Least Authority
PRIVACY MATTERS

Vanilla Smart Contracts
Security Audit Report

Equilibrium

Final Report Version: 23 March 2021

Table of Contents

Overview

Background

Equilibrium requested that Least Authority perform a security audit of the Vanilla smart contracts. Vanilla is implemented in three Solidity contracts where all user-facing calls happen via `VanillaRouter`, which inherits trading functionality from UniswapTrader. Profitable trading is rewarded with `VanillaGovernanceToken` ERC-20s.

Project Dates

- **February 16 - 26**: Code review *(Completed)*
- **March 2**: Delivery of Initial Audit Report *(Completed)*
- **March 16 - 17**: Verification Review *(Completed)*
- **March 18**: Final Audit Report delivered *(Completed)*
- **March 23**: Updated Final Audit Report delivered *(Completed)*

Review Team

- Alex Lewis, Security Researcher and Engineer
- David Braun, Security Researcher and Engineer
- Bryan White, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Vanilla smart contracts followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code files are considered in-scope for the review:

- VanillaRouter.sol: https://github.com/vanilladefi/contracts/blob/main/contracts/VanillaRouter.sol
- UniswapTrader.sol: https://github.com/vanilladefi/contracts/blob/main/contracts/UniswapTrader.sol
- VanillaGovernanceToken.sol: https://github.com/vanilladefi/contracts/blob/main/contracts/VanillaGovernanceToken.sol

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

```
9bf4f09a447fa846e8a8586da45c0745ef0896d9
```

For the verification, we examined the Git revision:

```
c3392b9400fbbf142dee0c1cd2d43520532ebac0
```

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

https://github.com/LeastAuthority/equilibrium-vanilla-smart-contracts-audit

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:

- `README.md`
- `contracts/README.md`
- Uniswap V2 Audit Report
- Uniswap Documentation

In addition, we reference the following material in this report:

- D. Que, 2018, "What we learned from auditing the top 20 ERC20 token contracts"

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the smart contracts;
- Potential misuse and gaming of the smart contracts;
- Attacks that impacts funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Economic incentives: ensure token economics (monetary incentives to punish bad behavior and reward good behavior) are functional;
- Denial of Service (DoS) and other security exploits that would impact the smart contracts intended use or disrupt execution;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit smart contracts;
- 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

System Design

The Equilibrium development team has taken a minimal approach to the system design of the Vanilla smart contracts. This design demonstrates a clear and concise separation of concerns, facilitating a more efficient review for potential security issues. The smart contracts rely heavily on functionality provided by Uniswap and avoid the introduction of unnecessary or unused features into the codebase. The design is robust in its consideration for security, which is evident both in the code and at a protocol level.

Overflow Checks

The smart contracts make consistent use of OpenZeppelin's `SafeMath`, providing wrappers over Solidity's arithmetic operations by adding overflow checks. The use of `SafeMath` is an industry standard for preventing a class of bugs that may result in the occurrence of overflows, and prompts a revert in the event of their occurrence.

Limiting Token Supply

The governance token's `mint` function is secured using access control checking, which prevents anyone but the `owner` (i.e., [`VanillaRouter`]) from minting tokens. Since the value of tokens depends on their scarcity, restricting the ability to mint tokens is critical and successfully limits the supply. While the Equilibrium development team's choice to check for access control exhibits considerations for security, it relies solely on trusting that the governance token `owner` will not mint tokens at will. However, given that the router owns the token and it is non-upgradeable, trust concerns around a human controlling the token are eliminated.

More generally, restrictions on minting tokens and limiting supply could be further expanded by having the owner receive a fixed amount of initially minted token when the smart contract is deployed. In addition, while the governance token is currently owned by the router contract, it could be deployed separately and owned by a DAO, or a similar governance contract. We recommend that the Equilibrium development team continue to consider the various security trade-offs resulting from design choices and to prioritize design decisions with lowest impact on the overall security of the smart contracts system.

