PUBLIC

Code Assessment

of the Carbon Smart Contracts

July 04, 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 Carbon 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 the valuation of the portfolios and their tranches, the fee and interest calculations, the interactions of the lenders and the borrowers with the system and the access control. For the tranche valuation, we uncovered a Waterfall miscalculation issue. Under certain circumstances, the value of riskier tranches could be absorbed by higher tranches. The issue was addressed in the second iteration of the report. Attack vectors initiated by the portfolio managers were considered out of scope. In the current version, all the uncovered issues have been either addressed or acknowledged.

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 really high and the system can be in many different states which might require different handling, and thus our confidence in that regard is limited.

Moreover, we would like to emphasize that portfolio managers are highly trusted and can introduce security risks to the protocol. The security of Carbon instances therefore ultimately depends on external factors.

In summary, we find that the codebase with the latest version greatly improved on the initial version. An iterative audit of many iterations adds risk as reviews of multiple small changes can introduce novel interactions with existing code which are easy to miss. Overall, we find that the codebase in its current state provides a good level of security.

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	2
• Code Corrected	1
• Acknowledged	1
Medium-Severity Findings	5
• Code Corrected	3
• Risk Accepted	2
Low-Severity Findings	14
• Code Corrected	10
• Specification Changed	2
• Risk Accepted	2

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 Carbon 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	21 November 2022	6a5daad4a75afb51b4ad6b313e2edf89634def43	Initial Version
2	03 January 2023	ee210818bee8f9a49b62a2d897ae36ac78e84d05	Second Version
3	18 January 2023	d79955bab2181048302fd920eda9931ddc138ccf	Third Version
4	26 January 2023	49ffbec1a180c5a4b4d5aa1a95ad4074f827e58b	Fourth Version
5	2 February 2023	e0c27c284e64b17f87e5cb35d85acf229762b63e	Fifth Version
6	7 February 2023	c3fd736fb77e06cc93eb98094270f47ad19c0fca	Sixth Version
7	23 February 2023	e8cda9fc375bb2cad91be5b09a5b81627d085efa	Seventh Version
8	27 February 2023	f209b1ed23db18485b585f5fb003d8eb0a64c962	Eighth Version
9	26 May 2023	81ea1870405723bc8686d84dc26ca5b15a903dd0	Eighth Version
10	29 June 2023	cf54dc484881ef1f6cde32915b18b51fe355abfd	Ninth Version

For the solidity smart contracts, the compiler version `0.8.16` was chosen.

The following smart contracts in `contracts` directory are in scope:

- `controllers/*`
- `lenderVerifiers/AllowAllLenderVerifier.sol`
- `proxy/*`
- `LoansManager.sol`
- `ProtocolConfig.sol`
- `StructuredPortfolioFactory.sol`
- `StructuredPortfolio.sol`
- `TrancheVault.sol`

2.1.1 Excluded from scope

Excluded from scope are all the contracts that are not explicitly mentioned above. In particular, this includes:

- `FixedInterestOnlyLoans`
- All the libraries (e.g., OpenZeppelin)

- Deployment process

The aforementioned contracts are out of scope and considered to function as expected. Moreover, all the possible attack vectors a manager can employ are out of 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. Users can be both borrowers and lenders on the platform.

The system consists of 3 main components:

- `StructuredPortfolio`: It implements the logic of the portfolios in the system.
- `TrancheVault`: It implements an EIP-4626 compliant tokenized vault where lenders offer their capital to be borrowed.
- `StructuredPortfolioFactory`: It instantiates a portfolio with a given amount of tranches.

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 limit 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`: Any user can call this as long as the portfolio is closed. It pays the fees to the protocol and the manager. In Version 2, it also updates the deficit checkpoints for defaulted loans with the accrued interest.

Due to the undercollateralized nature of the project, lenders and borrowers are supposed to be elaborately verified by the portfolio manager. This would, in general, require KYC for all participants. For this reason, the participants have to be whitelisted on-chain. As certain actions can be performed on behalf of addresses (e.g., `deposit`), Carbon verifies these addresses instead of the address on the receiving end. In Version 2, this has been changed in favor of verification of the receiver address. This means that any address is allowed to deposit tokens on behalf of a whitelisted address.

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 holds is very particular in Carbon. The algorithm used is named *Waterfall*. Waterfall is only used when the portfolio is `Live` and, thus, all the value of each tranche is deposited in the portfolio itself. In a nutshell, a tranche can hold some value if and only if all the tranches which are safer than this have met their desired performance. This means that if a tranche has a deficit, then all the riskier tranches hold no value.

