Veridise

Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

Navi Lending Protocol

Veridise Inc.

October 16, 2024

► **Prepared For:**

Navi
https://naviprotocol.io/

► **Prepared By:**

Jon Stephens
Ajinkya Rajput
Andreea Buțerchi

► **Contact Us:** contact@veridise.com

► **Version History:**

Oct. 14, 2024	V1
Mar. 28, 2024	Initial Draft

Contents

1 Executive Summary

From Mar. 11, 2024 to Mar. 26, 2024, Navi engaged Veridise to review the security of their Navi Lending Protocol. The review covered the on-chain contracts of the Navi Lending protocol. This included the core logic, on-chain centralized oracle infrastructure, on-chain reporting interface and several utility contracts. Veridise conducted the assessment over 6 person-weeks, with 3 engineers reviewing code over 2 weeks on commit `945878c`. The auditing strategy involved a tool-assisted analysis of the source code performed by Veridise engineers as well as an extensive manual code review.

Project summary. This project covered the on-chain infrastructure of the Navi protocol, which contains a lending protocol as well as a centralized oracle. Similar to AAVE, the lending protocol allows liquidity providers to deposit funds that may be loaned out to others in return for yields. The liquidity providers may additionally use their deposited funds as collateral to borrow funds from the protocol. Funds borrowed from the protocol will accrue interest over time to compensate both Navi and the liquidity providers who deposited the borrowed funds. The protocol tracks the viability of these loans by ensuring that users have more collateral deposited than the funds that they have borrowed. Should the health of a loan drop below a certain threshold, it will be subject to liquidation. On a liquidation, a liquidator will have the opportunity to purchase the collateral of the liquidated user at a discounted rate using the loaned asset. In addition to this functionality, the Navi protocol also allows users to receive flash loans in return for a fee that will be provided to the liquidity providers and the Navi protocol.

The Navi protocol also contains a centralized oracle that is used to provide asset prices to the lending protocol. The oracle allows owners to create a price feed for an asset which will be updated by "feeders." In this context, a feeder is an account that has access to a feeder capability. With this capability, they will be able to update the price feed of an asset. The lending protocol can then access the asset price provided most recently by a feeder.

Code assessment. The Navi Lending Protocol developers provided the source code of the Navi Lending Protocol contracts for review. The code appears to have been developed entirely by the Navi developers, but takes heavy inspiration from AAVE V2. To facilitate the Veridise auditors' understanding of the code, access to the protocol's developer documentation website was provided, which documents the intended behavior of the protocol at a high level. Additionally, the code contained some in-line comments on structs and functions. The delivered source code also contained a test suite which the Veridise auditors noted tested many of the expected user-flows and much of the protocol's behavior.

Summary of issues detected. The audit uncovered 23 issues, 6 of which are assessed to be of high or critical severity by the Veridise auditors. Specifically, V-NAVI-VUL-001 identified a case where an invalid oracle-id could be used, V-NAVI-VUL-002, V-NAVI-VUL-003 and V-NAVI-VUL-005 identified locations where values tracking internal funds were not updated correctly, V-NAVI-VUL-004 identified the potential for an admin to unknowingly overdraw

funds from the liquidity pool, and V-NAVI-VUL-006 identified missing validation that could allow users to make use of a stale API. The Veridise auditors also identified 2 medium-severity issues, including the potential for an oracle to update an asset's price twice in a single transaction (V-NAVI-VUL-007) and the potential for users to overcome restrictions placed on flash loans (V-NAVI-VUL-008, V-NAVI-VUL-004). In addition to this, 7 low-severity issues, 7 warnings, and 1 informational findings were also reported to developers.

Recommendations. After auditing the protocol, the auditors had a few suggestions to improve the Navi Lending Protocol and to avoid similar issues to those discovered in the audit in the future.

Naming Convention. In the project, there is some terminology that appears to be used inconsistently in a few locations. Two examples of this are "normalized balances" and "scaled balances". A normalized balance is one that has 9 decimals and a scaled balance is one that has been divided by the index to initialize the interest accrual. This terminology, however, is used inconsistently across the codebase as sometimes unscaled values are referred to as scaled and similarly a few unnormalized values are referred to as normalized. We would recommend that the developers adopt and document a specific naming convention that they use across the codebase to make it easier to reason about.

Programming Model. There are several best practices in the Move programming language, some of which are documented in our issues (V-NAVI-VUL-020, V-NAVI-VUL-018). We would recommend that the Navi developers adhere to these best practices as they can prevent potential mistakes in the future.

Use Custom Data Types. There are several types of values that are distinct but have the same data-type (normalized vs unnormalized values, scaled vs unscaled values, oracle vs asset identifiers). In this audit, we found a few issues (V-NAVI-VUL-001, V-NAVI-VUL-004) where a value was erroneously used, but was accepted by the codebase because the data type was correct. These issues could have been avoided by making these types unique. This could be achieved by wrapping the values in structs and adding functionality to operate on or convert between struct types. Doing so could allow the type-checker to find issues similar to those found in the audit.

Documentation While there is some inline documentation in the codebase, it would be useful to document the intended behavior of functions that perform state modifications. High-level inline documentation could help both developers and future auditors understand the intended behavior of the protocol quicker.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
Navi Lending Protocol	`945878c`	SUI Move	SUI

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Mar. 11 - Mar. 26, 2024	Manual & Tools	3	6 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	2	2	1
High-Severity Issues	4	4	3
Medium-Severity Issues	2	2	0
Low-Severity Issues	7	7	2
Warning-Severity Issues	7	7	1
Informational-Severity Issues	1	1	0
TOTAL	23	23	7

Table 2.4: Category Breakdown.

Name	Number
Logic Error	11
Data Validation	4
Maintainability	2
Version Validation	1
Centralization	1
Missing/Incorrect Events	1
Gas Optimization	1
Composability	1
Value Mutability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of Navi Lending Protocol's smart contracts. In our audit, we sought to answer questions such as:

- Does the protocol appropriately track user funds?
- Is the protocol version appropriately validated?
- Can a user maintain a loan that does not have adequate leverage?
- Can a user avoid protocol safety checks?
- Are computations performed on values with appropriate units?
- Are coin identifiers appropriately validated?
- Is it possible to manipulate oracle prices?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved a combination of human experts and automated program analysis & testing tools. In particular, we conducted our audit with the aid of the following techniques:

- *Property-based Testing.* To identify some behavioral inconsistencies, we used property-based testing using SUI's build-in testing framework. To do so, we formalized properties that we should hold in the protocol and wrote tests to pseudo-randomly exercise the protocol and check these properties.

Scope. The scope of this audit is limited to the on-chain contracts in the Navi Protocol repository at commit `945878c`.

Methodology. Veridise auditors reviewed the reports of previous audits for Navi Lending Protocol, inspected the provided tests, and read the Navi Lending Protocol documentation. They then began a manual review of the code assisted by property-based testing. During the audit, the Veridise auditors regularly met with the Navi Lending Protocol developers to ask questions about the code.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.1: 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

Table 3.2: 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) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

Table 3.3: 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 - OR - 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 - OR - 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

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-NAVI-VUL-001	Wrong oracle used during liquidation	Critical	Fixed
V-NAVI-VUL-002	Inaccurate Tracking of Funds	Critical	Partially Fixed
V-NAVI-VUL-003	Double counting of residual balance in withdraw	High	Acknowledged
V-NAVI-VUL-004	Admin can Overdraft Treasury	High	Fixed
V-NAVI-VUL-005	Double counting of treasury funds	High	Fixed
V-NAVI-VUL-006	No Version Validation in Flash Loan Repay	High	Fixed
V-NAVI-VUL-007	Oracle Price can be Set Twice in One Transaction	Medium	Partially Fixed
V-NAVI-VUL-008	Users can Exceed Maximum Flashloan	Medium	Acknowledged
V-NAVI-VUL-009	Duplicate Error Code	Low	Fixed
V-NAVI-VUL-010	Missing Validation during Resource Initialization	Low	Partially Fixed
V-NAVI-VUL-011	Centralization Risk	Low	Fixed
V-NAVI-VUL-012	No Validation asset validation in Flashloan modul…	Low	Acknowledged
V-NAVI-VUL-013	Incorrect Decimals used in Conversion	Low	Acknowledged
V-NAVI-VUL-014	No Validation that CoinType matches asset_id	Low	Acknowledged
V-NAVI-VUL-015	Liquidation Restrictions do not Consider Large …	Low	Acknowledged
V-NAVI-VUL-016	Wrong event emitted during liquidation	Warning	Fixed
V-NAVI-VUL-017	Unnecessary SafeMath module	Warning	Acknowledged
V-NAVI-VUL-018	Transfer to Sender Reduces Composability	Warning	Acknowledged
V-NAVI-VUL-019	Function should be #[test_only]	Warning	Acknowledged
V-NAVI-VUL-020	References are Unnecessairly Mutable	Warning	Acknowledged
V-NAVI-VUL-021	Duplicate Function	Warning	Acknowledged
V-NAVI-VUL-022	Unused events	Warning	Acknowledged
V-NAVI-VUL-023	Account utilities may result in undesired behav…	Info	Acknowledged

