Least Authority
PRIVACY MATTERS

Subnet EVM
Security Audit Report

Ava Labs

Final Audit Report: 03 April 2023

Table of Contents

Overview

Background

Ava Labs has requested that Least Authority perform a security audit of their Subnet EVM.

Project Dates

- **January 2, 2022 - February 22, 2022**: Code Review *(Completed)*
- **February 24, 2022**: Delivery of Initial Audit Report *(Completed)*
- **31 March:** Verification Review *(Completed)*
- **03 April:** Delivery of Final Audit Report *(Completed)*

Review Team

- Alicia Blackett, Security Researcher and Engineer
- DK, Security Researcher and Engineer
- Dylan Lott, Security Researcher and Engineer
- Xenofon Mitakidis, Security Researcher and Engineer
- ElHassan Wanas, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Subnet EVM 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:

- Subnet EVM:
  https://github.com/ava-labs/subnet-evm

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

`0fb0afe8a38daf13880e8d3c0fc43b4edc74c80e`

For the verification, we examined the Git revision:

`737e9c42cb5f02ffead0a4fd4a7970d8c3eb3e68`

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

- Subnet EVM:
  https://github.com/LeastAuthority/Ava-Labs-Subnet-EVM

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:

- Customize Your EVM-Powered Subnet | Avalanche Docs: https://docs.avax.network/subnets/customize-a-subnet

In addition, this audit report references the following documents:

- Source file `src/time/tick.go`: https://go.dev/src/time/tick.go
- Regular expression Denial of Service (ReDoS): https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS
- func `SetFinalizer`: https://pkg.go.dev/runtime#SetFinalizer

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Denial of Service (DoS) attacks;
- Attacks intending to misuse resources, cause unintended forks, and create unwanted or adversarial chains;
- Any attack that impacts funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Network attacks, which include the flooding or misuse of data, causing inappropriate taxing;
- Proper management of encryption and storage of private keys, including the key derivation process;
- Exposure of any critical information during user interactions with the blockchain and external libraries;
- Any potential attacks with a high Return on Investment (ROI);
- General use of external libraries;
- Vulnerabilities in the code and whether the interactions between the related and network components are secure;
- Inappropriate permissions, ambient, and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Subnet EVM is intended to allow users to create private chains (Subnets) running on the Ethereum Virtual Machine (EVM). Deploying an instance of Subnet EVM enables owners to leverage the functionality of traditional EVM while allowing more customization options for advanced use cases.

Subnets employ precompile contracts, which can be used to implement primitives on the Subnet after deployment, such as allowing chain creators to determine their own privacy through configurable access control, or incentivizing Platform Chain (P-Chain) validators to participate in a Subnet by configuring the reward scheme.

The Avalanche CLI allows a user to define a contract interface in Solidity, and then use the built-in `precompilegen` CLI tool to generate the Go method stubs for implementing that Solidity contract in Go rather than Solidity.

Precompile Contracts

Subnets have a base set of precompiles that are included by default for each created Subnet environment. The base set of precompile contracts inherits from two core contracts, `AllowListConfig` and `UpgradeableConfig`, which act as a simple set of permissions and contract upgrade features.

Our team examined precompiles for network vulnerabilities, consensus attacks, and smart contract vulnerabilities such as reentrancy, and could not identify any security issues. We also examined the precompile development process and lifecycle, including incorrect permissions or access controls, undefined behavior, and denial of service attacks. Additionally, we considered how precompiles related to network processes and the chain execution environment.

To supplement our team's manual code review, we performed static analysis and fuzzing of the precompile contracts to identify instances of unexpected behavior, unlocked mutex, unhandled errors, and others. The tools developed for, and the results of, the fuzzing tests will be shared with the Subnet EVM team, and our team recommends continued testing.

Our team found that the Subnet EVM implementation has been designed and implemented with consideration made for the security of the system. Our team identified implementation errors that could lead to undefined behavior or denial of service. We also identified suggestions to improve the overall security and quality of the implementation.

