Veridise

# Auditing Report

## Hardening Blockchain Security with Formal Methods

FOR

Volo Vault

Veridise Inc.

July 14, 2025

► **Prepared For:**

Navi Protocol
https://www.volosui.com/

► **Prepared By:**

Aayushman Thapa Magar
Ajinkya Rajput
Evgeniy Shishkin

► **Contact Us:**

contact@veridise.com

► **Version History:**

| | |
|---|---|
| July 14, 2025 | V3 |
| July 9, 2025 | V2 |
| July 1, 2025 | V1 |
| June 27, 2025 | Initial Draft |

# Contents



 

 

# 1 Executive Summary

From May 28, 2025 to Jun 20, 2025, Navi Protocol engaged Veridise to conduct a security assessment of the Volo Vault protocol. Veridise conducted the assessment over 9 person-weeks, with 3 security analysts reviewing the project over 3 weeks on commit `48b4fed`. The review strategy involved thorough manual code review of the program source code.

**Project Summary.** The **Volo Vault** is a yield-generating system implemented in the form of a vault. Users deposit their coins into the Vault and receive Vault shares in return. These shares can later be redeemed for their corresponding value — ideally higher than the initial purchase price — at a future time.

Yield within the Volo Vault protocol is expected to be generated by issuing flash loans from the deposited funds to a specifically designated set of users, known as *Operators*. These Operators utilize the borrowed assets to execute custom trading strategies across various DeFi protocols, such as *Navi*, *SuiLend*, *Momentum*, and others, thereby producing profits for the protocol.

The Volo Vault protocol operates on the **SUI Network blockchain** and follows a semi-centralized design. Part of the protocol is implemented through on-chain smart contract logic, while other components — notably the Operators' trading functionalities — are executed off-chain.

The typical operational workflow of the protocol is structured as follows:

1. **Deposits.** Users submit deposit requests to the Vault. In response, the Vault creates a *Receipt* object that uniquely identifies the pending deposit. The request is not fulfilled immediately; instead, it is added to a queue of pending deposit requests awaiting approval by an Operator.
2. **Deposit Approval.** At a chosen point, an Operator decides to process a pending deposit request. They explicitly execute this operation on-chain, specifying the unique identifier of the deposit to be fulfilled. Upon completion, the user receives the corresponding amount of Vault shares, while the deposited coins are allocated to a special *free principal balance*, making them available for flash loans.
3. **Yield Generation.** To generate yield, Operators implement custom trading strategies that temporarily borrow accumulated Vault funds via flash loans to interact with other DeFi protocols on SUI. These loans are executed within a single transaction using the *Programmable Transaction Blocks* (PTB) of SUI. The protocol enforces a rule ensuring that the amount of assets returned is no less than the amount borrowed*. Ideally, Operators return a greater amount than borrowed, thereby increasing the value of each Vault share.
4. **Withdrawals.** To redeem shares, users submit a withdrawal request, supplying the *Receipt* created during the deposit. Similar to deposits, withdrawal requests are not executed immediately but are placed into a queue of pending withdrawal requests overseen by Operators.

* An epoch loss mechanism allows for minor losses to accumulate if necessary.

5. **Withdrawal Approval.** At a selected time, an Operator decides to process a pending withdrawal request. They execute this operation on-chain, specifying the unique identifier of the withdrawal to be fulfilled. Upon completion, the user receives their coins — ideally, an amount exceeding the initial deposit, reflecting the yield generated over time.

**Note on Centralization.** The protocol includes a centralized component: users cannot withdraw their originally deposited assets unless an Operator explicitly approves their withdrawal requests.

**Code Assessment.** The Volo Vault developers provided the source code of the Volo Vault contracts for the code review. The source code appears to be mostly original code written by the Volo Vault developers. It contains some documentation in the form of comments on functions and storage variables. To facilitate the Veridise security analysts' understanding of the code, the developers provided a comprehensive overview of the project, detailing its core functions, system states, intended interaction patterns, and the range of supported operations.

The source code contained a test suite, which the Veridise security analysts noted covered many of the happy paths, but had comparatively fewer tests to check for potential errors or flaws in the code.

**Summary of Issues Detected.** The security assessment uncovered 13 issues, 6 of which are assessed to be of high or critical severity. Specifically, the review uncovered an issue V-VLT-VUL-001, which allows any arbitrary user to steal funds from other users. Additionally, the issue V-VLT-VUL-003 enable users to withdraw their funds twice, potentially resulting in losses for the vault. Furthermore, issues such as V-VLT-VUL-002 and V-VLT-VUL-004 reveal flaws in the price calculation, leading to an inaccurate accounting of asset values, rewards, and related metrics. The Veridise analysts also identified 3 medium-severity issues —including V-VLT-VUL-007, where the protocol can enter a permanent denial-of-service state if an improper value is provided for the reward rate parameter, and V-VLT-VUL-009 where user funds could have become forever stuck if a receipt from another Vault was mistakenly provided. Additionally, there were 1 low-severity and 3 warnings issues discovered. Issues V-VLT-VUL-013 and V-VLT-VUL-010 were identified and reported by protocol developers.

Following discussions with the developers, three issues were identified as reflecting intended behavior. Specifically, for V-VLT-APP-VUL-001 and V-VLT-APP-VUL-002, the flagged behaviors were deemed irrelevant under the current protocol configuration, which assumes the presence of only trusted Operators. In the case of V-VLT-APP-VUL-003, the developers clarified that monitoring the health factor of potentially problematic positions will be handled by the back-end component.

It is also important to note that during the fix review, Veridise analysts observed that additional logic had been introduced alongside the fixes. For instance, new functions such as `batch_execute_deposit()` and `batch_execute_withdraw()` were added, as well as a fourth vault receipt status, `PENDING_WITHDRAW_WITH_AUTO_TRANSFER_STATUS`. These changes were not part of the original review scope and were likewise excluded from the scope of the fix review.

