PUBLIC

Code Assessment

of the Vesu V2 Smart Contracts

September 30, 2025

Produced for

by

Contents

1 Executive Summary

Dear all,

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 V2 according to Scope to support you in forming an opinion on their security risks.

VESU implemented Vesu V2 focusing on simplifications of the original Vesu V1 design. Separate pools are deployed by the factory instead of the singleton architecture used in V1.

The most critical subjects covered in our audit are functional correctness and asset solvency. One low severity issue highlighting the effect of rounding in partial liquidations with bad debt has been reported, see Bad debt rounding can be exploited to pay off less debt. All the uncovered issues have been properly addressed.

The general subjects covered are usability, code quality and potential risks.

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

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

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

Sincerely yours,

ChainSecurity

1.1 Overview of the Findings

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3
• Code Corrected	3

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 V2 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	10 September 2025	37d0e422addc84f445f66ed5ef8bc5f31901e0b7	Initial Version
2	27 September 2025	9e681bcf76f168698e84b8e3e9d90106ea0ca370	After Intermediate Report
3	30 September 2025	977a04feb7efb67bdf6669c1d7885adb099d5308	Final Changes

For the cairo smart contracts, the compiler version `2.11.4` was chosen.

The following files are in scope of this review:

```
pool.cairo
pool_factory.cairo
oracle.cairo
```

In addition, changes in the following files are in scope. Previously reviewed logic is excluded:

```
v_token.cairo
units.cairo
packing.cairo
math.cairo
interest_rate_model.cairo
data_model.cairo
common.cairo
lib.cairo
```

2.1.1 Excluded from scope

Unchanged logic previously audited is excluded from this review. Test files and deployment scripts are also excluded from scope.

2.2 System Overview

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

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

VESU implemented Vesu V2, a permissionless lending protocol (with emergency features restricted to a Vesu admin address), focusing on simplifications of the original Vesu V1 design, built on four main contracts: `PoolFactory`, `Pool`, `Oracle` and the `VToken`. Separate pools are deployed by the factory instead of the singleton architecture used in v1.

2.2.1 Pool Creation

Pools are created permissionlessly through the factory.

When `create_pool` is called, a new pool is deployed with the specified name, curator, oracle and fee recipient. The function registers the initially supported assets and their pair configurations, selectively enabling loans per pair. For each listed asset, the factory deploys an associated `VToken`. The factory exposes helpers to look up the `VToken` for an asset and the asset for a `VToken`.

Curators can also add new assets to existing pools through the factory's `add_asset` function.

2.2.2 Modify Position

`modify_position` is the main entry point for users and delegates to modify their position in a pool. It changes collateral and debt of positions, ensures invariants and returns the applied deltas.

The function updates the interest accumulator for the debt asset, computes utilization, and applies fees. It enforces collateralization using current prices from the oracle. It supports four operation types through the amount denomination:

- Collateral in asset units
- Collateral in collateral shares
- Debt in asset units
- Debt in nominal debt units

Transfers of underlying tokens are executed via safe transfer helpers that support legacy ERC20 interfaces.

For simple lending to earn interest, one can open a position with no debt on any pair whose collateral asset is the desired lending asset. The choice of the pair is indifferent, as the interest rate is configured per-asset, not per-pair. The Vtoken makes use of this.

2.2.3 Position Delegation

Positions remain tied to their original owner address and cannot be transferred. However, users can grant management permissions through `modify_delegation`.

The relevant functions are:

- `modify_delegation(delegatee, delegation)` - grants or revokes delegation rights
- `delegation(delegator, delegatee)` - checks current delegation status

Delegatees can call `modify_position` on behalf of the owner (funds move to/from the caller, not the position owner), but cannot transfer ownership. Users retain full control and can revoke delegation at any time.

2.2.4 Liquidate Position

`liquidate_position` lets third parties repay debt of unhealthy position and seize collateral according to the pair configuration.

