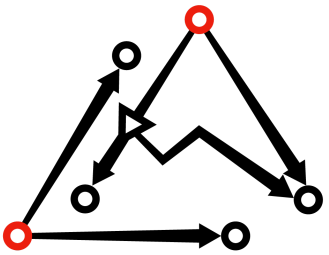

Least Authority

PRIVACY MATTERS

SSV Specification

Security Audit Report

Coin-Dash Ltd.

Final Audit Report: 3 July 2023

Table of Contents

Overview

Background

Coin-Dash Ltd. has requested that Least Authority perform a security audit of their Secret Shared Validator (SSV) specification. SSV is a unique technology that enables the distributed control and operation of an Ethereum validator.

Project Dates

- **March 13, 2023 - April 3, 2023:** Code Review *(Completed)*
- **April 5, 2023:** Delivery of Initial Audit Report *(Completed)*
- **June 8, 2023:** Verification Review *(Completed)*
- **July 3, 2023:** Delivery of Final Audit Report *(Completed)*

Review Team

- Shareef Maher Dweikat, Security Research and Engineer
- Mehmet Gönen, Cryptography Researcher and Engineer
- Jasper Hepp, Security Researcher and Engineer
- Anna Kaplan, Cryptography Researcher and Engineer
- Mirco Richter, Cryptography Researcher and Engineer
- ElHassan Wanas, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the SSV Specification 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:

- Blox Application SSV Specification:
  https://github.com/bloxapp/ssv-spec/tree/v0.3.0

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

- `0a414236cef5eca4dc0f7da464034325869a6477`

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

- Least Authority Blox SSV Specification:
  https://github.com/LeastAuthority/bloxssv-specification

For the verification, we examined the Git revisions:

- Blox Application SSV Specification:
  - https://github.com/bloxapp/ssv-spec/pull/207
  - https://github.com/bloxapp/ssv-spec/pull/216
  - https://github.com/bloxapp/ssv-spec/pull/218
  - https://github.com/bloxapp/ssv-spec/pull/268

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:

- Blox Staking Website: https://www.bloxstaking.com
- The Istanbul BFT Consensus Algorithm: https://arxiv.org/pdf/2002.03613.pdf
- IBFT 2.0 Research Paper [SW19]: https://arxiv.org/abs/1909.10194 (v1)
- An Introduction to Secret Shared Validators (SSV) for Ethereum 2.0: https://medium.com/bloxstaking/an-introduction-to-secret-shared-validators-ssv-for-ethereum-2-0-faf49efcabee
- Designing SSV: A Path to distributed staking infra for Ethereum: https://alonmuroch-65570.medium.com/designing-ssv-a-path-to-distributed-staking-infra-for-ethereum-9586433a536e

In addition, this audit report references the following documents and links:

- QBFT Formal Specification and Verification Repository by Roberto Saltini: https://github.com/ConsenSys/qbft-formal-spec-and-verification
- Annotated Specification by Blox: https://github.com/bloxapp/ssv/blob/stage/ibft/IBFT.md
- BSI TR-02102-1: "Cryptographic Mechanisms: Recommendations and Key Lengths" Version: 2023-1: https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.pdf
- QBFT Presentation for the EEA: https://wiki.hyperledger.org/download/attachments/58855126/QBFT%20Presentation%20for%20the%20EEA%20%281%29.pdf?version=1&modificationDate=1633120327000&api=v2

Areas of Concern

Our investigation focused on the following areas:

- Any attack that impacts the Consensus order;
- Correctness of the implementation and adherence of the implementation to the specification;
- Common and case-specific implementation errors;
- Attacks intending to misuse resources or cause unintended forks;
- Denial of Service (DoS) attacks;
- Any potentially profitable attacks;
- General use of external libraries; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Blox Secret Shared Validator (SSV) builds a distributed validator technology (DVT) to decentralize the control of an Ethereum validator node. The validator key is split among independent operators through

threshold cryptography using the Boneh Lynn Shacham (BLS) cryptography scheme. For each assigned duty for the validator, the Istanbul Byzantine Fault Tolerance (IBFT) Consensus Protocol allows the operators to find agreement about the data.