Non-Upgradeability

The Equilibrium development team has designed the smart contracts such that they are non-upgradeable, which reduces the attack surface by avoiding the introduction of additional functionality that increases the overall complexity of the system. Non-upgradeable smart contracts also consider a longer-term strategy for their use, which can be relied on to be consistent.

However, while non-upgradability can be beneficial from the perspective of secure design, as it minimizes attack surface on the smart contract by limiting functionality, it also comes with trade-offs. For example, in the event that the Equilibrium development team chooses to introduce updated functionality, a new version of the smart contract would need to be deployed and maintained. This would also raise concerns around the inability to migrate the original smart contract holding liquidity and, as a result, users would have to sell tokens and would be unable to transfer them to the new smart contract due to the original smart contract's limited function. Furthermore, this would impact governance token payout and introduce additional complexity, since the new smart contract would have to introduce a new governance token.

As noted previously, while non-upgradeability reduces complexity, thus minimizing the potential for attacks, the design decisions around functionality present other challenges which should be carefully considered. The Equilibrium development team has clearly taken these design trade-offs into consideration and should continue to do so in the future with security in mind.

Mitigating Price Manipulations

The exclusive use of WETH pairs contributes to mitigating the risk of price manipulation. In general, WETH pairs are expected to have higher liquidity, in comparison to other arbitrary ERC-20 token pairs. The expected higher liquidity resulting from the exclusive use of WETH pairs, the incorporation of time into the `_calculateReward` function, and the buy and sell functions being restricted from the same block are all contributing factors to mitigating flash loan attacks, or similar price manipulations.

Malicious Token Attack

We identified a malicious token vulnerability in which a maliciously designed token can trick `UniswapTrader` into allowing more tokens to be sold than were bought, rewarding the attacker with `VanillaGovernanceTokens`, since `UniswapTrader` trusts the `balanceOf` and transfer methods of ERC-20 tokens to behave correctly. However, an attacker can execute and repeat the attack at low costs in order to obtain an unlimited number of tokens. This may result in the decreased value for other token holders since the supply of `VanillaGovernanceTokens` in circulation will increase. In order to immediately mitigate this vulnerability, we recommend allowing only whitelisted tokens to be traded, which would protect `VanillaRouter` from this type of attack. In order to explore a potential remediation, we suggest that the Equilibrium development team investigate further whether it is possible to reduce exposure to the trusted `balanceOf` method by utilizing the return value of Uniswap's `swapExactETHForTokens` instead of using swap when buying token (Issue A).

Code Quality

The code is well organized and generally adheres to Solidity best practices. In addition, the code is sufficiently commented according to the NatSpec guidelines for Solidity comments, which helps reviewers and users of the code to easily understand the inputs and functionality of every method present, and aid in identifying potential issues.

Tests

We encourage the Equilibrium development team to adopt Test Driven Development (TDD), a development approach in which the code's behavior and intended functionality are specified and verified by test cases. The Equilibrium development team currently utilizes industry standard tools for writing smart contract tests, including Hardhat's ethers.js and Waffle. However, test coverage is currently insufficient in that it does not test the existing functionality to verify that the code is behaving as intended and may result in the introduction of potential vulnerabilities. We strongly recommend increasing test coverage to detect and prevent unintended behavior and edge cases, allowing future contributors, maintainers, and reviewers of the codebase a degree of confidence that the existing code functions properly. In doing so, the Equilibrium development team should adopt TDD practices, which should be applied in writing functional tests that provide complete coverage of the smart contract code (Suggestion 4).

During the audit, the Equilibrium development team issued a commit that introduced improvements to the smart contracts, which included the addition of property-based testing. However, given that the tests were introduced following the start of the review, an in depth analysis of the test suite, including the use of property-based tests, was considered out of scope. We recommend that the property-based tests be further evaluated to check that they work as intended. In particular, a broad range of invariants should be tested (e.g. users should always be able to sell and receive their balance from Vanilla's custody; VNL governance tokens should be minted in correlation with the reward function) to ensure comprehensive coverage of the property-based test cases (Suggestion 5).

