Zeebu Smart Contracts

Security Assessment (Summary Report)

September 9, 2024

Prepared for:
Mrudul Chauhan
Zeebu

Prepared by: **Elvis Skoždopolj and Samuel Moelius**

About Trail of Bits

Founded in 2012 and headquartered in New York, Trail of Bits provides technical security assessment and advisory services to some of the world's most targeted organizations. We combine high-end security research with a real-world attacker mentality to reduce risk and fortify code. With 100+ employees around the globe, we've helped secure critical software elements that support billions of end users, including Kubernetes and the Linux kernel.

We maintain an exhaustive list of publications at https://github.com/trailofbits/publications, with links to papers, presentations, public audit reports, and podcast appearances.

In recent years, Trail of Bits consultants have showcased cutting-edge research through presentations at CanSecWest, HCSS, Devcon, Empire Hacking, GrrCon, LangSec, NorthSec, the O'Reilly Security Conference, PyCon, REcon, Security BSides, and SummerCon.

We specialize in software testing and code review projects, supporting client organizations in the technology, defense, and finance industries, as well as government entities. Notable clients include HashiCorp, Google, Microsoft, Western Digital, and Zoom.

Trail of Bits also operates a center of excellence with regard to blockchain security. Notable projects include audits of Algorand, Bitcoin SV, Chainlink, Compound, Ethereum 2.0, MakerDAO, Matic, Uniswap, Web3, and Zcash.

To keep up to date with our latest news and announcements, please follow @trailofbits on Twitter and explore our public repositories at https://github.com/trailofbits. To engage us directly, visit our "Contact" page at https://www.trailofbits.com/contact, or email us at info@trailofbits.com.

Trail of Bits, Inc.
497 Carroll St., Space 71, Seventh Floor
Brooklyn, NY 11215
https://www.trailofbits.com
info@trailofbits.com

Notices and Remarks

Copyright and Distribution

The sole canonical source for Trail of Bits publications is the Trail of Bits Publications page. Reports accessed through any source other than that page may have been modified and should not be considered authentic.

Test Coverage Disclaimer

All activities undertaken by Trail of Bits in association with this project were performed in accordance with a statement of work and agreed upon project plan.

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

Trail of Bits uses automated testing techniques to rapidly test the controls and security properties of software. These techniques augment our manual security review work, but each has its limitations: for example, a tool may not generate a random edge case that violates a property or may not fully complete its analysis during the allotted time. Their use is also limited by the time and resource constraints of a project.

Table of Contents

Project Summary

Contact Information

The following project manager was associated with this project:

Sam Greenup, Project Manager
sam.greenup@trailofbits.com

The following engineering director was associated with this project:

Josselin Feist, Engineering Director, Blockchain
josselin.feist@trailofbits.com

The following consultants were associated with this project:

Elvis Skoždopolj, Consultant
elvis.skozdopolj@trailofbits.com

Samuel Moelius, Consultant
samuel.moelius@trailofbits.com

Project Timeline

The significant events and milestones of the project are listed below.

Date	Event
July 17, 2024	Pre-project kickoff call
July 26, 2024	Delivery of report draft
July 26, 2024	Report readout meeting
September 9, 2024	Delivery of summary report

Project Targets

The engagement involved a review and testing of the targets listed below.

Zeebu contracts

Repository	https://github.com/TechnologyZeebu/Zeebu-contracts-testnet
Version	`0cb5073c0690886b55a4e0c2bf93f9354bfd18f5`
Type	Solidity
Platform	Ethereum

Zeebu waitlist contracts

Repository	https://github.com/TechnologyZeebu/Waitlist-testnet
Version	`286638a085d0c53ef92aeb87ed9b7d06e410dfee`
Type	Solidity
Platform	Ethereum

Executive Summary

Engagement Overview

Zeebu engaged Trail of Bits to review the security of their locking and reward distribution (286638a) and user waitlist smart contracts (0cb5073). The locking contract is a fork of the Curve/Balancer `VotingEscrow` that allows users to lock tokens in exchange for voting power. This voting power balance is used to determine the amount of rewards a user gets during reward distributions. The user waitlist contracts allow users to register and refer other users to the app, earning reward points for each registration made with their referral code.

A team of two consultants conducted the review from July 22 to July 26, 2024, for a total of two engineer-weeks of effort. With full access to source code and documentation, we performed static and dynamic testing of the targets, using automated and manual processes.