Our team performed a comprehensive review of Blox SSV, a specification for the implementation of the IBFT2.0 Consensus Protocol [SW19], which provides a theoretical framework for Eventually Synchronous Networks. The Blox SSV specification integrates improvements to IBFT2.0, which are detailed in this presentation, and specified in the QBFT formal verification repository. We compared the Blox specification with the QBFT repository, and noted that the QBFT repository does not currently verify the liveness of the protocol. Our team assumed the correctness of the QBFT repository.

In our review, we compared the Quorum Byzantine Fault Tolerance (QBFT) implementation to the specification in the QBFT folder in the Blox SSV specification. In considering the areas of concern listed above, we closely investigated areas that could lead to liveness and safety issues, and that could compromise the robustness of the system. In addition, we reviewed the SSV and types folders, which implement the interaction of an operator with a Beacon Node to obtain information about validator duties and the encryption of the operator key.

System Design

Overall, the implementation of the Consensus Protocol does not show any critical issues and closely resembles the QBFT formal verification repository. We only found issues such as a missing check (Issue A) and the incorrect use of the justification fields (Issue B). Furthermore, the implementation of the Consensus Protocol code deviates slightly in the Round Change Protocol, which leads to a higher message complexity (Suggestion 5). During our review, we did not find any patterns in the issues we identified.

Code Quality

Overall, the code is well-written and organized. However, the interface functions lack detailed descriptions and explanations that would help developers using the specification. Given that the specification should help developers implement the protocol, we recommend improving the code quality by removing redundant checks (Suggestion 2) and the comments for interface functions (Suggestion 1).

Tests

Our team found that sufficient tests are implemented to check the different flows of the QBFT Consensus Protocol. However, we found missing tests for the QBFT Protocol (Issue C).

The implemented tests are not able to provide the Blox SSV Specification with a formal verification, such as that of QBFT. The test coverage is therefore limited as it is not possible to test Byzantine behavior in a thorough way. Byzantine behavior allows an operator to behave in any arbitrary way. The tests can only prove that all required checks are included in the code.

Documentation

The project documentation provided for this specification review was sufficient and offered an accurate description of the system.

Scope

The scope of this review included all security-critical components of the application.

Dependencies

Our team examined the planned use of dependencies. We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the `Blox-ssv-specifications` applications and to mitigate supply-chain attacks (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: Missing Check on Quorum for the RoundChange Justification	Resolved
Issue B: Incorrect Use of the Justification Fields for a Proposal	Unresolved
Issue C: Missing Tests Could Lead To Incorrect Implementations	Resolved
Issue D: Incorrect Message Type Check	Resolved
Suggestion 1: Improve Code Comments for Interface Functions	Unresolved
Suggestion 2: Improve Code Quality	Unresolved
Suggestion 3: Use ECIES Public Encryption Scheme Instead of RSA-2048 With Low Security Level	Unresolved
Suggestion 4: Update and Maintain Dependencies	Resolved
Suggestion 5: Remove Unnecessary Checks That Increase Message Complexity	Unresolved
Suggestion 6: Implement the Unpredictable Proposer Selection Mechanism Described in the Specification	Resolved

Issue A: Missing Check on Quorum for the RoundChange Justification

Location

`qbft/prepare.go#L59-L73`

`dafny/spec/L1/node_auxiliary_functions.dfy#L673`

Synopsis

The function `getRoundChangeJustification` in the `qbft/prepare.go` file does not check that the set of constructed `Prepare` messages is of size `quorum`. This is a deviation from the QBFT formal verification code.

Impact

The function returns a set of valid `Prepare` messages that is attached to a RoundChange message to justify the round change by the operator. As specified in the QBFT code, the operator needs to check the size of the set here. Not checking this can lead to sending a RoundChange message that is not accepted by other operators, which could, in turn, lead to liveness issues. Since the proposer for a higher round requires `quorum-many` valid RoundChange messages, this can lead to a state in which the operators do not find consensus, and liveness is not reached.

Preconditions

In order for this Issue to occur, an operator has to perform a round change. In addition, the operator should have reached the `Prepare` stage, during which the operator receives `quorum-many` `Prepare` messages, and sets the values `LastPreparedValue` and `LastPreparedRound` prior to the round change.

Feasibility

The requirement of having received `quorum-many` `Prepare` messages makes the scenario unlikely, and it is difficult to exploit the missing check for an attack.

