this race condition arises naturally under any non-trivial traffic load without requiring deliberate attacker manipulation.

Impact

Under concurrent traffic, session cookies and authorization headers from one user's request can be inadvertently returned to a different user's Server Component render. This enables effective session hijacking and account impersonation without requiring any special attacker capability beyond sending ordinary requests to the application during normal operation. Any route that calls `cookies()` or `headers()` in a Server Component — the standard App Router pattern — is affected.

The vulnerability constitutes an unauthorized disclosure of personal data and session credentials with significant regulatory implications under GDPR and similar data-protection frameworks.

Proof of Concept

exploit.py — Python

```python
import requests
import threading
import time

TARGET = "http://your-app.workers.dev"

# This PoC demonstrates the race condition conceptually.
# In practice, the race occurs naturally under concurrent load.

def slow_request():
    """Request that triggers slow async work (awaiting), yielding event loop."""
    # The attacker's request - will be in the async fn() when victim's request arrives
    resp = requests.get(f"{TARGET}/slow-page",
                        headers={"Cookie": "session=attacker_session"})
    print(f"Attacker response (may contain victim cookies in rendered content): {resp.text[:500]}")

def victim_request():
    """Victim's authenticated request sent during attacker's async gap."""
    time.sleep(0.05)  # Small delay to hit the async gap
    resp = requests.get(f"{TARGET}/dashboard",
                        headers={"Cookie": "session=victim_secret_session_token"})
    print(f"Victim response: {resp.status_code}")

# Send both concurrently
t1 = threading.Thread(target=slow_request)
t2 = threading.Thread(target=victim_request)
t1.start()
t2.start()
t1.join()
t2.join()
# The attacker's Server Component may receive the victim's session cookie
# when cookies() is called during the render phase after the victim's
# request overwrites _fallbackState.headersContext
```

Remediation

To remediate this issue, AISafe recommends to replace the `enterWith()` / `_fallbackState` pattern in `runWithHeadersContext()` with `als.run(state, fn)`, so that each request's context is properly isolated within an `AsyncLocalStorage` scope, and to address the RSC streaming case by using `als.bind()` or `als.snapshot()` to preserve the correct context across the stream lifetime.

Status

Open No action has been taken to address this finding yet.

Cross-Request URL Leakage via ALS Fallba... AIS-ASO-VIN-003

Medium

Description

The same broken `AsyncLocalStorage.enterWith()` pattern that affects the headers shim is also present in the navigation and router state modules. Both `setServerContext()` in `packages/vinext/src/shims/navigation-state.ts` and `setSSRContext()` in `packages/vinext/src/shims/router-state.ts` call `_enterWith()`, which fails silently on Cloudflare Workers, then fall back to mutating a shared module-level `_fallbackState` object with the current request's pathname, search parameters, and route parameters.

packages/vinext/src/shims/navigation-state.ts — TypeScript

```
_fallbackState.serverContext = ctx;  // User A's {pathname, searchParams, params}
```

While Request A is suspended at an `await` boundary, Request B arrives and overwrites the same `_fallbackState` with its own navigation context:

packages/vinext/src/shims/navigation-state.ts — TypeScript

```
setNavigationContext({ pathname: cleanPathname, searchParams: url.searchParams,
params: {} });
```

When Request A's Server Component subsequently calls `usePathname()`, `useSearchParams()`, or `useParams()`, the `_getState()` helper returns `null` from `getStore()` (because no `als.run()` scope exists) and falls back to the shared state — which now holds Request B's URL data:

packages/vinext/src/shims/navigation-state.ts — TypeScript

```
function _getState(): NavigationState {
  return _als.getStore() ?? _fallbackState;
```

The same issue is replicated in the Pages Router shim:

packages/vinext/src/shims/router-state.ts — TypeScript

```
setSSRContext(ctx: SSRContext | null): void {
  if (ctx !== null) {
    _enterWith({ ssrContext: ctx });  // FAILS on CF Workers
    _fallbackState.ssrContext = ctx;  // Shared global overwritten
    return;
  }
```

The Pages Router `SSRContext` additionally contains the locale, query object, and `asPath`, all of which can be cross-contaminated under concurrent load.

Impact

An attacker can infer another user's current URL path and query parameters by submitting requests that race with the victim's. For applications that encode sensitive identifiers — such as user IDs, resource tokens, or internal reference numbers — in URL paths or query strings, this constitutes a meaningful information disclosure. The Pages Router leak additionally exposes locale and full query objects, potentially revealing session-linked state.

Remediation

To remediate this issue, AISafe recommends to replace `_enterWith()` calls with `als.run()` in both `navigation-state.ts` and `router-state.ts`, eliminating direct `_fallbackState` mutations from the context-setting functions, consistent with the fix required in `headers.ts`.

Status

Open No action has been taken to address this finding yet.

Verbose Error Exposure in Worker Example... Low AIS-ASO-VIN-005

Description

Three example Cloudflare Worker entry points expose raw internal error messages to HTTP clients in their top-level `catch` blocks. Because these files are presented as reference implementations intended to be copied directly by users, they can be deployed to production with insecure error handling.

examples/realworld-api-rest/worker/index.ts TypeScript

```
} catch (error) {
      console.error('[vinext] Worker error:', error);
      return new Response(
        `Internal Server Error: ${error instanceof Error ? error.message :
String(error)}`,
        { status: 500 },
      );
    }
```

The same pattern is repeated in two additional examples:

examples/pages-router-cloudflare/worker/index.ts TypeScript

```
} catch (error) {
      console.error('[vinext] Worker error:', error);
      return new Response(
        `Internal Server Error: ${error instanceof Error ? error.message :
String(error)}`,
        { status: 500 },
      );
    }
```

examples/hackernews/worker/index.ts TypeScript

```
} catch (error) {
      console.error('[vinext] Worker error:', error);
      return new Response(
        `Internal Server Error: ${error instanceof Error ? error.message :
String(error)}`,
        { status: 500 },
      );
    }
```

Any internal exception propagated from `renderPage` or `handleApiRoute` — including KV operation failures, unresolvable module references, or upstream `fetch` errors — will have its message string embedded in the HTTP 500 response body and returned to the client. The auto-generated worker template produced by `vinext deploy` correctly returns a generic `'Internal Server Error'` string without any detail, making the discrepancy between the template and the examples a likely source of insecure deployments.

