PUBLIC

Code Assessment

of the Fluorine Smart Contracts

July 05, 2023

Produced for

by

Contents

1 Executive Summary

Dear TrueFi Team,

Thank you for trusting us to help TrueFi with this security audit. Our executive summary provides an overview of subjects covered in our audit of the latest reviewed contracts of Fluorine according to Scope to support you in forming an opinion on their security risks.

TrueFi implements an uncollateralized loan platform. Whitelisted users can create their own portfolios and have full control over them. Users can be lenders by buying shares of tranches which implement different investment strategies.

The most critical subjects covered in our audit are asset solvency, functional correctness and front-running resilience. Functional correctness has improved to a good level after the deficit calculation has been fixed in the underlying Carbon contracts, fixing a Wrong distribution of unpaid fees in repay(). Certain configurations and behaviors by the manager of a vault can enable a Sandwich attack on updateState which allows an attacker to extract value out of the protocol. It is therefore detrimental that managers disable withdrawals and/or deposits in `Live` state as soon as such attack vectors open up.

The general subjects covered are complexity, deployment, testing and documentation. We believe that all the other aforementioned areas offer a high level of security. The documentation is comprehensive and unit testing is extensive. However, we need to emphasize that the complexity of the codebase is high and the system can be in many different states which might require different handling, and thus our confidence in that regard is limited.

In summary, we find that the codebase provides a good level of security. Since the project is deeply intertwined with another TrueFi project, we would also like to refer to the note Relevant concerns of TrueFi Carbon smart contract audit report which details concerns that are also relevant for this project.

It is important to note that security audits are time-boxed and cannot uncover all vulnerabilities. They complement but don't replace other vital measures to secure a project.

The following sections will give an overview of the system, our methodology, the issues uncovered and how they have been addressed. We are happy to receive questions and feedback to improve our service.

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

Below we provide a brief numerical overview of the findings and how they have been addressed.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	2
• Code Corrected	1
• Risk Accepted	1
Low-Severity Findings	1
• Code Corrected	1

2 Assessment Overview

In this section, we briefly describe the overall structure and scope of the engagement, including the code commit which is referenced throughout this report.

2.1 Scope

The assessment was performed on the source code files inside the Fluorine repository based on the documentation files. The table below indicates the code versions relevant to this report and when they were received.

V	Date	Commit Hash	Note
1	02 June 2023	fdac576d9f67bbee7b2fec6738bd426bf7088567	Initial Version
2	29 June 2023	146eadf96e556cb599de4fa36b31ca9686733631	Initial Version

For the solidity smart contracts, the compiler version `0.8.18` was chosen.

The following contracts of the repository are in scope:

- contracts/StructuredAssetVault.sol
- contracts/StructuredAssetVaultFactory.sol
- contracts/proxy/Upgradeable.sol
- contracts/proxy/ProxyWrapper.sol

The contracts are tightly integrated with the contracts in the `trusttoken/contracts-carbon` repository. The contracts are out of scope for this audit report. However, some issues and concerns of this repository also hold for the contracts reviewed in this report.

We therefore refer to the audit report of the Carbon contracts: https://chainsecurity.com/wp-content/uploads/2023/07/TrueFi_Carbon_-Smart-Contract-Audit-_-ChainSecurity.pdf (snapshot).

Please see Relevant concerns of TrueFi Carbon smart contract audit report for further details.

2.1.1 Excluded from scope

Libraries and all contracts not mentioned in the previous section are excluded from scope.

2.2 System Overview

This system overview describes the initially received version (Version 1) of the contracts as defined in the Assessment Overview.

Furthermore, in the findings section, we have added a version icon to each of the findings to increase the readability of the report.

TrueFi offers an uncollateralized lending platform that is managed by trusted managers. Users can supply tokens to the platform or apply as borrowers to the manager of a portfolio.

The system consists of 3 main components:

- `TrancheVault` implements an EIP-4626 compliant tokenized vault. Tranches have different risk-return profiles. It is the entry-point for the lenders to deposit their assets.

- `StructuredAssetVault` or `SAV` implements the logic of a multi-tranche portfolio in the system. There can be up to 3 tranches managed by a Structured Asset Vault.
- `StructuredAssetVaultFactory` firstly instantiates tranches defined by the user and then an `SAV` to manage them.

