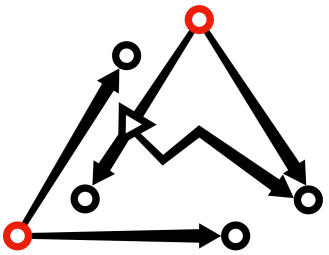

Least Authority

PRIVACY MATTERS

Hashi

Security Audit Report

Gnosis

Initial Audit Report: 12 June 2024

This Initial Audit Report is intended for internal use and discussion purposes only. We advise against sharing this report beyond trusted team members and recommend that publication take place only after the verification has been completed and the Final Audit Report has been delivered.

Table of Contents

Overview

Background

Gnosis has requested that Least Authority perform a security audit of Hashi and its integration with the Gnosis canonical bridges.

Project Dates

- **May 20, 2024 - June 10, 2024:** Initial Code Review *(Completed)*
- **June 12, 2024:** Delivery of Initial Audit Report *(Completed)*
- **TBD:** Verification Review
- **TBD:** Delivery of Final Audit Report

The dates for verification and delivery of the Final Audit Report will be determined upon notification from the Gnosis team that the code is ready for verification.

Review Team

- Nikos Iliakis, Security Researcher and Engineer
- Will Sklenars, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Hashi system and bridge integrations, followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repository and branches are considered in scope for the review:

- Gnosis - Hash repositoryi: https://github.com/gnosis/hashi
  - except for `GiriGiriBashi.sol`
- Gnosis Canonical Bridges and the Hashi integration:
  - AMB: https://github.com/crosschain-alliance/tokenbridge-contracts/tree/feat/hashi-integration-amb
  - XDAI: https://github.com/crosschain-alliance/tokenbridge-contracts/tree/feat/hashi-integration-xdai-bridge

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

- Hashi: `2f46eb2b8133add47badee9bdcffa380ab5e392`
- AMB: `0cef7054be1be91203b09333aac8f7621b07afd5`
- xDai: `7e60b0c46e168d1b73e766877c0d24cedbad6db6`

For the review, these repositories were cloned for use during the audit and for reference in this report:

- Gnosis - Hashi: https://github.com/LeastAuthority/Gnosis-Hashi
- Gnosis Canonical Bridges Hashi integration:

    - https://github.com/LeastAuthority/tokenbridge-contracts/

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:

- Gnosis Chain: https://docs.tokenbridge.net
- Introduction to Hashi 2.0: https://crosschain-alliance.gitbook.io/hashi/v0.2/introduction
- GIP-93: Hashi - Gnosis Chain Bridges integration - Milestone 1 (Google Doc) *(shared with Least Authority via Telegram on 28 May 2024)*
- Hashi Integration Blueprint (Google Doc) *(shared with Least Authority via Telegram on 22 May 2024)*
- Architectural Design.pdf *(shared with Least Authority via Telegram on 4 March 2024)*

In addition, this audit report references the following documents:

- NatSpec Format: https://docs.soliditylang.org/en/v0.8.26/natspec-format.html
- `Ownable2StepUpgradeable.sol`: https://github.com/OpenZeppelin/openzeppelin-contracts-upgradeable/blob/v4.8.1/contracts/access/Ownable2StepUpgradeable.sol

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Common and case-specific implementation errors;
- Adversarial actions and other attacks on the bridge;
- Attacks that impact funds, such as the draining or manipulation of funds;
- Mismanagement of funds via transactions;
- Denial of Service attacks and security exploits that would impact or disrupt execution of the bridge;
- Vulnerabilities within individual components and whether the interaction between the components is secure;
- Exposure of any critical information during interaction with any external libraries;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other methods of exploitation;
- Data privacy, data leaking, and information integrity;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Our team performed a security audit of Gnosis Hashi — an Ethereum Virtual Machine (EVM) Hash Oracle Aggregator that utilizes third-party hash oracles to aggregate hashed data, such as block headers or other arbitrary messages, in order to facilitate secure cross-chain communication. Hashi can be configured to use multiple oracles, and only confirm a message on the target chain once a threshold of oracles has reached an agreement. By using multiple oracles, Hashi aims to reduce the risk to the end user, as an attacker would have to compromise multiple oracles in order to affect the outcome of a message sent with Hashi.
In our review, we investigated Hashi's core functionality and its integration with the xDAI and AMB bridges. The integrations involve adding new functions to manage Hashi's configuration, and maintain compatibility with existing processes.

