# B² Network zkEVM

# Audit Report

Tue Feb 06 2024

contact@scalebit.xyz https://twitter.com/scalebit_

ScaleBit

# $B^2$ Network zkEVM Audit Report

## 1 Executive Summary

### 1.1 Project Information

| | |
|---|---|
| Description | $B^2$ Network is a Bitcoin layer2 network that bolsters transaction speed and broadens application diversity without sacrificing security. |
| Type | L2 |
| Auditors | ScaleBit |
| Timeline | Mon Jan 29 2024 - Mon Feb 05 2024 |
| Languages | Go, C++ |
| Platform | $B^2$ Network |
| Methods | Dependency Check, Fuzzing, Static Analysis, Manual Review |
| Source Code | https://github.com/b2network/b2-zkevm-prover<br>https://github.com/b2network/b2-zkevm-node |
| Commits | https://github.com/b2network/b2-zkevm-prover:<br>7accb0b5ff5afb9b2e597cd3059a850dc1d56aa4<br>https://github.com/b2network/b2-zkevm-node:<br>bd1db00dc6acc75de1c63e9fce0171c03a0c28f1 |

## 1.2 Files in Scope

The following are the directorys of the original reviewed files.

| Directory |
| --- |
| https://github.com/b2network/b2-zkevm-node/sequencer |
| https://github.com/b2network/b2-zkevm-node/ethtxmanager |
| https://github.com/b2network/b2-zkevm-node/etherman |
| https://github.com/b2network/b2-zkevm-node/jsonrpc |
| https://github.com/b2network/b2-zkevm-prover/src |

## 1.3 Issue Statistic

| Item | Count | Fixed | Acknowledged |
| --- | --- | --- | --- |
| Total | 2 | 0 | 2 |
| Informational | 2 | 0 | 2 |
| Minor | 0 | 0 | 0 |
| Medium | 0 | 0 | 0 |
| Major | 0 | 0 | 0 |
| Critical | 0 | 0 | 0 |

## 1.4 ScaleBit Audit Breakdown

ScaleBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Integer overflow/underflow
- Infinite Loop
- Infinite Recursion
- Race Condition
- Traditional Web Vulnerabilities
- Memory Exhaustion Attack
- Disk Space Exhaustion Attack
- Side-channel Attack
- Denial of Service
- Replay Attacks
- Double-spending Attack
- Eclipse Attack
- Sybil Attack
- Eavesdropping Attack
- Business Logic Issues
- Contract Virtual Machine Vulnerabilities
- Coding Style Issues

## 1.5 Methodology

Our security team adopted **"Dependency Check"**, **"Automated Static Code Analysis"**, **"Fuzz Testing"**, and **"Manual Review"** to conduct a comprehensive security test on the code in a manner closest to real attacks. The main entry points and scope of the security testing are specified in the **"Files in Scope"**, which can be expanded beyond the scope according to actual testing needs. The main types of this security audit include:

### (1) Dependency Check

A comprehensive check of the software's dependency libraries was conducted to ensure all external libraries and frameworks are up-to-date and free of known security vulnerabilities.

### (2) Automated Static Code Analysis

Static code analysis tools were used to find common programming errors, potential security vulnerabilities, and code patterns that do not conform to best practices.

### (3) Fuzz Testing

A large amount of randomly generated data was inputted into the software to try and trigger potential errors and exceptional paths.

### (4) Manual Review

The scope of the code is explained in section 1.2.

### (5) Audit Process

- Clarify the scope, objectives, and key requirements of the audit.
- Collect related materials such as software documentation, architecture diagrams, and lists of dependency libraries to provide background information for the audit.
- Use automated tools to generate a list of the software's dependency libraries and employ professional tools to scan these libraries for security vulnerabilities, identifying outdated or known vulnerable dependencies.
- Select and configure automated static analysis tools suitable for the project, perform automated scans to identify security vulnerabilities, non-standard coding, and potential risk points in the code. Evaluate the scanning results to determine which findings require further manual review.