Remediation

We recommend adding the check.

Status

The Blox team has added the check.

Verification

Resolved.

Issue B: Incorrect Use of the Justification Fields for a Proposal

Location

`qbft/proposal.go#L237-L238`

`qbft/prepare.go#L59`

`qbft/round_change.go#L349`

`dafny/spec/L1/types.dfy#L124-L140`

Synopsis

Justification data is attached when creating a new proposal for a higher round including the `RoundChangeJustification`, which is a set of `Prepare` messages, and the `PrepareJustification`, which is a set of RoundChange messages. The code mixes up the two fields and attaches the RoundChange messages to the `RoundChangeJustification` field and vice versa for the `Prepare` messages.

Impact

Due to the fact that the code also mixes up the two data fields upon receiving a proposal message, no harm is caused. However, theoretically, the data is not used in the correct way. In addition, since this is a specification, it is very important to ensure that it is consistent with the theoretical basis, such as the QBFT formal verification repository, to avoid any confusions.

Remediation

We recommend changing the assignment of the fields in `CreateProposal` and the processing of the data in `UponProposal`.

Status

Given that this Issue causes changes across several tests and has a very low impact, the Blox team stated that they prefer to resolve it at a later stage. Hence, at the time of the verification, the suggested remediation has not been resolved.

Verification

Unresolved.

Issue C: Missing Tests Could Lead To Incorrect Implementations

Location

`qbft/spectest`

Synopsis

The QBFT specification is tested by simple tests that run the Consensus Protocol with invalid data. Our team found some tests are missing. In particular there is no test for:

- a `Prepare` message that triggers quorum even though the commit stage is reached already; and
- a `Commit` message that is added to the message container but does not result in quorum.

Impact

The tests aim to verify the correctness of the flows in the QBFT folder. In addition, the generated JSON file from the tests can be run in any implementation and should therefore help developers in testing their own implementation against the specification code. Missing tests can lead to incorrect implementations by other developers.

Remediation

We recommend adding the tests.

Status

The Blox team has added the tests.

Verification

Resolved.

Issue D: Incorrect Message Type Check

Location

`qbft/messages.go#L127`

`qbft/messages.go#L38-L45`

Synopsis

For the validation of a message, the message type is checked to be no larger than 5. Since the message types are between 0 to 3, this allows a message to be of type 4 or 5, even though these cases are unspecified.

Remediation

We recommend adjusting the check to verify that a message `type` is not larger than 3.

Status

The Blox team has adjusted the check.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Code Comments for Interface Functions

Location

`qbft/instance.go#L13-L14`

`qbft/types.go#L20`

`bloxssv-specification`

Synopsis

Some critical components are not part of the Blox SSV Specification and only appear as interfaces. In this case, it is critical to have clear and correct code comments in order to help developers implement the specification correctly. The code comments must explain the expected behavior of the interface as well as all assumptions that are made on the pre and post conditions of the interface's functions.

We list a few non-exhaustive examples:

- `ProposerF` takes as inputs the round and the identifier and returns the proposer for the given values. It is assumed that the function is deterministic and unpredictable; and
- `TimeoutForRound` takes as input the round, resets a running timer, and starts a new round. It assumes that the time increases exponentially in the round.

Mitigation

We recommend improving code comments stating critical assumptions to the interface functions.

Status

The Blox team stated that they plan on implementing the suggested mitigation in the future. Hence, this Issue remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 2: Improve Code Quality

Location

bloxssv-specification

qbft/commit.go#L128

qbft/controller.go#L60

qbft/decided.go#L56-L86

qbft/controller.go#L60

qbft/messages.go#L118

Synopsis

We identified several redundancies in the code. Since this is a specification repository, the aim should be to achieve a very high-quality code such that developers understand the code easily.

Below, we list redundancies we found in the code:

- `signedMessage.Validate` is called several times for one message during the validation. For example, it is called two times for the COMMIT message and three times for a DECIDED message;
- A DECIDED message is checked two times by `isDecidedMsg` in the function `ProcessMsg`. Upon receiving a RoundChange message, it is checked twice that the operator has received `quorum-many` RoundChange messages;
- The check that the identifier is not zero in `msg.Validate` seems unnecessary. Some functions are independent of the instance (e.g. `CreatePrepare`), while others are not (e.g. `UponPrepare`). Additionally, there does not appear to be a clear pattern; and
- Some functions receive inputs twice. For example, `uponProposal` receives the `proposerMsgContainer` and the state, but the state already contains the `ProposerMsgContainer`. In the same way, the function `ProposerF` receives the state and the round (which is already in the state), and then uses round and `state.height`.

