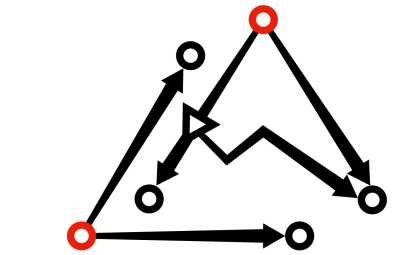

Least Authority

PRIVACY MATTERS

AvalancheJS V2

Security Audit Report

Ava Labs

Final Audit Report: 26 March 2024

Table of Contents

Overview

Background

Ava Labs has requested that Least Authority perform a security audit of their AvalancheJS V2, a JavaScript Library for interfacing with the Avalanche Platform.

Project Dates

- **July 31, 2023 - August 16, 2023:** Initial Code Review *(Completed)*
- **August 18, 2023:** Delivery of Initial Audit Report *(Completed)*
- **March 26,2024:** Verification Review *(Completed)*
- **March 26,2024:** Delivery of Final Audit Report *(Completed)*

Review Team

- Shareef Maher Dweikat, Security Research and Engineer
- Steven Jung, 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 AvalancheJS V2 followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository is considered in scope for the review:

- AvalancheJS V2:
  https://github.com/ava-labs/avalanchejs-v2

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

- `2f80667a3f81721f5296de6699ea9c7679bbff2c`

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

- AvalancheJS V2:
  https://github.com/LeastAuthority/ava-labs-Avalanchejs-v2-audit

For the verification, we examined the Git revision:

- 003e230cfe46af29c74e1d234121bf76d5117307

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:

- N/A

In addition, this audit report references the following document:

- OWASP Guidelines: https://cheatsheetseries.owasp.org/cheatsheets/Key_Management_Cheat_Sheet.html

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Malicious attacks and security exploits;
- Vulnerabilities in the code and whether the interaction between the related components is secure;
- Exposure of any critical or sensitive information during user interactions with the SDK and use of external libraries and dependencies;
- Proper management of encryption and storage of private keys;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team conducted a security audit of the AvalancheJS SDK, which handles transaction creation, importing and exporting, and state storage features for signed or unsigned transactions that are consumed by other applications. We investigated the design of the SDK and its coded implementation to identify security vulnerabilities, in addition to examining input validation implementation, error handling, as well as general implementation security relating to the use of third-party dependencies and secure coding and best practices.

We reviewed the state transition mechanism – that allows the transaction data to be passed to the SDK – for a variety of attack vectors, including possible ways to tamper or manipulate the transaction data, and did not identify any vulnerabilities. Furthermore, we investigated the correct handling of transaction data and the correctness of calculations for burned and staked amounts and could not identify any issues.

Code Quality

In our manual review of the AvalancheJS V2 source code, our team found that the SDK code is well-organized and easily understood. However, we identified several areas of improvement that would contribute to the overall security of the system, as detailed below.

We identified an instance of a missing grouping for an operation, which can lead to imprecise calculations (Issue A). Our team also found that the unencrypted private keys are unnecessarily stored in memory (Suggestion 1).

Additionally, we found instances where inputs are processed without any validation or sanitation, which could lead to unexpected behavior. We recommend that all inputs be validated and sanitized appropriately (Suggestion 4, Suggestion 5, Suggestion 7). Our team also found that errors are not handled appropriately (Suggestion 2).

Tests

The AvalancheJS V2 has implemented sufficient unit tests, which showed high coverage of important logic.

Documentation & Code Comments

The project documentation provided for this security review was insufficient in describing the general architecture of the system, each of the components, and how those components interact with each other. Furthermore, the codebase lacks comments in some areas. This reduces the readability of the code and, as a result, makes reasoning about the security of the system more difficult. We recommend that the project documentation and code comments be improved (Suggestion 8, Suggestion 9).

