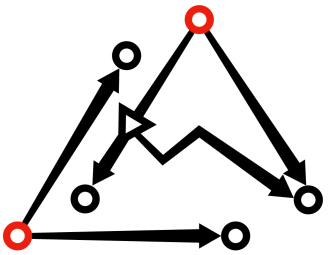

Least Authority

PRIVACY MATTERS

Secret Shared Validator (SSV)

Security Audit Report

Blox

Updated Final Audit Report: 29 August 2023

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - Code Quality
    - Documentation
    - Scope
  - Specific Issues & Suggestions
    - Issue A: Validity of Event Data Received by handleValidatorAddedEvent Is Not Checked
    - Issue B: Missing Validation on OperatorIds Array Size
    - Issue C: Unsafe Dependency Used
    - Issue D: Blox Node Does Not Sync With Contract
    - Issue E: Event Handler Ignores Errors in Executing Duty or Timeout Handlers
    - Issue F: Missing Check on Quorum for the RoundChange Justification
  - Suggestions
    - Suggestion 1: Improve Error Handling, Limit and Avoid Using Panics
    - Suggestion 2: Check That Type Assertion Succeeds
    - Suggestion 3: Prevent Badger InMemory From Running in Production
    - Suggestion 4: Fix Incorrect Throttling Implementation
    - Suggestion 5: Resolve TODOs in the Codebase
    - Suggestion 6: Update Outdated Dependencies
    - Suggestion 7: Specify the Versions of Installed Libraries
- About Least Authority
- Our Methodology

Overview

Background

Blox has requested that Least Authority perform a security audit of their Secret Shared Validator (SSV), an implementation based on the Istanbul BFT Consensus Algorithm paper, and written in Go.

Project Dates

- **April 6, 2023 - May 11, 2023:** Initial Code Review *(Completed)*
- **May 15, 2023:** Delivery of Initial Audit Report *(Completed)*
- **August 10, 2023:** Delivery of the Updated Initial Audit Report *(Completed)*
- **August 21, 2023 - August 23, 2023:** Verification Review *(Completed)*
- **August 23, 2023:** Delivery of Final Audit Report *(Completed)*
- **August 10, 2023 - August 28, 2023:** Additional Review *(Completed)*
- **August 29, 2023:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Shareef Maher Dweikat, Security Research and Engineer
- Nathan Ginnever, Security Researcher and Engineer
- Jasper Hepp, Security Researcher and Engineer
- Nikos Iliakis, 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 the Secret Shared Validator (SSV) 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:

- Bloxapp/ssv:
  https://github.com/bloxapp/ssv/tree/spec-align-qbft

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

- `0d8397c0192b2e832c147753a2941e8b90d94e30`

For the verification, we examined the Git revision:

- `8431edf5e6b8f87c022762d59a3946acb350811f`

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

- Bloxapp/ssv:
  https://github.com/LeastAuthority/blox-ssv-implementation

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:

- H. Moniz, "The Istanbul BFT Consensus Algorithm." *arXiv ePrint Archive*, 2020, [Moniz20]
- Blog post, “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

In addition, this audit report references the following documents:

- 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
- QBFT Formal verification repository by Roberto Saltini: https://github.com/ConsenSys/qbft-formal-spec-and-verification
- SSVNetwork: https://docs.ssv.network/developers/smart-contracts/ssvnetwork#public-registervalidator-publickey-operatorids-shares-amount-cluster
- Type assertions: https://go.dev/ref/spec#Type_assertions
- CompareAndSwap: https://pkg.go.dev/sync/atomic#Value.CompareAndSwap
- Blox SSV specification repository: https://github.com/bloxapp/ssv-spec
- Ethereum 2.0 Specification: https://github.com/ethereum/consensus-specs/blob/dev/specs/phase0/validator.md
- Blox eth2-key-manager: https://github.com/bloxapp/eth2-key-manager/tree/master/slashing_protection
- Attacks and weaknesses of BLS aggregate signatures: https://eprint.iacr.org/2021/377.pdf
- BLS Multi-Signatures With Public-Key Aggregation: https://crypto.stanford.edu/~dabo/pubs/papers/BLSmultisig.html
- Blox SIP ECIES Share Encryption: https://github.com/bloxapp/SIPs/blob/main/sips/ecies_share_encryption.md

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation for both the cryptography and consensus code;
- Common and case-specific implementation errors;
- Communication and interactions between network components;
- Adversarial actions and other attacks on the network;
- Attacks intending to misuse resources, cause unintended forks, and create unwanted or adversarial chains;
- Resistance to denial of service (DoS) and similar attacks;
- Investigation of operations and potential mismanagement of funds;
- Any attack that impacts funds, such as the draining or manipulation of funds;
- Protection against malicious attacks and other types of exploitation;
- 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

The Blox Secret Shared Validator (SSV) builds a distributed validator technology (DVT) intended to decentralize the control of an Ethereum validator node. Our team performed a comprehensive review of the Blox SSV implementation. In our review, we compared the Blox SSV implementation to the Blox SSV specification that integrates improvements to IBFT2.0, which are detailed in this presentation, and specified in the QBFT formal verification repository.