**Recommendations.** After conducting the assessment of the protocol, the security analysts had a few suggestions to improve the Volo Vault.

*More Restrictive Check Placement.* The current code structure places sanity checks within the entry-point functions, which subsequently delegate to internal functions. We recommend relocating as many of these sanity checks as possible into the internal functions themselves. This adjustment would help ensure that critical validations are consistently enforced, regardless of how or from where the internal functions are invoked. It also reduces the risk of inadvertently bypassing essential checks when introducing new entry points in future protocol upgrades.

*Improve testing.* The protocol includes a comprehensive test suite that partially covers both successful operations and anticipated failure scenarios. However, it is recommended to further extend the test coverage to encompass more complex situations — for example, scenarios where Operators execute multiple flash loans within a single transaction — to better simulate real-world interactions and edge cases.

Additionally, it is advisable to test the Vault's functionality with a variety of coins, including both principal coins and flash-loan assets with differing decimal configurations, in order to thoroughly validate the protocol's decimal handling logic.

*Additional review.* After thorough testing with trusted, controlled Operators, if it is later decided to assign Operator caps to less trusted users, it is recommended to conduct an additional security review.

*Borrowing function checks.* The Vault allows asset borrowing through an internal function that performs minimal validation before transferring assets. These validations are currently expected to be enforced by the caller of the function. It is recommended to tighten the conditions under which borrowing is permitted to reduce risk.

*Introducing new DeFi assets.* The Vault supports DeFi assets by acquiring their corresponding ownership capability objects. The actual value of these assets is determined by the positions governed through these capabilities. However, if a third-party protocol were integrated that permits the creation of child capabilities derived from a primary ownership capability, this could introduce a significant security risk. In such a case, an Operator could exploit the mechanism by swapping the Vault's assets for others of equivalent value during a flash-loan operation. After the loan is repaid, the Operator could retain an undeclared child capability, allowing them to liquidate the original positions at a later time without the Vault's awareness.

To mitigate this risk, it is recommended to explicitly avoid supporting third-party protocols that enable the creation of secondary or child capabilities for asset control. Additionally, it is recommended to actively monitor updates to any currently integrated protocols to detect the introduction of such features. If such functionality is identified, it is recommended to promptly disable support for the affected asset type and convert any existing holdings into secure, unaffected assets.

# 2 Project Dashboard

**Table 2.1:** Application Summary.

| Name | Version | Type | Platform |
|---|---|---|---|
| Volo Vault | `48b4fed` | Move | SUI |

**Table 2.2:** Engagement Summary.

| Dates | Method | Consultants Engaged | Level of Effort |
|---|---|---|---|
| May 28–Jun 20, 2025 | Manual & Tools | 3 | 9 person-weeks |

**Table 2.3:** Vulnerability Summary.

| Name | Number | Acknowledged | Fixed |
|---|---|---|---|
| Critical-Severity Issues | 3 | 3 | 3 |
| High-Severity Issues | 3 | 3 | 3 |
| Medium-Severity Issues | 3 | 3 | 3 |
| Low-Severity Issues | 1 | 1 | 1 |
| Warning-Severity Issues | 3 | 3 | 3 |
| Informational-Severity Issues | 0 | 0 | 0 |
| TOTAL | 13 | 13 | 13 |

**Table 2.4:** Category Breakdown.

| Name | Number |
|---|---|
| Logic Error | 9 |
| Data Validation | 2 |
| Access Control | 1 |
| Maintainability | 1 |

# 3 Security Assessment Goals and Scope

## 3.1 Security Assessment Methodology

The Security Assessment process consists of the following steps:

1. Gather an initial understanding of the protocol's business logic, users, and workflows by reviewing the provided documentation and consulting with the developers.
2. Identify all valuable assets in the protocol.
3. Identify the main workflows for managing these assets.
4. Identify the most significant security risks associated with these assets.
5. Systematically review the code base for execution paths that could trigger the identified security risks, considering different assumptions.
6. Prioritize one finding over another by assigning a severity level to each.

During the security assessment, the Veridise security analysts regularly met with the Volo Vault developers to ask questions about the code, discuss progress, and communicate identified issues. The Veridise security analysts also perused the shared documentation for the Navi Vault V2 - Strategy Vault.

## 3.2 Identified Security Risks

After the initial phase of the security assessment was completed, a list of potential security risks was generated. Security analysts used this list during the code review as a starting point to identify potential attack vectors. A few of these risks, when expressed as questions, include the following:

- Is there anything that could prevent legitimate users from depositing their assets and receiving the correct number of shares once the request is executed?
- Is there anything that could prevent users from withdrawing their assets and receiving the correct amount of coins once the request is executed?
- Is it possible to artificially inflate the share price without holding any privileged role?
- Is it possible for a malicious user to withdraw funds belonging to other users?
- Are there any ways to put the protocol into a permanent denial-of-service state?
- Is there a way for an Operator to withdraw funds from the Vault?
- Is there a way for an Operator to return significantly fewer funds than they borrowed during a flash-loan operation?
- Are there any DeFi-related risks associated with the current protocol integrations?
- Is it possible for a malicious user to claim rewards associated with Receipts belonging to other users?
- Are there any common protocol implementation defects, such as insufficient access control or improper handling of token decimals?
- Are there any opportunities for slippage exploitation within the protocol?