Structured Portfolio:

The portfolio is the smart contract which implements the borrowing logic of the system. A portfolio can be in three states/phases:

- `CapitalFormation`: This is the state the portfolio is in when it's instantiated. At this point it holds no funds and, thus, users cannot borrow.
- `Live`: In this phase, users can borrow and repay their loans. Only the manager can start the portfolio and turn it `Live`. A portfolio can remain live for `portfolioDuration` time. After that, anyone can close it.
- `Closed`: This is the terminal state of the portfolio. At this state, the portfolio returns the principal assets it holds to the tranches. Users are only allowed to repay assets.

Importantly, a portfolio can move from `CapitalFormation` directly to `Closed` state if needed.

Each portfolio has a manager who has full control over the portfolio 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 it has no active loans. 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.
- `addLoan`: The manager can add a loan to the accounting contract i.e., `FixedInterestOnlyLoans`. This just specifies the details of the loan.
- `fundLoan`: If the loan has been accepted by the receiver/borrower, the manager can send the required funds to the borrower. The loan is then added to the active loans of the portfolio.
- `markLoanAsDefaulted`: The manager can mark a loan as defaulted. This leads to the loan being removed from the active loans. Moreover, the total value the portfolio holds as well as the value of each tranche is updated.
- `cancelLoan`: The manager can cancel a loan.

Any user can call:

- `repayLoan`: A user can repay their own loans only. Note that they might pay them, even if they are marked as defaulted, if this has been specified when the loan was added. The value of the portfolio and each tranche is updated.

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 portfolio.

- Protocol/Manager fees: These are accrued over the life of the portfolio and they essentially are 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 Changes in Version 3

In the third version, the system allows loans to be marked as defaulted after closing the portfolio. Moreover the checkpoint calculation has been greatly simplified.

2.2.2 Trust Model and Roles

The roles defined by the system are the following:

- The portfolio manager: They need to be whitelisted by the portfolio factory. They can configure the various controllers. This is the most privileged role of a portfolio and it is completely trusted by Carbon 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 portfolio.
  - Use reentrant underlying tokens that can be used to exploit certain aspects of the code.
- The protocol admin: This is the role that can upgrade the contracts and assign roles to new addresses for `StructuredPortfolio` and `TrancheVault`. The protocol admin whitelists users to become portfolio managers and create their own structured portfolios.
- The pausers: This is the role that can pause the main functionality in `StructuredPortfolio`. `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 portfolio manager is able to change the controllers but the protocol admin can assign these rights to different addresses.
- The lenders: Any allowed user to mint shares of a tranche.
- The borrowers: The users who have received a loan from the system.

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	1

- Defaulted Loan Repayment Acknowledged

Medium-Severity Findings	2

- DoS for Start Risk Accepted
- Loan Default Frontrunning Risk Accepted

Low-Severity Findings	2

- Fee Transfer DoS Risk Accepted
- Lost Repayments in Closed State Risk Accepted

5.1 Defaulted Loan Repayment

Design High Version 1 Acknowledged

CS-TFCarbon-001

Loans can be marked to be repayable after default. In the case of a defaulted loan being repaid later, new investors of the equity tranche gain an unfair advantage over users that invested in the tranche before the loan has been marked as defaulted.

This behavior can even be exploited by borrowers to regain some of the repaid loan by investing in the equity tranche after the default of their own loan. Consider the following example (assuming no fees and interest for simplification):

- A portfolio consists of 3 tranches and is in `Live` status with no active loans.
- Users have deposited 100 tokens to each tranche with no accrued interest (i.e., 1 share per token).
- A new loan of 99 tokens is issued to a borrower.
- After some time, the manager marks the loan as defaulted.
- The value of the equity tranche is now 1 token with a total supply of 100 shares.
- The borrower deposits 100 tokens into the equity tranche and receives 10,000 shares back.
- The borrower repays the loan, raising the equity tranche's value to 200 tokens.
- The borrower is now entitled to ~198 tokens in the tranche.

Issue acknowledged:

TrueFi replied:

> we are aware of this issue, but it's more of manager responsibility to pause deposits/withdrawals in case of risk in portfolio (so before marking loan as defaulted manager should first disable deposits/withdrawals)

5.2 DoS for Start

Design | Medium | Version 1 | Risk Accepted

CS-TFCarbon-002

The manager can turn the system `Live` by calling `StructuredPortfolio.start()`. When this is done, the system checks if the ratios of the values stored in each tranche are appropriate. If it is not, the transaction reverts. This means that depositing or withdrawing an amount - if allowed - before the manager calls start can block the system from turning live since the ratios will not be correct. The `depositController` and `withdrawController` enforce a ceiling and a floor respectively but the issue can still arise.

