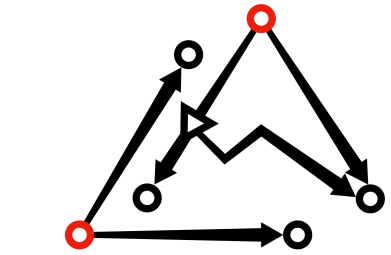

Least Authority

PRIVACY MATTERS

Tinlake 0.3.0

Security Audit Report

Centrifuge

Final Report Version: 16 October 2020

Table of Contents

Overview

Background

Centrifuge has requested that Least Authority perform a security audit of Tinlake 0.3.0, a smart contracts framework on Ethereum that enables borrowers to draw loans against non-fungible assets. Any assets represented on-chain as Non-Fungible Tokens (NFTs) are financed by issuing an ERC-20 token against all of the collateral NFTs that are deposited into the Tinlake contracts.

Tinlake 0.3.0 enables revolving investment pools, allowing investors to redeem and invest TIN and DROP tokens on an ongoing basis. The investment and redemption of tokens is executed in periodical epochs for which investors can place orders. The calculation of the token prices for an investment or redemption requires a Net Asset Value (NAV) evaluation of the ongoing loans based on the underlying collaterals.

The following components are considered in-scope for the review:

- Tinlake NAV Feed
    - NFT Feed Contract
- Revolving Pool Contracts
    - Tranche Contract
    - Epoch Coordinator Contract
    - Reserve Contract

Project Dates

- **September 8 - September 24**: Initial Review *(Completed)*
- **September 26:** Initial Audit Report delivered *(Completed)*
- **October 13 - 15:** Verification Review *(Completed)*
- **October 16:** Final Audit Report delivered *(Completed)*

Review Team

- Phoebe Jenkins, Security Researcher and Engineer
- Nathan Ginnever, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of Tinlake 0.3.0 followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

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

- Tinlake NAV Feed:
    - NFT Feed Contract: https://github.com/centrifuge/tinlake/tree/develop/src/borrower/feed
    - NAV Model: https://docs.google.com/spreadsheets/d/1O124ru0MsdKLsOjRRUqlb4zAoIC5RNgNfQpxbAv1wNw/edit#gid=1182979041
- Revolving Pool Contracts (Tranche Contract, Epoch Coordinator Contract, and Reserve Contract): https://github.com/centrifuge/tinlake/tree/develop/src/lender

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

`a2fc3dadf68151e2fa97fdeee3cb7c3d70268474`

For the verification, we examined the Git revision:

`95fc51ca085eb5190e0ab3b00fb4838524d26ea3`

All file references in this document use Unix-style paths relative to the project's root directory.

Supporting Documentation

The following documentation was available to the review team:

- Tinlake 0.3.0 Audit Scope: https://centrifuge.hackmd.io/GNDFFYZUQ0O95_QrH6Uq5A
- Documentation to be provided prior to the audit: https://centrifuge.hackmd.io/GNDFFYZUQ0O95_QrH6Uq5A#What-we-will-provide
- Tinlake 0.2.0 Documentation: https://tinlake.centrifuge.io/
- Tinlake 0.2.0 (Previous Audit Scope): https://docs.google.com/presentation/d/1XyWHoNVJEhSYDWJkKypg63rWloDgZ2tcH8epHSoE7JQ/edit#slide=id.p
- Tinlake 0.3.0 Audit Kickoff Desk: https://docs.google.com/presentation/d/1X8U0ya3XMiSHkpmpsKIVkVsrWyXzMQZkKwHqitcysYg/edit#slide=id.g9582e914c6_0_63
- Blog Posts (NAV)
    - Part I: https://medium.com/centrifuge/tinlake-pricing-and-valuation-series-part-1-how-to-price-real-world-assets-cf6655132bef
    - Part II: https://medium.com/centrifuge/tinlake-pricing-and-valuation-series-part-2-valuing-an-asset-portfolio-247d8f2f0d5

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation;
- Adversarial actions and other attacks on the network;
- 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 included and functional;
- DoS/security exploits that would impact the contracts intended use or disrupt the execution of the contract;
- Vulnerabilities in the smart contracts code;
- Protection against malicious attacks and other ways to exploit 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

Our team's previous audit of the Tinlake system included the core contracts and proxy actions of the Tinlake 0.2.0 framework. Tinlake 0.3.0 introduces several new contracts, including the NAV (Net Asset Value) Feed contract, the Epoch Coordinator contract, the Reserve contract, and the refactored Tranche contract. These new additions and changes are the core components in-scope for this review.

