PUBLIC

# Code Assessment

## of the Vesu Protocol Smart Contracts

June 07, 2024

Produced for

by

# Contents

# 1 Executive Summary

Dear Vesu Team,

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

VESU implements a fully permissionless DeFi lending protocol. Anyone can deploy and configure a pool. A core contract called Singleton holds all funds and manages all pools. All operations go through the Singleton, each pool has an extension which is called before/after any operation and defines the values for the operation. A default extension is provided, arbitrary extensions and/or misconfigured parameters can break their respective pools without affecting the rest of the protocol.

The most critical subjects covered in our audit are the isolation of the pools, asset solvency and functional correctness.

The general subjects covered are usability, oracle security, access control, adherence to the specification and general design issues.

All issues uncovered during the review process have been addressed with suitable fixes. We believe the codebase to have a satisfactory level of security. The high complexity and extensibility of the project present a large attack surface. VESU internally relies primarily on one smart contract developer which, even though supported by external reviewers, limits the ability for internal QA. During the audit timeline, significant improvements in design and overall code quality have been achieved, but some novel issues and regressions remained present during the last review cycle. In our experience, these factors combined present an elevated risk of undiscovered vulnerabilities in the current codebase.

Continuing to allocate sufficient time and resources, strengthening the robustness of the design, and introducing internal security-focused quality assurance practices such as thorough unit- and regression-testing can significantly increase the level of security of the codebase and our confidence in it.

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 | 5 |
| • Code Corrected | 5 |
| Medium-Severity Findings | 12 |
| • Code Corrected | 11 |
| • Risk Accepted | 1 |
| Low-Severity Findings | 10 |
| • Code Corrected | 9 |
| • Risk Accepted | 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 Vesu Protocol 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 | 13 February 2024 | 9331d91395a9f5181939969d357671378970cf96 | Initial Version |
| 2 | 17 March 2024 | 65b8687973ae609c11061ae20403ae858bfe3e4c | After Intermediate Report |
| 3 | 20 May 2024 | d825c4bfd824949e98802b25a3b975f0810f376e | Extended Functionality |
| 4 | 31 May 2024 | 306d0164ff991c58a670e18ac748eb14dad7d223 | Fixes |

For the cairo smart contracts, the compiler version `2.5.3` was chosen. After the intermediate report the compiler version was updated to `2.5.4`.

The following contracts are in the scope of this review:

```
src/singleton.cairo
src/units.cairo
src/packing.cairo
src/math.cairo
src/map_list.cairo
src/lib.cairo
src/data_model.cairo
src/common.cairo

src/vendor/erc20.cairo
src/vendor/pragma.cairo

src/extension/interface.cairo
src/extension/default_extension.cairo
src/extension/components/interest_rate_model.cairo
src/extension/components/position_hooks.cairo
src/extension/components/pragma_oracle.cairo
```

In Version 2 the following file has been added:

```
src/extension/components/fee_model.cairo
```

In Version 3 the following file has been added:

```
src/extension/components/tokenization.cairo
src/vendor/erc20_component.cairo
src/v_token.cairo
```

The audit covers the Singleton (core of the protocol) with pools using the default extension as well as the functional correctness of the Singleton in conjunction with other valid extensions. Any such extension must be audited separately. It is known that pools with arbitrary extensions can break.

### 2.1.1 Excluded from scope

Vesu is a permissionless protocol which allows anyone to deploy and configure a pool. A pool has several configuration parameters including token addresses, ltvs and liquidation discounts. Furthermore each pool is connected to an extension, a smart contract implementing several hooks.

The configuration of a pool and the deployment of pools with arbitrary extensions is out of scope. Individual issues in the report covering such scenarios do not imply a thorough investigation or complete coverage of those areas. These mentions are intended solely to highlight particular findings of interest and should not be construed as a comprehensive examination of all possible scenarios outside the defined scope of this report.

All contracts not explicitly in scope are considered out-of-scope.

Economic attacks to the protocol are beyond the scope of this review.

## 2.2 System Overview

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

At the end of this report section we have added subsections for each of the changes accordingly to the versions.

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

VESU offers a DeFi lending protocol consisting of a general singleton with a default extension, which runs in a fully permissionless and immutable manner. The singleton holds pooled liquidity and new pools can be created by anyone with any customized extension as plug-in. The default extension features an adaptive interest rate model and an autonomous fail-safe mechanism.

### 2.2.1 Singleton

The singleton manages the collateral and debt accounting with a pooled liquidity design: the accounting of individual pools is isolated while the singleton holds all the liquidity across all the pools. Price oracles, interest accrual, and liquidation mechanisms, among other features, are delegated to the extensions. Callbacks to the extension are implemented before / after position modification and liquidation.

#### 2.2.1.1 Pool Creation

A pool can be created `create_pool()` permissionlessly by providing an extension address with assets and pairs configurations. Each pool is uniquely identifiable by its pool id which is derived from the creator address and its nonce.

#### 2.2.1.2 Modify Position

`modify_position()` enables users to manipulate a position by adding / removing collateral / debt. It provides a general entrypoint where the modification of both collateral and debt could be batched in one call. The input amount of collateral / debt to increase or decrease could be expressed in delta or target and in terms of `Asset` (token amount) or `Native` (collateral shares / nominal debt).

1. At the beginning of the execution, a callback (`before_modify_position()`) will be made to the extension, with the user input and a context containing the asset configuration, the current asset price and the user position. Note the user intended modification amount will be overwritten by the return value of the callback.
2. The context will be reloaded after the callback as it may have been changed by the actions in the callback.
3. Then singleton's state change happens where the input amount will be deconstructed to delta and applied to user's position and asset accounting. Two invariants are checked at this step:
    - The underlying asset delta should be non-zero if and only if the shares delta is non-zero.
    - The collateral / debt value should be 0 or above a floor.
4. In case the position will become less collateralized after the operation (increasing debt or decreasing collateral), invariants and access rights are checked: The position must be collateralized sufficiently, the call must originate either from the owner or a delegatee and the `max_utilization` must be respected after the call. Notably `max_utilization` of a collateral must be respected even when one retrieves it's own collateral.
5. At the end of the execution, based on the actual modification of the position, assets are transferred between the user and the singleton. A final callback (`after_modify_position()`) to the extension is made which must succeed / return true for the execution to terminate successfully.

The default extension provided will update the shutdown status of the pool and check if the operation violates the restrictions of each status (see details in Default Extension).

#### 2.2.1.3 Transfer Position

Transfer position is provided to enable transfer of debt / collateral between two positions within the same pool. The debt / collateral token must match in case the amount to change is non-zero. Two subsequent internal position modification are done, the first based on the user supplied parameters, the second based on the change of the first.

#### 2.2.1.4 Liquidate Position

Liquidation is possible if a user position's LTV (Loan to Value) of a token pair has fallen below the liquidation threshold. The actual liquidation logic is delegated to the extension, which will return the amount of collateral that is released, debt that is repaid and bad debt that will be immediately redistributed to all the existing collateral asset liquidity providers. After the update of the position and settlement of the asset, a callback `after_liquidate_position()` will be made to the extension.

The following actions are further supported:

- `flash_loan()`: users can flash loan any asset held by the singleton.
- `modify_delegation()`: a user can delegate all its positions in a specific pool to a delegator, who will have full control as the user.
- `donate_to_reserve()`: anyone can donate to the pool to increase the reserve of a configured asset.
- `retrieve_from_reserve()`: this function can only be called by the extension to retrieve asset (limited by the reserve) from the pool to any receiver directly.

In addition, the following view functions are available:

- Getters of the pool state variables: `creator_nonce()`, `pools()`, `asset_configs()`, `max_position_ltv()`, `delegations()`.
- Estimation of the up-to-date accounting variables: `context()`, `positions()`, `rate_accumulator()`.
- Other helper functions: `calculate_pool_id()`, `calculate_debt()`, `calculate_nominal_debt()`, `calculate_collateral_shares()`, `calculate_collateral()`, `deconstruct_collateral_amount()`, `deconstruct_debt_amount()`.

### 2.2.2 Default Extension

A default extension is provided which uses pragma oracles and implements an adaptive interest rate model with an autonomous fail-safe mechanism.

#### 2.2.2.1 Callbacks

The mandatory callbacks are implemented:

- `before_modify_position()`: checks the caller is singleton and returns the unmodified collateral and debt.
- `after_modify_position()`: updates and checks the pool's shutdown status.
- `before_liquidate_position()`: computes the debt to repay, collateral to release, and bad debt that will be redistributed.
- `after_liquidate_position()`: simply returns true.

#### 2.2.2.2 Interest Rate Model

Implements an adaptive interest rate model where the evolution of interest rate is based on the parameters set upon the pool creation. The adaptive interest rate model is an utilization-based curve with a half-life growth (decay) rate controller, which adjusts the full utilization rate over time. In each update of the position, the interest will be accrued and the interest rate, utilization ratio, and full utilization rate will be adjusted.