Documentation

The Vanilla smart contracts use the Uniswap protocol and, as a result, some prior knowledge of Uniswap is assumed in the Vanilla documentation. However, references are provided to Uniswap's documentation where appropriate, contributing to the usability and understanding of the system.

The current project documentation adequately explains the smart contract's core design concepts, providing examples and safety considerations within each section. The documentation of safety considerations (e.g. noting that the use of limits does not completely protect users from front running attacks; links to the Uniswap V2 Audit Report) clearly exhibits transparency about the risks associated with these types of DeFi smart contracts. However, there are several areas where the documentation can be improved, as detailed below.

The inconsistent use of terminology found in the documentation may lead to confusion, increase difficulty in understanding how the systems work, and result in misinterpretations. For example, *Weighted Average Price of Purchases* and *Average Price of Purchases* are used interchangeably throughout the documentation. We recommend defining all terminology and using it consistently throughout the documentation, in addition to consistent use of formatting. Improved consistency will help to avoid confusion for users and developers, as well as misuse of the smart contracts or misunderstanding by security reviewers in their effort to identify security vulnerabilities (Suggestion 2).

The current developer documentation on the public API and events for the `VanillaRouter` is sufficient for developers who intend to use the vanilla router (i.e. calling it from another smart contract) by defining what the smart contracts do. However, the documentation would benefit from information for developers who intend to contribute to the project and need to better understand how the smart contracts work. Given that the smart contracts are able to handle significant amounts of value, additional developer documentation explaining the implementation is critical and would help to increase public confidence in the security and integrity of the smart contracts. As a result, we recommend improving the documentation, which can be further aided by utilizing `solidity-docgen` to generate developer documentation from the existing code comments (Suggestion 3).

Scope

The scope of the audit is sufficient and covered all the core components of the Vanilla smart contracts system.

Our team manually reviewed the Solidity code in addition to running the code against Slither, an open source static analysis framework for Solidity. Slither runs a suite of vulnerability detectors against the codebase, looking for common mistakes and known exploitable vulnerabilities. The results contained some issues, which we further investigated and confirmed to be false positives.

Dependencies

When the dependencies in `package.json`, including `vanilla-contracts`, are published, `npm install` will ignore the `package-lock.json` file. This will result in automatically updating the installed package to newer minor or patch versions of the dependency. We recommend pinning dependencies to exact versions to retain more control over when and where upgrades take place. This will prevent unwanted versions from being inadvertently installed, thus increasing the attack surface. In addition, this will help both developers and reviewers to identify new vulnerabilities discovered in dependencies, which is paramount to the security of the codebase (Suggestion 1).

As previously noted, both Uniswap and OpenZeppelin are utilized by the Vanilla smart contracts and are both well-known and widely used dependencies. The use of trusted and maintained dependencies minimizes the potential security risks and malicious code introduced as a result of utilizing third-party code. However, we recommend that the Equilibrium development team continue to maintain and update dependencies, in addition to checking that they continue to be regularly audited.

Formal Verification

Concurrent distributed systems benefit from formal verification, which is less prone to human error in identifying potential vulnerabilities in the core logic of the smart contracts. We recommend that the Equilibrium development team explore opportunities to conduct formal verification of the smart contracts and the high-level logic (Suggestion 6). While Uniswap has conducted a formal verification, it would demonstrate due diligence to conduct an independent formal verification of the Vanilla smart contracts.