Risk accepted:

TrueFi accepted the risk giving the following statement:

> Shouldn't occur, but in case managers want to be safe, withdrawals and deposits can be disabled before starting the portfolio.

5.3 Loan Default Frontrunning

Security | Medium | Version 1 | Risk Accepted

CS-TFCarbon-003

In `Live` state, and if withdrawals are enabled, lenders in any tranche can withdraw the full amount even if there are open loans that are using some of the available funds. Consider the following example:

- Each tranche has a value of 100 tokens.
- A loan for 150 tokens has been issued.
- Users of the junior tranche now withdraw all 100 tokens.
- The loan defaults resulting in the value of the senior tranche being reduced to 50.

If the users in the junior tranche observe the call to `StructuredPortfolio.markLoanAsDefaulted`, they can frontrun it to redeem all of their tokens, while the other tranches suffer from the loss.

Risk accepted:

TrueFi accepted the risk giving the following statement:

> Shouldn't occur, but in case managers want to be safe, withdrawals can be disabled before calling `StructuredPortfolio.markLoanAsDefaulted`.

5.4 Fee Transfer DoS

Security | Low | Version 1 | Risk Accepted

CS-TFCarbon-004

Fees are (as long as funds are available) transferred onto the protocol treasury and manager beneficiary address on every interaction. Since the manager beneficiary might be an EOA also used in other activities, and since some tokens implement blacklisting (e.g., USDC) or pausing (e.g., BNB), portfolios using such underlying tokens could be susceptible to a Denial of Service when the manager beneficiary address is used in an illicit activity or the token is set to a paused state.

Risk accepted:

TrueFi responded:

> Both addresses are trusted and can be changed in case of emergency. Changes were introducing too much complexity in the code. There was a code change in `setManagerFeeBeneficiary` to first change beneficiary and then pay the fee as otherwise function was reverting.

5.5 Lost Repayments in Closed State

Design | Low | Version 1 | Risk Accepted

CS-TFCarbon-005

If a portfolio is closed, users are encouraged to withdraw their assets as no interest accrues anymore but protocol fees are still accrued. If a tranche is completely emptied by withdrawals (i.e. there are no more shares) and a repayment occurs, the repaid value in that tranche is lost as no one is able to withdraw these assets (apart from the protocol admin that can update the implementation contract).

Risk accepted:

The client accepts the risk with the following statement:

> By default there is protocol fee in closed state to encourage withdrawal of the assets as there always is small smart contract risk. But in case there was a default and the manager is in process of regaining the assets they should create a proposal to disable protocol fees on that portfolio. It's the matter of communication between manager and lenders. If they want to exit early they can, but then they won't be able to profit from recovered funds.

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	1

- Waterfall Miscalculation Code Corrected

Medium-Severity Findings	3

- Calling StructuredPortfolio.updateCheckpoints Increases Deficit Code Corrected
- Fees in Deficit Code Corrected
- No Deficit Update on Deposit / Withdrawal Code Corrected

Low-Severity Findings	12

- Fee Accrual on Unpaid Fees Code Corrected
- Unpaid Fee Calculation Code Corrected
- Balance Underflow Code Corrected
- Gas Optimizations Code Corrected
- Imprecise Specifications Specification Changed
- Missing Interface Functions Code Corrected
- Missing Sanity Check Code Corrected
- Shadowed Variable Code Corrected
- Specification Nonconformity Specification Changed
- Unused Function Code Corrected
- Unused Return Value Code Corrected
- configure Function Does Not Include TransferController Code Corrected

6.1 Waterfall Miscalculation

Correctness High Version 1 Code Corrected

CS-TFCarbon-013

The value of a tranche can be updated by anyone by calling `StructuredPortfolio.updateCheckpoints`. We expect the following property to hold:

> Executing `updateCheckpoints` consecutively in one block (with no other transactions in between) should not change the checkpoints since no time has passed and the value of the portfolio has not been changed.

However, this property does not hold. To showcase that, we need to consider how the waterfall is calculated.

The waterfall is calculated by using each tranche's checkpoints and adding the deficit of a tranche to it. This is considered the total assumed value of the tranche. This means that if we would call

`updateCheckpoints` multiple times, the deficit of a tranche would be added up again on every calculation of the waterfall.

