Zellic

TrueFi Carbon

Smart Contract Security Assessment

January 18, 2022

Prepared for:

TrueFi Team

Prepared by:

Mark Griffin and Sampriti Panda

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please contact us at hello@zellic.io.

1 Executive Summary

Zellic conducted a security assessment for TrueFi from November 21st to December 9th, 2022. During this engagement, Zellic reviewed TrueFi Carbon's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.1 Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

- Does the structured portfolio allow investment during capital formation that gets appropriately paid out when the portfolio closes?
- Is the life cycle of loans properly tracked, and are the payments returned as value to the tranches?
- Could a malicious actor cause funds to be locked up or extracted from the portfolio or tranches?

1.2 Non-goals and Limitations

- Testing every combination of function and state between the tranches and structured portfolio
- Issues arising from contracts external to those listed in the scope of this audit
- Any financial or trust issues external to the blockchain
- Issues stemming from code or infrastructure outside of the assessment scope

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide. During this assessment, lack of line coverage prevented us from fully analyzing the effectiveness of existing unit tests on corner cases, and the duration of the audit did not support enough time to implement advanced simulation or fuzzing efforts.

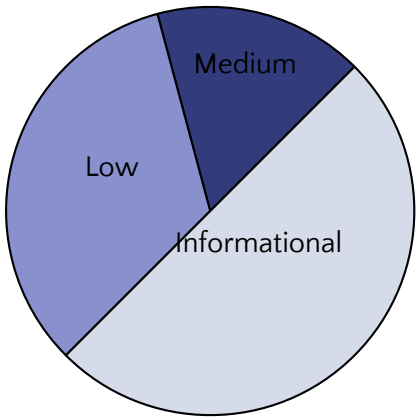
1.3 Results

During our assessment on the scoped TrueFi Carbon contracts, we discovered six findings. No critical issues were found. Of the six findings, one was of medium impact, two were of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for TrueFi's benefit in the Discussion section (4) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
Critical	0
High	0
Medium	1
Low	2
Informational	3



2 Introduction

2.1 About TrueFi Carbon

TrueFi Carbon is an uncollateralized lending platform implementing structured vaults.

Structured vaults are multi-vault portfolios allowing different types of investors to deposit into one bucket of funds managed by the portfolio manager while having different risk-return profiles separated by tranches.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review the contracts' external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the code base in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimiza-

tion, upgradeability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

TrueFi Carbon Contracts

Repository	https://github.com/trusttoken/contracts-carbon
Versions	`6a5daad4a75afb51b4ad6b313e2edf89634def43`
Programs	• StructuredPortfolio • StructuredPortfolioFactory • TrancheVault • ProtocolConfig • LoansManager • DepositController • WithdrawController
Type	Solidity
Platform	EVM-compatible

2.4 Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of two person-weeks. The assessment was conducted over the course of three

calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Chad McDonald, Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Mark Griffin, Engineer
mark@zellic.io

Sampriti Panda, Engineer
sampriti@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

November 21, 2022	Kick-off call
November 21, 2022	Start of primary review period
December 9, 2022	End of primary review period

3 Detailed Findings

3.1 Manager centralization risk

- **Target**: StructuredPortfolio.sol, TrancheVault.sol
- **Category**: Business Logic
- **Severity**: High
- **Likelihood**: Low
- **Impact**: Medium

Description

The manager of structured portfolios is given explicit control over almost all functionality of the portfolio, with very few safeguards in the contracts to safeguard against mistakes or outright abuse. The manager role must be trusted, but it is the most likely threat for attackers to compromise and should be guarded with the highest security possible. While it is impossible to wholly eliminate the centralization risk of the manager in the current design, imposing limits or removing some functionality that could affect movement of funds is appropriate.

Impact

A compromised portfolio manager transfers all of the liquid value from the structured portfolio as well as siphons all deposits and withdrawals.

Recommendations

There are a number of ways a compromised manager could remove funds from the portfolio, each with its own recommendation:

- Offer and fund loans of unlimited amounts: A maximum loan limit in terms of percentage or absolute value increases the number of steps a compromised manager would have to take to fund bogus loans for the full liquid value of the portfolio.
- Set `withdrawController` and `depositController` to arbitrary contracts: In the current design, the portfolio manager is granted `TRANCHE_CONTROLLER_OWNER_ROLE`, which allows them to change the controller contracts to any address. While allowing managers to change these is intended, we believe the risk of breaking the functionality of the portfolio outweighs any benefit gained from this functionality. We recommend removing this role and the ability to change controllers; if the contracts need to be updated, they should be updated via proxy implementation upgrades by the protocol.

- Set unbounded `depositFeeRate` and `withdrawFeeRate` on the respective controllers: The manager address for the `withdrawController` and `depositController` is part of the arguments to `StructuredPortfolioFactory.createPortfolio`, presuming it is the same manager as the portfolio, which would have the ability to set unbounded values (even over 100%) for deposit or withdrawal fees. This would allow the manager to siphon all funds being deposited and all of the liquid value of the portfolio via a withdrawal fee. We recommend that the fees be capped at a reasonable percentage in the controller contract's initialization and fee update functions as well as removing the ability to update controller contracts.
- Set unbounded value for `TrancheVault.managerFeeRate` via `setManagerFeeRate`: Similar to above, the manager could siphon portfolio funds by setting an unbounded rate value and triggering a checkpoint update. We recommend checking `managerFeeRate` against a maximum in `initialize` and `setManagerFeeRate` functions.

Remediation

The issue has been acknowledged by the TrueFi team. Their official reply is reproduced below:

> Manager needs to be trusted and whitelisted by TrueFi governance. We acknowledge that there is centralization risk, but managers will always have a lot of power in uncollateralized lending.

3.2 Lack of input validation on address arguments

- **Target**: DepositController, WithdrawController, ProtocolConfig, StructuredPortfolio, TrancheVault
- **Category**: Business Logic
- **Severity**: Low
- **Likelihood**: Low
- **Impact**: Low

Description

Many functions that take addresses as input parameters do not implement checks on addresses.

All addresses that can be sent as inputs, either directly or as members of structs, should be checked to ensure they are not zero.

Impact

Improper values during deployment could result in a botched deployment, costing time, effort, and gas to redeploy.

An improper argument to a reconfiguration function could cause reverts in key functions, denying use of the system until the error is fixed.

Recommendations

Some functions already include implicit zero checks due to ERC-20 template contracts, but we recommend ensuring the following functions will revert if an address parameter is zero, and consider implementing allowlists where appropriate.

- `DepositController.initialize`
- `DepositController.setLenderVerifier`
- `DepositController.configure`
- `WithdrawController.initialize`
- `ProtocolConfig.initialize`
- `ProtocolConfig.SetProtocolAdmin`
- `ProtocolConfig.SetProtocolTreasury`
- `ProtocolConfig.SetPauserAddress`
- `StructuredPortfolio.initialize`
- `StructuredPortfolioFactory.constructor`
- `StructuredPortfolioFactory.createPortfolio`
- `TrancheVault.initialize`
- `TrancheVault.configure`

- `TrancheVault.setDepositController`
- `TrancheVault.setWithdrawController`
- `TrancheVault.setTransferController`
- `TrancheVault.setManagerFeeBeneficiary`

Remediation

The TrueFi team has addressed the issue in commit e40945 by implementing address validation checks in some of the above functions where a lack of input validation was most likely to be an issue.

3.3 Insufficient token check

- **Target**: StructuredPortfolio
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

The function `StructuredPortfolio.initialize` only requires `underlyingToken.decimals() == tranchesInitData[i].tranche.decimals()` instead of checking the underlying token contract addresses.

In the case that `StructuredPortfolioFactory.createPortfolio` is used, `underlyingToken` will be the same as `TrancheVault.token` for each tranche, in which case the check is unnecessary.

Impact

The same underlying ERC-20 token should be used between the portfolio and the tranches, otherwise the token transfers between the tranches and portfolio would be unequal in value and payouts during different stages of the portfolio would also be unequal.

Recommendations

We recommend this check be changed to check token addresses are the same between the tranches and the portfolio, otherwise it should be omitted.

Remediation

The issue has been acknowledged by the TrueFi team. Their official reply is reproduced below:

> All portfolios need to be created through the portfolios factory to be displayed on the TrueFi app.

3.4 Unused function

- **Target**: StructuredPortfolio, TrancheVault
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The function `TrancheVault._requirePortfolioOrManager` is never used or referenced.

Impact

Unused variables and functions expend unnecessary gas during deployment and go against development best practices.

Recommendations

We recommend removing the function `TrancheVault._requirePortfolioOrManager`.

Remediation

This issue has been fixed by the TrueFi team in commit c94617 by removing the function.

3.5 Shadowed state variable name

- **Target**: StructuredPortfolio
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The function `StructuredPortfolio.initialize` has a parameter named `fixedInterestOnlyLoans` that shadows a state variable with the same name inherited from `LoansManager`.

Impact

Shadowed variable names go against best practice and can cause confusion, but in this case, there is no direct impact.

Recommendations

We recommend renaming the parameter to `_fixedInterestOnlyLoans`.

Remediation

This issue has been fixed by the TrueFi team in commit c94617 by renaming the variable.

