Least Authority
PRIVACY MATTERS

Core Web

Security Audit Report

Ava Labs

Updated Final Audit Report: 21 May 2024

Table of Contents

Overview

Background

Ava Labs has requested that Least Authority perform a security audit of their Core Web – a web application for the all-in-one Web3 command center for Avalanche.

Project Dates

- **August 21, 2023 - September 4, 2023**: Initial Code Review *(Completed)*
- **September 6, 2023**: Delivery of Initial Audit Report *(Completed)*
- **May 5, 2024 - May 10 2024 :** Verification Review *(Completed)*
- **May 10, 2024 :** Delivery of Final Audit Report *(Completed)*
- **May 21, 2024 :** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Shareef Maher Dweikat, Security Research and Engineer
- Nikos Iliakis, Security Researcher and Engineer
- Xenofon Mitakidis, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Core Web followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Core Web:
  https://github.com/ava-labs/core-web-properties

Specifically, we examined the Git revision for our initial review:

- `a1919d5324b3b51b138b923dab6a96f6c88348c1`

For the review, this repository was cloned for use during the audit and for reference in this report:

- Core Web:
  https://github.com/LeastAuthority/avalabs-core-web-audit/tree/develop

For the verification, we examined the Git revision:

- `075ec3a960faa934e7bbf6c7c614db8ea8063383`

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third-party code, unless specifically mentioned as in scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Build instructions:
  https://github.com/ava-labs/core-web-properties/tree/main/packages/web

In addition, this audit report references the following documents:

- TSDoc Specification:
  https://tsdoc.org
- You Might Not Need an Effect:
  https://react.dev/learn/you-might-not-need-an-effect

Areas of Concern

Our investigation focused on the following areas:

- Prevention of leakage of sensitive data to Posthog, Sentry, or other external services; and
- Protection from Cross-Site Scripting (XSS) attacks to prevent malicious information from misleading users, and to prevent an attacker modifying the address, amount, or other data sent to browser extension wallets for signing and sending transactions:
  - Including investigating the usage of the shadow dom as an effective measure to prevent attackers from modifying the crypto address displayed in the UI.

Additionally, the following are areas of concern that were investigated during the audit, along with any similar potential issues:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adversarial actions and other attacks;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Denial of Service attacks and security exploits that would impact or disrupt execution;
- Exposure of any critical information during interaction with any external libraries;
- Protection against malicious attacks and other methods of exploitation;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security review of Ava Labs' Core Web application, a command center for navigating Web3 across ecosystems. It offers streamlining user interactions with dApps, NFTs, and various blockchain protocols and simplifies asset management and on-chain activities.

In addition to performing a manual code review, our team investigated vulnerabilities and areas of concern that could affect the security features of the application. In our review, special emphasis was placed on checking all user input fields in the application, in addition to the information provided to the application to identify possible vulnerabilities, such as Cross-Site Scripting (XSS) attacks.

System Design

Our team identified a pattern of insufficient input sanitization, which could lead to unintended behavior. We recommend refraining from trusting data coming from external resources, in addition to properly sanitizing functions handling this data (Issue B).

Our team also identified a possible exploit whereby the user is directed to a malicious website that is a copy of the Core Web application tab and asks the user to authenticate. If the user provides credentials, the attacker could compromise the user account. We recommend a parameter configuration to prevent this vulnerability (Issue A).

Code Quality

We performed a manual review of the repositories in scope and found that the Core Web implementation is generally well-organized and adheres to coding principles. However, we identified some bugs related to the use of React and TypeScript, which we recommend be resolved (Suggestion 6, Suggestion 8). Our team also found that the implementation deviates from React best practices, as the React built-in hooks, `useEffect`, `useState`, and `useMemo` are used incorrectly, which can impact the overall performance of the dApp.

Additionally, we identified several suggestions that would improve the quality of the code and contribute to the overall security of the implementation, if addressed (Suggestion 3, Suggestion 5, Suggestion 9, Suggestion 10).

Tests

Our team found the test coverage of the repositories in scope to be insufficient. We recommend that test coverage be improved (Suggestion 7).

Documentation

The tutorials posted on the Ava Labs website, in addition to the medium article, offered a helpful description of the application and different use cases, which facilitates a comprehensive understanding of the system and helps onboard new reviewers and users.

Code Comments

We found that the implementation is sparsely commented, with code comments generally consisting of internal notes shared between the development teams. We recommend adhering to the TSDoc Specification to improve code comments (Suggestion 9).

Scope

The scope of this review was generally sufficient and included all security-critical components.

Dependencies

We examined all the dependencies implemented in the codebase and identified one vulnerable dependency, one outdated dependency, and two unused dependencies. We recommend improving dependency management (Suggestion 4).