Normally, however, this is not the case since the resulting waterfall values are bound by the total value of the portfolio (virtual token balance + loans value) that hasn't been used by the less risky tranches. Therefore, we assume the following property:

> If a tranche has a deficit > 0, all riskier tranches (i.e., all tranches with a lower waterfall index) have a value of exactly 0.

In practice, this assumption does not hold as can be seen in the following example (assuming no fees and no compounding for simplification):

- The senior tranche has a 1% interest.
- The junior tranche has a 2% interest.
- Each tranche holds a value of 100 tokens.
- Two loans are issued:
    - Loan A: 102 tokens with 0% interest.
    - Loan B: 180 tokens with 10% interest.
- Loan A defaults.
- A deficit of 2 tokens is added to the junior tranche and the equity tranche now holds 0 value.
- After one year, the total assets of the portfolio is 216 (18 tokens in balance + 180 tokens in principal of open loans + 18 tokens in interest).
- The Senior tranche should hold 101 tokens.
- The Junior tranche should hold 102 tokens but still has a deficit of 2 tokens.
- The equity tranche is assigned 13 tokens.

This violates the assumed property. We can now call `StructuredPortfolio.updateCheckpoints` multiple times and add up the deficit of 2 tokens, until the value of the junior tranche is 115 and the value of the equity tranche is 0. After 1 year, the junior tranche has now accrued 15% interest while it should have accrued 2%. In other words, calling the `StructuredPortfolio.updateCheckpoints` consecutively changes the value of the tranches.

Code corrected:

If some loan in a portfolio has defaulted, `StructuredPortfolio.updateCheckpoints` subtracts the delta of the previous and the updated value of a tranche from the tranche's deficit. This allows the deficit to be settled with accrued interest, fixing the described problem.

6.2 Calling `StructuredPortfolio.updateCheckpoints` Increases Deficit

Correctness | Medium | Version 3 | Code Corrected

CS-TFCarbon-007

Given a portfolio where all 3 tranches have a value of 100 tokens each and the manager fee is set to 2% for all tranches, consider the following sequence:

1. A loan is issued for 300 tokens, i.e., the full value held by the portfolio (with 1% interest).
2. The loan is marked as defaulted.
3. This means that the deficit for the senior and the junior tranches is 100 and the value held by them is 0.
4. A year passes by.
5. A user calls `StructuredPortfolio.updateCheckpoints`. At this point the assumed value of the senior tranche is 103 i.e., 2 tokens manager fee plus the deficit which is 101 for each tranche. The real assets held are 0 thus the new deficit is set to 103 which gets checkpointed.
6. A user calls `StructuredPortfolio.updateCheckpoints` again. At this point the assumed value is 105 (103 is the previous deficit plus 2 for the manager fees).

This means that the deficit increases with every call to `updateCheckpoints` while it shouldn't. Note that when the portfolio closes, the deficit is used to set the `maxValueOnClose` which determines the amount of assets expected to be filled into each tranche by possible loan repayments.

Code corrected:

In the current implementation, the fees are properly deducted so that they are not counted multiple times.

6.3 Fees in Deficit

Correctness | Medium | Version 1 | Code Corrected

CS-TFCarbon-010

`StructuredPortfolio._calculateLoansDeficit` updates tranche deficits by calculating the current assumed value of the tranche and subtracting the accrued fees and current waterfall value of that tranche. The fees are calculated on the full assumed value (including the deficit generated by defaulted loans):

```
uint256 assumedPendingFees = tranches[i].totalPendingFeesForAssets(assumedTotalAssets);
```

The actual checkpoints in the tranches are updated by `TrancheVault.updateCheckpointFromPortfolio`. It is calculating fees with the current waterfall value (which does not contain defaulted loans) of the tranche:

```
uint256 pendingFee = _pendingProtocolFee(_totalAssetsBeforeFees);
...
uint256 pendingFee = _pendingManagerFee(_totalAssetsBeforeFees);
```

Where `_totalAssetsBeforeFees` is calculated by this function:

```
function totalAssetsBeforeFees() public view returns (uint256) {
    if (portfolio.status() == Status.Live) {
        return portfolio.calculateWaterfallForTrancheWithoutFee(waterfallIndex);
    }

    return virtualTokenBalance;
}
```

In the case of defaulted loans, these 2 calculations diverge as the assumed value will be higher in tranches with a deficit. Therefore, the calculated fees will be higher as well, reducing the deficits over time while the (paid or unpaid) fees over that time frame are smaller. If at any point the defaulted loan is repaid, the difference will be awarded to the equity tranche.