3.6 Protocol fee is unbounded

- **Target**: ProtocolConfig
- **Category**: Business Logic
- **Likelihood**: N/A
- **Severity**: Informational
- **Impact**: Informational

Description

The `ProtocolConfig` contract allows the setting of both default and custom protocol fee rates with no upper limits.

Impact

While the protocol is trusted with the management of the system, if the protocol management were to be compromised even temporarily, unbounded fee rates could enable an attacker to siphon a large amount of value from all structured portfolios.

Indicating a maximum rate also communicates intent to users, even though proxy contracts could enable changes to the actual implementation of the system.

Recommendations

We recommend enforcing a maximum protocol fee rate in `initialize`, `setCustomProtocolFeeRate`, and `setDefaultProtocolFeeRate`.

Remediation

The issue has been acknowledged by the TrueFi team. Their official reply is reproduced below:

> TrueFi Dao is the owner of protocol config so the only way to change protocol fee is through dao voting.

4 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

4.1 Returns from structured vaults predicated on manager performance and loan repayment

The structured portfolio contains target APY values and an expected equity rate; these values are only used to calculate expected values and as a desired end state, respectively.

The actual value (and APY) is decoupled from the loans issued; there is no mechanism to guarantee that returns match any expectations.

The only time tranche ratios are checked are before the portfolio goes live; it is the responsibility of the portfolio manager to issue enough loans that deliver appropriate interest rates in order to achieve the desired return and there is no guarantee or enforcement that this occurs.

Indeed, the waterfall payout system only guarantees that the highest-index tranches get paid out first according to their own APY while the lower-index tranches could receive less than projected (as low as zero payout).

That being said, if the returns from the loans exceed the expected APYs of the higher tranches, the lowest tranche will receive all returns beyond projected rates.

Enforcement of loans is also not contained within the system; the grace period on repayment is only used to ensure that the manager cannot mark a loan as defaulted before the grace period has elapsed after an expected payment, and loans are only marked defaulted by deliberate action of the manager.

While the contracts contain enough functionality to determine the current performance of a portfolio and estimate returns, additional logic would need to be implemented to provide data to users in an effective and transparent fashion (such as via a website or API frontend).

This discussion item is not a judgment about the design or implementation, merely a remark on the risks inherent to uncollateralized lending and the limitations of blockchain technology in isolation.

4.2 Line coverage not implemented for unit tests

While TrueFi Carbon implements a reasonable suite of unit tests, a lack of line coverage information makes it harder to determine the effectiveness of these tests and increases the probability that problems exist that have escaped our notice.

In general, untested code is more likely to have bugs or diverge from expected behavior, and programmer cognitive bias tends to leave unexpected input handling or exceptional behavior untested.

This is especially true of error handling or corner-case functionality; without tests those parts of code may fail in unexpected ways, and without line coverage it is difficult to determine where such issues may lay dormant.

Most of the functions in the TrueFi Carbon contracts have very few branches, but there are still several functions with multiple returns and conditionals within loops that handle corner-case functionality, such as handling cases where the equity tranche's value is zero and deficits spill into higher tranches.

We highly recommend that unit tests be implemented for all such corner case handling and the coverage be verified by line coverage.

While 100% line and branch coverage do not guarantee code has no bugs, we also recommend implementing a goal of 100% line coverage for all of smart contracts included in this audit.

4.3 No guards against reentrancy

While there are no attacks leveraging reentrancy that we identified during this audit, there are no protections against reentrancy implemented.

All of the external calls in the codebase are between contracts that are specified during initialization, and in this way they are implicitly trusted.

If any of the contracts are changed, multiple reentrancy attack vectors could arise.

This either requires an attacker to have privileges beyond what is expected (such as being able to change an implementation for a proxy contract or one of the `address`-type state variables as with `setDepositController`) or could occur due to a mistake during deployment or if the contracts are upgraded in the future.

We recommend considering the implementing of modifiers such as the standard `nonReentrant` modifier on key state-changing functions such as those related to checkpoints, loans, deposits, or withdrawals in `TrancheVault` and `StructuredPortfolio` contracts.

4.4 Possible front-running of defaulted loans

The deposit and withdraw controllers include functions (`setWithdrawAllowed` and `setDepositAllowed`) which the manager can set that allow users to deposit and withdraw funds while the portfolio is live. When depositors withdraw their portion of the staked amount from the portfolio, the amount is calculated using a hypothetical value of the portfolio, which includes both the value of liquid assets in the portfolio and the illiquid value of loans that are currently being repaid.

In the case that a loan has defaulted and the borrower is unable to pay back the entire principal, the depositors of the portfolio absorb the losses according to the waterfall calculation. However, a user who notices that a manager is about to mark a high value loan as defaulted may front-run the operation by withdrawing their funds before the loan is marked as defaulted. This allows the depositor to extract an inflated value of their deposited shares while further reducing the value of the shares of the other depositors, including those who are part of a less risky tranche.

While this issue is inherent in the protocol's design, we suggest mitigating its impact by disabling withdrawals during later stages of the portfolio or by having the portfolio manager pause withdrawals before marking loans as defaulted. This can help prevent any potential negative effects on users.

5 Threat Model

This provides a full threat model description for various functions. As time permitted, we analyzed each function in the smart contracts and created a written threat model for some critical functions. A threat model documents a given function's externally controllable inputs and how an attacker could leverage each input to cause harm. Not all functions in the audit scope may have been modeled. The absence of a threat model in this section does not necessarily suggest that a function is safe.

5.1 File: `StructuredPortfolio.sol`

Function: `initialize`

Intended behavior

Initialization function designed to be called during construction via StructuredPortfolioFactory using the `initializer` modifier.

Initializes all of the primary state variables for the portfolio except `endDate` (which is set in `start`).

Each of the tranches has their portfolio set to the current contract and is approved to transfer `type(uint256).max` of the underlying token on behalf of the portfolio.

Preconditions

- Precondition: Contracts at each of the addresses supplied as inputs must be deployed
- Precondition: Tranches must be deployed and initialized, such as when being called as part of `StructuredPortfolioFactory.createPortfolio`.

Inputs

- `manager`: Address on which MANAGER_ROLE is granted
  - **Control**: No checks in this function
  - **Authorization**: No checks in this function, but if called from `StructuredPortfolioFactory.createPortfolio`, `msg.sender` will be used as `manager` parameter and it must have `WHITELISTED_MANAGER_ROLE`

  - **Impact**: This address has administrative control over the loans and starting the portfolio. When the tranches and portfolio are created with `StructuredPortfolioFactory.createPortfolio`, this address will also be given the `MANAGER_ROLE` and `TRANCHE_CONTROLLER_OWNER_ROLE` for the associated tranches.
- `underlyingToken`: Address of ERC20 token used by portfolio
  - **Control**: The `decimals` value is checked against the `tranche.decimals` for each entry in `tranchesInitData`. This check is redundant if the `StructuredPortfolioFactory` is used, since the same underlying token should be used for both the tranches and portfolio. A more appropriate check would be to ensure that the underlying token is the same between them if that is what is intended.
  - **Authorization**: No checks
  - **Impact**: This token is critically import as it underpins all value transfer for the portfolio.
- `fixedInterestOnlyLoans`: Address of FixedInterestOnlyLoans contract
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: This contract contains the loan implementation and is critically import to loan functions.
- `_protocolConfig`: Address of ProtocolConfig contract
  - **Control**: No checks
  - **Authorization**: No checks, this contract address is a state variable passed in from `StructuredPortfolioFactory` if that contract is used.
  - **Impact**: This contract is sets and receives protocol fees and can change the fee rate for a given portfolio at any time. It also defines the addresses that will receive the `DEFAULT_ADMIN_ROLE` and `PAUSER_ROLE` for the portfolio, which gives them full control over the portfolio contract.
- `portfolioParams`: Parameters to configure portfolio
  - **Control**: `portfolioParams.duration` is checked to be > 0.
  - **Authorization**: No checks
  - **Impact**: These parameters control duration of the capital formation and live periods as well as the minimum deposits required to start the portfolio.
- `tranchesInitData`: Parameters to configure tranches
  - **Control**: Each elements `tranche.decimals` must match the underlying token's decimals. The first element's `targetApy` and `minSubordinateRatio` cannot be 0. The length of the array is not checked.
  - **Authorization**: No checks, implicitly trusted as initializer.
  - **Impact**: This array's members contains the addresses of the tranches and is critical in terms of the functioning of the portfolio.
- `_expectedEquityRate`: APY range that is expected to be reached by Equity tranche

- **Control**: No checks
- **Authorization**: No checks
- **Impact**: This parameter is assigned to the state variable `expectedEquityRate`, which appears to be unused.

Function call analysis

Multiple external calls are made based on addresses from parameters; if any of these revert, the portfolio will not deploy.

The deployer is trusted to supply appropriate parameters, some of which are not checked and are used for external calls:

- `protocolConfig.protocolAdmin()`
- `protocolConfig.pauserAddress()`
- `underlyingToken.decimals()`
- `tranchesInitData[i].tranche.decimals()`
- `initData.tranche.setPortfolio(this)`
- `underlyingToken.safeApprove(address(initData.tranche), type(uint256).max)`

Function: `updateCheckpoints`

Intended behavior

This function calculates the total assets and updates the checkpoints for each of the tranches with that value.