## 3.3 Scope

The scope of this security assessment is limited to the `/sources` folder of the source code provided by the Volo Vault developers, which contains the smart contract implementation of the Volo Vault:

- `sources/1.move`
- `sources/manage.move`
- `sources/operation.move`
- `sources/oracle.move`
- `sources/receipt.move`
- `sources/reward_manager.move`
- `sources/user_entry.move`
- `sources/utils.move`
- `sources/vault_receipt_info.move`
- `sources/volo_vault.move`
- `sources/adaptors/cetus_adaptor.move`
- `sources/adaptors/momentum_adaptor.move`
- `sources/adaptors/navi_adaptor.move`
- `sources/adaptors/receipt_adaptor.move`
- `sources/adaptors/suilend_adaptor.move`
- `sources/requests/deposit_request.move`
- `sources/requests/withdraw_request.move`

# 4 Trust Model

## 4.1 Operational Assumptions

The Volo Vault integrates with several DeFi protocols, enabling Operators to earn profits for the protocol by temporarily borrowing Vault assets and utilizing them within these external platforms. While this integration adds significant functional complexity, it is important to note that not all of these DeFi protocols are thoroughly documented. Furthermore, the internal logic and security of these third-party protocols were explicitly out of scope for this security assessment. As a result, the assessment team had to rely on informed assumptions based on the behavior of similar protocols. The security posture of these integrated DeFi protocols remains uncertain.

Additionally, the Volo Vault relies on asset price data provided by Switchboard oracle price aggregators. The configuration, correctness, and security of these oracles were not within the scope of this review and were therefore not independently verified.

For the protocol to function as intended, Volo Vault depends on Operators to execute user requests and generate yield through flash-loan operations and other strategies. This operational logic resides entirely off-chain and was not reviewed as part of this assessment, nor was the security or trustworthiness of the Operators' execution environments evaluated.

This assessment was conducted under the following assumptions:

- Integrated DeFi protocols — specifically *Cetus*, *Momentum*, *Navi*, and *SuiLend* — behave correctly and do not exhibit unexpected or non-standard behavior.
- Switchboard oracle aggregators are correctly configured, provide accurate price data, and operate securely.
- Admin users and Operator users are expected to act in accordance with the protocol's intended design and behave without malicious intent. However, in the case of Operator users, the system should ensure that even if their keys are compromised, they cannot steal funds from the Vault.
- The off-chain Operator logic is free from critical programmatic defects and is executed within a secure, trusted environment.

## 4.2 Privileged Users

During the review, Veridise analysts assumed that the privileged users perform their responsibilities as intended. Protocol exploits relying on the below users acting outside of their privileged scope or intentionally abstaining from doing their duties are considered to be outside of scope of this security review.

- **Admin**. This user can enable, disable, and upgrade the Vault, set deposit and withdraw fees, set loss tolerance, assign Operator roles to user and freeze them; add, remove or

change switchboard aggregators, as well as update various other contract states. There is a single Admin for the Vault.

- **Operator**. This type of users can perform privileged actions such as initiating flash borrows, executing deposits and withdrawals, adding new asset types, and managing reward-related functionalities.

**Operational Recommendations.** Highly-privileged operations should be operated by a multi-sig contract or decentralized governance system. These operations should be guarded by a time-lock to ensure there is enough time for incident response. Highly-privileged operations should be tested in example scenarios to ensure the role operators are available and ready to respond when necessary.

Full validation of operational security practices is beyond the scope of this review. Users of the protocol should ensure they are confident that the operators of privileged keys are following best practices such as:

- Never storing a protocol key in plain text, on a regularly used phone, laptop, or device, or relying on a custom solution for key management.
- Using separate keys for each separate function.
- Storing multi-sig keys in a diverse set of key management software/hardware services and geographic locations.
- Enabling 2FA for key management accounts. SMS should *not* be used for 2FA, nor should any account which uses SMS for 2FA. Authentication apps or hardware are preferred.
- Validating that no party has control over multiple multi-sig keys.
- Performing regularly scheduled key rotations for high-frequency operations.
- Securely storing physical, non-digital backups for critical keys.
- Actively monitoring for unexpected invocation of critical operations and/or deployed attack contracts.
- Regularly drilling responses to situations requiring emergency response such as pausing/unpausing.

# 5 Vulnerability Report

## 5.1 Classification of Vulnerabilities

When Veridise security analysts discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise.

The severity of a vulnerability is evaluated according to the Table 5.1.

**Table 5.1:** Severity Breakdown.

| | Somewhat Bad | Bad | Very Bad | Protocol Breaking |
|---|---|---|---|---|
| Not Likely | Info | Warning | Low | Medium |
| Likely | Warning | Low | Medium | High |
| Very Likely | Low | Medium | High | Critical |

The likelihood of a vulnerability is evaluated according to the Table 5.2.

**Table 5.2:** Likelihood Breakdown

| | |
|---|---|
| Not Likely | A small set of users must make a specific mistake |
| Likely | Requires a complex series of steps by almost any user(s)<br>- OR -<br>Requires a small set of users to perform an action |
| Very Likely | Can be easily performed by almost anyone |

The impact of a vulnerability is evaluated according to the Table 5.3:

**Table 5.3:** Impact Breakdown

| | |
|---|---|
| Somewhat Bad | Inconveniences a small number of users and can be fixed by the user |
| Bad | Affects a large number of people and can be fixed by the user<br>- OR -<br>Affects a very small number of people and requires aid to fix |
| Very Bad | Affects a large number of people and requires aid to fix<br>- OR -<br>Disrupts the intended behavior of the protocol for a small group of users through no fault of their own |
| Protocol Breaking | Disrupts the intended behavior of the protocol for a large group of users through no fault of their own |

## 5.2 Summary of Vulnerabilities

This section presents the vulnerabilities found during the security assessment. For each issue found, the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.) is specified. Table 5.4 summarizes the issues discovered:

**Table 5.4:** Summary of Discovered Vulnerabilities.

| ID | Description | Severity | Status |
|---|---|---|---|
| V-VLT-VUL-001 | Claimable principal can be claimed by anyone | Critical | Fixed |
| V-VLT-VUL-002 | USD value of the receipt is incorrectly . . . | Critical | Fixed |
| V-VLT-VUL-003 | Double withdrawal is possible for auto- . . . | Critical | Fixed |
| V-VLT-VUL-004 | Incorrect normalization for decimals . . . | High | Fixed |
| V-VLT-VUL-005 | Receipt assets with pending withdraw . . . | High | Fixed |
| V-VLT-VUL-006 | Incorrect reward index updates allow . . . | High | Fixed |
| V-VLT-VUL-007 | Setting huge reward rate can lead to . . . | Medium | Fixed |
| V-VLT-VUL-008 | Stale asset prices can lead to value loss | Medium | Fixed |
| V-VLT-VUL-009 | Providing receipts from different vault can . . . | Medium | Fixed |
| V-VLT-VUL-010 | Reward index miscalculation due to loss . . . | Low | Fixed |
| V-VLT-VUL-011 | Maintainability Issues | Warning | Fixed |
| V-VLT-VUL-012 | Admins may lock in user funds in pending . . . | Warning | Fixed |
| V-VLT-VUL-013 | Potential loss of precision in rewards . . . | Warning | Fixed |

## 5.3 Detailed Description of Issues

### 5.3.1 V-VLT-VUL-001: Claimable principal can be claimed by anyone

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 48b4fed |
| **Type** | Access Control | **Status** | Fixed |
| **File(s)** | `sources/volo_vault.move` | | |
| **Location(s)** | sources/volo_vault.move:1396-1416 | | |
| **Confirmed Fix At** | 7f0b08f | | |

**Description** The `vault.claim_claimable_principal()` function allows users to claim principal after a withdrawal request has been fulfilled. However, because the function has `public` visibility, any caller can claim the principal by providing a `receipt_id`, even if they are not the rightful owner of the receipt.

```
1 public fun claim_claimable_principal<T>(self: &mut Vault<T>,receipt_id: address,
      amount: u64,)
```

**Impact** This vulnerability allows malicious users to steal funds belonging to other users by claiming their claimable principal.

**Recommendation** It is recommended to change the function to `public(package)` and ensure only authorized users are able to claim their claimable principal.

**Developers Response** The developers fixed the issue at commit 7f0b08fe146d47e4afb0f8e081bf484c4e109c75.

### 5.3.2 V-VLT-VUL-002: USD value of the receipt is incorrectly calculated

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `sources/adaptors/receipt_adaptor.move` | | |
| **Location(s)** | sources/adaptors/receipt_adaptor.move:31-231 | | |
| **Confirmed Fix At** | a3af9fe | | |

**Description** The vault supports receipts managed by VOLO vaults as defi assets. The USD value of the receipt is calculated by the function`receipt_adaptor::get_receipt_value()` in `sources/adaptors/receipt_adaptor.move` as sum of

- The value of the shares held by the receipt
- The pending deposit balance of the receipt and,
- The claimable principal by the receipt

as shown in the snippet below

```
let value =
        vault_utils::mul_d(shares, share_ratio) +  (vault_receipt.
    pending_deposit_balance() as u256) + (vault_receipt.claimable_principal() as u256
    );
```

The USD value calculated above is incorrect because the **number of assets** in `pending_deposit_balance` and `claimable_principal` are added to USD value of shares instead of USD value of `pending_deposit_balance` and USD value of `claimable_principal`

**Impact** A malicious operator can swap high value tokens for very low value tokens.

**Recommendation** Multiply `vault_receipt.pending_deposit_balance()` and `vault_receipt.claimable_principal()` with price of the `PrincipalCoinType` of the vault. This loss will not be reflected in the calculated loss.

**Developer Response** Developers fixed the issue at commit

a3af9fe248a2c3f06267755ee09c508590446fb4.

### 5.3.3 V-VLT-VUL-003: Double withdrawal is possible for auto-transfer withdrawal requests

| | | | |
|---|---|---|---|
| **Severity** | Critical | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `./sources/volo_vault.move` | | |
| **Location(s)** | ./sources/volo_vault.move:967-968 | | |
| **Confirmed Fix At** | f5fcf80 | | |

**Description** When a user issues a withdrawal request, it can come in two forms: either the funds get auto-transferred to the user, or they should explicitly withdraw them from the Vault once the funds become available.

The current implementation of `volo_vault.execute_withdrawal()` function increments the corresponding `vault_receipt_info.claimable_amount` value in both cases - rather than only when the user is expected to manually withdraw the funds, as it should. This gives opportunity for a malicious user to withdraw the amount twice.

**Impact** Due to the issue, users can withdraw their funds twice for fulfilled auto-transfer withdrawal requests - once through the automatic transfer and again by calling `user_entries.claim_claimable_principle()`.

**Recommendation** The `volo_vault.update_after_execute_withdraw()` function should not increase the `claimable_principle` amount of the vault receipt when the withdrawal is processed via auto-transfer.

**Developers Response** The developers fixed the issue at commit f5fcf80c4483bbc99a97bb1169d47095fc77de08.

### 5.3.4 V-VLT-VUL-004: Incorrect normalization for decimals greater than 9

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `sources/oracle.move` | | |
| **Location(s)** | sources/oracle.move:141 | | |
| **Confirmed Fix At** | f5fcf80 | | |

**Description** The function `oracle.get_normalized_asset_price()` is used to retrieve the normalized price of an asset. The normalization factor is calculated based on the number of decimals in the asset's coin type. The price returned by this function is 18 decimals but the value is the price of whatever number of assets represented by the integer 10^9. Note that the price returned by `oracle.get_normalized_asset_price()` is not the price of 1 asset.

This price is calculated as shown below

```
    let normalized_factor = if (decimals < 9) {
        pow(10, 9 - decimals) as u256
    } else {
        pow(10, decimals - 9) as u256
    };
    price * normalized_factor
```

However, this computation is incorrect when the asset for which the price is being queried has more than 9 decimals.