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: Malicious Token Attack	Resolved
Suggestion 1: Use Exact Version Numbers in package.json	Resolved
Suggestion 2: Improve Project Documentation	Unresolved
Suggestion 3: Improve Developer Documentation	Unresolved
Suggestion 4: Increase Test Coverage	Unresolved
Suggestion 5: Check that Property-Based Testing Works as Intended	Unresolved
Suggestion 6: Conduct Formal Verification	Unresolved

Issue A: Malicious Token Attack

Location

An attack demonstrating the vulnerability is found in `test/Issue_A.ts`.

Synopsis

`UniswapTrader` trusts the `balanceOf` and transfer methods of ERC-20 tokens to behave correctly. A maliciously designed token can trick `UniswapTrader` into allowing more tokens to be sold than were bought, rewarding the attacker with `VanillaGovernanceTokens`.

Impact

An attacker can execute the attack at low cost, repeating as often as desired to obtain an unlimited number of `VanillaGovernanceTokens`. This will increase the supply of `VanillaGovernanceTokens` in circulation and potentially lower their value for other token holders.

Preconditions

The attacker must create a malicious token and list it on Uniswap before executing trades via `VanillaRouter`.

Feasibility

The attack is straightforward and can be carried out easily.

Technical Details

In order to receive `VanillaGovernanceTokens`, a trader must make a *profitable* trade. The degree to which a trade is judged profitable is dependent on the number of tokens sold. The profitability of a trade in which the number of tokens sold is greater than the number previously bought (i.e. at a comparable price) increases as the delta between the number of tokens bought and sold increases.

With honest tokens, this attack is not possible. The strategy for tricking `VanillaRouter` with a malicious token consists of the following steps:

1. Buy a certain number (n) of tokens normally.
2. Change the behavior of the ERC-20 token so that before the next trade the `balanceOf` method will return `0`.
3. Buy more tokens. The `UniswapTrader._buyInUniswap` method computes the number of tokens purchased as the difference between the post-trade balance and the pre-trade balance. In normal conditions, the pre-trade balance will be n but the attack has artificially set it to `0`, inflating the perceived number of tokens to be the equal to the post-trade balance.
4. Change the behavior of the ERC-20 token so that, during the next trade, it will allow a transfer amount that exceeds the balance.
5. Sell the inflated perceived number of tokens and receive unearned `VanillaGovernanceTokens`.

Mitigation

`VanillaRouter` can protect itself immediately from this attack by allowing the trade of only whitelisted tokens. It is worth considering to audit tokens for whitelisting in a similar way that Bskt has implemented for their platform.

Remediation

It may be possible to reduce exposure to the trusted `balanceOf` method by utilizing the return value of Uniswap's `swapExactETHForTokens` instead of using swap when buying tokens. Further investigation would be required in order to determine whether this eliminates the vulnerability.

`UniswapTrader._sellInUniswap` should perform its own check to make sure that no more tokens are being sold in a transaction than have been previously purchased rather than relying on the ERC-20 token to enforce this.

Status

The Equilibrium team has addressed the vulnerability by amending the contract to only issue rewards for whitelisted tokens, in accordance with the recommended mitigation.

Verification

Resolved.

Suggestions

Suggestion 1: Pin Dependencies to Specific Versions

Location

`package.json`

Synopsis

The dependencies are listed in `package.json` in the following way:

```
 "dependencies": {

     "@openzeppelin/contracts": "^3.3.0",

     "@uniswap/v2-core": "^1.0.1",
    "@uniswap/v2-periphery": "^1.1.0-beta.0"
  }
```

As a result, if `vanilla-contracts` is published, `npm install` will ignore the `package-lock.json` file and allow the automatic updating of the installed package to newer minor or patch versions of the dependency.

Pinning dependencies to exact versions is a sound approach to retaining more control over when and where upgrades take place. This will prevent unwanted versions from being inadvertently installed, thus minimizing the attack surface. In addition, this will help both developers and reviewers to identify new vulnerabilities discovered in dependencies, which is paramount to the security of the codebase.

Mitigation

We recommend pinning dependencies to exact versions by choosing one of the alternatives options below.