Branches and code coverage

When line coverage is implemented, confirm coverage for all three possible return statements in `calculateWaterfallWithoutFees` and both return statements in `_calculateLoanValue`, which are both reachable from this function

Preconditions

- Precondition: portfolio has to be in live status

Inputs

No inputs

Function call analysis

- `calculateWaterfall()`
  - **What is controllable?** No parameters
  - **If return value controllable, how is it used and how can it go wrong?** The return value will be used to update the checkpoint for each of the tranches, which records the current state of their assets and will be used for asset calculations going forward.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function is key for determining current value of the tranches and if it reverts will break a large part of the functionality of the portfolio, but it only interacts with core contracts established during initialization.
- `tranches[i].updateCheckpointFromPortfolio(_totalAssets[i])`
  - **What is controllable?** `_totalAssets[i]` is calculated via `calculateWaterfall()` directly above
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function should not revert if the core contracts are configured correctly, and only uses external calls with core contracts (portfolio, trancheVaults, protocol, token, findexInterestOnlyLoans)

Function: `increaseVirtualTokenBalance`

Intended behavior

Increments the virtual token balance for the calling tranche. Reverts if `msg.sender` is not one of the associated tranches, or if the value provided is greater than `type(int256).max`.

Preconditions

Precondition: the calling tranche must be relying on the portfolio to track its `virtualTokenBalance` in the current state.

Inputs

- `increment`: Amount to add to `virtualTokenBalance`
  - **Control**: `SafeCast.toInt256` ensures the value is less than `type(int256).max`
  - **Authorization**: `_changeVirtualTokenBalance` checks that `msg.sender` is one of the associated tranches and reverts if it is not
  - **Impact**: An erroneous value here could break the bookkeeping for one of the tranches and result in the portfolio tracking its token balance to be higher or lower than it should be.

Function call analysis

- `SafeCast.toInt256(increment)`
  - **What is controllable?** `increment`
  - **If return value controllable, how is it used and how can it go wrong?** The value returned is the same as the value passed in, just cast as an int256. The function reverts if the value is not between 0 and `type(int256).max`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are intended, no other control flow possible.
- `_changeVirtualTokenBalance(SafeCast.toInt256(increment))`
  - **What is controllable?** `increment`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function reverts if `msg.sender` is not one of the associated tranches.

Function: `decreaseVirtualTokenBalance`

Intended behavior

Decrements the virtual token balance for the calling tranche. Reverts if `msg.sender` is not one of the associated tranches or if the value provided is greater than `type(int256).max`.

Preconditions

Precondition: the calling tranche must be relying on the portfolio to track its `virtualTokenBalance` in the current state.

Inputs

- `decrement`: Amount to subtract from `virtualTokenBalance`
  - **Control**: `SafeCast.toInt256` ensures the value is less than `type(int256).max`
  - **Authorization**: `_changeVirtualTokenBalance` checks that `msg.sender` is one of the associated tranches and reverts if it is not
  - **Impact**: An erroneous value here could break the bookkeeping for one of the tranches and result in the portfolio tracking its token balance to be higher or lower than it should be.

Function call analysis

- `SafeCast.toInt256(decrement)`
  - **What is controllable?** `decrement`
  - **If return value controllable, how is it used and how can it go wrong?** The value returned is the same as the value passed in, just cast as an int256. The function reverts if the value is not between 0 and `type(int256).max`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Reverts are intended, no other control flow possible.
- `_changeVirtualTokenBalance(-SafeCast.toInt256(decrement))`
  - **What is controllable?** `decrement`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function reverts if `msg.sender` is not one of the associated tranches.

Function: `start`

Intended behavior

This function transitions the portfolio from Capital Formation status to live status.

It uses `_requireManagerRole()` to enforce access control and requires that the current status is `Status.CapitalFormation`.

Branches and code coverage

When line coverage is implemented, confirm that functionality of `_checkTranchesRatios` both reverts and works as correctly.

Preconditions

- Precondition: Portfolio must be in capital formation status and its `totalAssets` must be greater than or equal to `minimumSize`.

Inputs

No inputs.

Function call analysis

- `checkTranchesRatios()`
  - **What is controllable?** No parameters
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function reverts if the ratio of subordinate tranches' values do not meet the `minSubordinateRatio` established upon initialization.
- `totalAssets()`
  - **What is controllable?** No parameters
  - **If return value controllable, how is it used and how can it go wrong?** The return value is the sum of all of the tranches total assets (just underlying token value during capital formation), and it is checked against the `minimumSize` state variable of the portfolio.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It only calls core contract functionality, but if the core contracts malfunction the portfolio will remain in capital formation status.
- `_changePortfolioStatus(Status.Live)`
  - **What is controllable?** The parameter is hardcoded.
  - **If return value controllable, how is it used and how can it go wrong?** No return value.
  - **What happens if it reverts, reenters, or does other unusual control flow?** It only sets a state variable and emits an event regarding the state changing.
- `tranches[i].onPortfolioStart()`
  - **What is controllable?** No parameters.
  - **If return value controllable, how is it used and how can it go wrong?** The value returned is added to the portfolio's `virtualTokenBalance`, so incorrect values would result in a discrepancy between the portfolio's tracked balance and its actual balance.

- **What happens if it reverts, reenters, or does other unusual control flow?** This function only makes external calls on core contract function and so should not revert, but reverts would stop the portfolio from starting and keep it in capital formation.

Function: `close`

Intended behavior

This function moves the portfolio into closed status and enables withdrawals from tranches.

This function reverts if the portfolio is already closed. This function also requires that current timestamp is passed the portfolio end date (which is set in `start`) or that there are no active loans (which can only be added during `live`).

If the portfolio is in capital formation status, it can only be closed by the manager or if the start deadline has passed.

If the portfolio is in the live status, it can only be closed by the manager or if the end date has passed.

Branches and code coverage

When line coverage is implemented, confirm that each of the possible combination of successful traversals of the `require` statements in `close` are tested.

Preconditions

- Precondition: tranches and portfolio must be in a valid state

Inputs

No inputs

Function call analysis

- `_closeTranches()`
  - **What is controllable?** No parameters

  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function only uses external calls to core contracts, reverts would break the ability to close the portfolio
- `_changePortfolioStatus(Status.Closed)`
  - **What is controllable?** Hardcoded parameter
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function just updates a state variable and emits an event.
- `tranches[i].updateCheckpoint()`
  - **What is controllable?** No parameters
  - **If return value controllable, how is it used and how can it go wrong?** No return Value
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function should not revert if the core contracts are configured correctly, and only uses external calls with core contracts (portfolio, trancheVaults, protocol, token, findexInterestOnlyLoans)

Function: `addLoan`

Intended behavior

This function adds the loan specified by the struct passed as the single argument.

Access control is controlled by `_requireManagerRole` and the function checks that the portfolio is in live status.

Preconditions

- Precondition: Portfolio must be in live status

Inputs

- `params`: AddLoanParams struct containing the parameters of the loan
  - **Control**: `FixedInterestOnlyLoans.issueLoans` checks that the calculated total duration and interest are greater than 0, but does not check the `principal`, `gracePeriod`, or `canBeRepaidAfterDefault` members.
  - **Authorization**: Uses `_requireManagerRole`

- **Impact**: The portfolio can offer any loan they want, but the recipient must accept it before the loan can be funded.

Function call analysis

- `_addLoan`
  - **What is controllable?** `params`
  - **If return value controllable, how is it used and how can it go wrong?** No return value, if the loan is valid and `fixedInterestOnlyLoans.issueLoan` does not revert, it will mint a loan NFT and return the ID which `_addLoan` denotes in `issuedLoanIds[loanId]`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Invalid loan parameters (described above) must cause reverts, but valid loans must not, otherwise the loan functionality of the portfolio would be broken.

Function: `fundLoan`

Intended behavior

Starts a loan with given id and transfers assets to loan recipient.

Uses the `_requireManagerRole` function to limit access to the portfolio manager and requires the portfolio's is in live status.

- LoansManager._fundLoan
  - fixedInterestOnlyLoans.principal(loanId)...
  - fixedInterestOnlyLoans.start(loanId)...
  - activeLoanIds.push(loanId)
  - fixedInterestOnlyLoans.recipient(loanId)...
  - asset.safeTransfer(borrower, principal)

Preconditions

- Precondition: a loan NFT with the specified loanId must have been added.
- Precondition: the recipient of the loan NFT with the specified loanId must have already accepted the loan.

Inputs

- `loanId`: loanId Id of a loan that should be started
  - **Control**: `LoansManager._fundLoan` checks that `issuedLoanIds[loanId]` is true
  - **Authorization**: The portfolio manager is the only address that is allowed to call this
  - **Impact**: The manager can fund any loan that has been added and then accepted by the recipient address as long as the portfolio has enough of the underlying `asset` token to fund it.

Function call analysis

- `_fundLoan(loanId)`
  - **What is controllable?** `loanId`
  - **If return value controllable, how is it used and how can it go wrong?** The return value is the amount of the underlying token sent to the recipient, and must be correct or a discrepancy between the `virtualTokenBalance` and actual token balance would occur.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Unexpected reverts would keep loan recipients from accessing funding. The only external calls are to `fixedInterestOnlyLoans` and `asset` contracts.

Function: `repayLoan`

Intended behavior

Allows sender to repay a loan with given ID while the portfolio is in either closed or live status.