Tranche Vaults

These are the tokenized vaults with which lenders interact to provide or remove capital from the system. They are compliant with the EIP-4626. It exposes the following functions:

- `deposit`: Users deposit an amount of the underlying token and mint some shares of the vault. There is a ceiling on the amount they can deposit determined by the `DepositController` contract. Part of the amount deposited is used to pay the deposit fee.
- `mint`: It works similarly to `deposit` but the users specify the amount of shares they want to mint. This call is also subject to the deposit fee.
- `withdraw`: Users withdraw an amount of the underlying token and burn the respective amount of shares. Part of the amount withdrawn is used for the withdrawal fee to get paid. There is a floor for the total assets that can be withdrawn determined by the `WithdrawController` contract.
- `redeem`: It works similarly to `withdraw`, but users specify an amount of shares they want to burn and get the corresponding underlying amount out of the tranche.
- `updateCheckpoint`: It distributes fees to the protocol and the manager and accrues interest on each tranche's values. It is callable by anyone.

TrueFi has designed Fluorine based on the assumption that the manager is a fully trusted entity, as they can remove funds from the system. Hence, a proposed manager should firstly be whitelisted by the TrueFi DAO. Managers can disburse funds either to users whitelisted by TrueFi or to anyone. The latter will show a warning on TrueFi's frontend.

The Manager, in this implementation, performs all disbursing/repaying calculations off-chain but has to provide an asset report which embodies all actions taken, assets, and any other off-chain information regarding the vault state. Each action emits an event with its own `actionId` which the report should refer to.

The tranches are divided into two categories. The Equity tranche (tranche with index 0) is suitable for the maximum risk investors, accruing all dynamic interest that remains from the tranches above which fall into the second category: tranches with fixed interest. The system is meant to be used with 3 tranches: The equity tranche as explained above, the junior tranche with a higher fixed interest rate and medium risk and the senior tranche with a lower fixed interest rate and low risk. Losses (due to defaulted loans) are absorbed by tranches with higher risk.

Waterfall Calculation

Calculating the value each tranche should hold is done in a *waterfall* approach. Waterfall is only used when the SAV is `Live` or `Closed`. In a nutshell, a tranche can hold some value if and only if all the tranches above (the safer tranches) have met their desired performance. This means that if a tranche has a deficit, then all the riskier tranches hold no value.

Structured Asset Vault:

SAV is the smart contract which implements the borrowing logic of the system. An SAV can be in one of the following states:

- `CapitalFormation`: This is the state the portfolio is in when it is instantiated. At this point, it holds no funds and, thus, users cannot borrow.
- `Live`: Upon entering this phase, assets are moved from the tranches to SAV. Therefore, users can borrow and repay their loans. Only the manager can change the state to `Live`

through calling `start()`. SAV can remain live for `assetVaultDuration` time. After that, anyone can close it.

- `Closed`: This is the terminal state of SAV. At this state, the portfolio returns the principal assets it holds to the tranches. The manager can still repay assets and update the outstanding assets.

Importantly, an SAV can move from `CapitalFormation` directly to `Closed` state, either through a call from the manager or when `startDeadline` has passed and system has not yet entered `Live` state.

Each SAV has a manager who has full control over it and is allowed to call most entry points of the system. In particular, a manager can call:

- `start()`: This sets the portfolio to `Live` status and withdraws the amount of principal tokens that have been accumulated during the `CapitalFormation` phase. This call makes sure that the ratio of the amounts deposited in all tranches is correct.
- `close()`: This sets the state of the portfolio to closed. A portfolio can close if its end date has passed or if there are no more outstanding assets to be repaid. In the former case, everyone can close the portfolio, in the latter only the manager can. A portfolio can also be closed from the `CapitalFormation` phase either by the manager or if the `startDeadline` has passed.
- `updateState()`: This function is used by the manager to update the outstanding assets. This can either be used to signal new interest payments (by increasing the outstanding assets) or to mark the value of a loan as defaulted (by decreasing the outstanding assets). It can be called during `Live` as well as during `Closed` state.
- `disburse()`: This function can only be called during `Live` state. When called, it transfers the defined amount of assets to the receiver and increases the outstanding assets. It is worth mentioning, that if `onlyAllowedBorrowers` is set to `true`, the receiver must hold `BORROWER_ROLE`. The manager can also call `disburseThenUpdateState()` to instantly add interest after a disbursal.

Any user holding `REPAYER_ROLE` (which is assigned by the manager and also held by the manager) can call:

- `repay()`: Any loan can be repaid at any time as long as the repaid amount is smaller than the vault's current outstanding assets. The manager can also call `updateStateThenRepay()` to update the outstanding assets before repaying.