#### 2.2.2.3 Price feed

The price data is fetched from the Pragma Oracles. For each asset, a triple (key, timeout, sources) is configured:

- The raw price will be the median value (`get_data_median()`) queried from a specific spot entry key.
- The raw price will be rescaled to 18 decimals.
- The price will be invalid in any of the following cases:

    1. The timeout is set and has elapsed since the last updated timestamp.
    2. The threshold of sources is set and is less than the aggregated number of sources that produce this price.

#### 2.2.2.4 Pool Emergency Shutdown

As there is no governance interference to pause and unpause pools in an emergency, the default extension employs an autonomous fail-safe mechanism to limit the risk of losses accumulated by the pool. The extension will pause and eventually shut down the pool in case of problems, e.g. if any of the pool's lending pairs falls into insolvency.

The pool could be in one of the four shutdown modes:

- None: The pool functions correctly and no pair falls into insolvency.
- Recovery: In case any pair of the pool is insolvent, it has the chance to recover during the recovery period. In this period, users can only increase collateral or decrease debt to increase the pool's overall health factor. Liquidation is also allowed in recovery mode.
- Subscription: In case any insolvent pair or another condition triggering the shutdown hasn't recovered within the recovery period, the pool will gradually wind down. User can only decrease their debt in this subscription period to free their collateral in the redemption phase.
- Redemption: Now user can only withdraw their collateral and the pool is shutdown.

At the end of every position modification, the status of the pool will be inferred and updated according to:

- The status of the pool before this modification.
- If in this modification:

    1. The price feed of the collateral or the debt token is invalid.
    2. The pair is under collateralization with respect to its shutdown ltv.
    3. The collateral or debt accumulator reaches `18 * SCALE` (close to the upper bound of u64).

In addition to updating the shutdown status in the position modification callback, it can also be updated in a permissionless way by directly calling `update_shutdown_status()` given a pair of assets.

The following view functions are available in the default extension:

- Getters of the extension's state variables: `pragma_oracle()`, `liquidation_config()`, `shutdown_config()`, `shutdown_ltvs()`, `singleton()`.
- Helpers to get the pair violation and pool shutdown status: `shutdown_status()`, `violation_timestamp_for_pair()`, `violation_timestamp_count()`, `oldest_violation_timestamp()`, `next_violation_timestamp()`.
- Functions for price and adaptive rate computation: `price()`, `interest_rate()`, `rate_accumulator()`.

### 2.2.3 Changes in Version 2

In Version 2 the `default_extension` was changed considerably:

- A `fee_model` component has been added, where anyone can claim fees (`claim_fees()`) to a predefined fee recipient (set at pool creation).
- The accounting of collateral / debt of individual pairs have been separated in Version 2, which will be updated when a position is modified to track the amount of collateral shares and nominal debt on each pair for the shutdown ltv check.
- In addition, in the subscription phase, a call to modify position would now change the collateral max utilization ratio to `100%`. The redemption phase now requires the nominal debt in a position to have been fully repaid in a previous phase; it can no longer be repaid in the redemption phase. Collateral can be withdrawn if the position has no nominal debt but only all the collateral of the position at once.
- An owner will be assigned to each pool at creation, which has the privileges to:

    - `set_pool_owner()`: transfer ownership to another address.
    - `set_asset_parameter()`: set the parameter (`liquidation_discount`, `max_utilization`, `floor`, `fee_rate`) of an asset.

- `add_asset()`: add an asset to the pool and initialize its configurations.

To facilitate this, the Singleton now features new functions `set_asset_config()`, `set_ltv_config()`, `set_asset_paramter()` which allow the extension to update these values.

## 2.2.4 Changes in Version 3

The following changes were made in **Version 3**:

In the singleton:

1. A reentrancy lock has been added in `asset_config()` and `context()`, which prevents the current config or context from becoming stale via reentrancy.
2. Transfer position has been changed to only support transfers in one direction: from the first position to the second position.
3. The extension has been granted delegation to all users' positions.

In the default extension and other components:

1. Liquidation has been updated where liquidators' repayment will be capped by the valuation of the remaining collateral in one position.
2. Pair accounting in the default extension has separated the accounting of accrued collateral and debt asset fees from the users' positions.
3. `liquidation_configs` has been separated for each pair of assets. Hence a collateral asset may have a different liquidation discount in different pairs.
4. Functionalities have been added to support the update of pool configurations by the pool owner:
    - `set_extension`: the owner can change its extension to another address.
    - `set_oracle_parameter`: the owner can update the timeout or number of sources of an asset.
    - `set_interest_rate_parameter`: the owner can change the interest rate parameters of an asset.
    - `set_liquidation_config`: the owner can change the liquidation discount of the collateral asset in a pair.
    - `set_ltv_config`: the owner can change the max ltv (loan to value) of a pair.
    - `set_shutdown_config`: the owner can change the shutdown ltv.

### 2.2.4.1 Tokenization of Collateral Shares

A tokenization (`v_token`) component of deposited assets has been added with the hooks before and after transfer position. The underlying erc20 implementation has also been added (`erc20_component`). Upon creating a pool from the default extension, a vToken will be deployed for each asset configured, which represents a tokenized vault similar to EIP-4626. The vToken represents aggregated user deposits in the position of the default extension, and no debt will be generated on this position.

vTokens is SNIP-2 (inspired by EIP-20) compliant with 18 decimals and will be:

- Minted when users deposit assets on behalf of the default extension or transfer assets shares from a position to the default extension for the pair `<asset, 0>`. Note that in case the minter is the first depositor of an asset in a pool, a small inflation fee will be charged on the shares minted.
- Burnt when users withdraw assets on behalf of the default extension or transfer assets shares from the default extension to another position for the pair `<asset, 0>`.

To facilitate the minting and burning of vTokens from or to users' existing positions, and to conduct necessary checks as after modify positions, two hooks were implemented for transfer position:

- `before_transfer_position`: in case the user is transferring collateral shares from the default extension's position to another position (without touching the debt asset), the default extension will temporarily grant delegation to the caller. This delegation will be revoked in the `after_transfer_position` callback.
- `after_transfer_position`: the pair accounting and shutdown status will be updated. In case only collateral assets are transferred between the user positions and the default extension, vTokens will be minted or burnt accordingly.

### 2.2.5 Changes in Version 4

The following changes were made in **Version 4**:

- The default extensions accounting now excludes special pairs where the debt asset is zero, as these pairs always have an LTV of 100%. Note that the Singleton does not inform the extension of changes to this special pair such as fee accruals. As a result, no extension can perform accounting on this pair.
- The vtoken now features a getter function to retrieve the `pool_id`.
- The Singleton has an additional public function to check the collateralization of a position: `fn check_collateralization`.
- Function `claim_fee_shares` has been added to Singleton to allow fees accrual in a permissionless way.
- The variable `liquidation_discount` has been renamed to `liquidation_factor`.
- The dust collateral check in `update_position` has been updated to enable liquidations which leave a dust collateral but no debt in the position. This also enables deposits of dust collateral into a position if it has not debt.
- To mitigate ready-only reentrancy, safe versions of view functions have been added in addition to the current unsafe versions.

### 2.2.6 Roles and Trust Model

The system is designed to be fully permissionless: anyone can deploy a pool with any assets, any extension with customized behavior, any oracles and any arbitrary configuration of the pool. Users must fully trust the pool configuration and extensions they interact with, they use the pool and extensions at their own risk. A non exhaustive list of risks:

1. The pool's parameters could be misconfigured so it does not function correctly.
2. The asset could be a random token contract that has a backdoor.
3. The oracle is misconfigured, centralized or vulnerable to price manipulation.
4. The extension is malicious so that users' are subject to all the risks including:
    - censorship on each operations.
    - smart contract upgradeability.
    - back door to steal user's funds via `retrieve_from_reserve()`.

In addition, a collateral chain could be configured in a pool where a token can be used as collateral and also be borrowed. This potentially introduces risks of cascading liquidation to the users in case bad debt occurs on this asset.

All assets involved are expected to be normal fully compliant ERC-20 tokens with no special behavior. Special tokens including rebasing tokens, tokens with fees on transfer, and tokens with blacklists are not supported.

In Version 2 an owner is introduced to each pool, which has the privilege to update important configurations of the pool such as the liquidation discount. It is assumed that the owner of each pool is fully trusted, otherwise the owner can adjust the configs to its own interest for instance:

1. Front-run / censor users by increasing the `fee_rate` or changing the `max_utilization`.
2. Set a higher liquidation discount for the owner itself to liquidate users.
3. Add risky assets to the pool.

In Version 3, more configurations of a pool with the default extension can be modified by owner. Updating the extension by the owner could be dangerous, the accounting of the current extension will be lost and the owner should make sure the new extension is compatible with the pool. In addition, each vToken is bound to a specific default extension, in case the default extension is changed to another extension, the vToken may not work and the corresponding collateral shares may be lost.