Scope

The scope of this review was sufficient and included all security-critical components relating to the transactions that are retrieved from the ledger and then exported to be consumed by other applications.

Dependencies

We examined all the dependencies implemented in the codebase and identified several instances of unused dependencies, and one outdated dependency. We recommend improving dependency management (Suggestion 3).

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: Missing Grouping for an Operation With Lower Precedence	Resolved
Suggestion 1: Do Not Store Private Key In Memory Unencrypted	Resolved
Suggestion 2: Handle Errors Gracefully	Unresolved
Suggestion 3: Improve Dependency Management	Partially Resolved
Suggestion 4: Validate the Input of the Delegate Function	Unresolved
Suggestion 5: Validate the Input of the HexToBuffer Function	Resolved
Suggestion 6: Do Not Hard Code Strings	Resolved
Suggestion 7: Validate and Sanitize API Response	Unresolved
Suggestion 8: Improve Documentation	Partially Resolved
Suggestion 9: Improve Code Comments	Resolved

Issue A: Missing Grouping for an Operation With Lower Precedence

Location

src/utils/baseTXBurnedAmount.ts#L8

Synopsis

The - operator has a higher priority than the ?? operator. Consequently, `agg[assetId]` will be set to NaN when `outAmount[assetId]` is `undefined`.

Impact

Users may calculate the burned amount using the `baseTxBurnedAmount` function.
If an asset is burnt completely in a transaction, an output amount would not be generated for the asset. In this case, the user receives NaN instead of the exact burned amount, which can lead to unexpected results.

Remediation

We recommend grouping operations with lower precedence, as follows:

```
agg[assetId] = amt - (outAmount[assetId] ?? 0n);
```

Status

The Ava Labs team has deleted `src/utils/baseTXBurnedAmount.ts`, as it is currently not being used.

Verification

Resolved.

Suggestions

Suggestion 1: Do Not Store Private Key In Memory Unencrypted

Location

src/signer/keychain.ts#L13

examples/x-chain/export.ts#L36

examples/x-chain/export.ts#L19

Synopsis

The private key is stored unencrypted in memory, which is not consistent with OWASP's best practices and guidelines for key storage.

Furthermore, in the case of AvalancheJS V2, our team did not identify a real need to store the private key in memory. Instead, the signing function can receive the private key as a parameter when a signature is needed, sign the transaction, then delete the received key, as shown below:

```
const keyChain = new Secp256K1Keychain();

await keyChain.addSignatures(tx, hexToBuffer(privateKey));
```

In the aforementioned lines of code, the transaction is passed along with the private key, the function signs the transaction and returns it, and then the function, along with its data, are terminated. Hence, private keys do not need to be stored.

Mitigation

We recommend refactoring the code to sign transactions without storing the private key.

Status

The Ava Labs team has removed the partial `Secp256K1Keychain` implementation from the library and replaced it with the stateless util function `src/signer/addTxSignatures`, which does not keep a copy of the private keys after signing.

Verification

Resolved.

Suggestion 2: Handle Errors Gracefully

Location

Examples (non-exhaustive):

`utils/calculateSpend/calculateSpend.ts#L74`

`src/utils/address.ts#L11`

`vms/common/rpc.ts#L54`

Synopsis

There are multiple instances in the code that would trigger an error. Functions that can cause the code to crash at runtime may lead to denial of service.

Mitigation

We recommend refactoring the code and removing `throw new Error` where possible. One of the possible improvements is to propagate errors to the caller and handle them on the upper layers. Error handling does not exclude using `throw new Error`. In addition, if a caller can return an error, the callee function may not trigger `throw new Error` but, instead, propagate an error to the caller.

Status

The Ava Labs team acknowledged the suggestion but stated that this pattern is an internal design choice of the library, and users must therefore – for the time being – be mindful about implementing proper error handling on their end.

Verification

Unresolved.