Consider the following (extreme for demonstration) example:

- Consider a portfolio with a really long duration (more than 50 years).
- Then manager (and/or protocol) fee are 2%.
- Junior tranche has 5% interest, senior tranche has 2% interest.
- Each tranche has 100 value.
- A loan of 300 value, 1 year runtime and 10% interest is issued.
- 1 year later, the loan defaults resulting in 0 value for all tranches and 105 / 103 deficit for junior / senior tranche.
- 50 years later `updateCheckpoints` is called. The deficit of junior and senior tranches is now set to 0. Each tranche has ~2 tokens in unpaid fees.
- The loan is repaid. Junior and senior tranche hold 0 tokens value, while the equity tranche holds 324 tokens value.

Code corrected:

Fees are now calculated only based on the value of the waterfall calculation and they are propagated along the execution of the `StructuredPortfolio.updateCheckpoints` as using the array `pendingFees`.

6.4 No Deficit Update on Deposit / Withdrawal

Correctness **Medium** **Version 1** **Code Corrected**

CS-TFCarbon-015

`TrancheVault.deposit()` / `mint()` and `withdraw()` / `redeem()` update their checkpoints locally but never call `StructuredPortfolio.updateCheckpoints()`. Therefore, the deficit in the `tranchesData` struct is not updated to the latest value before a user deposit / withdrawal is processed. This means, that the period from the last checkpoint update up until the user action is not accounted for. Consider the following example:

- The given vault accrues no fees.
- The junior tranche accrues 5% interest per year.
- The senior tranche accrues 3% interest per year.
- All tranches hold a value of 100 tokens in the beginning.
- 300 tokens have been disbursed.
- 180 tokens have been marked as defaulted.
- The junior tranche now has 20 `totalAssets` and 80 deficit.
- 1 year passes without any interaction on the protocol.
- A new user deposits 100 tokens to the junior tranche.
- The senior tranche now holds 103 tokens while the junior tranche holds 117 tokens and 80 tokens deficit.
- The manager updates the `outstandigAssets` back to 300 tokens.

- The junior tranche now holds a value of 201 tokens.

If we, instead, run `updateCheckpoints` right before the new deposit, the value of the junior tranche is 205 tokens instead. That is, because the call updates the deficit of the tranche to 88 tokens (80 + 3 tokens shifted to senior tranche + 5 tokens interest accrued), accounting for the accrued interest.

Code corrected:

Deficits are now stored in the `TrancheVault` checkpoints instead of the `StructuredPortfolio` and updated every time, the checkpoints are updated.

6.5 Fee Accrual on Unpaid Fees

Design | Low | Version 7 | Code Corrected

CS-TFCarbon-006

Fees are calculated by `TrancheVault.totalPendingFeesForAssets` on the current waterfall value of the tranche. The waterfall value is calculated by `_assumedTrancheValue` which calculates `totalAssets`:

```
uint256 assumedTotalAssets = _withInterest(checkpoint.totalAssets, targetApy, timePassedSinceCheckpoint) +
        checkpoint.unpaidFees;
```

`totalPendingFeesForAssets` calculates the `_pendingProtocolFee`:

```
_accruedFee(checkpoint.protocolFeeRate, _totalAssetsBeforeFees)
```

... and the `_pendingManagerFee`:

```
_accruedFee(managerFeeRate, _totalAssetsBeforeFees)
```

... based on the `totalAssets` value that includes the unpaid fee. That means fees are calculated on past fees if they cannot be paid out immediately.

Code corrected:

Unpaid fees are now deducted from the waterfall value. This means that unpaid fees do not contribute to the value on which the fees accrue.

6.6 Unpaid Fee Calculation

Correctness | Low | Version 5 | Code Corrected

CS-TFCarbon-020

If the protocol's `virtualTokenBalance` is lower than the amount of fees that have to be paid in a checkpoint update, the unpaid amount is stored to be paid for later soon as the system has some available tokens.

Unpaid fees are also added to each tranche's checkpoint. If unpaid fees have been added to a checkpoint and sometime later, the checkpoints are updated, the waterfall is calculated by applying the APY to the checkpoint (that contains the unpaid fee), effectively applying the APY to the fee. The saved unpaid fee is then subtracted from the calculated value. This waterfall value, in turn, is stored in the checkpoint by adding the unpaid fees again.

The fee is multiplied by the APY, then the fee is subtracted and then added again. This means, there is a slight gain added to the checkpoint in comparison to a checkpoint update without stored unpaid fees. This gain is ultimately taken from the equity tranche.

Code Corrected:

A new field (`unpaidFees`) was introduced in the checkpoint struct to store the fees. Thus fees do not contribute to interest accrual of the tranche.