System Design

In examining the design of Subnet EVM, our team focused on the modifications made to go-ethereum and how those could be leveraged for an attack. Although our team did not find any issues in the design of Subnet EVM, we found that a pattern of insufficient error handling in the implementation could lead to unexpected behavior, logic errors, or denial of service (Suggestion 9). Our team also found that the insufficient zeroization of sensitive data could lead to leakage of this data, and we recommend clearing this data from memory after use (Suggestion 10). In addition, we identified a pattern of inconsistent checks on the assumptions made in precompiles (Suggestion 11).

Code Quality

Our team performed a manual review of the codebase and identified implementation errors that lead to security vulnerabilities (Issue A, Issue B, Issue C). We found that although the codebase is structured very similarly to go-ethereum, the changes to the fork are not as organized as the original implementation. Our team noted a pattern of excessive use of locks where best practice recommends using channels over sharing state.

Tests

Our team examined the tests and all assertions for soundness and did not identify any issues. Although we found the test coverage to be generally sufficient, we identified some functions that had insufficient testing. We recommend writing a test suite that defines the behavior of the peer tracker more rigorously (Suggestion 3), and tests for newly implemented functions (Suggestion 4).

Documentation

The project documentation provided by the Subnet EVM team was generally accurate and helpful in explaining the functional description of the workings of the Subnets. However, the documentation could be further improved by adding more detailed descriptions for creating and deploying a Subnet.

Our team identified several missing flags in the `run script` documentation and required further support to run a Subnet. Furthermore, the documentation would benefit from an additional description of changes from the original go-ethereum implementation, such as the peer package (Suggestion 2).

Code Comments

We found code comments to be sufficient. However, we identified several areas where code comments can be improved. In particular, the implementation would benefit from additional file and package level comments explaining their interactions with the rest of the system.

Scope

Our team found that the scope of this security review was generally sufficient. However, given the reliance of Subnet EVM peers on Avalanche's `avalanchego/ids` implementation for peer discovery, we recommend that this component be audited by an independent team familiar with Subnet EVM.

Dependencies

Our team did not identify any issues in the use of dependencies.

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: Locked Mutex on Return From Functions	Resolved
Issue B: Tickers Leak	Resolved
Issue C: Out of Range Index Causes a Potential Panic	Resolved
Suggestion 1: Update precompilegen Template	Resolved
Suggestion 2: Improve Documentation	Partially Resolved
Suggestion 3: Define the Peer Tracker With a Test Suite	Resolved
Suggestion 4: Test Additional Functions	Resolved
Suggestion 5: Review the Regular Expression With ReDoS Pattern	Resolved
Suggestion 6: Disable Writing to Files For Non Owners	Resolved
Suggestion 7: Check the Size of the Slice When Accessing a Slice Element	Resolved
Suggestion 8: Implement Input Validation for the blockcount Parameter	Resolved
Suggestion 9: Improve Error Handling	Resolved
Suggestion 10: Clear Sensitive Cryptographic Keys	Unresolved

Suggestion 11: Validate Assumptions in Precompiles	Resolved

Issue A: Locked Mutex on Return From Functions

Location

`state/snapshot/snapshot.go#L600-L603`

`trie/database.go#L700-L707`

Synopsis

The referenced function does not unlock the mutex when returning.

Impact

Incorrect implementation of concurrent access to shared objects can lead to undefined behavior or be exploited for a denial of service attack.

Mitigation

We recommend unlocking the mutex using a `defer` statement, if possible. Otherwise, we recommend verifying that all mutexes are unlocked along all control-flow paths.

Status

The Ava Labs team has mitigated the issue by unlocking the mutexes.

Verification

Resolved.

Issue B: Tickers Leak

Location

`plugin/evm/gossiper.go#L240-L244`

Synopsis

The function `awaitEthTxGossipA` creates three tickers but does not stop them when it finishes. As a result, the allocated resources leak.

Impact

The allocated resources for tickers will leak. This could be exploited for a denial of service attack.

Mitigation

We recommend using the `Stop` function to stop a ticker.

Status

The Ava Labs team has mitigated the issue as suggested.

Verification

Resolved.