Code Quality + Documentation

Our team found the code to be well organized, which is consistently applied throughout the codebase. Although the Centrifuge Tinlake DeFi contracts are derived from the MakerDAO contracts, which have been reviewed for security and withstanding active attacks, the Tinlake contracts are heavily modified from the original MakerDAO system and should still be carefully reviewed as a result.

The code base includes considerable comments, with very few locations lacking sufficient detail or requiring additional information. The Centrifuge team provided thorough and extensive documentation, which facilitated a comprehensive understanding of the changes made in Tinlake v0.3.0. A small improvement to the documentation would be to provide further definition around financial terminology, allowing an easier understanding of the concepts for reviewers who are unfamiliar with the financial industry.

We commend the Centrifuge team for consistently adhering to development best practices, which include good code quality, clear and detailed comments, adequate test coverage that include fail cases, and thorough documentation. It is clear that the Centrifuge team has strongly considered security throughout the implementation. Our team did not identify any apparent structural flaws and have checked the contracts for common pitfalls.

System Design

The Tinlake framework is made up of a system of customizable modular contracts on Ethereum meant to support independent configurations for ERC-20 and ERC-721 minting, along with other financial functions, including allowing borrowers to draw loans against non-fungible assets. As noted in our previous report,

> *"The Tinlake collateralized debt system makes use of real world assets represented as NFTs with variable loan value, length, repayment schedules and interest rates. Given the complexity due to the broad spectrum of possible use cases, it is difficult to understand all feasible scenarios involved in the modular contracts provided and how they will operate under situations such as oracle failures, repayment failures, asset seizure and loan reimbursement."*

The most recent upgrade, Tinlake 0.3.0, introduces a new revolving pool feature that allows for a continuous lending and reward protocol. While this improves the usability of Tinlake, it also increases the complexity of states that the contracts may find themselves in. The introduction of the NAV Feed contract combines all outstanding financing, where finances were originally localized to specific pools.

Furthermore, Tinlake 0.3.0 introduces a scoring system. If enough preconditions are satisfied then an epoch may be closed with minimal computational complexity. However, if the orders in the epoch reach certain thresholds then the scoring system must be computed off-chain to preserve costs. In addition, a challenge mechanism is implemented to ensure the optimal solution to this pool balancing. Tinlake 0.3.0 introduces contract global variables to track this balancing across pools done at each epoch documented as pool variables.

While the documentation provides a description of these variables and the way they are calculated ensuring that they are correct for every state update possible in the Tinlake contracts, it is a complex and lengthy undertaking. We suggest further testing of the already implemented scenario tests, where the ranges of possible inputs for these variables are tested and observed for unexpected permutations.

Continual Testing

Test coverage is considerable throughout the code base. The Centrifuge team has created tests that cover edge fail cases, such as closing an epoch too early or after an epoch has been open for an extended period of time. We recommend further expanding tests so that they cover additional possible fail cases. At present, `ModelInputs` models are provided in tests as inputs to function test targets, such as closing epochs or solution submission. These simulate scoring parameters and are a good start to providing structured inputs where random inputs can be supplied for unguided coverage of states. There may gradually be more structure aware mutation models added that ensure the inputs fit within expected parameters of the Tinlake framework and test relevant paths of execution. Targeting the execution of the epoch could provide insight into edge cases as epoch execution is particularly complex. Furthermore, we suggest exploring other forms of testing such as simulating the interactions of all state transitions, fuzzing as many targets with structured inputs as possible, and defining invariants of the Tinlake system (Suggestion 7).