Health is evaluated using the oracle price and pair risk parameters. The function updates interest first then computes repayment and collateral seizure, including liquidation fees and reserve updates. If bad debt is caused to the system by a position going underwater, it gets socialized upon liquidation.

2.2.5 Price feed

Prices come from the `Oracle` contract, which serves as an adapter to integrate with Pragma price feeds. The oracle stores per asset configuration with fields:

- pragma key
- timeout in seconds
- number of sources
- start time offset
- time window
- aggregation mode

The oracle queries Pragma contracts and processes the responses according to the configuration, validating freshness and source requirements. The pool calls the oracle to obtain an `AssetPrice` for collateral and debt assets when evaluating health, mint, burn, and liquidation.

The curator can update the oracle address on the pool. The oracle's manager can update per asset oracle config on the oracle itself.

2.2.6 Pausing

The pool supports pausing. The owner, curator, or pausing agent can pause the pool. Only the owner or curator can unpause.

When paused, interactions with the pool are generally inhibited. Some configuration functions, e.g. `set_oracle()`, remain possible when the pool is paused, allowing configuration issues to be fixed in a paused state.

2.2.7 Tokenization of Collateral Shares

`VToken` implements an ERC4626 style vault interface for each supported asset. It exposes:

- `asset` and `total_assets`
- `convert_to_shares` and `convert_to_assets`
- `preview_deposit` and `deposit`
- `preview_mint` and `mint`
- `preview_withdraw` and `withdraw`
- `preview_redeem` and `redeem`
- the standard ERC20 views and transfers

Underlying the `VToken` there is a regular pool position, therefore the usual limitations apply: e.g. a `max_utilization` limit may prevent token redemptions.

2.2.8 Interest Rate Model

The pool embeds an interest rate model component. The adaptive interest rate is crucial to incentivize user interaction for the pool to reach/maintain it's target utilization. The curator can update the interest settings per asset of the pool using `set_interest_rate_parameter`.

The model computes the new rate accumulator and the full utilization rate based on current utilization and the previous state. Helper views expose the interest rate configuration.

2.2.9 Changelog

In (Version 2), the following changes were made to the system:

1. The factory is now upgradable without delay by the owner.
2. A new `update_v_token()` method was added to the factory, allowing to update the vToken-asset mapping.

In Version 3, the following changes were made to the system:

1. New setter functions have been added to the pool factory allowing the owner to update the class hashes for the pool, v_token and oracle. Future deployments will use these updated class hashes.

2.3 Trust Model

2.3.1 Factory

- **Owner** owns all deployed pools and can upgrade the factory contract itself.

  Trust level: fully trusted. Worst case: can upgrade all deployed pools and the factory with malicious logic.

- **Anyone** can call `create_pool` with valid parameters.

  Trust level: untrusted. Pools created under the correct class hashes remain safe.

- **Factory** deploys `VToken` instances per asset and wires them to the pool.

2.3.2 Pool

- **Owner** (the factory owner) can upgrade the contract via `upgrade`.

  Trust level: fully trusted. Worst case: replace pool implementation with malicious logic.

- **Curator** can set pausing agent, fee recipient, oracle address, and nominate successor curator; adjusts risk parameters via pair and asset configuration setters.

  Trust level: fully trusted. Worst case: switch to malicious oracle, making borrowers liquidatable then liquidating them himself. The owner is expected to detect such misbehaviour in time and remedy through an upgrade.

- **Pausing agent** can pause (but not unpause).

  Trust level: partially trusted. Worst case: cause denial of service by pausing, and cause losses if in times of market volatility users become liquidatable due to the DoS. The curator is expected to detect such misbehaviour in time and revoke the role from the compromised address.

- **Fee recipient** receives protocol fees through `claim_fees`

  Trust level: untrusted. No control over user funds.

- **Users** control their own positions and can grant or revoke delegation with `modify_delegation`

  Trust level: untrusted from protocol perspective. Can only affect their own balances.

2.3.3 Oracle

- **Owner** controls ownership transfer and can upgrade the oracle contract.