1. Pin `package.json` dependencies to specific versions.

   This option has the effect that all dependency upgrades will go through version control, which has the benefit of making dependency upgrades explicit and transparent. However, it also adds overhead to the development process as it relates to dependency management.

2. Pin `package.json` dependencies to specific versions for the release version only.

   This option leaves the development process unchanged and, instead, imposes a change on the release process, thus moving the overhead of pinning versions to each release.

Status

The Equilibrium team has pinned `package.json` dependencies to specific versions, in accordance with the first option (#1) in the mitigation section.

Verification

Resolved.

Suggestion 2: Improve Project Documentation

Location

`contracts/README.md`

Synopsis

We have identified several examples where the project documentation can be improved. In particular, there are instances throughout the documentation of terms being used inconsistently or without rigorous definitions. We describe some examples below:

1. “Weighted Average Price of Purchases” and “Average Price of Purchases” are used interchangeably. In addition, the structure of the document is not always consistent, as “Weighted Average Price of Purchases” is discussed both under *Price Calculations* and *Profit*. This introduces additional complexity in understanding how the system works and may lead to misiterpretations.

2. The excerpt below contains two facts without a clear logical link. The first half of the second sentence in the quote does not contribute to the explanation and may confuse the reader with

unrelated information and without additional context.

"Profit is calculated whenever a user sells tokens. Knowing the selling price and the average purchasing price (`ethSum/tokenSum`), the profit is calculated as `receivedEth - expectedEth` where `expectedEth = ethSum*tokensSold/tokenSum`. Basically, if the user received more Ether than was expected, a profit was made."

3. "Profit" is referenced as a defined term in formulae, with the definition described as multiple formulae combined in a prose paragraph. We recommend consistently defining terms using a consistent format including a complete formula and a separate description.

4. Several terms (e.g. `ethSum`, `tokenSum`) are used in examples, and then later referenced in other definitions. These should be clearly defined terms in their own right and then referenced by both the examples and other terms.

5. The term "reserve limit" is used without definition. For example:

   > "L is the immutable WETH reserve limit that is set when the `VanillaRouter` is deployed."

6. The variable P is used in both the *Tokens* and *Value Protection Coefficient* sections. Both of these section reference the same section for the definition of "Profit", but they provide slightly different descriptions in context:

   > "P is the absolute profit, which sets the theoretical maximum reward for any single trade." (*Tokens*).
   > vs.
   > "P is the absolute profit in Ether." (*Value Protection Coefficient*)

   In the *Tokens* section, "reward" is understood to be in terms of VNL. The use of "maximum reward" in the definition of P (which is in terms of ETH), as in the subsequent usage, is confusing without stating this explicitly .

Mitigation

We recommend using terms consistently throughout the documentation. In addition, we suggest re-structuring the text by grouping related terminology together. In addition, every term and variable should be defined, preferably in an unambiguous and formal way.

Status

The Equilibrium team has acknowledged that the documentation can be improved and have stated their intention to focus a specific iteration on making updates to the documentation as suggested, including improving the consistency of the terminology and the structure of the text. These changes were not implemented at the time of this verification.

Verification

Unresolved.

Suggestion 3: Improve Developer Documentation

Location

`contracts/README.md#api`

Synopsis

The current developer documentation on the public API and events for the `VanillaRouter` is lacking important information for developers who intend to contribute to the project.

For example, several references are made to internal APIs in prose explanations in the documentation (e.g. `_executeBuy`, `_executeSell`, `_tokenPriceData`). However, these functions are not documented in the developer documentation of the APIs:

> "Using this formula, Vanilla can keep track of average prices in a gas-efficient and fair way. VanillaRouter implements the price calculation logic in internal functions `_executeBuy` and `_executeSell` and keeps track of the exchange volumes in a `_tokenPriceData` mapping."

