Least Authority
PRIVACY MATTERS

Tinlake Contracts + Actions

Security Audit Report

Centrifuge

Final Report Version: 7 April 2020

Table of Contents

Overview

Background

Centrifuge has requested that Least Authority perform a security audit of their Tinlake Platform, 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.

Project Dates

- **January 27 - February 7:** Initial Review *(Completed)*
- **February 11:** Initial Audit Report delivered *(Completed)*
- **March 5:** Updated Audit Report delivered *(Completed)*
- **April 1 - 3:** Verification Review *(Completed)*
- **April 7:** Final Audit Report delivered *(Completed)*

Review Team

- Nathan Ginnever, Security Researcher and Engineer
- Emery Rose Hall, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Tinlake Platform 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:

- Core Contracts
  - Tinlake Core Contracts: https://github.com/LeastAuthority/tinlake
  - Tinlake Math: https://github.com/LeastAuthority/tinlake-math
  - ERC721 - NFT: https://github.com/LeastAuthority/tinlake-title
  - Value Registry: https://github.com/LeastAuthority/tinlake-registry
  - Tinlake Auth: https://github.com/LeastAuthority/tinlake-auth
- Proxy Actions
  - Tinlake Actions: https://github.com/LeastAuthority/tinlake-actions
  - Tinlake Proxy: https://github.com/LeastAuthority/tinlake-proxy

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

```
tinlake@05bfcd81bb71c9f3c8dc8161089ae5a673e0619a

tinlake-math@23ba2f6a1b0d23ebc8103508807fb0f709a574a8

tinlake-title@c02afb3a571be66cfa151b2234344374d735eb92

tinlake-registry@e022e3d756c4d7fa23becaed4c7fb543fc46171b

tinlake-auth@dc1bc196b5c1d24a543f613cab446ce62498420e

tinlake-actions@99219f891d9976ed3e0967d7f51a067ad2f5b6bb
```

`tinlake-proxy@ab51c770a10ce15f337b79cd9957f2d027e44649`

For the verification, we examined the Git revision:

`tinlake@d4a3b98b86e17e8f54c7915b1d9b84db506ba816`

`tinlake-math@0b4d99fec6e2c37a5cc5d8c1aa20fd95b4cbc93c`

`tinlake-title@827ac8736ccca9b2229e49ec6aec3da0ea6f5da4`

`tinlake-registry@1e05ed629d5804136e5d29c7b1a777865e22b5bd`

`tinlake-auth@0d6c11a1a6cb8505d1c6e9be503240dc0f2a06ab`

`tinlake-actions@3dfcb815b4f982454a0500c12a8f4f2288e57834`

`tinlake-proxy@aea3948e19f67fab3728df955cb7b4c691f676f95`

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:

- Centrifuge Developer documents: https://developer.centrifuge.io/tinlake/overview/introduction/
- Tinlake Naxos Audit slides: https://docs.google.com/presentation/d/1XyWHoNVJEhSYDWJkKypg63rWloDgZ2tcH8epHSoE7JQ/edit#slide=id.g7582d8baba_0_98
- Tinlake Audit Kickoff slides: https://docs.google.com/presentation/d/1chxnoHBJtRbK3dVFPo3vEm9WyQoJeVsm2N6oeehMbmA/edit#slide=id.g6e26b0590b_0_1
- Tinlake Documentation HackMD: https://centrifuge.hackmd.io/hkpb7qqYTNiFSADz0rOGXQ?both

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

Tinlake is a complex system of customizable modular contracts meant to support independent configurations for ERC-20 / ERC-721 minting and other financial functions (i.e. collateral governance or portfolio risk management). It is apparent that careful thought was placed into the modules of this system and that a good design pattern of customizability and upgradeability was applied throughout the codebase. Generally, the code quality is very good and only has minor duplication of code oversights.

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.

Because of this, it is important that both developers and auditors are able to comprehend these scenarios in order to prevent the introduction of security vulnerabilities during events like failed loan repayments that might take place. Our audit only found limited apparent vulnerabilities in the contract code, however, it is suggested that further review be conducted to examine all possibilities of how the Tinlake contracts will be used.

During the course of the audit, the Centrifuge team expanded the Tinlake system documentation by including additional flow charts and explanations for how the overall components work together. Our team found this aided in explaining the use of collateralized debt financial instruments on the blockchain. Furthermore, the inclusion of additional scenarios outlined in Drop & Tin: An Intro to Tranches were valuable in understanding how the modules work together over time. In addition to this list of possible scenarios, we suggest continuing to expand on this documentation over time such that it adequately covers other scenarios including different time frames for repayment, extreme cases where all loans default or all assets are priced incorrectly, and the inability of an asset to identify a market in which it can be sold in the event that the loan defaults.