The amount to be paid is dynamically calculated based on the loan parameters and the transaction will revert if the sender cannot transfer the required amount of underlying asset.

Branches and code coverage

When line coverage is implemented, confirm that all return statements, continue statements, and if-else branches, and possible reverts in both `_repayLoanInLive` and `_repayLoanInClosed` are covered.

Preconditions

- Precondition: Given loanId must refer to a loan owned by the portfolio that has been added, accepted, and funded.

Inputs

- `loanId`: ID of a loan that should be repaid
  - **Control**: `LoansManager._repayFixedInterestOnlyLoan` requires that `issuedLoanIds[loanId]` is true
  - **Authorization**: `LoansManager._repayFixedInterestOnlyLoan` requires `fixedInterestOnlyLoans.recipient(loanId) == msg.sender`.
  - **Impact**: Recipients must be able to repay their loans and only their loans.

Function call analysis

- `_repayLoanInLive(loanId)`
  - **What is controllable?** `loanId`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only core contracts are used for external calls; reverts are expected in the case of invalid loans being specified but otherwise reverts would be breaking repayment along with other large swaths of functionality.
- `_repayLoanInClosed(loanId)`
  - **What is controllable?** `loanId`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only core contracts are used for external calls; reverts are expected in the case of invalid loans being specified but otherwise reverts would be breaking repayment along with other large swaths of functionality.

Function: `updateLoanGracePeriod`

Intended behavior

Updates the grace period for the specified loan.

Uses `_requireManagerRole` to enforce access control.

Preconditions

- Precondition: loanId must exist and be owned by the portfolio, and be started.

Inputs

- `loanId`: loanId Id of a loan that should be updated
  - **Control**: `FixedInterestOnlyLoans.updateInstrument` requires the loan status be Started
  - **Authorization**: `_requireManagerRole` restricts calling this function to the portfolio manager. `FixedInterestOnlyLoans.updateInstrument` uses `onlyLoanOwner(instrumentId)` to restrict loan IDs to only those owned by the portfolio contract.
  - **Impact**: Managers can only update loans owned by portfolios they manage.
- `newGracePeriod`: The new grace period
  - **Control**: The new grace period must be greater than the existing grace period value.
  - **Authorization**: `_requireManagerRole` restricts calling this function to the portfolio manager. `FixedInterestOnlyLoans.updateInstrument` uses `onlyLoanOwner(instrumentId)` to restrict loan IDs to only those owned by the portfolio contract.
  - **Impact**: Managers can update the grace period to be any `uint32` value greater than the current one.

Function call analysis

- `_updateLoanGracePeriod`
  - **What is controllable?** `loanId` and `newGracePeriod`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only external call is to `FixedInterestOnlyLoans.updateInstrument`, no real risk of revert beyond invalid loan ID or grace period being specified.

Function: `cancelLoan`

Intended behavior

Cancels the specified loan as long as it hasn't been funded yet.

Uses `_requireManagerRole` for access control.

Preconditions

- Precondition: loanId must exist and be owned by the portfolio, and not yet be funded

Inputs

- `loanId`: loanId Id of a loan that should be canceled
  - **Control**: `FixedInterestOnlyLoans.cancel` requires the loan status be Created or Accepted
  - **Authorization**: `_requireManagerRole` restricts calling this function to the portfolio manager. `FixedInterestOnlyLoans.cancel` uses `onlyLoanOwner(instrumentId)` to restrict loan IDs to only those owned by the portfolio contract.
  - **Impact**: Managers can only cancel the loans owned by portfolios they manage.

Function call analysis

- `_cancelLoan(loanId)`
  - **What is controllable?** `loanId`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only external call is to `FixedInterestOnlyLoans.cancel`, no real risk of revert beyond invalid loan ID being specified.

Function: `markLoanAsDefaulted`

Intended behavior

Sets the status of a loan with given ID to Defaulted and excludes it from active loans array.

Uses `_requireManagerRole` for access control.

Branches and code coverage

When line coverage is implemented, confirm coverage for different combinations of loans, deficit, and tranche states.

Preconditions

- Precondition: loanId must exist, be started, and be owned by the portfolio

Inputs

- `loanId`: loanId Id of a loan that should be marked as defaulted
  - **Control**: `FixedInterestOnlyLoans.markAsDefaulted` uses `onlyLoanStatus(instrumentId, FixedInterestOnlyLoansStarts)` to require the loan to exist and to be started.
  - **Authorization**: `_requireManagerRole` restricts calling this function to the portfolio manager. `FixedInterestOnlyLoans.markAsDefaulted` uses `onlyLoanOwner(instrumentId)` to restrict loan IDs to only those owned by the portfolio contract.
  - **Impact**: Managers can only mark default the loans owned by portfolios they manage.

Function call analysis

- `updateCheckpoints`
  - **What is controllable?** No inputs
  - **If return value controllable, how is it used and how can it go wrong?** The values for each of the tranches is returned, it must be accurate or else the loan deficit bookkeeping will be affected and the portfolio value tracking will be incorrect.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only uses core contracts; reverts will break marking default as well as most other functionality
- `_markLoanAsDefaulted(loanId)`
  - **What is controllable?** `loanId`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only uses core contracts for external calls; reverts must occur on incorrect

loanIds.

- `_limitedBlockTimestamp`
  - **What is controllable?** No inputs
  - **If return value controllable, how is it used and how can it go wrong?** The timestamp will be the minimum of `block.timestamp` and the portfolio end date
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `SafeCast.toInt256(waterfallBeforeDefault[i] - waterfallAfterDefault[i])`
  - **What is controllable?** the values in the subtraction are affected by the loan that was just defaulted. Since the only change between checkpoints is that the specified loan's value is not counted in `waterfallAfterDefault[i]`, the result of the subtraction will be greater than or equal to 0.
  - **If return value controllable, how is it used and how can it go wrong?** The return value is used as a signed value to adjust the deficit value. If the returned value is incorrect, it would cause the portfolio's value tracking to also become incorrect.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Only reverts if the value provided would be problematic.
- `_updateDefaultedLoansDeficit(i, limitedBlockTimestamp, deficitIncrease)`
  - **What is controllable?** deficitIncrease is affected by the loan just marked as default.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** No external calls or real risk of reverting.

5.2 File: `StructuredPortfolioFactory.sol`

Function: `constructor`

Intended behavior

Construct a factory contract to make new StructuredPortfolios.

Whoever calls constructor (msg.sender) becomes ADMIN

Preconditions

- Precondition: Portfolio implementation already is deployed

- Precondition: Tranche implementation already is deployed
- Precondition: address with protocol implementation already is deployed

Inputs

- `_portfolioImplementation`: address of portfolio implementation
  - **Control**: No sanity checks
  - **Authorization**: No auth checks
  - **Impact**: Unexpected or misconfigured factory can be deployed
- `_trancheImplementation`: address of tranche implementation
  - **Control**: No sanity checks
  - **Authorization**: No auth checks
  - **Impact**: Unexpected or misconfigured factory can be deployed
- `_protocolConfig`: address of IProtocolConfig implementation
  - **Control**: No sanity checks
  - **Authorization**: No auth checks
  - **Impact**: Unexpected or misconfigured factory can be deployed

Function: `createPortfolio`

Intended behavior

Creates tranche vaults and then portfolio. Stores new portfolio in `portfolios` array.

msg.sender is checked, trusts sender to use good inputs.

Preconditions

- Precondition: portfolio implementation deployed

Inputs

- `underlyingToken`: ERC-20 token underlying
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Deploy Portfolio and tranches with malicious underlying token
- `fixedInterestOnlyLoans`: Loan NFT implementation
  - **Control**: No checks

  - **Authorization**: No checks
  - **Impact**: Deploy Portfolio with malicious NFT implementation
- `portfolioParams`: Params for portfolio deployment
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Deploy Portfolio with nonsense params
- `tranchesData`: data use to deploy tranche vaults for portfolio
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Bad input to `_deployTranches`
- `expectedEquityRate`: parameter for portfolio
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Deploy Portfolio with nonsense params

Function call analysis

Member variable `portfolioImplementation` has its `initialize` function called with unchecked parameters.

- `portfolioImplementation.initialize`
  - **What is controllable?** All calldata (except msg.sender and protocolConfig)
  - **If return value controllable, how is it used and how can it go wrong?** Malicious portfolio could be deployed.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Malicious or nonfunctional portfolio could be deployed.

All of the tranche-related initialization is done in `_deployTranches`. Details described separately below, but the primary point is that the caller of this function must be fully trusted because they could deploy unexpected or malicious tranches.

Function: `_deployTranches`

Intended behavior

Deploys all tranche vaults for a portfolio

Preconditions

- Precondition: Presumes deposit/withdraw/controller proxies/implementations already deployed
- Precondition: Presumes tranche implementation already deployed

Inputs

- `underlyingToken`: ERC-20 token underlying
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Deploy Portfolio and tranches with malicious underlying token
- `tranchesData`: Structure defining the tranche implementation
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: Arbitrary external calls via controllers and their init data, as well as deploying malicious tranche vault

Function call analysis

- `trancheData.[deposit|withdraw|transfer]Controller` have `trancheData.[deposit|withdraw|transfer]ControllerInitData` abi-encoded calls performed via the cloned controller's `.functionCall` method.
  - **What is controllable?** Implementation address and all params
  - **If return value controllable, how is it used and how can it go wrong?** Return values aren't checked
  - **What happens if it reverts, reenters, or does other unusual control flow?** Malicious or nonfunctional tranches or controllers could be deployed.

