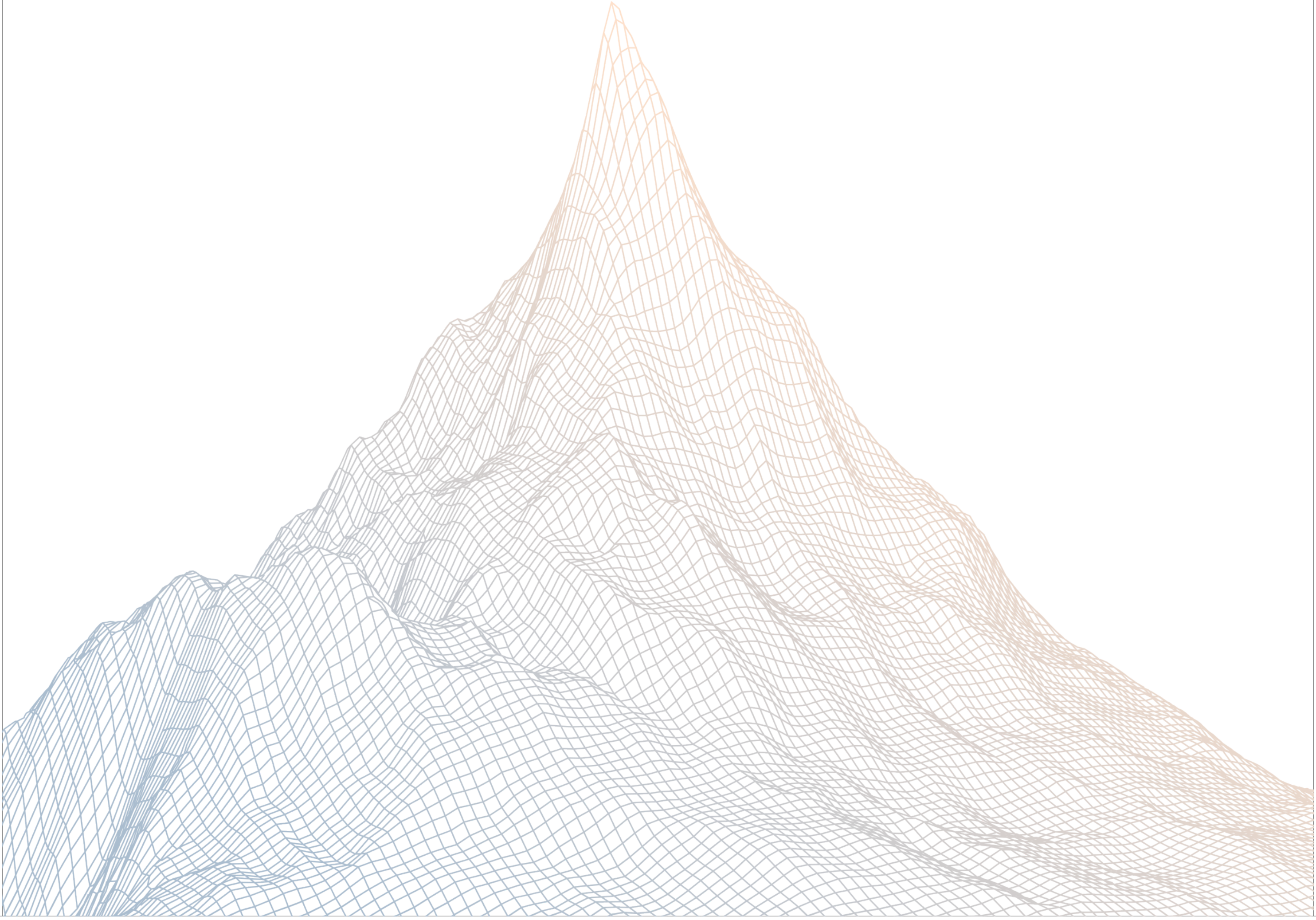
Zenith

Vesu

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: August 8th to September 2nd, 2025

AUDITED BY: J4X
Mario Poneder

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Vesu

Vesu is a fully open and permissionless lending protocol built on Starknet. Users can supply crypto assets (earn), borrow crypto assets and build new lending experiences on Vesu without relying on intermediaries. The Vesu lending protocol is not controlled by a governance body and there exists no governance token. Instead, Vesu is built as a public infrastructure giving everyone equal access to all functions and is free for everyone to use.