Suggestion 3: Improve Dependency Management

Location

`package.json`

Synopsis

Our team found that the outdated dependency `@noble/secp256k` is utilized. In addition, running the `depcheck` command shows a large number of unused dependencies.

Using outdated or vulnerable dependencies exposes the system to attacks that could result in the exfiltration of sensitive data. Furthermore every dependency (whether used or unused) and line of code added to the project can potentially increase the wallet's attack surface. Therefore, only used dependencies should be kept in the `package.json` file.

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 has partially resolved this mitigation by updating some dependencies. However, our team found that the outdated `@noble/secp256k` is still in use.

Verification

Partially Resolved.

Suggestion 4: Validate the Input of the Delegate Function

Location

`vms/pvm/builder.ts#L282`

Synopsis

The input of the `delegate` function is not being validated. This could cause the transactions to fail, thus resulting in the unnecessary spending of gas fees.

Mitigation

We recommend validating the input of the `newAddDelegatorTx` function.

Status

The Ava Labs team has not addressed this suggestion. As such, it remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 5: Validate the Input of the HexToBuffer Function

Location

`src/utils/buffer.ts#L21`

Synopsis

If an invalid hexadecimal input is passed to the function, it will either result in unexpected behavior or throw an error that would potentially crash the application if the error is not caught.

Mitigation

We recommend validating `hexToBuffer` function inputs to only allow valid hexadecimals, in addition to cashing all errors properly, as explained in Suggestion 2.

Status

The Ava Labs team stated that `HexToBuffer` is a considerably small wrapper around `@noble/hashes/utils`, which does the validation. Our team agrees with the development team's response and thus considers this suggestion resolved.

Verification

Resolved.

Suggestion 6: Do Not Hard Code Strings

Location

`src/fixtures/common.ts#L35`

`src/fixtures/common.ts#L25`

Synopsis

Some constants are hard coded. In general, constants that are hard coded in multiple locations can lead to mistakes during development, leading to different values throughout the codebase. Additionally, such constants increase the file size of the code and reduce its readability.

Mitigation

Strings should be accessed from one source of truth in the codebase and should not be scattered across the code. We recommend creating a `strings` file to export strings for usage.

Status

The Ava Labs team stated that the noted strings are hard coded in fixture files used for unit testing and do not end up in the released bundle. Our team agrees with the development team's response and thus considers this suggestion resolved.

Verification

Resolved.

Suggestion 7: Validate and Sanitize API Response

Location

`vms/common/rpc.ts#L41`

Synopsis

`callMethod` sends `Post` requests to the endpoint it receives. The returned response is stored in `resp object` then returned to the caller of the method without any validation or sanitation. However, data coming from an external resource should not be trusted, as it could carry malicious codes that can affect the safety of the system.

Mitigation

We recommend that the Ava Labs team sanitize and validate the response 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.

Status

The Ava Labs team acknowledged the suggestion however, it remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 8: Improve Documentation

Synopsis

The general documentation provided by the Ava Labs team was minimal. Robust and comprehensive documentation allows a security team to assess the in-scope components and understand the expected behavior of the system being audited.

Mitigation

We recommend that the Ava Labs team improve the project's general documentation by creating a high-level description of the system, each of the components, and the interactions between those components. This can include developer documentation and architectural diagrams.

Status

The Ava Labs team has partially improved the project documentation by updating `README.md`, the examples in the `examples` folder, as well as the article on `docs.avax.network`. The team further added that additional documentation is underway.

Verification

Partially Resolved.

Suggestion 9: Improve Code Comments

Synopsis

Our team found that there are minimal code comments outlining the calculation and handling of the transactions signed and describing the Keychain component. This inhibits code clarity and readability and increases the likelihood of potential errors that may lead to security vulnerabilities.

Mitigation

We recommend expanding and improving the code comments within the components to facilitate reasoning about the security properties of the system.

Status

The Ava Labs team has removed the Keychain component. Hence, we consider this suggestion resolved.

Verification

Resolved.

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.