5.3 File: `TrancheVault.sol`

Function: `initialize`

Intended behavior

External initialize function with `initializer` modifier. Initializes the ERC-20, proxy, and access control states. Also sets all of the major state variables except `portfolio` and uint256 variables that appropriately default to zero.

The `manager` address gets both `TRANCHE_CONTROLLER_OWNER_ROLE` and `MANAGER_ROLE`, and is also set as the `managerFeeBeneficiary`.

Preconditions

- Precondition: Key functionality contracts (those that are stored in state variables such as `token`, `depositController`, `protocolConfig`, `portfolio`, etc) must be deployed and properly initialized.

Inputs

None of the inputs are checked, each is trusted to be a valid and appropriate address, contract, or uint256 value.

Function call analysis

No function calls other than functions inherited from OpenZeppelin standard contracts.

Function: `deposit`

Intended behavior

Deposits a given amount and transfers shares to the specified address.

Branches and code coverage

Calling the `deposit` function reaches a large number of functions and can cover many different states.

When line coverage is implemented, ensure the state space of the deposit functionality is fully covered.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `amount`: uint256 value of underlying asset to deposit
  - **Control**: Must be less than or equal to 'maxDeposit(msg.sender), is allowed to be 0.
  - **Authorization**: Must be `<= maxDeposit(msg.sender)`
  - **Impact**: None
- `receiver`: Address that will receive any ERC-20 tokens minted
  - **Control**: `_mint` ensures address is not 0.
  - **Authorization**: No checks
  - **Impact**: None

Function call analysis

- `maxDeposit(msg.sender)`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Return value will be 0 unless msg.sender is approved (by `DepositController.lenderVerifier.isAllowed` in `DepositController.maxDeposit`).
  - **What happens if it reverts, reenters, or does other unusual control flow?** Any returns of 0 will cause either revert or allow a mint of 0 tokens. External calls are only to the depositController which then calls back to the TrancheVault
- `depositController.onDeposit(msg.sender, amount, receiver)`
  - **What is controllable?** `msg.sender`, `amount`, `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** Only amount is used, and it is bounded to 0 and reasonable values by the `maxDeposit` call
  - **What happens if it reverts, reenters, or does other unusual control flow?** A broken depositController 'would break deposits.
- `_payDepositFee(depositFee)`
  - **What is controllable?** `depositFee` is a calculated from `amount`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Trusts that `token` is valid as part of the initialization
- `_depositAssets(amount - depositFee, shares, receiver)`
  - **What is controllable?** `amount`, `receiver`; `depositFee` and `shares` are calculated based on `amount`

- **If return value controllable, how is it used and how can it go wrong?** No return value
- **What happens if it reverts, reenters, or does other unusual control flow?** One of the many functions in the call tree may revert and deny a deposit

Function: `mint`

Intended behavior

Determine amount needed to mint the number requested shares, then mint and transfer them to the provided receiver address.

Uses `cacheTotalAssets` and `portfolioNotPaused` modifiers.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `shares`: Number of shares to mint
  - **Control**: No checks, but assetAmount calculated from it is checked to not be zero and less than maxDeposit for the sender.
  - **Authorization**: Calculated assetAmount is checked against `maxDeposit(msg.sender)`.
  - **Impact**: What impact can an attacker have if they control this input? None
- `receiver`: Address that will receive any ERC-20 tokens minted
  - **Control**: Only check: `_mint` ensures address is not 0.
  - **Authorization**: No checks
  - **Impact**: None

Function call analysis

- `depositController.onMint(msg.sender, shares, receiver)`
  - **What is controllable?** `shares` and `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** Return `assetAmount` and `depositFee` are calculated from `shares`, but are limited to sane values. Other args are ignored.

  - **What happens if it reverts, reenters, or does other unusual control flow?** The depositController is trusted to properly calculate values or else minting could be broken or performed at a lower or higher cost than intended.
- `maxDeposit(msg.sender)`
  - **What is controllable?** `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** The value returned will be 0 if `msg.sender` is not approved by `depositController.lenderVerifier.isAllowed`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Mints will be broken or denied if depositController or the lenderVerifier do not function properly.
- `_payDepositFee(depositFee)`
  - **What is controllable?** `depositFee` is calculated from `shares`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the fee cannot be transferred from `msg.sender` to the `managerFeeBeneficiary` the transaction will revert.
- `_depositAssets(assetAmount, shares, receiver)`
  - **What is controllable?** `shares` and `receiver`; `assetAmount` is based on `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** `_depositAssets` has a large call tree that has many child functions, many of which can perform checks and are expected to revert the current transaction in some cases. All of the trusted state contracts must be correct or critical undesired behavior could occur.

Function: `withdraw`

Intended behavior

Calculate the number of shares to redeem based on the assets requested, then burn those shares and send the assets to the specified receiver.

Uses `cacheTotalAssets` and `portfolioNotPaused` modifiers.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `assets`: Amount of underlying asset to be withdrawn
  - **Control**: Can be between 0 and `maxWithdraw(owner)`, inclusive.
  - **Authorization**: None, though the value is checked against authorization for the `owner` address.
  - **Impact**: None
- `receiver`: Address to send withdrawn amount to
  - **Control**: No checks other than ERC-20 checks to prevent 0 address.
  - **Authorization**: No checks
  - **Impact**: Address receives some amount of the underlying asset
- `owner`: Address that owns shares that will be redeemed.
  - **Control**: If `owner == msg.sender`, `_safeBurn` appropriately just calls `_burn` without checking or updating allowance.
  - **Authorization**: `_safeBurn` either checks that `owner` is `msg.sender` or that `allowance(owner, msg.sender)` is enough to cover the number of shares to be redeemed.
  - **Impact**: None, approval is required or implicit if `msg.sender` is `owner`.

Function call analysis

- `maxWithdraw(owner)`
  - **What is controllable?** `owner`
  - **If return value controllable, how is it used and how can it go wrong?** Assets to withdraw must be less than the value returned; this function must deny withdrawal if the owner doesn't have enough shares to cover the withdrawal or if it would put the vault below the withdrawController's `floor`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Withdrawals would be blocked on reverts.
- `withdrawController.onWithdraw(...,assets,...)`
  - **What is controllable?** `assets` is the controllable value that is used.
  - **If return value controllable, how is it used and how can it go wrong?** The values returned are a function of `assets`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Withdrawals would be blocked on reverts.
- `_payWithdrawFee(withdrawFee)`
  - **What is controllable?** `withdrawFee` is a function of `assets`
  - **If return value controllable, how is it used and how can it go wrong?** No return value

  - **What happens if it reverts, reenters, or does other unusual control flow?** Withdrawals would break if any of these functions revert.
- `_withdrawAssets(assets, shares, owner, receiver)`
  - **What is controllable?** `assets`, `owner`, `receiver`. `shares` is calculated from `assets`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Withdrawals will break if any of the functions called inside `_withdrawAssets`, such as a lack of authorization or not having enough shares during `_safeBurn`.

Function: `redeem`

Intended behavior

Redeem the specified number of shares on behalf of the specified owner and send the resulting amount of underlying token to the specified receiver.

Uses the `cacheTotalAssets` and `portfolioNotPaused` modifiers.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `shares`: The number of shares to redeem
  - **Control**: Can be between 0 and `maxRedeem(owner)`, inclusive.
  - **Authorization**: Checked against `maxRedeem(owner)`
  - **Impact**: None.
- `receiver`: Address that will receive the amount resulting from the redeemed shares
  - **Control**: ERC-20 function checks will prevent 0 address.
  - **Authorization**: None
  - **Impact**: Controls where the token resulting from redeem goes.
- `owner`: Address that owns the shares to be redeemed
  - **Control**: `maxRedeem(owner)` will return the number of shares the address owns and is used to gate the amount of shares that can be redeemed.

  - **Authorization**: `_safeBurn` checks that the `msg.sender` is owner or is approved to burn the specified number of shares.
  - **Impact**: None, the approvals appropriately check this value.

Function call analysis

- `maxRedeem(owner)`
  - **What is controllable?** `owner`
  - **If return value controllable, how is it used and how can it go wrong?** Assets to withdraw must be less than the value returned; this function must deny withdrawal if the owner doesn't have enough shares to cover the withdrawal or if it would put the vault below the withdrawController's `floor`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Redeeming would be blocked on reverts.
- `withdrawController.onRedeem( ... , shares, ... )`
  - **What is controllable?** `shares` is the controllable value that is used.
  - **If return value controllable, how is it used and how can it go wrong?** The values returned are a function of `shares`.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Redeeming would be blocked on reverts.
- `_payWithdrawFee(withdrawFee)`
  - **What is controllable?** `withdrawFee` is a function of `shares`
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Redeeming would break if any of these functions revert.
- `_withdrawAssets(assets, shares, owner, receiver)`
  - **What is controllable?** `shares`, `owner`, `receiver`. `assets` is calculated from `shares`.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Redeeming will break if any of the functions called inside `_withdrawAssets`, such as a lack of authorization or not having enough shares during `_safeBurn`.