Observations and Impact

We reviewed Solidity and Vyper contracts in two repositories. The first repository (`Zeebu-contracts-testnet`) contains code forked from Curve/Balancer with some minor modifications. Our review focused on the new code added to these contracts and any other contracts that interact with them. While we reviewed all parts of the code, we applied less scrutiny to the unchanged forked code. The second repository (`Waitlist-testnet`) contains contracts to manage a user waiting list.

We focused on investigating issues that could lead to loss of funds (such as reentrancy attacks), insufficient and incorrect signature verification, and logic issues related to the calculation and distribution of rewards. We discovered multiple issues that would allow users or administrators to steal funds or circumvent access control (TOB-ZEB-1, TOB-ZEB-7, TOB-ZEB-8, TOB-ZEB-11, TOB-ZEB-15), most of which could have been caught by employing better testing practices. We also discovered lower-severity design and logic issues related to user suspension, claiming of user rewards, incorrect validation, and potential denial of service due to out-of-gas errors.

The `Zeebu-contracts-testnet` repository has markedly higher code quality than the `Waitlist-testnet` repository. The former has good documentation and a clear naming style, which are missing from the latter. Both repositories' testing suites lack full coverage over the expected user paths, indicating that the team's development practices can be significantly improved.

Recommendations

Based on the codebase maturity evaluation and findings identified during the security review, Trail of Bits recommends that Zeebu take the following steps before deploying the contracts:

- **Remediate the findings disclosed in this report.** These findings should be addressed as part of a direct remediation or as part of any refactor that may occur when addressing other recommendations.

- **Improve the testing suite.** We recommend that the Zeebu team define a clear testing strategy and create guidelines on how testing is performed in the codebase. The testing suite should have full coverage over all of the contracts and code paths in the system. More detailed guidance on improving testing practices can be found in appendix D.

- **Improve development practices.** The level of maturity between the two codebases differs, indicating that development practices can be more clearly defined. We recommend that the Zeebu team work on improving the consistency of code styling (Solidity Style Guide, Coinbase Style Guide); create guidelines and standards around developer- and user-facing documentation; create a testing strategy; and implement automated processes (e.g., running the testing suite in CI, using Slither) to decrease the chance of bugs being introduced during development.

Codebase Maturity Evaluation

Trail of Bits uses a traffic-light protocol to provide each client with a clear understanding of the areas in which its codebase is mature, immature, or underdeveloped. Deficiencies identified here often stem from root causes within the software development life cycle that should be addressed through standardization measures (e.g., the use of common libraries, functions, or frameworks) or training and awareness programs.

Category	Summary	Result
Arithmetic	The `VotingEscrow` checkpointing system contains complex arithmetic; however, this part of the contract is identical to the original Curve/Balancer code. The rest of the codebase relies on simple arithmetic formulas that are well documented. We did not find any arithmetic issues during the review. The codebase relies on Solidity >=0.8.0 native overflow protection.	**Satisfactory**
Auditing	We found a couple of functions where event emission could be added in order to improve the monitorability of the system; however, the system as a whole does a good job of emitting events for critical state updates. The contracts contain `SphereX` proxies, which indicates that this monitoring and threat detection tool will be used to actively monitor the system. The efficacy of this tool and its use was not part of this review.	**Satisfactory**
Authentication / Access Controls	Although most of the access control in the system is robust, the signature verification done in the `UserWaitList` contract is flawed, allowing a single valid signer to execute this function as long as an admin signature is provided. We did not find any access control issues that do not involve signatures.	**Moderate**
Complexity Management	As provided, the code in the `Zeebu-contracts-testnet` repo does not build. Both repos lack CI. The repos feature numerous inaccurate comments and error messages. Rather than refer to libraries directly, the repos copy code from libraries. Code is duplicated. There is significant dead code (e.g., unused contracts, fields, and modifiers). Several mutable fields could be made constant or immutable.	**Weak**