6.7 Balance Underflow

Correctness | Low | Version 1 | Code Corrected

CS-TFCarbon-021

`StructuredPortfolio.fundLoan` calls `LoansManager._fundLoan` which in turn checks the contract's balance of the underlying token to determine how much principal is available for loan funding. An error message is returned when the requested principal exceeds the balance of the contract. However, `fundLoan` subtracts the amount of principal from the `virtualTokenBalance` afterward. If any amount of tokens has been directly transferred to the contract before, the first check could pass, while the subtraction fails, resulting in a revert without the given error message.

Code corrected:

The following check has been added which prints an informative message:

```
require(virtualTokenBalance >= principal, "SP: Principal exceeds balance");
```

6.8 Gas Optimizations

Design | Low | Version 1 | Code Corrected

CS-TFCarbon-012

The following parts of the code could potentially be optimized for gas efficiency:

- The `CustomFeeRate` struct used in a mapping in `ProtocolConfig` requires two storage slots. As the `feeRate` does not require 256 bits, the struct field could be reduced to a smaller type in order to shrink the size requirement of the struct to 1 storage slot. This should, however, be carefully handled.
- Redundant storage loads are performed in various places. Some examples are:
  - `StructuredPortfolio.initialize` loads the `tranches` variable to emit an event while it is already available in memory variables.
  - `StructuredPortfolio.fundLoan` checks that the status is `Live`, then calls `updateCheckpoints` which performs the same check again.
- Redundant external calls are performed in various places. Some examples are:
  - `StructuredPortfolio.start` calls `checkTranchesRatios` which gets the `totalAssets` of each tranche. It then calls `totalAssets`, which, in a status other than `Live`, gets the `totalAssets` of each tranche again.

- `StructuredPortfolio.calculateWaterfall` calculates the `calculateWaterfallWithoutFees`, then proceeds to call `totalPendingFees` on each tranche which in turn calls `StructuredPortfolio.calculateWaterfallForTrancheWithoutFee` twice.

- Redundant storage writes are performed in various places. Some examples are:
  - The loop in `StructuredPortfolio.start` adds values from each tranche to the `virtualTokenBalance` storage variable, resulting in multiple storage writes to the same variable.
  - `StructuredPortfolio.markLoanAsDefaulted` updates the checkpoints of the tranches and then immediately overwrites these checkpoints.
  - In `LoanManager._tryToExcludeLoan`, in case the last loan is deleted a redundant assignment is performed: `activeLoanIds[i] = activeLoanIds[loansLength - 1]`

- Complicated call paths are performed in various places. Some examples are:
  - `StructuredPortfolio.checkTranchesRatios` calls each tranche's `totalAssets` which (in `Live` status) call back to `StructuredPortfolio.calculateWaterfallForTrancheWithoutFee` which then calls back to the tranche's `getCheckpoint` function.

- `StructuredPortfolio._defaultedLoansDeficit` computes interest even when the deficit of the given checkpoint is 0.
- Some state variables are never updated and could be set to `immutable` in `StructuredPortfolio`:
  - `trancheImplementation`
  - `portfolioImplementation`
  - `protocolConfig`
- Cheap deployment has been chosen over cheap contract interactions. Gas for user interactions could be greatly reduced by sacrificing deployment costs of managers. For example, in a full deployment of `TrancheVault` (as opposed to a proxy deployment of a given implementation contract), 4 state variables that are used in many of the state-changing functions could be set to `immutable`.
- `TrancheVault.checkpoint` is declared public while a redundant getter `getCheckpoint` exists.
- In `Live` state, each vault transfers a chunk of fees from the portfolio to the beneficiary's addresses. As the funds are held in the portfolio anyways, one transfer in `StructuredPortfolio.updateCheckpoints` is sufficient. Even then, fee transfers on every interaction are unnecessarily costly.

Code corrected:

Most relevant code optimizations have either been implemented or chosen not to be implemented due to design choices. The gas consumption of some code parts has increased (e.g., `StructuredPortfolio.start`) slightly and some new inefficiencies have been introduced (e.g., `StructuredPortfolio.getTranchesData` has been replaced with `StructuredPortfolio.getTrancheData` which is a copy of the existing automatic getter).