4.1 Detailed Description of Issues

4.1.1 V-NAVI-VUL-001: Wrong oracle used during liquidation

Severity	Critical	Commit	945878c
Type	Logic Error	Status	Fixed
File(s)	logic.move		
Location(s)	calculate_liquidation		
Confirmed Fix At	597bf5d		

Similar to other lending protocols, users with unhealthy positions may be the target of a liquidation where a portion of their collateral is auctioned off at a discount to recover some of the users' debt. The function `calculate_liquidation` in the logic module, shown below, is used to determine the amount of collateral that is purchased by the liquidator and the cost of that collateral in the debt asset.

```
fun calculate_liquidation(
    clock: &Clock,
    storage: &mut Storage,
    oracle: &PriceOracle,
    liquidated_user: address,
    collateral_asset: u8,
    loan_asset: u8,
    repay_amount: u256, // 6000u
): (u256, u256, u256, u256, u256, bool) {
    ...

    let bonus = ray_math::ray_mul(max_liquidable_collateral_value, liquidation_bonus)
    ; // 5950u * 5% = 297.5u
    let treasury_factor = storage::get_treasury_factor(storage, collateral_asset); //
     10%

    let treasury_reserved_collateral = ray_math::ray_mul(bonus, treasury_factor); //
    297.5u * 10% = 29.75u
    //  max_liquidable_collateral - bonus * treasury_factor
    let liquidator_bonus_collateral = max_liquidable_collateral_value -
    treasury_reserved_collateral; // 5950u - 29.75
    // max_liquidable_collateral_value
    let liquidable_collateral_balance = calculator::calculate_amount(clock, oracle,
    max_liquidable_collateral_value, collateral_asset);
    //  max_liquidable_collateral_value
    let liquidable_loan_balance = calculator::calculate_amount(clock, oracle,
    max_liquidable_collateral_value, loan_asset);

    let bonus_balance = calculator::calculate_amount(clock, oracle,
    liquidator_bonus_collateral, collateral_asset);
    let treasury_reserved_collateral_balance = calculator::calculate_amount(clock,
    oracle, treasury_reserved_collateral, collateral_asset);
    // excess_value + bonus
    let excess_amount = calculator::calculate_amount(clock, oracle, excess_value +
    bonus, loan_asset);
```

Snippet 4.1: Definition of the `calculate_liquidation` function

This function takes the amount of debt to liquidate and returns the amount of collateral purchased, any unused liquidation funds to be returned to the liquidator and the amount of collateral that should be given to the treasury as a fee.

It first gets the value of a borrower's collateral value and loan value in USD. To do so, it gets the `loan_oracle_id` by calling `storage::get_oracle_id(loan_asset)` where `loan_asset` is an internal identifier for the given asset assigned by the storage module. Note that while both the oracle id and asset id both have type `u8`, the identifiers might not be the same as they are assigned by different modules.

The protocol then calculates the value of borrow tokens repaid by the liquidator and the value of the collateral and borrow tokens that is equivalent to the `repay_value`. The function then converts the value of the tokens to be transferred back to liquidator and to be transferred to the protocol treasury using the `calculate_amount` function. This function takes an oracle id as its second argument and uses the indicated oracle to return the number of coins that have the given value as shown below.

```
1 public fun calculate_amount(clock: &Clock, oracle: &PriceOracle, value: u256,
      oracle_id: u8): u256 {
2     let (is_valid, price, decimal) = oracle::get_token_price(clock, oracle, oracle_id
      );
3     assert!(is_valid, error::invalid_price());
4     value * (sui::math::pow(10, decimal) as u256) / price
5 }
```

Snippet 4.2: Snippet from `calculate_amount()` in calculator.move

While the `calculate_amount` function expects to be passed an `oracle_id`, the `calculate_liquidation` function instead passes it an asset id (either `collateral_asset` or `loan_asset`). Since it is not enforced that an asset id is equivalent to an oracle id, this can return the wrong number of coins.

Impact The wrong oracle can be used to calculate the amount of tokens for the following values in `calculate_liquidation`. In the function, the potentially incorrect values are:

- `liquidable_collateral_balance`
- `liquidable_loan_balance`
- `bonus_balance`
- `treasury_reserved_collateral_balance`
- `excess_amount`

Using the wrong id can cause the oracle to misreport the price of a token and therefore can cause the incorrect number of coins to be returned by the function.

Recommendation Get the oracle id for the collateral asset by calling `storage::get_oracle_id()` in `collateral_oracle_id` and pass

- `collateral_oracle_id` to `convert_amount` to convert the following values to tokens
  - `max_liquidable_collateral_value`
  - `liquidator_bonus_collateral`

- `treasury_reserved_collateral`

- And `loan_oracle_id` to `convert_amount` to convert the following values to tokens
  - `liquidator_bonus_collateral`
  - `excess_value+bonus`

4.1.2 V-NAVI-VUL-002: Inaccurate Tracking of Funds

Severity	Critical	**Commit**	945878c
Type	Logic Error	**Status**	Partially Fixed
File(s)	`logic.move`		
Location(s)	update_state		
Confirmed Fix At	N/A		

The NAVI protocol collects compound interest from borrowers and provides simple interest to liquidity providers. A part of the borrow interest, in proportion to `reserve_factor`, is collected by the protocol in the the treasury. The remaining part of the borrow interest is provided to the liquidity providers and is distributed via a simple interest calculation.

Interest Rate Calculations

The simple and compound interest rate is calculated according to following snippets

```
public fun calculate_borrow_rate(storage: &mut Storage, asset: u8): u256 {
    let (base_rate, multiplier, jump_rate_multiplier, _, optimal_utilization) =
    storage::get_borrow_rate_factors(storage, asset);

    let utilization = caculate_utilization(storage, asset);

    if (utilization < optimal_utilization) {
        // Equation: borrow_rate = base_rate + (utilization * multiplier)
        base_rate + ray_math::ray_mul(utilization, multiplier)
    } else {
        // Equation: borrow_rate = base_rate + (optimal_utilization * multiplier) +
    ((utilization - optimal_utilization) * jump_rate_multiplier)
        base_rate + ray_math::ray_mul(optimal_utilization, multiplier) + ray_math::
    ray_mul((utilization - optimal_utilization), jump_rate_multiplier)
    }
}
```

Snippet 4.3: Snippet from `calculate_borrow_rate()` in calculator.move

```
public fun calculate_supply_rate(storage: &mut Storage, asset: u8, borrow_rate: u256)
    : u256 {
    let (_, _, _, reserve_factor, _) = storage::get_borrow_rate_factors(storage,
    asset);
    let utilization = caculate_utilization(storage, asset);

    ray_math::ray_mul(
        ray_math::ray_mul(borrow_rate, utilization),
        ray_math::ray() - reserve_factor
    )
    // borrow_rate * utilization * (ray_math::ray() - reserve_factor)
}
```

Snippet 4.4: Snippet from `calculate_supply_rate()` in calculator.move

Here, the simple interest rate is $1 - reserve\_factor$ fraction of $borrow\_rate \times utilization$.