Function: `onPortfolioStart`

Intended behavior

Intended to be called from `portfolio.start` which move the portfolio from `Status.CapitalFormation` to `Status.Live`. Updates the tranche's balance to moved to the portfolio and set the initial checkpoint.

Uses the `_requirePortfolio` function for access control, ensuring `msg.sender == portfolio`.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

No inputs.

Function call analysis

- `_requirePortfolio`
  - **What is controllable?** `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Used as a guard to revert unless `msg.sender` is `portfolio`
- `_transferFromTranche(address(portfolio), balance)`
  - **What is controllable?** `balance` is based on withdrawals during capital formation stage.
  - **If return value controllable, how is it used and how can it go wrong?** The `balance` amount of `token` is transferred to the portfolio address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If the token is broken, the tranche is broken.
- `_updateCheckpoint(balance)`
  - **What is controllable?** `balance` is based on withdrawals during capital formation stage.
  - **If return value controllable, how is it used and how can it go wrong?** No return value

- **What happens if it reverts, reenters, or does other unusual control flow?** If this transfer reverts, the portfolio won't be able to start.

Function: `onTransfer`

Intended behavior

Function called to update the virtualTokenBalance of the tranche when it receives `token` transfers via the portfolio.

Access control enforced by the `_requirePortfolio` function.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `assets`: The amount of `token` that the tranche address just received
  - **Control**: None
  - **Authorization**: None
  - **Impact**: Bad values would desynchronize `virtualTokenBalance`; `portfolio` must call this with appropriate amounts.

Function call analysis

- `_requirePortfolio`
  - **What is controllable?** `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Used as a guard to revert unless `msg.sender` is `portfolio`

Function: `updateCheckpoint`

Intended behavior

External function that updates the tranche's checkpoints with the current value `totalAssets()`.

Only usable once the portfolio is closed and not paused.

Any address can call this function and trigger a checkpoint update and the associated protocol and management fee payments.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

No inputs.

Function call analysis

- `portfolio.status()`
  - **What is controllable?** Nothing
  - **If return value controllable, how is it used and how can it go wrong?** This function reverts unless the portfolio's status is closed
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function reverts unless the portfolio's status is closed
- `_updateCheckpoint(totalAssets())`
  - **What is controllable?** `totalAssets` could be affected by deposit/withdrawals.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** If any of the external calls to `portfolio`, `protocolConfig` or `token` functions revert, this function will not function.

Function: `updateCheckpointFromPortfolio`

Intended behavior

Updates the tranche's checkpoint based on the new total asset value passed in.

Access is controlled by `_requirePortfolio`.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `newTotalAssets`: Value describing the current total assets as tracked by the portfolio
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: No checks, this value is presumed to be correct since it is coming from the portfolio.

Function call analysis

- `_requirePortfolio`
  - **What is controllable?** `msg.sender` to some extent.
  - **If return value controllable, how is it used and how can it go wrong?** No return value
  - **What happens if it reverts, reenters, or does other unusual control flow?** Used as a guard to revert unless `msg.sender` is `portfolio`
- `_updateCheckpoint(newTotalAssets)`
  - **What is controllable?** `newTotalAssets` is an argument, but it is trusted to be correct since it has to come from the portfolio.
  - **If return value controllable, how is it used and how can it go wrong?** No return address.
  - **What happens if it reverts, reenters, or does other unusual control flow?** If this function reverts due to a failure in external calls, the portfolio will be unable to update the tranche's checkpoint directly.

Function: `configure`

Intended behavior

External function that sets `managerFeeRate`, `managerFeeBeneficiary`, `depositController`, and `withdrawController` if the corresponding argument differs from the current value.

No checks are performed in this function, but each of the three called functions use

`_requireManagerRole` or `_requireTrancheControllerOwnerRole` for access control. As a result this function is only a helper; refer to each of the four called functions for notes: `setManagerFeeRate`, `setManagerFeeBeneficiary`, `setDepositController`, and `setWithdrawController`

Function: `set[Deposit|Withdraw|Transfer]Controller`

Intended behavior

These functions each update their respective controller address with a new one and emit the appropriate related event.

They are all public but guarded by `_requireTrancheControllerOwnerRole`.

Inputs

- `newController`: Address to be set as the new controller
  - **Control**: No checks
  - **Authorization**: `_requireTrancheControllerOwnerRole`
  - **Impact**: These controllers implement critical functionality; erroneous values would break associated withdraw, deposit, or transfer functionality until a new correct value is set.

Function call analysis

- `_requireTrancheControllerOwnerRole`
  - **What is controllable?** `msg.sender` is the sender.
  - **If return value controllable, how is it used and how can it go wrong?** This function guards access to critical functionality.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function will revert unless `msg.sender` has the appropriate role to change sensitive state variables.

Function: `setManagerFee[Rate|Beneficiary]`

Intended behavior

These functions both update the checkpoint, then update their respective state variable with a new one and emit the appropriate related event.

They are both public but guarded by `_requireManagerRole`.

Preconditions

- Precondition: Key functionality contracts must be deployed and properly initialized.

Inputs

- `_managerFee[Rate|Beneficiary]`: Value to write to corresponding state variable
  - **Control**: No checks
  - **Authorization**: `_requireManagerRole`
  - **Impact**: These values manage the percentage and recipient of management fees and should only be changed by the appropriate address.

Function call analysis

- `_requireTrancheControllerOwnerRole`
  - **What is controllable?** `msg.sender` is the sender.
  - **If return value controllable, how is it used and how can it go wrong?** This function guards access to critical functionality.
  - **What happens if it reverts, reenters, or does other unusual control flow?** This function will revert unless `msg.sender` has the appropriate role to change sensitive state variables.

Function: `setPortfolio`

Intended behavior

Sets the portfolio address if not set.

This function is designed to be called from the associated portfolio's initialization so the portfolio variable is set only once and not changeable.

Preconditions

None.

Inputs

- `portfolio`: Address to set as the portfolio for this tranche

  - **Control**: Cannot be 0
  - **Authorization**: No checks
  - **Impact**: This is a critically important variable and must be set in the same block that the tranche is deployed to avoid problems. The current design of deploying via `StructuredPortfolioFactory` accomplishes this.

Function call analysis

No function calls.

5.4 File: `ProtocolConfig.sol`

Function: `initialize`

Intended behavior

Contract initializer; uses `initializer` modifier.

msg.sender gets Admin and Pauser roles and is trusted to provid valid input parameters to this call.

Preconditions

- Precondition: protocolAdmin, protocolTreasury, and pauserAddress are assumed to be deployed before contract will be functional.

Inputs

- `_defaultProtocolFeeRate`: the default protocol fee rate
  - **Control**: Not checked
  - **Authorization**: Not checked
  - **Impact**: Fees could be unreasonably high or low
- `_protocolAdmin`: administrator address
  - **Control**: Not checked
  - **Authorization**: Not checked
  - **Impact**: Install malicious admin over tranches/protocols
- `protocolTreasury`: Treasury address
  - **Control**: Not checked

  - **Authorization**: Not checked
  - **Impact**: Wrong treasury address installed for tranches
- `_pauserAddress`: address that can pause the address
  - **Control**: Not checked
  - **Authorization**: Not checked
  - **Impact**: Malicious pauser installed for loans and portfolios

Function: `protocolFeeRate`

Intended behavior

View function that returns either the customFeeRate for the contract or defaultProtocolFee if the contract's customFeeRate is zero or not set. One variant takes an address as an argument, the has no arguments and uses `msg.sender` instead.

Function call analysis

The function `_protocolFeeRate(address)` does the actual work.

Function: `setCustomProtocolFeeRate`

Intended behavior

Sets `customFeeRates[contractAddress]`. Requires msg.sender to have default admin role.

Preconditions

- Precondition: If customFeeRate was previously set, the newFeeRate can't be the same as previously set.

Inputs

- `contractAddress`: Address to set fee rate for
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: An unexpected rate could be set for the wrong contract
- `newFeeRate`: The new rate
  - **Control**: Can't be same as existing rate if one was set

- **Authorization**: No checks
- **Impact**: A fee that is too high or too low could be set

Function call analysis

Calls `_requireDefaultAdminRole()`

Function: `removeCustomProtocolFeeRate`

Intended behavior

Removes the custom protocol fee rate for a given contract.

Requires msg.sender has the default admin role and for the feeRate to have been set.

Sets `customFeeRates[contractAddress]` to have a `feeRate` of 0 `isSet` of false.

Inputs

- `contractAddress`: Address to set fee rate for
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: A rate could be unexpectedly removed for a contract

Function call analysis

Calls `_requireDefaultAdminRole()`

Function: `set[DefaultProtocolFeeRate|ProtocolAdmin|ProtocolTreasury|PauserAddress]`

Intended behavior

Each of these functions allows the respective state variable to be changed and emits an event on successful change.

Each requires msg.sender has the default admin role and that the new value is not the same as the old.

Inputs

- `new[DefaultProtocolFeeRate|ProtocolAdmin|ProtocolTreasury|PauserAddress]`: new value for state variable
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: An unexpected or malicious value could be set

Function call analysis

Each calls `_requireDefaultAdminRole()`

5.5 File: `LoanManager.sol`

No threat model is included for this module because it is an abstract contract with no non-trivial external state-changing functions.