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Misuse of Window.open Makes the dApp Vulnerable To Reverse Tabnabbing Attacks	Resolved

Issue B: Missing Sanitization on Inputs Makes the dApp Vulnerable to XSS Attacks	Resolved
Issue C: Imprecise Calculations	Resolved
Suggestion 1: Add Checks for Every Website Included in the Project	Resolved
Suggestion 2: Add Source Code for ABIs	Planned
Suggestion 3: Do Not Hard Code Strings	Resolved
Suggestion 4: Improve Dependency Management	Resolved
Suggestion 5: Resolve All TODOs in the Codebase	Planned
Suggestion 6: Refactor Type Intersections	Planned
Suggestion 7: Improve Test Coverage	Planned
Suggestion 8: Simplify Type Syntax	Resolved
Suggestion 9: Adhere to the TSDoc Specification	Planned
Suggestion 10: Adhere to React Best Practices	Planned

Issue A: Misuse of Window.open Makes the dApp Vulnerable To Reverse Tabnabbing Attacks

Location

`src/utils/share.ts#L12C1-L12C1`

`components/UniversalSearch/UniversalSearch.tsx#L196`

`src/hooks/useOpenExplorerTab.ts#L26`

Synopsis

The current implementation of the dApp uses `window.open` to access external websites. If an attacker injects malicious code into that website, they can rewrite the source page and substitute it with a phishing website. This type of attack is known as reverse tabnabbing.

Impact

With a reverse tabnabbing attack, the user who initially opens the original and correct webpage is unlikely to notice the webpage has been replaced with the phishing website. As a result, the user will likely share sensitive data with the phishing website, or connect their wallets to it. This could lead to a complete loss of funds or sensitive/private data leakage.

Preconditions

There are two preconditions that must be met for the attack to be successful:

1. The page that is linked from the target must contain malicious code that utilizes the location property in the `opener` object sent by the the `window.open` function, as follows:

   ```
   <script> if (window.opener) { window.opener.location =
   "https://www.phishingsite.ps"; } </script>
   ```

2. `noopener`, `noreferrer` should not be passed to the `window.open` function in the dApp.

Feasibility

If the preconditions are met, the attack is trivial.

Remediation

We recommend, for all instances of `window.open`, that the `noopener`, `noreferrer` parameter be passed as a third parameter for the `window.open` function, as shown below:

```
window.open(url, target, 'noreferrer, noopener');
```

Status

The Ava Labs team has introduced the `openWindowSecurely` function to the codebase, which follows secure properties and uses them to access external sites.

Verification

Resolved.

Issue B: Missing Sanitization on Inputs Makes the dApp Vulnerable to XSS Attacks

Location

Example (non-exhaustive):

`utils/search/contentfulCreateFuseSearch.ts#L11`

`utils/generateOpenApiClient/generateOpenApiClient.ts#L64`

`components/UniversalSearch/UniversalSearch.tsx#L213`

`services/dapp/dappService.ts`

Synopsis

Inputs of the system, such as text fields, API responses, data coming from the smart contract or the connected wallet, and fs package reading files stored on the server are not sanitized. Data coming from an external resource should not be trusted, as it could carry malicious codes that can affect the safety of the system.

Impact

Receiving data from untrusted sources opens an attack vector for different kinds of Cross Site Scripting (XSS) attacks.

Preconditions

A malicious actor would have to take over the server or inject malicious code in it.

Remediation

We recommend that the Ava Labs team sanitize and validate data coming from untrusted sources to verify that it does not contain any malicious codes that could affect the system, and confirm that the data obtained aligns with their expectations. We suggest using the `DomPurify` package for sanitization.

Status

The Ava Labs team has added input sanitization in places that are not already sanitized by `React/JSX`.

Verification

Resolved.

Issue C: Imprecise Calculations

Location

`src/utils/calculatePercentageChange.ts#L1`

Synopsis

The utility function calculates its return value by subtracting variables of the number `type`.

Impact

Since this is a utility function that may be used in sensitive calculations where precision is paramount, this could lead to incorrect calculations due to precision errors.

Remediation

We recommend using the `toPrecision` or the `BigInt` library.

Status

The Ava Labs team has started using the big data type from the `Big.hs` library to perform calculations.

Verification

Resolved.

Suggestions

Suggestion 1: Add Checks for Every Website Included in the Project

Location

`Discover/SubmitProject/SubmitProject.tsx#153`

Application: `https://core.app/discover/my-projects/submit/Synopsis`

Synopsis

Input fields that accept websites in the form allow http URLs to be inserted. Http websites are vulnerable to interception, lack encryption, and can be subject to man-in-the-middle attacks. Users of Core Web and its projects could be susceptible to these attacks, which could affect the reputation of the platform. Moreover, users are allowed to enter any website URL in the input field that is meant for X (formerly

Twitter) only. Although Ava Labs appears to conduct manual reviews of projects, this method is highly susceptible to human error.

Mitigation

We recommend adding checks for the websites included in each project.

Status

The Ava Labs team has started using the `zod` library for schema validation.

Verification

Resolved.

Suggestion 2: Add Source Code for ABIs

Location

`constants/src/abis`

Synopsis

ABIs are implemented and used in the application. However, there is no way to verify the ABI.