A second step would be to build upon such a list with an adversarial analysis through threat modeling. Although this is helpful activity for any system, it could be especially so for a complex system like Tinlake where there are many different ways for the contracts to be used and documenting the scenarios can help onboard reviewers and capture specific areas of risk.

Finally, given that real world events are difficult to model statistically in a comprehensive manner, it is strongly suggested to continue conducting audits over periodically to identify new potential vulnerabilities.

Specific Issues

We list the issues we 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
Issue A: ERC20 Implementation’s `Approve()` Susceptible to Front-Running	Unresolved
Issue B: Incorrect Equality Checks	Resolved

Issue C: Outdated Compiler Versions	Resolved
Issue D: Ambiguous NULL Ownership	Unresolved
Issue E: Shelf Issue Function May Be Re-entered	Resolved
Issue F: Division By Zero Is Unchecked In Safe Math	Resolved
Suggestion 1: Author Test Suite for Registry Contract	Resolved
Suggestion 2: Improve Documentation	Resolved
Suggestion 3: Improve Consistency	Resolved
Suggestion 4: Remove Redundant Code	Resolved

Issue A: ERC20 Implementation's `Approve()` Susceptible to Front-Running

Location
https://github.com/centrifuge/tinlake-erc20/blob/master/src/erc20.sol#L85

Synopsis
There are a few places in the Tinlake system that rely on the transfer of approved tokens. It may be possible that some of these transfers could be done in such a way that the approver is left with an unexpected balance. This depends on the ability or requirement of the approver to update their balance. If this case is present then this issue recorded within [REC19] may be of relevance:

> *An ERC20 security issue, known as the "multiple withdrawal attack", was raised on GitHub and has been open since November 2016. The issue concerns ERC20's defined method `approve()` which was envisioned as a way for token holders to give permission for other users and dapps to withdraw a capped number of tokens. The security issue arises when a token holder wants to adjust the amount of approved tokens from `N` to `M` (this could be an increase or decrease). If malicious, a user or dapp who is approved for `N` tokens can front-run the adjustment transaction to first withdraw `N` tokens, then allow the approval to be confirmed, and withdraw an additional `M` tokens.*

Impact
High. Possible loss of tokens.

Preconditions
There must be an approval adjustment that is vulnerable to this attack.

Feasibility
Approval adjustments that could be vulnerable must be able to be acted upon by the approved party and must be able to be spent quickly enough to make this feasible. An example of this may be in the lender contract that mints tokens based on an approved currency amount. However, we are unsure if balance adjustments to the approval can be made to incentivise the lender into acting on minting or burning in favor of receiving or redeeming more currency than was expected.

Technical Details
Full details of this attack, along with a number of approaches to mitigation, is contained in [REC19].

Mitigation
Enforce approval balance is set to 0 before adjusting an approval or CAS approved to eliminate the possibility that the approved amount of a token is ever less or more than expected. See the paper linked in the technical details for more in-depth mitigation details.

In addition, create detailed documentation of every token transfer that uses this method to ensure that there are no cases where a party may conceal the intended approval amount of a token.

Status
The Centrifuge team has acknowledged the existence of the multiple withdrawal attack within the `approve()` function and have decided not to deviate from the MakerDAO DAI stablecoin ERC 20 implementation that also includes this attack vector.

We strongly encourage the Centrifuge team to examine every possible way that a token's approved amount to a spender, whether user or contract, might need to be adjusted and whether or not it could be possible for the multiple withdrawal attack to cause any amount of tokens to be spent unexpectedly. If understanding every possible scenario that a token's approval may need to be adjusted is not possible, then we encourage the Centrifuge team to implement the suggested safety method to prevent unknown or unforeseen situations.

Verification
Unresolved.

Issue B: Incorrect Equality Checks

Location
https://github.com/LeastAuthority/tinlake/blob/89adc6386e5e8bae73f88d6e2a92bb70f6c8f7df/src/borrower/pile.sol#L66

https://github.com/LeastAuthority/tinlake/blob/89adc6386e5e8bae73f88d6e2a92bb70f6c8f7df/src/borrower/pile.sol#L77

Synopsis
The condition now `<= lastUpdated` can never be less than now. This requirement is placed on `decDebt()` and `debt()`, and given that `lastUpdated` is not initialized and defaults to 0, these functions can not be called until after `file()` sets it to now. Given that any subsequent call to `incDebt()` or `debt()` will always be after now unless within the same block, these functions may only be called in the same block that `file()` is called. There is no documentation on the intended behavior of these checks.

