PUBLIC

Code Assessment

of the Perimeter Smart Contracts

September 26, 2023

Produced for

CIRCLE

by

CHAINSECURITY

Contents

1 Executive Summary

Dear Rachel,

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

Circle implements Perimeter, which can be used as on-chain infrastructure to facilitate the operations of loans that are secured off-chain. This includes custody and transfer of lender's funds, interest payments, and fee handling.

The most critical subjects covered in our audit are asset solvency, functional correctness, and access control. The general subjects covered are fee handling, event handling, gas efficiency, and upgradeability. Several Possible Gas Optimizations exist that would increase gas efficiency. Furthermore, the implementation of EIP-4626 can be improved: EIP-4626 Non-Compliance. All other mentioned subjects show a high level of security.

In summary, we find that the codebase provides a high level of security.

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	1
• Code Corrected	1
High-Severity Findings	2
• Code Corrected	2
Medium-Severity Findings	5
• Code Corrected	4
• Code Partially Corrected	1
Low-Severity Findings	8
• Code Corrected	5
• Specification Changed	1
• Risk Accepted	1
• Acknowledged	1

2 Assessment Overview

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

2.1 Scope

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

V	Date	Commit Hash	Note
1	01 January 2023	384571416209d08623c6ace9422613fc8970475d	Initial Version
2	17 February 2023	cc1ef9d85e085fa0fbf286037ee725e69cc62419	Second Version

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

Correct handling of ERC-20 underlying tokens was only considered for **USDC** and **EUROC**.

2.1.1 Excluded from scope

External libraries like the OpenZeppelin `contracts` and `contracts-upgradeable`.

2.2 System Overview

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

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

Circle offers a lending protocol that allows under-collateralized loans. At its core, the protocol implements an EIP-4626-compliant vault (the Pool) that can hold a single token. Tokens are whitelisted by the protocol owner and each Pool is created by a Pool Admin that manages the loans and collects fees. Lenders can deposit tokens to the Pool and receive non-transferable ERC-20 Pool Tokens. The Pool Admin uses the funds of the lenders to create loans that can optionally be collateralized on-chain with ERC-20 and ERC-721 tokens. On-chain collateralization is, however, not enforced and no liquidation mechanism is implemented. The Pool Admin can claim the collateral when a loan is defaulted and handle its liquidation as needed. The protocol is designed to work with trusted entities. For this reason, the core protocol is extended with a permission system using Verite on-chain identity management. In this permissioned version, Pool Admins, borrowers, and lenders all have to be identified by trusted verifiers who issue signatures following a certain scheme. If this scheme is accepted by the Pool, the signature can be used to get verified on this Pool.

Pool Admins issue loans to identified borrowers and negotiate the terms of the loan off-chain. The repayment of loans must be enforceable through off-chain agreements, as the smart contracts do not enforce repayment. Interest payments and the final repayment of the loan's principal happen on-chain and increase the Pool's balance, resulting in accrued interest for lenders.

If a borrower defaults on a loan, the Pool Admin can signal the default on-chain. At this point, the loan's principal is no longer counted towards the Pool's assets and results in a loss for the lenders. To ease the danger of defaults, Pool Admins have to deposit a certain amount of tokens as first loss capital before deposits are activated. These tokens are used to compensate users in case of a default.

Lenders wishing to withdraw their tokens have to request a withdrawal first. Withdrawal requests are periodically served, but only a certain amount becomes available per period depending on the settings the Pool Admin has created the Pool with.

2.2.1 Contracts

The following sections discuss the contracts in detail:

ServiceConfiguration

The main contract defining all protocol-wide settings and roles is called `ServiceConfiguration`. It is operated by the deployer of the Perimeter protocol and implements pausing functionality, token whitelisting, First Loss handling, and `LoanFactory` verification.

The `ServiceConfiguration` also defines a protocol fee but it is set to 0 and never used.

Factories

The contracts are typically deployed by factories (except `ServiceConfiguration` and `ToSAcceptanceRegistry`) as Beacon Proxies. The Factories store the address of the Beacon implementation. Some Factories (e.g., the `LoanFactory`) also store the addresses of created contracts for verification purposes.

Vault

`Vaults` are simple contracts that hold tokens. These tokens can then be transferred out by the owner of the contract using the methods `withdrawERC20` and `withdrawERC721`. Different Vaults are created upon initialization of the base contracts:

- `firstLossVault` is owned by the `PoolController` and contains the funds for first loss protection.
- `collateralVault` holds the posted collateral of a `Loan`.
- `fundingVault` contains the tokens that can be withdrawn by a borrower of a `Loan`.
- `feeVault` receives the Service and Origination fees that can be withdrawn by the Pool Admin.

Loan

A `Loan` is created by the `LoanFactory`. It encapsulates the settings and vaults for a single Loan catered to a single borrower. This loan can have one of two types:

1. `Fixed`: A Fixed Loan has a pre-determined end date. Principal can be paid back earlier but the interest over the whole period has to be paid back completely nonetheless. To start the Loan, the borrower has to withdraw the full principal amount.
2. `Open`: An Open Loan can be paid back at any time. It can also be called back by the Pool Admin at any time. The borrower is not required to withdraw the full Loan amount and can pay back parts of the Loan amount during the runtime. The interest payments are, however, not reduced by this behavior. The Pool Admin callback functionality is not currently implemented. A Callback can be signaled on-chain but must be enforced off-chain (or by marking the Loan as defaulted).

A created `Loan` runs through different stages:

1. `Requested`: After Loan creation, the Loan is in this stage awaiting funding from the Pool. At this stage, the borrower can call `cancelRequested` to cancel the loan issuance.
2. `Canceled`: Non-active loans can be canceled to reach this stage.
3. `Collateralized`: This optional stage is reached after the borrower either deposits ERC-20 tokens via `postFungibleCollateral` or ERC-721 tokens via `postNonFungibleCollateral` as collateral in the `Requested` stage. The functions can also be used in later stages to add or

increase a collateral position. The loan can still be canceled by the borrower in this stage, but only after a certain timestamp (`dropDeadTimestamp`) has been reached. This is done by calling `cancelCollateralized`. Collateral is kept in the Loan's `collateralVault`.