System Design

Our team examined the design of the Hashi protocol and the integration of its components, along with investigating how the components communicate and interact with each other. We reviewed the codebase for common vulnerabilities (re-entrancy, access control, etc.), any bypasses of the consensus, corruptions of the storage or the messages.

The `ShoyuBashi` and GiriGiriBashi contracts (Hashi codebase) and `HashiManager` contract (Gnosis canonical bridges) provide controls for the system owner, enabling them to add and remove oracles for different domains and set the thresholds for consensus. `Yaho` dispatches cross-chain messages from the source chain, while `Yaru` executes messages on the destination chain. Hashi facilitates domain-specific hash queries, allowing users to retrieve hashed data by providing a chainID and messageID. The threshold consensus mechanism allows Hashi to reach a high-confidence conclusion, increasing the reliability of the retrieved data. Once a majority of adaptors have relayed a hash from the source chain to the destination chain, the information is considered valid by the Hashi system, and users or contracts on the destination chain can access the agreed upon hash. Hashi's redundant approach prioritizes security and reliability, seeking to mitigate the issues present in traditional trusted bridge solutions.

In our review, we discovered some potential issues relating to a race condition, which were present in both the Hashi codebase (Issue A) as well as in the `tokenbridge-contracts` codebase (Issue B). We also found that `hashiManager` may be misconfigured, and suggested adding input validations for functions that update the Hashi configuration (Issue D, Suggestion 4, Suggestion 11).

Our team additionally found that some parts of the codebase use the deprecated function `safeApprove` (Suggestion 1), and that the codebase uses the solidity `transfer` function, which has the potential to revert when sending funds to a smart contract (Suggestion 3).

Hashi utilizes the OpenZeppelin `OwnableUpgradeable` module, which facilitates transfer of ownership. We suggest using two-step ownership transfer to mitigate the risk of transferring contract ownership to an invalid address (Suggestion 5).

Code Quality

We performed a manual review of the contracts in scope and found the code to be clean, well-organized, and in adherence to Solidity best practices. The code was generally easy to follow, and the separation of

the code into modules followed logical patterns. We identified some potential areas where repeated logic and strings could be abstracted and reused (Suggestion 8, Suggestion 9).

The Hashi codebase uses TypeScript for the testing and deployment logic. We found that the `tokenbridge-contracts` codebase uses vanilla JavaScript. We recommend that the Gnosis development team consider migrating to TypeScript for added type safety (Suggestion 6).

In the Hashi codebase, we found the function naming to be clear, but identified some cases where naming could be improved (Suggestion 10, Suggestion 13).

Tests

Our team found that sufficient test coverage of the smart contracts (100% in `contracts` and 99.37% in `contracts/ownable`) has been implemented.

Documentation and Code Comments

The project documentation provided for this review provided a clear overview of the system and its intended behavior. Additionally, code comments sufficiently describe the intended behavior of security-critical components and adhere to NatSpec guidelines. We identified one instance where commenting could be improved regarding `public onlyOwner` functions (Suggestion 12).

Scope

The scope of this review was sufficient and included all the contracts in `packages/evm/contracts`, (with the exception of `GiriGiriBashi.sol`) as well the changes implemented for the integration on the `hashi-integration-amb` and `hashi-integration-xdai-bridge` repositories. However, our team recommends performing a comprehensive, follow-up audit once more integrations or functionality are added or updated.

Dependencies

Our team did not identify any security issues in the use of dependencies. The Gnosis team uses the well-audited OpenZeppelin libraries, which are a standard for smart contracts development.

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: Hashi System May Confirm Messages Without Meeting Threshold of Adaptors	Reported
Issue B: Gnosis Canonical Bridges May Allow Confirmation of Messages Without Waiting for Agreement From a Sensible Quorum of Adaptors	Reported
Issue C: Native Tokens May Be Locked in Hashi Contracts Indefinitely	Reported
Issue D: HashiManger Can Be Configured With a Threshold of 0	Reported
Suggestion 1: Replace Deprecated safeApprove Function With approve	Reported

