Navi vSUI Pool Manager

Security Assessment

May 23rd, 2025 — Prepared by OtterSec

Michał Bochnak embe221ed@osec.io

Thiago Tavares thitav@osec.io

Table of Contents

Appendices

01 — Executive Summary

Overview

Navi Protocol engaged OtterSec to assess the `lending-upgrade` program. This assessment was conducted between April 13th and May 16th, 2025. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 5 findings throughout this audit engagement.

In particular, we identified a vulnerability while withdrawing the reserve balance, where the function validates pool balance before allowing the pool manager to rebalance liquidity by unstaking vSUI, resulting in withdrawals to fail even though the required funds may be unstaked (OS-NSP-ADV-00). Additionally, rounding issues in conversions between SUI and vSUI may result in unstaking insufficient vSUI, potentially blocking withdrawals and enabling denial-of-service (OS-NSP-ADV-01). Lastly, repayments to managed pools will fail due to the withdraw balance function aborting when a pool manager exists (OS-NSP-ADV-02).

We also recommended modifying the codebase to improve efficiency and ensure adherence to best coding practices (OS-NSP-SUG-00). We further advised including additional validation logic and utilizng proper rounding direction (OS-NSP-SUG-01).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/naviprotocol/protocol. This audit was performed against commit f8f6118.

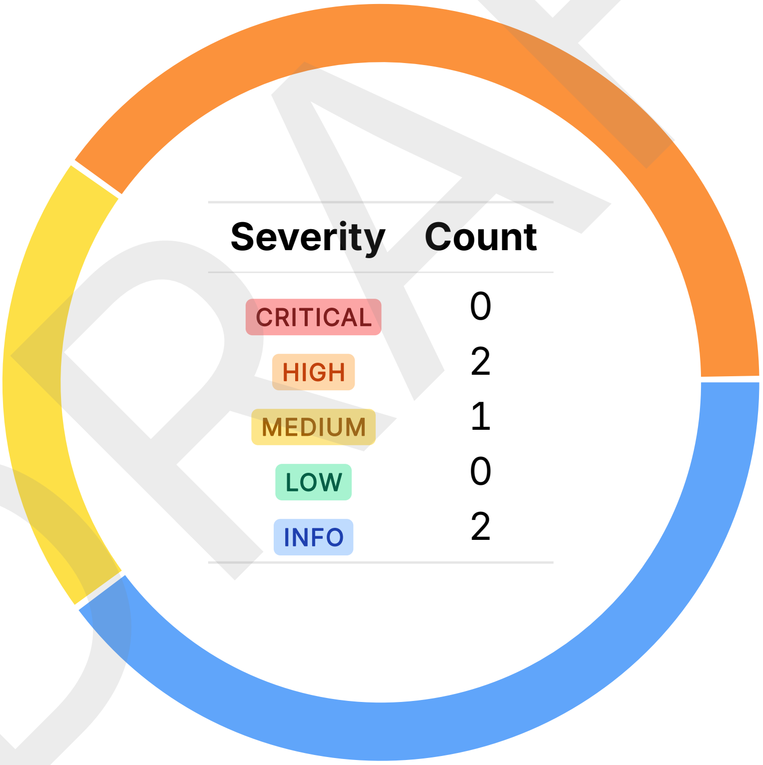
A brief description of the program is as follows:

Name	Description
lending-upgrade	It introduces a dynamic Pool Manager that optimizes idle SUI liquidity using vSUI staking, a whitelist-based liquidation system allowing only designated liquidators, and a customizable borrow fee framework for assets and users.

03 — Findings

Overall, we reported 5 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
CRITICAL	0
HIGH	2
MEDIUM	1
LOW	0
INFO	2

04 — Vulnerabilities

Here, we present a technical analysis of the vulnerabilities identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-NSP-ADV-00	HIGH	RESOLVED ✓	`withdraw_reserve_balance_v2` validates pool balance before allowing the pool manager to rebalance liquidity by unstaking `vSUI`, resulting in withdrawals to fail even though the required funds may be unstaked.
OS-NSP-ADV-01	HIGH	RESOLVED ✓	Rounding down in `sui_amount_to_lst_amount` and `lst_amount_to_sui_amount` may result in unstaking insufficient `vSUI`, potentially blocking withdrawals and enabling denial-of-service.
OS-NSP-ADV-02	MEDIUM	RESOLVED ✓	Repayments to managed pools will fail due to `pool::withdraw_balance` aborting when a `PoolManager` exists.