4. `Funded`: To reach this stage, the Pool Admin calls `fundLoan` in the `PoolController` (explained later) with the Loan address as argument. If the associated Pool holds enough funds, the principal is transferred to the Loan's `fundingVault`. Funded Loans can be canceled either by the borrower or by the Pool Admin (in (Version 2), the Pool Admin can only call this function via the `PoolController`) using `cancelFunded`. In both cases, the `dropDeadTimestamp` has to be reached.
5. `Active`: As soon as the borrower calls `drawdown` on a Funded Loan, this stage is reached. From this point on, the borrower owes interest payments in fixed intervals that can be paid using the `completeNextPayment` function. Active Loans can not be canceled anymore.
6. `Matured`: The borrower can call `completeFullPayment` and pay back the principal plus any interest that has not been paid yet. In the case of a Fixed Loan, this includes all future payments until the end date of the Loan. In case of an Open Loan, this includes the part of the current payment from period start to the current timestamp.
7. `Defaulted`: If a borrower does not pay interest in time, the Pool Admin has the possibility of defaulting the Loan on-chain by calling `defaultLoan` in the `PoolController`. Performing this action lies at the sole discretion of the Pool Admin and cannot be reversed. If the Loan enters this stage, the outstanding principal is removed from the Pool's assets and lenders incur a loss.

When a loan reaches the `Canceled` or `Matured` stage, the borrower is allowed to withdraw the posted collateral. If it reaches the `Defaulted` stage, the Pool Admin can withdraw the collateral instead. Both use the function `claimCollateral` for this purpose (in (Version 2), the Pool Admin can only call this function via the `PoolController`.).

For Open Loans, three additional functions are used:

- `paydownPrincipal` can be used by the borrower to repay some of the principal during the Loan runtime.
- `reclaimFunds` can be used by the Pool Admin to transfer principal of an Open Loan that has not been drawn down or was repaid early back to the Pool. If an Open Loan has been defaulted, the Pool Admin also has to call this function to ensure that all funds return to the Pool. If an Open Loan has matured but some funds have not been drawn down before, the funds also have to be sent back manually by the Pool Admin. In (Version 2), this function is only callable by the `PoolController`.
- `markCallback` can be used by the Pool Admin to signal on-chain that the Loan has been recalled. In (Version 2), this function is only callable by the `PoolController`.

A Loan can only be repaid by the borrower's address. If the borrower loses their keys, they will have to send the missing funds to the Pool Admin that can deposit the amount to the `firstLossVault` and then default the Loan.

Pool

A `Pool` is created by the `PoolFactory` with various settings including the fee percentages, the underlying token (whitelisted by the `ServiceConfiguration`), and the Withdraw Gate (explained below). Corresponding `PoolController` and `WithdrawController` are created and associated with the Pool. The creator is set as the Pool Admin in the `PoolController`. This privilege cannot be transferred to another address.

The Pool implements all EIP-4626 functions and some additional functions that have been derived from the standard:

- `deposit` / `mint` allow lenders to transfer funds to the protocol in exchange for Pool Tokens (`shares`).
- `requestRedeem` / `requestWithdraw` allows lenders to create withdrawal requests for their assets that can be redeemed after some time.

- `cancelRedeemRequest` / `cancelWithdrawRequest` allows lenders to abort running withdrawal requests. This is only possible for assets that have not been marked as withdrawable yet.
- `withdraw` / `redeem` allow lenders to redeem assets that have been marked as withdrawable after a withdrawal request.
- In (Version 2), the Pool Admin can withdraw fees with the function `withdrawFeeVault`.

The Pool also offers a `snapshot` function that allows any user to update withdrawal snapshots when a new period is started. This is needed to ensure snapshots can be generated even when no other interactions happen on the contracts for a longer duration.

PoolController

Each `Pool` creates a `PoolController` on initialization. The Pool Controller allows a Pool Admin to manage the associated Pool. The Pool Controller handles the life-cycle of the Pool with the following stages:

- `Initialized`: In this stage, the Pool Admin can change most of the Pool settings except the First Loss amount and Withdraw Request duration.
- `Active`: By calling `depositFirstLoss` and transferring the First Loss amount to the `firstLossVault`, the Pool Admin can activate the Pool. In this stage, Loans can be funded using `fundLoan` and defaulted using `defaultLoan`. More First Loss tokens can also be transferred.
- `Closed`: Each Pool has an end date. As soon as this date is reached, the `Closed` stage is automatically reached. In this stage, the Withdraw Gate is automatically set to 100% and the Withdraw Period duration is set to 1 day (or less, if it was less before) allowing users to withdraw all of their funds at once, given that the funds are already available (i.e., all Loan principals have been paid back). If all Loans have been paid back, the Pool Admin can now also call `withdrawFirstLoss` to get the First Loss amount (and possible accrued First Loss Fees) back.

Using `claimFixedFee`, the Pool Admin can also periodically withdraw a fee that is taken directly from the Pool funds.

WithdrawController

Each `Pool` creates a `WithdrawController`. It encapsulates the state of withdrawal requests and all associated data like snapshots. State-changing functions are only callable by the Pool and are used for snapshotting, requesting withdrawals, and actual withdrawals.

VeriteAccessControl

`VeriteAccessControl` is an abstract contract that is used to verify Verite signatures. For this purpose, the admin of the contract can add and remove trusted verifiers whose signatures are accepted as verification proof. This is done using the functions `addTrustedVerifier` and `removeTrustedVerifier`. The admin can also enable or disable accepted Verite schemas (JSON representations of what is verified. These include at least an `attribute` and a URL to the `process` used) using `addCredentialSchema` and `removeCredentialSchema`.

Users with a valid Verite signature that has been issued by one of the trusted verifiers using one of the allowed schemas can call `verify` to get a verification entry. The `isAllowed` function of the contract now returns `true` for their address. Once verified, verifications are checked on each interaction with the permissioned contracts as they are only valid for a limited time.

ToSAcceptanceRegistry

The `ToSAcceptanceRegistry` allows any user to accept a given terms-of-service URL on-chain by calling `acceptTermsOfService`. The terms-of-service URL can be updated by the protocol operators using `updateTermsOfService`.

PoolAccessControl

The `PoolAccessControl` is used for both lenders and borrowers to be verified for the permissioned part of the protocol. It extends the `VeriteAccessControl` with a `ToSAcceptanceRegistry`. Only after accepting the terms, users can verify themselves. Additionally, it allows the Pool Admin to allow certain addresses without using the Verite system by calling `allowParticipant`.