The condition now `>= lastUpdate` seems to be an unnecessary check as `lastUpdate` can only be set to now, where now can either be within the current block and equal to `lastUpdated`, or from a future block and always greater than `lastUpdated`.

Impact
The use of the less than operator will never be used. Without documentation as to why it is used, this will lead to confusion in understanding this functionality.

Remediation

The intended equality check might be now `== lastUpdated` for `debt()` and `incDebt()`, to ensure that a function is called within the same block. Document the purpose of this check.

Status

A commit that changes the equality check to the suggested remediation of `== rather than <=` has been added.

Verification

Resolved.

Issue C: Outdated Compiler Versions

Synopsis

A number of contracts are using an outdated pragma. Some are as low as 0.4.23, such as the dapphub/ds-note contract. In most other cases, the pragma is set to 0.5.3, which is still over a year behind the current release.

Impact

The outdated compiler version can subject contracts to security issues fixed in newer compiler versions.

Remediation

Update all contracts to use the latest version 0.5.16 or 0.6.2 (at the time of this report).

Status

Commits have been accepted throughout the codebases, including the DS-Note dependency, which enforce any build to use solidity >=0.5.15.

Verification

Resolved.

Issue D: Ambiguous NULL Ownership

Location

https://github.com/LeastAuthority/tinlake-auth/blob/master/src/auth.sol#L23

Synopsis

The ability to drain the ownership of a contract to 0 owners is an ambiguous way to disable functionality. Auth is used similar to a 1 of N multisig, where wards may be added or removed and only one ward is required to execute a function modified with Auth. It could become possible for the removal of all wards from the state by calling `deny()`. This would leave the contracts that depend on this authentication in an unusable state.

Impact

All contracts that rely on the authentication could become null and unusable.

Preconditions
The last owner or ward of the authenticated contract must deny themselves access by accident in order for a contract that relies heavily on authentication to lose control, where or when it was not intended to lose all ownership.

Remediation
Add a minimal amount of state that tracks the amount of wards that own a contract and ensure that it is intentional if the amount of wards becomes 0. Add a function that explicitly declares the intended functionality of removing all wards is to remove all functionality from a ward controlled contract.

Status
The Centrifuge team has stated that they are aware of the possibility that total ownership of a contract or functionality could be inadvertently removed. However, they note that this is an unlikely edge case, as an owner must remove all other owners and then lastly remove themselves. As a result, the Centrifuge team does not intend to implement the suggested remediation as they consider the ability to remove all owners from the contract with the `deny()` function an intended feature. We encourage the Centrifuge team to be explicit about when and where it is intended to remove all owners to prevent any accidental cases, which could potentially cause irreparable damage.

Verification
Unresolved.

Issue E: Shelf Issue Function May Be Re-entered

Location
https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/shelf.sol#L112

Synopsis
`Issue()` in `Shelf.sol` calls the `ownerOf()` function of a supplied NFT. If a custom NFT is provided to this function, the `ownerOf()` function may call back to `issue()` and update the state of the shelf contract in unexpected ways.

Impact
There may be a case that an NFT is vetted and contains a custom malicious `ownerOf()` that will register many NFT loans to the shelf contract. There may be other unintended state manipulations from `ownerOf()`.

Remediation
Modify `ownerOf()` with `view` so that it is unable to make state updates.

Status
A commit that modifies `ownerOf()` to be a view function and can no longer alter state upon re-entry has been added to the NFT contract.

Verification
Resolved.

Issue F: Division By Zero Is Unchecked In Safe Math

Location
https://github.com/LeastAuthority/tinlake-math/blob/master/src/math.sol#L34

Synopsis
Safe division requires checking that there is no division by zero. While the default behavior of Solidity is to revert in this case as of compiler version 0.4.0, it reverts using an invalid opcode rather than a gas preserving revert. The safeDiv provided in the math library for Tinlake does not do any checks for zero division.

Impact
Any call to safe division that may divide by zero will throw an opcode that does not provide information as to what happened and deplete the gas provided to the transaction.

Remediation
Place a require statement in `safeDiv` that ensures the divisor must be greater than zero, and supply an error message if this is not the case.

```
require(y > 0, “Division by zero”);
```

Status
The pull request to the safe math library now requires that division by zero is not possible and an error message is supplied.

Verification
Resolved.

Suggestions

Suggestion 1: Author Test Suite for Registry Contract

Location
https://github.com/LeastAuthority/tinlake-registry/blob/master/src/registry.t.sol

Synopsis
The test file for the registry contract contains stubbed test functions and does not actually implement a test suite.