As suggested, further testing is recommended for the already implemented scenario tests, covering the ranges of possible inputs for these variables. All inflows and outflows as listed in the documentation should be tested to account for these possibilities. This process would help in identifying logic vulnerabilities that may be more difficult to detect. Furthermore, reducing complexity in state wherever possible and documenting the authentication dependency graph, will help improve security of this system.

Specific Issues

We list the issues and suggestions found in the code, 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
Suggestion 1: Remove Unnecessary Safe Arithmetic	Resolved
Suggestion 2: Improve Linting	Unresolved
Suggestion 3: Improve Documentation	Unresolved
Suggestion 4: Use Modern Solidity Compiler with Non-Experimental ABI Encoder	Unresolved
Suggestion 5: Improve Math Library's Use of Fixed Point Numbers	Unresolved
Suggestion 6: Internal Modifier Broken	Unresolved
Suggestion 7: Add Invariant Testing for Complex State Transitions	Unresolved

Suggestions

Suggestion 1: Remove Unnecessary Safe Arithmetic

Location

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/coordinator.sol#L172

Synopsis

The example linked in the location shows a case where the additional use of overflow protection is not necessary. If the max integer size could be reached on the epoch counter, the contract would freeze as the safe addition checks would always revert.

Mitigation

The epoch count is not intended to ever reach the max size of `uint256` and should be removed. Further inspection of all safe arithmetics should review whether or not it is necessary to provide any over or under flow protections.

Status

The Centrifuge team issued a commit which removed the call to `safe_add` in the `closeEpoch` function.

Verification

Resolved.

Suggestion 2: Improve Linting

Synopsis

Solhint does not report any linting errors when run against the code base, however, it is apparent through manual code review that there is a considerable amount of inconsistencies in code format, such as line spacing, indentation, and comment format.

Mitigation

Usage of Solidity linting and formatting tools help to automatically keep code style consistent, particularly if they are added into a continuous integration process. Solhint, a standard tool that is not configured to detect certain inconsistencies, does not capture many style and formatting errors. As a result, we recommend using Ethlint, which does have the capability to automatically correct many minor style errors in the codebase, thus reducing the likelihood for missed vulnerabilities.

Status

The Centrifuge team has acknowledged this suggestion and responded that they intend to implement the suggested mitigation in the future.

Verification

Unresolved.

Suggestion 3: Improve Documentation

Location

Tinlake Revolving Pool (v3) Documentation: https://centrifuge.hackmd.io/OMYT-Gh6Tm-D91CynWX0fA

Synopsis

There are some inconsistencies with the documentation of the protocol and the coded implementation. For example, the provided documentation defines a rolling submission process that will continue to reset the thirty minute timer upon new submissions while the code only allows for one submission window of thirty minutes to be initiated. As quoted from the documentation, "*If a competing viable solution is submitted resulting in a higher "max function" a new 30min challenging period starts.*"

In addition, there are several locations in the code comments where typos are present, thus making the comments difficult to read.

Mitigation

Address inconsistencies in the project documentation and correct typos in the code comments.

Status

The Centrifuge team has acknowledged this suggestion and responded that they intend to implement a solution in the future.

Verification

Unresolved.

Suggestion 4: Use Modern Solidity Compiler with Non-Experimental ABI Encoder

Location

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/coordinator.sol#L15-L16

Synopsis

The current compiler version used is generally considered to be out of date. A list of bug fixes that are not present in earlier versions of the compiler is available on the `solc` release pages. As of solc v0.6.0, the experimental ABI encoder referenced in the location above is no longer considered experimental and is safe to use. As a result, it is recommended to update the compiler in order to incorporate important changes.

Mitigation

Update the compiler version to be above v0.6.0. We suggest using compiler v0.6.2 or v0.6.10.

Status

The Centrifuge team has acknowledged this suggestion and responded that they intend to implement the suggested mitigation in the future.

Verification

Unresolved.