Impact

Clients who trigger an unhandled server error receive internal details such as KV namespace names, module paths, environment variable identifiers, or upstream service URLs embedded in the response body. This information reduces the reconnaissance effort required to mount further attacks against the application's infrastructure.

Remediation

To remediate this issue, AISafe recommends to update all three example worker entry points to return a generic `'Internal Server Error'` response string without interpolating `error.message`, consistent with the output produced by the `vinext deploy` auto-generated template.

Status

Open No action has been taken to address this finding yet.

Open Redirect via %2F Bypass in Catch-Al... Medium AIS-ASO-VIN-007

Description

The catch-all redirect matching logic in `packages/vinext/src/config/config-matchers.ts` decodes percent-encoded characters from request path segments before substituting them into redirect destination templates. An attacker can encode a forward slash as `%2F` in the URL to bypass the protocol-relative URL guard that only inspects the raw, undecoded pathname.

The raw pathname guard fires on the undecoded input and is therefore not triggered by `/old/%2Fevil.com`:

packages/vinext/src/server/prod-server.ts TypeScript

```
if (pathname.startsWith("//")) { res.writeHead(404); res.end("404 Not Found");
return; }
```

The catch-all segment is extracted from the raw path, so `rest` still contains the percent-encoded form:

packages/vinext/src/config/config-matchers.ts TypeScript

```
    const rest = pathname.slice(prefix.replace(/\/$/, "").length);
```

The leading slash is stripped, leaving `%2Fevil.com` in `restValue`:

packages/vinext/src/config/config-matchers.ts TypeScript

```
    let restValue = rest.startsWith("/") ? rest.slice(1) : rest;
```

`decodeURIComponent()` is then applied, converting `%2F` to a literal `/` and producing `/evil.com`:

packages/vinext/src/config/config-matchers.ts TypeScript

```
    try { restValue = decodeURIComponent(restValue); } catch { /* malformed
percent-encoding */ }
```

When substituted into a destination template such as `/:path*`, the result is `//evil.com`:

packages/vinext/src/config/config-matchers.ts TypeScript

```
        dest = dest.replace(`:${key}*`, value);
```

This protocol-relative URL is written directly to the `Location` response header, causing browsers to redirect to `http://evil.com`:

packages/vinext/src/server/prod-server.ts TypeScript

```
res.writeHead(redirect.permanent ? 308 : 307, { Location: dest });
```

The same vulnerable code path is present across the production server, the Vite development server, and the Pages Router Cloudflare Worker entry point.

Impact

An attacker can construct a URL on the legitimate domain — such as `https://myapp.com/old/%2Fevil.com` — that silently redirects visitors to an attacker-controlled site. Because the initial link appears to originate from a trusted domain, it can bypass browser safe-browsing warnings and be used in phishing campaigns to harvest user credentials or distribute malware.

Proof of Concept

exploit.py — Python

```python
import requests

TARGET = "http://localhost:3000"  # Edit to point at your vinext server

# Assume next.config.js has:
# redirects: [{ source: '/old/:path*', destination: '/:path*', permanent: false }]

print("Step 1: Verifying normal redirect works...")
r = requests.get(f"{TARGET}/old/about", allow_redirects=False)
print(f"  Status: {r.status_code}, Location: {r.headers.get('location')}")
# Expected: 307 Location: /about

print("
Step 2: Testing %2F-encoded slash bypass...")
r = requests.get(f"{TARGET}/old/%2Fevil.com", allow_redirects=False)
print(f"  Status: {r.status_code}, Location: {r.headers.get('location')}")
# Expected: 307 Location: //evil.com (open redirect!)
if r.status_code in (307, 308) and r.headers.get('location', '').startswith('//'):
    print("  [+] CONFIRMED: Open redirect via protocol-relative URL!")
    print(f"  Browser would redirect to: http:{r.headers['location']}")
else:
    print("  [-] Not vulnerable or different configuration")
```

Remediation

To remediate this issue, AISafe recommends to validate the fully substituted redirect destination after parameter interpolation, rejecting or normalizing any value that starts with `//`, and to avoid applying `decodeURIComponent()` to catch-all parameter values that will be used in URL construction.

Status

Open No action has been taken to address this finding yet.

Auth Bypass via Dot-Path in Pages Router Medium AIS-ASO-VIN-008

Description

The default middleware matcher in vinext's Pages Router unconditionally skips middleware execution for any pathname that contains a dot character. This behaviour deviates silently from Next.js's actual default, which runs middleware against all paths. The condition is present in both the Node.js server and the inlined Cloudflare Worker entry.

packages/vinext/src/server/middleware.ts TypeScript

```
  if (!matcher) {
    // Default: match all paths except static files, _next, and favicon
    return (
      !pathname.startsWith("/_next") &&
      !pathname.startsWith("/api") &&
      !pathname.includes(".") &&
      pathname !== "/favicon.ico"
    );
  }
```

The same exclusion is inlined into the generated Pages Router Cloudflare Worker bundle:

packages/vinext/src/index.ts TypeScript

```
    return !pathname.startsWith("/_next") && !pathname.startsWith("/api") && !
pathname.includes(".") && pathname !== "/favicon.ico";
```

The attacker's URL — for example `/protected.json` — does not start with `/_next`, but `pathname.includes(".")` evaluates to `true`, causing `matchesMiddleware` to return `false`. Middleware is skipped entirely:

packages/vinext/src/server/middleware.ts TypeScript

```
      !pathname.includes(".") &&
```

Execution proceeds directly to page rendering without any middleware checks:

packages/vinext/src/server/prod-server.ts TypeScript

```
response = await renderPage(webRequest, resolvedUrl, ssrManifest);
```

Static assets are already handled before middleware runs by a separate `tryServeStatic()` call, making the dot-exclusion redundant for its stated purpose and purely harmful. Applications using catch-all routes (`pages/[[...slug]].tsx` or `pages/[...slug].tsx`) will serve catch-all page content for dot-containing paths while completely bypassing any authentication or authorization logic defined in middleware.