Function	
Suggestion 2: Update Solidity Compiler Version	Reported
Suggestion 3: Consider Using call Function Instead of transfer Function	Reported
Suggestion 4: Check Equality of Array Length	Reported
Suggestion 5: Implement Two-Step Ownership Transfer for Ownable Contracts	Reported
Suggestion 6: Consider Migrating JavaScript Files to TypeScript	Reported
Suggestion 7: Update Broken NPM Scripts	Reported
Suggestion 8: Abstract RPC URL Equality Check Into Helper Function	Reported
Suggestion 9: Abstract chainIds Into Constants	Reported
Suggestion 10: Rename the Argument _adapters to adaptorSettings (Out of Scope)	Reported
Suggestion 11: Check for the Zero Address When Setting Bond Recipient (Out of Scope)	Reported
Suggestion 12: Comment Public Functions as onlyOwner Where Applicable (Out of Scope)	Reported
Suggestion 13: Update Naming of bondRecipient Variable (Out of Scope)	Reported

Issue A: Hashi System May Confirm Messages Without Meeting Threshold of Adaptors

Location

`contracts/ownable/ShuSo.sol#L77`

`contracts/ownable/ShuSo.sol#L103`

`contracts/ownable/ShuSo.sol#L216`

Synopsis

ShuSho provides the functionality to add or remove adapters and reporters, and update the threshold. However, there is currently no way to do so in an atomic transaction. If a Hashi user sends a transaction concurrently with an administrator updating adapters, reports, and the threshold, the user's transaction could interact with Hashi while it is in an inconsistent state.

Impact

If a Hashi administrator were to add new adaptors and reporters, and increase the threshold to suit the new configuration, a user's concurrent transaction could be sandwiched between the administrator's transactions, resulting in the user's transaction running on Hashi while it has an inconsistently high or low threshold. This could either result in hashes being confirmed when they should not have been, or not

being confirmed when they should, depending on the order in which the concurrent transactions are executed on the EVM.

Preconditions

This Issue occurs when an administrator updates the Hashi configuration concurrently with a user calling the `getHash` function.

Feasibility

This scenario is somewhat likely to occur in the event of a Hashi configuration update, presuming that calls to the `getHash` function occur regularly.

Remediation

We suggest extending ShuSho `.sol`, such that adaptors, reportors, and the threshold can be updated atomically.

Status

Reported.

Issue B: Gnosis Canonical Bridges May Allow Confirmation of Messages Without Waiting for Agreement From a Sensible Quorum of Adaptors

Location

`contracts/upgradeable_contracts/HashiManager.sol`

Synopsis

This Issue relates to the same scenario reported in Issue A, but concerns `HashiManager.sol` in the `tokenbridge-contracts`. `HashiManager` provides the functionality to update adapters, reporters, and the threshold. However, there is no functionality to do so in an atomic transaction. If a user of Gnosis canonical bridges had their transaction executed on the EVM as the adaptors are updated and when threshold is set, their transaction would interact with the bridge while Hashi is in an inconsistent state.

Impact

A message could be passed without a reasonable amount of adaptor confirmations, or a message could be denied from being passed despite a reasonable threshold of adaptors.

Preconditions

This Issue can occur when a user interacts with Gnosis canonical bridges concurrently with an administrator updating the Hashi configuration.

Feasibility

Presuming that the Gnosis canonical bridges are used regularly, it is likely that any time `HashiManager.sol` adaptors, reporters, or the threshold are updated, some user transactions may be executed on an inconsistently configured bridge.

Remediation

We suggest extending `HashiManager.sol`, such that adaptors, reporters, and the threshold can be configured in an atomic transaction.

Status

Reported. This Gnosis development team has already resolved this Issue in commit `0cef705`, as recommended.

Issue C: Native Tokens May Be Locked in Hashi Contracts Indefinitely

Location

`evm/contracts/Yaho.sol`

`evm/contracts/ownable/GiriGiriBashi.sol`

Synopsis

Multiple functions within the Hashi codebase are payable. Due to misconfigured transactions by a user, or subtle bugs in the current implementations, native tokens could end up locked in the Hashi contracts. Without the presence of a backup withdrawal mechanism, these funds could remain locked in the contracts indefinitely.

Impact

A user interacting with the Hashi `payable` functions could have their native tokens locked in the contract.

Preconditions

A user would have to unwittingly send native tokens to a payable function.

Feasibility

Not likely to occur.

Remediation

For contracts with payable functions, we recommend implementing a generic withdrawal mechanism with appropriate access control to facilitate the withdrawal of funds in the event of mistakes or failures.

Status

Reported.

Issue D: HashiManger Can Be Configured With a Threshold of 0

Location

`contracts/upgradeable_contracts/HashiManager.sol#L23`

Synopsis

In the function `setReportersAdaptersAndThreshold`, it is possible to pass an inappropriate threshold, such as `0` or `1`.