2.2 Scope

The engagement involved a review of the following targets:

Target	vesu-v1
Repository	https://github.com/vesuxyz/vesu-v1
Commit Hash	`1c746477a6ef490e5d4fb076046365544d5fb769`
Files	`src/* (excluding /vendor folder)`

2.3 Audit Timeline

August 8, 2025	Audit start
September 2, 2025	Audit end
October 13, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	3
Low Risk	9
Informational	10
Total Issues	**23**

3

Findings Summary

ID	Description	Status
H-1	Scaling mismatch in interest rate model denominator leading to flat rates above target utilization	Resolved
M-1	Liquidator will unknowingly overpay on full liqudiation	Acknowledged
M-2	Price validity checks are neglected in several code paths	Acknowledged
M-3	Fee transfer could silently fail leading to stuck fees	Resolved
L-1	Division by zero risk in full_utilization_rate function due to invalid max_target_utilization validation	Resolved
L-2	ERC4626 entrypoints in v_token/v_token_v2 skip explicit shutdown gating	Acknowledged
L-3	Oracle could return a single price as TWAP	Acknowledged
L-4	Oracle can be configured to return outdated prices	Acknowledged
L-5	Rounding direction in calculate_withdrawable_assets favors users	Resolved
L-6	Missing check for enough asset to asset configs on pool creation	Acknowledged
L-7	The target_utilization is not bounded by min_target_utilization and max_target_utilization	Resolved
L-8	set_shutdown_mode emits wrong mode	Resolved
L-9	Insufficient support of fee-on-transfer tokens leading to accounting discrepancies	Acknowledged
I-1	flash_loan NatSpec omits is_legacy: bool argument.	Resolved
I-2	is_legacy in flashloan can be passed incorrectly leading to revert	Acknowledged
I-3	Incorrect description of assert_ownership()	Resolved
I-4	Incorrect NatSpec for rate_accumulator()	Resolved
I-5	Division by zero and undefined mathematical operations in pow_scale and pow functions	Acknowledged

ID	Description	Status
I-6	Missing round_up in Natspec for multiple common functions	Resolved
I-7	Slightly underestimated utilization in calculate_utilization function due to rounding	Acknowledged
I-8	Inconsistent rounding in calculate_fee_shares undermines intended conservative fee calculation approach	Resolved
I-9	Migrator can only be set by migrator leading to potential blockage	Acknowledged
I-10	retrieve_from_reserve can underflow without error	Resolved

4 Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Scaling mismatch in interest rate model denominator leading to flat rates above target utilization

SEVERITY: High	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: High

Target

- src/extension/components/interest_rate_model.cairo#L296

Description:

The interest rate model in `src/extension/components/interest_rate_model.cairo` contains a scaling mismatch in the `calculate_interest_rate` function when `utilization ⩾ target_utilization`.

Affected code:

```
let new_rate_per_second = if utilization < target_utilization {
    zero_utilization_rate + (utilization * (target_rate -
    zero_utilization_rate)) / target_utilization
} else {
    target_rate
        + ((utilization - target_utilization) *
    (next_full_utilization_rate - target_rate))
            / (SCALE - target_utilization)    // ⟵ scaling mismatch
    (target_utilization ~ UTILIZATION_SCALE)
};
```

Scale values:

- `SCALE = 1e18` - from `units.cairo`
- `UTILIZATION_SCALE = 1e5` - from `interest_rate_model.cairo`
- `utilization` and `target_utilization` are in `UTILIZATION_SCALE` units

Mathematical impact:

- `SCALE - target_utilization` ≈ `1e18 - 1e5` ≈ `1e18`
- The denominator becomes effectively constant at `1e18`, instead of being < `1e5` as intended
- The slope calculation `(utilization - target_utilization) * (next_full_utilization_rate - target_rate) / 1e18` produces negligible interest rate changes
- Interest rates above target utilization become practically flat instead of increasing, i.e. `new_rate_per_second ≈ target_rate`

Protocol functions impacted in first order:

- `calculate_interest_rate`
- `interest_rate`
- `rate_accumulator`

Protocol operations impacted in second order via *`asset_config.last_rate_accumulator`***:**

- Interest accrual
- Fee calculations
- Collateral and debt calculations
- Position modifications
- Liquidations

The interest rate model is non-functional above target utilization, potentially causing the protocol to fail economically through incorrect interest calculations.

Recommendations:

It is recommended to change the denominator from `SCALE - target_utilization` to `UTILIZATION_SCALE - target_utilization`.

Vesu: Resolved with PR-5.

Zenith: Verified.

4.2 Medium Risk

A total of 3 medium risk findings were identified.