Impact

Applications that rely on middleware for authentication or authorisation — and that have not configured an explicit `config.matcher` — can have those protections entirely bypassed by any unauthenticated attacker who appends a dot-containing extension to a protected path. Because this deviation from Next.js behaviour is silent and undocumented, developers migrating existing Next.js middleware are unlikely to anticipate or test for it. The impact depends on the application, but may include unauthorized access to sensitive user data, administrative functionality, or confidential content.

Proof of Concept

exploit.py Python

```python
import requests

TARGET = "http://localhost:3000"  # Edit to point at your vinext Pages Router server

# Assumes:
# 1. Middleware (no explicit matcher) redirects unauthenticated users to /login
# 2. App has pages/[[...slug]].tsx that renders all paths
# 3. /protected is auth-protected

print("Step 1: Confirm normal protection works...")
r = requests.get(f"{TARGET}/protected", allow_redirects=False)
print(f"  /protected: {r.status_code}")
# Expected: 307 redirect to /login (middleware blocked)

print("
Step 2: Test dot-containing bypass...")
r = requests.get(f"{TARGET}/protected.config", allow_redirects=False)
print(f"  /protected.config: {r.status_code}")
# Expected: 307 (middleware should still run)
# Actual (vulnerable): 200 (catch-all serves content, middleware was skipped)
if r.status_code == 200:
    print("  [+] CONFIRMED: Middleware bypass via dot-containing path!")
    print(f"  Content-Type: {r.headers.get('content-type')}")
else:
    print(f"  Status {r.status_code} - middleware still ran or no catch-all route")
```

Remediation

To remediate this issue, AISafe recommends to remove the `!pathname.includes('.')` exclusion from the default middleware matcher in both `packages/vinext/src/server/middleware.ts` and `packages/vinext/src/index.ts`, aligning the default behaviour with Next.js, and to advise developers who wish to exclude static file paths to configure an explicit `config.matcher` pattern.

Status

Open No action has been taken to address this finding yet.

Incomplete ReDoS Guard in isSafeRegex() Low AIS-ASO-VIN-009

Description

The `isSafeRegex()` function in `packages/vinext/src/config/config-matchers.ts` is intended to prevent Regular Expression Denial of Service by rejecting dangerous patterns before they are compiled via `new RegExp()`. The function correctly identifies nested quantifier patterns such as `(a+)+` or `(.*)*` by tracking quantifier depth, but it performs no analysis of alternation branches (`|`) inside repeated groups.

packages/vinext/src/config/config-matchers.ts TypeScript

```typescript
export function isSafeRegex(pattern: string): boolean {
  // Track parenthesis nesting depth and whether we've seen a quantifier
  // at each depth level.
  // ... (heuristic: detects (a+)+ but NOT (a|aa)+)
  return true;
}
```

Overlapping alternation patterns such as `(a|aa)+` or `(ab|a)+` — where one alternate is a prefix of another — pass the safety check and are compiled without restriction. When an attacker submits a request path crafted to match N repetitions of the overlapping character sequence followed by a non-matching suffix, the V8 NFA regex engine backtracks exponentially, yielding O(2^N) CPU usage.

The attack requires a developer to have written such a pattern in a middleware `config.matcher`, a redirect or rewrite source, or a `has`/`missing` condition. Although overlapping alternation patterns are uncommon in typical route configurations, the guard's stated purpose is to catch exactly this class of mistake, and its failure to do so is silent.

Impact

If a developer introduces an overlapping alternation pattern into a middleware matcher or routing rule, the incomplete guard will allow it to compile. An attacker who discovers such a route can send crafted requests to cause CPU exhaustion, resulting in 503 errors on Cloudflare Workers (CPU time limit exceeded) or event loop stalls on Node.js, degrading service availability for all users. The precondition of a vulnerable developer-written pattern limits the likelihood, but the impact when present is a denial of service.

Remediation

To remediate this issue, AISafe recommends to extend `isSafeRegex()` to reject any pattern containing overlapping alternation inside a repeated group (e.g., `(X|Y)+` where one alternate is a prefix of another), or to replace the custom heuristic entirely with a proven linear-time regex safety library such as `safe-regex` or `re2`.

Status

Open No action has been taken to address this finding yet.

Cross-Request <Head> Leakage in CF Worke... AIS-ASO-VIN-010

Medium

Description

The Pages Router production Cloudflare Worker entry generated by vinext imports `next/head` for its `resetSSRHead` and `getSSRHeadHTML` helpers but does not import `vinext/head-state`, the module that registers `AsyncLocalStorage`-backed per-request isolation for `<Head>` state. Without that import, `head.ts` falls back to a raw module-level `_ssrHeadElements` array shared across all concurrently executing requests in the same Worker isolate.

Two distinct issues arise from this shared state.

Issue 1 — Wrong call ordering. `getSSRHeadHTML()` is called before any rendering occurs. Because the `<Head>` elements are populated during the render phase (via `renderToStringAsync` / `renderToReadableStream`), reading the head state prior to rendering always yields an empty or stale result. Pages Router `<Head>` content is therefore silently omitted from every SSR response.

Issue 2 — Cross-request race condition. The sequence around head state at lines 1367–1370 contains an `await` between the reset and the read:

packages/vinext/src/index.ts — TypeScript

```
if (typeof resetSSRHead === 'function') resetSSRHead();  // L1367: clears shared
array
if (typeof flushPreloads === 'function') await flushPreloads();  // L1368: ASYNC
YIELD
// ^^ Another request can render between reset and getSSRHeadHTML!
const ssrHeadHTML = typeof getSSRHeadHTML === 'function' ? getSSRHeadHTML() :
'';  // L1370
```

While Request A is suspended at the `await flushPreloads()` yield point, a concurrently executing Request B can render its `DocumentComponent` and push its `<Head>` elements to the shared module-level array:

packages/vinext/src/shims/head.ts — TypeScript

```
if (html) _getSSRHeadElements().push(html);  // User B renders, pushes to shared
array
```

When Request A resumes and calls `getSSRHeadHTML()` at line 1370, it reads the array — which now contains User B's personalized head elements — and embeds them in User A's HTML response:

packages/vinext/src/index.ts — TypeScript

```
const ssrHeadHTML = getSSRHeadHTML();  // User A reads User B's elements from
shared array!
```

For applications that render personalized content in `<Head>` — such as `<title>Welcome, Alice!</title>` or user-specific Open Graph tags — this constitutes an information disclosure across users.