Interest Amount Calculations

The interest calculations return the factor that, if multiplied with the total supply, returns the new total supply with accrued interest. The simple interest calculation is performed as shown in the snippet below.

```
public fun calculate_linear_interest(
    timestamp_difference: u256,
    rate: u256
): u256 {
    ray_math::ray() + rate * timestamp_difference / constants::seconds_per_year()
}
```

Snippet 4.5: Snippet from `calculate_linear_interest()` in calculator.move

The compound interest collected from borrows is calculated as shown in the following snippet

```
public fun calculate_compounded_interest(
    timestamp_difference: u256,
    rate: u256
): u256 {
    // // e.g. get the time difference of the last update --> (1685029315718 - 1685029255718) / 1000 == 60s
    if (timestamp_difference == 0) {
        return ray_math::ray()
    };

    // time difference minus 1 --> 60 - 1 = 59
    let exp_minus_one = timestamp_difference - 1;

    // time difference minus 2 --> 60 - 2 = 58
    let exp_minus_two = 0;
    if (timestamp_difference > 2) {
        exp_minus_two = timestamp_difference - 2;
    };

    // e.g. get the rate per second --> (6.3 * 1e27) / (60 * 60 * 24 * 365) --> 1.9977168949771689 * 1e20 = 199771689497716894977
    let rate_per_second = rate / constants::seconds_per_year();

    let base_power_two = ray_math::ray_mul(rate_per_second, rate_per_second);
    let base_power_three = ray_math::ray_mul(base_power_two, rate_per_second);

    let second_term = timestamp_difference * exp_minus_one * base_power_two / 2;
    let third_term = timestamp_difference * exp_minus_one * exp_minus_two * base_power_three / 6;
    ray_math::ray() + rate_per_second * timestamp_difference + second_term + third_term
    // ray_math::ray() + rate_per_second * timestamp_difference
}
```

Snippet 4.6: Snippet from `calculate_compounded_interest()` in calculator.move

The collected interest on borrows is distributed to the treasury and suppliers' `supply_index` in `update_state()` as shown below.

```
fun update_state(clock: &Clock, storage: &mut Storage, asset: u8) {
    // e.g. get the current timestamp in milliseconds
    let current_timestamp = clock::timestamp_ms(clock);

    let last_update_timestamp = storage::get_last_update_timestamp(storage, asset);
    // get the last timestamp from reserve
    let timestamp_difference = (current_timestamp - last_update_timestamp as u256) /
    1000; // calcualte the time difference

    let (current_supply_index, current_borrow_index) = storage::get_index(storage,
    asset); // get the current index from reserve
    let (current_supply_rate, current_borrow_rate) = storage::get_current_rate(
    storage, asset); // get the current rate from reserve

    // get new supply index
    let linear_interest = calculator::calculate_linear_interest(timestamp_difference,
     current_supply_rate);
    let new_supply_index = ray_math::ray_mul(linear_interest, current_supply_index);

    // get new borrow index
    let compounded_interest = calculator::calculate_compounded_interest(
    timestamp_difference, current_borrow_rate);
    let new_borrow_index = ray_math::ray_mul(compounded_interest,
    current_borrow_index);

    ////////////////////////////////////////////////////////////////////////
    // Calculate the portion of the amount spent on community governance //
    ////////////////////////////////////////////////////////////////////////
    // e.g. get the reserve factor
    let (_, _, _, reserve_factor, _) = storage::get_borrow_rate_factors(storage,
    asset);

    // e.g. get the total amount of the current pool that has been lent out 10ETH -->
     10 * 1e9
    let (total_supply, total_borrow) = storage::get_total_supply(storage, asset);

    let interest_on_borrow = ray_math::ray_mul(total_borrow, (new_borrow_index -
    current_borrow_index));
    let interest_on_supply = ray_math::ray_mul(total_supply, (new_supply_index -
    current_supply_index));

    let scaled_borrow_amount = ray_math::ray_div(interest_on_borrow, new_borrow_index
    );
    let scaled_supply_amount = ray_math::ray_div(interest_on_supply, new_supply_index
    );

    let reserve_amount = ray_math::ray_mul(
        ray_math::ray_mul(total_borrow, (new_borrow_index - current_borrow_index)),
        reserve_factor
    );

    let scaled_reserve_amount = ray_math::ray_div(reserve_amount, new_borrow_index);
    storage::update_state(storage, asset, new_borrow_index, new_supply_index,
    current_timestamp, scaled_reserve_amount);
}
```

Snippet 4.7: Snippet from `update_state()` in logic.move

Here, the number of fees to be collected in treasury is stored in the `reserve_amount` variable and is calculated by multiplying the total interest collected by the `reserve_factor`. The rewards for the liquidity providers are distributed by updating the `supply_index`. The updated supply index is calculated in `supply_index` by multiplying `current_supply_index` by the interest rate factor calculated as shown above. Interest is then accrued to the treasury by multiplying the interest accrued on borrows by the reserve factor and scaled by the borrow index.

Even though the supply (simple) interest rate is $1 - reserve\_factor$ of the borrow rate, the interest on borrows is compounded and therefore $interest\_on\_borrow > (treasury\_fee + supply\_interest)$. This excess interest remains unaccounted and cannot be withdrawn because:

- `withdraw_treasury` in storage.move intends to limit the amount that can be withdrawn from the treasury to `ReserveData.treasury_balance` which is updated by `reserve_amount` only
- This excess interest is not provided to liquidity providers due to the difference between compounding interest and simple interest.

In addition to this, the treasury is scaled by the borrow index, meaning that for the existing balance of the treasury, it will accrue interest at the same rate as the borrows. As a result, the treasury will accrue compound interest faster than it otherwise should, potentially resulting in $interest\_on\_borrow < (treasury\_fee + supply\_interest)$.

Impact If the incoming interest on borrows is less than the interest paid to the treasury and liquidity providers, the protocol will experience a shortfall where either users or the treasury will not be able to withdraw all funds. On the other hand if the incoming interest from borrows is greater than the interest paid to the treasury and liquidity providers, some funds will be locked within the protocol as only supplier and liquidity provider funds are tracked. To determine the impact of the above issues on the protocol we produced the following graphs. They show the difference between the borrow interest accrued and the supply interest accrued + the treasury fee, thereby showing the amount of funds locked.

Recommendation Rather than maintaining a separate scaled treasury value, instead maintain a separate treasury balance that is increased by the difference between the interest accrued by borrowers and supplied to liquidity providers.

Remediation Navi has addressed the issue where the treasury is scaled by the borrow index but the discrepancy between linear and compound interest has not been addressed.

Developer Response NAVI has not withdrawn any treasury from the protocol during this period until the index is properly scaled. As a result, the situation where $interest\_on\_borrow < (treasury\_fee + supply\_interest)$ has been prevented. Additionally, the linear and compound model, as applied by AAVE, has been proven in the market as a solid and safe approach.

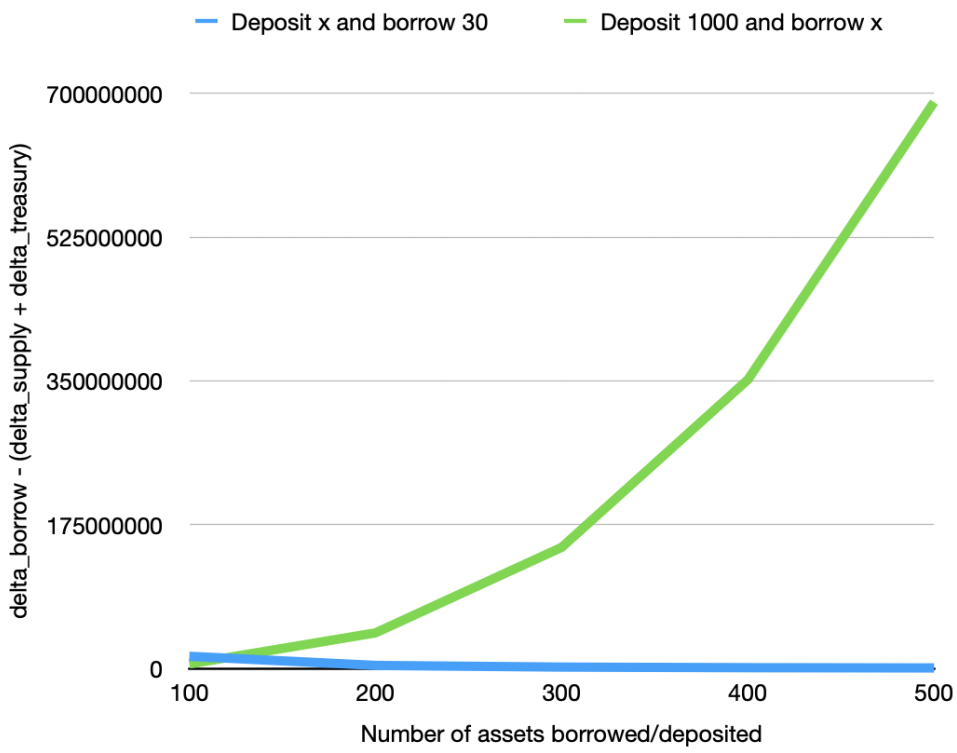

Figure 4.8: Here the y-axis represents values of `delta_borrow - (delta_supply + delta_treasury)` Each x-axis tick represents a separate scenario where the protocol is initialized and the `update_state()` is called after a year of borrow. Green curve: For each tick on the x-axis, a scenario is run where a deposit of 1000 is made and protocol is run with borrow of x Blue curve: For each x on x-axis, a scenario is run where a deposit of x is made and withdraw of 30 is made in each scenario