Incorrect Placement of Balance Assertion Check HIGH OS-NSP-ADV-00

Description

`pool::withdraw_reserve_balance_v2` immediately asserts that the pool already has enough liquid reserves (`pool.balance`) to cover the withdrawal. If the pool's liquid balance is less than `amount`, the transaction aborts early with an `insufficient_balance` error. However, this is incorrect behaviour for pools that may dynamically rebalance liquidity utilizing staked assets. When the pool's liquid `SUI` reserves are low, the `PoolManager` unstakes `vSUI` back into `SUI` to replenish liquidity via `pool_manager::prepare_before_withdraw` . Thus, due to the current ordering of the assertion check, withdrawals may fail when the pool's liquid reserves are insufficient, even though the required funds may be unstaked. As a result, managed pools cannot dynamically rebalance liquidity as intended.

>_ lending_core/sources/pool.move RUST

```
public(friend) fun withdraw_reserve_balance_v2<CoinType>(
    _: &PoolAdminCap,
    pool: &mut Pool<CoinType>,
    amount: u64,
    recipient: address,
    system_state: &mut SuiSystemState,
    ctx: &mut TxContext
) {
    let total_supply = balance::value(&pool.balance);
    assert!(total_supply >= amount, error::insufficient_balance());

    if (dynamic_field::exists_(&pool.id, PoolManagerKey {})) {
        let manage = dynamic_field::borrow_mut(&mut pool.id, PoolManagerKey {});
            let _prepare_balance = pool_manager::prepare_before_withdraw<CoinType>(manage,
                ↪ amount,balance::value(&pool.balance), system_state, ctx);
        balance::join(&mut pool.balance, _prepare_balance);
        pool_manager::update_withdraw(manage, amount);
    };
    [...]
}
```

Remediation

Move the balance assertion after the `prepare_before_withdraw` call to ensure withdrawals succeed once liquidity is replenished.

Patch

Resolved in ad613bc.

Improper Rounding During Stake Conversion HIGH OS-NSP-ADV-01

Description

The issue concerns integer rounding in proportional conversions between `SUI` and `vSUI` in `stake_pool`. `sui_amount_to_lst_amount` calculates amounts utilizing integer division, which truncates any fractional part. When the `SUI` per `vSUI` exchange rate changes due to staking rewards, these calculations may round down, producing slightly less `vSUI` than needed to cover a withdrawal.

>_ *volo_liquid_staking/sources/stake_pool.move* RUST

```rust
public fun sui_amount_to_lst_amount(
    self: &StakePool,
    metadata: &Metadata<CERT>,
    sui_amount: u64
): u64 {
    let total_sui_supply = self.total_sui_supply();
    let total_lst_supply = metadata.get_total_supply_value();
    if (total_sui_supply == 0 || total_lst_supply == 0) {
        return sui_amount
    };
    let lst_amount = (total_lst_supply as u128)
        * (sui_amount as u128)
        / (total_sui_supply as u128);
    lst_amount as u64
}

public fun lst_amount_to_sui_amount(
    self: &StakePool,
    metadata: &Metadata<CERT>,
    lst_amount: u64
): u64 {
    let total_sui_supply = self.total_sui_supply();
    let total_lst_supply = metadata.get_total_supply_value();
    assert!(total_lst_supply > 0, EZeroSupply);
    let sui_amount = (total_sui_supply as u128)
        * (lst_amount as u128)
        / (total_lst_supply as u128);
    sui_amount as u64
}
```

In `pool_manager::prepare_before_withdraw` (which calls `sui_amount_to_lst_amount`), this will result in unstaking an insufficient amount of `vSUI`, so the resulting `SUI` is less than requested. Functions such as `pool::withdraw_balance_v2`, which rely on this calculation, will fail, effectively creating a denial-of-service scenario. The similar rounding issue also applies to `lst_amount_to_sui_amount`.