The exposed functionality that reaches these functions is addressed in the `StructuredPortfolio` module.

5.6 File: `DepositController.sol`

Function: `initialize`

Intended behavior

External initializer function (has `initializer` modifier). Sets state variables for the manager role, lenderVerifier, depositFeeRate, and ceiling, as well as marking deposits allowed for the capitol formation state.

No checks are performed on the inputs; this a sensitive initialization function.

Preconditions

- Precondition: lenderVerifier must exist and implement the ILenderVerifier interface

Function: `maxDeposit`

Intended behavior

Returns what the maximum deposit allowed given the calling tranche's assets and ceiling for the designated receiver address.

Branches and code coverage

When line coverage is implemented, ensure all four return statements are covered.

Preconditions

Violation of sanity requirements (lenderVerifier must exist and be implement the proper interface) would cause a revert.

Inputs

- `receiver`: A potentially valid depositor address
  - **Control**: No checks
  - **Authorization**: `lenderVerifier.isAllowed(receiver)`, returns 0 if not allowed
  - **Impact**: Nonzero deposit values could be returned for undesired addresses.

Function call analysis

- `lenderVerified.isAllowed(receiver)`
  - **What is controllable?** `receiver`
  - **If return value controllable, how is it used and how can it go wrong?** Nonzero deposit values could be returned for undesired addresses.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nonzero deposit values could be returned for undesired addresses, reverts would stop deposits.
- `msg.sender.portfolio().status()`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Nothing, this function returns a value for the sender based on the sender's status.

- **What happens if it reverts, reenters, or does other unusual control flow?** Nothing, this function returns a value for the sender based on the sender's status.

Function: `previewDeposit`

Intended behavior

Returns the number of shares of a tranche that can be minted for a given amount of the underlying asset.

Preconditions

- Precondition: msg.sender must implement the ITrancheVault interface

Inputs

- `assets`: Quantity of underlying asset to convert to number of shares
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: None, this is a public view function

Function call analysis

- `_getDepositFee(assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** Return value is a uint256 output of multiply then divide
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `msg.sender.convertToShares(assets - depositFee)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** msg.sender controls the function and the value returned.
  - **What happens if it reverts, reenters, or does other unusual control flow?** Nothing of note; msg.sender is responsible for the input and output.

Function: `maxMint`

Intended behavior

Returns the maximum number of shares the designated receiver can mint.

Just returns `previewDeposit(maxDeposit(receiver))` where `receiver` is the lone argument. Further details in `maxDeposit` and `previewDeposit`.

Function: `onDeposit`

Intended behavior

Returns the deposit fee and number of shares for a deposit of the specified amount.

Preconditions

- Precondition: msg.sender must implement the ITrancheVault interface

Inputs

- `assets`: Amount of underlying asset in the deposit
  - **Control**: None
  - **Authorization**: None
  - **Impact**: None, it is a non-state-changing external view function

Function call analysis

- `_getDepositFee(assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** The value is used to calculate a fee, but can't be abused in and of itself.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A
- `previewDeposit(assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** The value is used to calculate number of shares, but is a non-state-changing view function.

  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

Function: `onMint`

Intended behavior

Returns the amount of assets and deposit fee for minting the specified number of shares.

Preconditions

- Precondition: msg.sender implements the ITrancheVault interface.

Inputs

- `shares`: Number of shares to mint
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: The caller controls the value that will be passed back into their own `convertToAssetsCeil` function

Function call analysis

- `msg.sender.convertToAssetsCeil(shares)`
  - **What is controllable?** `shares`
  - **If return value controllable, how is it used and how can it go wrong?** The sender controls the input and the function
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A, sender controls input and the function
- `_getDepositFee(assets)`
  - **What is controllable?** `assets`
  - **If return value controllable, how is it used and how can it go wrong?** The value is used to calculate a fee, but can't be abused in and of itself.
  - **What happens if it reverts, reenters, or does other unusual control flow?** N/A

Function: `set[Ceiling|DepositFeeRate|LenderVerifier]`

Intended behavior

These public functions each set the respective state variable, using `_requireManagerRole` to guard access and emitting the corresponding change event.

Preconditions

None

Inputs

- `new[Ceiling|DepositFeeRate|LenderVerifier]`: new value for state variable
  - **Control**: No checks
  - **Authorization**: Requires manager role
  - **Impact**: The caller is trusted, but unexpected values could cause unexpected behavior.

Function: `setDepositAllowed`

Intended behavior

This public function sets whether deposits are allowed for the CapitalFormation or Live statuses, using `_requireManagerRole` to guard access, emitting the `DepositAllowedChanged` event on success.

Preconditions

None

Inputs

- `newDepositAllowed`: boolean indicating whether deposits are allowed in the corresponding status
  - **Control**: No checks
  - **Authorization**: Requires manager role
  - **Impact**: The caller is trusted to specify whether deposits should be enabled.

- `portfolioStatus`: Status indicating which status to change in the depositAllowed mapping
  - **Control**: require check allows only Status.CapitalFormation and Status.Live
  - **Authorization**: Requires manager role
  - **Impact**: None

Function: `configure`

Intended behavior

External function that sets each of the four primary state variables if the corresponding argument differs from the current value.

No checks are performed in this function, but each of the four called functions use `_requireManagerRole` for access control. As a result this function is only a helper; refer to each of the four called functions for notes: `setCeiling`, `setDepositFeeRate`, `setLenderVerifier`, `setDepositAllowed`.

5.7 File: `WithdrawController.sol`

Function: `initialize`

Intended behavior

External initializer function with `initializer` modifier. Sets specified manager role, floor, and withdrawFeeRate, as well as setting `withdrawAllowed[Status.Closed] = true`

No checks are performed on the inputs; this a sensitive initialization function.

Preconditions

None

Inputs

- `manager`: Address to set as manager
- `_withdrawFeeRate`: Rate to set on withdrawals
- `floor`: Minimum amount to ensure withdrawals to not go beyond

Function call analysis

None

Function: `maxWithdraw`

Intended behavior

A non-state-changing public view function returning the maximum amount that the specified address can withdraw.

The sender is presumed to be a tranche vault determining how much to allow an address to withdraw.

Branches and code coverage

When line coverage is implemented, ensure all three return statements are covered as well as both possibilities of the `Math.min` function.

Preconditions

- Precondition: msg.sender is presumed to implement the ITrancheVault interface

Inputs

- `owner`: Address to determine maximum withdrawal amount for
  - **Control**: No checks, but `vault.balanceOf` may return 0
  - **Authorization**: No checks
  - **Impact**: Nothing, this is a non-state-changing view function that calls back to msg.sender

Function call analysis

- `msg.sender.portfolio().status()`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.

- `msg.sender.balanceOf(owner)`
  - **What is controllable?** `msg.sender` and `owner`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.
- `msg.sender.convertToAssets(ownerShares)`
  - **What is controllable?** `msg.sender` and `ownerShares`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.
- `_globalMaxWithdraw(msg.sender)`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.

Function: `maxRedeem`

Intended behavior

A non-state-changing public view function returning the maximum number of shares that the specified address can redeem.

The sender is presumed to be a tranche vault determining how much to allow an address to redeem.

Branches and code coverage

When line coverage is implemented, ensure all three return statements are covered as well as both possibilities of the `Math.min` function.

Preconditions

- Precondition: msg.sender is assumed to implement the ITrancheVault interface.

Inputs

- `owner`: Address to determine maximum redeem amount for
  - **Control**: No checks, but `vault.balanceOf` may return 0
  - **Authorization**: No checks
  - **Impact**: Nothing, this is a non-state-changing view function that calls back to msg.sender

Function call analysis

- `msg.sender.portfolio().status()`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.
- `msg.sender.balanceOf(owner)`
  - **What is controllable?** `msg.sender` and `owner`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.
- `_globalMaxWithdraw(msg.sender)`
  - **What is controllable?** `msg.sender`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.
- `msg.sender.convertToShares(globalMaxWithdraw)`
  - **What is controllable?** `msg.sender` and `globalMaxWithdraw`
  - **If return value controllable, how is it used and how can it go wrong?** Sender can return whatever status they desire
  - **What happens if it reverts, reenters, or does other unusual control flow?** None, this is a non-state-changing view function that calls back to msg.sender.

Function: `set[Floor|WithdrawFeeRate]`

Intended behavior

These public functions each set the respective state variable, using `_requireManagerRole` to guard access and emitting the corresponding change event.

Preconditions

None

Inputs

- `new[Floor|WithdrawFeeRate|LenderVerifier]`: new value for state variable
  - **Control**: No checks
  - **Authorization**: Requires manager role
  - **Impact**: The caller is trusted, but unexpected values could cause unexpected behavior.

Function: `setWithdrawAllowed`

Intended behavior

This public function sets whether withdrawals are allowed for the CapitalFormation or Live statuses, using `_requireManagerRole` to guard access, emitting the `WithdrawAllowedChanged` event on success.

Preconditions

None

Inputs

- `newWithdrawAllowed`: boolean indicating whether withdrawals are allowed in the corresponding status
  - **Control**: No checks
  - **Authorization**: Requires manager role
  - **Impact**: The caller is trusted to specify whether withdrawals should be enabled.