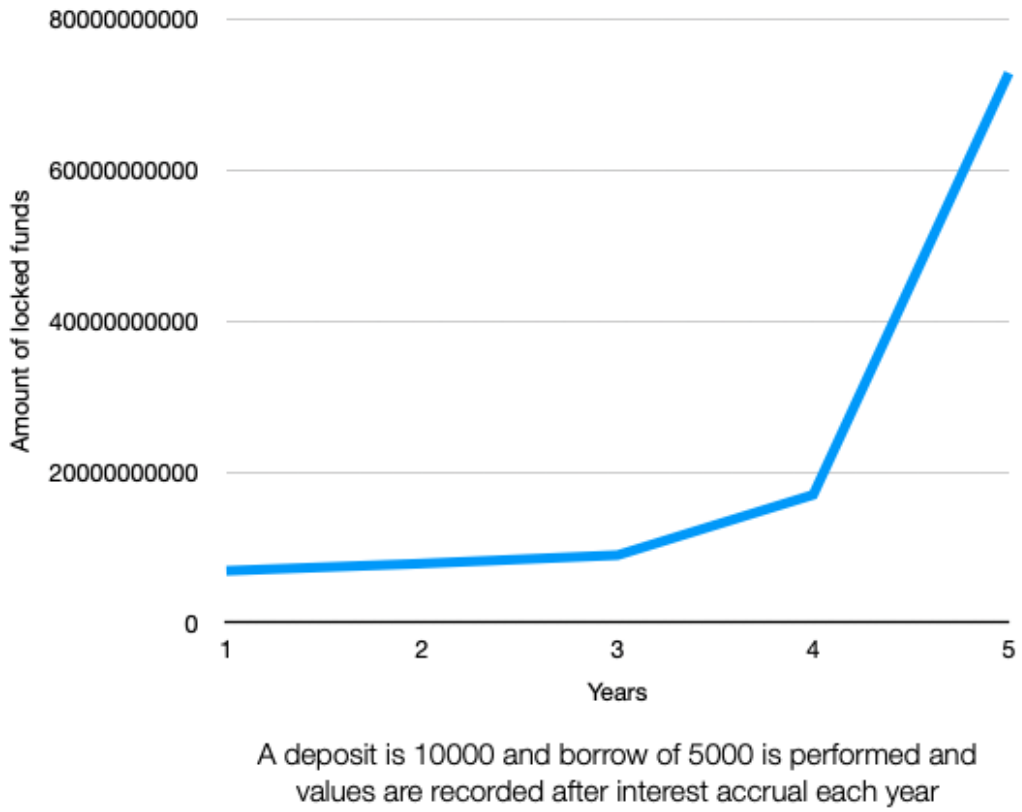

Figure 4.9: A deposit of 10000 assets is made and 5000 is borrowed. The interest is accrued i.e. `update_state()` once each year for 5 times

4.1.3 V-NAVI-VUL-003: Double counting of residual balance in withdraw

Severity	High	**Commit**	945878c
Type	Logic Error	**Status**	Acknowledged
File(s)	`logic.move`		
Location(s)	execute_withdraw		
Confirmed Fix At	N/A		

The `execute_withdraw` function in the `logic` module contains the bulk of the logic to perform a withdraw on behalf of some user. After performing the withdraw and updating the user's balance, the protocol checks to see if there is any residual balance remaining in the account (an amount < 0.000001 token). Since several tokens use 6 decimals rather than 9 and since withdrawing the residual amount would likely cost more in gas, the protocol automatically takes the small residual as fees as shown below.

```
public(friend) fun execute_withdraw<CoinType>(
        clock: &Clock,
        oracle: &PriceOracle,
        storage: &mut Storage,
        asset: u8,
        user: address,
        amount: u256 // e.g. 100USDT -> 100000000000
    ): u64 {
        ...

        decrease_supply_balance(storage, asset, user, actual_amount);
        ...

        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);
                }
            };
        };

        update_interest_rate(storage, asset);
        (actual_amount as u64)
    }
```

Snippet 4.10: Snippet from `execute_withdraw()` that takes residual balances as fees

When the protocol takes these residual balances for the treasury, however, it does not adjust the user's balance to remove the residual amount. As a result, these residual balances are counted twice, once in the treasury balance and once in the original user's balance. In addition, for coins who have decimals greater than or equal to 9, these residual balances can still be withdrawn by the user.

Impact Since the user's balance is not decreased, some coins are double counted, causing some funds not to be backed. Therefore, if the treasury withdraws all of its funds, some users will not be able to withdraw theirs. While over time this discrepancy between the tracked balances and the actual funds would increase, an attacker could exacerbate the situation while costing themselves a small amount of funds. To do so, for SUI, they could deposit 1e-6 SUI and withdraw 1e-9 SUI in a loop. Each withdraw would increase the residual balance in the treasury, even though the attacker will eventually withdraw all of their funds.

Recommendation Reduce the user balance after the residual balance is added to the treasury.

4.1.4 V-NAVI-VUL-004: Admin can Overdraft Treasury

Severity	High	Commit	945878c
Type	Logic Error	Status	Fixed
File(s)	storage.move		
Location(s)	withdraw_treasury		
Confirmed Fix At	604afc3		

The protocol receives fees from various sources, including interest on loans and fees from liquidations. Some of these fees are stored in a separate treasury, but others are kept in the same pool as user funds. The protocol uses the `treasury_balance` field in the `ReserveData` struct to track the funds in the user pool that are owed to the protocol. As fees accumulate, an admin with the `StorageAdminCap` can call the `withdraw_treasury` function to withdraw the treasury fees from the pool as shown below.

```
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 coin_type = get_coin_type(storage, asset);
    assert!(coin_type == type_name::into_string(type_name::get<CoinType>()), error::
    invalid_coin_type());

    let reserve = table::borrow_mut(&mut storage.reserves, asset);

    let treasury_balance = &mut reserve.treasury_balance;

    // let treasury_balance = get_treasury_balance(storage, asset);
    let unnormal_amount = pool::unnormal_amount(pool, (*treasury_balance as u64));

    let withdraw_amount = amount;
    if (amount < unnormal_amount) {
        withdraw_amount = unnormal_amount;
    };

    *treasury_balance = *treasury_balance - (withdraw_amount as u256);
    pool::withdraw_reserve_balance<CoinType>(
        pool_admin_cap,
        pool,
        withdraw_amount,
        recipient,
        ctx
    );
}
```

Snippet 4.11: Snippet from `withdraw_treasury()` in storage.move

This function allows the user to specify the amount of funds to withdraw in the same decimals

as the coin. The `treasury_balance`, however, is normalized so that it always has 9 decimals regardless of how many decimals the coin uses. The function therefore first "unnormalizes" the `treasury_balance` so that `unnormal_amount` contains the treasury's balance in the coin's decimals. The function then sets the amount to withdraw to the treasury's balance if too few funds were requested. Finally, it reduces the normalized `treasury_balance` by the unnormalized `withraw_amount` and sends the funds to the recipient address.

The main problem with this function is that the treasury balance will be reduced by an unnormalized amount. Depending on the relationship between the normalized decimals (9) and the coin's decimals, this could allow an admin to retrieve too few or too many funds. The problem is further exacerbated by the check against the normalized treasury balance as an admin could accidentally request too many funds from the protocol. In such cases, all of those funds would be withdrawn, as long as the normalized treasury balance isn't exceeded.