Cryptography and Key Management	The codebase uses hashing in two places. In both cases, data that should be included in the hash is not, leaving the contracts vulnerable.	**Weak**
Decentralization	Most of the contracts in the system are upgradeable by a single entity, it is not documented if this entity will be an EOA account, a multisignature wallet, or governance. The privileged roles in the contracts can update important system parameters, which have an immediate effect and could impact user funds. The Zeebu team could benefit from clearly documenting the roles inside of the system, the privileged actions, and when such actions would be taken. Any inherent risks of interacting with the system should be a part of the user-facing documentation.	**Weak**
Documentation	As provided, the code in the `Zeebu-contracts-testnet` repo contains thorough NatSpec documentation; a PDF that outlines the differences between the Zeebu team's modified VotingEscrow contract and the original source; and documentation explaining each of the components in the repo. Clearly linking the original source of the code in the document would be beneficial. However, the code in the `Waitlist-testnet` repo does not contain any external documentation and very sparse inline documentation. There is no documentation on the nature of the privileged roles nor on when privileged actions would be taken. The Zeebu team could benefit by defining clear standards and processes around the creation and maintenance of external and inline documentation of all contracts and actions. The privileged roles in the system should be documented, along with expected user flows when interacting with the system.	**Moderate**
Low-Level Manipulation	While there is some use of low-level manipulation in the dependencies, the system contracts themselves use it very sparingly. We did not discover any issues related to low-level manipulation or use of assembly.	**Satisfactory**

Testing and Verification	Because the `Waitlist-testnet` repo does not include a `package.json` file, its tests cannot be run. When the necessary files are added, one of the tests fails. The `Zeebu-contracts-testnet` repo uses hardhat and contains function tests that can be run with minimal modification. However, the testing suite does not have full coverage over all of the features of the system and no advanced testing techniques are used. Additionally, some core contracts are mocked, which can reduce the efficacy of the tests and potentially make issues harder to discover. The Zeebu team could benefit from defining a clear testing strategy and workflow which includes both unit and integration tests. More guidance on testing practices can be found in appendix D.	**Weak**
Transaction Ordering	No transaction ordering risks were identified during the review.	**Satisfactory**

Summary of Findings

The table below summarizes the findings of the review, including type and severity details.

ID	Title	Type	Severity
1	Reentrancy in `claim` allows to claim extra rewards	Undefined Behavior	**High**
2	Missing zero checks	Data Validation	**Informational**
3	`VotingEscrow` administrator is not required to accept role	Undefined Behavior	**Informational**
4	No way to run `Waitlist-testnet` tests	Testing	**Informational**
5	Multiple referral codes can point to the same user	Undefined Behavior	**Informational**
6	`SafeERC20` is not used	Undefined Behavior	**Informational**
7	`isValidAdminSigner` does not take amount as an argument	Cryptography	**High**
8	`isValidWithdrawSigner` does not take a nonce as an argument	Cryptography	**Medium**
9	`lastRedemptionTime` is never updated	Undefined Behavior	**Low**
10	Being suspended has no downsides	Access Controls	**Undetermined**
11	Single withdraw signer can trigger a withdrawal due to incorrect check	Access Controls	**Low**
12	Referrals are never removed, which can lead to denial of service	Denial of Service	**Low**

13	Code treats `UserWaitList` as a user when `allowFromContractOnly` is false	Undefined Behavior	**Undetermined**
14	Users' email addresses are leaked	Undefined Behavior	**Informational**
15	Users can drain the contract reward balance by referring their own accounts and reusing their token balance	Data Validation	**High**
16	`suspendUser` and `unsuspendUser` update the wrong referrer	Data Validation	**Low**

A. Vulnerability Categories

The following tables describe the vulnerability categories, severity levels, and difficulty levels used in this document.

Vulnerability Categories	
Category	**Description**
Access Controls	Insufficient authorization or assessment of rights
Auditing and Logging	Insufficient auditing of actions or logging of problems
Authentication	Improper identification of users
Configuration	Misconfigured servers, devices, or software components
Cryptography	A breach of system confidentiality or integrity
Data Exposure	Exposure of sensitive information
Data Validation	Improper reliance on the structure or values of data
Denial of Service	A system failure with an availability impact
Error Reporting	Insecure or insufficient reporting of error conditions
Patching	Use of an outdated software package or library
Session Management	Improper identification of authenticated users
Testing	Insufficient test methodology or test coverage
Timing	Race conditions or other order-of-operations flaws
Undefined Behavior	Undefined behavior triggered within the system

Severity Levels	
Severity	**Description**
Informational	The issue does not pose an immediate risk but is relevant to security best practices.
Undetermined	The extent of the risk was not determined during this engagement.
Low	The risk is small or is not one the client has indicated is important.
Medium	User information is at risk; exploitation could pose reputational, legal, or moderate financial risks.
High	The flaw could affect numerous users and have serious reputational, legal, or financial implications.