6.9 Imprecise Specifications

Design | Low | Version 1 | Specification Changed

CS-TFCarbon-017

The following specifications are imprecise:

- The field `targetAPY` of struct `TrancheData` is not documented to be in basis points.
- The field `targetAPY` of struct `TrancheInitData` is not documented to be in basis points.
- `StructuredPortfolio.endDate` is commented to return the actual end date after the close. This is not true if the portfolio has been closed prematurely.
- `StructuredPortfolio.close` is commented to revert if any loans are still active. However, portfolios can always be closed after the end date.
- `StructuredPortfolio.calculateWaterfallForTranche` is commented to be only executable by the tranche with the given ID. Such restriction is however not enforced.
- `TrancheVault.updateCheckpointFromPortfolio` is commented to be only executable in `Live` state. However, there is no restriction preventing it from being executed in `Closed` state. Nevertheless, `StructuredPortfolio` is calling the function in `Live` state only.

Moreover in the extra documentation provided to us:

It is mentioned that deposits are blocked in `Closed` state and withdrawals are blocked in `CapitalFormation` state. However, this is never enforced in the code.

Specification changed:

The specification has been updated to correctly describe the code.

6.10 Missing Interface Functions

Design | Low | Version 1 | Code Corrected

CS-TFCarbon-016

Some interfaces are missing public functions that are implemented in the contracts. Here are two examples:

- `ITrancheVault` is missing the declaration for `setTransferController`.
- `ITrancheVault` is missing the declaration for `totalAssetsWithoutFees`.

Code corrected:

Both functions have been added to the interface. `totalAssetsWithoutFees` was renamed to `totalAssetsBeforeFees`.

6.11 Missing Sanity Check

Design | Low | Version 1 | Code Corrected

CS-TFCarbon-008

A sanity check is missing in `TrancheVault.deposit`. More specifically, the function does not prevent the parameter amount from being 0 while `mint` which implements similar functionality does.

Code corrected:

The missing sanity check has been added.

6.12 Shadowed Variable

Design Low Version 1 Code Corrected

CS-TFCarbon-019

The variable `fixedInterestOnlyLoans` in `StructuredPortfolio.initialize` is shadowed by a storage variable with the same name. While this does not impact the functionality of the given code, it could create problems during code maintenance.

Code corrected:

The name of the variable has been changed to `_fixedInterestOnlyLoans`.

6.13 Specification Nonconformity

Design Low Version 1 Specification Changed

CS-TFCarbon-011

The architecture specification states that

> They [vaults] do not store the capital, they only handle interactions between the investors and the portfolio.

This is in contrast to the fact that vaults store balances both in `CapitalFormation` and `Closed` states.

Specification changed:

The specification has been updated to the following:

> They store the capital when the portfolio is NOT in the Live state (so in the Capital Formation state and in the Closed state).

6.14 Unused Function

Design Low Version 1 Code Corrected

CS-TFCarbon-014

The internal function `TrancheVault._requirePortfolioOrManager` is never used inside of `TrancheVault`.

Code corrected:

The function has been removed.

6.15 Unused Return Value

Design | Low | Version 1 | Code Corrected

CS-TFCarbon-009

`TrancheVault._payProtocolFee` and `TrancheVault._payManagerFee` return the protocol fee and the manager fee paid respectively. However, this value is never used.

Code corrected:

The return values have been removed.

6.16 `configure` Function Does Not Include `TransferController`

Design | Low | Version 1 | Code Corrected

CS-TFCarbon-018

The function `TrancheVault.configure` can be used to simultaneously set multiple variables configurable by the portfolio manager. While `DepositController` and `WithdrawController` can be set, `TransferController` is not included in the function.

Code corrected:

The `TransferController` can now be set in the `TrancheVault.configure` function.

7 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.

7.1 Ambiguous Deficit Data in `Closed` State

Note Version 3

Deficit checkpoints are used to determine the potential interest of tokens that have been lost due to defaulted loans (which can still be repaid in some circumstances). In `Closed` state, no interest is accrued. Therefore, repayments of defaulted loans are not decreasing the deficit again. However, a call to `StructuredPortfolio.close` still updates the deficit checkpoints. This update works in a non-intuitive fashion and results in ambiguous data: If a portfolio with no defaulted loan is closed, the deficit will be 0. If, on the other hand, a portfolio with at least one defaulted loan is closed, all loan values (including the loans that have not defaulted) are counted towards the deficit.

While this is not problematic for Carbon itself, other contracts integrating with it might rely on consistent data.

7.2 Compounding Interest Computed in Arbitrary Intervals

Note Version 1

Every time the checkpoints are updated through a state-changing function, interest is compounded. Since `StructuredPortfolio.updateCheckpoints` can be called at any time during `Live` status, the results can be different. Consider the following example:

- A user deposits 100 tokens on a tranche with 10% APY.
- After 1 year of inactivity on the platform, they receive a yield of 10 tokens.
- If the user calls `updateCheckpoints` after 6 months, they receive a yield of 10.25 tokens after 1 year.
- If the user calls `updateCheckpoints` each month, they receive a yield of 10.47 tokens after one year.