Impact Since it is common for coins to have less than 9 decimals (such as USDC, USDT and WETH), admins are likely to overdraft the treasury and withdraw user funds.

Recommendation Change the above code so that:

1. Normalize the `withdraw_amount` and scale the result by the borrow index
2. Reduce the `treasury_balance` by the scaled and normalized `withdraw_amount`
3. Change the conditional so that the treasury balance is withdrawn if the admin requests more than the balance of the treasury.

4.1.5 V-NAVI-VUL-005: Double counting of treasury funds

Severity	High	Commit	945878c
Type	Logic Error	Status	Fixed
File(s)	`logic.move`		
Location(s)	update_state		
Confirmed Fix At	9e4ab4d		

The NAVI protocol collects interest from borrowers and collects `reserve_factor` portion of it as fees in the `update_state()` function shown below.

```
fun update_state(clock: &Clock, storage: &mut Storage, asset: u8) {
    ...

    // e.g. get the reserve factor
    let (_, _, _, reserve_factor, _) = storage::get_borrow_rate_factors(storage,
    asset);

    // e.g. get the total amount of the current pool that has been lent out 10ETH -->
     10 * 1e9
    let (total_supply, total_borrow) = storage::get_total_supply(storage, asset);

    let interest_on_borrow = ray_math::ray_mul(total_borrow, (new_borrow_index -
    current_borrow_index));
    let interest_on_supply = ray_math::ray_mul(total_supply, (new_supply_index -
    current_supply_index));

    let scaled_borrow_amount = ray_math::ray_div(interest_on_borrow, new_borrow_index
    );
    let scaled_supply_amount = ray_math::ray_div(interest_on_supply, new_supply_index
    );

    let reserve_amount = ray_math::ray_mul(
        ray_math::ray_mul(total_borrow, (new_borrow_index - current_borrow_index)),
        reserve_factor
    );

    let scaled_reserve_amount = ray_math::ray_div(reserve_amount, new_borrow_index);

    //////////////////////
    // update reserve. //
    //////////////////////
    storage::update_state(storage, asset, new_borrow_index, new_supply_index,
    current_timestamp, scaled_reserve_amount);
    storage::increase_balance_for_pool(storage, asset, scaled_supply_amount,
    scaled_borrow_amount+scaled_reserve_amount)
}
```

Snippet 4.12: Snippet from `update_state()` in logic.move

The fee amount is calculated in `reserve_amount` variable by multiplying the collected interest by `reserve_factor`. This amount is then scaled down according to the `new_borrow_index` in `scaled_reserve_amount` variable. Then this variable is passed to `increase_balance_for_pool()` which adds the `scaled_reserve_amount` to `ReserveData.treasury_balance` in the reserve data for

the pool.

Impact Since the `scaled_treasury_amount` is part of the borrow interest, the treasury balance is already accounted for in the borrows and adding it twice will lead to double counting of treasury balance in the borrow. This adds erroneous borrows that will not be repaid and will keep accumulating with each call to `update_state()` when interest is accrued.

Eventually this will cause the utilization rate of the pool to keep rising indefinitely and reach `borrow_cap_ceiling` after which the borrows will be disabled for the protocol.

Recommendation Do not accumulate the treasury balance to the borrows as the borrow interest already includes the treasury fees.

4.1.6 V-NAVI-VUL-006: No Version Validation in Flash Loan Repay

Severity	High	**Commit**	945878c
Type	Version Validation	**Status**	Fixed
File(s)	`flash_loan.move`		
Location(s)	repay		
Confirmed Fix At	9412a5c		

The SUI ecosystem allows developers to update their protocol over time. When they do so, however, users may still access previous versions that were deployed. As a result, it is recommended that applications include a version field in struct and to check that the invoked source code matches the documented version in the struct. If developers forget to do so, users could modify resources using stale versions of the protocol, potentially resulting in unexpected states. In most locations, the Navi protocol does perform appropriate version checks, however, this check is missing in the flashloan module's repay function shown below.

Impact Should changes be made to the repay function in the future, users would still be able to repay flashloans using the version shown above. This could compromise the developers' abilities to fix bugs and to add new behaviors.

Recommendation Validate the version of the source code.

Remediation The storage version will be checked when `logic::cumulate_to_supply_index` is invoked. As long as Navi updates the storage version every time the flashloan module is updated, the flashloan version will implicitly be checked.


```
public(friend) fun repay<CoinType>(...): Balance<CoinType> {
    let Receipt {user, asset, amount, pool, fee_to_supplier, fee_to_treasury} =
    _receipt;
    assert!(user == _user, error::invalid_user());
    assert!(pool == object::uid_to_address(pool::uid(_pool)), error::invalid_pool());

    // handler logic
    {
        logic::update_state_of_all(clock, storage);
        let asset_id = get_storage_asset_id_from_coin_type(storage, type_name::
    into_string(type_name::get<CoinType>()));

        let normal_value = pool::normal_amount(_pool, fee_to_supplier);
        let (supply_index, _) = storage::get_index(storage, asset_id);
        let scaled_fee_to_supplier = ray_math::ray_div((normal_value as u256),
    supply_index);

        logic::cumulate_to_supply_index(storage, asset_id, scaled_fee_to_supplier);
        logic::update_interest_rate(storage, asset_id);
    };

    let repay_value = balance::value(&_repay_balance);
    assert!(repay_value >= amount + fee_to_supplier + fee_to_treasury, error::
    invalid_amount());

    let repay = balance::split(&mut _repay_balance, amount + fee_to_supplier +
    fee_to_treasury);
    pool::deposit_balance(_pool, repay, _user);
    pool::deposit_treasury(_pool, fee_to_treasury);

    emit(FlashRepay {
        sender: _user,
        asset: asset,
        amount: amount,
        fee_to_supplier: fee_to_supplier,
        fee_to_treasury: fee_to_treasury,
    });

    _repay_balance
}
```

Snippet 4.13: Definition of the repay function in the flashloan module

4.1.7 V-NAVI-VUL-007: Oracle Price can be Set Twice in One Transaction

Severity	Medium	**Commit**	945878c
Type	Data Validation	**Status**	Partially Fixed
File(s)	`oracle.move`		
Location(s)	update_token_price		
Confirmed Fix At	N/A		

A centralized oracle currently provides token prices to the Navi protocol. With this oracle, admins can appoint "feeders" who regularly update pricing information with prices from external sources. While these feeders are considered to be trusted, there are few restrictions placed on them when they provide updates. Most notably the feeder can update an asset's price multiple times in a single transaction using the `update_token_price()` function. Since this function also does not emit an event whenever the price is updated, it can be difficult for someone monitoring the protocol to determine this occurred.

```
public entry fun update_token_price(
    _: &OracleFeederCap,
    clock: &Clock,
    price_oracle: &mut PriceOracle,
    oracle_id: u8,
    token_price: u256,
) {
    version_verification(price_oracle);

    let price_oracles = &mut price_oracle.price_oracles;
    assert!(table::contains(price_oracles, oracle_id), ENONEXISTENT_ORACLE);
    let price = table::borrow_mut(price_oracles, oracle_id);
    price.value = token_price;
    price.timestamp = clock::timestamp_ms(clock);
}
```

Snippet 4.14: Snippet from `update_token_price()` in oracle.move

Impact If a feeder were to be compromised, the attacker could manipulate the protocol with few indications that they were doing so. As an example, a compromised feeder could do the following within a single transaction:

1. Update the price of some asset `C` to a low value
2. Liquidate positions that use `C` as collateral to purchase it at the reduced price
3. Return the price of `C` to the real price.

Recommendation Add additional validation when `update_token_price()` is called so that it cannot be invoked multiple times in the same transaction. Also consider emitting an event whenever the price is updated and limiting large changes in a single update.

Remediation To remediate this issue as well as some centralization risks, Navi has switched their centralized oracle to a decentralized oracle and transferred ownership of the feeders tot he

multi-sig. While the code referenced in this issue still remains in the decentralized oracle, this issue is addressed as long as no new feeders are created and the existing feeders are unused.