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploitation was not determined during this engagement.
Low	The flaw is well known; public tools for its exploitation exist or can be scripted.
Medium	An attacker must write an exploit or will need in-depth knowledge of the system.
High	An attacker must have privileged access to the system, may need to know complex technical details, or must discover other weaknesses to exploit this issue.

B. Code Maturity Categories

The following tables describe the code maturity categories and rating criteria used in this document.

Code Maturity Categories	
Category	**Description**
Arithmetic	The proper use of mathematical operations and semantics
Auditing	The use of event auditing and logging to support monitoring
Authentication / Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system
Complexity Management	The presence of clear structures designed to manage system complexity, including the separation of system logic into clearly defined functions
Cryptography and Key Management	The safe use of cryptographic primitives and functions, along with the presence of robust mechanisms for key generation and distribution
Decentralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Documentation	The presence of comprehensive and readable codebase documentation
Low-Level Manipulation	The justified use of inline assembly and low-level calls
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage
Transaction Ordering	The system's resistance to transaction-ordering attacks

Rating Criteria	
Rating	**Description**
Strong	No issues were found, and the system exceeds industry standards.
Satisfactory	Minor issues were found, but the system is compliant with best practices.
Moderate	Some issues that may affect system safety were found.

Weak	Many issues that affect system safety were found.
Missing	A required component is missing, significantly affecting system safety.
Not Applicable	The category is not applicable to this review.
Not Considered	The category was not considered in this review.
Further Investigation Required	Further investigation is required to reach a meaningful conclusion.

C. Token Integration Checklist

The following checklist provides recommendations for interactions with arbitrary tokens. Every unchecked item should be justified and its associated risks understood. For an up-to-date version of the checklist, see `crytic/building-secure-contracts`.

For convenience, all Slither utilities can be run directly on a token address, such as the following:

```
slither-check-erc 0xdac17f958d2ee523a2206206994597c13d831ec7 TetherToken --erc erc20
slither-check-erc 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d KittyCore --erc erc721
```

To follow this checklist, use the following output from Slither for the token:

```
slither-check-erc [target] [contractName] [optional: --erc ERC_NUMBER]
slither [target] --print human-summary
slither [target] --print contract-summary
slither-prop . --contract ContractName # requires configuration, and use of Echidna
and Manticore
```

General Considerations

- ❏ **The contract has a security review.** Avoid interacting with contracts that lack a security review. Check the length of the assessment (i.e., the level of effort), the reputation of the security firm, and the number and severity of the findings.

- ❏ **You have contacted the developers.** You may need to alert their team to an incident. Look for appropriate contacts on `blockchain-security-contacts`.

- ❏ **They have a security mailing list for critical announcements.** Their team should advise users when critical issues are found or when upgrades occur.

Contract Composition

- ❏ **The contract avoids unnecessary complexity.** The token should be a simple contract; a token with complex code requires a higher standard of review. Use Slither's `human-summary` printer to identify complex code.

- ❏ **The contract uses `SafeMath` or Solidity 0.8.0+.** Contracts that do not use `SafeMath` require a higher standard of review. Inspect the contract by hand for `SafeMath`/Solidity 0.8.0+ usage.

- ❏ **The contract has only a few non-token-related functions.** Non-token-related functions increase the likelihood of an issue in the contract. Use Slither's `contract-summary` printer to broadly review the code used in the contract.

- ❏ **The token has only one address.** Tokens with multiple entry points for balance updates can break internal bookkeeping based on the address (e.g., `balances[token_address][msg.sender]` may not reflect the actual balance).

Owner Privileges

- ❏ **The token is not upgradeable.** Upgradeable contracts may change their rules over time. Use Slither's `human-summary` printer to determine whether the contract is upgradeable.

- ❏ **The owner has limited minting capabilities.** Malicious or compromised owners can misuse minting capabilities. Use Slither's `human-summary` printer to review minting capabilities, and consider manually reviewing the code.

- ❏ **The token is not pausable.** Malicious or compromised owners can trap contracts relying on pausable tokens. Identify pausable code by hand.

- ❏ **The owner cannot denylist the contract.** Malicious or compromised owners can trap contracts relying on tokens with a denylist. Identify denylisting features by hand.

- ❏ **The team behind the token is known and can be held responsible for misuse.** Contracts with anonymous development teams or teams that reside in legal shelters require a higher standard of review.

ERC-20 Tokens

ERC-20 Conformity Checks

Slither includes a utility, `slither-check-erc`, that reviews the conformance of a token to many related ERC standards. Use `slither-check-erc` to review the following:

- ❏ **`Transfer` and `transferFrom` return a Boolean.** Several tokens do not return a Boolean on these functions. As a result, their calls in the contract might fail.

- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-20 standard and may not be present.

- ❏ **`Decimals` returns a `uint8`.** Several tokens incorrectly return a `uint256`. In such cases, ensure that the value returned is less than 255.

- ❏ **The token mitigates the known ERC-20 race condition.** The ERC-20 standard has a known ERC-20 race condition that must be mitigated to prevent attackers from stealing tokens.

Slither includes a utility, `slither-prop`, that generates unit tests and security properties that can discover many common ERC flaws. Use `slither-prop` to review the following:

- ❏ **The contract passes all unit tests and security properties from `slither-prop`.** Run the generated unit tests and then check the properties with Echidna and Manticore.

Risks of ERC-20 Extensions

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **The token is not an ERC-777 token and has no external function call in `transfer` or `transferFrom`.** External calls in the transfer functions can lead to reentrancies.

- ❏ **`Transfer` and `transferFrom` should not take a fee.** Deflationary tokens can lead to unexpected behavior.

- ❏ **Potential interest earned from the token is accounted for.** Some tokens distribute interest to token holders. This interest may be trapped in the contract if not accounted for.

Token Scarcity

Reviews of token scarcity issues must be executed manually. Check for the following conditions:

- ❏ **The supply is owned by more than a few users.** If a few users own most of the tokens, they can influence operations based on the tokens' repartition.

- ❏ **The total supply is sufficient.** Tokens with a low total supply can be easily manipulated.

- ❏ **The tokens are in more than a few exchanges.** If all the tokens are in one exchange, a compromise of the exchange could compromise the contract relying on the token.

- ❏ **Users understand the risks associated with a large amount of funds or flash loans.** Contracts relying on the token balance must account for attackers with a large amount of funds or attacks executed through flash loans.

- ❏ **The token does not allow flash minting.** Flash minting can lead to substantial swings in the balance and the total supply, which necessitate strict and comprehensive overflow checks in the operation of the token.

ERC-721 Tokens

ERC-721 Conformity Checks

The behavior of certain contracts may differ from the original ERC specification. Conduct a manual review of the following conditions:

- ❏ **Transfers of tokens to the `0x0` address revert.** Several tokens allow transfers to `0x0` and consider tokens transferred to that address to have been burned; however, the ERC-721 standard requires that such transfers revert.
- ❏ **`safeTransferFrom` functions are implemented with the correct signature.** Several token contracts do not implement these functions. A transfer of NFTs to one of these contracts can result in a loss of assets.
- ❏ **The `name`, `decimals`, and `symbol` functions are present if used.** These functions are optional in the ERC-721 standard and may not be present.
- ❏ **If it is used, `decimals` returns a `uint8(0)`.** Other values are invalid.
- ❏ **The `name` and `symbol` functions can return an empty string.** This behavior is allowed by the standard.
- ❏ **The `ownerOf` function reverts if the `tokenID` is invalid or is set to a token that has already been burned.** The function cannot return `0x0`. This behavior is required by the standard, but it is not always properly implemented.
- ❏ **A transfer of an NFT clears its approvals.** This is required by the standard.
- ❏ **The `tokenID` of an NFT cannot be changed during its lifetime.** This is required by the standard.

Common Risks of the ERC-721 Standard

To mitigate the risks associated with ERC-721 contracts, conduct a manual review of the following conditions:

- ❏ **The `onERC721Received` callback is accounted for.** External calls in the transfer functions can lead to reentrancies, especially when the callback is not explicit (e.g., in `safeMint` calls).
- ❏ **When an NFT is minted, it is safely transferred to a smart contract.** If there is a minting function, it should behave like `safeTransferFrom` and properly handle the minting of new tokens to a smart contract. This will prevent a loss of assets.
- ❏ **The burning of a token clears its approvals.** If there is a burning function, it should clear the token's previous approvals.

D. Testing Improvement Recommendations

This appendix aims to provide general recommendations on improving processes and enhancing the quality of the Zeebu smart contracts test suite.

Identified Testing Deficiencies

During the audit, we encountered a number of issues that could have been prevented or minimized through better test practices and improved test coverage (e.g., TOB-ZEB-6, TOB-ZEB-7, TOB-ZEB-9). We identified several deficiencies in the test suite that could make further testing and development more difficult and thereby reduce the likelihood that the test suite will find security issues:

- The test suite does not fully cover all of the contracts and functions in the codebase.
- The `VotingEscrow` and `Launchpad` contracts are mocked instead of using the contracts themselves, which can make issues harder to discover.
- The test suite contains only unit tests; integration and fuzz tests are not present.
- The test suite mostly contains positive unit tests (i.e., expected behavior), with a low amount of negative and adversarial unit tests (i.e., unexpected behavior and common attack vectors).