- Design a series of fuzz testing cases aimed at testing the software's ability to handle exceptional data inputs. Analyze the issues found during the testing to determine the defects that need to be fixed.
- Based on the results of the preliminary automated analysis, develop a detailed code review plan, identifying the focus of the review. Experienced auditors perform line-by-line reviews of key components and sensitive functionalities in the code.
- If any issues arise during the audit process, communicate with the code owner in a timely manner. The code owners should actively cooperate (this may include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- Necessary information during the audit process will be well documented in a timely manner for both the audit team and the code owner.

# 2 Summary

This report has been commissioned by $B^2$ Network with the objective of identifying any potential issues and vulnerabilities within the source code of the $B^2$ Network zkEVM repository, as well as in the repository dependencies that are not part of an officially recognized library. In this audit, we have employed the following techniques to identify potential vulnerabilities and security issues:

### (1) Dependency Check

A comprehensive analysis of the software's dependency libraries was conducted using the Govulncheck tool.

### (2) Automated Static Code Analysis

The code quality was examined using a code scanner.
**For Go language, risk codes detected include:**

- Look for hard-coded credentials
- Bind to all interfaces
- Audit errors not checked
- Potential integer overflow caused by strconv.Atoi result conversion to int16/32
- Potential DoS vulnerability via decompression bomb
- Potential directory traversal
- Use of net/http serve function that has no support for setting timeouts
- SQL query construction using format string
- SQL query construction using string concatenation
- Audit use of command execution
- File path provided as taint input
- File traversal when extracting zip/tar archive
- Detect the usage of DES, RC4, MD5, or SHA1

- Look for bad TLS connection settings
- Ensure minimum RSA key length of 2048 bits
- Insecure random number source (rand)
- Slice access out of bounds
- Array access out of bounds
- Other types of risk codes

**For C++ language, risk codes detected include:**

- Bounds checking
- Check function usage
- Exception safety
- IO using format string
- Leaks (auto variables)
- Memory leaks (address not taken)
- Memory leaks (class variables)
- Memory leaks (function variables)
- Memory leaks (struct members)
- Null pointer
- Uninitialized variables
- Unused functions
- Unused var
- Using postfix operators
- Other types of risk codes

### (3) Fuzz Testing

Based on the go-fuzz tool and by writing harnesses, we have performed fuzz testing on the following targets:

- b2-zkevm-node/jsonrpc

- b2-zkevm-node/etherman

### (4) Manual Code Review

The b2-zkevm-node is based on the 0xPolygonHermez/zkevm-node project, and the b2-zkevm-prover is based on the 0xPolygonHermez/zkevm-prover project. Given that these projects have already been audited and only minor modifications have been made to the b2-zkevm-node and b2-zkevm-prover repositories, our scope of manual code review was quite limited. The primary focus of the manual code review was:

- b2-zkevm-node/etherman
- b2-zkevm-node/synchronizer

During the audit, we identified 2 issues of varying severity, listed below.

| ID | Title | Severity | Status |
| --- | --- | --- | --- |
| GO-1 | Using Unsafe Dependencies. | Informational | Acknowledged |
| FNE-1 | Mismatched Variable Names: Function Parameters vs Implementation. | Informational | Acknowledged |