[M-1] Liquidator will unknowingly overpay on full liqudiation

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- src/extension/components/position_hooks.cairo#L707-L714

Description:

The calculation of the liquidation amounts paid/received is done in the `before_liquidate_position` hook. If the position is so undercollateralized that it does not have enough collateral to cover the debt repayment by the liquidator sufficiently, the following calculation is done.

```
bad_debt =
    u256_mul_div(
        debt_value - collateral_value,
        context.debt_asset_config.scale,
        context.debt_asset_price.value,
        Rounding::Floor,
    );
debt_to_repay = debt;
```

Here, `bad_debt` will be rounded down, and the `debt_to_repay` will be set to the full debt in the position. Later on in `settle_position`, the user will need to repay `debt_to_repay - bad_debt`. However, as `bad_debt` was rounded down and the user's original `debt_to_repay` was overwritten, this might result in the user overpaying his intended `debt_to_repay`.

Recommendations:

We recommend rounding the bad debt up in this case.

Vesu: Acknowledged.

[M-2] Price validity checks are neglected in several code paths

SEVERITY: Medium	IMPACT: Medium
STATUS: Acknowledged	LIKELIHOOD: Medium

Target

- src/extension/components/pragma_oracle.cairo#L97-L138
- src/extension/default_extension_po_v2.cairo#L949-L958
- src/singleton_v2.cairo#L1254-L1336
- src/singleton_v2.cairo#L945-L990
- src/singleton_v2.cairo#L570-L598
- src/singleton_v2.cairo#L625-L639

Description:

The protocol has inconsistent price validity checking that can lead to safety assertions or collateralization checks being ineffective when oracle prices are invalid. While invalid prices correctly block liquidations and trigger Recovery mode in the `shutdown_status` function, several code paths still use potentially invalid price data for calculations without validation.

The oracle's `price` function correctly returns both, the scaled price value and a validity flag (`is_valid`). However, the `context_unsafe` and `context` functions in `singleton_v2.cairo` simply pass through this price data without assessing the validity status. This creates a gap where invalid prices can propagate through the system.

Affected functions:

1. `check_collateralization_unsafe` and `check_collateralization`: These functions calculate collateral and debt values to assess the collateralization state using `calculate_collateral_and_debt_value`, which relies on context data containing potentially invalid prices.

2. `assert_collateralization`: Directly uses the calculated values from context without price validation.

3. `assert_position_invariants`: Relies on `calculate_collateral_and_debt_value` which uses potentially invalid price data.

4. `assert_floor_invariant`: Also depends on context data with unvalidated prices for floor checks.

Furher details: The `calculate_collateral_and_debt_value` function in `common.cairo` performs USD value calculations using `context.collateral_asset_price.value` and `context.debt_asset_price.value` without checking the corresponding `is_valid` flags. This means that even when the protocol is in Recovery mode due to invalid prices, these core calculations continue to use potentially invalid price data.

Recommendations:

It is recommended to revise the usage of price data without validation and add necessary safety checks without impeding the intended Recovery flows when modifying and transferring positions.

Vesu: Acknowledged. The protocol needs to accept invalid prices in case it's in an advanced shutdown mode for completely autonomous unwinding of the assets in the pool.

[M-3] Fee transfer could silently fail leading to stuck fees

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- src/extension/components/fee_model.cairo#L138

Description:

The `claim_fees()` function is used to claim the fees accrued by the fee receiver. After calculation, it transfers the fees using the ERC20's `transfer()` function.

```
IERC20Dispatcher { contract_address:
    collateral_asset }.transfer(fee_config.fee_recipient, amount);
```

However, this call does not check the return value of the `transfer()` function. Per the ERC20 standard, a transfer does not need to revert on error, but returning false is required. As a result, the token could, for example, be paused, which would lead to no tokens being transferred and the funds getting stuck in the extension.

Recommendations:

We recommend checking the return value of the transfer to revert the whole transaction if the transfer failed.

Vesu: Resolved with PR-8.

Zenith: Verified.

4.3 Low Risk

A total of 9 low risk findings were identified.

[L-1] Division by zero risk in `full_utilization_rate` function due to invalid `max_target_utilization` validation

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Resolved	`LIKELIHOOD`: Low

Target

- src/extension/components/interest_rate_model.cairo#L95
- src/extension/components/interest_rate_model.cairo#L330

Description:

The `assert_interest_rate_config` function in `src/extension/components/interest_rate_model.cairo` contains a validation flaw that could lead to division by zero in the `full_utilization_rate` function.