Mitigation
Author tests for the registry contract for better coverage.

Status
Unit tests for the registry have been committed, which provide coverage of the functionality of the registry.

Verification
Resolved.

Suggestion 2: Improve Documentation

Location

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/tranche/tranche.sol

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/shelf.sol

Synopsis

The overview documentation for Tinlake in the HackMD provides a good overview of the entire system, including diagrams that show internal dependencies of the interworking modules. We suggest that the internal dependencies of each module be further documented, specifically the events that may occur over time that trigger the actions taken by these modules or oracles acting external to these modules.

We also strongly recommend Increasing documentation on the authentication system with information on why each method is modified with wards. There are many wards throughout the system, some of which are contracts while others are trusted oracles or potentially other permissioned roles. Adding documentation to these wards will increase the ability to reason about where trust is being placed.

Furthermore, a few external methods are missing documentation. For example, there is no description of the `recover()` and `close()` functions in the shelf contract. The operator documentation is also still nonexistent. A few contracts also have minimal code comments (i.e. the shelf and tranche contracts listed in the locations section above).

Mitigation

Create more documentation for all of the specific functionality of each Tinlake contract. In addition, include more code comments (i.e. the rationale behind deployment strategies, inheritance, and ownership of modularized components). Place a comment on each ward setting as that documents the role of this permission.

Status

A pull request including a list of comments has been added to the codebase. As a result, the Tinlake Developer Documentation now contains more information about the way modules interact, thus providing a clearer understanding and easier comprehension for those who are new to the system. The Centrifuge team has also notified Least Authority that they will continue to update the documentation, particularly as additional examples and scenarios of possible events that may occur over time throughout the system are made apparent.

Verification

Resolved.

Suggestion 3: Improve Consistency

Location

Byte string not used to select storage:

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/shelf.sol#L102

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/pile.sol#L118

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/collect/collector.sol#L84 https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/ceiling/principal.sol#L33

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/ceiling/creditline.sol#L57

Byte string used to select storage:

https://github.com/LeastAuthority/tinlake/blob/develop/src/borrower/price/pool.sol#L38

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/tranche/senior_tranche.sol#L55

https://github.com/LeastAuthority/tinlake/blob/develop/src/lender/assessor/base.sol#L79

Synopsis

In several instances, we found some inconsistency with design patterns, where optimization is the goal for choosing a superior pattern (i.e. The `file()` functionality sometimes uses the byte string selector with a single value approach and other times supplies all values with no selector). This could cause some confusion in a client implementation as to which method to apply for various modules.

Mitigation

Choose to implement `file()` functions to always take a byte string to select the functionality of storage so that it remains consistent.

Status

A pull request that replaces the functions that were not using the byte string selector with a version where all functions are consistent has been added to the codebase.

Verification

Resolved.

Suggestion 4: Remove Redundant Code

Location

https://github.com/LeastAuthority/tinlake-math/blob/master/src/interest.sol#L28

https://github.com/LeastAuthority/tinlake-math/blob/master/src/interest.sol#L42

https://github.com/LeastAuthority/tinlake/blob/develop/src/root.sol#L65

Synopsis

We found redundant code in several locations:

The `block.timestamp >= lastUpdated` check in `Interest#compounding()`on `line 28` is redundant as it will always be checked again on line 42 in `Interest#chargeInterest()`.

The function call that sets the distributor address in the collector contract is called twice within the same function body and appears to be redundant. This is a simple oversight, and we suggest that more time is spent ensuring that the code base is correctly linted and does not contain any unnecessary complexity.

Impact

This is not a security threat and only causes incoherency in the code.

Remediation

Remove the check from `Interest#compounding()`.

Remove the second call `toDependLike(borrowerDeployer.collector()).depend("distributor", distributor_)`.

Status

A commit that removes the duplicate `DependLike()` code has been added.

A commit that creates an internal function for `chargeInterest()` to create an alternative execution path that will not call the time check twice has been added.

Verification

Resolved.

Recommendations

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

We commend the Centrifuge team for expanding on the documentation such that it covers additional scenarios and we encourage continuous improvement of the documentation so that it incorporates new potential scenarios in addition to threat modeling.

We also recommend improved documentation on the ward system for a specific NFT deployment, including additional details on all permission levels for the example provided on SME invoices. Further clarification and documentation on the Events section would also help in providing more complete and comprehensive documentation. These changes will simplify the effort to both further develop and audit the codebase, therefore minimizing the risk that security vulnerabilities will go undiscovered.

Finally, we recommend periodic security reviews to build upon this foundation and reduce the risk for new vulnerabilities in the system.

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.