PoolAdminAccessControl

The `PoolAdminAccessControl` contract is used for Pool Admins to be verified for the permissioned part of the protocol. It extends `VeriteAccessControl` with a `ToSAcceptanceRegistry`. Only after accepting the terms, Pool Admins can verify themselves.

Permissioned Contracts

The following extensions exist for the permissioned part of the protocol:

- `PermissionedPool` allows access to some of its state-changing functions only for lenders verified in the `PoolAccessControl` contract created upon initialization.
- `PermissionedLoan` allows access to some of its state-changing functions only for borrowers verified in the `PoolAccessControl` contract of the corresponding `Pool`.
- `PermissionedServiceConfiguration` exposes a new field that returns a `PoolAdminAccessControl` instance.
- `PermissionedPoolController` allows access to its state-changing functions only for Pool Admins verified in the `PoolAdminAccessControl` contract returned by the associated `ServiceConfiguration`.

Lenders and borrowers are verified through the same contract. Therefore, each lender can also be a borrower and vice-versa.

2.2.2 Snapshot algorithm

A crucial part of the protocol is the snapshot algorithm that is handled by the `WithdrawController`. Withdrawal requests are handled in the following manner:

- The runtime of a `Pool` is divided into periods of equal length.
- A `withdrawGate` is set upon `Pool` creation (e.g., 25%) that determines how many tokens of the currently available tokens (balance of the `Pool` minus assets that have already been marked as withdrawable) can be withdrawn in each period.
- A user can perform withdrawal requests up to the full amount of shares they possess.
- In the period following a withdrawal request, the shares become eligible for withdrawal. The eligible shares of all users combined are evaluated against the current assets and the `withdrawGate` and a portion is marked as redeemable. These shares can now be withdrawn by each user.
- If not all eligible shares become redeemable in a period, they will be evaluated again in the next period.
- Redeemable shares have a fixed exchange rate for assets that is determined at the end of a period. At this point, the shares are not accruing any more yield. This rate can differ from the rate during the withdrawal request as Loan payments still might arrive before the period ends.

Because withdrawal requests are likely scattered over different periods and can also be redeemed after an arbitrary number of periods, Circle developed an algorithm that allows to calculate a single user's redeemable shares at any period in constant time. This is achieved in the following way:

- Each withdrawal request is added to a global state.
- At the beginning of each period, the ratio of eligible shares to shares that become redeemable is calculated in this global state.
- This ratio is stored in a snapshot in a way that allows the application of ratios from multiple periods at once to a user withdrawal request when it is necessary (e.g., on another withdrawal request or an actual withdrawal).

This is illustrated by the following formula:

`ratio1 + (1 - ratio1) * ratio2 + (1 - ratio1) * (1 - ratio2) * ratio3 + ...`

`ratio1` indicates the ratio in period 1 etc. (in period 0, the ratio is 0 as no eligible shares are available yet). Each period, the previous part of the formula is saved into a snapshot:

- Period 1: `ratio1`
- Period 2: `ratio1 + (1 - ratio1) * ratio2`
- Period 3: `ratio1 + (1 - ratio1) * ratio2 + (1 - ratio1) * (1 - ratio2) * ratio3`
- ...

Additionally, the factor of the current period's ratio is saved separately:

- Period 1: `1`
- Period 2: `(1 - ratio1)`
- Period 3: `(1 - ratio1) * (1 - ratio2)`
- ...

Sums converge to `1 RAY` (i.e., 1e27) and factors converge to 0. As soon as the factor hits 0, it is reset to `1 RAY`. At this point, the sums will start to converge to `2 RAY`.

To apply all ratios of the periods from when a withdrawal request was created up until the current period, we can multiply the eligible shares with `sum of the current period + sum of the starting period and divide the result by factor of the starting period`. For example:

- `ratio1 = 0.5`
- `ratio2 = 0.25`
- `ratio3 = 0.5`
- `ratio1 + (1 - ratio1) * ratio2 + (1 - ratio1) * (1 - ratio2) * ratio3 = 0.8125`

If we want to find the redeemable shares in period 3 for 500 shares that were requested in period 1, we calculate: `(500 * (0.8125 - 0.5)) / 0.5 = 312.5`

The same can be achieved (but not in constant time) by applying the ratio for each period to the eligible shares:

- Period 1: Withdrawal request performed.
- Period 2: `500 eligible shares * 0.25 = 125`. 375 eligible shares remain, 125 shares are now redeemable.
- Period 3: `375 eligible shares * 0.5 = 187.5`. 187.5 eligible shares remain, 312.5 shares are now redeemable.

Additionally, each snapshot contains the period sums with the assets <-> shares exchange rate factored in so that the amount of withdrawable assets can also be easily calculated.

In **Version 2**, the described algorithm has been replaced: The ratio of each period is now directly stored in the period's snapshot. To bring user states up to date, eligible shares have to be multiplied with the ratio of each period between the last snapshot period the state was changed and the current period. Users have to manually update their states to be able to create further withdrawal requests or use all of their redeemable shares.

2.2.3 Fees

Perimeter defines a wide array of different fees to accommodate most use cases:

- **First Loss Fee**: This fee is set on the protocol level and is taken as a percentage of Loan payments. It is used to cover losses of defaulted loans. After a Pool has closed, the Pool Admin will receive these fees if they have not been used to cover.
- **Service Fee**: This fee is chosen by the Pool Admin for a given Pool and taken as a percentage of Loan payments. It is sent to the `feeVault` for later collection by the Pool Admin.
- **Origination Fee**: This fee is set for each Loan individually and is taken as a percentage of the Loan principal and paid on top of an interest payment. It is calculated for each year of the Loan duration and sent to the `feeVault` for later collection by the Pool Admin.
- **Late payment Fee**: This fee is a constant fee set for each Loan individually and is charged when a payment has not been made in time. In (Version 1), it is sent to the Pool.
- **Request Fee**: This fee is taken on each withdrawal request and is a percentage of the shares that are requested. The shares are burnt.
- **Request Cancellation Fee**: This fee is taken on each withdrawal cancellation request and is a percentage of the shares that are canceled. The shares are burnt.