Issue C: Out of Range Index Causes a Potential Panic

Location

`internal/ethapi/api.go#L1717`

Synopsis

A panic can be triggered in the `GetTransactionReceipt` function. If the value of the `index` variable is equal, for example, to `(1<<64)-1` (max uint for 64-bit system) or `(1<<32)-1` (max uint for 32-bit system) then that `index` is cast to the `int` type, its value changes to `-1,` and the execution will not be canceled on the `if` statement. After that, a panic will be triggered upon accessing a slice element with a negative index.

Impact

Triggering a panic can be exploited for a denial of service attack.

Mitigation

We recommend checking that the result of casting to `int` type is a positive number.

Status

The Ava Labs team has added the check as suggested.

Verification

Resolved.

Suggestions

Suggestion 1: Update Precompilegen Template

Synopsis

The Go code generated using the Precompilegen tool does not compile due to a missing import for the JSON package.

Mitigation

We recommend adding an `import"json"` line to the precompile template when using the Precompilegen tool.

Status

The precompile generator templates were refactored and don't require the json package or any other package that is missing after generating the precompile.

Verification

Resolved

Suggestion 2: Improve Documentation

Synopsis

The project documentation provided for this review can be further improved by including more accurate instructions for creating and deploying Subnets. Similarly, a detailed and comprehensive description of changes made to the original go-ethereum implementation would facilitate a more efficient understanding of the expected behavior of the system.

Mitigation
We recommend improving the project documentation to include working instructions for creating and deploying subnets, and to better describe the modifications that have been made to go-ethereum.

Status
The Ava labs team has indicated that they plan on improving the documentation as it pertains to differences from go-ethereum. Additionally, the documentation for deploying Subnets and the relevant tooling is an ongoing effort and is not final yet.

Verification
Partially Resolved

Suggestion 3: Define the Peer Tracker With a Test Suite

Location
`master/peer/peer_tracker.go`

Synopsis
There was no test coverage for this component of the code. The peer tracker acts as a reputation system for selecting node peers for requests. Reputation systems are notoriously difficult to model and verify and are thus susceptible to gamification and manipulation since they act as entry points for carrying out complex attacks, such as eclipse attacks and related node identity exploits.

Mitigation
Eclipse attacks require a vector for biasing the selection process of a victim node. We recommend developing a test harness for the peer tracker that codifies its rules and models possible attacks on that system.

Status
The Ava labs team has implemented a test for the peer tracker that asserts that connectivity to responsive peers takes precedence over non-responsive ones. The current test coverage for the peer tracker is at 90%.

Verification
Resolved

Suggestion 4: Test Additional Functions

Location
`accounts/abi/abi.go#L218`

`accounts/abi/abi.go#L144`

Synopsis
Additional functions have been added by Avalanche to the original go-ethereum codebase, but there are no test cases for them.

Mitigation

We recommend creating separate tests for new functionality to ensure it is tested.

Status

The Ava labs team has added additional tests to cover the functions that were previously not tested in the `abi` package.

Verification

Resolved.

Suggestion 5: Review the Regular Expression With ReDoS Pattern

Location

`precompile/utils.go#L15`

Synopsis

This regular expression contains a pattern that creates a condition to vulnerability to the Regular expression Denial of Service (ReDoS) attack.

Mitigation

We recommend encouraging software engineers to learn more about ReDos attacks to prevent introducing the vulnerability in the future.

Status

The Ava Labs team has verified and confirmed that the regular expression is not run on untrusted input and that the function is only run on the hardcoded precompiles' function signatures.

Verification

Resolved.

Suggestion 6: Disable Writing to Files For Non Owners

Location

`master/eth/api.go#LL99C68-L99C79`

Synopsis

In the current configuration, all users of the operating system can modify the file in the aforementioned location.

Mitigation

We recommend using more restrictive access control masks, for example, 700 or 770 depending on the access policy.

Status

The Ava Labs team has disabled writing to the file for non owners.

Verification

Resolved.

Suggestion 7: Check the Size of the Slice When Accessing a Slice Element

Location

Examples (non-exhaustive):

`bind/backends/simulated.go#L179`

`bind/backends/simulated.go#L724`

`bind/backends/simulated.go#L839`

`abi/bind/base.go#L497`

`abi/bind/base.go#L516`

`abi/bind/bind.go#L117`

`abi/bind/bind.go#L303`

`accounts/abi/abi.go#L372`

Synopsis

Accessing an element of a slice without checking that the element exists can lead to a runtime error and a panic.