Mitigation

We recommend fixing the listed items.

Status

The Blox team stated that they plan on implementing the suggested mitigation after the upcoming mainnet release. Hence, this suggestion remains unresolved at the time of verification.

Verification

Unresolved.

Suggestion 3: Use ECIES Public Encryption Scheme Instead of RSA-2048 With Low Security Level

Location

types/encryption.go

Synopsis

Although `RSA-2048` provides 112-bit security and is recommended by the National Institute of Standards and Technology (NIST), a higher security level is recommended for critical systems. Bundesamt für Sicherheit in der Informationstechnik (BSI) – the Federal Office for Information Security – published a new technical guideline, titled: Cryptographic Mechanisms: Recommendations and Key Lengths. According to this guideline, it is recommended to use a 120-bit security level, instead of 112-bit, by the end of 2023. For this reason, `RSA-2048` should be replaced with another encryption algorithm.

Mitigation

Instead of using RSA with a low-security level, the first solution may be to increase the key size (e.g. `RSA-4096`). But in this case, there may be a performance problem. Therefore, we recommend implementing the ECIES encryption system, which offers security levels greater than 112 (128-bit) and a more effective performance.

Status

The Blox team stated that the remediation is already planned in a System Improvement Process and will be implemented after the upcoming mainnet release, as noted here. Hence, at the time of the verification, the suggested mitigation has not been resolved.

Verification

Unresolved.

Suggestion 4: Update and Maintain Dependencies

Location

`go.mod`

Synopsis

Analyzing `go.mod` for dependency versions using '`go list -json -m all | nancy sleuth`' shows 14 vulnerable dependencies.

Mitigation

We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the `Blox-ssv-specifications` applications 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 Blox team has updated all dependencies identified in the audit.

Verification

Resolved.

Suggestion 5: Remove Unnecessary Checks That Increase Message Complexity

Location

qbft/round_change.go#L202-L268

spec/L1/node_auxiliary_functions.dfy#L546-L561

Synopsis

QBFT comes with a message complexity of $O(n^2)$, which is an improvement over the IBFT2.0 Consensus algorithm [SW19] that has $O(n^3)$ (see slide 10 in the QBFT Presentation for the EEA). The lower complexity is achieved by improving the Round Change Protocol. In the QBFT Protocol, a proposal message for a higher round attaches the RoundChange messages and only once the`Prepare` messages. Upon receiving a proposal for a higher round, an operator validates the RoundChange and `Prepare` justifications. For the RoundChange justification in the QBFT Protocol, it is not necessary to check any `Prepare` messages related to the RoundChange message. In contrast, the implementation of the Consensus Protocol by Blox checks that, for all RoundChange messages, the `Prepare` messages attached to the RoundChange messages are valid. This is unnecessary and is not included in the QBFT formal verification repository.

Mitigation

We recommend removing these checks to improve the efficiency of the Consensus Protocol.

Status

The Blox team stated that they plan on implementing the suggested mitigation after the upcoming mainnet release. Hence, this suggestion remains unresolved at the time of the verification

Verification

Unresolved.

Suggestion 6: Implement the Unpredictable Proposer Selection Mechanism Described in the Specification

Location

qbft/round_robin_proposer.go

qbft/instance.go#L14

Synopsis

As described in the annotated specification by the Blox team, the proposer selection mechanism must be deterministic and unpredictable. However, in the current setup, the mechanism is predictable.

The code takes the round number and the height of each validator instance. Both values are counters that are easily predicted. The annotated specification suggests taking the slot number of the duty, which is unpredictable beyond the next slot.

Mitigation

We recommend taking the slot number as an input to the proposer selection mechanism.

Status

The Blox team has changed the annotated specification and stated that they will continue to use the Round Robin proposer selection mechanism, which is deterministic but predictable. Note that since the main requirement for a selection mechanism is that it is deterministic, this is not a security issue.

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.