Fees:

Many different fees are accrued by the system. These are:

- Deposit/Withdraw fees: These are a percentage of the amount deposited or withdrawn. They are sent to the manager of the SAV.
- Protocol/Manager fees: These are accrued over the life of SAV and are essentially a performance fee. They are accrued on the total balance (including interest) in any operation that updates a vault's checkpoint.

Controllers:

The controllers implement the specific logic for particular actions in the tranches. They are configurable by the manager. In the current implementation the controllers are the following:

- `DepositController`: It defines a configurable ceiling i.e., the max amount that can be deposited to a vault. It implements the constraints for depositing/minting.
- `WithdrawController`: It defines a configurable floor i.e., the minimum amount that can be left in a vault after a withdrawal. It implements the constraints for withdrawing/redeeming.
- `TransferController`: It determines which transfers of the vault tokens are allowed.

2.2.1 Workflow of System

Fluorine goes through the following states:

- A whitelisted manager calls `StructuredAssetVaultFactory.createAssetVault()` with a tranche configuration including deployed deposit, withdraw and transfer controller addresses for each tranche. This function firstly deploys tranches along with Clones of the given controllers and then deploys an SAV containing these tranches.
- A freshly deployed SAV starts with the `CapitalFormation` state, during which lenders can deposit into their tranches of interest and receive LP tokens in return as long as their deposited value does not reach a limit (specified by the manager through the `DepositController`). During this period, neither fees/interest are accrued, nor assets can be disbursed.
- When enough capital has been gathered (summation of funds in all tranches exceeding `minimumSize`) and the amounts in each tranche are in a certain ratio to each other, the manager can call `start()` and move the state of SAV to `Live`. As soon as this state is entered, protocol and manager fees, as well as interest in the tranches is getting accrued.
- During `Live` state, funds can be disbursed to borrowers (either whitelisted or not). Note that even though `onlyAllowedBorrowers` might be set and funds be transferred to the whitelisted addresses, they can later be forwarded from these whitelisted ones to any other address.
- Any user holding `REPAYER_ROLE` can call `repay()` to repay a loan. It is up to the users how they desire to distribute the repayment between principal and interest repayment, as long as they do not exceed the outstanding principal for the returned principal and the outstanding assets for the total amount.
- Once `endDate` has been reached (or if no assets are outstanding anymore), the SAV can be closed. In this case, the asset values are moved to the tranches according to the waterfall calculation.
- Despite being in `Closed` state, users can still repay their loans to the SAV. For senior and then junior tranche, SAV tries to distribute the repaid loans until the underlying assets reach the expected value according to the waterfall algorithm. For equity tranches, however, no limits are enforced.

2.2.2 Trust Model and Roles

The roles defined by the system are the following:

- SAV manager: They have to be whitelisted by the TrueFi DAO. They can configure the various controllers, update outstanding assets and repay loans. This is the most privileged role of an SAV and it is completely trusted by Fluorine to never try to harm the users. Among others, the managers can (intentionally or unintentionally):
  - Issue a loan to themselves and disappear with the money.
  - Set arbitrary controllers that can be used to rug pull the SAV.
  - Use reentrant underlying tokens that can be used to exploit certain aspects of the code.
  - Close an SAV prematurely by setting the outstanding assets to 0 and calling `close()`.
  - Manipulate outstanding assets so that user deposits receive less (or no) shares.
- The protocol admin: This is the role that can upgrade the contracts and assign roles to new addresses for `StructuredAssetVault` and `TrancheVault`. It holds `DEFAULT_ADMIN_ROLE` and can whitelist users to become managers and create their own SAVs.
- The pausers: This is the role that can pause/unpause the main functionalities of `StructuredAssetVault`. `TrancheVault` functions also rely on the pause status set in the portfolio.

- The tranche controller owner: Each tranche is using a different set of controllers which can be changed after deployment. Initially, the SAV manager is able to change the controllers but the protocol admin can assign these rights to different addresses.
- The lenders: Any user to mint shares of a tranche.
- The borrowers: The users who have received a loan from the system. Depending on the setup of the system, borrowers can be whitelisted or not.

3 Limitations and use of report

Security assessments cannot uncover all existing vulnerabilities; even an assessment in which no vulnerabilities are found is not a guarantee of a secure system. However, code assessments enable the discovery of vulnerabilities that were overlooked during development and areas where additional security measures are necessary. In most cases, applications are either fully protected against a certain type of attack, or they are completely unprotected against it. Some of the issues may affect the entire application, while some lack protection only in certain areas. This is why we carry out a source code assessment aimed at determining all locations that need to be fixed. Within the customer-determined time frame, ChainSecurity has performed an assessment in order to discover as many vulnerabilities as possible.