2.2.4 Roles & Trust Model

Internally, four different roles are defined:

- `DEFAULT_ADMIN_ROLE` is assigned to the address that deploys the `ServiceConfiguration`. This address can then assign the remaining three roles freely.
- `OPERATOR_ROLE`: Operators can update various data in the `ServiceConfiguration`, including the whitelisted tokens, minimum First Loss and the First Loss Fee, as well as valid Loan Factories and Terms-Of-Service Registries. In the permissioned case, Operators can also set up the `PoolAdminAccessControl` for verification of Pool Admins.
- `PAUSER_ROLE`: Accounts assigned to this role can pause and unpause all the contracts created with the given `ServiceConfiguration` instance.
- `DEPLOYER_ROLE`: Deployers can upgrade all contracts created with the given `ServiceConfiguration` instance.

Circle claims that the Deployer role will be maintained only for critical security upgrades and might be discarded sometime in the future. New feature upgrades will be deployed using a set of new Factory contracts and implementations.

Using the `PoolFactory`, anyone can create a new `Pool`. Using the `PermissionedPoolFactory`, Verite verified users can create a new `Pool`. The Pool deployer automatically becomes the Pool Admin of the Pool. This role cannot be transferred to another account and comes with a set of enormous privileges that require full trust. Pool Admins can...

- ... set and change fees, including the fixed fee that allows them to directly retrieve any amount of funds out of the Pool.
- ... set the `withdrawGate`, allowing them to completely close withdrawals in active Pools.
- ... adjust the Pool Capacity, allowing them to disable deposits at any time.
- ... move the Pool end date to a prior date.
- ... withdraw the first loss amount after a Pool has closed and all Loans have ended.
- ... fund any Loan with the available resources in the Pool.
- ... default any loan on-chain.
- ... cancel Funded Loans after the `dropDeadTimestamp` has been reached.
- ... claim collateral of a Loan after it has defaulted.
- ... reclaim funds of Open Loans to the Pool.

Most notable is the ability of a Pool Admin to issue loans arbitrarily, theoretically allowing them to drain a Pool's funds and keep the tokens, as well as marking a Loan as defaulted and claiming the collateral.

If a Pool Admin loses their keys, on-chain defaults are not possible anymore which results in users not receiving First Loss in case a Loan is defaulted. Open Loan amounts that have not been withdrawn can also not be sent back to the Pool with `Loan.reclaimFunds`.

For this reason, Pool Admins have to be completely trusted. Circle, therefore, claims to have plans to deploy only the permissioned contracts on Mainnet in order to be able to verify the identity of Pool Admins thoroughly.

3 Limitations and use of report

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

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

4 Terminology

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

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

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

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

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

5 Findings

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

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

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

Critical-Severity Findings	0

High-Severity Findings	0

Medium-Severity Findings	1

- EIP-4626 Non-Compliance Code Partially Corrected

Low-Severity Findings	2

- ERC-4626 Donation Attack Acknowledged
- ToS Acceptance Registry Update Risk Accepted

5.1 EIP-4626 Non-Compliance

Correctness Medium Version 1 Code Partially Corrected

CS-CPER-001

A `Pool` implements the EIP-4626 Tokenized Vaults standard. Some code parts are, however, not fully compliant with the standard as can be seen in the following list:

- `convertToAssets` and `convertToShares` revert when `PoolLib.isSolvent` returns `false`. This is in violation of the requirement **MUST NOT revert unless due to integer overflow caused by an unreasonably large input**.
- `maxDeposit` and `maxMint` revert on `totalAvailableAssets > poolMaxCapacity` which violates the rule **MUST NOT revert**.
- `maxDeposit`, `maxMint`, `maxWithdraw` and `maxRedeem` do not return 0 when the `Pool` is paused. This behavior is not allowed under the **MUST factor in both global and user-specific limits, like if deposits are entirely disabled (even temporarily) it MUST return 0** requirement.
- `withdraw` and `redeem` require the `owner` parameter to be equal to `msg.sender`. This makes the scheme required by the rule **MUST support a withdraw flow where the shares are burned from owner directly where msg.sender has EIP-20 approval over the shares of owner.** not possible.
- The first parameter of the events `Withdraw` and `Deposit` is named `caller` while the standard requires it to be named `sender`.
- `PermissionedPool.maxWithdraw` and `maxRedeem` are not checking permissions. Since permissions can invalidate after some time, the functions violate the requirement **MUST factor in both global and user-specific limits**.

Code corrected:

- `maxDeposit` and `maxMint` no longer revert on underflow.

- `maxDeposit`, `maxMint`, `maxWithdraw` and `maxRedeem` now return 0 when the Pool is paused.
- The `Withdraw` and `Deposit` are now emitted with the correct parameter naming.
- `PermissionedPool.maxWithdraw` and `maxRedeem` are now correctly checking permissions.

Code not corrected:

- `withdraw` and `redeem` still don't support EIP-20 approval for `owner`.

Risk accepted:

Circle accepts the risk of the `convertToAssets` and `convertToShares` non-compliance, stating:

```
We implemented nearly all the changes recommended in the finding, except for the convertToAssets()/shares() functions
reverted on Pool insolvency. Since that’s essentially a terminal state for the Pool, and the code change being non-trivial,
we opted to leave it as-is.
```

5.2 ERC-4626 Donation Attack

Security | Low | Version 1 | Acknowledged

CS-CPER-002

The `Pool.deposit()` function is susceptible to a frontrunning attack that involves donations of the underlying token. Consider the following example:

1. A new `Pool` has been deployed.
2. An attacker deposits 1 wei of the liquidity asset into the pool and mints 1 share.
3. A regular user deposits 10000 full tokens of the liquidity asset into the pool.
4. Before the transaction of the user is executed, the attacker executes another transaction in which they transfer 5000 full tokens directly onto the pool.
5. The user now receives 1 share for the 10000 tokens deposited. The attacker also holds 1 share while they only deposited 5000 tokens (+ 1 wei).
6. The attacker profited 50%.

Since the attacker is not able to instantly withdraw the funds, this attack can be noticed and necessary steps (e.g., change of `withdrawGate` or removal of the necessary permissions of the attacker) can be taken before any real danger occurs.

Acknowledged:

Circle has acknowledged the issue.