In addition to the areas of concern listed above, we investigated specific security issues and concerns highlighted by the Blox Team. Specifically, our team examined denial of service attack vectors as well as the usage of libp2p at the underlying gossip protocol layer of the validator subnet, and did not identify any issues in the adherence of the implementation to the specification.

We also examined the deviation from the specification repository. The core of the consensus protocol closely follows the Blox SSV Specification (previously audited by Least Authority) and only adds logging, metrics, and improvements for the storing of messages. As such, we did not identify any new security vulnerabilities. However, the Issues from our previous report transfer partially to this report. In particular, Issue A from the previous report also applies to this codebase and must be remediated in this context as well (Issue F). An issue risking the liveness of the pre-consensus protocol has been found by Blox (documented here) and has been fixed in the latest version of the code.

We verified the alignment with the Ethereum 2.0 specification, in particular with the slashing conditions for validators. The Blox SSV depends on the slashing conditions implemented in the Blox `eth2-key-manager` repository. These conditions are stricter than the Ethereum 2.0 slashing conditions for reasons of efficiency. We did not find any issue with this approach, especially with regards to falsely accepting any slashable attestations – that is, the set of accepted attestations is a subset of the set of valid attestations defined in the Ethereum 2.0 specification. Tests to verify this are specified in a PR.

Our team identified areas of improvement within the system's implementation that would improve the security of Blox SSV, if resolved. We recommend that error handling be improved by avoiding the use of panics (Suggestion 1). During our review, we identified patterns of missing event validations within the codebase. We looked at the smart contracts' event handling and found that there are no checks verifying the validity of the data received from events, which could result in DoS attack vectors (Issue A, Issue B). In adding new validators, a bad actor can fill any number of events with trash data and consume node resources to fill up the storage.

Blox uses the Boneh Lynn Shacham (BLS) cryptography scheme to split the validator key among independent operators. We have checked the security of the scheme against attacks mentioned in both the the "Attacks and weaknesses of BLS aggregate signatures" paper and the "BLS Multi-Signatures With Public-Key Aggregation" paper, but since the public keys of the operators are registered on-chain, we did not find any attack against the Blox system. In addition, as noted in our previous audit report, we recommend replacing RSA, the encryption scheme for the validator shares, with ECIES encryption scheme to ensure a security level of 128 bits. This has not been implemented yet but is an approved improvement proposal SIP and expected to be added soon.

Code Quality

Our team found that Blox SSV's codebase is well-organized and generally adheres to best practice guidelines. However, we identified some instances of missing `type` assertion checks that could cause a panic. We recommend that a check be implemented (Suggestion 2).

Tests

There are sufficient tests implemented. In our examination of a node's initializing code, we found code written for use in tests only or in the development environment, which could be harmful if kept in the production release (Issue D, Suggestion 3).

Documentation

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

Code Comments

Generally, we found the code to be well-organized. However, there are multiple instances of unresolved TODOs and some instances of unused code comments, which may create confusion about the intended functionality and completeness of the implementation. This could potentially result in errors or missed vulnerabilities by developers and security researchers. As a result, we recommend resolving all TODOs (Suggestion 5) and removing unused code from the codebase.

Scope

The scope of this review included all security-critical components of the application. In our review, we compared the Quorum Byzantine Fault Tolerance (QBFT) implementation to the specification in the QBFT folder in the Blox SSV specification, per our previous review. Furthermore, during this audit, our team compared the Blox SSV Specification to the Blox SSV Implementation.

