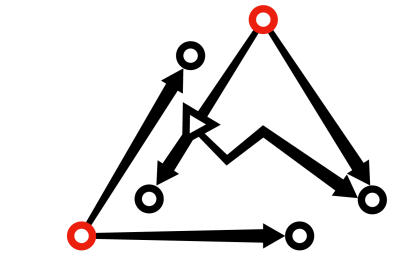

Least Authority

PRIVACY MATTERS

Security Audit Report

Zest Protocol

Final Audit Report: 16 March 2023

Table of Contents

- Overview
  - Background
  - Project Dates
  - Review Team
- Coverage
  - Target Code and Revision
  - Supporting Documentation
  - Areas of Concern
- Findings
  - General Comments
    - System Design
    - Code Quality
    - Documentation
    - Scope
  - Specific Issues & Suggestions
    - Issue A: Lack of Access Control in unwind Function
    - Issue B: late-fee Can Be Changed by Owner
  - Suggestions
    - Suggestion 1: Reconsider Incentive Schemes for Pool Delegates and Cover Providers
    - Suggestion 2: Increase and Improve Code Comments
    - Suggestion 3: Remove Unused and Commented-Out Code
    - Suggestion 4: Add Error Handling for a Possible Underflow Condition
    - Suggestion 5: Avoid Incorrect Use of end-block-height
    - Suggestion 6: Update and Improve Project Documentation
    - Suggestion 7: Make Liquidity Provision Compatible with Stacks Multisigs
- About Least Authority
- Our Methodology

Overview

Background

Trust Machines has requested that Least Authority perform a security audit of their Zest Protocol, a Clarity DeFi app.

Project Dates

- **September 14 - October 1:** Initial Code Review *(Completed)*
- **October 14:** Delivery of Initial Audit Report *(Completed)*
- **November 4:** Delivery of Updated Initial Audit Report *(Completed)*
- **January 3-12:** Verification Review *(Completed)*
- **January 24:** Delivery of Final Audit Report *(Completed)*
- **March 16:** Delivery of Updated Final Audit Report *(Completed)*

Review Team

- Alicia Blackett, Security Researcher and Engineer
- Mukesh Jaiswal, Security Researcher and Engineer
- Steven Jung, Security Researcher and Engineer
- ElHassan Wanas, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Zest Protocol followed by issue reporting, along with mitigation and remediation instructions as outlined in this report.

The following code repositories are considered in scope for the review:

- Zest Audit:
  https://github.com/Trust-Machines/Zest-Audit

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

`58a0a9ba366a8836fc2c456a925511e99e811777`

For the verification, we examined the Git revision:

`f4c3e22ec671db17722200814d69b025653aa354`

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

- Zest Audit:
  https://github.com/LeastAuthority/Zest-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:

- Zest Protocol (GitBook): https://zestprotocol.gitbook.io/zest
- Coinfabrik Audit Report (Before Revision): https://github.com/Trust-Machines/Zest-Audit/raw/temp-files/docs/2022-08%20Zest%20Audit%20(before%20revision).pdf
- Coinfabrik Commit (Before Revision): https://github.com/y0s0n/zestAudit/tree/c94cd6fdc98effcea10329fff99da2f0fb471479
- Coinfabrik Audit Report (After Revision) : https://github.com/Trust-Machines/Zest-Audit/raw/temp-files/docs/2022-08%20Zest%20Second%20Audit%20(revised).pdf
- Coinfabrik Commit (After Revision): https://github.com/Trust-Machines/Zest-Audit/tree/bc191d8246bb3d0b086bc68270ad5365d27c7292
- User Stories *(shared with Least Authority via email on 22 June 2022)*

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the smart contract;
- Attacks that impacts 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 the execution of the smart contract;
- Vulnerabilities in the smart contract code;
- Proper management of encryption and signing keys;
- Protection against malicious attacks and other ways to exploit the smart contract;
- Inappropriate permissions and excess authority; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

Zest is a lending protocol, which allows institutions to borrow on-chain loans in Bitcoin that are collateralized by off-chain assets. The protocol is advertised as lending against a balance sheet. Liquidity providers and stakers lend Bitcoin in return for rewards. In addition, liquidity providers get back the principal lent only if the loan is successfully paid back, or if there is enough cover in the cover pool to make up for any shortfalls. Pool delegates are responsible for vetting borrowers, creating cover pools and originating loans. Borrowers are given a grace period in which to make payments, and if this is breached, the off-chain liquidation process is started.