The same situation produces different results based on how often the checkpoints are updated (without any other interactions).

7.3 Fee Accrual in `Closed` State

Note Version 1

Fees are still accrued in `Closed` state. TrueFi claims this is done to incentivize fast withdrawals. However, withdrawals in `Closed` state might still be delayed by late loan repayments.

7.4 Fee Accrual on Yield

Note Version 1

Fees are accrued on the principal plus the yield on each tranche. This means that the same percentage of yield and fees on a tranche results in negative growth. For example, a tranche with 100 tokens value, 2% yield and 2% fees will have 99.96 value after one year.

7.5 Manager Fee Accrual

Note Version 1

Manager fees are accrued on the `virtualTokenBalance` plus the value of the currently running loans. If a loan defaults and the manager marks the loan as defaulted on-chain, manager fees are no longer accrued on the loan. Therefore, managers are incentivized to not mark loans as defaulted until the portfolio is closed, resulting in bigger fees than necessary for lenders.

The fee accrual for non-defaulted loans can even go beyond the runtime of the loan.

7.6 Only the Recipient of a Loan Can Repay It

Note Version 1

The system allows only the recipients of a loan to repay it. This is enforced in the `LoanManager._repayFixedInterestOnlyLoan` by the following line:

```
require(fixedInterestOnlyLoans.recipient(loanId) == msg.sender, "LM: Not an instrument recipient");
```

Users losing their private keys will not be able to repay their loans in any way.

TrueFi claims that only KYC addresses should be able to repay loans. However, there is no mechanism implemented that deals with the aforementioned problem.

7.7 Skewed Interest Distribution

Note Version 1

Protocol and manager fees are calculated from the whole balances instead of the accrued interest. This potentially results in skewed interest rate distributions depending on the chosen fee rates per tranche. Consider the following example:

- The protocol fee is 1% and the manager fee is 1% on every tranche.
- The junior tranche has 5% interest and 100 tokens in value.
- The senior tranche has 3% interest and 100 tokens in value.
- Without fees (and disregarding compounding effects), the tranches will yield 5 and 3 tokens in yield respectively.
- With fees (and disregarding compounding effects), the tranches will yield 2.9 and 0.94 tokens respectively.

The ratio is now different after accounting for fees as `5 / 3 !== 2.9 / 0.94`.

7.8 Use of Non-standard ERC20 Tokens

Note Version 1

Managers (and users) should be aware that using a non-standard ERC20 token as the underlying can be dangerous for the system. Non-standard ERC20 tokens include but are not limited to the following behaviors:

Tokens With Reentrancies:

If a portfolio is set up with an underlying token that is reentrant (e.g., `ERC-777`), various possibilities of reentrancy attacks are enabled. Here is an example of one possible attack vector:

- A portfolio consists of 3 tranches and is in `Live` status with no active loans.
- Users have deposited 100 tokens to each tranche with no accrued interest (i.e. 1 share per token).
- An attacker calls `deposit` on the junior tranche with 99 tokens.
- In the `safeTransferFrom` call, the underlying token calls back to the attacker's contract.
- At this point, the checkpoint of the junior tranche has already been updated, while the `virtualTokenBalance` in the portfolio has not.
- The attacker now reenters into the deposit function if the equity tranche with a deposit of 100 tokens.
- 10,000 shares are minted to the attacker as the `virtualTokenBalance` is still 300, while the checkpoints of senior and junior tranches return a sum of 299 tokens, leaving only 1 token for the equity tranche waterfall value.
- After the call, the attacker now holds shares representing 99 tokens in the junior tranche and ~198 tokens in the equity tranche resulting in an instant profit of ~98 tokens.

Tokens With Fees:

When transferring tokens with fees, the receiver does not get the amount the sender sends but a part of it as fees are deducted. However, updating the `virtualTokenBalance` for example makes the assumption the whole amount has been received. Thus, repetitive transfers will create a discrepancy between the internal accounting of the portfolio which uses the `virtualTokenBalance` and the actual amount held by the portfolio.

Rebasing Tokens:

With rebasing tokens, the amount of tokens each account holds changes over time. This will lead, similarly to tokens with fees, to internal accounting being wrong.

Pausable Tokens:

When a token is paused, it might revert on every call to functions like `transfer`. As Carbon extensively uses transfers in many functions, the system could become unusable on such occasions.