Remediation

Add a check to verify if the unstaked amount is sufficient, if not, unstake 1 extra `vSUI`.

Patch

Resolved in ad7f4c2.

Repayment Failure in Managed Pools MEDIUM OS-NSP-ADV-02

Description

`lending::base_repay` always calls `pool::withdraw_balance` when `excess_amount > 0`, but this function aborts if the pool is managed by a `PoolManager`. As a result, any repayment that includes an overpayment will fail for managed pools. This creates a denial-of-service condition where users cannot complete repayments successfully.

>_ *lending_core/sources/pool.move* RUST

```
public(friend) fun withdraw_balance<CoinType>(pool: &mut Pool<CoinType>, amount: u64, user:
    ↪  address): Balance<CoinType> {
    [...]
    if (dynamic_field::exists_(&pool.id, PoolManagerKey {})) {
        abort error::invalid_function_call()
    };
    [...]
    return _balance
}
```

Remediation

Ensure to handle managed pools utilizing `withdraw_balance_v2`.

Patch

Resolved in 3346d56.

05 — General Findings

Here, we present a discussion of general findings identified during our audit. While these findings do not pose an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-NSP-SUG-00	Suggestions regarding ensuring adherence to coding best practices.
OS-NSP-SUG-01	Recommendations to ensure proper rounding direction and validation logic.

Code Maturity OS-NSP-SUG-00

Description

1. The current assertion in `pool::init_sui_pool_manager` utilizes a raw numeric code (0), which is not descriptive when it fails. Replace it with a more descriptive custom error to improve clarity.

   >_ *lending_core/sources/pool.move* RUST

   ```rust
   public fun init_sui_pool_manager(_: &PoolAdminCap, pool: &mut Pool<SUI>, stake_pool:
       ↪  StakePool, metadata: Metadata<CERT>, target_sui_amount: u64, ctx: &mut TxContext)
       ↪  {
       assert!(!dynamic_field::exists_(&pool.id, PoolManagerKey {}), 0);
       let pool_manager = pool_manager::new(stake_pool, metadata,
           ↪  balance::value(&pool.balance), target_sui_amount, tx);
       dynamic_field::add(&mut pool.id, PoolManagerKey {}, pool_manager);
   }
   ```

2. In `refresh_stake`, the `FundUpdated` event is always emitted, even when no staking or unstaking occurs, resulting in unnecessary events that do not reflect real state changes. This is inefficient and may mislead off-chain listeners. Emit the event only if one of the staking or unstaking branches executes.

   >_ *lending_core/sources/pool_manager.move* RUST

   ```rust
   public(friend) fun refresh_stake(manager: &mut SuiPoolManager, pool_balance: &mut
       ↪  Balance<SUI>, system_state: &mut SuiSystemState, ctx: &mut TxContext) {
       [...]
        emit(FundUpdated {
            original_sui_amount: manager.original_sui_amount,
            current_sui_amount: balance::value(pool_balance),
            vsui_balance_amount: balance::value(pool_vsui_balance),
            // treasury_amount: treasury_amount,
            target_sui_amount: manager.target_sui_amount
        });
   }
   ```

3. The `else if` condition in `storage::is_liquidatable` (`!table::contains(designated_liquidators, liquidator`) checks whether the liquidator is not designated, and in that case, the function returns false, implying liquidation is not allowed. However, the comment currently says *"liquidatable if not designated,"* which contradicts the logic. Fix the comment to accurately reflect the implementation.

```
>_ lending_core/sources/storage.move                                                    RUST

public fun is_liquidatable(storage: &mut Storage, liquidator: address, user: address):
    ↪  bool {
    [...]
    if (!table::contains(protected_liquidation_users, user) ||
        ↪  !*table::borrow(protected_liquidation_users, user)){ // liquidatable if not
        ↪  protected or protection is false
        return true
    } else if (!table::contains(designated_liquidators, liquidator)) { // liquidatable if
        ↪  not designated
        return false
    };
    [...]
}
```