Affected code:

```
// Line 95: Validation allows max_target_utilization to equal
    UTILIZATION_SCALE
assert!(max_target_utilization ≤ UTILIZATION_SCALE,
    "max-target-utilization-gt-100%");

// Lines 329-330: Division by zero risk
let utilization_delta = ((utilization - max_target_utilization) * SCALE)
    / (UTILIZATION_SCALE - max_target_utilization);    // ⟵ division by
    zero (when max_target_utilization = UTILIZATION_SCALE )
```

Recommendations:

It is recommended to apply the following changes:

```
 assert!(max_target_utilization ≤ UTILIZATION_SCALE, "max-target-
     utilization-gt-100%");

   assert!(max_target_utilization < UTILIZATION_SCALE, "max-target-utilization
     -gte-100%");
```

Vesu: Resolved with @7b2f3c1....

Zenith: Verified.

[L-2] ERC4626 entrypoints in `v_token`/`v_token_v2` skip explicit shutdown gating

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- src/v_token.cairo#L177-L191
- src/v_token_v2.cairo#L184-L198

Description:

Both implementations (`v_token` and `v_token_v2`) expose ERC4626 entrypoints (`deposit`, `mint`, `withdraw`, `redeem`) without asserting pool shutdown gates locally:

- `deposit` and `mint` do not assert `can_deposit`.
- `withdraw` and `redeem` do not assert `can_withdraw`.

While `max_*` and `preview_*` functions do check these gates, the state-changing entrypoints themselves don't. Currently, shutdown enforcement happens indirectly inside the pool's position hook (`after_modify_position`) when using the default extension.

If a non-default extension is used, a custom extension omits/weakens these checks, the ERC4626 methods can execute in prohibited modes (e.g., deposit during Subscription/Redemption or withdrawal during Recovery/Subscription), causing a mismatch between previews and execution and enabling unwanted operations.

Recommendations:

It is recommended to add explicit guards at the top of ERC4626 entrypoints in both versions.

Vesu: Acknowledged. The security model assumes a trusted Extension and VToken implementation. So this should not be a problem.

[L-3] Oracle could return a single price as TWAP

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/extension/components/pragma_oracle.cairo#L118

Description:

The `price()` function uses a TWAP to retrieve a mean or median price. To define over which timeframe this price should be calculated, the `time_window` parameter is passed.

```
let (value, decimals) = summary
    .calculate_twap(
        DataType::SpotEntry(pragma_key),
        aggregation_mode,
        time_window,
        get_block_timestamp() - start_time_offset,
    );
```

Based on the documentation, the pragma oracle will save a checkpoint every 5 minutes and then calculate the twap based on those checkpoints.

In the current implementation, there is no check present that ensures that the `time_window` is big enough so that it covers more than one checkpoint, resulting in the twap using a potentially older price than the returned one.

Recommendations:

We recommend ensuring that `time_window` is at least 25 minutes to ensure that a minimum of 5 checkpoints are included in the TWAP.

Vesu: Acknowledged

[L-4] Oracle can be configured to return outdated prices

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/extension/components/pragma_oracle.cairo#L118

Description:

The `price()` function is used to retrieve the price of an asset from the pragma oracle. If both `start_time_offset` and `time_window` are configured, the function will request a TWAP instead of using the returned price.

```
let summary = ISummaryStatsABIDispatcher { contract_address:
    self.summary_address.read() };
let (value, decimals) = summary
    .calculate_twap(
        DataType::SpotEntry(pragma_key),
        aggregation_mode,
        time_window,
        get_block_timestamp() - start_time_offset,
    );
u256_mul_div(value.into(), SCALE, pow_10(decimals.into()), Rounding::Floor)
```

The twap works based on two values. The `start_time_offset` is used to reduce the current timestamp and start from there, and the `time_window` will be the number of seconds the twap goes up from that.

However, currently, nothing ensures that these cover the current price. Theoretically, `start_time_offset` can be 10 years, and `time_window` can be one day. As a result, the oracle will happily return an outdated price.

Recommendations:

We recommend adapting the functionality so that it uses `start_time_offset` instead of `time_window` as the `time` parameter passed to `calculate_twap`.

```
let summary = ISummaryStatsABIDispatcher { contract_address:
    self.summary_address.read() };
let (value, decimals) = summary
    .calculate_twap(
        DataType::SpotEntry(pragma_key),
        aggregation_mode,
        start_time_offset,
        get_block_timestamp() - start_time_offset,
    );
u256_mul_div(value.into(), SCALE, pow_10(decimals.into()), Rounding::Floor)
```

This way, the program ensures that the latest checkpoint is included in the TWAP. Additionally, we recommend setting a maximum for the `start_time_offset` to prevent outdated checkpoints from being included in the twap.