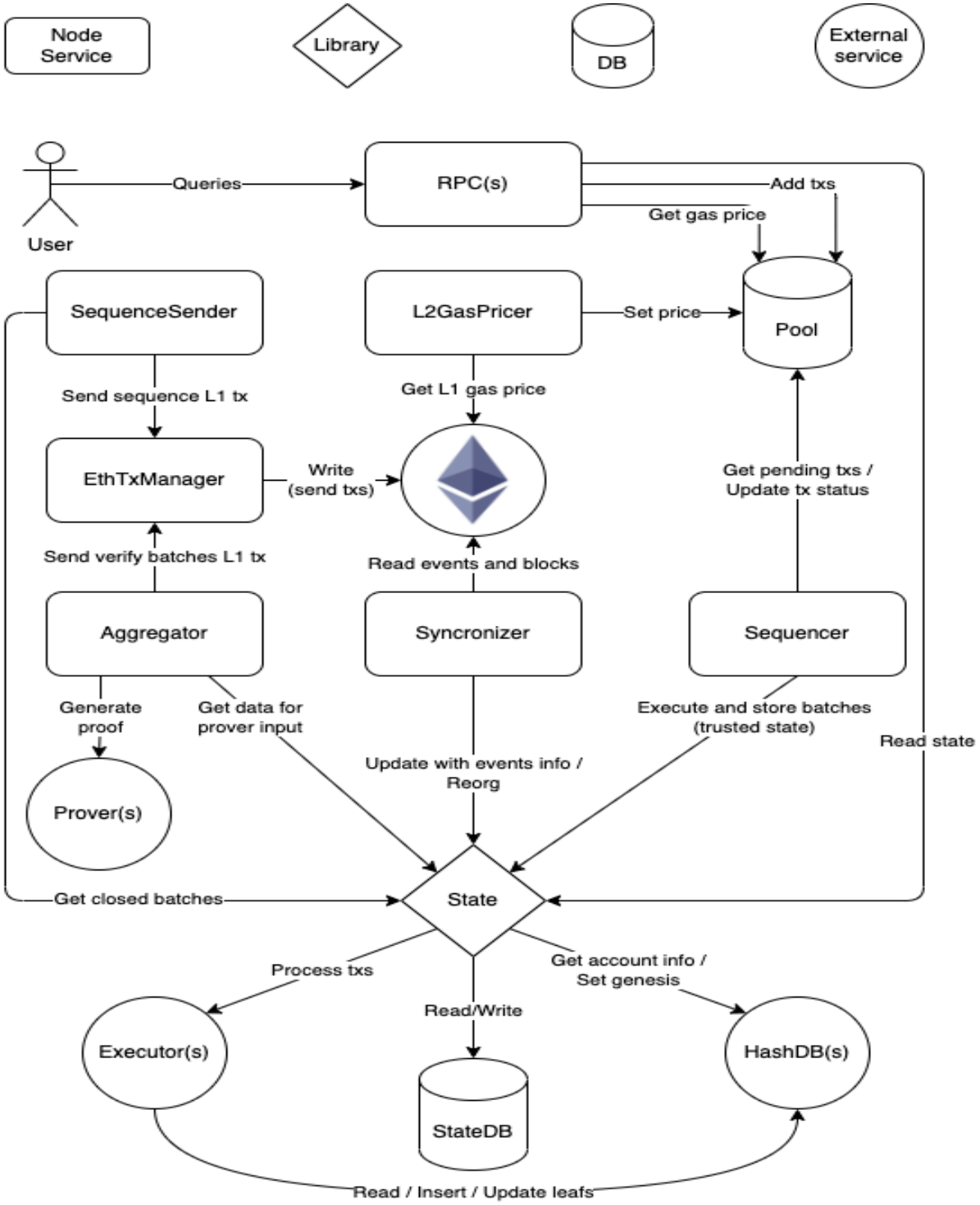
# 3 Participant Process

Here are the relevant actors with their respective abilities within the $B^2$ Network zkEVM repository :

**b2-zkevm-node**

Accepts JSON-RPC requests from users, executes EVM transactions, requests zero-knowledge proofs from the b2-zkevm-prover, and sends state changes and zero-knowledge proofs to the underlying EVM-compatible chain.

The diagram below shows its architecture.




Below is an introduction to the various functional modules inside the b2-evm-node.

