Steakhouse App Security Review

Auditors

Dreyand, Security Researcher

Mikey96, Security Researcher

Report prepared by: Lucas Goiriz

September 25, 2025

Contents

1 About Spearbit

Spearbit is a decentralized network of expert security engineers offering reviews and other security related services to Web3 projects with the goal of creating a stronger ecosystem. Our network has experience on every part of the blockchain technology stack, including but not limited to protocol design, smart contracts and the Solidity compiler. Spearbit brings in untapped security talent by enabling expert freelance auditors seeking flexibility to work on interesting projects together.

Learn more about us at spearbit.com

2 Introduction

Steakhouse builds transparent, efficient, and accessible financial primitives to power the next generation of capital markets on public blockchains.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Steakhouse App according to the specific commit. Any modifications to the code will require a new security review.

3 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

3.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users.
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable.
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies.

3.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

3.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

4 Executive Summary

Over the course of 3 days in total, Steakhouse engaged with Spearbit to review the app protocol. In this period of time a total of **7** issues were found.

Summary

Project Name	Steakhouse
Repository	app
Commit	caeaa12a
Type of Project	Web2, Frontend
Audit Timeline	Sep 11th to Sep 14th

Issues Found

Severity	Count	Fixed	Acknowledged
Critical Risk	0	0	0
High Risk	0	0	0
Medium Risk	0	0	0
Low Risk	7	6	1
Gas Optimizations	0	0	0
Informational	0	0	0
Total	**7**	**6**	**1**

5 Findings

5.1 Low Risk

5.1.1 XSS in Curator Data Rendering

Severity: Low Risk

Context: *(No context files were provided by the reviewers)*

Description: A Cross-Site Scripting (XSS) vulnerability exists in the vault detail pages where curator data from the WHISK API is rendered without proper sanitization. An attacker could potentially inject malicious JavaScript through curator URLs and images, leading to session hijacking, wallet theft, or other malicious activities.

We have attempted to create a vault on the Morpho curator app that would contain these malicious payloads, however filtering is in place to use only trusted vaults.

```
supportedVaults: {
    [mainnet.id]: [
      { address: getAddress("0xBEEF01735c132Ada46AA9aA4c54623cAA92A64CB"), tag: "Prime" }, // Steakhouse
      ↪ USDC
      { address: getAddress("0xbEEF02e5E13584ab96848af90261f0C8Ee04722a"), tag: "Prime" }, // Steakhouse
      ↪ PYUSD
      { address: getAddress("0xbEef047a543E45807105E51A8BBEFCc5950fcfBa"), tag: "Prime" }, // Steakhouse
      ↪ USDT
// ...
```

Due to the filtering being place we were unable to get any of these vaults to render within the context of the web-application, however this doesn't mean that it may not be possible in the future.

Proof of Concept: The application renders curator data directly from the WHISK API without sanitization:

```
<LinkExternal href={curator.url} className="text-foreground">
  {curator.name}
  <Image src={curator.image} alt={curator.name} width={24} height={24} />
</LinkExternal>
While vaults are restricted to a predefined whitelist in src/config/index.ts, the curator data comes
↪  from the WHISK API and is not validated or sanitized before rendering.
```

Example:

```
<a href="javascript:alert('XSS')" class="text-foreground">
  Malicious Curator
  <img src="https://example.com/logo.png" alt="Malicious Curator" width="24" height="24" />
</a>
```

Likelihood: Low

Impact: High

Recommendation: Sanitize any data that is used directly in sinks from the WHISK API.

Steakhouse: Fixed in PR 8.

Spearbit: Fix verified. Seems like `defaultUrlTransform()` does a good job at sanitizing URLs.

5.1.2 CSS Injection in Chart Component

Severity: Low Risk

Context: src/components/ui/chart/index.tsx#L70-L94

Description: The chart component uses `dangerouslySetInnerHTML` to inject CSS styles dynamically. The id parameter is directly interpolated into CSS selectors without proper sanitization, potentially allowing CSS injection that could lead to XSS.

The current id is set using `React.useId();`, however if in the future the id parameter can be controlled by user input (e.g., through URL parameters, form inputs, or API responses), an attacker could inject malicious CSS.

Although this is theoretical, it is always good practice to ensure any data flowing into `dangerouslySetInnerHTML` is sanitised. We have tried various different methods of being able to control the data flowing into here or modify, however these have been unsuccessful, however this does not mean in the future it may be an option which could lead to XSS.