  Trust level: fully trusted. Worst case: replace oracle with malicious implementation / configuration.

- **Manager** can set per asset `OracleConfig` and nominate a new manager.

  Trust level: fully trusted. Worst case: misconfigure oracle sources, leading to wrong prices used within the pool. This may allows to steal assets. The owner is expected to detect this misbehaviour in time and remedy with an upgrade so as to revoke the role from the compromised address.

- The oracle reads from upstream Pragma contracts and aggregates per the stored configuration

  Assumption: Pragma sources report correct prices.

2.3.4 VToken

- Users interact through ERC20 and ERC4626 interfaces.

  Trust level: untrusted. They can only affect their own balances.

2.3.5 Special Observations

- Some emergency situations (like a pool curator compromise) are expected to only be remedied through an upgrade by the Vesu owner address.
- The pool curator has to have an understanding of risk parameter design, since it's the one in charge of configuring the oracle and the asset/pairs in the pools.
- Upgrades are restricted to the Vesu owner address, even though pools can be curated by anyone. As explained, the curator can still rug-pull users through malicious configurations, but users can be sure that the pools are running legit Vesu code.

2.3.6 External Dependencies

- **Pragma Oracle** provides price data through `IPragmaABI` and `ISummaryStatsABI`

  Trust level: fully trusted. The system assumes Pragma is properly configured, returns accurate prices, and remains available. *Worst case*: Incorrect or manipulated Pragma prices could lead to wrongful liquidations or undercollateralized borrowing.

- **ERC-20 Tokens** used as assets in the pool

  Trust level: fully trusted. The system assumes tokens are standard compliant without special behaviors. *Required properties*: No blacklists, no transfer fees, non-rebasing, standard `transfer`/`transferFrom` behavior. *Worst case*: Tokens with special mechanics could break accounting, prevent withdrawals, or cause loss of funds.

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 Open 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	0
Low-Severity Findings	0

6 Resolved Findings

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

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

Critical-Severity Findings	0
High-Severity Findings	0
Medium-Severity Findings	0
Low-Severity Findings	3

- claim_fees() Rounding up Code Corrected
- Bad Debt Rounding Can Be Exploited to Pay off Less Debt Code Corrected
- VToken Checks Invariants on the Caller Address Code Corrected

Informational Findings	10

- Liquidation Rounding May Prevents Full Liquidation in Edge Cases Code Corrected
- Natspec Describes Wrong Asset Approval Code Corrected
- Unused Constant Code Corrected
- Event SetOracleConfig Code Corrected
- Outdated Comment Code Corrected
- Redundant Multiplication and Division by SCALE Code Corrected
- Redundant Non-Zero Address Checks Code Corrected
- Redundant Position Argument Code Corrected
- Small Liquidation Factor Implies Immediate Bad Debt Code Corrected
- VToken Configuration Allows Debt Asset Equal to Asset Code Corrected

6.1 `claim_fees()` Rounding up

Design Low Version 2 Code Corrected

CS-VESUV2-013

In Version 2, the `claim_fees` function was refactored. Among other changes, the calculation of the fee amount was modified to round up instead of down.

This rounding direction is inconsistent with regular share to asset conversions during withdrawals, which round down. The (privileged) caller could exploit this by repeatedly calling `claim_fees()` with 1 wei of fee shares per call. Each call would round up the resulting fee amount, allowing the caller to extract slightly more fees than the shares entitle them to. Through repeated calls, this can accumulate.

The `get_fees()` preview function also implements the same rounding up behavior.

Code corrected

The fee amount calculation was changed to round down.