Impact

An overly low threshold could lead to messages being confirmed without reaching a reasonable quorum of adaptor confirmations.

Preconditions

An administrator of the Gnosis canonical bridges would have to unwittingly pass an inappropriate value when setting the threshold.

Feasibility

As the incorrect configuration would be due to an unintentional action performed by an administrator, this Issue is not considered likely to occur. However, the risk scales with the frequency of configuration updates.

Technical Details

```
    function setReportersAdaptersAndThreshold(address[] reporters, address[] adapters, uint256 threshold)

    external

    onlyOwner

    {

        // missing threshold check

        _setArray(N_REPORTERS, "reporters", reporters);

        _setArray(N_ADAPTERS, "adapters", adapters);

        uintStorage[THRESHOLD] = threshold;

    }
```

Remediation

Upon setting the threshold, we recommend checking that the supplied parameter is a reasonable value. For example, `contracts/ownable/GiriGiriBashi.sol#L219` checks that the threshold is greater than 50% of the count of adapters.

Status

Reported.

Suggestions

Suggestion 1: Replace Deprecated safeApprove Function With approve Function

Location

`adapters/ZetaChain/ZetaReporter.sol#L37`

Synopsis

The `safeApprove` function was added to `OpenZeppelin` as a solution to a front running vulnerability associated with the `approve` function. However, the `safeApprove` function has issues of its own, which are similar to the issues relating to the `approve` function. Consequently, it has the potential to convey a false sense of security. Because of this, it was deprecated by `OpenZeppelin`, who recommended the use of the `approve` function instead.

Mitigation

We recommend using the `approve` function instead of the deprecated `safeApprove` function.

Status

Reported.

Suggestion 2: Update Solidity Compiler Version

Location

packages/evm/hardhat.config.ts#L126

truffle-config.js#L41

Synopsis

The Hashi `hardhat.config.js` file in Hashi references solidity version `0.8.21`, and the Gnosis canonical bridges reference version `0.4.24`. The current version, as of the writing of this report, is version `0.8.25`. Older compiler versions may have known and fixed issues that could be leveraged for an attack.

Mitigation

We recommend updating the solidity compiler to version `0.8.24`. This minimizes exposure to unknown issues that may have been introduced in the latest release, while also including recent bug fixes.

Status

Reported.

Suggestion 3: Consider Using call Function Instead of transfer Function

Location

Multiple occurrences throughout the `Gnosis-Hashi` and `tokenbridge-contracts` codebases

Synopsis

The Solidity `transfer` function allows only `2300` gas. In the event that the receiving account is a smart contract, the transaction may run out of gas and revert. Failure could occur if the receiving contract is an upgradable contract, or if the receiving contract's fallback function has custom logic. By default, the `call` function allows the maximum gas available, but the `call` function can also be passed a gas allowance as an argument.

Mitigation

We recommend checking all occurrences of the `transfer` function and investigating the likelihood that the recipient may be a smart contract. If it is possible that the recipient could be a smart contract, we recommend using the `call` function instead. Note that if the `call` function is used, extra caution should be taken to protect against re-entrancy.

Status

Reported.

Suggestion 4: Check Equality of Array Length

Location

`contracts/upgradeable_contracts/HashiManager.sol#L27`

Synopsis

The function `setReportersAdaptersAndThreshold` in the `HashiManager` contract does not check whether the array lengths of reporters and adapters are equal. This could lead to an inconsistent configuration of the `HashiManager` contract.

Mitigation

We recommend checking that `reporters.length == adapters.length`, and throwing an exception if not.

Status

Reported.

Suggestion 5: Implement Two-Step Ownership Transfer for Ownable Contracts

Location

`contracts/ownable/ShuSo.sol`

`contracts/upgradeable_contracts/HashiManager.sol`

`contracts/upgradeable_contracts/BasicBridge.sol`

Synopsis

Having a two-step claimable ownership reduces the risk of transferring ownership to an invalid address.

Mitigation

We recommend using `OpenZepplin`'s `Ownable2StepUpgradeable.sol` instead of `OwnableUpgradeable.sol`.

Status

Reported.

Suggestion 6: Consider Migrating JavaScript Files to TypeScript

Location

`hashi-integration-amb/deploy/src`

`hashi-integration-amb/test`

Synopsis

The `tokenbridge-contracts` use JavaScript files for testing and deployment. JavaScript lacks a static type system, due to which sublte errors are sometimes not discovered until runtime.