Dependencies

Our team identified a library used in the implementation that could cause unintended behavior (Issue C). We also found several unmaintained dependencies that could lead to the introduction of vulnerabilities and bugs into the codebase. We recommend using up-to-date, well-maintained libraries (Suggestion 6). The suggestion is

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: Validity of Event Data Received by handleValidatorAddedEvent Is Not Checked	Resolved
Issue B: Missing Validation on OperatorIds Array Size	Resolved
Issue C: Unsafe Dependency Used	Resolved
Issue D: Blox Node Does Not Sync With Contract	Resolved
Issue E: Event Handler Ignores Errors in Executing Duty or Timeout Handlers	Resolved
Issue F: Missing Check on Quorum for the RoundChange Justification	Resolved
Suggestion 1: Improve Error Handling, Limit and Avoid Using Panics	Resolved

Suggestion 2: Check That Type Assertion Succeeds	Resolved
Suggestion 3: Prevent Badger InMemory From Running in Production	Resolved
Suggestion 4: Fix Incorrect Throttling Implementation	Resolved
Suggestion 5: Resolve TODOs in the Codebase	Unresolved
Suggestion 6: Update Outdated Dependencies	Partially Resolved
Suggestion 7: Specify the Versions of Installed Libraries	Resolved

Issue A: Validity of Event Data Received by handleValidatorAddedEvent Is Not Checked

Location

`operator/validator/event_handler.go#L32`

Synopsis

The smart contract's `registerValidator` function can be used to trigger events. However, the validity of the event data received by the `handleValidatorAddedEvent` function is not checked. As a result, a malicious actor can send multiple `ValidatorAddedEvent` events and add several invalid (nonexistent) validators. The unchecked data is stored in the smart contract and an event is emitted to operator nodes.

Impact

The function does not check for the validity of the passed event's data. Consequently, if the storage is limited, the node can be overloaded, which results in legitimate validators not being added.

This Issue could also be exploited for DoS attacks that could potentially bring the server down.

Preconditions

The storage capacity of the server must be limited for an attacker to be able to overload it with invalid validators.

For DOS attacks, no preconditions are needed.

Feasibility

Straightforward.

Remediation

We recommend checking the validity of the `ValidatorAddedEvent` event's validator. We also recommend setting maximum limits for the number of validators that can be added per unit of time, and based on the storage capacity of the server.

Status

The Blox team stated that such attacks (DoS attacks and overloading the server with invalid validators) are unlikely, as they would be very expensive to execute. The malicious actor would have to pay a transaction fee for every invalid validator they want to add.

Verification

Resolved.

Issue B: Missing Validation on OperatorIds Array Size

Location

`operator/validator/utils.go#L60`

Synopsis

There is no validation performed for the array size of `OperatorIds`. A malicious actor could perform a DoS attack by sending a large array size in, for example, a `ValidatorAddedEvent` event, causing the loop to run for a long period of time. Alternatively a numerous number of events could be sent with a normal array size (Issue A), creating a DoS attack vector. The smart contract's `registerValidator` function can be used to trigger such an event. Alternatively, an attacker can find a way to send an event to the Blox node using another compromised function or without even going through the smart contract at all.

Impact

Enough requests can deplete the server's resources and cause unintended behavior.

Feasibility

Straightforward.

Remediation

We recommend adding a check to verify that arrays are not larger than the smart contract's `registerValidator` function standard (13 operators).

Status

The Blox team has added a check to prevent the operator's array length from exceeding 13 items.

Verification

Resolved.

Issue C: Unsafe Dependency Used

Location

`blob/master/go.mod`

Synopsis

During our manual review of the codebase, our team found that the `Decoder` function of the gob package is being used. The `Decoder` function only performs basic sanity checks on decoded input sizes.

Impact

If an input with a large size is passed to the function, it may deplete the server's resources and cause unintended behavior.

Remediation

We recommend wrapping the `Decoder` function with a condition to set a maximum size for inputs.

Status

The Blox team has added a limit on the input size that is passed to the `Decoder` function.

Verification

Resolved.

Issue D: Blox Node Does Not Sync With Contract

Location

cli/operator/node.go#L149

Synopsis

In the development environment, the Blox node retrieves events from a local file (for testing purposes) instead of syncing with the smart contract. The code does not check the environment in which it is running, rather only uses the file, if it is available. If the file gets deployed to the production server, the node will not sync with the smart contract.

Impact

Actions taken due to unintended or malicious events could prevent the network and the consensus protocol from functioning as intended.

Preconditions

For this Issue to occur, the `events` file should be deployed to the server.

Feasibility

If the precondition is met, it would be relatively straightforward to execute the attack.

Remediation

We recommend restricting the loading of the local `events` file to the `development` environment by wrapping it with a condition.

Status

The Blox team has added a flag in newly created databases that indicates whether the local file is set. If the node is restarted with values different from what is persisted, it would crash with an error.

Verification

Resolved.

Issue E: Event Handler Ignores Errors in Executing Duty or Timeout Handlers

Location

protocol/v2/ssv/validator/events.go#L23

protocol/v2/ssv/validator/events.go#L29

Synopsis

As part of processing a network message, the `handleEventMessage` function is called, which in turn calls the respective handler either for validating or handling a timeout. These handlers could return an error. However, such an error would only be logged instead of being propagated to the message

processing function that called it. This could lead to the loss of messages, and logs, unnoticed with the exception of the logs.