5.3 ToS Acceptance Registry Update

Design | Low | Version 1 | Risk Accepted

CS-CPER-003

`PoolAccessControlFactory.create` sets the `tosAcceptanceRegistry` in the newly created `PoolAccessControl` contract from the value in `ServiceConfiguration`. This value cannot be updated anymore. The value, however, can be updated in `ServiceConfiguration`. In `PermissionedPool`, the `poolAccessControl` address cannot be updated either. This means, if the `ToSAacceptanceRegistry` ever changes to a new address, permissioned pools can only be updated by updating the beacon implementation of all `PoolAccessControl` instances.

Risk accepted:

Circle accepts the risk with the following statement:

```
We have no plans to introduce separate ToS Acceptance Registries, so it feels premature to build around that right now. We accept
the risk given that in a worst-case scenario, we could upgrade PoolAccessControl contracts to point to a new registry if needed.
```

6 Resolved Findings

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

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

Critical-Severity Findings	1

- cancelFunded Counts Assets Twice Code Corrected

High-Severity Findings	2

- Withdrawal DoS Code Corrected
- feeVault Stuck Funds Code Corrected

Medium-Severity Findings	4

- Full Cancel Request Not Possible Code Corrected
- Late Fees Do Not Go to First Loss Vault Code Corrected
- Missing Permission Checks Code Corrected
- paymentDueDate Updated After Last Payment Code Corrected

Low-Severity Findings	6

- Callback State Not Used Code Corrected
- Inconsistent State After Withdrawal Cancellation Code Corrected
- Missing Sanity Checks Code Corrected
- Pool Tokens Not Transferable Specification Changed
- completeFullPayment Return Value Code Corrected
- onlyPoolAdmin Modifier Code Corrected

6.1 cancelFunded Counts Assets Twice

Correctness Critical Version 1 Code Corrected

CS-CPER-024

`Loan.cancelFunded` does not call `Pool.onLoanPrincipalReturned`.

This means that `outstandingPrincipals` will not be reduced by the loan amount and the loan will be counted twice in `totalAssets`.

This will increase the value of a pool share, allowing lenders to withdraw more assets than they deposited, leading to the Pool becoming insolvent.

Code corrected:

LoanLib's `returnCanceledLoanPrincipal` now calls `Pool.onLoanPrincipalReturned`.

6.2 Withdrawal DoS

Correctness High Version 1 Code Corrected

CS-CPER-026

The snapshot algorithm creates snapshots in each period containing aggregation sums and differences (as pointed out in the System Overview). Over time, the sums converge to 1 RAY while the differences converge to 0. When the difference hits the 0 value, it is set to 1 RAY resulting in the whole process starting from the beginning (the sums now converge to 2 RAY).

As the difference converges to 0, rounding errors intensify. If the difference is exactly 1, rounding errors approach 100%: Eligible shares in the global state are converted to redeemable shares according to the withdraw gate, while in the user state, no eligible shares are converted at all.

In the next call to `WithdrawController.simulateSnapshot`, the user's eligible shares are then multiplied by the difference between the aggregation results (in RAY), multiplied with 1 RAY, divided by 1 and then divided by 1 RAY, leaving a number for redeemable shares that is orders of magnitude higher than the actual requested shares of the user:

```
uint256 sharesRedeemable = withdrawState.eligibleShares.mul(
    endingSnapshot.aggregationSumRay - offsetSnapshot.aggregationSumRay
);
sharesRedeemable = sharesRedeemable
    .mul(offsetSnapshot.aggregationDifferenceRay > 0 ? PoolLib.RAY : 1)
    .div(
        offsetSnapshot.aggregationDifferenceRay > 0
            ? offsetSnapshot.aggregationDifferenceRay
            : 1
    )
    .div(PoolLib.RAY);
```

The function then reverts on buffer underflow in the following section:

```
withdrawState.eligibleShares -= sharesRedeemable;
```

All functions (including `Pool.withdraw`) that calculate the user's withdrawal state will revert from this point on.

This issue can be exploited by an attacker, with low cost:

To achieve a difference of exactly 1, an attacker has to perform a few withdrawal requests with amounts below the withdrawal gate. The decimals of the amounts used in the request must sum up to 27 (the decimals of RAY). Depending on the withdrawal gate and the deposited amounts, a difference of 1 can be achieved in a few (~three) withdraw periods. The next withdrawal in any following period will result in at least one user being unable to withdraw. All users performing withdrawal requests in the same period will be affected. Withdrawal requests in the following periods will work as expected again.

Consider the following example:

- The withdrawal gate is 25%.
- The token has 6 decimals.
- User 1 deposits 40,000 tokens.
- User 2 deposits 40,000 tokens.
- An attacker deposits 20,010 tokens.
- In period 1, the attacker requests a withdrawal of 10,000 tokens.
- In period 2, the attacker requests a withdrawal of 10,000 tokens.

- In period 3, the attacker requests a withdrawal of 10 tokens minus 2 wei.
- The difference is now 1 and the aggregation sum is 99999999999999999999999997.
- The attacker withdraws their whole balance.
- A few periods pass without any interaction.
- In period 10, user 1 requests a withdrawal of 40,000 tokens.
- In period 11, user 1 still has 0 withdrawable assets due to the rounding error.
- Starting from period 12, interactions with the contracts revert for user 1.
- In period 12, user 2 can request a withdrawal and everything works out as expected.

The attacker has performed a DoS attack on a subset of the protocol's users for a low cost: They paid only the gas fees and the pool's request fees. If User 2 had also tried to withdraw in period 10, their funds would be stuck too. If the attacker performs the attack in the last period before the end date of the pool, chances are high that a large amount of users are affected as they try to withdraw immediately after the pool closes. The attacker could repeat the attack by depositing again and making more withdrawal requests.

Code corrected:

The snapshot algorithm has been replaced by a mechanism that saves the conversion rates of eligible shares to redeemable shares individually. Users that want to request a withdrawal (or cancel requested shares) are now required to manually update their state by calling `claimSnapshots` if their last action was performed in a past period and their current state contains some eligible shares.

6.3 feeVault Stuck Funds

Correctness **High** **Version 1** **Code Corrected**

CS-CPER-019

`Pool.initialize` creates a `feeVault`. The owner of the `Vault` is set to the `Pool`. Only the owner can withdraw funds from the `Vault`.

As `Pool` does not contain any function that withdraws from the `feeVault`, any funds sent to it will be stuck.

Code corrected:

A function `withdrawFeeVault` has been added to the Pool, which can be called by the Pool Admin through the `PoolController` to withdraw from the Fee Vault.