Vesu: Acknowledged.

[L-5] Rounding direction in `calculate_withdrawable_assets` favors users

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/v_token.cairo#L131-L151
- src/v_token_v2.cairo#L138-L158

Description:

The `calculate_withdrawable_assets` function in both `v_token.cairo` and `v_token_v2.cairo` contains multiple rounding direction choices that systematically favor users over the protocol, potentially allowing withdrawals that violate the intended maximum utilization constraints.

1. Utilization calculation rounds down. This underestimates the actual utilization, making the pool appear less utilized than it actually is.

```
let utilization = u256_mul_div(total_debt, SCALE, asset_config.reserve +
    total_debt, Rounding::Floor);
```

2. Inner and outer multiplications round down. This implicitly rounds up the withdrawable amount due to the subtraction.

```
(asset_config.reserve + total_debt)
 - u256_mul_div(
     total_debt,
     (u256_mul_div(SCALE, scale, asset_config.max_utilization,
    Rounding::Floor)),
     scale,
     Rounding::Floor,
 )
```

Recommendations:

It is recommended to change the rounding direction in all of the above `u256_mul_div` instances to `Rounding::Ceil`.

Vesu: Resolved with PR-7.

Zenith: Verified.

[L-6] Missing check for enough asset to asset configs on pool creation

SEVERITY: Low	IMPACT: Low
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/extension/default_extension_po_v2.cairo#L613-L650

Description:

For each pair of liquidation parameters, debt caps, and maximum shutdown LTV, a corresponding entry must be created.

```
// set the liquidation config for each pair
let mut liquidation_params = liquidation_params;
while !liquidation_params.is_empty() {
    let params = *liquidation_params.pop_front().unwrap();
    let collateral_asset =
    *asset_params.at(params.collateral_asset_index).asset;
    let debt_asset = *asset_params.at(params.debt_asset_index).asset;
    self
        .position_hooks
        .set_liquidation_config(
            pool_id,
            collateral_asset,
            debt_asset,
            LiquidationConfig { liquidation_factor:
    params.liquidation_factor },
        );
}

// set the debt caps for each pair
let mut debt_caps = debt_caps;
while !debt_caps.is_empty() {
    let params = *debt_caps.pop_front().unwrap();
    let collateral_asset =
    *asset_params.at(params.collateral_asset_index).asset;
    let debt_asset = *asset_params.at(params.debt_asset_index).asset;
    self.position_hooks.set_debt_cap(pool_id, collateral_asset, debt_asset,
    params.debt_cap);
}

// set the max shutdown LTVs for each pair
let mut shutdown_ltv_params = shutdown_params.ltv_params;
while !shutdown_ltv_params.is_empty() {
    let params = *shutdown_ltv_params.pop_front().unwrap();
    let collateral_asset =
    *asset_params.at(params.collateral_asset_index).asset;
    let debt_asset = *asset_params.at(params.debt_asset_index).asset;
    self
        .position_hooks
        .set_shutdown_ltv_config(
            pool_id, collateral_asset, debt_asset, LTVConfig { max_ltv:
    params.max_ltv },
        );
}
```

However, the current implementation does not check that there is one provided for each pair, allowing some to be left empty.

Recommendations:

We recommend that you make sure that enough parameters are provided to cover every asset added pair generated from the assets in the pool.

Vesu: Acknowledged

[L-7] The `target_utilization` is not bounded by `min_target_utilization` and `max_target_utilization`

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- src/extension/components/interest_rate_model.cairo#L83-L105

Description:

The `assert_interest_rate_config` function in `src/extension/components/interest_rate_model.cairo` contains a validation gap that allows the `target_utilization` to be set outside the valid range defined by `min_target_utilization` and `max_target_utilization`.

Recommendations:

It is recommended to add the following assertions:

```
assert!(min_target_utilization <= target_utilization,
    "target-utilization-lt-min-target-utilization");
assert!(target_utilization <= max_target_utilization,
    "target-utilization-gt-max-target-utilization");
```

Vesu: Resolved with PR-5 and @7b2f3c18b79....

Zenith: Verified.