Impact

Pages Router applications deployed to Cloudflare Workers experience two simultaneous problems. All `<Head>` elements — including page titles, meta descriptions, and canonical URLs — are silently dropped from every SSR response, degrading SEO rankings and social sharing previews. Under concurrent load, personalized head content from one user's request can appear in another

user's HTML response, disclosing user-specific information such as names, identifiers, or contextual data encoded in page titles or Open Graph tags.

Remediation

To remediate this issue, AISafe recommends to add `import 'vinext/head-state'` to the generated Pages Router Cloudflare Worker entry so that `AsyncLocalStorage`-backed isolation is active, and to move the `getSSRHeadHTML()` call to after `renderToStringAsync` / `renderToReadableStream` so that the current request's rendered head elements are correctly captured.

Status

Open No action has been taken to address this finding yet.

Server Actions CSRF Bypass via x-forward... AIS-ASO-VIN-011

Medium

Description

The `__validateCsrfOrigin()` function used to protect Server Actions reads the `x-forwarded-host` header from the incoming request and treats it as a trusted source of the application's host identity, without first validating that the header originates from a trusted proxy. In Cloudflare Workers deployments, Cloudflare does not set or overwrite `x-forwarded-host`, meaning the value is entirely attacker-controlled.

packages/vinext/src/server/app-dev-server.ts — TypeScript

```
const hostHeader = (
    request.headers.get("x-forwarded-host") ||
    request.headers.get("host") ||
    ""
  ).split(",")[0].trim().toLowerCase();
```

The extracted `hostHeader` value is stored without any validation against a trusted host list:

packages/vinext/src/server/app-dev-server.ts — TypeScript

```
const hostHeader = (request.headers.get("x-forwarded-host") ||
request.headers.get("host") || "").split(",")[0].trim().toLowerCase();
```

The CSRF check then compares this attacker-controlled `hostHeader` against the `Origin` header, which the attacker also controls in a non-browser HTTP client:

packages/vinext/src/server/app-dev-server.ts — TypeScript

```
if (originHost === hostHeader) return null;  // CSRF check passes!
```

An attacker can therefore bypass CSRF protection for any Server Action by sending a request in which both `Origin` and `x-forwarded-host` are set to the same attacker-controlled value. The Node.js production server applies `VINEXT_TRUSTED_HOSTS` validation before resolving the host, but this validation is absent from the generated RSC virtual module used in the Cloudflare Workers deployment path.

Impact

An attacker with access to a victim's session credentials can invoke authenticated Server Actions cross-origin by spoofing the `x-forwarded-host` header, bypassing the CSRF origin check entirely. The impact depends on the sensitivity of the exposed Server Actions, which may include data mutations, privilege escalation, or account takeover. Exploitation from a browser context additionally requires permissive CORS or an existing XSS vulnerability, but is directly exploitable from any HTTP client.

Proof of Concept

exploit.py — Python

```
import requests

TARGET = "https://your-worker.workers.dev"  # Edit this
VICTIM_COOKIE = "session=abc123"  # Victim's session cookie
ACTION_ID = "discovered-action-id"  # From JS bundle
```

```python

print("Step 1: Sending Server Action request with spoofed x-forwarded-host...")
# Attacker sets both Origin and x-forwarded-host to their domain
response = requests.post(
    f"{TARGET}/",
    headers={
        "Origin": "http://attacker.com",
        "x-forwarded-host": "attacker.com",
        "x-rsc-action": ACTION_ID,
        "Cookie": VICTIM_COOKIE,
        "Content-Type": "text/plain",
    },
    data="[]",
)

print(f"Status: {response.status_code}")
# Expected if fixed: 403 Forbidden
# Actual (vulnerable): 200 OK - Server Action executed!
if response.status_code != 403:
    print("[+] CSRF bypass confirmed! Server Action executed cross-origin.")
else:
    print("[-] Request blocked (may be fixed)")
```

Remediation

To remediate this issue, AISafe recommends to update `__validateCsrfOrigin()` in the Cloudflare Workers deployment path to use only the `Host` header for CSRF host resolution, or to validate `x-forwarded-host` against a configured `VINEXT_TRUSTED_HOSTS` allowlist before treating it as authoritative, consistent with the approach used in the Node.js production server.

Status

Open No action has been taken to address this finding yet.

Wildcard CSRF Origin Allows Apex Domain Low AIS-ASO-VIN-012

Description

The `__isOriginAllowed()` function that evaluates `serverActionsAllowedOrigins` wildcard patterns includes an unintended check that permits requests from the apex domain when a subdomain wildcard such as `*.example.com` is configured. The check uses `pattern.slice(2)` to extract the bare domain from the wildcard string and compares it directly against the request origin:

packages/vinext/src/server/app-dev-server.ts TypeScript

```
if (origin === pattern.slice(2) || origin.endsWith(suffix)) return true;  // BUG: pattern.slice(2)='example.com' matches apex domain
```

For a pattern of `*.example.com`, `pattern.slice(2)` yields `example.com`. The condition `origin === pattern.slice(2)` therefore allows requests where `Origin` is `example.com` — the apex domain — in addition to any subdomain. Next.js's own implementation contains no such apex domain check; it matches only via `endsWith()` on the dot-prefixed suffix. The developer comment describing the wildcard as matching only subdomains is inaccurate with respect to the actual code behaviour.

Impact

Developers who configure a wildcard pattern such as `*.preview.myapp.com` to restrict CSRF-allowed origins to preview subdomain deployments will unknowingly also permit requests from the apex domain `preview.myapp.com`. This deviates from the documented Next.js behaviour developers rely on. In scenarios involving shared public hosting domains — for example `*.github.io` — the apex `github.io` would also be CSRF-allowed, potentially enabling any GitHub Pages user to invoke the application's Server Actions cross-origin.

Remediation

To remediate this issue, AISafe recommends to remove the `origin === pattern.slice(2)` apex domain check from `__isOriginAllowed()`, retaining only the `origin.endsWith(suffix)` subdomain match, so that the wildcard behaviour is consistent with Next.js's documented semantics.

Status

Open No action has been taken to address this finding yet.