Suggestion 5: Improve Math Library's Use of Fixed Point Numbers

Location

https://github.com/centrifuge/tinlake-math/blob/098860a680866e9e85912dcace0efe9ba3c8eb45/src/math.sol

Synopsis

The `Fixed27` struct helps in checking proper use of math functions that simulate fixed point calculations. However, some of the functions in the tinlake-math library would benefit from ensuring that they are used properly, especially confirming that their arguments are proper fixed-point representations.

Mitigation

Functions such as `rmul`, `rdiv` and `rdivup` that are intended to operate upon and return fixed-point values should utilize the Fixed27 type for appropriate arguments and return Fixed27 values. This will help prevent accidentally calling the function with unexpected numerical values, which could cause errors that are several orders of magnitude off.

Additionally, implementing equivalent rounding functions rather than directly using Solidity's built-in division operator, when possible, is advised. Since Solidity's integer division operator always rounds down, the order of operations can cause subtle issues with rounding, as well as propagating numerical errors that may be difficult to detect.

Status

The Centrifuge team has responded that they have been investigating proper handling of fixed-point numbers in their math library and that they intend to address these concerns in an upcoming release.

Verification

Unresolved.

Suggestion 6: Internal Modifier Broken

Location

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/reserve.sol#L73-L81

Synopsis

There are functions in the codebase that have a `public` modifier entry point with a single line of code that jumps execution to an `internal` modified function. This is equivalent to simply implementing only a single public function.

Mitigation

Remove the unnecessary `internal` modified functions as they are not kept internal for any apparent reason. Simply call the `public` modified function internally if needed.

Status

The Centrifuge team has responded that they do not intend on implementing the suggested mitigation and they want to maintain the public interfaces as they currently exist. This decision does not pose an identified security risk.

Verification

Unresolved.

Suggestion 7: Add Invariant Testing for Complex State Transitions

Synopsis

The structure of the Tinlake system consists of many different components, each with its own internal state, and interacting with other components in complex ways. For example, flag variables such as `minimumEpochTimePassed` and `submissionPeriod` in the Epoch Coordinator contract, and `waitingForUpdate` in the Tranche contract, indicate that there is a complicated set of state transitions that occur during normal system operation, making it challenging to enumerate all possible configurations. A tool automatically stress testing these configurations throughout the system would help ensure that the system cannot be put into an invalid state.

Mitigation

Echidna is a grammar based fuzzing tool that checks for invariants. For example, in the Epoch Coordinator contract, ensuring that `gotFullValidSolution` should only be true when `submissionPeriod` is also true. With these set parameters, it then exhaustively fuzzes the contracts, ensuring that this condition can never be violated during contract operation. We recommend writing a number of such invariants and incorporating Echidna into a testing or continuous integration process to ensure that contract interactions remain sound.

Additionally, sFuzz is an adaptive or guided fuzzing technique using the AFL (American Fuzzy Lop) strategy. This is similar to Echidna in that it will spin up a mock blockchain network and attempt to randomly call the entry points and examine the output states against an oracle of known vulnerabilities or crash detect if an EVM exception is found. AFL coverage-guided approaches are not restricted to a single input type and do not require grammar definitions. Mutation based fuzzers are easier to set up, however, they may not reach the same coverage as a grammar based method. With some configuration, AFL can be made grammar aware.

Status

The Centrifuge team has acknowledged this suggestion and responded that they intend to implement the suggested mitigation in the future.

Verification

Unresolved.

Recommendations

We recommend that the *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We also recommend that the Centrifuge team expand their testing to cover fail cases in particular. Simulation testing running structured and random scenarios to help uncover unknown states that have yet to be discovered will assist in identifying potential vulnerabilities.

Furthermore, reducing complexity in state wherever possible will help improve security of this system. Additional documentation defining the financial industry terminology will also facilitate easier

understanding of the complex concepts and terms within the Tinlake system, particularly for those unfamiliar with the nuances of the financial industry.

We commend the Centrifuge team for their adherence to development best practices and their continued commitment and strong consideration for security.

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.