[L-8] `set_shutdown_mode` emits wrong mode

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- src/extension/components/position_hooks.cairo#L396

Description:

The `set_shutdown_mode` instruction is used to set a new shutdown mode on a pool. At the end, it emits an event.

```
self.emit(SetShutdownMode { pool_id, shutdown_mode, last_updated:
    shutdown_state.last_updated });
```

However, this emits the prior shutdown mode, not the new one.

Recommendations:

We recommend emitting the newly set shutdown mode:

```
self.emit(SetShutdownMode { pool_id, new_shutdown_mode, last_updated:
    shutdown_state.last_updated });
```

Vesu: Resolved with PR-6

Zenith: Verified.

[L-9] Insufficient support of fee-on-transfer tokens leading to accounting discrepancies

`SEVERITY`: Low	`IMPACT`: Low
`STATUS`: Acknowledged	`LIKELIHOOD`: Low

Target

- src/singleton_v2.cairo#L692-L719
- src/singleton_v2.cairo#L1827-L1843
- src/v_token.cairo#L344-L374
- src/v_token.cairo#L403-L445
- src/v_token_v2.cairo#L366-L396
- src/v_token_v2.cairo#L425-L467
- src/extension/default_extension_po_v2.cairo#L319-L344

Description:

The Vesu protocol is vulnerable to fee-on-transfer tokens due to transfer implementations that assume the transferred amount equals the received amount. This creates severe accounting discrepancies between the protocol's internal state and actual token balances, potentially leading to protocol and user fund loss. Note that pool creation is permissionless, assets do not require explicit whitelisting and the protocol does not actively reject fee-on-transfer tokens.

Affected functions:

1. `settle_position` in `singleton_v2.cairo`: When users deposit collateral or repay debt, the protocol credits their position with the full `collateral_delta.abs()/debt_delta.abs()` amount but actually receives less due to transfer fees. As a result, the `reserve` in `AssetConfig` becomes inflated, showing more than the protocol actually holds

2. `donate_to_reserve` in `singleton_v2.cairo`: The protocol increases the recorded reserve by the full `amount` before the transfer, but receives less actual tokens. As a result, the reserve accounting becomes corrupted, leading to potential withdrawal failures.

3. vToken `deposit` functions in `v_token.cairo` and `v_token_v2.cairo`: Users receive vToken shares based on the full `assets` amount they specified, but the protocol receives

fewer actual tokens. As a result, share calculations become inflated, allowing early depositors to drain value from later depositors.

4. vToken `mint` functions in `v_token.cairo` and `v_token_v2.cairo`: The refund calculation assumes the protocol received the full `assets_estimate`, but it actually received less due to transfer fees. As a result, users and protocol receive incorrect refunds, and the protocol's balance tracking becomes corrupted.

5. `burn_inflation_fee` in `default_extension_po_v2.cairo`: The protocol transfers the full `INFLATION_FEE` amount but receives less than expected.

Recommendations:

It is recommended to explicitly reject fee-on-transfer tokens by using balance-validated transfers:

```
fn transfer_asset_validated(
    asset: ContractAddress,
    sender: ContractAddress,
    to: ContractAddress,
    amount: u256,
    is_legacy: bool
) {
    let balance_before = IERC20Dispatcher{contract_address:
    asset}.balance_of(to);

    let erc20 = IERC20Dispatcher { contract_address: asset };
    if sender == get_contract_address() {
        assert!(erc20.transfer(to, amount), "transfer-failed");
    } else if is_legacy {
        assert!(erc20.transferFrom(sender, to, amount),
    "transferFrom-failed");
    } else {
        assert!(erc20.transfer_from(sender, to, amount),
    "transfer-from-failed");
    }

    let balance_after = IERC20Dispatcher{contract_address:
    asset}.balance_of(to);
    let actual_received = balance_after - balance_before;

    // Reject fee-on-transfer tokens
    assert!(actual_received == amount, "fee-on-transfer-not-supported");
}
```

Vesu: Acknowledged. Fee-on-transfer tokens are not supported

4.4 Informational

A total of 10 informational findings were identified.

[I-1] `flash_loan` NatSpec omits `is_legacy: bool` argument.