Draft Mode Cookie Missing Secure Flag Medium AIS-ASO-VIN-014

Description

The `draftMode().enable()` implementation in `packages/vinext/src/shims/headers.ts` constructs a `Set-Cookie` header that includes `HttpOnly` and `SameSite=Lax` attributes but omits the `Secure` flag. The cookie carries the raw draft secret value derived from the `__VINEXT_DRAFT_SECRET` environment variable.

packages/vinext/src/shims/headers.ts TypeScript

```
const secret = process.env.__VINEXT_DRAFT_SECRET;
```

packages/vinext/src/shims/headers.ts TypeScript

```
state.draftModeCookieHeader =
        `${DRAFT_MODE_COOKIE}=${secret}; Path=/; HttpOnly; SameSite=Lax`;
```

Without the `Secure` attribute, browsers will transmit the cookie over plain HTTP connections. When the vinext production Node.js server is accessed without a TLS-terminating reverse proxy in front of it, the draft mode cookie and its secret value are sent in cleartext. A network-positioned attacker — such as one on the same LAN, a malicious Wi-Fi access point, or a compromised ISP — can intercept HTTP responses or subsequent requests and extract the cookie. Possessing the cookie allows the attacker to enable draft mode on any page, bypassing Incremental Static Regeneration caching.

Impact

A network-positioned attacker who intercepts the draft mode cookie over a plain HTTP connection gains the ability to force any page to render in draft mode, bypassing the ISR cache and potentially accessing unpublished or draft content. This may also increase origin server load by defeating caching. The risk is most relevant to Node.js production deployments that rely on the server's default plain-HTTP listener without a TLS-terminating proxy; Cloudflare Workers deployments are unaffected as TLS terminates at the CDN edge.

Remediation

To remediate this issue, AISafe recommends to add the `Secure` attribute to the `Set-Cookie` header constructed in both `draftMode().enable()` and `draftMode().disable()` in `packages/vinext/src/shims/headers.ts`.

Status

Open No action has been taken to address this finding yet.

KV Cache Entries Lack HMAC Integrity Che... High AIS-ASO-VIN-016

Description

The `KVCacheHandler` in `packages/vinext/src/cloudflare/kv-cache-handler.ts` stores and retrieves cache entries as plain JSON from Cloudflare KV without any cryptographic integrity protection. An attacker with write access to the KV namespace — obtained via a compromised `CLOUDFLARE_API_TOKEN`, a malicious build dependency, or a supply chain attack — can write crafted entries containing arbitrary HTML, RSC data, or API response bodies.

When a cached URL is requested, the handler reads the raw KV value and deserialises it:

packages/vinext/src/cloudflare/kv-cache-handler.ts TypeScript

```
const raw = await this.kv.get(kvKey);
if (!raw) return null;
let parsed: unknown;
try {
  parsed = JSON.parse(raw);
} catch { ... }
```

The `validateCacheEntry()` function is then called, but it performs only structural checks — verifying that `lastModified` is a number, `tags` is an array, and `value.kind` is an allowed string. The content fields (`html`, `body`, `rscData`) are not validated:

packages/vinext/src/cloudflare/kv-cache-handler.ts TypeScript

```
const entry = validateCacheEntry(parsed);
```

The unchecked cache value is then returned directly to the server and served to all users:

packages/vinext/src/cloudflare/kv-cache-handler.ts TypeScript

```
return { lastModified: entry.lastModified, value: entry.value };
```

A well-formed malicious KV entry — for example `{ value: { kind: 'APP_PAGE', html: '<script>...</script>', status: 200 }, tags: [], lastModified: ..., revalidateAt: ... }` — passes structural validation and is served verbatim to every visitor to the affected URL until the cache entry expires. No application logs record KV writes performed via the Cloudflare API, making such attacks difficult to detect.

Impact

A single write operation to the Cloudflare KV namespace can inject arbitrary content — including malicious scripts, credential-harvesting pages, or defacement content — into any cached page, causing it to be served to all visitors until the entry expires or is explicitly invalidated. Because the attack requires no ongoing attacker presence and leaves no application-level log trace, it enables persistent cross-site scripting and large-scale content tampering at a level of stealth and reach that is disproportionate to the access required.

Remediation

To remediate this issue, AISafe recommends to add HMAC-SHA256 integrity protection to all KV cache entries by computing a signature over the serialized entry at write time using `crypto.subtle.sign()`, storing the signature alongside the data, and verifying the signature at read time before parsing or using any entry content, treating verification failures as cache misses.

Status

Risk Accepted Requires pre-exploitation of another service.

Unvalidated Dynamic Import in navigateCl... High AIS-ASO-VIN-017

Description

During client-side navigation, `packages/vinext/src/shims/router.ts` fetches the target page's HTML from the server, extracts a JavaScript module URL from the `__NEXT_DATA__.__vinext.pageModuleUrl` field of the embedded JSON, and passes it directly to `import()` without any path validation.

The server HTML is fetched and parsed:

packages/vinext/src/shims/router.ts TypeScript

```
const res = await fetch(url, { headers: { Accept: 'text/html' } });
```

packages/vinext/src/shims/router.ts TypeScript

```
const html = await res.text();
```

packages/vinext/src/shims/router.ts TypeScript

```
const nextData = JSON.parse(match[1]);
```

The module URL is extracted from the untrusted `__NEXT_DATA__` payload:

packages/vinext/src/shims/router.ts TypeScript

```
nextData.__vinext?.pageModuleUrl;
```

It is then supplied directly to a dynamic `import()` call:

packages/vinext/src/shims/router.ts TypeScript

```
const pageModule = await import(/* @vite-ignore */ pageModuleUrl);
```

The initial hydration path in `entry.ts` applies an `isValidModulePath()` guard that rejects external URLs containing `://` and path traversal sequences. The `navigateClient()` function that handles all subsequent client-side navigations does not apply this guard. A legacy regex fallback (used when `__NEXT_DATA__` is absent) also extracts module URLs from raw HTML `import()` patterns, widening the attack surface further.

If an attacker can influence the `__NEXT_DATA__` JSON embedded in a server response — for example, via stored XSS that injects a forged `<script id="__NEXT_DATA__">` tag ahead of the legitimate one, or via a server-side HTML injection vulnerability — the attacker can supply an arbitrary external URL as `pageModuleUrl`, causing the browser to dynamically import and execute attacker-controlled JavaScript during client navigation.