To address these deficiencies and improve the test coverage and processes, we recommend that the Zeebu team define a clear testing strategy and create guidelines on how testing is performed in the codebase. Our general guidelines for improving test suite quality are as follows:

1. **Define a clear test directory structure.** A clear directory structure helps organize the work of multiple developers, makes it easier to identify which components and behaviors are being tested, and gives insight into the overall test coverage.
2. **Write a design specification of the system, its components, and its functions in plain language.** Defining a specification can allow the team to more easily detect bugs and inconsistencies in the system, reduce the likelihood that future code changes will introduce bugs, improve the maintainability of the system, and allow the team to create a robust and holistic testing strategy.
3. **Use the function specifications to guide the creation of unit tests.** Creating a specification of all preconditions, postconditions, failure cases, entrypoints, and execution paths for a function will make it easier to maintain high test coverage and identify edge cases.

4. **Use the interaction specifications to guide the creation of integration tests.** An interaction specification will make it easier to identify the interactions that need to be tested and the external failure cases that need to be validated or guarded against, and it will help identify issues related to access controls and external calls.

5. **Use fork testing for integration testing with third-party smart contracts and to ensure that the deployed system works as expected.** Fork testing can be used to test interactions between the protocol contracts and third-party smart contracts by providing an environment that is as close to production as possible. Additionally, fork testing can be used to identify whether the deployed system is behaving as expected.

6. **Implement fuzz testing by first defining a set of system- and function-level invariants and then testing them with Echidna, Foundry, and/or Medusa.** Fuzz testing is a powerful technique for exposing security vulnerabilities and finding edge cases that are unlikely to be found through unit testing or manual review. Fuzz testing can be done on a single function by passing in randomized arguments, and on an entire system or on specific components by generating a sequence of random calls to various functions inside the system or component. Both testing approaches should be applied using one or multiple smart contract fuzzers.

7. **Use mutation testing to identify gaps in the test coverage and more easily identify bugs in the code.** Mutation testing can help identify coverage gaps in unit tests and help discover security vulnerabilities. Taking a two-pronged approach using Necessist to mutate tests and universalmutator to mutate source code can prove valuable in creating a robust test suite.

Directory Structure

Creating a specific directory structure for the system's tests will make it easier to develop and maintain the test suite and find coverage gaps. This section contains brief guidelines on defining a directory structure.

- Create individual directories for each test type (e.g., `unit/`, `integration/`, `fork/`, `fuzz/`) and for the utility contracts. The individual directories can be further divided into directories based on components or behaviors being tested.

- Create a single base contract that inherits from the shared utility contracts and is inherited by individual test contracts. This will help reduce code duplication across the test suite.

- Create a clear naming convention for test files and test functions to make it easier to filter tests and understand the properties or contracts that are being tested.

Unit Testing

We provide the following general recommendations based on our findings:

- **Define a specification for each function** and use it to guide the development of the unit tests.
- **Improve the unit tests' coverage** so that they test all functions and contracts in the codebase. Use coverage reports and mutation testing to guide the creation of additional unit tests.
- **Use positive unit tests** to test that functions and components behave as expected. Ideally, each unit test should test a single property, with additional unit tests testing for edge cases. The unit test should test that all expected side effects are correct.
- **Improve the use of negative unit tests** by testing for specific failure cases and common adversarial situations.
- **Reduce the use of mock contracts.** While using mock contracts for simple contracts (e.g., ERC20 tokens) can save time, mocking core contracts of the system can hide issues and make the testing suite less effective.

Integration and Fork Testing

Integration tests build on unit tests by testing how individual components integrate with each other or with third-party contracts. It can often be useful to run integration testing on a fork of the network to make the testing environment as close to production as possible and to minimize the use of mock contracts whose implementation can differ from third-party contracts. We provide the following general recommendations on performing integration and fork testing:

- **Use the interaction specification to develop integration tests.** Ensure that the integration tests aid in verifying the interactions specification.
- **Identify valuable input data for the integration tests** that can maximize code coverage and test potential edge cases.
- **Use negative integration tests**, similar to negative unit tests, to test common failure cases.
- **Use fork testing to build on top of the integration testing suite.** Fork testing will aid in testing third-party contract integrations and in testing the proper configuration of the system once it is deployed.
- **Enrich the forked integration test suite with fuzzed values and call sequences** (refer to the Fuzz Testing recommendations below). This will aid in increasing code coverage, validating system-level invariants, and identifying edge cases.