Example:

```
[data-chart="<img src=x onerror=alert('XSS')>"] {
  --color-primary: red;
}
```

In the context of the application XSS would be critical as executing Javascript can interact with a users already connected wallet on the origin and appear to be a legitimate transaction executed under `test.shfin.app`.

Proof of Concept:

- `app-main/src/components/ui/chart.tsx#L73-L94`

```
  return (
    <style
      // biome-ignore lint/security/noDangerouslySetInnerHtml: <shadcn component>
      dangerouslySetInnerHTML={{
        __html: Object.entries(THEMES)
          .map(
            ([theme, prefix]) => `
${prefix} [data-chart=${id}] {
${colorConfig
  .map(([key, itemConfig]) => {
    const color = itemConfig.theme?.[theme as keyof typeof itemConfig.theme] || itemConfig.color;
    return color ? `  --color-${key}: ${color};` : null;
  })
  .join("\n")}
}
`,
          )
          .join("\n"),
      }}
    />
  );
};
```

- `app-main/src/components/ui/chart/index.tsx#L70-L90`

```
  return (
    <style
      // biome-ignore lint/security/noDangerouslySetInnerHtml: Allow to get colors into chart (shadcn
      ↪  recommendation)
      dangerouslySetInnerHTML={{
        __html: Object.entries(THEMES)
          .map(
            ([theme, prefix]) => `
${prefix} [data-chart=${id}] {
${colorConfig
  .map(([key, itemConfig]) => {
    const color = itemConfig.theme?.[theme as keyof typeof itemConfig.theme] || itemConfig.color;
    return color ? `  --color-${key}: ${color};` : null;
```

```
  })
  .join("\n")}
}
`,
          )
          .join("\n"),
      }}
    />
  );
};
```

Likelihood: Low

Impact: High

Recommendation: Use filtering to ensure it can prevent CSS injection attacks by ensuring the id can only contain safe characters that won't break CSS selectors or allow injection of malicious CSS/JavaScript:

```
const sanitizedId = id.replace(/[^a-zA-Z0-9-_]/g, '');
```

Steakhouse: Fixed in PR 4.

Spearbit: Fix verified. `dangerouslySetInnerHTML` was removed.

5.1.3 XSS in Markdown Component Link Rendering

Severity: Low Risk

Context: src/components/ui/markdown/index.tsx#L25-L30

Description: A Cross-Site Scripting (XSS) vulnerability exists in the custom Markdown component where external data from the WHISK API is rendered without proper sanitization. The component overrides react-markdown's default link sanitization, allowing malicious href attributes to be injected through vault descriptions, leading to JavaScript execution and potential wallet compromise.

We have attempted to create a vault on the Morpho curator app that would contain these malicious payloads, however filtering is in place to use only trusted vaults.

```
supportedVaults: {
    [mainnet.id]: [
      { address: getAddress("0xBEEF01735c132Ada46AA9aA4c54623cAA92A64CB"), tag: "Prime" }, // Steakhouse
      ↪ USDC
      { address: getAddress("0xbEEF02e5E13584ab96848af90261f0C8Ee04722a"), tag: "Prime" }, // Steakhouse
      ↪ PYUSD
      { address: getAddress("0xbEef047a543E45807105E51A8BBEFCc5950fcfBa"), tag: "Prime" }, // Steakhouse
      ↪ USDT
// ...
```

Due to the filtering being place we were unable to get any of these vaults to render within the context of the web-application, however this doesn't mean that it may not be possible in the future.

Proof of Concept: The custom Markdown component overrides react-markdown's default link sanitization with a custom a component that directly assigns the href attribute without validation:

- `src/components/ui/markdown/index.tsx#L25-L30`

```
<ReactMarkdown
  components={{
    // ... other components
    a: ({ children, href }) => (
      <a href={href} target="_blank" rel="noopener noreferrer" className="text-primary hover:underline">
        {children}
      </a>
```

```
    ),
  }}
>
  {children}
</ReactMarkdown>
```

Vault descriptions are fetched from the WHISK API and rendered through the vulnerable Markdown component, allowing malicious markdown content to inject JavaScript protocol URLs.