Impact

An attacker who can inject a crafted `__NEXT_DATA__` payload into a server response can cause victim browsers to dynamically import and execute arbitrary external JavaScript during any client-side navigation event. The imported code runs with full access to the user's session, DOM, and authentication tokens, enabling session hijacking, credential theft, and arbitrary actions performed on the victim's behalf. The inconsistency with the explicit protection present in the initial hydration path suggests that this check was intentionally omitted rather than overlooked, but the client navigation code path is equally exposed.

Remediation

To remediate this issue, AISafe recommends to apply the existing `isValidModulePath()` validation from `entry.ts` to the `navigateClient()` function in `router.ts` before the `import()` call at line 342, and to apply the same check to `appModuleUrl` before it is imported at line 360, by moving `isValidModulePath()` to a shared utility module.

Status

Open No action has been taken to address this finding yet.

Auth Bypass: /api Routes Skip Middleware Medium AIS-ASO-VIN-019

Description

In the Pages Router, vinext's default middleware matcher unconditionally excludes all paths starting with `/api` when no explicit `config.matcher` is configured. This deviates silently from Next.js's documented behaviour, where middleware with no matcher runs on every path including `/api/*`.

packages/vinext/src/server/middleware.ts TypeScript

```
if (!matcher) {
  return (
    !pathname.startsWith('/_next') &&
    !pathname.startsWith('/api') &&  // ⟵ silently skips ALL /api routes
    !pathname.includes('.') &&
    pathname !== '/favicon.ico'
  );
}
```

When `matchesMiddleware` returns `false` for an `/api` path, the middleware is skipped entirely and the request proceeds directly to the route handler:

packages/vinext/src/server/middleware.ts TypeScript

```
if (!matchesMiddleware(url.pathname, matcher)) { return { continue: true }; }
```

The same exclusion is inlined into the generated Pages Router Cloudflare Worker bundle, meaning production Worker deployments are equally affected:

packages/vinext/src/index.ts TypeScript

```
    return !pathname.startsWith('/_next') && !pathname.startsWith('/api') && !
pathname.includes('.') && pathname !== '/favicon.ico';
```

A developer who migrates authentication middleware from Next.js to vinext without adding an explicit `config.matcher` will expect middleware to protect API routes — exactly as it does in Next.js — but will instead find all `/api/*` paths completely unprotected.

Impact

Developers migrating from Next.js to vinext who rely on the default middleware matcher to protect API routes will unknowingly leave all `/api/*` endpoints accessible to unauthenticated requests. Depending on the application, this can expose sensitive user data, administrative functionality, or data mutation operations through the API layer with no authentication check applied. The deviation is silent and undocumented, making it difficult to detect without explicit testing of API routes.

Remediation

To remediate this issue, AISafe recommends to remove the `!pathname.startsWith('/api')` exclusion from the default middleware matcher in both `packages/vinext/src/server/middleware.ts` and `packages/vinext/src/index.ts`, aligning the default behaviour with Next.js, and to document that developers who wish to exclude API routes from middleware must configure an explicit `config.matcher`.

Status

Open No action has been taken to address this finding yet.

Host Header Injection Open Redirect in N... Medium AIS-ASO-VIN-022

Description

The `resolveHost()` function in `packages/vinext/src/server/prod-server.ts` validates the `x-forwarded-host` header against the `VINEXT_TRUSTED_HOSTS` allowlist but returns `req.headers.host` — the plain HTTP `Host` header — without any corresponding validation when `x-forwarded-host` is absent. In HTTP/1.1, the client sets the `Host` header freely, making it attacker-controlled.

packages/vinext/src/server/prod-server.ts TypeScript

```
function resolveHost(req, fallback) {
  const rawForwarded = req.headers['x-forwarded-host'];
  const hostHeader = req.headers.host;
  if (rawForwarded) {
    const forwardedHost = rawForwarded.split(',')[0].trim().toLowerCase();
    if (forwardedHost && trustedHosts.has(forwardedHost)) return forwardedHost;
  }
  return hostHeader || fallback;  // hostHeader is user-controlled, not validated
}
```

The unvalidated host value is assigned and then embedded directly into the `webRequest` URL:

packages/vinext/src/server/prod-server.ts TypeScript

```
const hostHeader = resolveHost(req, `${host}:${port}`);
```

packages/vinext/src/server/prod-server.ts TypeScript

```
const webRequest = new Request(`${protocol}://${hostHeader}${url}`, { ... });
```

The crafted `webRequest` is passed to the middleware execution pipeline:

packages/vinext/src/server/prod-server.ts TypeScript

```
const result = await runMiddleware(webRequest);
```

Any middleware that constructs a redirect target using the standard Next.js pattern `new URL('/path', request.url)` will resolve the destination URL relative to the attacker-supplied host. The resolved URL is then written verbatim to the `Location` response header:

packages/vinext/src/server/prod-server.ts TypeScript

```
res.writeHead(result.redirectStatus ?? 307, { Location: result.redirectUrl });
```

An attacker sends `GET /middleware-redirect HTTP/1.1` with `Host: evil.com` and no `X-Forwarded-Host`. The server constructs `request.url = "http://evil.com/middleware-redirect"`, middleware resolves `new URL('/about', request.url)` to `"http://evil.com/about"`, and the server responds with `307 Location: http://evil.com/about`. The developers explicitly protected against `x-forwarded-host` injection by requiring the value to appear in `VINEXT_TRUSTED_HOSTS`, but applied no equivalent control to the plain `Host` header.

Impact

An attacker with network access to the Node.js production server can craft HTTP requests with an arbitrary `Host` header, causing middleware to issue redirect responses pointing to attacker-controlled domains. This enables phishing attacks, OAuth token theft via redirect-based authentication flows, and credential harvesting against any user who follows a server-issued

redirect. The vulnerability affects all `vinext start` deployments where middleware performs redirects using `request.url` as a base URL, which is the standard pattern documented in Next.js examples.