Our team performed a comprehensive review of the design and implementation of the Zest Protocol. We considered the overall design and architecture of the protocol, its components, and how users interact with it. We examined the roles and access rights within the system, and looked at the ways in which asset balances and states could be changed. We reviewed arithmetic edge cases, which could cause an overflow or underflow, and conducted a general review of the code quality. The Magic Protocol used by

Zest has also been audited by Least Authority, and this audit report should be read in conjunction with the Magic Protocol audit report. We considered remediation progress made from the findings of the previous audit performed by another third-party security team, in addition to assessing the economics behind incentive schemes, and comparing lending policies of Zest to other market actors.

We found that the design of the system exhibits excessive centralization and that the system is generally well implemented. We identified issues and suggestions to improve the overall security and quality of the implementation.

System Design

We performed a comprehensive review of the design of the Zest Protocol. Although we did not identify issues in the design, our teams identified areas of concern that could affect the security and privacy of borrowers and liquidity providers within the system.

Excessive Centralization

The Zest Protocol aims to achieve the lending of cryptocurrency collateralized by off-chain assets. In order to achieve this, a high degree of centralization was found to be necessary in the architecture of the system as well as a limited utilization of smart contracts automation.

The majority of the control is centered in the role of the pool delegate. The delegate is responsible for selecting the borrowers and performing liquidations, if necessary. In return, the delegate receives a payment related to growing the size of the liquidity pool and upon repayments made by the borrower. This may negatively incentivize the delegate to only focus on growing the pool. As a result, we recommend reconsidering the incentive scheme for pool delegates and cover pool providers (Suggestion 1).

One of the aims of decentralized finance is the reduction of fees by removing intermediaries. However, there are a number of fees being paid in the Zest Protocol system, including the investor fee, treasury fee, staking fee, cover fee, and liquidity provider fee. In some cases, fees can be changed after the deployment of the smart contracts. Late fees can be changed by the owner despite the documentation stating that they are static. Therefore, we recommend that fee modification result only from a decentralized autonomous organization (DAO) proposal (Issue B).

Within the Zest Protocol, there are a number of roles, such as the emergency team of emergency proposals, the admin, and governor, that have the ability to transfer or stop the transfer of assets. However, it is not clear how these roles are designated. We recommend that the project documentation describe in detail how these roles are assigned (Suggestion 6).

Off-Chain Risks

The Zest Protocol documentation claims to remove counterparty risk. However, if a borrower defaults, there is a chance that the liquidity provider will not get all their principal back because the cover pool is optional and may not cover all losses even if it is used. Therefore, there is a certain degree of counterparty risk that liquidity providers will be exposed to within the system.

Liquidation occurs if the borrower is unable to make a repayment within the grace period. The borrower enters into a Master Loan Agreement during onboarding, which is the only legally binding off-chain agreement that the pool delegate has recourse to. In on-chain liquidations, liquidators bid to be given the right to liquidate a pool. This ensures the best price is reached for the pool. However, the off-chain liquidation process proposed by Zest is to sell the debt over-the-counter (OTC), which is usually between two parties and does not ensure the best price is reached for the liquidity providers.

Another concern is that the liquidity provider has no knowledge of the liquidity level of the borrower's assets since the pool delegate is the only contact to the borrower. Additionally, as the borrower receives

Bitcoin, they are vulnerable to price volatility swings and could have to repay far more than borrowed with illiquid assets.

Code Quality

Overall, we consider the code to be well written but hurriedly put together for the purpose of the audit. We found that the implementation of the `unwind` function lacks an appropriate level of access control, which could result in an attacker preventing a loan from being drawn down. We recommend that access to the `unwind` function be secured (Issue A).

We also observed that there was a general lack of comments in the code, and some contracts contained commented-out variable definitions and functions, which interrupted the flow of the code. Sometimes, errors were made when copying functions to reuse them (Suggestion 2, Suggestion 3). We also identified an opportunity to improve error handling resulting from a possible underflow condition (Suggestion 4) as well as a couple of instances where the incorrect `end-block-height` is used, preventing the voting process from functioning as intended (Suggestions 5).

Tests

The repository of the smart contracts in scope did not contain any tests. However, in our discussions with the Zest Protocol team, we became aware that tests have been implemented in other branches of the protocol. Our team found that the tests implemented provide sufficient coverage of the codebase.

Documentation

The documentation providing a system level description of the implementation of the protocol was being updated while the audit was in progress, which hindered our review. We recommend that the project documentation be updated to reflect the system as implemented (Suggestion 6).