Furthermore, the documentation is missing information on the deployment process, both for the router and the system as a whole. This leads to difficulty in understanding and using the system, and increases the likelihood of human error.

Given that the smart contracts are able to handle significant amounts of value, additional developer documentation explaining the implementation is critical and would help to increase public confidence in the security and integrity of the contracts.

Mitigation

We recommend improving the developer documentation, such that all functions are described consistently in the API documentation, and the deployment process is thoroughly documented. This effort can be further aided by utilizing `solidity-docgen` to generate developer documentation from the existing code comments.

Status

The Equilibrium team has responded that they agree with the recommendation to improve the developer documentation. However, they believe that the internal functions should be described in the technical documentation, in order to point out how specific algorithms are implemented in Solidity, as opposed to the API documentation.

In addition, they have stated that they intend to improve the developer documentation in an effort to alleviate any difficulty in understanding and using the system by more thoroughly documenting the deployment process and experimenting with `solidity-docgen`. These changes were not implemented at the time of this verification.

Verification

Unresolved.

Suggestion 4: Increase Test Coverage

Location

/tree/main/test

Synopsis

At the beginning of the audit, the code base only included smoke tests with limited code coverage. For example, the tests did not cover the complex reward function. Unit tests provide reviewers with a thorough understanding of the function, as it allows visibility into the values returned for several use cases. The absence of adequate test coverage decreases the efficiency of a review, as time is spent writing tests to understand how the system is expected to work.

During the audit, the Equilibrium development team pushed a commit that provided improvements to test coverage, including coverage for the `calculate reward` function. However, adequate test coverage has not yet been achieved. Tests should account for all success, failure, and edge case scenarios. This helps to detect and prevent unintended behavior and edge cases, which may result in the potential vulnerabilities.

Mitigation

We recommend extending test coverage to account for the entire system. Additionally, we recommend tests be expanded to include expected failure cases and edge cases in order to help determine unexpected outcomes. We encourage the Equilibrium development team to adopt TDD, an approach in which the code's behavior and intended functionality is specified and verified by test cases. This approach should be applied in writing functional tests that cover the entirety of the contract code.

Status

The Equilibrium team acknowledges the need for increased test coverage and have stated their intention to increase tests in the future in order to achieve sufficient coverage. These changes were not implemented at the time of this verification.

Verification

Unresolved.

Suggestion 5: Check that Property-Based Testing Works as Intended

Location

/tree/main/test

Synopsis

As previously noted (see Suggestion 4), the Equilibrium development team issued a commit during the audit that introduced testing improvement to the smart contracts, which included the addition of property-based testing. However, given that the tests were introduced following the start of the review, an in depth analysis of the property-based tests was considered out of scope.

Mitigation

We recommend that the property-based tests be further evaluated to check that they work as intended. In particular, a broad range of invariants should be tested (e.g. users should always be able to sell and receive their balance from Vanilla's custody; VNL governance tokens should be minted in correlation with the reward function) to ensure comprehensive coverage of the property-based tests.

Status

The Equilibrium team has responded in agreement with the recommendation to check that property-based tests work as intended and have stated their intention to continue verifying a wider range of system properties. These changes were not implemented at the time of this verification.

Verification

Unresolved.

Suggestion 6: Conduct Formal Verification

Location

`main/contracts`

Synopsis

In addition to manual code reviews, concurrent distributed systems benefit from formal verification, which is less prone to human error and is likely to uncover potential security issues.

Mitigation

Create a formal specification and implement formal verification of the Vanilla smart contracts system. For example, a similar use case for MakerDAO has been implemented in collaboration with Runtime Verification.

Status

The Equilibrium team has responded in agreement with the recommendation to conduct formal verification but have not specified plans to have formal verification conducted.

Verification

Unresolved.

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 C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. 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. Although we are a small team, we believe that 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 included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site 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. While 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, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, 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 though we have not yet 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 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 report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, 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 resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.