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/singleton_v2.cairo#L1804

Description:

The `flash_loan()` function's NatSpec comments are missing the `is_legacy` bool

```
/// Executes a flash loan
/// # Arguments
/// * `receiver` - address of the flash loan receiver
/// * `asset` - address of the asset
/// * `amount` - amount of the asset to loan
/// * `data` - data to pass to the flash loan receiver
fn flash_loan(
    ref self: ContractState,
    receiver: ContractAddress,
```

Recommendations:

We recommend fixing the NatSpec as follows:

```
/// # Arguments
/// * `receiver` - address of the flash loan receiver
/// * `asset` - address of the asset
/// * `amount` - amount of the asset to loan
/// * `is_legacy` - whether the asset uses legacy ERC20 (camelCase) interface
/// * `data` - data to pass to the flash loan receiver
```

Vesu: Resolved with PR-6

Zenith: Verified

[I-2] `is_legacy` in flashloan can be passed incorrectly leading to revert

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/singleton_v2.cairo#L1805

Description:

In the `flash_loan` function, the caller can freely pass the `is_legacy`.

```
/// Executes a flash loan
/// # Arguments
/// * `receiver` - address of the flash loan receiver
/// * `asset` - address of the asset
/// * `amount` - amount of the asset to loan
/// * `data` - data to pass to the flash loan receiver
fn flash_loan(
    ref self: ContractState,
    receiver: ContractAddress,
    asset: ContractAddress,
    amount: u256,
    is_legacy: bool,
    data: Span<felt252>,
) {
    transfer_asset(asset, get_contract_address(), receiver, amount,
    is_legacy);
    IFlashLoanReceiverDispatcher { contract_address: receiver }
        .on_flash_loan(get_caller_address(), asset, amount, data);
    transfer_asset(asset, receiver, get_contract_address(), amount,
    is_legacy);

    self.emit(Flashloan { sender: get_caller_address(), receiver, asset,
    amount });
}
```

However, if the wrong flag is accidentally passed, the flash loan will revert.

Recommendations:

We recommend retrieving the `is_legacy` parameter from the stored asset config of the asset.

Vesu: Acknowledged. If the flash loan reverts the transaction will revert. This shouldn't cause any problems.

[I-3] Incorrect description of `assert_ownership()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/singleton_v2.cairo#L554

Description:

The `assert_ownership` function is described as follows.

```
/// Asserts that the delegatee has the delegate of the delegator for a
    specific pool
fn assert_ownership(
    ref self: ContractState, pool_id: felt252, extension: ContractAddress,
    delegator: ContractAddress,
) {
    let has_delegation = self.delegations.read((pool_id, delegator,
    get_caller_address()));
    assert!(
        delegator == get_caller_address()
    || extension == get_caller_address() || has_delegation,
        "no-delegation",
    );
}
```

However actually the function will also pass if the delegator himself or the extension is calling.

Recommendations:

We recommend adapting the comment to:

```
/// Asserts one of the following 3:
/// 1. The delegatee has the delegate of the delegator for a specific pool
/// 2. The delegator himself is calling
/// 3. The extension is calling
```

```
fn assert_ownership(
    ref self: ContractState, pool_id: felt252, extension: ContractAddress,
    delegator: ContractAddress,
) {
    let has_delegation = self.delegations.read((pool_id, delegator,
    get_caller_address()));
    assert!(
        delegator == get_caller_address()
    || extension == get_caller_address() || has_delegation,
        "no-delegation",
    );
}
```

Vesu: Resolved with PR-9.

Zenith: Verified.

[I-4] Incorrect NatSpec for `rate_accumulator()`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/singleton_v2.cairo#L477-L484

Description:

The `rate_accumulator()` function includes the `asset_config` param however it is not documented in the functions NatSpec.

Recommendations:

We recommend adding the `asset_config` param to the NatSpec.

Vesu: Resolved with PR-6

Zenith: Verified.

[I-5] Division by zero and undefined mathematical operations in `pow_scale` and `pow` functions

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/math.cairo#L4-L57

Description:

The `pow_scale` function in `src/math.cairo` has a division by zero vulnerability when `x == 0` and `is_negative == true`. In this case, the function attempts to compute `SCALE * SCALE / 0`, which will panic.

Additionally, both `pow` and `pow_scale` handle the edge case `x == 0` and `n == 0` (`0^0`) incorrectly from a mathematical perspective:

- In `pow`: When `x == 0` and `n == 0`, the function returns `0`, but mathematically `0^0` is undefined.
- In `pow_scale`: When `x == 0`, `n == 0` and `is_negative == false`, the function returns `SCALE`, but this is also mathematically incorrect for the same reason.

Recommendations:

It is recommended to add explicit checks in `pow_scale` to prevent division by zero. Furthermore, consider whether `0^0` should be treated as an error or handled with a specific business logic decision.

Vesu: Acknowledged and behavior changed in @f1089675d9....