Code Comments

We recommend that code comments be improved to describe the intended behavior of security-critical functions and components (Suggestion 2).

Scope

The Zest Protocol team has integrated the Magic Protocol into their front-end to enable the generation of hash timelock contract (HTLC) addresses. We recommend that a third-party security team perform a security review of how sensitive data, such as the pre-image in the front-end, is handled.

Dependencies

Zest uses the Magic Protocol and the Alex Protocol when liquidations are needed. Both have been audited by Least Authority, and our team has identified points in those implementations, which require remediation at the time of writing of this report.

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: Lack of Access Control in unwind Function	Resolved

Issue B: late-fee Can Be Changed by Owner	Resolved
Suggestion 1: Reconsider Incentive Schemes for Pool Delegates and Cover Providers	Resolved
Suggestion 2: Increase and Improve Code Comments	Resolved
Suggestion 3: Remove Unused and Commented-Out Code	Resolved
Suggestion 4: Add Error Handling for a Possible Underflow Condition	Resolved
Suggestion 5: Avoid Incorrect Use of end-block-height	Resolved
Suggestion 6: Update and Improve Project Documentation	Resolved
Suggestion 7: Make Liquidity Provision Compatible with Stacks Multisigs	Resolved

Issue A: Lack of Access Control in unwind Function

Location

`contracts/pool/pool-v1-0.clar#L633`

Synopsis

The `unwind` function can be called by anyone after a loan has been funded. Due to the lack of appropriate access control to the `unwind` function, any funded loan can be set to the expired status and stop the drawdown process, eventually stopping the whole loan process.

Impact

Without appropriate access control on the `unwind` function, anyone has the ability to cancel the loan.

Preconditions

This attack requires only that the attacker know whether the loan is funded or not. Once the loan is funded, the attacker can call `unwind`.

Feasibility

Straightforward.

Mitigation

We recommend limiting access control of the `unwind` function.

Status

The Zest Protocol team implemented appropriate access control for the `unwind` function.

Verification

Resolved.

Issue B: late-fee Can Be Changed by Owner

Location

`contracts/loan/payment-fixed.clar#L44`

Synopsis

Late fees, which should be static according to the documentation, can be changed by the owner.

Impact

Late fees could be changed without the knowledge of the borrowers.

Mitigation

We recommend that the `set-late-fee` functionality be removed in order for the late fee to be static, or that the documentation be updated to accurately reflect the implementation.

Remediation

We recommend implementing changes in the parameters (e.g., `late-fee`, `pool-cover-fee`, `pool-delegate-fee`) only through the DAO.

Status

The Zest Protocol team improved the security of the process for setting and changing the late fee.

Verification

Resolved.

Suggestions

Suggestion 1: Reconsider Incentive Schemes for Pool Delegates and Cover Providers

Location

`https://zestprotocol.gitbook.io/zest/the-protocol/pool-delegates/what-do-pool-delegates-earn`

Synopsis

Pool delegates earn rewards when the size of the pool increases and when borrowers make repayments. As liquidity pool providers can send funds before the creation of the loan but not receive liquidity provider rewards, there is an incentive for the pool delegate to keep increasing the size of the pool, without lending, to still receive pool delegate rewards. Users who contribute to the cover pool are able to send funds before the pool has started, which results in them not earning rewards.

Mitigation

We recommend allowing the liquidity provider to deposit funds only after the loan has been agreed and the required collateral is paid. The cover pool providers should also only deposit funds when they are eligible for rewards.

Status

The Zest protocol team stated that since pool delegates will only earn rewards when borrowers make repayments, the incentive mechanism is in alignment with the objectives of the system. As a result of this clarification, our team has determined this suggestion is resolved.

Verification

Resolved.

Suggestion 2: Increase and Improve Code Comments

Location

Examples (non-exhaustive):

`contracts/pool/staking-pool.clar`

`contracts/loan/coll-vault.clar`

`contracts/loan/coll-vault.clar#L55`

`contracts/loan/coll-vault.clar#L78`

Synopsis

The level and degree of comments within the code varies greatly. Some contracts do not contain any comments, while others contain only a few. In certain instances, copying and pasting comments and functions has led to misleading descriptions of functions.

Mitigation

We recommend adding relevant comments to improve the readability of the code.

Status

The Zest protocol team improved the code comments.

Verification

Resolved.

Suggestion 3: Remove Unused and Commented-Out Code

Location

Examples (non-exhaustive):

`contracts/token/zest-reward-dist.clar`

`contracts/rewards-calc.clar`

Synopsis

A number of contracts contain previous settings of variables or implementations of functions, which are no longer used.