Proof of Concept

exploit.py — Python

```python
import requests

TARGET = "http://localhost:3000"  # vinext start default

# Attack: Host header injection → open redirect via middleware redirect
# Middleware at /middleware-redirect does: NextResponse.redirect(new URL("/about", request.url))
# With Host: evil.com → request.url = "http://evil.com/middleware-redirect"
# → redirect target = "http://evil.com/about"

print("Testing Host Header Injection → Open Redirect...")

response = requests.get(
    f"{TARGET}/middleware-redirect",
    headers={"Host": "evil.com"},
    allow_redirects=False  # Don't follow redirect - we want to see Location header
)

print(f"Status: {response.status_code}")
print(f"Location: {response.headers.get('Location', 'N/A')}")

# Expected: Location = "http://evil.com/about"
# This redirects the victim's browser to the attacker's domain
assert response.status_code in (307, 308), f"Expected redirect, got {response.status_code}"
location = response.headers.get("Location", "")
assert "evil.com" in location, f"Expected evil.com in Location, got: {location}"
print(f"[+] Open redirect confirmed via Host header injection: {location}")
```

Remediation

To remediate this issue, AISafe recommends to extend `resolveHost()` to validate `req.headers.host` against the same `VINEXT_TRUSTED_HOSTS` allowlist used for `x-forwarded-host`, returning `fallback` when the `Host` header does not match any trusted entry.

Status

Open No action has been taken to address this finding yet.

Timing Attack on Draft Mode Secret Check Low AIS-ASO-VIN-004

Description

The draft mode secret comparison in `packages/vinext/src/shims/headers.ts` uses the standard JavaScript `===` operator, which short-circuits on the first differing character and is therefore not constant-time. An attacker who can measure HTTP response latency across many repeated requests could exploit this timing side-channel to enumerate the `__VINEXT_DRAFT_SECRET` value character-by-character.

packages/vinext/src/shims/headers.ts TypeScript

```
state.headersContext.cookies.get(DRAFT_MODE_COOKIE) === secret
```

The user-supplied `__prerender_bypass` cookie value is retrieved from the request context and compared directly against the server-side secret in a single non-constant-time expression. Although Cloudflare Workers coarsen local timers by freezing `performance.now()` during CPU execution, remote timing attacks that measure full round-trip HTTP response latency remain theoretically feasible with a sufficiently large sample set. Cloudflare's own documentation explicitly recommends `crypto.subtle.timingSafeEqual()` for this exact use case.

Impact

If the draft mode secret is successfully enumerated through a timing side-channel attack, an attacker can activate draft mode for any URL, bypassing Incremental Static Regeneration caching and receiving uncached server-side-rendered responses. This could expose unpublished or draft content before it is intentionally made public, and may also cause elevated origin load by defeating the caching layer.

Remediation

To remediate this issue, AISafe recommends to replace the `===` string comparison with `crypto.subtle.timingSafeEqual()`, encoding both operands to `Uint8Array` with `TextEncoder` before comparison and returning `false` immediately if their lengths differ.

Status

Open No action has been taken to address this finding yet.

Verbose Middleware Errors in Dev Server Low AIS-ASO-VIN-006

Description

The development-mode middleware runner in `packages/vinext/src/server/middleware.ts` concatenates the raw exception message into the body of HTTP 500 responses returned to clients. Any exception thrown from a user's middleware function is caught and its `.message` property is embedded directly in the response.

packages/vinext/src/server/middleware.ts TypeScript

```
} catch (e: any) {
    console.error('[vinext] Middleware error:', e);
    return {
      continue: false,
      response: new Response('Middleware Error: ' + (e?.message ?? String(e)), {
        status: 500,
      }),
    };
  }
```

This code path is exercised only by the Vite development server, as the function requires a `ViteDevServer` parameter. The production equivalent correctly returns a generic `'Internal Server Error'` string without any additional detail. However, if a preview or staging deployment based on the development server is publicly accessible — a common pattern in CI/CD pipelines — the exposed exception messages can include internal module paths, dependency version strings, and configuration details.

Impact

If a publicly accessible preview or CI/CD deployment uses the development server, any triggered middleware exception reveals internal details such as module paths, environment variable names, and library error messages in the HTTP response body. This information can accelerate reconnaissance for attackers targeting non-production environments.

Remediation

To remediate this issue, AISafe recommends to update the `catch` block in `packages/vinext/src/server/middleware.ts` to return a generic `'Internal Server Error'` response body without interpolating `e.message`, consistent with the production code path.

Status

Open No action has been taken to address this finding yet.

Insufficient Path Escaping in startLocal... Low AIS-ASO-VIN-013

Description

The `startLocalServer()` function in `packages/vinext/src/cloudflare/tpr.ts` constructs a JavaScript code string that is evaluated by a spawned Node.js child process via the `-e` flag. Filesystem paths are embedded into the string with only backslash escaping applied, leaving double-quote characters, newlines, and other JavaScript metacharacters unhandled.

packages/vinext/src/cloudflare/tpr.ts TypeScript

```
  const escapedProdServer = prodServerPath.replace(/\/g, "\\");
  const escapedOutDir = outDir.replace(/\/g, "\\");

  const script = [
    `import("file://${escapedProdServer}")`,
    `.then(m => m.startProdServer({ port: ${port}, host: "127.0.0.1", outDir:
"${escapedOutDir}" }))`,
    `.catch(e => { console.error("[vinext-tpr] Server failed to start:", e);
process.exit(1); });`,
  ].join("");
```

The `outDir` value is derived from the process working directory:

packages/vinext/src/cloudflare/tpr.ts TypeScript

```
const outDir = path.join(root, "dist");
```

Only backslashes are replaced before embedding. The value is then embedded inside a double-quoted JavaScript string literal. A working directory containing a double-quote character — for example `/work/test"});process.exit(0);//` — would break out of the string context and inject arbitrary JavaScript code into the child process. The code would execute with the same privileges as the invoking user. Exploitation requires the `--experimental-tpr` build flag and a working directory path containing a double-quote, which is unusual but valid on Unix systems.

Impact

This vulnerability affects the `vinext deploy --experimental-tpr` build tool and carries no remote attack surface. Any injected code would run with the same privileges as the developer who invoked the command. The practical risk is limited to unusual path environments, but the incomplete escaping pattern represents a defensive coding gap that should be corrected to prevent unexpected behavior.

Remediation

To remediate this issue, AISafe recommends to replace the manual backslash replacement with `JSON.stringify()` when embedding filesystem paths into the generated JavaScript string, or to pass path values to the child process as environment variables rather than inlining them in evaluated code.