# 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 |

- Unrestricted Interest Model Could Inflate the Collateral Shares Risk Accepted

| | |
|---|---|
| Low-Severity Findings | 1 |

- Liquidator Slippage Protection Is Delegated to Extensions Risk Accepted

## 5.1 Unrestricted Interest Model Could Inflate the Collateral Shares

Security Medium Version 1 Risk Accepted

*CS-VEP-011*

The default extension adopts an adaptive interest rate model, where interest rate changes over time regarding the utilization ratio. The interest rate model contains the following parameters:

```
struct InterestRateModel {
    min_target_utilization: u256,
    max_target_utilization: u256,
    target_utilization: u256,
    min_full_utilization_rate: u256,
    max_full_utilization_rate: u256,
    zero_utilization_rate: u256,
    rate_half_life: u256,
    target_rate_percent: u256,
}
```

The interest rate should always stay within the `min_full_utilization_rate` and `max_full_utilization_rate`, however, there is no upper bound on the `max_full_utilization_rate`. In case the interest rate becomes too high, the price per collateral share may be remarkably inflated during a short period, which enables the share holders to:

- Withdraw more collateral than one deserves before bad debt occurs.
- Borrow more assets with inflated collateral value and create more bad debt.

For instance:

1. [Block1] Attacker creates a pool with pairs A->B, B->C and sets up a constant high interest rate `10000%` per second for borrowing asset B.
2. [Block1] Attacker deposits 100 A token and 100 B tokens to the pool.
3. [Block2] Attracted by the high interest rate, Victim deposits 100 B tokens to the pool.
4. [Block2] Attacker borrow 1 B (1 usd) token with 2 A (2 usd with 80% ltv) tokens as collateral.
5. [Block3] Assume 10 seconds elapsed since Block2. Now Attacker's position is insolvent as the debt accrues to 1000 B tokens, but no one is incentivized to liquidate Attacker due to the small amount of collateral compared to the tx gas fee. Moreover, now total assets of B becomes 1200 in the perspective of the protocol and the share price becomes 6x. Attacker could do the following to profit:

    - Directly redeem the half of total B token's shares and withdraw from all the 200 B tokens (still need to respect the utilization ratio).
    - Borrow 6x more asset C from the pool and default.

---

**Risk accepted:**

VESU states:

```
We will not change the behavior. Given Vesu's trust model, we don't see this as an
issue but simply a risk users have to factor into their assessment of a Vesu pool.
```

## 5.2 Liquidator Slippage Protection Is Delegated to Extensions

Design | Low | Version 1 | Risk Accepted

*CS-VEP-013*

The slippage protection for the liquidator is not enforced in the singleton but in the extension. In the default extension, the liquidators can protect themselves from slippage by setting the amount of debt they are willing to repay and a minimum amount of collateral to receive.

```
let LiquidationData{min_collateral_to_receive, mut debt_to_repay } =
    Serde::deserialize(ref data).unwrap();

// check that a min. amount of collateral is released
assert(collateral_to_receive >= min_collateral_to_receive, 'less-than-min-collateral');
```

However, slippage protection may not be enforced in other extensions. Furthermore a malicious extension may also overwrite the debt the liquidator wants to repay.

Liquidators must exercise caution and inspect the extensions before initiating the liquidation. They should ensure self-enforcement of their slippage tolerance by approving only the amount they are willing to repay, or by wrapping the liquidation call in code that performs the required additional checks.

---

**Risk accepted:**

VESU states:

We will not change the behavior. Given Vesu's trust model, we don't see this as an issue. Liquidation bots can still enforce the invariant across all pools in a separate contract (which they will use in most cases anyway for funding liquidations through flash loans).

# 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 |
|---|---|

- Transfer Position Across Different Collateral/Debt; Pool Isolation Broken Code Corrected

| High-Severity Findings | 5 |
|---|---|

- Pair Accounting Misses Fee Shares Code Corrected
- Double Accounting in Shutdown Ltv Check Code Corrected
- Partial Liquidations Are Broken Code Corrected
- Pool Status Update Does Not Validate the Violation Status for All Pairs Code Corrected
- Shutdown Status Can Be Updated on Unconfigured Pairs Code Corrected

| Medium-Severity Findings | 11 |
|---|---|

- Fees Are Not Passed to Extension in Transfer Position Code Corrected
- Issues With Charging Inflation Fee Code Corrected
- VToken ERC-4626 Preview Functions Ignore Shutdown Status Code Corrected
- Liquidations With receive_as_shares Distribute Fees Twice Code Corrected
- Updated Fee Rate May Also Apply to the Past Code Corrected
- Donation and Retrieval From Reserve Do Not Update Accrued Interests Code Corrected
- Fee Shares Are Locked on the Default Extension Code Corrected
- Max Utilization Is Enforced in Redemption Phase Code Corrected
- Price Validity Check Could Be Skipped Code Corrected
- Rounding Errors Code Corrected
- Target Amount Sign Is Ignored Code Corrected

| Low-Severity Findings | 9 |
|---|---|

- Missing Storable Check Code Corrected
- Callback May Change Solvency During Liquidation Code Corrected
- Inaccurate Function Description Specification Changed Code Corrected
- Mapping From Collateral Asset to VToken Is Not Written Code Corrected
- No Liquidation Possible Without Available Collateral Code Corrected
- Outdated Dependency Code Corrected
- Public Calculation Functions Use Stale Asset Config Code Corrected
- Read Only Reentrancy Is Possible Code Corrected
- Unchecked Fee Rate in Packing Code Corrected

| Informational Findings | 8 |
|---|---|

- Mutability of Function Delegation Could Be Restricted to View Code Corrected
- Incorrect Specification of Rate Accumulator Return Value Specification Changed
- Caller Is Not Checked in After Liquidation Callback Code Corrected
- ModifyDelegation Does Not Contain the Delegator Address Code Corrected
- Transfer to the Same Position Prohibited Code Corrected
- Liquidation Does Not Update the Pool Status Code Corrected
- Last Rate Accumulator Can Be Set to 0 Code Corrected
- Unused Code Code Corrected

## 6.1 Transfer Position Across Different Collateral/Debt; Pool Isolation Broken

Security Critical Version 1 Code Corrected

*CS-VEP-007*

`fn transfer_position` is intended to facilitate transfer of debt or collateral from one position to another position in the same pool. Among the user specified function parameter are the assets (`from_collateral_asset`, `from_debt_asset`, `to_collateral_asset`, `to_debt_asset`) and the amounts (`collateral`, `debt`). Note that the amounts are not integers but of type `struct Amount`:

```
#[derive(PartialEq, Copy, Drop, Serde, Default)]
struct Amount {
    amount_type: AmountType,
    denomination: AmountDenomination,
    value: i257,
}

#[derive(PartialEq, Copy, Drop, Serde, Default)]
enum AmountType {
    #[default]
    Delta,
    Target,
}

#[derive(PartialEq, Copy, Drop, Serde, Default)]
enum AmountDenomination {
    #[default]
    Native,
    Assets,
}
```

Notably the amount can be specified in terms of `Delta` or `Target`.

When transferring any amount of collateral/debt from one position to another it's imperative that this can only be done if the corresponding collateral/debt asset is equal.

The function tries to enforce this:

```
if collateral.value.is_non_zero() {
    assert!(from_collateral_asset == to_collateral_asset, "collateral-asset-mismatch");
}
```

```
if debt.value.is_non_zero() {
    assert!(from_debt_asset == to_debt_asset, "borrow-asset-mismatch");
}
```

This check however is only effective if the `AmountType` of the `Amount` specified is of type `Delta`. If the `AmountType` is `Target` a value of 0 applied to a non zero position leads to a change of balance, the assets however have not been checked to be equal.

The first position is modified based on the supplied collateral/debt amounts. The second position is modified based on the reported delta of the first modification:

```
let response = self
    ._modify_position(
        ModifyPositionParams {
            pool_id: pool_id,
            collateral_asset: from_collateral_asset,
            debt_asset: from_debt_asset,
            user: from_user,
            collateral,
            debt,
            data: from_data
        }
    );

self
    ._modify_position(
        ModifyPositionParams {
            pool_id: pool_id,
            collateral_asset: to_collateral_asset,
            debt_asset: to_debt_asset,
            user: to_user,
            collateral: Amount {
                amount_type: AmountType::Delta,
                denomination: AmountDenomination::Assets,
                value: -response.collateral_delta,
            },
            debt: Amount {
                amount_type: AmountType::Delta,
                denomination: AmountDenomination::Assets,
                value: -response.collateral_delta,
            },
            data: to_data
        }
    );
```

(There is a separate problem/typo that the debt Amount is constructed using `response.collateral_delta` instead of `response.debt_delta`).

For the assets the function parameter `to_collateral_asset` and `to_debt_asset` are passed ,the asset with the non-zero balance change may not match the corresponding from asset. Consequently the second positions debt/collateral can be manipulated despite being a different asset.

No token transfer happen since this function only works on the internal accounting of the balances. Nevertheless the accounting update not only touches positions of the same pool: since the code internally uses functions `update_position` and `common.apply_position_update_to_context` modifying the positions also updates the AssetConfig tracking the pool's balances (either `context.collateral_asset_config.reserve` or

`context.debt_asset_config.total_nominal_debt` ). This elevates the consequences significantly: E.g. one can increase the `reserve` of the AssetConfig without the respective tokens having been transferred to the Singleton. This effectively breaks the isolation between the pools in the Singleton, due to the increased reserve of the AssetConfig this pool can now tap into collateral tokens belonging to other pools. Similarly this can be done by manipulating the debt instead of the collateral.