4.1.8 V-NAVI-VUL-008: Users can Exceed Maximum Flashloan

Severity	Medium	**Commit**	945878c
Type	Logic Error	**Status**	Acknowledged
File(s)	`flash_loan.move`		
Location(s)	loan		
Confirmed Fix At	N/A		

The Navi protocol allows users to take out flashloans of supported assets for a fee. As these assets will be loaned out from the liquidity pool, restrictions are placed on the size of flashloans by imposing a maximum and minimum loan size as shown below. While this will limit the amount of funds loaned in a *single* flashloan, it does not consider the case where a user takes out multiple flashloans on the same asset.

```
public(friend) fun loan<CoinType>(config: &Config, _pool: &mut Pool<CoinType>, _user:
     address, _loan_amount: u64): (Balance<CoinType>, Receipt<CoinType>) {
    version_verification(config);
    let str_type = type_name::into_string(type_name::get<CoinType>());
    assert!(table::contains(&config.support_assets, *ascii::as_bytes(&str_type)),
    error::reserve_not_found());
    let asset_id = table::borrow(&config.support_assets, *ascii::as_bytes(&str_type))
    ;
    let cfg = table::borrow(&config.assets, *asset_id);

    let pool_id = object::uid_to_address(pool::uid(_pool));
    assert!(_loan_amount >= cfg.min && _loan_amount <= cfg.max, error::invalid_amount
    ());

    ...
}
```

Snippet 4.15: Snippet of the loan function that limits the size of a flashloan

Impact Users can exceed the maximum flashloan size set by developers by taking out multiple flashloans. These checks are therefore insufficient to reduce risk for the liquidity pool.

Recommendation Limit users to a single flashloan per asset.

4.1.9 V-NAVI-VUL-009: Duplicate Error Code

Severity	Low	**Commit**	945878c
Type	Logic Error	**Status**	Fixed
File(s)	`error.move`		
Location(s)	duplicate_config, invalid_user		
Confirmed Fix At	fc6e845		

The protocol emits various numerical errors codes through out the protocol. Typically these error codes are intended to be unique to disambiguate between types of errors. Both `duplicate_coinfig` and `invalid_user`, however, return the same error code as shown below.

```
public fun duplicate_config(): u64 {2001}
public fun invalid_user(): u64 {2001}
```

Snippet 4.16: Definitions of `duplicate_coinfig` and `invalid_user`

Impact Same error code returned for two different errors will hamper the usability of the protocol where a user repaying flash loan will get a wrong error and will not be able to identify the cause of the revert.

This will also cause difficulties in testing and debugging the protocol.

Recommendation Return different unused error codes for `duplicate_config()` and `invalid_user()`.

4.1.10 V-NAVI-VUL-010: Missing Validation during Resource Initialization

Severity	Low	**Commit**	945878c
Type	Data Validation	**Status**	Partially Fixed
File(s)	`storage.move`, `incentive_v2.move`, `incentive.move`		
Location(s)	init_reserve, create_incentive_pool, add_pool		
Confirmed Fix At	N/A		

The Navi protocol has several functions that admins can use to create and initialize new resources such as pools and reserves. While the input to these functions is trusted, they do not adequately protect against potential configuration mistakes that could have significant impacts on the protocol.

```
public entry fun init_reserve<CoinType>(
    ...
) {
    version_verification(storage);

    let current_idx = storage.reserves_count;
    assert!(current_idx < constants::max_number_of_reserves(), error::
    no_more_reserves_allowed());
    reserve_validation<CoinType>(storage);

    let reserve_data = ReserveData {
        ...
    };

    table::add(&mut storage.reserves, current_idx, reserve_data);
    storage.reserves_count = current_idx + 1;

    let decimals = coin::get_decimals(coin_metadata);
    pool::create_pool<CoinType>(pool_admin_cap, decimals, ctx);
}
```

Snippet 4.17: Definition of the `init_reserve` function

As an example, consider the `init_reserve` function which creates and initializes a `ReserveData` struct for a new asset as shown above. This function configures the various parameters for a pool including the `oracle_id` corresponding to the asset's pricing oracle. There is, however, no validation that the referenced pricing oracle corresponds to the same asset. In addition, various values such as the `reserve_factor` and `optimal_utilization` expect to receive values in a particular range (< 1 RAY), but that is never validated.

Impact The absence of validation may lead to configuration errors which might lead to financial losses to users and/or protocol.

Recommendation Add appropriate validation when initializing new resources.

Developer Response We have implemented validations for the core parameters in the protocol, such as TVL, liquidation threshold, and borrow factor. However, we have decided to allow some

flexibility in the validation of `incentive_v2.move` and `incentive.move`, as they are built on top of the lending core and do not affect the core data.

4.1.11 V-NAVI-VUL-011: Centralization Risk

Severity	Low	**Commit**	945878c
Type	Centralization	**Status**	Fixed
File(s)	`oracle.move`, `storage.move`, `pool.move`, `incentive.move`		
Location(s)	N/A		
Confirmed Fix At	N/A		

Similar to many projects, the Navi Protocol's Oracle, Storage and Pool modules declare declare administrator capabilities that are given special permissions. In particular, these administrators can take the following actions:

- Create new oracle feeders
- Create new oracle price feeds
- Update oracle price feeds
- Manage liquidity pool reserve metadata
- Withdraw funds from the protocol's treasury
- Add flashloan assets
- Manage protocol incentives and withdraw incentive funds

Impact If a private key were stolen, a hacker would have access to the capabilities that manage sensitive functionality and could compromise the project.

Recommendation As these are all particularly sensitive operations, we would encourage the developers to utilize a secure private key management system, decentralized governance or multi-sig contract as opposed to a single account, which introduces a single point of failure.

Remediation Navi has updated their protocol to make use of decentralized oracles and transferred feeders to the multi-sig. This reduces the risk of private-key theft so we have resolved this issue but the owners of the threshold key should ensure their share is stored in a safe location. Protocol users should also be aware of the potential for counterparty risk if all keys are owned by the same entity. We would also further recommend that different capabilities be managed by separate multi-sig wallets.

4.1.12 V-NAVI-VUL-012: No Asset Validation in Flashloan module

Severity	Low	Commit	945878c
Type	Data Validation	Status	Acknowledged
File(s)	flash_loan.move		
Location(s)	get_storage_asset_id_from_coin_type		
Confirmed Fix At	N/A		

When an asset is made available to flashloan by calling the `create_asset` function shown below, the asset's identifier and the coin type name must be provided. In the flashloan module, it is assumed that the asset's coin has the same coin type name as the one passed into the function but it is not validated by the function.

```
public(friend) fun create_asset(
    config: &mut Config,
    _asset_id: u8,
    _coin_type: String,
    _pool: address,
    _rate_to_supplier: u64,
    _rate_to_treasury: u64,
    _max: u64,
    _min: u64,
    ctx: &mut TxContext
) {
    version_verification(config);
    assert!(!table::contains(&config.support_assets, *ascii::as_bytes(&_coin_type)),
    error::duplicate_config());

    let new_id = object::new(ctx);
    let new_obj_address = object::uid_to_address(&new_id);

    let asset = AssetConfig {
        id: new_id,
        asset_id: _asset_id,
        coin_type: _coin_type,
        pool_id: _pool,
        rate_to_supplier: _rate_to_supplier,
        rate_to_treasury: _rate_to_treasury,
        max: _max,
        min: _min,
    };
    table::add(&mut config.assets, new_obj_address, asset);
    table::add(&mut config.support_assets, *ascii::as_bytes(&_coin_type),
    new_obj_address);

    emit(AssetConfigCreated {
        sender: tx_context::sender(ctx),
        config_id: object::uid_to_address(&config.id),
        asset_id: new_obj_address,
    })
}
```

Snippet 4.18: Snippet from `example()`

Impact While the `manage` module does call `create_asset` correctly, it is possible that future updates to the protocol could call the function incorrectly. If the asset identifier did not match the coin type name, a flashloan could be taken out while updating the wrong metadata in storage.

Recommendation We recommend that each module enforce any invariants it expects to hold.