Example:

```
<div class="body-large">
  <h1>Steakhouse USDC Vault</h1>
  <p>This vault provides excellent yields!
    <a href="javascript:alert('XSS - Wallet Compromised!')" target="_blank" rel="noopener noreferrer"
    ↪  class="text-primary hover:underline">
      Click here for more info
    </a>
  </p>
  <p>
    <a href="data:text/html,<script>window.ethereum.request({method:'eth_sendTransaction',params:[{to:'
    ↪  0xattacker',value:'0x'}]})</script>" target="_blank" rel="noopener noreferrer"
    ↪  class="text-primary hover:underline">
      Visit our docs
    </a> for detailed information.
  </p>
</div>
```

Likelihood: Low

Impact: High

Recommendation: Sanitize any data that is used directly in sinks from the WHISK API.

Steakhouse: Fixed in PR 5.

Spearbit: Fix verified. `defaultUrlTransform` is used to prevent XSS in the case of dangerous protocols being used in `href`.

5.1.4 Client-Side RPC Endpoint and API Key Exposure

Severity: Low Risk

Context: src/config/index.ts#L95-L111, src/providers/WalletProvider/wagmi.ts#L11-L16, src/components/ActionFlow/ActionFlowProvider.tsx

Description: The application violates security best practices by exposing RPC endpoints and API keys to client-side code. All blockchain operations (gas estimation, transaction broadcasting, receipt polling) are performed directly from the browser without server-side validation or control. The key issues are the following:

- RPC URLs with embedded API keys are exposed via `NEXT_PUBLIC_` environment variables
- All blockchain operations bypass server-side validation
- API keys are transmitted in plaintext and visible in browser dev tools
- No server-side monitoring or rate limiting of RPC calls

Affected Files:

- `src/config/index.ts#L95-L111` - RPC URL configuration.
- `src/providers/WalletProvider/wagmi.ts#L11-L16` - Wagmi transport configuration.
- `src/components/ActionFlow/ActionFlowProvider.tsx` - Client-side RPC operations.

Proof of Concept:

1. Client-Side RPC Operations:

```
// Gas estimation happens client-side
const gasEstimate = await estimateGas(publicClient, { ...txReq, account: client.account });

// Transaction broadcasting happens client-side
const hash = await sendTransaction(client, { ...txReq, gas: gasEstimateWithBuffer });

// Transaction receipt polling happens client-side
const receipt = await waitForTransactionReceipt(publicClient, { hash, pollingInterval: 1000 });
```

2. Network Request Visibility:

- Open browser dev tools → Network tab.
- Perform any blockchain operation (supply, borrow, etc...).
- Observe RPC requests containing API keys in plaintext.
- All blockchain interactions are visible and controllable from client-side.

> Important Note: While `getSimulationState()` is located in the actions folder (server-side by default), it receives a publicClient parameter created client-side and executes in the browser context, making all 42-47 RPC calls from the client.

Recommendation:

- Immediate Actions:
  - Remove `NEXT_PUBLIC_` prefix from RPC environment variables.
  - Implement server-side RPC proxy for all blockchain operations.
  - Move all RPC calls to server-side API routes.
- Security Benefits:
  - API keys remain server-side only.
  - Server-side validation and monitoring.
  - Rate limiting and abuse prevention.
  - Centralized logging and audit trail.
  - Compliance with security best practices.

Steakhouse: Fixed in PR 7.

Spearbit: Fix verified. Some slight suggestions though:

- Checking the body size is a great idea, but we believe this implementation is bound to have bypasses:

```
// Check payload size (help prevent abuse)
const contentLength = request.headers.get("Content-Length");
if (!contentLength || Number.parseInt(contentLength, 10) > MAX_PAYLOAD_BYTES) {
  return Response.json({ error: `Payload too large` }, { status: 413 });
}
```

  `Content-Length` is not the only way to pass the body size, and some servers don't even follow the spec correctly regarding it.

  Usually it's also possible to replace it with `Transfer-Encoding: chunked` to completely bypass the check.

  Ideally you want to be looking at enforcing the NextJS bodyParser size limits or trying to get the request size directly from the `.byteLength` property or similar

- Another one is related to the CORS `Origin` header:

```
  const origin = request.headers.get("Origin") ?? request.headers.get("Referer");
if (!origin) {
  return Response.json({ error: "Unauthorized" }, { status: 401 });
}
```

  Ideally you want to validate the `Origin` hostname if you really want to prevent client-side abuses.