6.4 Full Cancel Request Not Possible

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

CS-CPER-020

A discrepancy between `Pool.maxRequestCancellation` and `cancelRedeemRequest` / `cancelWithdrawRequest` leads to users not being able to cancel all of their requested (but not yet redeemable) shares. Consider the following example:

- A user holds 500 shares on a Pool.
- Request fee is 0% and request cancellation fee is 10%.
- The user requests a withdraw for 500 shares. His requested shares are now 500.

- The user now decides to cancel the full withdrawal request.
- `maxRequestCancellation` returns 450.
- The user calls `cancelRedeemRequest` with 450 shares.
- 45 shares are burned, 450 shares are removed from the withdrawal request.
- 5 tokens remain in the withdrawal request because the mentioned functions calculate fees in a different way.

Code corrected:

`PoolLib.calculateMaxCancellation` now returns all requested / eligible shares (ignores fees). When executing the cancellation, fees are burned, and then the requested amount is deducted from the withdrawal state.

6.5 Late Fees Do Not Go to First Loss Vault

Correctness Medium Version 1 Code Corrected

CS-CPER-021

The late payment fees are supposed to be a fixed amount that goes to the first loss vault.

However, they are instead paid to the Pool in `completePayment`.

Additionally, the documentation incorrectly states that "Late fees are [...] a percent of the payment amount on interest." This is incorrect, as late fees are a fixed amount. In particular, if multiple late payments are made at once using `completeFullPayment()`, the fee is only charged once.

Code corrected:

Late Payment Fees are now transferred to the First Loss Vault instead of the Pool.

6.6 Missing Permission Checks

Correctness Medium Version 1 Code Corrected

CS-CPER-023

Contracts in the `permissioned` directory extend the base contracts by adding permission checking using the Verite protocol. This is done by overriding dummy modifiers of the base contracts. As Verite identities can expire, permissions have to be checked on each interaction. This is, however, not enforced in all parts of the base contracts:

- `Loan.cancelFunded` does not check borrower and admin permissions.
- `Loan.claimCollateral` does not check borrower and admin permissions.
- `Loan.reclaimFunds` does not check admin permissions.

Code corrected:

`Loan` and `PoolController` have been refactored. The Pool Admin now only interacts with the `Loan` via the `PoolController`, which enforces permissioning. It was clarified that permissions should not be checked for the borrower on `Loan.cancelFunded`.

6.7 paymentDueDate Updated After Last Payment

Correctness Medium Version 1 Code Corrected

CS-CPER-025

The `paymentDueDate` in `Loan` is updated after the last payment made through `completeNextPayment`. This is incorrect as the loan should end in the period of the last payment. The final `paymentDueDate` is one period after the loan end date. Repayment of principal within this period will not be considered late, meaning there will be no late fee charged even though there should be.

Code corrected:

`paymentDueDate` is now only incremented if `paymentsRemaining` is larger than zero.

6.8 Callback State Not Used

Design Low Version 1 Code Corrected

CS-CPER-015

The `LifeCycleState` enum contains the `Callback` state. This state is never used, as callbacks are not enforced on-chain in the current version.

Code corrected:

The `Callback` state has been removed.

6.9 Inconsistent State After Withdrawal Cancellation

Correctness Low Version 1 Code Corrected

CS-CPER-022

`WithdrawController.performRequestCancellation` performs a `PoolLib.calculateWithdrawStateForCancellation` update on both the requesting user's state and the global state. The function first tries to match all requested shares before matching eligible shares. If multiple users have open requested shares, the values differ between global and user state. This leads to an inconsistency: If the user's cancellation request removes eligible shares, the global state will have more requested shares removed. Consider the following example:

- User 1 has requested 500 shares and 500 shares are already eligible.
- User 2 has requested 500 shares and 500 shares are already eligible.
- The global state, therefore, has 1000 requested shares and 1000 eligible shares.
- User 1 now cancels 1000 shares.
- 0 requested shares and 1000 eligible shares remain in the global state.
- 500 requested shares and 500 eligible shares remain in user 2's state.

As soon as a new period starts, the data matches again.

While we have found no issues arising from this inconsistency, third party protocols might rely on the data.

Code corrected:

The changes in user state are now saved in memory, so that an equal amount can be removed from the global state.

6.10 Missing Sanity Checks

Correctness | Low | Version 1 | Code Corrected

CS-CPER-016

The following checks are not performed, leading to misconfiguration possibilities:

- `LoanFactory.createLoan` allows the creation of a Loan that does not match the `liquidityAsset` of its `Pool`.
- `PoolFactory.createPool` does not check that `serviceFeeBps` is lower than 10,000.
- `Pool.redeem` does not revert with an error message when `maxRedeem` is reached (as opposed to `Pool.withdraw`).
- `VaultFactory.createVault` does not revert with error when beacon implementation is not set.
- `WithdrawControllerFactory.createController` does not revert with error when beacon implementation is not set.

Code corrected:

All aforementioned problems have been resolved.

6.11 Pool Tokens Not Transferable

Correctness | Low | Version 1 | Specification Changed

CS-CPER-027

The `_beforeTokenTransfer()` function of Pool has been overwritten to disallow token transfers.

This is a mismatch with the specification, which states:

```
Pool Tokens are transferable, but to redeem the token back the new Pool Token holder will still need to comply with the pool’s lender access requirements.
```

Specification changed:

The documentation has been updated to clarify that Pool Tokens should not be transferable.

6.12 completeFullPayment Return Value

Correctness | Low | Version 1 | Code Corrected

CS-CPER-017

`Loan.completeFullPayment` returns `payment` even when a different amount has been paid.

Code corrected:

The return values for both `completeFullPayment` and `completeNextPayment` have been removed.

Circle stated that an event may be added in a future version to support easier off-chain accounting.

6.13 onlyPoolAdmin Modifier

Design | Low | Version 1 | Code Corrected

CS-CPER-018

In `Loan`, all Pool Admin functions are accessed through the `PoolController`, except `reclaimFunds` and `markCallback`. For these, the `onlyPoolAdmin` modifier is used which allows the Pool Admin to call the `Loan` contract directly. The interface is not consistent.

Code corrected:

The functions `reclaimLoanFunds`, `claimLoanCollateral`, `cancelFundedLoan` and `markLoanCallback` have been refactored so that they can only be accessed through the `PoolController`.

