OtterSec

Navi Protocol

Security Assessment

May 23rd, 2025 — Prepared by OtterSec

Sangsoo Kang	sangsoo@osec.io
Michał Bochnak	embe221ed@osec.io
Robert Chen	r@osec.io

Table of Contents

01 — Executive Summary

Overview

Navi Protocol engaged OtterSec to assess the `protocol` 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 7 findings throughout this audit engagement.

In particular, we identified a high-risk vulnerability concerning improper scaling and nomralization logic. Specifically, transferring small residual balances to the treasury without proper scaling or clearing user state may result in under-collateralized debt risks, incorrect treasury accounting, and potential user exploitation (OS-NVP-ADV-01). Furthermore, we highlighted multiple rounding issues, including truncation during normalization, which results in a mismatch between the calculated borrow amount and the actual withdrawal, potentially allowing users to withdraw more than intended from the pool (OS-NVP-ADV-00), and a rounding issue in the base withdrawal functionality, resulting in the withdrawable amount being zero, creating a mismatch between user storage and pool reserves (OS-NVP-ADV-02).

Lastly, the oracle configuration logic lacks proper validation, particularly when setting a new price feed, allowing invalid configurations that may later cause assertion failures. Also, critical price parameters may be set to zero, potentially breaking oracle logic or disabling deviation checks (OS-NVP-ADV-03).

We also recommended modifying the codebase to improve efficiency and ensure adherence to best coding practices (OS-NVP-SUG-00). Moreover, we advised including additional safety checks within the codebase to render it more robust and secure (OS-NVP-SUG-01, OS-NVP-SUG-02).

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 6ae8e49.

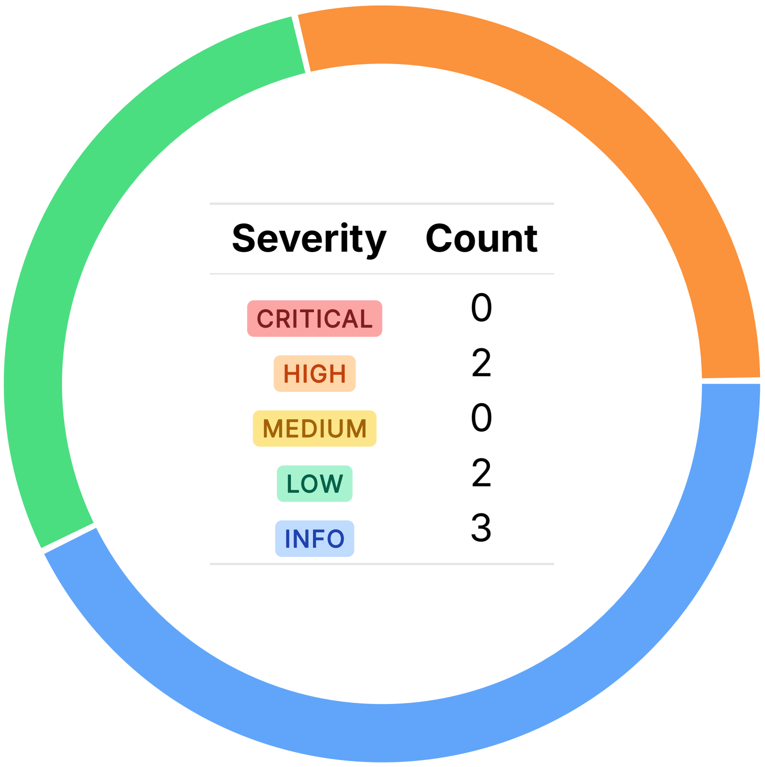
A brief description of the program is as follows:

Name	Description
protocol	Enables users to supply assets, borrow against collateral, and earn yield on SUI. The pool module manages asset reserves, handling user deposits and withdrawals, and maintains a treasury balance for protocol revenue or reserves.