---

**Code partially corrected:**

The checks to ensure that the respective assets are equal if the resulting change is non-zero have been corrected: Instead of only checking on non-zero amount values (which is effective for amounts of type `Delta` only) the code now enforces the assets are equal if the amount is of type `Target`. This fixes the issue described above.

Note that this is restrictive in case the caller specifies an amount of type `Target` which doesn`t change the position; he may have to specify the amounts in type of `Delta` and a value of zero instead.

The underlying root issue breaking the pool isolation has not been addressed: Transferring a position within a pool still operates on the accounting of the Singleton hence any potential issue has the potential to break the pool isolation within the Singleton.

---

**Code corrected:**

In Version 3, the code ensures that the reserves, which keep track of the assets belonging to a pool, remain unchanged. This mitigates the risk of pool isolation breaches.

## 6.2 Pair Accounting Misses Fee Shares

Correctness High Version 2 Code Corrected

*CS-VEP-001*

In Version 2, the default extension introduces accounting of collateral shares and nominal debt for each pair. The accounting is not updated correctly in all scenarios where positions of this pair change.

The fee shares minted to the extension are not added to the pair accounting. As a result, the shares tracked are less than the actual amount. In case fees are claimed to the recipient, the last several users may not be able to withdraw their assets due to the pair accounting reverts on underflow.

---

**Code corrected:**

Accrued fee shares for both the collateral and debt asset are now accounted for in the default extension's accounting when the pair is updated (`position_hook.update_pair()`). Note that fee shares are tracked separately in special pairs (`asset, <zero>`).

In Version 4 the extensions accounting has been changed to completely ignore pairs with a debt asset of `<zero>`. Since the ltv of such a pair is always 100%, there's no need for the extension to do the accounting. This simplifies the fix for the issue above.

## 6.3 Double Accounting in Shutdown Ltv Check

Design High Version 1 Code Corrected

*CS-VEP-008*

According to the specification the pool will fall into recovery mode if any pair of assets is under the shutdown ltv. The total collateral is computed based on the total collateral shares and the total debt is computed based on the total nominal debt.

```
fn is_pair_collateralized(ref context: Context, shutdown_ltv: u256) -> bool {
        let Context{collateral_asset_config, debt_asset_config, .. } = context;
        let collateral = calculate_collateral(
            collateral_asset_config.total_collateral_shares,
            collateral_asset_config);
        let debt = calculate_debt(
            debt_asset_config.total_nominal_debt,
            debt_asset_config.last_rate_accumulator, debt_asset_config.scale
        );
        is_collateralized(
            collateral * context.collateral_asset_price.value /
            collateral_asset_config.scale,
            debt * context.debt_asset_price.value / debt_asset_config.scale,
            shutdown_ltv
        )
    }
```

Only the total debt of an asset is tracked, the accounting cannot distinguish which portion of this debt is backed by a particular collateral. If multiple pairs exist in the same pool and one asset exists in multiple pairs, the total debt and collateral accounting of one pair is not isolated between different pairs. Consequently, the shutdown ltv check above cannot reflect the collateralization status of a single pair. For instance:

- Assume there are two pairs: A->B and C->B, with A and C as collateral and B as debt token.
- The check against pair (A->B) will use the total collateral of A and total debt of B, which however consists of the debt backed by both collateral A and collateral C.
- As a result, the check may easily assumes pair A->B falls under-collateralization and the pool will fall into recovery mode.

---

**Code partially corrected:**

Code has been partially corrected to account collateral shares and nominal debt for individual pairs, ensuring there is no double accounting in the shutdown ltv checks any more. However, the accounting misses the fee shares after `modify_position`, please refer to Pair Accounting Misses Fee Shares for more details.

---

**Code corrected:**

In Version 3 issue Pair Accounting Misses Fee Shares has been resolved which completes the fix for this issue.

## 6.4 Partial Liquidations Are Broken

Correctness High Version 1 Code Corrected

*CS-VEP-002*

For liquidations, `fn liquidate_position` calls the extension of the pool to calculate the amounts:

```
let (collateral, debt, bad_debt) = extension.before_liquidate_position(context, data);
```

These amounts are then used as follows:

- `collateral`: used to update the position/reserve and for the transfer of the collateral
- `debt`: used to update the position
- `bad_debt`: The liquidator only transfers debt - bad_debt. The reserve is updated accordingly, taking the loss.

For partial liquidations the bad debt calculation in the default extension is broken. In the following code snippet, `debt_value` is the full debt value of the position, `collateral_value` is the discounted collateral value (liquidation discount applied on full collateral value). These values do not differ for full or partial liquidations. The calculation of the bad debt is shown below:

```
// account for bad debt if the entire position is liquidated and there's not enough collateral to cover the debt
let bad_debt = if collateral_to_receive == collateral && collateral_value >= debt_value {
    0
} else {
    // only the part of the debt that is covered by the collateral is liquidated
    (debt_value - collateral_value) * context.debt_asset_config.scale / context.debt_asset_price.value
};
```

### 6.4.1 Case I: Insufficient collateral

In such a scenario the discounted collateral value `collateral_value` is strictly < the `debt_value` hence the `else` branch is reached.

The bad debt calculated is the full bad debt as if the entire position was liquidated. The partial liquidated position however still exists with remaining collateral / debt.

The reserve is updated which includes writing off this bad debt. A second (partial) liquidation of the position would recalculate the bad debt based on the remaining position value and write off bad debt again.

There's a corner case possible where the partial liquidation takes all the collateral and the position remains with only debt remaining.

### 6.4.2 Case II: Sufficient collateral

When a position falls below the liquidation threshold, the value of the discounted collateral could still be worth more than the debt. No bad debt will occur in this case. In case a partial liquidation occurs (`collateral_to_receive < collateral` and `debt_to_repay < debt`), the bad debt will be computed through the else branch, which will underflow and revert due to the debt value being smaller than the discounted collateral value.

Hence partial liquidation is not supported when bad debt does not occur. This contradicts what is described in the whitepaper.

---

For Case I (insufficient collateral) in **Version 2** the bad debt is intended to be calculated proportional to the amount that is being repaid. However, the formula used is wrong. The bad debt to write off in this liquidation is calculated and scaled by `debt_to_repay` divided by `debt`. Since there is insufficient collateral, even a full liquidation (or multiple partial liquidations of the position) wouldn't recover `debt`. Hence using `debt_to_repay/debt` is not the correct scaling factor to calculate the proportional share of bad debt this partial liquidation results in.

Case II (sufficient collateral) has been fixed in **Version 2**, in case of `collateral_value` being equal or exceeding `debt_value` the bad debt is set to `0` for full and partial liquidations.

---

**Code corrected:**

The calculation for Case I (insufficient collateral) has been corrected in Version 3:

If the discounted value of the positions collateral is worth less than the debt to be repaid, debt to be repaid at the position is updated to the full debt of the position (liquidator's repayment is capped by the valuation of the collateral). Note that in Version 3 this value is recalculated based on the value and price, instead of taking the position's debt value directly. This has been fixed in Version 4. The bad debt is the delta of the `debt_value` and `collateral_value`, expressed in units of the debt asset.

Else the bad debt is calculated proportional to the amount of debt being repaid.

## 6.5 Pool Status Update Does Not Validate the Violation Status for All Pairs

Correctness High Version 1 Code Corrected

*CS-VEP-024*

The shutdown status of a pool is dependent on the violation status of all the pairs in the pool. The pool will only recover if all the pairs have exited the recovery mode.

In `update_shutdown_status()`, the new shutdown status of the pool is dependent on:

- The current shutdown status of the pool, which is determined by the oldest violation timestamp.
- The new shutdown status of the current pair.

In case the current pair will exit the recovery mode and it is the only one at that violation timestamp, the pool will be regarded as back normal again. However, this only checks the oldest violation timestamp and the violation timestamp of the current pair. In case there is any other pair that has a violation timestamp in the past, the pool will falsely recover.

For instance:

- Consider two pairs, PA and PB, with a recovery period of 10.
- Assume PA is the only one that is violating at the oldest timestamp 1. PB has a violation timestamp 5.
- Now assume PA is back to collateralization and `update_shutdown_status()` is called on PA at timestamp 8. The violation timestamp at 1 will be cleared and pool is regarded back to normal. However, PB is never checked and the pool's shutdown status should actually be dependent on PB.

---

**Code corrected:**

Code has been corrected to take the second oldest violation timestamp into consideration if the oldest violation timestamp will be cleared. This ensures the pool shutdown status will be dependent on the status of all pairs.

## 6.6 Shutdown Status Can Be Updated on Unconfigured Pairs

Security High Version 1 Code Corrected

*CS-VEP-026*

The shutdown status can be updated on any pair in a permissionless way by calling `update_shutdown_status()` on the default extension. In case this pair falls below the shutdown ltv, the pool will enter the recovery mode.