**Impact** The price of the asset will be inflated by a factor of `pow(10, decimals-9)` for assets with decimals more than 9.

**Recommendation** It is recommended to impliment the correct formula to normalize the asset price when the asset decimals is greater than 9:

```
price / normalized_factor
```

**Developers Response** The developers fixed the issue at commit f5fcf80c4483bbc99a97bb1169d47095fc77de08.

### 5.3.5 V-VLT-VUL-005: Receipt assets with pending withdraw balance can be used to take out funds from Vault

| Severity | High | Commit | 48b4fed |
|---|---|---|---|
| Type | Logic Error | Status | Fixed |
| File(s) | `./sources/adaptors/receipt_adaptor.move` | | |
| Location(s) | ./sources/adaptors/receipt_adaptor.move:31 | | |
| Confirmed Fix At | 8860666 | | |

**Description** The Vault allows users to supply their own Receipts as assets. Each asset type has its own method for calculating USD value. For Receipts, the USD value is determined based on the following components:

- The number of shares held by the Receipt
- The current USD price per share
- The `pending_deposit_balance` amount
- The `claimable_principal` amount

However, this valuation method does not account for the possibility that a Receipt may be in a `pending_withdraw` state. In the context of flash loans - particularly when withdrawals are executed via auto-transfer - a malicious Operator could exploit this oversight through the following steps:

1. Prepare a `Receipt1` holding shares worth **X** USD, but place it in a `pending_withdraw` state by submitting a withdrawal request.
2. Initiate a flash loan and borrow principal assets valued at **X**.
3. Repay the flash loan by returning `Receipt1` as collateral.
4. When the protocol evaluates `Receipt1`, it will still be valued at **X** because the shares remain intact - the withdrawal was requested but not yet executed.
5. Finalize the flash loan; since the total USD value appears consistent, no immediate discrepancy is detected.
6. Immediately after finalization, call `Operation.execute_withdraw()` with the request ID for `Receipt1`. This will transfer the corresponding funds out of the Vault to the Operator's address.

As a result, the Operator effectively removes **X** worth of principal assets from the Vault, creating a discrepancy and exploiting the protocol's failure to properly account for the `pending_withdraw` status in its USD valuation logic.

**Impact** A malicious Operator can withdraw funds from the Vault equal to the value of their Receipt that is in the pending withdrawal status.

**Recommendation** It is recommended to account Receipts value with pending withdrawal status in such a way that would prevent the described attack vector.

**Developers Response** Developers fixed the issue at commit 88606668e1a035c5c441ddad5325664fe911214b.

### 5.3.6 V-VLT-VUL-006: Incorrect reward index updates allow exploitable reward dilution

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `./sources/volo_vault.move` | | |
| **Location(s)** | ./sources/volo_vault.move:782-783 | | |
| **Confirmed Fix At** | 7ad9d85 | | |

**Description** The reward distribution system implemented in the Vault and Reward Manager (`RM`) is vulnerable to a dilution issue when new shares are minted after a reward balance has been added but before the reward indices of individual Receipts are updated. This creates a window of opportunity where newly created or increased Receipts can claim a disproportionate share of existing rewards - effectively extracting rewards that should rightfully belong to earlier depositors.

This issue arises because the global reward index is not adjusted retroactively for new shares minted after `RM.add_reward_balance()` is called. As a result, the protocol allows those new shares to immediately participate in reward claims based on an outdated index, diluting the rewards available to existing Receipts.

**Attack Scenario:**

Consider the following scenario involving a Vault and a Reward Manager (`RM`):

1. **Initialization:**
   - Operator calls `RM.add_new_reward_type(with_buffer=false)`
     - `RM.reward_amounts[reward_token]` = 0
     - `RM.reward_indices[reward_token]` = 0
2. **User1 Deposit:**
   - User1 creates `Receipt1` with **100 shares**
     - `Vault.total_shares()` = 100
     - `vault_receipt1.reward_indices[reward_token]` = 0
3. **User2 Initial Deposit:**
   - User2 creates `Receipt2` with **1 share**
     - `Vault.total_shares()` = 101
     - `vault_receipt2.reward_indices[reward_token]` = 0
4. **Reward Added:**
   - Operator calls `RM.add_reward_balance(100)`
     - `RM.reward_amounts[reward_token]` = 100
     - `RM.reward_indices[reward_token]` = 100 / 101 = **0.99**
5. **User2 Increases Position:**
   - User2 deposits for **99 additional shares**
     - `Receipt2.shares` = 100

6. **User2 Claims Rewards:**
   - User2 calls `claim_reward()`
     - `vault_receipt2.update_reward(reward_token, 0.99)`
     - `vault_receipt2.unclaimed_reward` += 0.99 * 100 = **99**
     - **99 reward tokens are transferred to User2**
7. **User1 Attempts to Claim:**
   - User1 calls `claim_reward()`
     - `vault_receipt1.update_reward(reward_token, 0.99)`
     - `vault_receipt1.unclaimed_reward` += 0.99 * 100 = **99**
     - Call **fails due to insufficient reward balance remaining**

**Impact** The provided scenario demonstrates how User2 can effectively claim a portion of rewards that were generated by User1's deposit before User2's additional shares existed. This results in:

- **Unfair reward distribution**, where existing users receive less than their proportional share.
- **Potential malicious exploitation**, where a user deliberately front-runs reward distribution to siphon value from earlier participants.

Depending on context and intent, this may be classified as either a protocol design flaw or an active economic exploit.

**Recommendation** To prevent this form of reward dilution it is required to enforce reward index updates immediately before any share minting operation in the Vault when a reward distribution exists.