Steakhouse: Changes made to address feedback:

- Kept "Content-Length" check as a cheap first-layer check when the header is provided
- Added a check for actual payload size after parsing which always runs. Note that `bodyParser` config is not available in Next.js App Router route handlers like it was with Pages Router API handlers.
- Didn't change the Origin check. Our CORS policy already prevents cross-origin requests from browsers. The origin check is a simple server-side abuse prevention measure, but it can be easily bypassed by spoofing this header. This remains true even when checking against specific hostnames, so we're not adding hostname validation as it would complicate dev/preview deployments without meaningfully improving security.

5.1.5 Vulnerable NPM Dependencies may allow critical server-side impact

Severity: Low Risk

Context: *(No context files were provided by the reviewers)*

Description: Several critical, moderate, and low-severity vulnerabilities were identified in third-party NPM dependencies used by the application. These issues affect cryptographic functions, input validation, file handling, and middleware operations. Vulnerabilities are present in both direct and transitive dependencies.

The most severe flaws affect the `pbkdf2` and `sha.js` packages, which can lead to weak cryptographic keys, predictable memory outputs, and bypasses in data hashing. Attackers could exploit these flaws to compromise user credentials or application integrity. Moderate vulnerabilities in the `next` package enable cache key confusion, content injection, and server-side request forgery (SSRF). Additional issues in packages such as `brace-expansion`, `tmp`, and `vite` may enable denial-of-service (DoS) attacks or unsafe file operations.

Key exploitation risks:

- Credential compromise via static or predictable keys.
- Data integrity failures through hash function bypass.
- Cache poisoning and SSRF in Next.js APIs.
- DoS via regular expression or unsafe file handling.

Most vulnerabilities are inherited through dependencies on build tools and development utilities (e.g., Storybook, Next.js, code generation).

Unpatched dependencies could allow attackers to compromise authentication, escalate privileges, access sensitive data, or cause denial of service, with significant business and compliance impact.

Proof of Concept: The following is a summarized and properly escaped excerpt from the pnpm audit output:

Severity: **critical**	Description: `pbkdf2` silently disregards Uint8Array input, returning static keys
Package	`pbkdf2`
Vulnerable versions	`<=3.1.2`
Patched versions	`>=3.1.3`

Paths	`.>@storybook/nextjs>node-polyfill-webpack-plugin>crypto-browserify>pbkdf2`
More info	GHSA-v62p-rq8g-8h59

Severity: **critical**	Description: `pbkdf2` returns predictable uninitialized/zero-filled memory for non-normalized or unimplemented algos
Package	`pbkdf2`
Vulnerable versions	`>=3.0.10 <=3.1.2`
Patched versions	>=3.1.3
Paths	`.>@storybook/nextjs>node-polyfill-webpack-plugin>crypto-browserify>pbkdf2`
More info	GHSA-h7cp-r72f-jxh6

Severity: **critical**	Description: `sha.js` is missing type checks leading to hash rewind and passing on crafted data
Package	`sha.js`
Patched versions	`>=2.4.12`
Paths	`.>@storybook/nextjs>node-polyfill-webpack-plugin>crypto-browserify>create-hash>sha.js`
More info	GHSA-95m3-7q98-8xr5

Severity: **moderate**	Description: `Next.js` Affected by Cache Key Confusion for Image Optimization API Routes
Package	`next`
Vulnerable versions	`>=15.0.0 <=15.4.4`
Patched versions	`>=15.4.5`
More info	GHSA-g5qg-72qw-gw5v

Severity: **moderate**	Description: `Next.js` Content Injection Vulnerability for Image Optimization
Package	`next`
Vulnerable versions	`>=15.0.0 <=15.4.4`
Patched versions	`>=15.4.5`

More info	GHSA-xv57-4mr9-wg8v

Severity: **moderate**	Description: `Next.js` Improper Middleware Redirect Handling Leads to SSRF
Package	`next`
Vulnerable versions	`>=15.0.0-canary.0 <15.4.7`
Patched versions	`>=15.4.7`
More info	GHSA-4342-x723-ch2f

(*additional low-severity advisories omitted for brevity*)

Likelihood: Medium

Impact: High

Recommendation:

- Upgrade all vulnerable dependencies to their latest secure versions as specified in the audit output.