For a pair that is not configured, its `shutdown_ltv` would be 0. However, this is not checked by `update_shutdown_status()`. As a result, it is possible to update the pool's shutdown status using an unconfigured pair. Such a pair will always be regarded as under-collateralized as long as there is any debt because the shutdown_ltv (passed as `max_ltv_ratio` to `is_collateralized()`) is 0.

```
fn is_collateralized(collateral_value: u256, debt_value: u256, max_ltv_ratio: u256)
-> bool {
    collateral_value * max_ltv_ratio >= debt_value * SCALE
}
```

Therefore, anyone can easily trigger a recovery by simply picking an unconfigured pair A->B. The pool can only recover if all the debt on token B has been repaid.

---

**Code corrected:**

Code has been corrected to account collateral shares and nominal debt for individual pairs. Thus the collateral and debt for unconfigured pairs will remain zero and will not trigger a shutdown.

## 6.7 Fees Are Not Passed to Extension in Transfer Position

Correctness | Medium | Version 3 | Code Corrected

*CS-VEP-031*

In `transfer_position()`, the fee shares will be accrued and cached in the `from_context` and `to_context`. When passing both contexts to the extension in `after_transfer_position()`, the `from_context` is reloaded to avoid changes introduced by `to_context`. However, the reloaded `from_context` misses the fee shares as it has already been written to storage by `update_position()` on the `from_context`. As a result, the extension may not obtain the information of accrued fee shares in the `from_context`.

---

**Code corrected:**

Code has been corrected to cache the `collateral_asset_fee_shares` and `debt_asset_fee_shares` when loading the `from_context` at the first time, and they will be passed to the reloaded `from_context` before calling `after_transfer_position()`. Even though the fees are not accounted by the default extension anymore in Version 4, it is important to pass these values to other extensions correctly.

## 6.8 Issues With Charging Inflation Fee

Correctness | Medium | Version 3 | Code Corrected

*CS-VEP-033*

When updating a position in the Singleton, upon the first deposit for an asset an inflation fee is charged. This protection will never be triggered however, due to a wrong execution order:

1. When a user makes the first deposit, the minted shares will be added to the total collateral shares of the context in `apply_position_update_to_context()` (line 1).
2. After the update of the context, its total collateral shares are compared with 0 (line 7). Hence the singleton will assume there are already some deposits, which leads to 0 inflation fee at the first deposit.

In addition, the inflation fee of a collateral asset will be locked to 0 address of the pair `<collateral, debt>` (line 13). This implicitly assumes the user is not 0 address. However, in case the first deposit is made on behalf of 0 address, the position of 0 address will be overwritten by a second update to the position (line 23), leading to a discrepancy between total collateral shares and sum of all positions' shares.

```
:linenos:

let (collateral_delta, mut collateral_shares_delta, debt_delta, nominal_debt_delta)
    =   apply_position_update_to_context(ref context, collateral, debt, bad_debt);

let Context { pool_id, collateral_asset, debt_asset, user, .. } = context;

// charge the inflation fee for the first depositor for that asset in the pool
let inflation_fee = if context.collateral_asset_config.total_collateral_shares == 0
    && !collateral_shares_delta.is_negative {
    let inflation_fee = 1000;
    self
        .positions
        .write(
            (pool_id, collateral_asset, debt_asset, Zeroable::zero()),
            Position { collateral_shares: inflation_fee, nominal_debt: 0 }
        );
    context.position.collateral_shares -= inflation_fee;
    inflation_fee
} else {
    0
};

// store updated context
self.positions.write((pool_id, collateral_asset, debt_asset, user), context.position);
```

---

**Code corrected:**

The code has been fixed to charge the inflation fee upon the first deposit. The if condition is now based on the initial total collateral shares before it has been updated by `apply_position_update_to_context()`.

Furthermore, the code now includes a safeguard that prevents minting the inflation fee to the zero-user.

## 6.9 VToken ERC-4626 Preview Functions Ignore Shutdown Status

**Correctness** **Medium** **Version 3** **Code Corrected**

*CS-VEP-032*

The default extension implements a shutdown mechanisms that can prevent deposits or withdrawals. However, the VTokens ERC-4626 preview functions (`max_deposit`, `preview_deposit`, `max_mint`, `preview_mint`, `max_withdraw`, `preview_withdraw`, `max_redeem`, `preview_redeem`) do not

account for this shutdown status. Instead, they return optimistic values as if the pool is live, which contradicts the ERC-4626 specifications.

---

**Code corrected:**

The code of VToken has been updated to take the shutdown status into account. Functions now return `0` if deposits/withdrawals are not possible.

## 6.10 Liquidations With `receive_as_shares` Distribute Fees Twice

Correctness Medium Version 2 Code Corrected

*CS-VEP-034*

Since Version 2, as a solution to issue No Liquidation possible without available collateral collateral of liquidations can also be received as shares. The implementation however has a bug where the fees are distributed twice.

Fees are calculated when the context (asset config) is loaded and distributed when the position is updated.

When `receive_as_shares` is `True`, `fn liquidate_position()` updates the position of the account being liquidated and the position of the caller using the same context, thereby distributing the previously calculated fees twice.

---

**Code corrected:**

The context is now reloaded prior to updating the position of the liquidator if he chooses `receive_as_shares`. As the context has already been updated, this prevents a second distribution of the same fees.

## 6.11 Updated Fee Rate May Also Apply to the Past

Correctness Medium Version 2 Code Corrected

*CS-VEP-003*

In Version 2 the owner can update `fee_rate` of an asset after the pool creation by `set_asset_parameter()`, which simply overwrite the asset config. In case there is any interest that has not been accrued in the past (since the last update of rate accumulator), the fees on this period will also be charged with the new `fee_rate` instead of the old one. As a result, the owner can accrue more fee shares to the extension by increasing the `fee_rate` after a long period of no update of an asset.

In Version 3 the default extension received a public function `set_interest_rate_parameter` allowing the owner to update these parameters. A similar issue occurs when the interest rate model parameters are changed in the extension. The next time the Singleton calculates the fee shares, they will be calculated from the last update, using the rate accumulator with the new interest rate parameters.

---

**Code partially corrected:**

In Version 3 the calculation of the fee shares has been moved to `fn asset_config` hence upon loading the asset config, the fees shares are updated before the rate is changed. Note that starting from Version 3 fee shares are tracked in special pairs `(asset, <zero>)`.

However, the fee shares are added to the total shares but not minted to the extension (usually done in `Singleton.update_position()` which isn't executed here), resulting in the forfeiture of the accrued shares. Furthermore the extension is not informed about these new fee shares. If the extension features an internal accounting, which the default extension does, this accounting isn't updated (done in `position_hook.update_pair()` which is invoked in the `after_transfer/modify_position()` callbacks.

---

**Code corrected:**

Starting from Version 4, the default extension's accounting ignores pairs with a debt asset of `<zero>` (special reserved pair for fees and vtokens). Therefore, it is acceptable for the extension not to be informed about minted fees. Note that all extensions must support this restriction, as the Singleton does not notify them of this update.

`Singleton.set_asset_parameter()` now mints the fee share calculated based on the old `fee_rate` to the extension.

The Singleton now exposes a function to attribute the outstanding fee shares to the pool's extension. The `set_interest_rate_parameter` function of the default extension uses this to update the fees before changing the parameters. Arbitrary extensions must make sure to implement a similar mechanism if needed.

## 6.12 Donation and Retrieval From Reserve Do Not Update Accrued Interests

Correctness Medium Version 1 Code Corrected

*CS-VEP-009*

Vesu does not update the asset config before the reserve is changed by donations `donate_to_reserve()` or extension's asset retrieval (`retrieve_from_reserve()`). Both action lead to an abrupt change in utilization.

Vesu accrues interest and fees based on the rate accumulator increase. When preparing the context, upon loading the asset_config, if it's the first time in a block, the rate accumulator is updated, and fees are distributed. The calculation of the rate accumulator is delegated to the extension. Several arguments are passed including `utilization` and `last_updated`, it's up to the extension how these are used. The default extension adopts an adaptive interest rate model, where interest rate changes over time regarding the utilization ratio.

- In case of an asset donation the utilization ratio decreases, which implies a interest rate decrease. Consequently users' debt before the donation will be charged at a lower interest than it should be.
- In case of an asset retrieval the utilization ratio increases, which implies a interest rate increase. Consequently users' debt before the retrieval will be charged at a higher interest than it should be.

---

**Code partially corrected:**

Both functions have been changed to update the rate accumulator before changing the reserve. This updates the AssetConfig, namely the fields `last_rate_accumulator`, `last_full_utilization_rate` and `last_updated`. The fees however, which depend on

`last_updated` and the current rate accumulator are not updated. Consequently, since `last_updated` has been updated without updating the fees, the fees since the last update will be forfeited.

In Version 3 the calculation of the fee shares has been moved to `fn asset_config` hence upon loading the asset config, the fees shares are updated. However, the fee shares are only added to the total shares but not minted to the extension (usually done in `Singleton.update_position()`), still resulting in the forfeiture of the accrued shares. Furthermore the extension is not informed about these new fee shares. If the extension features an internal accounting, which the default extension does, this accounting isn't updated (done in `position_hook.update_pair()` which is invoked in the `after_transfer/modify_position()` callbacks.

---

**Code corrected:**

In Version 4 the fees are attributed to the extension. Starting from Version 4, the default extension's accounting ignores pairs with a debt asset of `<zero>` (specially reserved pair for fees and vtokens). Therefore, it is acceptable for the extension not to be informed in this case. Note that all extensions must support this restriction, as the Singleton does not notify them of this update.