**Developers Response** Developers fixed the issue at commit 7ad9d85bf553ce3aeefc8c053aadfe8c9f781380.

### 5.3.7 V-VLT-VUL-007: Setting huge reward rate can lead to permanent denial-of-service

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 48b4fed |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `./sources/reward_manager.move` | | |
| **Location(s)** | ./sources/reward_manager.move:586 | | |
| **Confirmed Fix At** | 04ef337 | | |

**Description** The `reward_manager.set_reward_rate()` function updates the reward rate parameter, which defines the number of reward tokens the reward manager allocates per millisecond. This function does not perform validation or sanity checks on the provided rate value.

**Impact** If the supplied rate is set to `u64::MAX`, the protocol will enter a permanent denial-of-service state. This occurs because most contract entry functions invoke `update_reward_buffers()` as their first operation. Within this function, an unconditional multiplication involving the reward rate takes place - and with a value of `u64::MAX`, this operation would cause an overflow.

Notably, even `set_reward_rate()` itself calls `update_reward_buffers()` before applying the new rate, meaning recovery from this state would be impossible once the invalid rate is set.

Configuring such an unreasonably high reward rate has no legitimate use case and would likely be an intentional act by a malicious Operator user aiming to disrupt the protocol.

**Recommendation** The sanity check has to be implemented for this function.

**Developers Response** The developers fixed the issue at commit 04ef337bf1c3528d33f0910b65bb9cc45858a039.

### 5.3.8 V-VLT-VUL-008: Stale asset prices can lead to value loss

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 48b4fed |
| **Type** | Data Validation | **Status** | Fixed |
| **File(s)** | `./sources/operation.move` | | |
| **Location(s)** | ./sources/operation.move:175-176 | | |
| **Confirmed Fix At** | 0834ec1 | | |

**Description** The function `vault.get_total_usd_value()` is responsible for recording the total USD value of the Vault at the start of a flash loan (see reference). This value is calculated based on asset valuations previously determined using oracle price data (see reference). However, due to the time gap between the current transaction timestamp (`now()`) and the last asset price update, asset valuations within the Vault can become stale. This discrepancy introduces a risk of **assets value miscalculation** during flash loan operations, as the Vault may rely on outdated prices when recording or verifying its state.

**Problematic Scenario:**

Consider a Vault holding a volatile asset such as BTC:

1. The BTC price stored in the Vault is stale and 25% lower than the current asset price.
2. The Operator initiates a flash loan, borrowing 10 BTC. The Vault records its total USD value as `10 BTC * stale_price` in a `TxBagForCheckValueUpdate` object.
3. The Operator returns only 8 BTC to the Vault.
4. Before closing the flash loan, the Operator calls `Vault.update_free_principal_value()`, which updates the BTC price to the latest (higher) asset value.
5. As a result, **8 BTC * updated BTC price = original total USD value** remembered at the start of the flash loan.
6. The protocol's check comparing the updated total USD value to the remembered value passes successfully, allowing the flash loan to finalize without error.

**Notably, this discrepancy does not affect** `epoch_loss` **calculations**, as the total USD value appears consistent with the value recorded at flash-loan initiation, despite the Vault effectively losing **2 BTC** in principal.

**Impact** Even if Operators are considered trusted, this behavior can still lead to silent value extraction, especially if automated systems (e.g., trading bots) are only incentivized to meet protocol validation thresholds, not preserve absolute principal. Stale price data may also adversely impact `execute_withdraw()` and `execute_deposit()` operations:

In the case of a deposit, the number of shares to be minted is calculated based on the share price, which itself depends on `total_usd_value()`. If this value is outdated, it may result in sub-optimal amounts of minted shares. Although slippage protection mechanisms exist - where the user specifies a minimum acceptable number of shares and the Operator defines a maximum number to mint - the actual amount granted within this range may still be lower than it would have been if asset valuations were current.

A similar risk applies during withdrawals. In this case, the amount of funds returned to the user for their shares is determined based on the same potentially stale asset values, which could lead to users receiving less than the fair value of their holdings.

**Recommendation** Enforce a fresh price update immediately before initiating a flash loan or before recording `total_usd_value`, before executing `execute_withdraw()` and `execute_deposit()`. If more frequent updates are infeasible, clearly document and quantify the risk.

**Developers Response** The issue was addressed in commit 0834ec1ba6e3494bf1236725cd24a75d13c2dc41: the value of the `MAX_UPDATE_INTERVAL` parameter was reduced to `0`, effectively disabling the interval. However, the developers have retained the ability to modify this value in the future, with the understanding that any changes must be made with full consideration of the associated risks.

### 5.3.9 V-VLT-VUL-009: Providing receipts from different vault can lock user's funds

| | | | |
|---|---|---|---|
| **Severity** | Medium | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `sources/user_entry.move` | | |
| **Location(s)** | sources/user_entry.move:18-56 | | |
| **Confirmed Fix At** | f753bb1 | | |

**Description** The `vault.deposit()` function does not check if the provided `receipt.vault_id` matches `vault.vault_id`. This makes it possible to provide a receipt from a different vault for the deposit process.

**Impact** If a user provides a receipt from a different vault, the deposited amount becomes unrecoverable. This is because all other user-facing functions include a call to `vault.assert_vault_receipt_matched(receipt)`, which prevents operations using mismatched receipts. Since `deposit()` lacks this check, the funds are effectively stuck and cannot be withdrawn.

**Recommendation** Include a validation in `vault.deposit()` to ensure that `receipt.vault_id` matches `vault.vault_id` before processing the deposit.

**Developers Response** The developers fixed the issue at commit f753bb1f67b4b52147e7410a30daf5f3b992d218.

### 5.3.10 V-VLT-VUL-010: Reward index miscalculation due to loss of precision

| | | | |
|---|---|---|---|
| **Severity** | Low | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `sources/reward_manager.move` | | |
| **Location(s)** | sources/reward_manager.move:371 | | |
| **Confirmed Fix At** | 96406c6 | | |

**Description** The issue was identified and reported by the protocol developers. In the `reward_manager.update_reward_indices()` function, new indices are calculated by dividing the expression `reward_amount * 10^9` with `total_shares`. However, due to integer division rounding, `add_index` can become zero in case the `total_shares` is greater than `reward_amount * 10^9`.