Mitigation

We recommend migrating the testing and deployment scripts to TypeScript. This would add static typing, which could help eliminate potential, unknown bugs in the testing and deployment scripts. It would also improve code readability and maintainability, and facilitate safer refactoring.

Status

Reported.

Suggestion 7: Update Broken NPM Scripts

Location

hashi-integration-amb/deploy/package.json#L8

hashi-integration-amb/deploy/package.json#L9

hashi-integration-amb/deploy/package.json#L18

Synopsis

Some of the scripts defined in the tokenbridge-contracts `package.json` (`test:gasreport`, `test:gasreport:ci`, and `coverage`) do not function correctly.

Mitigation

We recommend either fixing the scripts or deleting them if they are no longer required.

Status

Reported.

Suggestion 8: Abstract RPC URL Equality Check Into Helper Function

Location

deploy/src/deploymentUtils.js#L240

deploy/src/deploymentUtils.js#L259

deploy/src/deploymentUtils.js#L326

Synopsis

The equality check `url === HOME_RPC_URL` is performed multiple times, making it a good candidate for abstraction.

Mitigation

We recommend creating the new helper function `isHomeRPCUrl` and using it instead of the equality checks to improve readability, as follows:

```
function isHomeRPCUrl (url) {
  return url === HOME_RPC_URL;
}
```

Status
Reported.

Suggestion 9: Abstract chainIds Into Constants

Location
deploy/src/deploymentUtils.js#L240

deploy/src/deploymentUtils.js#L259

Synopsis
Strings representing chain IDs are defined in multiple instances in `deploymentUtils.js`. To improve code readability and maintainability, these could be abstracted into constant variables and defined at the top of the file.

Mitigation
We recommend defining constant variables for the chain IDs, as follows:

```
const HOME_CHAIN_ID = "0x27d8";

const FOREIGN_CHAIN_ID = "0xaa36a7";
```

These constants can then be referenced in the code, as such:

```
const chainId = isHomeRPCUrl(url) ? HOME_CHAIN_ID : FOREIGN_CHAIN_ID;
```

Status
Reported.

Suggestion 10: Rename the Argument _adapters to adaptorSettings (Out of Scope)

Location
evm/contracts/ownable/GiriGiriBashi.sol#L252

Synopsis
In `GiriGiriBashi`'s `initSettings`, an array of adapter settings is passed in. However, the argument is named `_adaptors`.

Mitigation
We recommend renaming the argument to `adaptorSettings` to improve readability.

Status
Reported.

Suggestion 11: Check for Zero Address When Setting Bond Recipient (Out of Scope)

Location

contracts/ownable/GiriGiriBashi.sol#L18

contracts/ownable/GiriGiriBashi.sol#L231

Synopsis

When setting the bond recipient, there is no zero address check validating the correctness of the supplied address, thereby preventing an incorrectly set `bondRecipient` value.

Mitigation

We recommend checking the referenced parameters against the zero address.

Status

Reported.

Suggestion 12: Comment Public Functions as onlyOwner Where Applicable (Out of Scope)

Location

contracts/ownable/ShoyuBashi.sol#L17

contracts/ownable/ShoyuBashi.sol#L22

contracts/ownable/ShoyuBashi.sol#L27

contracts/ownable/GiriGiriBashi.sol#L63

contracts/ownable/GiriGiriBashi.sol#L105

contracts/ownable/GiriGiriBashi.sol#L248

Synopsis

When reading the code, it is not instantly apparent that the functions `enableAdapters`, `disableAdaptors`, and `setThreshold` will reject non-owner access, as the `onlyOwner` modifier is applied to internal functions that are called by the public function.

Mitigation

To improve code readability, we recommend either updating the public function comments to state that they are `onlyOwner`, or applying the `onlyOwner` modifier to the public functions.

Status

Reported.

Suggestion 13: Update Naming of bondRecipient Variable (Out of Scope)

Location

contracts/ownable/GiriGiriBashi.sol#L10

Synopsis

The storage variable `bondRecipient` is intended to be a Hashi-controlled address, which would receive a challenger's bond in the event that the challenge is rejected. Although this functionality can be clearly understood upon reading the code, the `bondRecipient` variable could be better named to more effectively communicate its purpose, as a `bondRecipient` could be interpreted as either the challenging user, or the bridge authority.

Mitigation

We recommend updating the naming of the `bondRecipient` variable to a more descriptive variable name, such as `bondForfeitureAddress`.

Status

Reported.

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.