The focus of our assessment was limited to the code parts defined in the engagement letter. We assessed whether the project follows the provided specifications. These assessments are based on the provided threat model and trust assumptions. We draw attention to the fact that due to inherent limitations in any software development process and software product, an inherent risk exists that even major failures or malfunctions can remain undetected. Further uncertainties exist in any software product or application used during the development, which itself cannot be free from any error or failures. These preconditions can have an impact on the system's code and/or functions and/or operation. We did not assess the underlying third-party infrastructure which adds further inherent risks as we rely on the correct execution of the included third-party technology stack itself. Report readers should also take into account that over the life cycle of any software, changes to the product itself or to the environment in which it is operated can have an impact leading to operational behaviors other than those initially determined in the business specification.

4 Terminology

For the purpose of this assessment, we adopt the following terminology. To classify the severity of our findings, we determine the likelihood and impact (according to the CVSS risk rating methodology).

- *Likelihood* represents the likelihood of a finding to be triggered or exploited in practice
- *Impact* specifies the technical and business-related consequences of a finding
- *Severity* is derived based on the likelihood and the impact

We categorize the findings into four distinct categories, depending on their severity. These severities are derived from the likelihood and the impact using the following table, following a standard risk assessment procedure.

Likelihood	Impact		
	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

As seen in the table above, findings that have both a high likelihood and a high impact are classified as critical. Intuitively, such findings are likely to be triggered and cause significant disruption. Overall, the severity correlates with the associated risk. However, every finding's risk should always be closely checked, regardless of severity.

5 Findings

In this section, we describe any open findings. Findings that have been resolved have been moved to the Resolved Findings section. The findings are split into these different categories:

- Security: Related to vulnerabilities that could be exploited by malicious actors
- Design: Architectural shortcomings and design inefficiencies
- Correctness: Mismatches between specification and implementation

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1
• Sandwich Attack on updateState Risk Accepted	
Low-Severity Findings	0

5.1 Sandwich Attack on updateState

Security Medium Version 1 Risk Accepted

CS-TFFlourine-001

`StructuredAssetVault.updateState()` allows the manager to add interest payments to the vault or default loans on-chain. The frequency of such updates is entirely up to the manager. Thus, it is possible that such updates occur rarely and add a large amount of interest at once.

If deposits are allowed during the `Live` state of the vault, this is problematic as users depositing right before an update receive the same amount of interest as users that deposited earlier.

If withdrawals are also allowed during the `Live` state of the vault, this behavior becomes exploitable. Consider the following example:

- A given vault accrues 0% fees.
- Each tranche accrues 0% interest (for demonstration purposes).
- Each tranche holds a value of 100 tokens.
- The manager disburses 150 tokens, setting `outstandingAssets` to 150.
- After 1 year, the manager updates the `outstandingAssets` to 200.
- An attacker frontruns the call to `updateState()` with a deposit of 100 tokens to the equity tranche.
- After the `updateState()` call has been processed, the attacker can withdraw their received shares for an instant profit of 25 tokens.

Risk accepted:

The client accepts the risk with the following statement:

> Manager should disable withdrawal and deposits if there will be a big change of value in update state.

6 Resolved Findings

Here, we list findings that have been resolved during the course of the engagement. Their categories are explained in the Findings section.

Below we provide a numerical overview of the identified findings, split up by their severity.

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	1

- Wrong Distribution of Unpaid Fees in Repay() Code Corrected

Low-Severity Findings	1

- Disburse to 0-Address Code Corrected

6.1 Wrong Distribution of Unpaid Fees in Repay()

Correctness Medium Version 1 Code Corrected

CS-TFFlourine-003

`StructuredAssetVault._calculateWaterfall()` first calculates the waterfall without fees, then subtracts the fees from each waterfall value:

```
uint256[] memory waterfall = _calculateWaterfallWithoutFees(assetsLeft);
uint256[] memory fees = new uint256[](tranches.length);
for (uint256 i = 0; i < waterfall.length; i++) {
    uint256 waterfallValue = waterfall[i];
    uint256 pendingFees = tranches[i].totalPendingFeesForAssets(waterfallValue);
    waterfall[i] = _saturatingSub(waterfallValue, pendingFees);
    fees[i] = pendingFees;
}
```

If all value of a vault has been disbursed before and there are unpaid fees in the tranches, a `repay()` will repay the unpaid fees of the tranches first and then start to add value to the senior tranche. However, due to the aforementioned calculation of the waterfall, the unpaid fees of the lower tranches are not removed from the value that is added to the senior tranche's `checkpoint.totalAssets`. Consider the following example:

- Each tranche has 0 value, 5 tokens in unpaid fees and a deficit of 95 tokens.
- A repayment of 20 tokens occurs.
- Each tranche now has 0 tokens in unpaid fees.
- The senior tranche now has a value of 15 tokens in its checkpoint and a deficit of 85 tokens.