6.2 Bad Debt Rounding Can Be Exploited to Pay off Less Debt

Security | Low | Version 1 | Code Corrected

CS-VESUV2-007

In case there is bad debt, for partial liquidations the function `compute_liquidations_amounts()` rounds *up* the calculated `bad_debt`; also, the code does not enforce a minimum amount of debt to repay. Therefore, one can supply a `debt_to_repay` parameter equal to e.g. 1 wei, and if the position is slightly underwater, the returned `bad_debt` will be equal to 1 wei. This means that the liquidator repays 1 wei, but writes off 2 weis from the position's debt. This 2-to-1 ratio is skewed from the intended one, particularly when the position is only slightly underwater (therefore the ratio between the debt value and the discounted collateral value is slightly above 1).

Notice that an "external" liquidator cannot profit from this, since the collateral to be received and the loan amount to be repaid are not affected by roundings in the bad debt. The only effect is that the loss inflicted to the system (written off from the position's debt) proportionally exceeds the real one. This can therefore be exploited by the position owner himself liquidating his own position, and thus achieving a better effective liquidation discount than configured by the curator.

The attached PoC shows how the owner of a position that is slightly underwater can iterate this several million times to bring the position back to health, by repaying almost half of the needed debt amount.

Code corrected

The issue was fixed by always rounding *down* the bad debt (which renders the PoC's exploit scenario impossible), and enforcing a minimum liquidation amount for partial liquidations (which removes the root cause, i.e. that large relative errors were possible).

6.3 VToken Checks Invariants on the Caller Address

Correctness | Low | Version 1 | Code Corrected

CS-VESUV2-003

The function `can_modify_position()` of `v_token.cairo()` invokes `check_invariants()` on the pool, but wrongly passes `get_caller_address()` as a parameter, instead of `get_contract_address()`. This can unduly block access to the VToken for a user who, for example, has a dusty position (hence `assert_floor_invariants()` reverts).

Code corrected

The check was fixed to pass `get_contract_address()`.

6.4 Liquidation Rounding May Prevents Full Liquidation in Edge Cases

Informational | Version 2 | Code Corrected

CS-VESUV2-011

In Version 2, the rounding direction in `compute_liquidation_amounts()` was changed from `Rounding::Ceil` to `Rounding::Floor` when determining whether to perform a full or partial liquidation in case of bad debt:

```
if collateral_value < debt_value {
    // limit the bad debt by the outstanding collateral and debt values (in usd)
    if collateral_value < u256_mul_div(
        debt_to_repay, context.debt_asset_price.value, context.debt_asset_config.scale, Rounding::Floor,
    ) {
```

This condition checks whether the position's collateral value is less than the value of the debt the liquidator is attempting to repay. If true, the position is fully liquidated (entering this branch). If false, only a partial liquidation occurs.

In edge cases, this introduces a one wei gap between the `debt_value` (rounded up) in the outer condition and the recomputed value (rounded down) here. If `collateral_value` equals the rounded-down value, the code incorrectly enters the partial liquidation branch, writing off almost all debt and leaving a dusty position that violates the floor invariant and reverts.

Code corrected

In Version 3, the rounding direction was changed back to `Rounding::Ceil`.

6.5 Natspec Describes Wrong Asset Approval

Informational | Version 2 | Code Corrected

CS-VESUV2-009

In Version 2, the vToken was refactored. The functionality for the vToken to grant the pool allowance to spend its underlying asset was moved into its own function.

While the function implementation is correct, the natspec incorrectly states "Approves the pool contract to spend the vToken".

Code corrected

The natspec was updated to "Approves the pool contract to spend the underlying asset of the vToken".

6.6 Unused Constant

Informational | Version 2 | Code Corrected

CS-VESUV2-001

As of Version 2, the constant `SCALE_DECIMALS` is unused and can be removed from the codebase

---

Code corrected

The constant has been removed.

6.7 Event SetOracleConfig

Informational | Version 1 | Code Corrected

CS-VESUV2-002

Oracle.cairo defines an event `SetOracleConfig`.

```
#[derive(Drop, starknet::Event)]
pub struct SetOracleConfig {
    asset: ContractAddress,
    oracle_config: OracleConfig,
}
```

Contrary to other events in the contracts, no field is annotated with `#[key]`. Annotating the field `asset` with `#[key]` would enable filtering by the asset.