  Example: `pbkdf2` to `>=3.1.3`, `sha.js` to `>=2.4.12`, `next` to `>=15.4.7`, and others as required.
- Review and update all direct and transitive dependencies using automated tools (pnpm audit, npm audit, yarn audit).
- Utilize tools like npm-check-updates to identify outdated dependencies.
- Implement automated vulnerability monitoring with GitHub Dependabot or similar services.
- Enforce dependency pinning and lockfile usage to ensure reproducible builds.
- Follow the OWASP Dependency-Check and OWASP Node.js Security Cheat Sheet for ongoing best practices.
- Audit CI/CD pipelines to prevent introduction of outdated or insecure dependencies.

Check the following references:

- GHSA-v62p-rq8g-8h59.
- GHSA-h7cp-r72f-jxh6.
- GHSA-95m3-7q98-8xr5.
- OWASP Node.js Security Cheat Sheet.

Steakhouse: Fixed in PR 3.

Spearbit: Fix verified.

5.1.6 Improper Client-Side Validation allows querying internal-Whisk API data unauthenticated

Severity: Low Risk

Description: The application's client-side enforcement of wallet connection within src/hooks/useMarketPositions.ts prevents users from querying other users' market positions via the UI. However, this restriction is not enforced server-side, allowing unauthenticated requests to the internal market positions API endpoint. By directly querying the endpoint with any wallet address, an attacker can bypass UI restrictions and trigger backend calls to the Whisk GraphQL API:

```
GET /api/account/0x6Ce5437E8372059BA18D729c99c27795Aa7bCdB0/market-positions HTTP/2
Host: test.shfin.app
...
```

Key points:

- The API does not require authentication to fetch any address's market positions data.
- The internal Whisk GraphQL API is queried for each such request, regardless of client authentication state.
- While the queried data is ultimately sourced from public on-chain information, unauthenticated programmatic access via the backend can be automated at scale.
- This could lead to excessive backend API traffic, resulting in unnecessary operational costs or potential rate-limiting issues for the Whisk API provider.

The primary risk is elevated infrastructure cost and resource utilization due to uncontrolled, unauthenticated access to internal API endpoints, rather than unauthorized data disclosure.

Likelihood: High

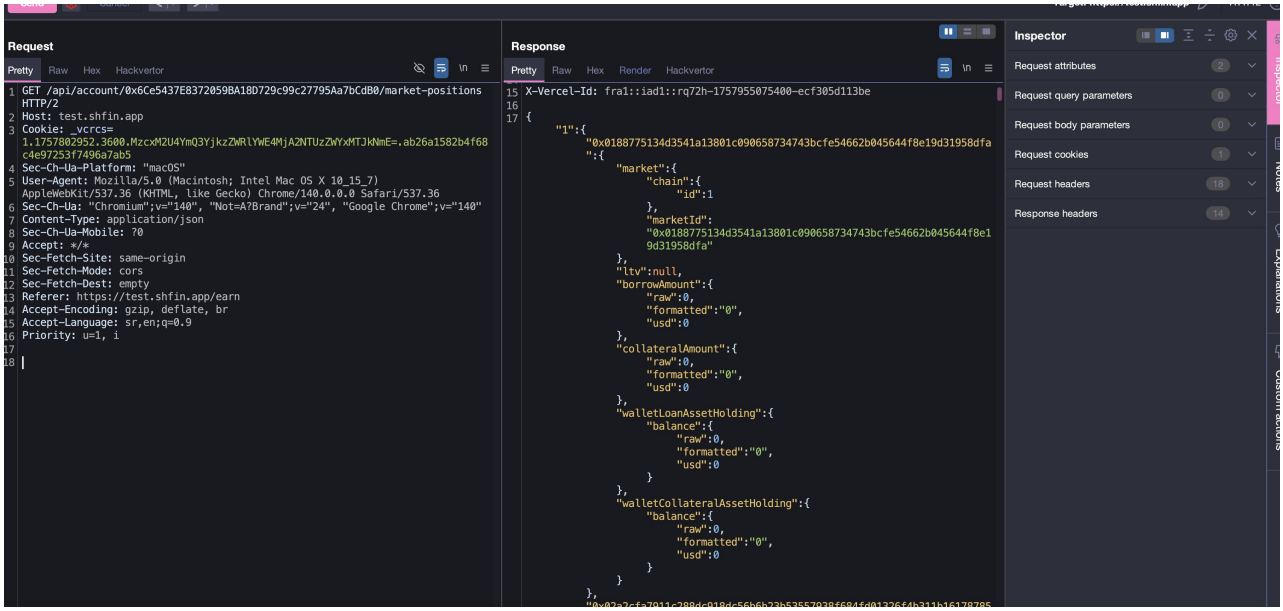
Impact: Low

Proof of Concept: The following client-side code restricts UI access based on wallet connection but does not prevent unauthenticated API requests:

```
export function useMarketPositions() {
  const { address } = useAccount();
  return useQuery({
    queryKey: ["market-positions", address],
    queryFn: async () =&gt;
    ↪  fetchJsonResponse&lt;MarketPositionMap&gt;(`/api/account/${address}/market-positions`),
    enabled: !!address,
  });
}
```