Since only 5 tokens are in the end kept in the vault's `virtualTokenBalance`, another call to `updateCheckpoints()` updates the senior tranches checkpoint to just 5 tokens and a deficit of 95 tokens.

However, since `TrancheVault.deposit()` does not update its own deficit before processing a deposit, any deposit (e.g., 1 wei) can cause this wrong deficit to become persistent because the checkpoint is updated with the actual waterfall value (going from 15 tokens to 5 tokens + 1 wei) and

subsequent calls to `updateCheckpoints()` can only calculate the deficit with the currently stored values.

Code corrected:

Deficits are now stored in the `TrancheVault` checkpoints instead of `StructuredAssetVault` and are calculated on every checkpoint update in the tranches. Additionally, `TrancheVault._payFee` now limits the fees to the waterfall value of the tranche so that unpaid fees in empty tranches are no longer paid out.

6.2 Disburse to 0-Address

Design | Low | Version 1 | Code Corrected

CS-TFFlourine-004

`StructuredAssetVault.disburse()` allows the transfer of tokens to the 0-address. This could be problematic, for example, if a bug in the frontend the manager uses passes an uninitialized parameter to the function call.

Code corrected:

Disbursals to the 0-address are no longer possible.

7 Informational

We utilize this section to point out informational findings that are less severe than issues. These informational issues allow us to point out more theoretical findings. Their explanation hopefully improves the overall understanding of the project's security. Furthermore, we point out findings which are unrelated to security.

7.1 Event Emitted When Nothing Has Changed

Informational **Version 1**

CS-TFFlourine-002

`StructuredAssetVaultFactory.setAllowedBorrower()` emits the event `AllowedBorrowersChanged` every time a borrower is added or removed. If a borrower is added that has already been added or if a borrower is removed that has already been removed, the event is still emitted because the return values of the `EnumerableSet` (the type of the `_allowedBorrowers` state variable) functions `add()` and `remove()` are never checked.

8 Notes

We leverage this section to highlight further findings that are not necessarily issues. The mentioned topics serve to clarify or support the report, but do not require an immediate modification inside the project. Instead, they should raise awareness in order to improve the overall understanding.

8.1 Asset Report Identifiers Not Unique

Note Version 1

Each time the manager changes the state of the vault (regarding loans) a string identifier is added to the contract that is linked to a JSON report. However, there is no guarantee, that the manager reuses a string they have used before (except if the same string was used in the last call).

8.2 Relevant Concerns of TrueFi Carbon Smart Contract Audit Report

Note Version 1

This audit report covers Fluorine which is dependent on some logic of another product of TrueFi: Carbon. Since both projects share similarities, the following points, which have been covered in the Carbon audit report, are also valid for Fluorine.

The report can be found at https://chainsecurity.com/wp-content/uploads/2023/07/TrueFi_Carbon_-Smart-Contract-Audit-_-ChainSecurity.pdf (snapshot).

- *5.2 DoS for Start*
- *5.3 Loan Default Frontrunning*
- *5.4 Fee Transfer DoS*
- *7.1 Ambiguous Deficit Data in Closed State*
- *7.2 Compounding Interest Computed in Arbitrary Intervals*
- *7.3 Fee Accrual in Closed State*
- *7.4 Fee Accrual on Yield*
- *7.5 Manager Fee Accrual*
- *7.7 Skewed Interest Distribution*
- *7.8 Use of Non-standard ERC20 Tokens*

Please note, that the Carbon equivalent to `StructuredAssetVault` is called `StructuredPortfolio`.

Please also note, that a "loan default" in some of these issues is equivalent to a call to `StructuredAssetVault.updateState()` with a value that decreases the outstanding assets of the vault.

8.3 updateState in Closed State

Note Version 1

The manager is able to call `updateState()` in `Closed` state. Users should be aware that it is still possible to receive interest payments that were not accounted for when the vault has been closed.