Impact

This Issue could result in a false positive on messages that have failed to be processed successfully.

Remediation

In case of an error, we recommend returning the error received from the `OnTimeout` or `OnExecuteDuty` functions from the `handleEventMessage`.

Status

The Blox team has implemented the remediation as recommended.

Verification

Resolved.

Issue F: Missing Check on Quorum for the RoundChange Justification

Location

`protocol/v2/qbft/instance/prepare.go#L84-L97`

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

Synopsis

The function `getRoundChangeJustification` in the `qbft/instance/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, and the Issue has been also previously reported in an audit on the Blox SSV specification.

Impact

Low. 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 implemented the `HasQuorum` function from the `QBFT` package to verify whether quorum size is reached.

Verification

Resolved.

Suggestions

Suggestion 1: Improve Error Handling, Limit and Avoid Using Panics

Location

`blox-ssv-implementation/operator/fork.go`

Synopsis

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

Mitigation

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

Status

The Blox team has refactored the code to return fmt.Errorf error objects instead of triggering a panic.

Verification

Resolved.

Suggestion 2: Check That Type Assertion Succeeds

Location

`operator/validator/event_handler.go`

Synopsis

The code in the function `Eth1EventHandler` does not check that type assertion holds. In the case that type assertion is `false`, a runtime panic can occur.

Mitigation

We recommend that a check be added to verify that type assertion holds.

Status

The Blox team has removed type assertions that could cause panics.

Verification

Resolved.

Suggestion 3: Prevent Badger InMemory From Running in Production

Location

`storage/main.go#L17`

`storage/kv/badger.go#L42`

Synopsis

Tests can instantiate Badger with `InMemory` mode. In this mode, all data is stored in memory, not disk. Although reads and writes will be faster this way, if the node shuts down, all data will be lost. While such a mode is acceptable to use in tests, it is not recommended in production. Usually, memory capacity is much smaller and more expensive than the disk's capacity.

If the node is launched on production with `InMemory` set to `true` by accident, all the data will be loaded to the memory, making the node slower due to the filled up memory, or even resulting in the nodes potentially crashing if the memory is depleted completely.

Mitigation

We recommend only instantiating Badger with `InMemory` mode inside test files. Alternatively, we recommend wrapping the instantiating code with a condition to prevent it from running in the production environment.

Status

The Blox team has moved the `InMemory` mode logic into a special function called `NewInMemory` that can be called by test files.

Verification

Resolved.

Suggestion 4: Fix Incorrect Throttling Implementation

Location

`network/syncing/syncer.go`

Synopsis

The `syncing` package contains a function that is intended to be used to throttle calls to an underlying function, such that the function is called at most once during a given time period. Our team found that the implementation of this function is not thread-safe, which allows multiple calls to occur within a given time period. This might allow for race conditions or other concurrency-related issues.

Mitigation

We recommend implementing `CompareAndSwap` in the `Throttle` implementation to do a swap, only if the time stored in the pointer has not changed after the call to `time.Sleep`.

If the value has changed, the subsequent `handler` call must wait. This can be achieved by repeatedly trying `CompareAndSwap` until it returns `true`.

Status

The Blox team has removed the `Throttle` function because it was not being utilized.

Verification

Resolved.

Suggestion 5: Resolve TODOs in the Codebase

Location

network/peers/conn_manager.go#L63

network/discovery/service.go#L18

network/discovery/subnets.go#L12

identity/store.go#L15

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

Verification

Unresolved.

Suggestion 6: Update Outdated Dependencies

Synopsis

Analyzing `go.mod` for dependency versions using `go list -json -m all | nancy sleuth` showed that there are 13 vulnerable dependencies in the project. In addition, running `go list -u -m -json all | go-mod-outdated -direct` revealed a total of 42 outdated dependencies.

Using outdated or vulnerable dependencies exposes the system to attacks that could result in the exfiltration of sensitive data.

Remediation

We recommend following a process that emphasizes secure dependency usage to avoid introducing vulnerabilities to the Blox Implementation 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 `go.mod` 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 stated that they resolved the suggested mitigation. However, our team found 6 remaining vulnerable packages at the time of the verification.

Verification

Partially Resolved.

Suggestion 7: Specify the Versions of Installed Libraries

Location

`blob/master/Dockerfile#L52`

Synopsis

Unpinned versions in the `Dockerfile` can lead to failure of the build due to library updates.

Mitigation

We recommend specifying the versions of any used libraries.

Status

The Blox team has implemented the mitigation as recommended.

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.