4.1.13 V-NAVI-VUL-013: Incorrect Decimals used in Conversion

Severity	Low	Commit	945878c
Type	Logic Error	Status	Acknowledged
File(s)	incentive.move		
Location(s)	get_incentive_apy, get_incentive_apy_one		
Confirmed Fix At	N/A		

The `lending_ui::incentive_getter` module allows users to request information about the protocol's incentives. Two of these functions, including the one shown below, calculate the APY of the protocol's incentives. As part of the calculation, this module will normalize an incentive pool's total supply to 9 decimals. When doing so, however, the number of decimals used by the incentive asset is assumed to be a constant value, which may not match the actual decimals used by the asset.

```
public fun get_incentive_apy_one(clock: &Clock, incentive: &Incentive, storage: &mut
    Storage, price_oracle: &PriceOracle, pool: address): IncentiveAPYInfo {
    ...

    let normal_total_supply = pool::convert_amount(total_supply, 6, 9);

    ...
}
```

Snippet 4.19: Location in `get_incentive_apy_one` that always uses 6 decimals

Impact The APYs reported by the `incentive_getter` module may be incorrect.

Recommendation Retrieve the number of decimals used by the incentive asset rather than assuming it is 6 or the same number of decimals as the price oracle.

4.1.14 V-NAVI-VUL-014: No Validation that CoinType matches asset_id

Severity	Low	**Commit**	945878c
Type	Data Validation	**Status**	Acknowledged
File(s)	`dynamic_calculator.move`		
Location(s)	dynamic_health_factor, dynamic_calculate_apy		
Confirmed Fix At	N/A		

The Navi protocol allows users to request information about the state of the protocol using the `dynamic_calculator` module. To do so, two functions, including `dynamic_calculate_apy` shown below, require the user to pass in Navi's fund pool for an asset along with the asset's storage identifier that can be used to retrieve information about the asset from storage. To ensure the Pool's coin matches the asset id, other locations of the codebase will compare a stored identifier for the asset id to the Coin's type name string. This check is missing in the `dynamic_calculator` module, however, allowing someone to pass in an asset identifier that does not correspond to the input coin.

```
public fun dynamic_calculate_apy<CoinType>(
    clock: &Clock,
    storage: &mut Storage,
    pool: &mut Pool<CoinType>,
    asset: u8,
    estimate_supply_value: u64,
    estimate_borrow_value: u64,
    is_increase: bool
): (u256, u256){
    assert!(!(estimate_supply_value > 0 && estimate_borrow_value > 0), error::
    non_single_value());
    ...
}
```

Snippet 4.20: Definition of the `dynamic_calculate_apy` function

Impact If the Coin does not correspond to the input asset id, the module will report incorrect information to the user.

Recommendation Validate that the type name of the Coin matches the asset ID's type name.

4.1.15 V-NAVI-VUL-015: Liquidation Restrictions do not Consider Large Price Fluctuations

Severity	Low	Commit	945878c
Type	Logic Error	Status	Acknowledged
File(s)	`logic.move`		
Location(s)	calculate_liquidation		
Confirmed Fix At	N/A		

Similar to AAVE V2, Navi only allows a certain percentage of a user's collateral or loan to be liquidated at once. This is done to provide users extra time to deposit additional collateral and to not penalize users too harshly during quick price fluctuations. In certain cases, however, when the price of an asset is rapidly decreasing, it can be important to allow liquidators to act quickly so that they may purchase collateral before the value of the loan is worth more than the value of the collateral.

```
fun calculate_liquidation(
    clock: &Clock,
    storage: &mut Storage,
    oracle: &PriceOracle,
    liquidated_user: address,
    collateral_asset: u8,
    loan_asset: u8,
    repay_amount: u256, // 6000u
): (u256, u256, u256, u256, u256, bool) {
    // define eth price = 1700
    let (liquidation_ratio, liquidation_bonus, _) = storage::get_liquidation_factors(
    storage, collateral_asset); // 35%, 5%
    let collateral_value = user_collateral_value(clock, oracle, storage,
    collateral_asset, liquidated_user); // 10eth * price = 1700 * 10 = 17000u

    // 17000 * 35% = 5950u
    let max_liquidable_collateral_value = ray_math::ray_mul(collateral_value,
    liquidation_ratio); // 17000 * 35% = 5950u
    let loan_value = user_loan_value(clock, oracle, storage, loan_asset,
    liquidated_user); // 14400u

    ...
}
```

Snippet 4.21: Location where the maximum liquidation is calculated in the `calculate_liquidation` function

Impact Since a full liquidation can take a long time (theoretically infinite), a user may not be fully liquidated before their loan balance is greater than the value of their collateral. In this case, the user would have no incentive to pay their loan back, resulting in a loss for the protocol.

Recommendation Similar to AAVE V3, consider adding another threashold that would allow 100% of a user's loan to be liquidated.

4.1.16 V-NAVI-VUL-016: Wrong event emitted during liquidation

Severity	Warning	**Commit**	945878c
Type	Incorrect Events	**Status**	Fixed
File(s)	`logic.move`		
Location(s)	execute_liquidation		
Confirmed Fix At	5e8a485		

The `execute_liquidate` function in logic module performs validation and updates the internal protocol state for liquidation. During the validation, the protocol checks if the user account is healthy as only unhealthy user accounts can be liquidated. The check happens in the following snippet:

```
update_state_of_all(clock, storage);

validation::validate_liquidate<CoinType, CollateralCoinType>(storage, loan_asset,
    collateral_asset, amount);

// Check the health factor of the user
assert!(!is_health(clock, oracle, storage, liquidated_user), error::user_is_unhealthy
    ());
```

Snippet 4.22: Snippet from `execute_liquidate()` in logic.move

If an attempt is made to liquidate a healthy user account, `error::user_is_unhealthy()` error is thrown. This is inconsistent with the cause of the revert, i.e. the user account is healthy and hence cannot be liquidated.

Impact This may hamper the usability and the maintainability of the protocol. The wrong error does not help in understanding the cause of the revert. This will also hamper the tracking errors in the transaction therefore hampering testing/debugging.

Recommendation Emit a `user_is_healthy()` error.

4.1.17 V-NAVI-VUL-017: Unnecessary SafeMath module

Severity	Warning	Commit	945878c
Type	Gas Optimization	Status	Acknowledged
File(s)	safe_math.move		
Location(s)	N/A		
Confirmed Fix At	N/A		

The Navi protocol defines a `safe_math` module which appears to provide utilities that prevent arithmetic errors similar to `SafeMath` contracts in Solidity before 0.8.0. The Move language, however, already reverts operations that cause integers to overflow. As such, using this module costs extra gas but does not provide additional benefit, especially since some operations will revert before causing an assertion violation as shown below.

```
public fun add(a: u256, b: u256): u256 {
    let c = a + b;
    assert!(c >= a, SAFE_MATH_ADDITION_OVERFLOW);
    return c
}
```

Snippet 4.23: Definition of the `add` function which will revert on an overflow before violating the assertion

Impact Invoking these functions costs the protocol additional gas.

Recommendation Remove the `safe_math` module.

4.1.18 V-NAVI-VUL-018: Transfer to Sender Reduces Composability

Severity	Warning	**Commit**	945878c
Type	Composability	**Status**	Acknowledged
File(s)	`incentive_v2.move`, `oracle.move`		
Location(s)	Multiple		
Confirmed Fix At	N/A		

Unlike Solidity, the Move programming language does not support dynamic dispatch. Instead they rely on resource passing and Move scripts to compose complex operations. Move does have a few operations that reduce composability by locking resources or passing resources directly to the EOA that invoked the transaction. One such example is making a `public_transfer` to the transaction sender as (1) resources transferred in this way cannot be retrieved in the transaction and (2) it potentially bypasses intermediate modules that cannot retrieve the resource. In the Navi protocol, there is significant use of this pattern, but typically there is an alternative function that may be invoked which will instead return the resource. In the cases of `entry_deposit_on_behalf_of_user`, `entry_repay_on_behalf_of_user` and several admin functions such as `add_funds` no such alternative exists.

```
public fun entry_repay_on_behalf_of_user<CoinType>(
    ...
) {
    update_reward_all(clock, incentive, storage, asset, user);

    let _balance = lending::repay_on_behalf_of_user<CoinType>(clock, oracle, storage,
     pool, asset, user, repay_coin, amount, ctx);

    let _balance_value = balance::value(&_balance);
    if (_balance_value > 0) {
        let _coin = coin::from_balance(_balance, ctx);
        transfer::public_transfer(_coin, tx_context::sender(ctx));
    } else {
        balance::destroy_zero(_balance)
    }
}
```

Snippet 4.24: Definition of the `entry_replay_on_behalf_of_user` function

Impact The use of `public_transfer` may become restrictive as intermediate modules will likely not be able to interact with these functions even though they are public. As an example, by using `public_transfer` in admin functions, the protocol will have to re-implement several APIs if they want to use on-chain governance as funds will always be sent to the sender of the transaction rather than the governance treasury.