- (JSON) RPC: an HTTP interface that allows users (dApps, metamask, etherscan, ...) to interact with the node. Fully compatible with Ethereum RPC + some extra custom endpoints specifics of the network. It interacts with the state (to get data and process transactions) as well as the pool (to store transactions)
- L2GasPricer: it fetches the L1 gas price and applies some formula to calculate the gas price that will be suggested for the users to use for paying fees on L2. The suggestions are stored on the pool, and will be consumed by the rpc
- Pool: DB that stores transactions by the RPC to be selected/discarded by the sequencer later on
- Sequencer: responsible for building the trusted state. To do so, it gets transactions from the pool and puts them in a specific order. It needs to take care of opening and closing batches while trying to make them as full as possible. To achieve this it needs to use the executor to actually process the transaction not only to execute the state transition (and update the hashDB) but also to check the consumed resources by the transactions and the remaining resources of the batch. After executing a transaction that fits into a batch, it gets stored on the state. Once transactions are added into the state, they are immediately available through the rpc
- SequenceSender: gets closed batches from the state, tries to aggregate as many of them as possible, and at some point, decides that it's time to send those batches to L1, turning the state from trusted to virtualized. In order to send the L1 tx, it uses the ethtxmanager
- EthTxManager: handles requests to send L1 transactions from sequencesender and aggregator. It takes care of dealing with the nonce of the accounts, increasing the gas price, and other actions that may be needed to ensure that L1 transactions get mined
- Etherman: abstraction that implements the needed methods to interact with the Ethereum network and the relevant smart contracts
- Synchronizer: Updates the state (virtual batches, verified batches, forced batches, ...) by fetching data from L1 through the etherman. If the node is not a trusted sequencer it also updates the state with the data fetched from the rpc of the trusted sequencer. It also detects and handles reorgs that can happen if the trusted sequencer sends different data in the rpc vs the sequences sent to L1 (trusted reorg aka L2 reorg). Also handles L1 reorgs (reorgs that happen on the L1 network)
- State DB: persistence layer for the state data (except the Merkletree that is handled by the HashDB service), it stores informationrelated to L1 (blocks, global exit root

updates, ...) and L2 (batches, L2 blocks, transactions, ...)

- Aggregator: consolidates batches by generating ZKPs (Zero Knowledge proofs). To do so it gathers the necessary data that the prover needs as input through the state and sends a request to it. Once the proof is generated it sends a request to send an L1 tx to verify the proof and move the state from virtual to verified to the ethtxmanager. Note that provers connect to the aggregator and not the other way arround. The aggregator can handle multiple connected provers at once and make them work concurrently in the generation of different proofs

**b2-zkevm-prover**
Accepts requests from the b2-zkevm-node and generates Stark zero-knowledge proofs. the prover provides critical services through three primary RPC clients: the Aggregator client, Executor service, and StateDB service.

- Aggregator client:The Aggregator client connects to an Aggregator server and harnesses multiple zkEVM Provers simultaneously, thereby maximizing proof generation efficiency. This involves a process where the Prover component calculates a resulting state by processing EVM transaction batches and subsequently generates a proof based on the PIL polynomials definition and their constraints
- Executor service:the Executor service offers a mechanism to validate the integrity of proposed EVM transaction batches, ensuring they adhere to specific workload requirements
- StateDB service:The StateDB service interfaces with a system's state (represented as a Merkle tree) and the corresponding database, thus serving as a centralized state information repository

# 4 Findings

## GO-1 Using Unsafe Dependencies.

Severity: Informational

Discovery Methods: Dependency Check

Status: Acknowledged

Code Location:

go.mod#11

Descriptions:

The current project is using `go-git@v5.10.0` and two vulnerabilities, GO-2024-2456 and GO-2024-2466, have been discovered. The code that uses this dependency in this project is located at `test/scripts/cmd/dependencies/github.go:56`.

Suggestion:

Update the `go-git` dependency to the latest version.

# FNE-1 Mismatched Variable Names: Function Parameters vs Implementation.

Severity: Informational

Discovery Methods: Automated Static Code Analysis

Status: Acknowledged

Code Location:

src/ffiasm/fnec.hpp#143;

src/ffiasm/fnec.cpp#283;

src/ffiasm/fq.hpp#143;

src/ffiasm/fq.cpp#283

Descriptions:

In `src/ffiasm/fnec.hpp:143` and `src/ffiasm/fq.hpp:143`, the declaration of `fromMpz` is:

```
void fromMpz(Element &a, const mpz_t r);
```

However, in `src/ffiasm/fenc.cpp:283` and `src/ffiasm/fq.cpp:283`, the function implementation variables are as follows:

```
void RawFnec::fromMpz(Element &r, const mpz_t a) {
```

This could potentially confuse developers.

Suggestion:

Ensure Parameter Order Consistency: Make sure that the order of parameters in the function declaration and implementation matches exactly.

# Appendix 1

## Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information or assets at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information or assets at risk. All critical issues should be fixed.

## Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

# Appendix 2

## Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.