Mitigation

When accessing the `i`th element of a slice, we recommend verifying that the element exists by checking the length of the slice.

Status

The Ava Labs team has added the recommended checks.

Verification

Resolved.

Suggestion 8: Implement Input Validation for the blockCount Parameter

Location

`internal/ethapi/api.go#L106`

Synopsis

Our team found that the `blockCount` variable is cast from type `uint64` to a type `int` variable. This could lead to unexpected behavior in the `FeeHistory` function since the result value of `blockCount` after casting can be less than 1.

Mitigation

We recommend implementing appropriate input validation for the `blockCount` variable.

Status

The Ava Labs team has removed the cast from `uint64` to `int` so that no cast is necessary.

Verification
Resolved.

Suggestion 9: Improve Error Handling

Location
Examples (non-exhaustive):

`master/trie/stacktrie.go#L168`

`master/trie/stacktrie.go#L154`

`bind/backends/simulated.go#L177`

`bind/backends/simulated.go#L180`

`bind/backends/simulated.go#L722`

`bind/backends/simulated.go#L725`

`bind/backends/simulated.go#L749`

`bind/backends/simulated.go#L834`

`bind/backends/simulated.go#L837`

`bind/backends/simulated.go#L840`

`accounts/abi/abi.go#L367`

Synopsis
In the codebase of the project, there are multiple instances where a returned error is ignored (not checked), but the corresponding returned values are used. This can lead to undefined behavior, panics, logic errors, or denial of service.

Mitigation
We recommend checking returned errors before using the corresponding values. We also recommend adding a linter rule detecting this pattern in the code.

Status
The Ava Labs team added the recommended checks for the returned errors.

Verification
Resolved.

Suggestion 10: Clear Sensitive Cryptographic Keys

Location
`accounts/keystore/passphrase.go#L116:` `keyjson`

`accounts/keystore/passphrase.go#L155:` `derivedKey`

`accounts/keystore/passphrase.go#L195:` `keyBytes`

`accounts/keystore/passphrase.go#L226:` `keyBytes`

`accounts/keystore/passphrase.go#L238:` `key`

`accounts/keystore/passphrase.go#L269:` `derivedKey`

`accounts/keystore/passphrase.go#L323:` `derivedKey`

`accounts/keystore/plain.go#L48:` `key`

`accounts/keystore/presale.go#L46:` `key`

`accounts/keystore/presale.go#L93:` `derivedKey`

`accounts/keystore/presale.go#L99:` `ecKey`

`accounts/keystore/keystore.go#L491:` `passphrase`

`accounts/keystore/keystore.go#L432:` `passphrase`

Synopsis

Secret values are not cleared from memory. The leakage of cryptographic keys could result in the loss of security capabilities and properties, such as authentication, integrity, and non-repudiation.

Mitigation

We recommend using the `SetFinalizer` mechanism in Golang to clear the memory, despite the fact that it does not guarantee that the secret values will be removed effectively from the memory.

Status

The Ava Labs team has acknowledged this suggestion and responded that since users should only use this API on their node and should not trust a third-party node to store sensitive data, they have decided not to implement any changes.

Verification

Unresolved.

Suggestion 11: Validate Assumptions in Precompiles

Location

Examples (non-exhaustive):

`master/precompile/allow_list.go#L125`

`master/precompile/contract_deployer_allow_list.go#L88`

`master/precompile/contract_native_minter.go#L150`

`master/precompile/contract_native_minter.go#L156` `master/precompile/contract_native_minter.go#L169`

`master/precompile/utils.go#L38`

`master/precompile/utils.go#L45`

Synopsis

In the current implementation, precompiles depend upon certain assumptions such as the validity of a role specification, the size of the binary representation of an amount, or the existence of a file. These assumptions may either be validated in the code and trigger a panic or error, or may not be validated altogether. This may lead the system to reach unexpected states, such as an incorrect packing of a minting operation. Nevertheless, we could not identify a way to leverage this inconsistency for an attack.

Mitigation

We recommend validating the assumptions being made in the precompiles.

Status

The Ava Labs team has changed the panics to the functions to have a consistent response.

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.