7 Informational

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

7.1 Events Could Be More Informative

Informational | Version 1

CS-CPER-004

This is a collection of events that could benefit from containing more information. The list contains examples and is non-exhaustive:

- PoolController emits the `PoolSettingsUpdated` event. This event is emitted by 4 different functions and contains no information about which values were changed or what the old and new values are.
- PoolLib emits the `FirstLossApplied` event. It contains the `loan` address and `firstLossRequired`. However, it does not contain the `outStandingLoanDebt`. This means the event is not sufficient to know whether the `firstLoss` vault had sufficient funds to cover the defaulted loan or if the loss was socialized among lenders.
- Some setters (e.g., `ServiceConfiguration.setPaused`) emit events even when the value of the respective storage variable is not changed.
- `Pool._performRedeemRequest` emits an event with `shares` and `assets`. The amount of `assets`, however, is non-conclusive at this point, since it can be higher if another loan repayment occurs before the period ends.

7.2 Inconsistent Naming

Informational | Version 1

CS-CPER-005

`Pool` contains multiple functions that are called by `PoolController`. Their naming, however, does not seem to follow a common scheme. For example, `PoolController.defaultLoan` calls `Pool.onDefaultedLoan`, while `PoolController.fundLoan` calls `Pool.fundLoan`.

7.3 Missing Events

Informational | Version 1

CS-CPER-006

Some events are missing. We list some examples here. Note that this list may be incomplete:

- `markCallback()` is used to have on-chain evidence of the timestamp at which a callback of an open loan was initiated. Currently this only sets a storage variable, but does not emit an event.
- `ILoan` defines the `LifeCycleStateTransition` event. It is only emitted in `Loan.markDefaulted`. The event is missing in all other functions of `Loan` that change the Lifecycle State. Note that `IPool` also defines an event with the same name, which may be confusing.

- The `PoolAccessControlFactory` does not emit an event when a proxy is created, the other factories do.
- The `PoolSettingsUpdated` event in `PoolLib` exists but is never emitted.

7.4 Possible Gas Optimizations

Informational **Version 1**

CS-CPER-007

The following code parts can be optimized for gas efficiency. The list is non-exhaustive:

- Redundant storage reads are performed in various places, for example:
  - `Loan.postFungibleCollateral` (as well as many other functions in `Loan`) possess the actual value of `_state` (because it has just been written) and still return the state by reading from storage.
  - `Loan.fund` loads the `_state` two times in a row from storage.
- Redundant storage writes are also performed in various place, for example:
  - `Loan.postFungibleCollateral` writes to `_state` even though the contents of `_state` are already known and might be identical to the new value.
  - `WithdrawController.performRequest` sets the `latestSnapshotPeriod` of the user state to the `latestSnapshotPeriod` of the global state although this has already been done in the prior `snapshotLender` call.
  - `ToSAcceptanceRegistry._termsSet` is set on an update to `_termsOfService`. A length check of `_termsOfService`, however, is sufficient to determine whether the variable was set.
- External calls are more expensive than internal calls. Redundant external calls are performed in various places, including:
  - The `RAY` constant is read from `PoolLib` and `LoanLib` many times. Reading a constant from an external library requires an extra `delegatecall` each time.
  - Some `Pool` settings are only used in `Pool` (e.g., `requestFee`, `requestCancellationFee`) but are stored in `PoolController`. Accessing these variables (and their respective calculation functions) requires an external call every time.
  - `Pool.onActivated` is called by the `PoolController`, then proceeds to call `PoolController.settings` while the settings could have just been passed directly to the function.
  - Some limited functionality (3 lines of code) is extracted to external libraries, e.g., `PoolLib.executeFirstLossWithdraw`. This barely helps with deployment costs but requires an additional external call.
  - `PoolController.withdrawFirstLoss` calls `Pool.firstLossVault`, which in turn calls `PoolController.firstLossVault`.
- Redundant calculations can be found in the following places:
  - `Pool._performRedeemRequest` calls `WithdrawController.maxRedeemRequest` which calculates `_currentWithdrawState`. Then, `WithdrawController.performRequest` is called which calculates `_currentWithdrawRequest` again.
  - `WithdrawController.performRequest` calculates `_currentWithdrawState` with a state that has already been updated by the prior call to `snapshotLender`.
  - `WithdrawController.snapshot` calculates the `withdrawPeriod`, then calls `_currentGlobalWithdrawState` which calculates the `withdrawPeriod` again.

- The initializer of Loan sets the `_state` to `Requested`. Since `Requested` is item `0` in the enum, this is already the default value and is unnecessary. The same is the case in `PoolController.initialize` which sets the Pool state to `Initialized` even though it is the default value.
- `Pool.onlyPoolController` checks that `poolController` is not the 0-address and that it is `msg.sender`. The first check is redundant.
- Some contracts (e.g., `Pool`) use `transferFrom` with the `from` address set to themselves. To make this work, they also set an approval to themselves beforehand. A simple `transfer` would be sufficient.
- In `IPoolAccountings`, the fields `totalAssetsDeposited`, `totalAssetsWithdrawn`, `totalDefaults` and `totalFirstLossApplied` are written but never read. Off-chain accounting can also be achieved with events.
- `LoanLib.postFungibleCollateral` inserts new collateral addresses into the `collateral` state variable in non-constant time (as opposed to e.g., an `EnumerableSet`).
- `PoolLib.calculateWithdrawStateForCancellation` could return early in the first condition.
- `PoolLib.calculateMaxCancellation` uses `Math.max` on an unsigned integer and 0 and is thus redundant.
- `WithdrawController.simulateSnapshot` could return early on `eligibleshares == 0` or `endingSnapshot == offsetSnapshot`.

In Version 2, the following gas optimization can be achieved:

- Each `IPoolSnapshotState` occupies 4 slots in storage. As `redeemableRateRay` is not larger than RAY, `fxRateRay` is not larger than several multiples of RAY, `sharesRedeemable` is not used anywhere in the code and any amount of periods can easily be captured in 40 bits, the whole struct could be reduced to 1 word per snapshot:
  - `uint108 redeemableRateRay`
  - `uint108 fxRateRay`
  - `uint40 nextSnapshotPeriod`

7.5 Shadowed Variable

Informational **Version 1**

CS-CPER-008

`Vault.initialize` uses a parameter `owner` that shadows the `owner` function in `OwnableUpgradeable`.