Fuzz Testing

Fuzz testing, also known as fuzzing, is an automated testing technique that involves testing program behavior with a large number of inputs and call sequences to discover bugs and vulnerabilities. It can help identify arithmetic errors such as precision loss, logical errors such as insufficient access controls, and other unexpected edge cases that may be difficult to discover through unit testing or manual review. We provide the following general recommendations on performing fuzz testing:

- **Define system- and function-level invariants.** Invariants are properties that should always hold within a system, component, or function. Defining invariants is a prerequisite for developing effective fuzz tests that can detect unexpected behavior. Developing a robust system specification will directly aid in the identification of system- and function-level invariants.

- **Improve the fuzz testing coverage.** When using Echidna or Medusa, regularly review the coverage files generated at the end of a run to determine whether the property tests' assertions are reached and what parts of the codebase are explored by the fuzzer. To improve the fuzzer's exploration and increase the chances that it finds an unexpected edge case, avoid overconstraining the function arguments.

- **Integrate fuzz testing into the CI/CD workflow.** Continuous fuzz testing can help quickly identify any code changes that will result in a violation of a system property, and it forces developers to update the fuzz test suite in parallel with the code. Running fuzz campaigns stochastically may cause a divergence between the operations in the code and the fuzz tests.

- **Add comprehensive logging mechanisms to all fuzz tests to aid in debugging.** Logging during smart contract fuzzing is crucial for understanding the state of the system when a system property is broken. Without logging, it is difficult to identify the arithmetic or operation that caused the failure.

- **Enrich each fuzz test with comments explaining the preconditions and postconditions of the test.** Strong fuzz testing requires well-defined preconditions (for guiding the fuzzer) and postconditions (for properly testing the invariant[s] in question). Comments explaining the bounds on certain values and the importance of the system properties being tested will aid in test suite maintenance and debugging efforts.

Mutation Testing

At a high level, mutation tests make several changes to each line of a target file and rerun the test suite for each change. Changes that result in test failures indicate adequate test coverage, while changes that do not result in test failures indicate gaps in the test coverage. Although mutation testing is a slow process, it allows auditors to focus their review on areas of the codebase that are most likely to contain latent bugs, and it allows developers to identify and add missing tests.

We recommend using two mutation tools, both of which can help detect redundant code, insufficient test coverage, incorrectly defined tests or conditions, and bugs in the underlying source code being tested:

- Necessist performs mutation of the testing suite by iteratively removing lines in the test cases.
- universalmutator performs mutation of the underlying source code.
- slither-mutate performs mutation of the underlying source code.

E. Fix Review Results

When undertaking a fix review, Trail of Bits reviews the fixes implemented for issues identified in the original report. This work involves a review of specific areas of the source code and system configuration, not comprehensive analysis of the system.

On August 19, 2024, Trail of Bits reviewed the fixes and mitigations implemented by the Zeebu team for the issues identified in this report. We reviewed each fix to determine its effectiveness in resolving the associated issue.

In summary, of the 16 issues described in this report, Zeebu has resolved 13 issues and has not resolved the remaining three issues. For additional information, please see the Detailed Fix Review Results below.

ID	Title	Status
1	Reentrancy in `claim` allows to claim extra rewards	Resolved
2	Missing zero checks	Unresolved
3	`VotingEscrow` administrator is not required to accept role	Resolved
4	No way to run `Waitlist-testnet` tests	Resolved
5	Multiple referral codes can point to the same user	Resolved
6	`SafeERC20` is not used	Resolved
7	`isValidAdminSigner` does not take amount as an argument	Resolved
8	`isValidWithdrawSigner` does not take a nonce as an argument	Resolved
9	`lastRedemptionTime` is never updated	Resolved
10	Being suspended has no downsides	Resolved
11	Single withdraw signer can trigger a withdrawal due to incorrect check	Resolved
12	Referrals are never removed, which can lead to denial of service	Resolved

13	Code treats `UserWaitList` as a user when `allowFromContractOnly` is false	**Resolved**
14	Users' email addresses are leaked	**Unresolved**
15	Users can drain the contract reward balance by referring their own accounts and reusing their token balance	**Unresolved**
16	`suspendUser` and `unsuspendUser` update the wrong referrer	**Resolved**

Detailed Fix Review Results

TOB-ZEEB-1: Reentrancy in `claim` allows to claim extra rewards
Resolved commit `d76012b`. The `nonReentrant` modifier was added to the `claim` function, preventing this reentrancy.