03 — Findings

Overall, we reported 7 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	0
LOW	2
INFO	3

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-NVP-ADV-00	HIGH	RESOLVED ✓	Truncation during normalization results in a mismatch between the calculated borrow amount and the actual withdrawal, potentially allowing users to withdraw more than intended from the pool.
OS-NVP-ADV-01	HIGH	RESOLVED ✓	Transferring small residual balances to the treasury without proper scaling or clearing user state may result in under-collateralized debt risks, incorrect treasury accounting, and potential user exploitation.
OS-NVP-ADV-02	LOW	RESOLVED ✓	A rounding issue in `base_withdraw` may result in `withdrawable_amount` to be zero, creating a mismatch between user storage and pool reserves.
OS-NVP-ADV-03	LOW	RESOLVED ✓	The oracle configuration logic lacks proper validation, particularly when setting a new price feed, allowing invalid configurations that may later create assertion failures. Also, critical price parameters may be set to 0, potentially breaking oracle logic or disabling deviation checks.

Inaccurate Withdrawal Due to Truncated Normalization HIGH OS-NVP-ADV-00

Description

The issue in `lending::base_borrow` stems from the normalization of the borrow amount and potential truncation during decimal place conversion. Specifically, when a coin's decimal places exceed nine, truncating the amount during normalization may result in discrepancies where the amount withdrawn from the pool in `pool::withdraw_balance` may be greater than the amount originally calculated in `logic::execute_borrow`, resulting in an inaccurate withdrawal.

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

```rust
fun base_borrow<CoinType>([...]): Balance<CoinType> {
    storage::when_not_paused(storage);
    storage::version_verification(storage);

    let normal_borrow_amount = pool::normal_amount(pool, amount);
    logic::execute_borrow<CoinType>(clock, oracle, storage, asset, user, (normal_borrow_amount
        ↪  as u256));

    let _balance = pool::withdraw_balance(pool, amount, user);
    emit(BorrowEvent {
        reserve: asset,
        sender: user,
        amount: amount
    });
    return _balance
}
```

Remediation

Utilize the un-normalized value of `normal_borrow_amount` when calling `pool::withdraw_balance`. This way, the system ensures that the withdrawal aligns with the exact amount the borrow logic calculated in `execute_borrow`.

Patch

Resolved in bf485b5.

Inconsistent Handling of Residual Balance HIGH OS-NVP-ADV-01

Description

In `logic::execute_withdraw`, if a user's remaining balance after withdrawal (`token_amount - actual_amount`) is less than or equal to 1000, it is forcibly transferred to the treasury. This approach introduces several issues. Even small balances may still be critical for maintaining healthy collateralization if the user has an active debt, resulting in incorrect health factor calculations. Also, when increasing the treasury balance `storage::increase_treasury_balance`, `token_amount` and `actual_amount` are real values, adjusted utilizing the supply index (`scaled*index`), while the `treasury_balance` expects a scaled amount. This may create discrepancies between the treasury's actual holdings and its recorded balance, resulting in accounting inconsistencies.

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

```
public(friend) fun execute_withdraw<CoinType>([...]): u64 {
    [...]
    if (token_amount > actual_amount) {
        if (token_amount - actual_amount <= 1000) {
            // Tiny balance cannot be raised in full, put it to treasury
            storage::increase_treasury_balance(storage, asset, token_amount - actual_amount);
            if (is_collateral(storage, asset, user)) {
                storage::remove_user_collaterals(storage, asset, user);
            }
        };
    };
    [...]
}
```

Moreover, after transferring the balance to the treasury, the user's `TokenBalance.user_state` is not updated, allowing them to later re-utilize this non-existent balance if they deposit again. Thus, the current logic for transferring residual amounts to the treasury introduce risks of under-collateralized debt positions, treasury reserve inconsistencies, and potential exploitation of phantom balances.

Remediation

Reconsider the requirement to transfer the leftover balance to the treasury. If this logic is necessary, ensure that the corresponding amount is subtracted from `TokenBalance.user_state`, and convert `token_amount - actual_amount` to its scaled representation before passing the value to `increase_treasury_balance`.

Patch

Resolved in 044f386, 6af978f, and 7d0182f.

Rounding Discrepancy During Withdrawal LOW OS-NVP-ADV-02

Description

In `lending_core::base_withdraw` , there may be a discrepancy between the on-chain pool and internal storage accounting, as the function converts a normalized withdrawal amount into an actual token amount utilizing `pool::unnormal_amount`. Due to rounding, `withdrawable_amount` may become zero even when the normalized amount is non-zero. If not checked, this will result in the user's internal position to be reduced, without actually transferring any tokens from the pool.