Recommendation Try to only use `public_transfer` to the transaction's sender only in entry functions and prefer returning resources in public functions.

4.1.19 V-NAVI-VUL-019: Function should be #[test_only]

Severity	Warning	Commit	945878c
Type	Logic Error	Status	Acknowledged
File(s)	storage.move		
Location(s)	get_reserve_for_testing		
Confirmed Fix At	N/A		

The Move language allows developers to define functions that may only be used during testing by tagging them with `#[test_only]`. The Navi developers typically define these functions in a group at the end of a module and annotate the function name with the suffix "for_testing". In the storage module, the function `get_reserve_for_testing` appears to be intended as one of these test only functions, but is not annotated with the property.

```
#[test_only]
public fun get_reserve_info_for_testing(storage: &Storage, asset: u8): (bool) {
    let reserve = table::borrow(&storage.reserves, asset);
    (
        reserve.is_isolated
    )
}

public fun get_reserve_for_testing(storage: &Storage, asset: u8): (&ReserveData) {
    table::borrow(&storage.reserves, asset)
}
```

Snippet 4.25: Snippet showing the definition of `get_reserve_for_testing()` in storage.move

Recommendation Annotate the function with `#[test_only]`.

4.1.20 V-NAVI-VUL-020: References are Unnecessairly Mutable

Severity	Warning	Commit	945878c
Type	Value Mutability	Status	Acknowledged
File(s)	`storage.movegetter.move`		
Location(s)	Mulitple		
Confirmed Fix At	N/A		

The Move language allows references to be either mutable or immutable depending on whether the referenced value can be changed. To prevent accidental changes, it is considered best practice to only accept mutable references if a function may modify a value. In the Navi protocol, several functions are passed mutable references that can instead be immutable references, including the `get_supply_cap_ceiling()`, `get_user_state()`, and `get_user_balance()` functions shown below.

```
public fun get_supply_cap_ceiling(storage: &mut Storage, asset: u8): u256 {
    table::borrow(&storage.reserves, asset).supply_cap_ceiling
}
```

Snippet 4.26: Definition of the `get_supply_cap_ceiling()` in `storage.move`

This function receives a mutable reference to storage and then borrows a reference to `ReserveData` for `asset` and returns the `supply_cap_ceiling()` field in the `ReserveData` struct. The function `get_supply_cap_ceiling()` only retrieves a field and does not modify the `ReserveData` struct. Therefore, the reference to storage need not be mutable.

Definitions of the `get_user_state()` in `getter.move` and `get_user_balance()` in `storage.move`

```
public fun get_user_state(storage_: &mut Storage, user: address): (vector<
    UserStateInfo>) {
    let info = vector::empty<UserStateInfo>();
    let length = storage::get_reserves_count(storage_);

    while (length > 0) {
      let (supply, borrow) = storage::get_user_balance(storage_, length-1,
    user);

      vector::push_back(&mut info, UserStateInfo {
        asset_id: length-1,
        supply_balance: supply,
        borrow_balance: borrow,
      });

      length = length - 1;
    };

    info
}
```

```
public fun get_user_balance(storage: &mut Storage, asset: u8, user: address
    ): (u256, u256) {
```



```
    let reserve = table::borrow(&storage.reserves, asset);
    let supply_balance = 0;
    let borrow_balance = 0;

    if (table::contains(&reserve.supply_balance.user_state, user)) {
        supply_balance = *table::borrow(&reserve.supply_balance.user_state,
     user)
    };
    if (table::contains(&reserve.borrow_balance.user_state, user)) {
        borrow_balance = *table::borrow(&reserve.borrow_balance.user_state,
     user)
    };

    (supply_balance, borrow_balance)
}
```

The same applies to the `get_user_state()` function. It invokes `get_reserves_count()` and `get_user_balance()`, but since these two methods only retrieve information about the user's reserves, there's no need in passing storage as a mutable reference.

Impact In future updates of the protocol, updates to above-mentioned functions may accidentally modify the underlying struct which can lead to undesirable effects on the protocol.

Recommendation Whenever possible, prefer to use immutable references over mutable references. For functions such as `get_supply_cap_ceiling()`, `get_user_state()`, and `get_user_balance()` that do not need a mutable reference, change the implementation to take in an immutable reference.

4.1.21 V-NAVI-VUL-021: Duplicate Function

Severity	Warning	**Commit**	945878c
Type	Maintainability	**Status**	Acknowledged
File(s)	`logic.move`		
Location(s)	dynamic_liquidation_threshold() calculate_avg_threshold()		
Confirmed Fix At	N/A		

The Navi protocol includes two functions in the logic module, `dynamic_liquidation_threshold` and `calculate_avg_threshold`. Both perform the same computation—calculating the average liquidation threshold for a user's collateral assets—using nearly identical logic, with the only difference being variable names.

Impact Should the developers decide to modify the threshold logic, both locations will need to be changed.

Recommendation Remove one of the functions and change all callsites to invoke the remaining function.

4.1.22 V-NAVI-VUL-022: Unused events

Severity	Warning	**Commit**	945878c
Type	Maintainability	**Status**	Acknowledged
File(s)	`pool.movestorage.move`		
Location(s)	Multiple		
Confirmed Fix At	N/A		

We have identified unused events in several locations such as the ones shown below.

```
struct PoolBalanceRegister has copy, drop {
        sender: address,
    amount: u64,
    new_amount: u64,
    pool: String,
}
```

Snippet 4.27: Snippet showing the `PoolBalanceRegister` event in `pool.move`

```
struct StorageConfiguratorSetting has copy, drop {
        sender: address,
        configurator: address,
        value: bool,
}
```

Snippet 4.28: Snippet showing the `StorageConfiguratorSetting` event in `storage.move`

Impact It may be the case that the developers intended to emit an event, but forgot. If another monitoring service relies on these events to determine the health of the protocol, certain irregularities may go unnoticed.

Recommendation Consider emitting these events if that is what is intended.

4.1.23 V-NAVI-VUL-023: Account utilities may result in undesired behaviors

Severity	Info	Commit	945878c
Type	Logic Error	Status	Acknowledged
File(s)	`account.move`		
Location(s)	create_child_account_cap, delete_account_cap		
Confirmed Fix At	N/A		

An `AccountCap` can be used to represent the ownership of a particular account, allowing anyone who owns the capability to interact with the protocol. The capability also has several utility functions such as `create_child_account_cap` and `delete_account_cap` shown below. The `create_child_account_cap` allows for the creation of a new capability that is able to access the funds of the "parent" capability. Additionally, the `delete_account_cap` allows the deletion of the capability by the owner. While neither of these functions are currently used, we would recommend removing them as they could produce some odd behaviors.

```
1  public(friend) fun create_child_account_cap(parent_account_cap: &AccountCap, ctx: &
       mut TxContext): AccountCap {
2      let owner = parent_account_cap.owner;
3      assert!(object::uid_to_address(&parent_account_cap.id) == owner, error::
       required_parent_account_cap());
4
5      AccountCap {
6          id: object::new(ctx),
7          owner: owner
8      }
9  }
10
11 public(friend) fun delete_account_cap(cap: AccountCap) {
12     let AccountCap { id, owner: _} = cap;
13     object::delete(id)
14 }
```

Snippet 4.29: Snippet from `account.move`

Impact We would not recommend allowing for the deletion of a capability as it could cause funds to become locked within the protocol if the owner of the capability has not removed all of their collateral. While some protections could be added to prevent this from occurring, it seems as though there would be little gain to providing the API.

We also would not recommend allowing the creation of a child capability as the child has almost the same abilities as the parent. This would allow the child to manage all of the parent's funds, including the ability to take out new loans against the parent's collateral. As it currently stands if the child account were compromised, the parent could not delete it. Additionally, if the parent deleted their account, all children would still have the ability to trade in their name which may not be expected.

Recommendation Since some of these abilities could be created externally (e.g. transferring the AccountCap to a dead address or using a multi-sig to control an AccountCap) we would

recommend not adding this functionality. If there is demand for these additional utilities and they are added, ensure proper validation is performed to avoid the above scenarios.