TOB-ZEEB-2: Missing zero checks
Unresolved. A zero check was added to the constructor of the `SmartWalletWhitelist` contract. No other zero checks were added.

The client provided the following context for this finding's fix status:

> *Acknowledged.*

TOB-ZEEB-3: `VotingEscrow` administrator is not required to accept role
Resolved in commit `d76012b`. The `apply_transfer_ownership` function was renamed to `accept_transfer_ownership` and the implementation was updated to require the proposed owner to accept the proposal, instead of only being callable by the current owner.

TOB-ZEEB-4: No way to run `Waitlist-testnet` tests
Resolved commit `58da84b`. The necessary files to run the tests were added to the repository.

TOB-ZEEB-5: Multiple referral codes can point to the same user
Resolved in commit dd7bed9. A check was added to the `addUserWithReferralCode` function that reverts if the user already has a referral code. This prevents users from having multiple referral codes.

TOB-ZEEB-6: `SafeERC20` is not used
Resolved in commit dd7bed9. The `redeemRewards` function was renamed to `redeemUserRewards` and the token transfer was removed. The withdraw function now uses `safeTransfer` from SafeERC20.

TOB-ZEEB-7: `isValidAdminSigner` does not take amount as an argument
Resolved in commit dd7bed9. The token address and the amount were added to the `isValidAdminSigner` function.

TOB-ZEEB-8: `isValidWithdrawSigner` does not take a nonce as an argument
Resolved in commit dd7bed9. A nonce was added to the `isValidWithdrawSigner` function and nonce validation was added to the `withdraw` function. This prevents the same signature from being submitted multiple times.

TOB-ZEEB-9: `lastRedemptionTime` is never updated
Resolved in commit dd7bed9. The `lastRedemptionTime` is updated to the current `block.timestamp` at the end of the function. The function was renamed to `redeemUserRewards` and it now has an access control: it is only callable by an allowlisted address.

TOB-ZEEB-10: Being suspended has no downsides
Resolved in commit dd7bed9. A suspended user can no longer be a referrer, activate a referral, verify their email, or claim their rewards. This was achieved by adding the `require(!users[user].isSuspended)` check to the aforementioned actions.

TOB-ZEEB-11: Single withdraw signer can trigger a withdrawal due to incorrect check
Resolved in commit dd7bed9. The signer uniqueness check in the `isValidWithdrawSigner` function was modified to ensure that all three signers need to provide their signatures in order to trigger a withdrawal.

TOB-ZEEB-12: Referrals are never removed, which can lead to denial of service
Resolved in commit dd7bed9. The `getCountOfReferral` function was updated to avoid looping over the referrals, preventing denial of service due to an unbounded sized array. The `addChild` and `acceptChild` function were updated to increment and decrement the inactive and total referral count, respectively.

TOB-ZEEB-13: Code treats `UserWaitList` as a user when `allowFromContractOnly` is false
Resolved in commit dd7bed9. The `allowFromContractOnly` state variable and the codepath that handles calls from users that are not the `UserWaitList` contract have been removed. Since the function is only callable by allowlisted users, care should be taken that a normal user (or any other contract apart from the `UserWaitList`) is not added to the allowlist.

TOB-ZEEB-14: Users' email addresses are leaked
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged.*

TOB-ZEEB-15: Users can drain the contract reward balance by referring their own accounts and reusing their token balance
Unresolved. The issue has not been resolved.

The client provided the following context for this finding's fix status:

> *Acknowledged.*

TOB-ZEEB-16: `suspendUser` and `unsuspendUser` update the wrong referrer
Resolved in commit dd7bed9. The `suspendUser` and `unsuspendUser` functions now use the correct address when fetching the referrer for a user.

F. Fix Review Status Categories

The following table describes the statuses used to indicate whether an issue has been sufficiently addressed.

Fix Status	
Status	**Description**
Undetermined	The status of the issue was not determined during this engagement.
Unresolved	The issue persists and has not been resolved.
Partially Resolved	The issue persists but has been partially resolved.
Resolved	The issue has been sufficiently resolved.