**Impact** If `add_index` evaluates to zero, the reward index will remain unchanged. This is more likely to occur with tokens that have low decimal precision (e.g., USDC with 6 decimals) and in vaults with a large total share count.

**Recommendation** Increase the decimal precision used in the index calculation to ensure accurate updates even in edge cases.

**Developers Response** The developers have fixed the issue at commit 96406c6ef7626cfad822e5c3f189be7d1f24891c

### 5.3.11 V-VLT-VUL-011: Maintainability Issues

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 48b4fed |
| **Type** | Maintainability | **Status** | Fixed |
| **File(s)** | `sources/volo_vault.move`, `sources/reward_manager.move`, `sources/oracle.move` | | |
| **Location(s)** | sources/volo_vault.move,sources/reward_manager.move,sources/oracle.move | | |
| **Confirmed Fix At** | 96406c6 | | |

#### Missing Events

- `vault.set_status()` does not emit an event to signal changes in the vault's status.
- `vault.set_locking_time_for_cancel_request()` lacks an event to indicate updates to the locking time for canceling requests.

#### Misleading Comments

- The comment in the `oracle.get_asset_price()` function states that the value must be updated within `MAX_UPDATE_INTERVAL` , however, the code enforces the update based on `config.update_interval` instead.
- The comment in the `reward_manager.update_reward_indices()` function states that the index should have 18 decimals; however, the actual result of the calculation has only 9 decimals, resulting in a discrepancy between the comment and the implementation.

#### Missing Checks and Validations

- The function `vault.execute_withdraw()` is missing a proper check for `receipt.vault_id == self.vault_id`, leading to a potential unhandled revert without a clear error message.
- The `vault.set_reward_manager()` function is missing a check on the vault's version before updating the reward manager.
- The `vault.denormalize_amount()` function is missing a check to see if `amount` is greater than `10 ^ decimals` and that the value for decimal exists and is not 0.
- The `oracle.set_update_interval()` function lacks a validation check to ensure that the provided `update_interval` is less than `MAX_UPDATE_INTERVAL`.
- The `volo_vault::vault_receipt_info_mut()` does not check if the `vault.receipts` contains the key `receipt_id`. If this function is called with `receipt_id` that is not present, the transaction will revert without proper error codes. This should be fixed by defining the error code for case `receipt_id` is not present in the `vault.receipts` and emitting this error when requested `receipt_id` is not present in `vault.receipts`

#### Miscellaneous

- Error codes in this `volo_vault` module coincide with those from other modules, potentially causing confusion. These codes were likely intended to be globally unique, but that is not currently enforced.

- In the `vault_oracle` module, the coin decimals are stored in two places; `PriceInfo.decimals` and `OracleConfig.coin_decimals`. Seems that at least one of them is rudimentary and needs to be removed.
- The `RewardManager.receipt_unclaimed_rewards` struct field is not used anywhere in the codebase and needs to be removed.
- The `RewardManager.receipt_reward_indices` struct field is not used anywhere in the codebase and needs to be removed.
- The `reward_manager.add_reward_balance()` function will revert if the vault has no shares due to division by zero, although no specific error code will be provided, which makes harder to identify the reason of the revert.

**Developers Response** The developers have fixed most of the observations at commit 96406c6ef7626cfad822e5c3f189be7d1f24891d.

### 5.3.12 V-VLT-VUL-012: Admins may lock in user funds in pending deposit requests

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `sources/manage.move` | | |
| **Location(s)** | sources/manage.move:1-12 | | |
| **Confirmed Fix At** | 2963e4a | | |

**Description** Users can deposit funds to the vault by calling `user_entry::deposit()` function that sets the receipt status to `PENDING_DEPOSIT_STATUS`. These funds remain in request buffer till an operator executes the request. These funds do not generate any yield till the deposit request is executed. The user can cancel the deposit request by calling `user_entry::cancel_deposit()` which in turn calls `volo_vault::cancel_deposit()`. This function asserts that the vault is in normal status as shown in the snippet below

```
public(package) fun cancel_deposit<PrincipalCoinType>(
    self: &mut Vault<PrincipalCoinType>,
    clock: &Clock,
    request_id: u64,
    receipt_id: address,
    recipient: address,
): Coin<PrincipalCoinType> {
    self.check_version();
    self.assert_normal();
```

An address with `AdminCap` may disable the vault at any time by calling `vault_manage::set_vault_enabled()`.

**Impact** When an address with `AdminCap` disables the vault, the funds in the request buffer remain stuck till the Admin enables the vault again. These funds do not generate any yield and users cannot cancel these requests

**Recommendations** Allow cancellation of deposit request when vault is disabled.

**Developers Response** Developers have fixed the issue at commit 2963e4abc2aee58999919d40c2f6ec2d544918c0.

### 5.3.13 V-VLT-VUL-013: Potential loss of precision in rewards distribution logic

| | | | |
|---|---|---|---|
| **Severity** | Warning | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Fixed |
| **File(s)** | `./sources/reward_manager.move` | | |
| **Location(s)** | ./sources/reward_manager.move:592-593 | | |
| **Confirmed Fix At** | 96406c6 | | |

**Description** This issue was identified and reported by protocol developers. The `set_reward_rate()` function is responsible for configuring the rate at which rewards are allocated. Its sole input parameter, `rate`, specifies the number of smallest reward token units to be distributed per millisecond to eligible users. The system enforces a minimum allocation of 1 smallest token unit per millisecond, which may not provide sufficient precision in all scenarios.

**Impact** Consider a scenario where the Operator intends to set a reward rate equivalent to 100 USDC per 24 hours. On the SUI Network, USDC uses 6 decimal places.