4. `pool_manager::refresh_stake` redundantly calls `balance::value(pool_vsui_balance)` to fetch the `vSUI` balance, even though this value was already stored in `pool_vsui_amount` earlier in the function. Utilize `pool_vsui_amount` instead to improve efficiency and readability.

```
>_ lending_core/sources/pool_manager.move                                               RUST

public(friend) fun refresh_stake(manager: &mut SuiPoolManager, pool_balance: &mut
    ↪  Balance<SUI>, system_state: &mut SuiSystemState, ctx: &mut TxContext) {
    [...]
    let pool_vsui_amount = balance::value(pool_vsui_balance);
    [...]
    if (pool_sui_amount + MIN_OPERATION_AMOUNT < target) { // if pool sui amount doesn't
        ↪  meet target, unstake vsui
        [...]
        if (vsui_to_unstake > MIN_OPERATION_AMOUNT) {
            // unstake all if left vsui is less than MIN_OPERATION_AMOUNT
            if (balance::value(pool_vsui_balance) < vsui_to_unstake + MIN_OPERATION_AMOUNT)
                ↪  {
                vsui_to_unstake = balance::value(pool_vsui_balance);
            };
            [...]
        };
    }
```

Remediation

Implement the above-mentioned suggestions.

Code Refactoring OS-NSP-SUG-01

Description

1. The borrow fee calculations in `get_borrow_fee` and `get_borrow_fee_v2` within `incentive_v3` calculate fees via integer division, which truncates any fractional part, potentially undercharging borrowers. Over time, this may result in significant revenue loss for the protocol. Round up the fee to ensure the protocol collects the full intended amount.

   >_ lending_core/sources/incentive_v3.move RUST

```
fun get_borrow_fee(incentive: &Incentive, amount: u64): u64 {
    if (incentive.borrow_fee_rate > 0) {
        amount * incentive.borrow_fee_rate / constants::percentage_benchmark()
    } else {
        0
    }
}

fun get_borrow_fee_v2(incentive: &Incentive, user: address, asset_id: u8, amount: u64):
    ↪  u64 {
    [...]
    if (fee_rate > 0) {
        amount * fee_rate / constants::percentage_benchmark()
    } else {
        0
    }
}
```

2. In `flash_loan::loan_v2`, fee calculations currently truncate fractional amounts due to integer division, potentially underpaying suppliers and the treasury. Round up to ensure accurate fee collection.

   >_ *lending*$_c$*ore/sources/flash_loan.move* *RUST*

```
public(friend) fun loan_v2<CoinType>(config: &Config, _pool: &mut Pool<CoinType>, _user:
    ↪  address, _loan_amount: u64, sui_system: &mut SuiSystemState, ctx: &mut TxContext):
    ↪  (Balance<CoinType>, Receipt<CoinType>) {
    [...]
    let to_supplier = _loan_amount * cfg.rate_to_supplier /
        ↪  constants::FlashLoanMultiple();
    let to_treasury = _loan_amount * cfg.rate_to_treasury /
        ↪  constants::FlashLoanMultiple();
    [...]
}
```

3. `new` and `set_target_sui_amount` in `pool_manager` currently allow setting an arbitrarily small `target_sui_amount`, which will result in inefficient staking operations. The module relies on `MIN_OPERATION_AMOUNT` as the lower bound for staking/unstaking, to prevent zero value operations due to rounding. Add a validation to ensure `target_sui_amount > MIN_OPERATION_AMOUNT` in `new` and `set_target_sui_amount`.
4. The current implementation of `remove_asset_borrow_fee_rate` and `remove_user_borrow_fee_rate` in `incentive_v3` emits `AssetBorrowFeeRateRemoved` events even when no fee entry exists to remove. Emit events only when an actual removal occurs to avoid misleading logging of data.

Remediation

Include the above refactors in the codebase.

A — Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings.

CRITICAL

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

HIGH

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

MEDIUM

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

LOW

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

INFO

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B — Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that others may have missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.