This restriction is bypassed by sending a direct HTTP request, with or without wallet connection:

```
GET /api/account/0x6Ce5437E8372059BA18D729c99c27795Aa7bCdB0/market-positions HTTP/2
Host: test.shfin.app
```



Recommendation:

- Implement server-side validation to ensure that only authenticated users with a connected wallet can access the `/api/account/[address]/market-positions` endpoint.
- Enforce authorization checks to ensure that a user can only query their own account's market positions, unless there is a business requirement for broader access.
- Rate limit or throttle requests to the API endpoint to prevent automated enumeration or abuse.
- Monitor backend usage patterns for spikes in traffic to the market positions endpoint that could indicate abuse or excessive usage.
- Review internal API proxying logic to ensure only necessary data is exposed and backend calls are minimized.
- Refer to the OWASP API Security Top 10, specifically API2:2019 Broken User Authentication and API4:2019 Lack of Resources & Rate Limiting, for further guidance.

Steakhouse: Acknowledged.

Spearbit: Acknowledged.

5.1.7 Missing API-wide Rate Limiting – Potential Whisk API Token Exhaustion

Severity: Low Risk

Context: *(No context files were provided by the reviewers)*

Description: The application does not enforce API-wide rate limiting on endpoints that internally proxy requests to the Whisk GraphQL API. This allows both unauthenticated and automated users to send large volumes of requests to backend API endpoints, each of which consumes Whisk API usage. For example, the endpoint:

```
GET /api/account/{address}/market-positions
```

can be programmatically spammed with any wallet address, triggering backend requests to the Whisk API. As no authentication or throttling is required, malicious users can rapidly generate thousands of backend calls, either targeting a single address or rotating through many.

The Whisk API's pricing model and quota limits are not publicly disclosed, but it is standard for such API integrations to be subject to usage quotas, rate limits, or pay-per-request models. Abuse of this weakness could result in:

- Depletion of prepaid or allocated API usage (token or credit exhaustion)
- Unexpected operational costs for excessive API usage
- Temporary service disruptions or denial of service if the Whisk API key is rate limited, suspended, or exhausted

While the returned data is public on-chain information, the impact is a business risk from infrastructure cost and loss of service, rather than a data confidentiality issue.

Lack of backend rate limiting may enable attackers to cause direct financial loss and operational risk by exhausting Whisk API quotas or incurring excess usage fees.

Proof of Concept: Automated scripts can repeatedly trigger backend API calls. For example:

```python
import requests

address = "0x6Ce5437E8372059BA18D729c99c27795Aa7bCdB0"
url = f"https://test.shfin.app/api/account/{address}/market-positions"

for _ in range(10000):
    requests.get(url)
```

Each request to `/api/account/{address}/market-positions` will initiate a backend call to the Whisk API, quickly consuming API token usage or funds.

Recommendation:

- Implement robust, API-wide rate limiting on all endpoints that proxy or call external APIs such as Whisk.
- Enforce per-IP and per-account request limits using middleware (e.g., express-rate-limit) or at the API gateway.
- Apply stricter thresholds to unauthenticated requests, and consider requiring authentication for endpoints that initiate costly backend operations.
- Use circuit breakers to temporarily disable backend calls if usage patterns exceed safe thresholds.
- Monitor backend usage and establish alerting for abnormal API consumption or cost spikes.
- Review your API integration with Whisk to understand contractual limits, and ensure fallback procedures are in place.
- Refer to OWASP API Security Top 10, specifically API4:2019 Lack of Resources & Rate Limiting, and OWASP Rate Limiting Cheat Sheet for further best practices.

Steakhouse: Fixed in PR 6.

Spearbit: Fix verified. Rate limiting added with a 100 requests per 10 second window. This has been added to the middleware to prevent excessive requests being sent as flagged in the review.