Code corrected

The event field is now annotated with `#[key]`.

6.8 Outdated Comment

Informational | Version 1 | Code Corrected

CS-VESUV2-006

The comment above `compute_liquidation_amount()` mentions recovery mode which no longer exists in this version of Vesu.

Code corrected

The comment was removed.

6.9 Redundant Multiplication and Division by SCALE

Informational | Version 1 | Code Corrected

CS-VESUV2-008

The function `calculate_collateral()` of `common.cairo` uses a precision of `SCALE` to compute its proportion. This is, however, redundant, because the result is immediately divided by `SCALE` after the multiplication, without any intermediate divisions actually requiring the upscaled precision.

Code corrected

The scaling was removed.

6.10 Redundant Non-Zero Address Checks

Informational | Version 1 | Code Corrected

CS-VESUV2-012

The functions `accept_manager_ownership()` of `oracle.cairo` and `accept_curator_ownership()` of `pool.cairo` perform a redundant non-zero check on the pending manager/curator. It can be omitted because it is already checked to be equal to `get_caller_address()`, therefore it cannot be 0.

Code corrected

The check was removed.

6.11 Redundant Position Argument

Informational | Version 1 | Code Corrected

CS-VESUV2-004

`calculate_collateral_and_debt_value()` in `common.cairo` takes `context` (which is bound to a user address) and `position` as arguments. This is redundant since the context already holds the user position.

Code corrected

The redundant argument was removed.

6.12 Small Liquidation Factor Implies Immediate Bad Debt

Informational | Version 1 | Code Corrected

CS-VESUV2-005

If `liquidation_factor < max_ltv`, then a position incurs bad debt as soon as it becomes liquidatable. This is because, as soon as the position is liquidatable, we have `debt_value > collateral_value * max_ltv`, but that implies `debt_value > collateral_value * liquidation_factor = discounted_collateral_value`.

If, instead, `liquidation_factor > max_ltv`, then there is some leeway between the time where a position is liquidatable and the time where it causes bad debt to the system.

Code corrected

A check now enforces that `liquidation_factor >= max_ltv`. Note that, in case of equality, the issue still applies. In general, the curator has to choose a large-enough leeway that is appropriate for the asset.

6.13 VToken Configuration Allows Debt Asset Equal to Asset

Informational | Version 1 | Code Corrected

CS-VESUV2-010

The VToken creation flow allows to configure it with `asset == debt_asset`. However, this configuration will lead to reverts in the user flows, as the function `pool.context()` always reverts if `collateral_asset == debt_asset`.

Code corrected

A sanity check was introduced in the function `pool_factory.create_v_token()` to prevent this particular misconfiguration. The VToken constructor, however, does not feature this sanity check.

7 Notes

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

7.1 Liquidations Must Always Be Possible

Note Version 1

The liquidation of unhealthy positions is crucial to maintain a pool's health.

Vesu allows pools to configure a `max_utilization` per asset. Liquidations are not subject to this limit and can draw from the full asset reserve of the pool. The buffer created by the max utilization setting is intended to ensure that enough liquidity is always available for liquidations. When liquidity is constrained, liquidators can work around this by first supplying liquidity to the pool and then liquidating, achieving the same result as receiving collateral shares directly.

However, this workaround relies on liquidators having sufficient capital and willingness to supply liquidity first. In extreme market conditions, liquidators may lack the resources or risk appetite to do so. If liquidations are delayed, positions can deteriorate further if the collateral value drops, if the debt value rises, or if the debt increases over time due to interest accrual. If the liquidation is delayed and the collateral no longer covers the debt, this results in bad debt.

Worse, before the liquidation occurs, this bad debt isn't socialized. Anyone who recognizes a liquidation resulting in bad debt early has a strong incentive to withdraw their position before that liquidation happens.

Pool operators should therefore set conservative `max_utilization` parameters to minimize reliance on the liquidity-supply workaround during stressed market conditions.