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

```rust
fun base_withdraw<CoinType>([...]): Balance<CoinType> {
    [...]
    let withdrawable_amount = pool::unnormal_amount(pool, normal_withdrawable_amount);
    let _balance = pool::withdraw_balance(pool, withdrawable_amount, user);
    emit(WithdrawEvent {
        reserve: asset,
        sender: user,
        to: user,
        amount: withdrawable_amount,
    });
    return _balance
}
```

Remediation

Add a check to ensure `withdrawable_amount > 0` before proceeding.

Patch

Resolved in ee8c0b2.

Improper Validation Logic During Oracle Configuration LOW OS-NVP-ADV-03

Description

The current implementation of `oracle::config` lacks proper validation of the oracle configuration. Specifically, `config::new_price_feed` allows creating a price feed with `price_diff_threshold1 > price_diff_threshold2`, which violates the constraint enforced in `set_price_diff_threshold1_to_price_feed` (shown below). Consequently, later attempts to update `threshold1` (via `set_price_diff_threshold1_to_price_feed`) will revert unless the new value is `≤ threshold2`. Ensure to add consistent validation during creation to mitigate this issue.

>_ *oracle/sources/config.move* RUST

```rust
public(friend) fun set_price_diff_threshold1_to_price_feed(cfg: &mut OracleConfig, feed_id:
    ↪  address, value: u64) {
      assert!(table::contains(&cfg.feeds, feed_id), error::price_feed_not_found());
      let price_feed = table::borrow_mut(&mut cfg.feeds, feed_id);
      let before_value = price_feed.price_diff_threshold1;
      if (price_feed.price_diff_threshold2 > 0) {
          assert!(value <= price_feed.price_diff_threshold2, error::invalid_value());
      };
      [...]
}
```

Additionally, `set_price_diff_threshold2_to_price_feed` and `set_maximum_effective_price_to_price_feed` lack explicit checks to reject `value == 0`, allowing potentially invalid configurations. If `threshold2` or `maximum_effective_price` is set to zero, it will break the oracle logic or disable deviation checks. For example, it will render it impossible for any real price to pass validation. Add assertions to check `value > 0` in `set_price_diff_threshold2_to_price_feed` and `set_maximum_effective_price_to_price_feed`.

Remediation

Incorporate the above validations into the oracle configuration logic.

Patch

Resolved in d6a6ad7.

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-NVP-SUG-00	Suggestions regarding ensuring adherence to coding best practices.
OS-NVP-SUG-01	There are several instances where proper validation is not performed, resulting in potential issues.
OS-NVP-SUG-02	Additional safety checks may be incorporated within the codebase to make it more robust and secure.

Code Maturity OS-NVP-SUG-00

Description

1. In `storage`, getter functions unnecessarily utilize `&mut Storage` instead of `&Storage`. This restricts parallel access, creates ambiguity about whether a function mutates state, and reduces efficiency. Getter functions should always take an immutable `&Storage` since they only read data without modifying it.

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

   ```rust
   public fun get_oracle_id(storage: &Storage, asset: u8): u8 {
       table::borrow(&storage.reserves, asset).oracle_id
   }

   public fun get_coin_type(storage: &Storage, asset: u8): String {
       table::borrow(&storage.reserves, asset).coin_type
   }
   ```

2. Currently, `oracle::update_price` sets the price timestamp to the blockchain's current time (`clock::timestamp_ms`), rather than the time when the price was actually generated by the oracle (Pyth or Supra). To accurately detect stale prices, the function should utilize the timestamp from Pyth or Supra.