Status

Open No action has been taken to address this finding yet.

Placeholder Email Endpoint in CI Notifie... Low AIS-ASO-VIN-015

Description

The CI failure notification script at `.github/scripts/send-email.mjs` sends `POST` requests to `https://api.example.com/send-email` — a non-functional placeholder URL accompanied by an inline `TODO` comment acknowledging that it must be replaced. Every invocation of this script will receive a connection error or `404` response, log the failure to stderr, and exit with a non-zero code.

Code

```
    // TODO: replace with your actual email API endpoint and payload format
    const res = await fetch("https://api.example.com/send-email", {

```

As a result, all CI/CD failure notifications generated by the `tip.yml` workflow are silently discarded. Although the `EMAIL_API_TOKEN` secret is validated at startup and will cause an early exit if absent, the actual notification delivery never succeeds regardless of token availability.

Impact

Engineering teams relying on automated email alerts to monitor Next.js canary build health will not receive any notification when the tip workflow fails. Build regressions or infrastructure issues may go undetected for an extended period, delaying remediation.

Remediation

To remediate this issue, AISafe recommends to replace the placeholder URL `https://api.example.com/send-email` in `.github/scripts/send-email.mjs` with the actual email service endpoint and update the request payload to match the chosen provider's API format.

Status

Open No action has been taken to address this finding yet.

Filesystem Path Exposed in API Route Err... Low AIS-ASO-VIN-018

Description

When a Pages Router API route module exists but does not export a default handler function, the Vite development server responds with an HTTP 500 message that includes the full absolute filesystem path to the module file.

The absolute path is loaded during module resolution:

packages/vinext/src/server/api-handler.ts TypeScript

```
const apiModule = await server.ssrLoadModule(route.filePath);
```

It is then concatenated verbatim into the response body:

packages/vinext/src/server/api-handler.ts TypeScript

```
if (typeof handler !== "function") {
  res.statusCode = 500;
  res.end(`API route ${route.filePath} does not export a default function`);
  return true;
}
```

Any HTTP client can trigger this error condition by requesting an API route that lacks a default export. The resulting response body discloses the full server-side path, which typically includes the developer's home directory name, the project directory structure, and implicitly the operating system convention (e.g., `/home/username/project/pages/api/hello.ts`). This path information is available to unauthenticated clients.

Impact

This issue is limited to the development server. If a development or CI/CD preview environment is network-accessible, an attacker who discovers an improperly exported API route can retrieve the server's absolute filesystem layout, including developer usernames and internal directory names, which can aid reconnaissance for more targeted attacks.

Remediation

To remediate this issue, AISafe recommends to replace the `route.filePath` interpolation in the HTTP 500 response with a generic message that omits internal path information, logging the full path to the server console instead for developer debugging purposes.

Status

Open No action has been taken to address this finding yet.

XML Injection via Unescaped priority in ... Low AIS-ASO-VIN-020

Description

The `sitemapToXml()` function in `packages/vinext/src/server/metadata-routes.ts` applies `escapeXml()` to all string fields in a sitemap entry (`url`, `lastModified`, `changeFrequency`, `images`) but directly interpolates the `priority` field into the XML output without any escaping or type validation.

packages/vinext/src/server/metadata-routes.ts TypeScript

```typescript
export function sitemapToXml(entries: SitemapEntry[]): string {
```

packages/vinext/src/server/metadata-routes.ts TypeScript

```typescript
if (entry.priority !== undefined) {
  lines.push(`    <priority>${entry.priority}</priority>`);
}
```

Although `priority` is typed as `number` in TypeScript, there is no runtime enforcement of this constraint. A developer whose `sitemap.ts` reads priority values from a database or external data source may pass a string value. If that value contains XML special characters or closing tags — for example `0.8</priority><fake>injected</fake><priority>0.8` — the resulting sitemap XML will have a malformed or injected structure. The inconsistency with the explicit escaping applied to every other field indicates an oversight rather than an intentional decision.

Impact

Exploitation requires a developer's `sitemap.ts` to pass a user-influenced or externally sourced non-numeric value for `priority`. If achieved, an attacker could inject arbitrary XML into the sitemap document, potentially manipulating search engine crawl priority signals or injecting misleading entries. There is no direct path to end-user data exposure or code execution.

Remediation

To remediate this issue, AISafe recommends to add a runtime numeric check to the `priority` field in `sitemapToXml()` — passing it through unchanged if it is a `number`, or calling `escapeXml(String(entry.priority))` otherwise — consistent with the escaping applied to all other fields.

Status

Open No action has been taken to address this finding yet.

robots.txt Directive Injection via Newli... Low AIS-ASO-VIN-021

Description

The `robotsToText()` function in `packages/vinext/src/server/metadata-routes.ts` constructs `robots.txt` output by directly interpolating string fields — including `userAgent`, `allow`, `disallow`, `sitemap`, and `host` — without stripping embedded newline characters. Because `robots.txt` uses newlines as directive and record separators per RFC 9309, a field value containing or will inject additional directives into the generated output.

packages/vinext/src/server/metadata-routes.ts TypeScript

```
for (const agent of agents) {
  lines.push(`User-Agent: ${agent}`);
}
```

The same unsanitized pattern applies to all other string fields. vinext's development server explicitly supports asynchronous `robots.ts` exports where developers commonly fetch configuration from a CMS or external API. If such a source returns a value containing an embedded newline — for example `"Googlebot Disallow: /"` — the resulting `robots.txt` will contain an injected `Disallow` directive that was never explicitly configured by the developer.

Impact

An attacker with write access to a CMS or external configuration source that supplies `robots.txt` field values could inject arbitrary `robots.txt` directives. The practical impact is limited to manipulating search engine crawling behaviour — for example, preventing all crawlers from indexing the site, causing SEO damage. User data exposure, authentication bypass, and code execution are not possible through this vector.

Remediation

To remediate this issue, AISafe recommends to sanitize all string fields in `robotsToText()` by replacing and characters with a space before interpolation, for example using `s.replace(/[\r\n]/g, ' ')` applied to `userAgent`, `allow`, `disallow`, `sitemap`, and `host` values.

Status

Open No action has been taken to address this finding yet.