Mitigation

We recommend adding a link in the source code of the ABI that points to the correct source code, such that a future maintainer, for example, would be able to verify it easily. This would also provide more information to the developers (e.g. about the compiler version).

Status

The Ava Labs team stated that they intend to implement the recommended mitigation in the future.

Verification

Planned.

Suggestion 3: Do Not Hard Code Strings

Location

Examples (non-exhaustive):

`src/utils/tokenUtils.ts#L6`

`src/utils/tokenUtils.test.ts#L9`

`StakingCenter/components/StakingDuration.tsx#L87`

Synopsis

Some constants are hard coded, such as addresses and strings. Constants that are hard coded in multiple locations can result in mistakes during development, leading to different values throughout the codebase.

Mitigation

Important strings should be accessed from one source of truth in the codebase and should not be scattered throughout the codebase and UI tags.

Status
The Ava Labs team has introduced global constants.

Verification
Resolved.

Suggestion 4: Improve Dependency Management

Location
`apps/web/package.json`

Synopsis
Running the `pnpm audit` command shows one vulnerable dependency (`@openzeppelin/contracts`). Running the `npx depcheck` command shows two unused dependencies (`@emotion/css` and `@core/prerender-cli`). Running `pnpm outdated` command shows one outdated dependency (`@emotion/css`).

Mitigation
We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the AvalancheJS V2 library and to mitigate supply-chain attacks, which includes:

- Manually reviewing and assessing currently used dependencies;
- Upgrading dependencies with known vulnerabilities to patched versions with fixes;
- Replacing unmaintained dependencies with secure and battle-tested alternatives, if possible;
- Pinning dependencies to specific versions, including pinning build-level dependencies in the `package.json` file to a specific version;
- Only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities; and
- Including automated dependency auditing reports in the project's CI/CD workflow.

Status
The Ava Labs team has integrated `Dependabot` into the Git repository to enhance dependency management.

Verification
Resolved.

Suggestion 5: Resolve All TODOs in the Codebase

Location
`pages/Swap/Swap.tsx#L82`

`hooks/bridge/useTransferAsset.ts#L12`

Synopsis
Our team identified several instances of unresolved TODOs. Unresolved TODOs decrease code readability and may create confusion about the completeness of the protocol and the intended functionality of each of the system components. This can hinder the ability for security researchers to identify implementation errors.

Mitigation
We recommend identifying and resolving all pending TODOs in the codebase.

Status
The Ava Labs team resolved the TODO in `useTransferAsset` and noted that they intend to resolve the TODO in `Swap.tsx` in the future.

Verification
Planned.

Suggestion 6: Refactor Type Intersections

Location
`components/LazyLoadDialog/LazyLoadDialog.tsx#L5`

`wallet/hooks/useAtomicUtxos.ts#L8`

`wallet/hooks/useSendImport.ts#L18`

Synopsis
The intersections either add no features or resolve to `any`.

Mitigation
We recommend simplifying the intersections to allow only the desired type features.

Status
The Ava Labs team stated that they consider this to be a low priority task and, hence, they intend to implement the recommended mitigation in the future.

Verification
Planned.

Suggestion 7: Improve Test Coverage

Location
`packages`

apps/web

Synopsis
There is insufficient test coverage implemented to test the correctness of the implementation and that the system behaves as expected. A robust test suite improves the quality of the code by providing a mechanism through which new developers and security researchers gain an understanding of how the code works, as well as a means to interact with them. In addition, tests build confidence that the code works as intended for known use cases.

Mitigation
We recommend that comprehensive unit test coverage be implemented in order to check all success, failure, and edge cases and to verify that the implementation behaves as expected.

Status
The Ava Labs team stated that they plan on improving test coverage in the future.

Verification
Planned.

Suggestion 8: Simplify Type Syntax

Location
[providers/CurrenciesProvider/CurrenciesProvider.tsx#L9]

Synopsis
Both the ? specifier and the `undefined` type are implemented, which is redundant.

Mitigation
We recommend making the syntax explicit by removing the optional property syntax.

Status
The Ava Labs team has implemented the recommended syntax changes.

Verification
Resolved.

Suggestion 9: Adhere to the TSDoc Specification

Synopsis
The codebase has minimal comments that will make the code more difficult to maintain in the long run.

Mitigation
We recommend adhering to the TSDoc specification, as this would simultaneously result in code that is sufficiently documented, and also make it possible to use comment parsing tools.

Status
The Ava Labs team stated that they intend to implement the recommended mitigation during the development of tasks.

Verification
Planned.

Suggestion 10: Adhere to React Best Practices

Location
`pages/Chat/Chat.tsx#L77`

`providers/BridgeProvider/BridgeProvider.tsx#L57`

Synopsis

Our team identified several instances where `useEffect`, `useState`, and `useMemo` are misused. Although this does not lead to security implications, it might affect the performance of the dApp.

Mitigation

We recommend adhering to React best practices to prevent such cases.

Status

The Ava Labs team stated that they intend to implement the recommended mitigation in the future.

Verification

Planned.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.