   >_ *oracle/sources/oracle.move* RUST

   ```rust
   // function to internally update prices by oracle_pro
   public(friend) fun update_price(clock: &Clock, price_oracle: &mut PriceOracle, oracle_id:
       ↪  u8,token_price: u256) {
       // TODO: update_token_price can be merged into update_price
       version_verification(price_oracle);

       let price_oracles = &mut price_oracle.price_oracles;
       assert!(table::contains(price_oracles, oracle_id), error::non_existent_oracle());

       let price = table::borrow_mut(price_oracles, oracle_id);
       let now = clock::timestamp_ms(clock);
       [...]
       price.timestamp = now;
   }
   ```

3. It will be appropriate to utilize Pyth's `conf` (confidence interval) field when updating prices in `Oracle`. The `conf` value measures uncertainty around the price. Incorporating it helps detect unreliable or volatile price updates, improving protocol safety by filtering out risky price data during high market instability.

4. In Move, arithmetic operations such as addition and multiplication automatically abort on overflow, rendering manual `assert!` checks for overflows in `ray_math` and `safe_math` redundant. Remove the overflow checks in `add` and `mul` within `safe_math`, and in `ray_to_wad` and `wad_to_ray` in `ray_math`.

Remediation

Implement the above-mentioned suggestions.

Missing Validation Logic

OS-NVP-SUG-01

Description

1. Add a version check in `storage::withdraw_treasury` to ensure the function only operates on the correct storage layout in the correct version.
2. Ensure that `withdrawable_amount` is non-zero in `storage::withdraw_treasury` before proceeding to prevent zero-value withdrawals, which waste gas and clutter the event log with unnecessary events.

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

```rust
    public fun withdraw_treasury<CoinType>(
        _: &StorageAdminCap,
        pool_admin_cap: &PoolAdminCap,
        storage: &mut Storage,
        asset: u8,
        pool: &mut Pool<CoinType>,
        amount: u64,
        recipient: address,
        ctx: &mut TxContext
    ) {
    [...]
    let withdrawable_amount = pool::unnormal_amount(pool, (withdrawable_value as u64));
    [...]
}
```

3. Within `Storage`, when setting the `LiquidationFactors` (`liquidation_ratio` and `liquidation_bonus`), it is important to validate that the sum of `liquidation_ratio` and `liquidation_ratio × liquidation_bonus` does not exceed 100%. Without this check, liquidations may unfairly seize more than a user's total collateral. This validation should be added in both `set_liquidation_ratio` and `set_liquidation_bonus` to ensure safe liquidation parameters.

Remediation

Include the above validations into the codebase.

Additional Safety Checks OS-NVP-SUG-02

Description

1. The current assertion in `dynamic_calculator::dynamic_health_factor` ensures either `estimate_supply_value` or `estimate_borrow_value` are non-zero. However, it does not handle the case when both `estimate_supply_value` and `estimate_borrow_value` are zero. The correct condition should enforce that exactly one of the two values is non-zero.

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

   ```
   public fun dynamic_health_factor<CoinType>([...]): u256 {
       assert!(!(estimate_supply_value > 0 && estimate_borrow_value > 0),
           ↪  error::non_single_value());
       [...]
   }
   ```

2. `oracle_dynamic_getter::get_dynamic_single_price` first checks if the entire oracle configuration is paused and immediately returns a `paused` error without verifying if the requested `feed_address` exists in the oracle config. This implies that if a user requests a price for a feed that is not actually configured, they will receive a misleading `paused` error instead of a `price_feed_not_found` error.

   >_ *oracle/sources/oracle_dynamic_getter.move* RUST

   ```
   /// a dynamic function to fetch a latest price without actually updating the price
   /// return (result, price)
   #[allow(unused_assignment)]
   public fun get_dynamic_single_price(clock: &Clock, oracle_config: &OracleConfig,
       ↪  price_oracle: &PriceOracle, supra_oracle_holder: &OracleHolder, pyth_price_info:
       ↪  &PriceInfoObject, feed_address: address): (u64, u256) {
           config::version_verification(oracle_config);
           if(config::is_paused(oracle_config)) {
               return (error::paused(), 0)
           };
       [...]
   }
   ```

3. `create_incentive_pool` and `get_pool_from_funds_pool` in `incentive_v2` do not validate that the provided `IncentiveFundsPool<T>` is actually associated with the given `Incentive` object. This missing check may result in logical inconsistencies, such as pools referencing unrelated or unauthorized funds. The module should enforce ownership by validating the `funds.id` against a registry.

Remediation

Add the missing validations mentioned above.

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.