According to the current rate calculation:

`rate = (100 * 10^6) / (24 * 60 * 60 * 1000) = 10^8 / 86400000 = 1` (integer division)

As a result, only **86.4 USDC** would be distributed over 24 hours, falling short of the intended **100 USDC**. It's also important to note that most tokens on the SUI Network have a higher decimal precision than USDC. For these tokens, the loss of precision would be less significant, but this limitation still exists for tokens with lower decimal precision.

**Recommendation** This issue can be mitigated by either modifying the back-end logic to use rates divisible by 86,400,000, or by implementing support for additional decimal places in the reward rate to enhance precision.

**Developers Response** The developers have fixed the issue at commit 96406c6ef7626cfad822e5c3f189be7d1f24891c

# A Appendix

## A.1 Intended Behavior: Non-Issues of Note

### A.1.1 V-VLT-APP-VUL-001: Operator can withdraw rewards for the borrowed receipts

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 48b4fed |
| **Type** | Access Control | **Status** | Intended Behavior |
| **File(s)** | `sources/reward_manager.move` | | |
| **Location(s)** | sources/reward_manager.move:421 | | |

**Description** The owners of the `receipt` objects can claim rewards by calling `reward_manager::claim_reward()`. This function takes a mutable reference to the receipt object and returns the reward_balance object for the RewardCoinType to the caller as shown in the snippet below. Note that this function only checks that the vault is enabled. Therefore, the rewards can be claimed when a flash loan is active by an operator.

```
public fun claim_reward<PrincipalCoinType, RewardCoinType>(
    self: &mut RewardManager<PrincipalCoinType>,
    vault: &mut Vault<PrincipalCoinType>,
    clock: &Clock,
    receipt: &mut Receipt,
): Balance<RewardCoinType> {
  self.check_version();
  vault.assert_enabled();
  ...
  vault_reward_balance.split(reward_amount)
}
```

An operator can get access to any receipt object held by vault as a defi asset as part of the flashloan. A malicious operator can claim the rewards held by the receipt.

**Impact** An operator can steal all the rewards accumulated by all receipts held by a given vault as its defi assets by borrowing those receipts. This withdrawal wont cause any increase in loss calculations while closing the flashloan because rewards are not included in the USD value of the receipt and they are not included in the USD value caluclation of the vault.

**Recommendation** Allow `reward_manager::claim_principal()` to be called only when vault is not in operation.

**Developer Response** The developers have acknowledged the issue and clarified that it should not affect the current protocol configuration, which involves only trusted Operators. However, if the protocol later grants Operator roles to less trusted parties, the issue could become relevant.

### A.1.2 V-VLT-APP-VUL-002: Operator can withdraw rewards of borrowed assets that are not factored in USD value

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 48b4fed |
| **Type** | Access Control | **Status** | Intended Behavior |
| **File(s)** | `sources/reward_manager.move` | | |
| **Location(s)** | sources/reward_manager.move | | |

**Description** The vault is designed to hold both coin-type assets and DeFi assets. It maintains control over DeFi assets through ownership of associated account capabilities or position objects. Operators can initiate a flash loan, which involves temporarily borrowing assets from the vault. In the case of DeFi assets, this is achieved by transferring ownership of the relevant position objects to the operator.

To protect against loss, the protocol records the USD value of the vault at the beginning of a flash loan and verifies that any change in USD value post-loan remains within a defined tolerance.

The USD value of DeFi assets is calculated strictly based on the current value of their positions and corresponding market prices. Importantly, this valuation **does not** account for any rewards accumulated by the position objects.

**Impact** Operators can potentially extract value from the vault by exploiting reward mechanisms: Operators can borrow DeFi assets, claim any unclaimed rewards, and return the original assets-retaining the rewards without impacting the vault's recorded USD value. Since claiming rewards does not alter the USD valuation of the underlying position. Therefore, rewards can be siphoned off without triggering the vault's value-checking safeguards.

**Recommendation**

- The vault should automatically claim and distribute any accumulated rewards before allowing DeFi assets to be borrowed.
- The USD value of unclaimed rewards should be factored into the overall valuation of DeFi assets when calculating the vault's USD value.

**Developer Response** The developers have acknowledged the issue and clarified that it should not affect the current protocol configuration, which involves only trusted Operators. However, if the protocol later grants Operator roles to less trusted parties, the issue could become relevant.

### A.1.3 V-VLT-APP-VUL-003: Lending position assets do not factor in potential liquidations

| | | | |
|---|---|---|---|
| **Severity** | High | **Commit** | 48b4fed |
| **Type** | Logic Error | **Status** | Intended Behavior |
| **File(s)** | `./sources/adaptors/suilend_adaptor.move` | | |
| **Location(s)** | ./sources/adaptors/suilend_adaptor.move:22-23 | | |

**Description** The Vault can hold multiple asset types for flash loans. During a flash loan, Operators are allowed to withdraw one batch of assets and return a different batch, as long as two conditions are met: the USD value of the assets withdrawn must equal the USD value of the assets returned, and the asset types must match. The issue arises with how assets originating from lending platforms, such as SUI Lend, are valued - creating a potential for value loss that could be triggered either intentionally by a malicious Operator, or accidentally.

The current method for calculating the USD value of a position uses the formula:

**USD value = usd_value(position.borrowed) - usd_value(position.deposited)**

**Impact** This approach to calculating a position's value fails to account for whether the position is flagged for liquidation on the originating lending platform. In such cases, the formula produces a misleading result: after the flash loan is completed, the position may lose a significant portion of its perceived value due to liquidation, exposing the system to potential risk.

**Recommendation** For asset types originating from lending platforms, it is necessary to implement logic that evaluates their immediate value, adjusted for potential liquidation risk.

**Developers Response** The developers acknowledged the issue, stating that the intention is to allow the back-end to monitor the health factors of potentially problematic positions.