- `portfolioStatus`: Status indicating which status to change in the withdrawAllowed mapping
  - **Control**: require check allows only Status.CapitalFormation and Status.Live
  - **Authorization**: Requires manager role
  - **Impact**: None

Function: `configure`

Intended behavior

External function that sets each of the three primary state variables if the corresponding argument differs from the current value.

No checks are performed in this function, but each of the three called functions use `_requireManagerRole` for access control. As a result this function is only a helper; refer to each of the three called functions for notes: `setFloor`, `setWithdrawFeeRate`, `setWithdrawAllowed`.

5.8 File: `FixedInterestOnlyLoans.sol`

While this module is used in tests as an example Loans provider module could look like, we include a threat model so that it can maybe helpful when designing production-ready Loan modules in the future.

Function: `initialize`

Intended behavior

Initialization function designed to be called during the construction of the `FixedInterestOnlyLoans` contract. This contract is supposed to be deployed and initialized separately and is not called as part of the creation process in `StructuredPortfolioFactory`.

It calls the initialization methods for the Upgradeable and ERC721 modules.

Preconditions

- The `_protocolConfig` contract must contain a valid pauser address which acts as the person who can pause the Loans contract.

Inputs

- `_protocolConfig`: Address of the ProtocolConfig contract.
  - **Control**: No checks
  - **Authorization**: No checks
  - **Impact**: They can control the pauser for the Loans contract.

Function call analysis

- `__Upgradeable_init`
  - This sets up the proper roles and authorization for upgrading and pausing the contract.
  - The caller of the `initialize` function is set as the admin for the Upgradeable contract, and the pauser address is pulled from the protocol config.
- `__ERC721_init`
  - ERC721 initialization function is called with static arguments, and nothing is controlled.

Function: `issueLoan`

Intended behavior

The `issueLoan` function creates a new `ERC721` NFT which represents a loan in the Loans contract.

Preconditions

- The Loans contract must not be in a paused state. This is verified using the `whenNotPaused` modifier.

Inputs

- `asset`: ERC20 token which the loan is denominated in.
  - **Control**: No checks.
  - **Authorization**: No checks.
  - **Impact**: Within the context of the Loans contract, the asset variable is stored in the LoanMetadata object. The actual value transfers are performed in `LoansManager` which

use the `asset` variable stored in that contract instead of the variable stored in LoanMetadata. As a result, there is no critical impact as of now if this is controlled by the attacker. However, in the future, if this variable is used for transfers or accounting purposes, it could lead to critical vulnerabilities.

- `_principal`: The amount of the loan which is disbursed to the recipient.
  - **Control**: No checks
  - **Authorization**: No checks.
  - **Impact**: This controls how much money is sent from the portfolio to a loan recipient.

If this is different from what has been approved by the portfolio manager, this could lead to misappropriation/loss of funds.

- `_periodCount`, `_periodDuration`, `_periodPayment`: These inputs control the terms of the loan such as the repayment period, interest, and the frequency of payments.
  - **Control**: It is ensured that the duration of the loan and the total payment made to repay the loan are both non-zero. This ensures that all the three parameters are non-zero.
  - **Authorization**: No checks.
  - **Impact**: An attacker can change the terms of the loan to their benefit.
- `_recipient`: The person who receives the funds from the loan.
  - **Control**: The addresses is checked to ensure it is non-zero.
  - **Authorization**: No checks.
  - **Impact**: An attacker can redirect the funds of the loan from the intended recipient to themselves without having to repay the loan, which results in the portfolio taking the loss.
- `_gracePeriod`: How long after the end date of the loan can the portfolio manager mark the loan as defaulted.
  - **Control**: No checks.
  - **Authorization**: No checks.
  - **Impact**: An attacker can indefinitely stall the loan preventing the portfolio manager from marking the loan as defaulted.
- `_canBeRepaidAfterDefault`: Can the loan still be paid towards after it is marked as defaulted.
  - **Control**: No checks.
  - **Authorization**: No checks.
  - **Impact**: An attacker can change the intended behavior of the loan if it defaults.

Function call analysis

- `_safeMint`
  - Calls the internal ERC721 function to mint a NFT which is used for book-keeping of the loans in the contract.
  - The owner of the NFT is set to the caller of the `issueLoan` function, which in the normal use case is the `StructuredPortfolio` contract.

Function: `acceptLoan`

Intended behavior

The recipient of the loan calls the functions after reviewing the details and terms of the loan on chain and ensuring that they are as expected.

Preconditions

- The Loans contract must not be paused.

Inputs

- `instrumentId`: The loan ID returned by the `issueLoan` function.
  - **Control**: The loan specified must be in the `Created` state.
  - **Authorization**: The caller of the function must be the same as the recipient specified in the loan metadata.
  - **Impact**: No impact.

Function call analysis

- `_changeLoanStatus`
  - Changes the status of the loan from `Created` to `Accepted`.

Function: `start`

Intended behavior

Marks the loan as started and calculates some book-keeping timestamps such as the end of the loan and the end date of the current period

Preconditions

- Assumes that the caller of the function will fund the loan by transferring the assets to the recipient upon successful execution of this function.
- The loan contract must not be paused.

Inputs

- `instrumentId`: The ID of the loan which is being started.
  - **Control**: The loan must be in the accepted state.
  - **Authorization**: Only the owner of the loan, i.e the caller of the `issueLoan` function is allowed to call this function to start the loan.
  - **Impact**: An attacker may be able to mess up the state of the loan contract by marking loans as incorrectly started.

Function call analysis

- `_changeLoanStatus`
  - Changes the status of the loan from `Accepted` to `Started`.

Function: `repay`

Intended behavior

Marks one period of the loan as repaid. If it is the last period of the loan, then it ensures that the entire principal of the loan is also repaid at the end.

Preconditions

- The loan contract must not be paused.
- The `amount` passed to the `repay` function must also be transferred in the underlying token to the portfolio from the borrower after the successful execution of the `repay` function.

Branches and code coverage

- If the loan being repaid is in it's last period, then the entire principal of the loan is also included in the amount that must be transferred by the borrower to the

portfolio. The status of the loan is thus changed to `Repaid`

Inputs

- `instrumentId`: The ID of the loan which is being started.
  - **Control**: The loan must be in a valid state where it can be repaid.
  - **Authorization**: Only the owner of the loan, i.e the caller of the `issueLoan` function is allowed to call this function to start the loan.
  - **Impact**: An attacker may be able to mess up the state of the loan contract by repaying the wrong loans.
- `amount`: The amount of the underlying token that is being transferred by the borrower to the portfolio.
  - **Control**: The amount must be equal to expected repayment which calculated based on the current period of the loan being repaid.
  - **Authorization**: No checks.
  - **Impact**: An attacker may be able to overstate the amount of money they repaid for the loan leading to loss of funds for the depositors and the protocol.

Function call analysis

- `_changeLoanStatus`
  - Changes the status of the loan from to `Repaid` if the loan is its last period and is repaid.
- `_canBeRepaid`
  - Ensures that the loan is in a valid state:
- The loan is currently started and in its repayment period.
  - The loan has defaulted but the terms of the loan allow it to be repaid after default.

Function: `cancel`

Intended behavior

Cancels a loan before it has been funded.

Preconditions

- The loan contract must not be paused.

Inputs

- `instrumentId`: The ID of the loan which is being cancelled.
  - **Control**: The loan must be in the created or accepted state.
  - **Authorization**: Only the owner of the loan, i.e the caller of the `issueLoan` function is allowed to call this function to start the loan.
  - **Impact**: An attacker may be able to mess up the state of the loan contract by incorrectly marking loans as cancelled.

Function call analysis

- `_changeLoanStatus`
  - Changes the status of the loan from `Accepted/Created` to `Canceled`.

Function: `markAsDefaulted`

Intended behavior

Mark a loan as defaulted.

Preconditions

- The loan contract must not be paused.

Inputs

- `instrumentId`: The ID of the loan which is being marked as defaulted.
  - **Control**: The loan must be in the started state.
  - **Authorization**: Only the owner of the loan, i.e the caller of the `issueLoan` function is allowed to call this function to start the loan.
  - **Impact**: An attacker may be able to mess up the state of the loan contract by incorrectly marking loans as defaulted.

Function call analysis

- `_changeLoanStatus`
  - Changes the status of the loan from `Started` to `Defaulted`.

Function: `updateInstrument`

Intended behavior

Increases the grace period for a loan.

Preconditions

- The loan contract must not be paused.

Inputs

- `instrumentId`: The ID of the loan which is being marked as defaulted.
  - **Control**: The loan must be in the started state.
  - **Authorization**: Only the owner of the loan, i.e the caller of the `issueLoan` function is allowed to call this function to start the loan.
  - **Impact**: An attacker may be able to mess up the state of the loan contract by incorrectly marking loans as defaulted.
- `newGracePeriod`: The new grace period for the loan.
  - **Control**: It must be greater than the previous grace period.
  - **Authorization**: No checks.
  - **Impact**: An attacker can indefinitely stall the loan preventing the portfolio manager from marking the loan as defaulted.

6 Audit Results

At the time of our audit, the code was not deployed to mainnet EVM.

During our audit, we discovered six findings. Of these, one was medium risk, two were low risk, and three were suggestions (informational). TrueFi acknowledged all findings and implemented fixes.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.