[I-6] Missing `round_up` in Natspec for multiple common functions

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- src/common.cairo#L17
- src/common.cairo#L40
- src/common.cairo#L62
- src/common.cairo#L95

Description:

Multiple functions inside the `commonf` file are missing the `round_up` parameter in their NatSpec.

Recommendations:

We recommend adding the parameter.

Vesu: Resolved with PR-6.

Zenith: Verified.

[I-7] Slightly underestimated utilization in `calculate_utilization` function due to rounding

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Acknowledged	`LIKELIHOOD:` Low

Target

- src/common.cairo#L120-L133

Description:

The `calculate_utilization` function in `common.cairo` slightly underestimates pool utilization by using `Rounding::Floor` when calculating the utilization ratio. This creates a cascading effect throughout the protocol's core financial mechanisms that depend on accurate utilization metrics.

Recommendations:

It is recommended to change the rounding direction to `Rounding::Ceil`.

Vesu: Acknowledged.

[I-8] Inconsistent rounding in `calculate_fee_shares` undermines intended conservative fee calculation approach

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- src/common.cairo#L153-L185

Description:

The `calculate_fee_shares` function in `common.cairo` attempts to round down the final fee shares calculation to be conservative, but this intent is undermined by incorrect rounding directions in the denominator calculations. The function uses `Rounding::Floor` for the final `u256_mul_div` operation, but the components that make up the denominator are also rounded down, which actually increases the final fee shares amount rather than decreasing it.

1. Intended: Final calculation uses `Rounding::Floor`:

```
u256_mul_div(fee, total_collateral_shares, total_assets +
    (accrued_interest - fee), Rounding::Floor)
```

2. Inconsistent `accrued_interest` calculation uses `round_up = false`:

```
let accrued_interest = calculate_debt(
    asset_config.total_nominal_debt, rate_accumulator_delta,
    asset_config.scale, false,
);
```

3. Inconsistent : `total_assets` calculation uses `round_up = false`:

```
let total_assets = reserve + calculate_debt(total_nominal_debt,
    last_rate_accumulator, scale, false);
```

Recommendations:

It is recommended to set `round_up = true` when calculating the `accrued_interest` and `total_assets` via the `calculate_debt` function.

Vesu: Resolved with PR-3.

Zenith: Verified.

[I-9] Migrator can only be set by migrator leading to potential blockage

SEVERITY: Informational	IMPACT: Informational
STATUS: Acknowledged	LIKELIHOOD: Low

Target

- src/singleton_v2.cairo#L2012

Description:

The `set_migrator` function can be used to set a new `migrator`.

```
/// Sets the migrator address
/// # Arguments
/// * `migrator` - the new migrator address
fn set_migrator(ref self: ContractState, migrator: ContractAddress) {
    assert!(self.migrator.read() == get_caller_address(),
    "caller-not-migrator");
    self.migrator.write(migrator);
}
```

However, the address can currently only be reset by the current migrator. This could lead to issues if access to the key is lost.

Recommendations:

We recommend allowing the owner to set a new migrator in case of key compromise or loss of keys.

Vesu: Acknowledged.

[I-10] `retrieve_from_reserve` can underflow without error

`SEVERITY:` Informational	`IMPACT:` Informational
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- src/singleton_v2.cairo#L1862

Description:

The `retrieve_from_reserve` function can be used to transfer assets out of the reserve to a `receiver` provided by the extension.

```
fn retrieve_from_reserve(
    ref self: ContractState, pool_id: felt252, asset: ContractAddress,
    receiver: ContractAddress, amount: u256,
) {
    let extension = self.extensions.read(pool_id);
    assert!(extension == get_caller_address(), "caller-not-extension");
    let (mut asset_config, fee_shares) = self.asset_config(pool_id, asset);
    assert_asset_config_exists(asset_config);
    // attribute the accrued fee shares to the pool's extension
    self.attribute_fee_shares(pool_id, extension, asset, fee_shares);
    // retrieve amount from the reserve
    asset_config.reserve -= amount;
    self.asset_configs.write((pool_id, asset), asset_config);
    transfer_asset(asset, get_contract_address(), receiver, amount,
    asset_config.is_legacy);

    self.emit(RetrieveReserve { pool_id, asset, receiver });
        }
```

However, the function never checks if the requested amount exceeds the actual amount held in the reserve. Thus, this might lead to an underflow, which will result in an error without any explanation.

Recommendations:

We recommend adding a check that ensures that `asset_config.reserve >= amount`.

```
assert!(asset_config.reserve ≥ amount, "reserve-insufficient")
```

Vesu: Resolved with PR-10.

Zenith: Verified.