## 6.13 Fee Shares Are Locked on the Default Extension

Design Medium Version 1 Code Corrected

*CS-VEP-021*

If fees are configured, part of the interest accrued will be credited to the extension by minting new shares to the extension's position on the same pair of assets. However, the default extension implements no logic to handle the fee shares and should fees be accrued, these shares will simply be locked forever. The description of the Factory Extension in the Whitepaper does not cover these fees at all.

---

**Code corrected:**

A new component `fee_model` has been added, which exposes `claim_fees()` functionality to the extension to redeem the fee shares and send to the fee recipient.

## 6.14 Max Utilization Is Enforced in Redemption Phase

Design Medium Version 1 Code Corrected

*CS-VEP-010*

In the redemption phase of a pool's shutdown, users are allowed to withdraw their collateral without modifying their debt. However, the max utilization check is still enforced when withdrawing the collateral. In case there is some loan on the collateral asset that has not been paid back, a proportional collateral will be locked forever to maintain the utilization ratio.

```
if collateral_delta < Zeroable::zero() {
    // max. utilization of the collateral is not exceed
    self.assert_max_utilization(context.collateral_asset_config);
}
```

For instance, in case a pool is in redemption phase with 10 ether of reserve and 2 ether of debt with 80% max utilization ratio. Then there would be 0.5 ether locked in the reserve forever to maintain the utilization ratio for the 2 ether unpaid debt.

---

**Code partially corrected:**

Code has been changed to set the `max_utilization` rate of a collateral to `100%` in `after_modify_position()` hook in case the pool has entered the subscription phase. Hence all available funds in the reserve could be withdrawn once the pool enters the redemption phase.

Nevertheless, in case of no execution of `modify_positions()` during the subscription phase for a collateral, the `max_utilization` rate for this collateral will not be set to `100%`. Consequently withdrawal of idle funds will still be blocked in the redemption phase.

---

**Code corrected:**

In Version 3 setting `max_utilization` to `100%` upon shutdown has been moved to the redemption phase. Note that when modifying a position, the check whether the action respects the current `max_utilization` is done before this update ((in `assert_position_invariants()` and afterwards `extension.after_modify_position()`). Hence transactions may revert unexpectedly if they do not meet the current limit. A successful transaction respecting the current limit allows to set the limit to `100%`. If `max_utilization` has already been reached, users can call `update_shutdown_status` directly to set the `max_utilization` to `100%` in the redemption phase.

## 6.15 Price Validity Check Could Be Skipped

Design | Medium | Version 1 | Code Corrected

*CS-VEP-028*

The price validity from the oracle is only checked when updating the shutdown status `update_shutdown_status()`. In case the price of either the collateral or the debt asset in this operation is invalid, the pool can only be in recovery / subscription / redemption status. An invalid price may lead to the following consequences:

1. In the recovery mode, liquidation is allowed. A liquidator may leverage an invalid price to repay less debt and get more collateral.
2. In the redemption mode, users can withdraw their collateral respecting the floor and over-collateralization check. An invalid price could be leveraged to withdraw more collateral back.

---

**Code corrected:**

1. A check of collateral / debt asset price validity has been added to `before_liquidate_position()` which ensures liquidation cannot leverage an invalid price. Note that this introduces another risk of being unable to liquidate unhealthy positions in time (or even at all) leading to bad debt for the protocol. This must be evaluated carefully.
2. One can only redeem if all the nominal debt has been repaid in the subscription phase, which eliminates the possibility of abusing an invalid price in the redemption phase.

## 6.16 Rounding Errors

Design | Medium | Version 1 | Code Corrected

*CS-VEP-004*

Conversion between debt <=> nominal debt or collateral <=> collateral shares is subject to rounding errors. The current implementation does not enforce these rounding errors to be in favor of the protocol. For instance:

- When modifying a position depending on whether the delta is positive or negative and whether the conversion is to or from `Asset` or `Native` the rounding may be in the protocol or in the user's favor.
- When computing the position collateralization ratio before liquidation (`calculate_collateral_and_debt_value()`), the debt value is rounded down, thus a slightly under-collateralized position may not be regarded liquidatable.
- When bad debt occurs in a liquidation, the bad debt is rounded hence the reserve is rounded up. The pool may actually hold less assets than its reserve.

Rounding errors may lead to transactions reverting unexpectedly if the resulting amount leads to a calculation underflowing. Within Vesu, Users are generally free to specify Amounts in terms of `Asset` (Token) or `Native` (Collateral Shares or Nominal Debt) with the protocol converting the value as needed. In corner cases transactions may revert unexpectedly, e.g. when trying to remove all collateral/debt from a position.

Although these rounding inaccuracies are minimal, past exploits have shown that they can be exploited under specific circumstances. To avoid any potential risk associated with rounding errors it's generally best practice to always round in favor of the protocol.

**Caution:**

In case the rounding direction is fixed and the newly minted shares are rounded down (in favor of the protocol), the protocol still imposes insufficient checks against share inflation attack. The current mitigation ensures the shares will not be rounded down to 0:

```
assert!(collateral_delta.is_non_zero() == collateral_shares_delta.is_non_zero(),
    "zero-shares-minted");
```

Nevertheless, the user can still get 1 wei of share less, which could be leveraged to exploit the incoming users at the creation of a new pool by inflating the price per share. For instance:

- At the creation of the new pool, the attacker mint 1 wei of share by depositing 1 wei of WETH The current price per share is 1.
- The victim send the tx to deposit 1 WETH.
- The attacker front-run the victim to donate 0.5 WETH (`donate_to_reserve()`), which inflates the price per share to (0.5*1e18 + 1).
- In case the shares to the victim get rounded down, the victim will get only 1 share.
- Eventually the price per share becomes 0.75*1e18, causing a lose of 0.25 WETH to the victim.

---

In Version 2 a rounding flag (`round_up`) has been added to round in favor of the protocol when converting between shares and underlying tokens.

However it may fail to round to the correct direction. For instance, in `calculate_debt()`, if the remainder of the first division (`u512_safe_div_rem_by_u256`) is 0, then the `round_up` flag will be ignored and the result will always be round down in the second division (`safe_div`).

```
fn calculate_debt(nominal_debt: u256, rate_accumulator: u256,
    asset_scale: u256, round_up: bool) -> u256 {
    let scaled_nominal_debt = integer::u256_wide_mul(nominal_debt * rate_accumulator, asset_scale);
    let (debt, remainder) = integer::u512_safe_div_rem_by_u256(
        scaled_nominal_debt, SCALE.try_into().unwrap());
    assert!(debt.limb2 == 0 && debt.limb3 == 0, "debt-overflow");
    let debt = u256 { low: debt.limb0, high: debt.limb1 };
    if remainder == 0 || !round_up {
        safe_div(debt, SCALE, false)
    } else {
        safe_div(debt + 1, SCALE, true)
    }
}
```

In addition, mechanisms to prevent share inflation attacks at the pool creation have not been implemented in Version 2. The first depositor could still be sandwiched and lose half of its deposit.

---

**Code corrected:**

In Version 3, the code has been corrected to use the correct rounding flag (`round_up`) in the conversion between assets and shares.

An inflation fee has been introduced in `update_position()` for the first depositor of an asset to mitigate the share inflation attack. However, this protection will never be triggered due to a wrong execution order, please refer to issue Issues with charging inflation fee. This issue has been resolved in Version 4.

## 6.17 Target Amount Sign Is Ignored

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

*CS-VEP-005*

Amounts can be denoted as target amount or delta amount. The actual value is express as a signed integer `i257`. The integer is stored in a struct with an `u256 abs` for the absolute value and a `boolean is_negative` for the sign. A negative target amount is invalid in practice.

The amounts are interpreted in functions `deconstruct_collateral_amount()` or `deconstruct_debt_amount()`. For amounts expressed as delta, the sign indicates the funds inflow or outflow. However for amounts expressed as target, the sign is simply ignored. A negative target value will always be treated as a positive value.

Resulting executing `modify_position()` or `transfer_position()` with negative target amount will simply treat them as positive value for the manipulation of the position. However, the negative value could still be emitted in events `UpdatePosition` and `TransferPosition`. This behavior may be unexpected and can potentially have severe consequences in upstream contracts interacting with Vesu.

---

In Version 2 `transfer_position` has been updated and always overwrites the sign of a non-zero input amount to negative:

- In case of a `Delta` amount, it is only allowed to transfer non-negative collateral / debt from the first position to the second one.
- In case of a `Target` amount, the sign does not have any effect and only the absolute value matters.

Consequently the issue is not resolved, and a negative `Target` amount will still be accepted as a positive amount, which may pose a threat to upstream contracts.

---

**Code corrected:**

In Version 3 functions `deconstruct_collateral_amount()` and `deconstruct_debt_amount()` revert upon negative target amounts. This resolves the original issue.

Furthermore `transfer_position` no longer uses data type `Amounts` for parameters `collateral` and `debt`, but a new data type `UnsignedAmount`. When updating the first position, a negative sign is used for the values of these amounts. Callers must be aware of this behavior. When `amount_type` `Target` is used, only amounts of 0 are supported.