7.6 Snapshot Restricted to Admin in PoolController

Informational **Version 1**

CS-CPER-009

`Pool.snapshot` can be called by anyone. `PoolController.snapshot` calls `Pool.snapshot` but is restricted to pool admin access. The function could be safely removed or given public access.

7.7 Spelling Errors

Informational **Version 1**

CS-CPER-010

Some code comments inside the contracts contain spelling errors. Here are some examples:

- The parameter `liquidityAsset` in `Pool.initialize` is an `asset held by the poo`.
- `Pool.liquidityPoolAssets` contains the following comment: `do not include any loan principles`.
- `LoanLib.payFees` contains the following comment: `This include both the service fee and origiantion fees`.

7.8 Unused Code

Informational **Version 1**

CS-CPER-011

- `PoolController.isInitializedOrActive` and `isActiveOrClosed` are not used and do not have value for external parties.
- `ILoanLifeCycleState.Callback` is never used.
- `ServiceConfiguration.protocolFeeBps` is set to 0 and never used.

7.9 Withdrawal Request Interface

Informational **Version 1**

CS-CPER-012

The following functions have been added as an extension to the existing EIP-4626 interface:

- `maxRedeemRequest`
- `maxWithdrawRequest`
- `maxRequestCancellation`
- `previewRedeemRequest`
- `previewWithdrawRequest`
- `requestRedeem`
- `requestWithdraw`
- `cancelRedeemRequest`
- `cancelWithdrawRequest`

The functions should roughly mimic their counterparts from the EIP. However:

- The standard defines that `withdraw` and `redeem` burn exactly `shares` from a user. `requestWithdraw` and `requestRedeem` make `shares` available for later withdrawal and burn additional shares as a fee.
- Referencing the previous point, `previewWithdrawRequest` and `previewRedeemRequest` subtract the fee from the input parameter. The rule **MUST return as close to and no more than the exact amount of assets that would be withdrawn in a redeem call in the same transaction** is therefore violated.

- The current interface of `requestWithdraw` and `requestRedeem` is now inconsistent with `cancelWithdrawRequest` and `cancelRedeemRequest`, which handle the `shares` argument in the same way as the standard.
- In Version 2, functions that rely on the current state of the user (`maxRedeemRequest` and `maxRequestCancellation`) should return 0 if `claimRequired` is true for the given user address.

7.10 Withdrawn Collateral Is Shown in View Functions

Informational **Version 1**

CS-CPER-013

When a borrower posts collateral, the collateral's address (and ID for NFTs) is added to a storage array. When withdrawing collateral, it is not removed from storage. The storage can be read using the `fungibleCollateral()` and `nonFungibleCollateral()` view functions.

For example, a borrower may have posted a certain ID of an NFT collection as collateral. After the collateral has been withdrawn, calling `nonFungibleCollateral()` will still return the address and ID of the NFT, even though the contract no longer owns it. This may be different from expected behavior.

7.11 Wrong Inline Comment

Informational **Version 1**

CS-CPER-014

`PoolLib.calculateWithdrawStateForCancellation` contains a comment that reads `ensure the "latestRequestPeriod" is set to the current request period`. This is, however, never done in the function.

The function is always called in a context where the `latestSnapshotPeriod` is already updated.

8 Notes

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

8.1 Borrower Can Withdraw Additional Tokens

Note Version 1

Tokens sent to the `collateralVault` of a `Loan` in error by other users can be redeemed by the borrower after the loan has matured.

8.2 Fees Can Be Changed During Runtime

Note Version 1

The `PoolController` allows a pool admin to change fees during the runtime of a pool:

- Request fees can be changed in `Initialized` state only.
- Request cancellation fees can be changed in `Initialized` state only.
- Service fees can be changed at any time.
- Fixed fees can be changed at any time.

Note that there is no maximum on how high fixed fees can be.

8.3 Instant Withdrawal After Deposit

Note Version 1

Instant withdrawal after deposit can lead to loss (on top of the withdrawal fee). Users that deposit after the start of a period are subject to expected interest that has accrued but has not been paid yet. On deposit, the resulting amount of shares for a given amount of assets is calculated taking this expected interest into account. This means, an instant withdrawal after a deposit leads to a loss as assets to shares are calculated without expected interest.

Furthermore, existing lenders experience an instant increase in value per share after another user deposits with expected interest.

As soon as the expected interest is actually paid, all ratios normalize.

8.4 PoolController Approvals

Note Version 1

`PoolController.depositFirstLoss` calls `transferFrom` on the liquidity asset with a `from` address supplied by the caller. If any user gives approvals to the contract, their funds can therefore be transferred to the First Loss Vault by the Pool Admin.

Note that there is no reason for a user to give approval to the PoolController, it would only happen accidentally.

8.5 Proxy Deployment

Note Version 1

`ServiceConfiguration` and `ToSAcceptanceRegistry` should always be deployed using the `upgradeToAndCall` mechanism of the used UUPS proxy to ensure that the initializer cannot be frontrun.

8.6 Request Fee Paid in Shares

Note Version 1

Request fees and request cancellation fees are paid by burning users' shares. This means that the value of the burned shares is distributed among all users of the platform. In the case where only one single user is using a Pool, the fees are therefore non-existent.

8.7 Snapshot Every Period

Note Version 1

Withdrawal requests are processed at the beginning of each period. If a period is skipped due to inactivity on the contracts, withdrawal requests will also not be processed in this period. Users that have open withdrawal requests have to make sure that `Pool.snapshot` or any other function that triggers the snapshot mechanism is called at least once per period. Otherwise, it may take longer until they can withdraw their full amount.

8.8 ToSAcceptanceRegistry Not Versioned

Note Version 1

The `ToSAcceptanceRegistry` allows an operator to update its terms of service URL. This means it is assumed that acceptances are automatically given to changes in the ToS at a later time.

8.9 withdrawPeriods After Close

Note Version 1

`WithdrawController.withdrawPeriod` calculates the current period the following way:

```
(currentTimestamp - activatedAt) / withdrawalWindowDuration;
```

The `withdrawalWindowDuration` possibly decreases after a `Pool` enters `Closed` state, resulting in suddenly inflated period numbers.