Mitigation

We recommend removing unused code to enhance the readability of the smart contracts.

Status

The Zest Protocol team removed unused and commented-out code from the codebase.

Verification

Resolved.

Suggestion 4: Add Error Handling for a Possible Underflow Condition

Location

contracts/loan/coll-vault.clar#L85

Synopsis

The function to remove collateral from a loan that has already been initiated has been copied from the function to add collateral to a loan that has already been initiated. As a result, it is possible to create an integer underflow panic if the amount is greater than collateral.

Mitigation

We recommend implementing more graceful error handling for this case.

Status

The Zest Protocol team modified the code to prevent the overflow condition. As of the verification phase of the project, if the entered amount is greater than the collateral amount, the entire collateral is transferred. Otherwise, the `(collateral_amount - amount)` is transferred.

Verification

Resolved.

Suggestion 5: Avoid Incorrect Use of end-block-height

Location

dao/extensions/zge001-proposal-voting.clar#L107

dao/extensions/zge001-proposal-voting.clar#L131

Synopsis

The incorrect operator is used in the `vote` and `conclude` functions to check the duration of the voting period.

In the `Vote` function, the condition for the `end-block-height` is as follows:

```
(asserts! (< block-height (get end-block-height proposal-data))
err-proposal-inactive)
```

If the above conditions are allowed, then the voter trying to vote on the `end-block-height` will not be able to vote.

In the `conclude` function, the condition for the `end-block-height` is as follows:

```
(asserts! (>= block-height (get end-block-height proposal-data))
err-end-block-height-not-reached)
```

As a result, the votes are concluded on the `end-block-height` and not after the voting period is over.

Impact

Due to usage of the incorrect operator in checking the condition for the `end-block-height`, the voter will not be able to vote on the `end-block-height`, and the result can be concluded on the `end-block-height`.

Mitigation

In the `vote` function, we recommend that the condition be corrected to:

```
(asserts! (<= block-height (get end-block-height proposal-data))
err-proposal-inactive)
```

This condition will allow the voter to vote on the `end-block-height`.

In the `conclude` function, we recommend that the condition be corrected to:

```
(asserts! (> block-height (get end-block-height proposal-data))
err-end-block-height-not-reached)
```

This condition will allow the result to be concluded after the `end-block-height` period has passed.

Status

The Zest Protocol team updated the implementation of the usage of `end-block-height`, as suggested.

Verification

Resolved.

Suggestion 6: Update and Improve Project Documentation

Synopsis

The project documentation was being updated during the code review, which inhibited an efficient understanding of the critical components of the implementation.

Mitigation

We recommend updating and improving the documentation to include:

- Design specifications that provide detailed and concise information about the system design and requirements. A design specification allows security auditors to check whether the code has been implemented correctly, adheres to the specification, and avoids incorrect assumptions about the expected behavior of the system, which may lead to missed security vulnerabilities;
- Architectural diagrams describing components of the system and their interaction and;
- A more detailed description of the process for onboarding to critical roles within the system.

Status

The project documentation has been updated as suggested.

Verification

Resolved.

Suggestion 7: Make Liquidity Provision Compatible with Stacks Multisigs

Synopsis

In the current implementation, the protocol is only compatible with external addresses. Multi-Sig compatibility allows users to interact with the protocol more securely.

Mitigation

We recommend implementing compatibility for liquidity providers controlling their assets with Multi-sigs.

Status

Multi-sig compatibility for liquidity providers has been implemented as suggested.

Verification

Resolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in multiple Languages, such as C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, JavaScript, ZoKrates, and circom, for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture in cryptocurrency, blockchains, payments, smart contracts, zero-knowledge protocols, and consensus protocols. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. We are an international team that believes we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. As we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present and possibly resulting in Issue entries, then for each, we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative and transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even before having verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate and comprehensive mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our Initial Audit Report, and before we perform a verification review.

Before our report, including any details about our findings and the solutions are shared, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for a resolution that balances the impact on the users and the needs of your project team.

Resolutions & Publishing

Once the findings are comprehensively addressed, we complete a verification review to assess that the issues and suggestions are sufficiently addressed. When this analysis is completed, we update the report and provide a Final Audit Report that can be published in whole. If there are critical unaddressed issues, we suggest the report not be published and the users and other stakeholders be alerted of the impact. We encourage that all findings be dealt with and the Final Audit Report be shared publicly for the transparency of efforts and the advancement of security learnings within the industry.