In Version 4, the code fully supports the `amount_type` `Target` again: When updating the first position unsigned amounts for collateral and debt are now only negated if the `amount_type` is `Delta`. However, if the `amount_type` is `Target`, the amount is used without negation.

The `transfer_position` function is intended to facilitate transfers from position 1 (the 'from position') to position 2 (the 'to position'). Therefore the changes in Version 4 include checks to ensure that transfers only occur in this direction, preventing transfers that would add to position 1 before taking from position 2.

## 6.18 Missing Storable Check

Correctness Low Version 2 Code Corrected

*CS-VEP-006*

Function `set_asset_parameter` has been added to facilitate the owner's update of `max_utilization`, `floor` or `fee_rate` of an existing asset config. However, this function does not check if the new `floor` or `fee_rate` is storable.

---

**Code corrected:**

`set_asset_parameter()` now uses `assert_storable_asset_config()` to ensure the new config can be stored correctly. Furthermore, `assert_asset_config()` is used to validate the values.

## 6.19 Callback May Change Solvency During Liquidation

Design Low Version 1 Code Corrected

*CS-VEP-022*

In `fn liquidation`, the solvency of a user's position is checked before the callback `before_liquidate_position()`. After the callback, the context is reloaded as it might have been changed. The callback could also have changed the positions solvency status, e.g. made a position solvent again, however, the position will still be liquidated.

---

**Code corrected:**

The over-collateralization check has been moved after the `before_liquidate_position()` callback, which ensures only under-collateralized positions will be liquidated.

## 6.20 Inaccurate Function Description

Correctness Low Version 1 Specification Changed Code Corrected

- `position_hooks.before_liquidate_position` misses a description of the last return value `bad_debt`. Due to the importance of this value for the liquidation process an accurate description should be present.
- `math::pow_scale` is annotated with a non-existing parameter b.

```
/// * `is_negative` - true if `b` is negative
```

- There is a typo of `contain` in `position_hooks::Storage`:

```
// contais the shutdown configuration for each pool
```

- `pragma_oracle.oracle_configs` misses a description of the minimum required number of sources.

```
// (pool_id, asset) -> (pragma oracle key, timeout)
oracle_configs: LegacyMap::<(felt252, ContractAddress), (felt252, u64, u32)>,
```

- The specification of `creator_nonce()` implies the nonce of a previously created pool will be returned, however, it is actually the nonce of the next pool to be created.
- The specification of `calculate_pool_id()` implies the input nonce should be `creator_nonce() + 1`, however, in `create_pool()`, `creator_nonce()` will be used to computed the new pool id.

---

**Specification changed and code corrected:**

The specifications have been updated for the first 3 points, and code have been corrected for the last 3 points.

## 6.21 Mapping From Collateral Asset to VToken Is Not Written

Correctness | Low | Version 1 | Code Corrected

*CS-VEP-030*

In `tokenization`, function `create_v_token()` only updates the mapping `v_token_for_collateral_asset`, whereas the mapping `collateral_asset_for_v_token` is not updated. Hence the getter `collateral_asset_for_v_token()` will always return 0 even though there is an VToken for the input collateral asset.

---

**Code corrected:**

Code has been corrected to also write the mapping `collateral_asset_for_v_token` in `create_v_token()`.

## 6.22 No Liquidation Possible Without Available Collateral

Design Low Version 1 Code Corrected

*CS-VEP-018*

Liquidation settles debt in exchange for collateral. If the pool's collateral reserve is depleted liquidations are inhibited until collateral is returned / deposited.

Several measures including `max_utilization` which is strictly enforced not only upon borrowing but also when a user wants to close his position / withdraw collateral helps to mitigate this scenario, if configured with an appropriate conservative value.

Nevertheless in a black swan scenario with many liquidations in a short time frame, the available collateral reserve may deplete and further liquidations are inhibited. The collateral still exists, it's just borrowed to lenders and backed by another collateral held by the pool.

---

**Code corrected:**

A flag `receive_as_shares` has been added to the liquidation parameters which allows the liquidator to receive collateral as shares. This allows successful liquidations when there are insufficient idle funds in the reserve.

## 6.23 Outdated Dependency

Security Low Version 1 Code Corrected

*CS-VEP-023*

`alexandria_math::i257` is used in the system to facilitate delta representation in position modifications. The version of the dependency currently used has an issues when dealing with negative zero inputs in the implementations of `i257PartialEq` and `i257PartialOrd`. These concerns have been rectified in the newer version, `cairo-v2.5.4`.

In Vesu this potentially affects code comparing `i257` with zero, e.g.:

```
if collateral_delta < Zeroable::zero() || debt_delta > Zeroable::zero()
```

An outdated openzeppelin erc20 interface (v0.8.1) is used. In the latest version the entrypoints for increase / decrease allowance have been removed.

---

**Code corrected:**

The dependency of `alexandria_math` has been fixed to `e7b6957` where the issue has been rectified.

## 6.24 Public Calculation Functions Use Stale Asset Config

Design Low Version 1 Code Corrected

*CS-VEP-038*

`calculate_collateral_shares()` will compute the shares corresponding to an amount of collateral based on the asset config, which contains the asset accounting information such as total nominal debt,

total collateral shares and the last rate accumulator. However, these fields may be stale due to fees haven't been accrued since the last update. As a result, the conversion result between underlying collateral and shares may be stale.

Similarly, this applies to `calculate_collateral()`, `deconstruct_collateral_amount()` and `deconstruct_debt_amount()` as well.

---

**Code corrected:**

In Version 3 the fee accrual has been moved from `fn context()` to `fn asset_config()`, hence the asset config loaded and used in `calculate_collateral_shares()` and `calculate_collateral()` is up to date.

`deconstruct_collateral_amount()` and `deconstruct_debt_amount()` which previously used the asset config now load the full context including the asset config. The asset config used is now up to date.

## 6.25 Read Only Reentrancy Is Possible

Security | Low | Version 1 | Code Corrected

*CS-VEP-037*

In the Singleton, some view functions (for instance `utilization()`, and `rate_accumulator()`) are not protected by the reentrancy guard. Hence read-only reentrancy is possible. This does not directly influence Vesu, however, it may influence other protocols that interact with Vesu through these getters. For instance, stale information could be returned if the view function is invoked in a callback to a malicious extension / token, which may pose security risks to external systems.

---

**Code corrected:**

In Version 4, these view functions have been marked with an `unsafe` suffix together with specifications to warn the users of the read-only reentrancy risks. In addition, safe versions of these functions have also been implemented where the reentrancy guard is added.

## 6.26 Unchecked Fee Rate in Packing

Correctness | Low | Version 1 | Code Corrected

*CS-VEP-027*

A customized packing scheme is implemented to optimize the storage of the `AssetConfig` struct. At the time of pool creation, `assert_storable_asset_config()` will be called on the input configurations in order to validate:

1. The input parameters fit within the packing data types without overflow.
2. There is no rounding errors during packing.

However, the rounding error of `asset_config.fee_rate` during packing is not checked. Thus the `fee_rate` stored could be different from what the user intended.

```
let fee_rate: u8 = (value.fee_rate / PERCENT).try_into().expect('pack-fee-rate');
```

---

**Code corrected:**

A check has been added to `assert_storable_asset_config()` which ensures the unpacked `fee_rate` matches the input value.

## 6.27 Incorrect Specification of Rate Accumulator Return Value

Informational | Version 3 | Specification Changed

*CS-VEP-036*

In the Singleton, the specification of `rate_accumulator()` states the return value is the new rate accumulator and the new interest rate at full utilization. Since the code has been changed, the asset config is returned.

---

**Specification changed:**

The specification has been updated to describe the correct return value.

## 6.28 Transfer to the Same Position Prohibited

Informational | Version 2 | Code Corrected

*CS-VEP-014*

Along with the fix for issue Transfer Position across different collateral/debt; Pool Isolation broken an additional restriction preventing a transfer within the same pair has been introduced in `fn transfer_position`:

```
// ensure that it is not a transfer to the same position
assert!(
    !(from_collateral_asset == to_collateral_asset && from_debt_asset == to_debt_asset), "same-position"
);
```

The description is inaccurate, this prevents a transfer within the same pair of a pool (which includes a transfer to the same position). Note that there might be valid use cases to transfer between positions of the same pair: When the caller has access to multiple position within a pair, e.g. his own position and a position he is a delegatee of.

---

**Code corrected:**

Code has been corrected to also check if *from_user == to_user*. Hence transferring between different positions within the same pair of a pool is allowed, but transferring to the same position is prohibited.

## 6.29 Caller Is Not Checked in After Liquidation Callback

Informational | Version 1 | Code Corrected

*CS-VEP-019*

The default extension is designed to work with multiple pools and its callback should only be invoked by the singleton. However, the caller is not checked in `after_liquidate_position()`. Although this

state-changing function does not modify any state variables in its body, its caller should still be restricted to the singleton in theory.

---

**Code corrected:**

A check has been added to restrict the caller of `after_liquidate_position()` to the singleton.

## 6.30 Last Rate Accumulator Can Be Set to 0

**Informational** **Version 1** **Code Corrected**

*CS-VEP-025*

`asset_config_exists()` determines whether an asset configuration exists by checking if the `last_rate_accumulator` is non-zero:

```
fn asset_config_exists(asset_config: AssetConfig) -> bool {
    asset_config.last_rate_accumulator != 0
}
```

When creating a pool (`create_pool()`) and configuring the assets, there is no check that the input `last_rate_accumulator` is non-zero.

`create_pool()` relies on the following check to prevent setting the same asset twice (the second configuration would overwrite the first configuration stored). Furthermore whenever an asset_config is loaded in the singleton, this check is used to ensure the asset exists. Consequently should a pool be created with an assets `last_rate_accumulator` set to 0, this pair isn't usable.

---

**Code corrected:**

A check has been added to `assert_asset_config()` which ensures the initial `last_rate_accumulator` is not less than SCALE.

## 6.31 Liquidation Does Not Update the Pool Status

**Informational** **Version 1** **Code Corrected**

*CS-VEP-015*

Liquidation is allowed when the pool is in normal or recovery mode. In case the pool is in recovery mode, the pair's health factor could potentially be improved after the liquidation. However, liquidation does not update the pool status. Though anyone can update the pool status by directly calling `update_shutdown_status()` on the default extension after liquidation.

---

**Code corrected:**

Code has been corrected and the shutdown status will be updated in `after_liquidate_position()` now.

## 6.32 ModifyDelegation Does Not Contain the Delegator Address

Informational | Version 1 | Code Corrected

*CS-VEP-020*

The event `ModifyDelegation` does not contain the delegator address. Thus observer of this event cannot figure out who is changing the delegation by only looking at the event.

---

**Code corrected:**

The caller address (delegator) has been added to the `ModifyDelegation` event.

## 6.33 Mutability of Function Delegation Could Be Restricted to View

Informational | Version 1 | Code Corrected

*CS-VEP-035*

In singleton, `fn delegations` (renamed to `fn delegation` in Version 2) will return the delegation status of the given addresses. This function simply reads an entry from the storage mapping and does not modify any states, hence its state mutability could be restricted to view.

---

**Code corrected:**

In Version 4 the mutability was changed to view.

## 6.34 Unused Code

Informational | Version 1 | Code Corrected

*CS-VEP-029*

In singleton, `calculate_fee_shares()` accepts a redundant parameter `fee_rate`. The actual rate used is retrieved from the asset config (`asset_config.fee_rate`).

In singleton, the trait `ICallee` is defined but never used.

In packing, the following imports are never used:

```
use alexandria_math::i257::i257;
use starknet::{ContractAddress};
```

---

**Code corrected:**

The redundant parameter `fee_rate` has been removed from `calculate_fee_shares()` and `ICallee` has been deleted.

# 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 Minted Fees Are Not Subject to Dust Check

**Informational** **Version 1**

*CS-VEP-016*

The floor value of the `AssetConfig` prevents dusty positions: Whenever a position is modified, the code of `_modify_position()` ensures the amounts at the position are not below their respective floor values.

Note that the fees minted to the extension are not subject to this floor check. Hence the position of the extension may have a collateral share value below the minimum threshold specified by the `floor` value.

---

VESU states:

```
Won't fix as enforcing the floor check may prohibit the accrual of fees.
```

## 7.2 Oracle Price May Lose Precision

**Informational** **Version 1**

*CS-VEP-017*

The oracle component will fetch the median price from pragma oracle. The price will be rescaled from its original decimals to SCALE (18 decimals). As a result, in case the original decimal is above 18, the price would be rounded down thus slightly lose precision.

# 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 Callback Before Modify Position Will Overwrite User Input

Note Version 1

To modify a position, a user needs to pass the collateral and debt amount one wishes to change by calling `modify_position()`. However, the user's input will be overwritten by the callback `before_modify_position()`. This design adds more flexibility to the protocol, however, implies users need to fully trust the extension.

The extension has the potential to prevent users from doing what they expect. For instance:

- Preventing withdrawing collateral and repaying debt.
- Force users to withdraw more collateral or increase more debt, thus making the position prone to liquidation.

## 8.2 Delegatee Controls Position and Transferred Tokens

Note Version 1

Users can set their delegatees (`modify_delegation()`) to help maintain their positions. One user can grant multiple delegatees in the same pool. Each delegatee will have full control over the owner's positions in a single pool.

Note that asset transfers will be to and from the caller, the delegatee. Consequently, this functionality cannot be utilized for operational segregation, e.g. separating asset ownership from a less privileged account that is intended solely for administrative modifications on positions.

## 8.3 Extension Can Directly Retrieve From Reserve

Note Version 1

`retrieve_from_reserve()` has been implemented in the singleton, which enables the extension to directly transfer any configured asset from reserve to any designated receiver. The extension of a pool should be checked and is assumed to be fully trusted by the users, otherwise, a malicious extension can:

- Directly drain all user's funds.
- Decrease the reserve of an asset so that positions backed by this asset become insolvent.

At the time of this review, the default extension features no functionality to call `retrieve_from_reserve()` on singleton.

## 8.4 Loosely Restricted Configurations

Note Version 1

The protocol is designed to be used in a permissionless way and it loosely restricts the configuration of a pool at its creation. In case the parameters are misconfigured, the pool will not function correctly and user's funds will be at risk. The following non-exhaustive list details various important considerations:

- `fee_rate` could be larger than `100%` which will take part of user's collateral as fees in addition to the yield generated from the loan.
- `last_rate_accumulator` could be set to 0, which will make `asset_config_exists()` return false on this asset config. In case it is too large, it can overflow or trigger the violation easily

```
let safe_rate_accumulator = collateral_accumulator < 18 * SCALE &&
    debt_accumulator < 18 * SCALE;
```

- `max_position_ltv` is not restricted within `100%`, and it should not be a legitimate configuration if it is above `100%`.
- `liquidation_discount` (renamed to `liquidation_factor` in the latest version) should not exceed `max_position_ltv`, otherwise it would be profitable to liquidate oneself by repaying less debt while getting all collateral back.
- There are no sanity checks in the default extension when setting the interest rate model:

    1. The order of `min_target_utilization` and `max_target_utilization` (also `zero_utilization_rate`, `min_full_utilization_rate` and `max_full_utilization_rate`) is not enforced.
    2. `max_target_utilization` can be set to 0, which can bypass the model existence check and set the interest model of the same asset again.
    3. `target_rate_percent` is not restricted to be below `100%`. In case it is above `100%`, the `target_rate` at `target_utilization` will be even higher than the `full_utilization_rate`.

## 8.5 Missing Repay in Subscription Phase Results in Locked Collateral

Note Version 2

In case a pool fails to recover during the recovery period, the pool will enter the subscription phase and will eventually be shutdown. The `subscription_period` is restricted to be no less than 1 day at pool creation.

Users are expected to repay all their debt in time during the subscription phase. During the redemption phase no repayment of debt is possible and collateral can only be withdrawn from positions with zero debt.

This is designed to incentivize borrowers to repay their debt as soon as possible in the subscription phase as the lenders may be unable to fully withdraw their funds if the same token has not been fully repaid by the borrowers. However, should lenders miss this timeframe, collateral may be locked forever.

## 8.6 Race Condition When Bad Debt Is Redistributed

**Note** **Version 1**

In case bad debt will occur in a liquidation, the bad debt will be immediately absorbed by the reserve thus redistributed among all the depositors of this asset in this pool. A race condition may occur among the users to withdraw the funds before bad debt occurs hence only the remaining users who have not exited will share the bad debt.

## 8.7 Read Only Reentrancy

**Note** **Version 1**

A Mandatory callback (`after_modify_position()`) will be invoked on the extension after position modifications. During the callback, the pool is in a intermediate state where internal accounting has been updated but the underlying assets haven't been transferred. Third-party systems that integrate Vesu should be aware of this behavior and mitigate the potential risks of reading intermediate states.

## 8.8 Rehypothecation May Lead to Cascading Liquidations

**Note** **Version 1**

In case rehypothecation is supported where a chain of collateral is configured in a pool, users are further exposed to the risks of cascading liquidation if bad debt occurs. For instance:

- Assume there is a chain of pairs configured in a pool: A->B->C (User can deposit A to borrow B, and deposit B to borrow C). Assume the starting price of all three assets are 1 usd.
- Assume the total assets in the pool is (A:100, B:100, C:100).
- Assume Alice has the position: collateral-(A:100, B:0, C:0) | debt-(A:0, B:80, C:0).
- Assume Bob has the position: collateral-(A:0, B:100, C:0) | debt-(A:0, B:0, C:80).
- In case the price of asset A falls to 0.5 and liquidation discount is 0.8. Alice got liquidated and 40 bad debt was absorbed by the pool.
- Now the pool's total assets becomes (A:0, B:60, C:100).
- Due to the bad debt, now Bob's 100 shares of B only worth 60 usd, which immediately triggers insolvency of Bob's position and Bob will be liquidated.

Users should be fully aware of this risk when using a pool, otherwise, users may suffer from cascading liquidations in case of a black swan event.