Navi Protocol Core

Audit

Presented by:

OtterSec contact@osec.io

Ajay Shankar d1r3wolf@osec.io

Robert Chen r@osec.io

Contents

01 Executive Summary **2**

Overview . . . 2

Key Findings . . . 2

02 Scope **3**

03 Findings **4**

04 Vulnerabilities **5**

OS-NVI-ADV-00 [crit] | Loss Of Funds In Lending . . . 6

OS-NVI-ADV-01 [crit] | Flawed Validations Lead To Inaccuracies . . . 8

OS-NVI-ADV-02 [high] | Erroneous Calculation Leads To Unfair Liquidation . . . 9

OS-NVI-ADV-03 [med] | Health Check Performed On Outdated State . . . 11

OS-NVI-ADV-04 [low] | Inability To Withdraw Treasury Amount . . . 12

05 General Findings **13**

OS-NVI-SUG-00 | Inappropriate Transfer Of Zero-Value Coins . . . 14

OS-NVI-SUG-01 | Optimizing Computation In Storage . . . 15

OS-NVI-SUG-02 | Presence Of Unused Fields . . . 16

OS-NVI-SUG-03 | Useless Withdraw Operation In Lending . . . 17

OS-NVI-SUG-04 | Removing Redundant Calculations . . . 18

OS-NVI-SUG-05 | Useless Storing Of Boolean . . . 19

OS-NVI-SUG-06 | Unused Reserves List . . . 20

OS-NVI-SUG-07 | Lack Of Functionalities In Storage . . . 21

OS-NVI-SUG-08 | Optimizing Update Function . . . 22

OS-NVI-SUG-09 | Unallocated Interest Remains Locked In Pool . . . 23

OS-NVI-SUG-10 | Rounding Error Leads To Inconsistency In Pool . . . 24

Appendices

A Vulnerability Rating Scale **25**

B Procedure **26**

01 | Executive Summary

Overview

Navi Protocol engaged OtterSec to perform an assessment of the `protocol-core` program. This assessment was conducted between May 22nd and June 7th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 16 findings in total.

In particular, our assessment revealed a concern related to the lack of essential checks, which could potentially lead to fund loss (OS-NVI-ADV-00), and an incorrect calculation was detected, enabling malicious actors to exploit improper liquidation scenarios for profit (OS-NVI-ADV-02). Additionally, we have identified flawed validations that could result in inaccuracies within the pool (OS-NVI-ADV-01).

Furthermore, our analysis brought to light a concern related to the health checks for liquidations, as these checks are conducted using an outdated collateral asset state (OS-NVI-ADV-03). Additionally, we observed an issue related to the inability to withdraw the collected treasury amount from the treasury balance due to the lack of a withdraw functionality (OS-NVI-ADV-04).

We also made recommendations around gas and computation optimizations (OS-NVI-SUG-01), OS-NVI-SUG-03), and pointed out the presence of unused fields in specific objects (OS-NVI-SUG-02), and further recommended the elimination of the possibility of storing false values in the table (OS-NVI-SUG-05). In addition, we provided a solution to address the lack of getter functions for the storage object, facilitating the retrieval of these vital states (OS-NVI-SUG-07).

02 | Scope

The source code was delivered to us in a git repository at github.com/naviprotocol/protocol-core. This audit was performed against commit 838650c.

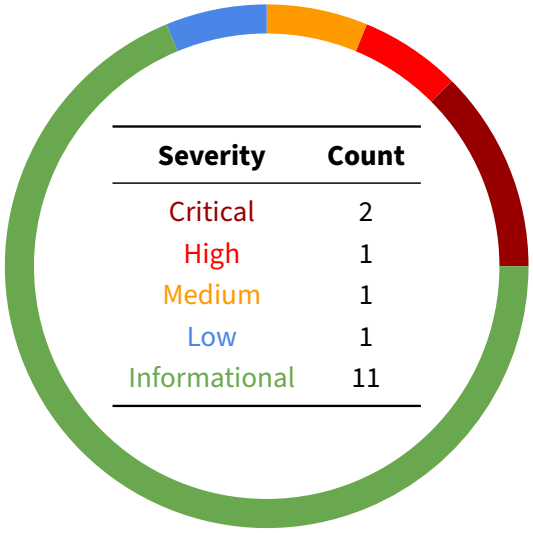
A brief description of the programs is as follows.

Name	Description
protocol-core	A liquidity protocol that allows users to provide liquidity and borrow funds within the Sui ecosystem. Navi offers users to earn passive income as liquidity providers or obtain loans as borrowers. The protocol introduces innovative features like Automatic Leverage Vaults and Isolation Mode, enabling users to leverage their assets and access novel trading opportunities with minimized risks.

03 | Findings

Overall, we reported 16 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 help mitigate future vulnerabilities.



Severity	Count
Critical	2
High	1
Medium	1
Low	1
Informational	11

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we 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-NVI-ADV-00	Critical	Resolved	Absence of a check for the `CoinType` leads to a loss of funds.
OS-NVI-ADV-01	Critical	Resolved	Validator functions use elements with different scales, leading to a loss of accuracy.
OS-NVI-ADV-02	High	Resolved	The calculations within `calculate_max_liquidation` are currently inaccurate.
OS-NVI-ADV-03	Medium	Resolved	`is_health` is invoked on the outdated state in the execute function.
OS-NVI-ADV-04	Low	Resolved	Missing withdrawal functionality leads to the program locking the user's funds.

OS-NVI-ADV-00 [crit] | Loss Of Funds In Lending

Description

In all functions in `lending.move`, the `CoinType` type tag lacks verification against the coin type set in `ReserveData` for the corresponding `asset`. This absence may lead to the loss of funds when calling the functions.

Consider the `withdraw` function in `lending.move`:

sources/lending.move RUST

```
    public entry fun withdraw<CoinType>(
        clock: &Clock,
        oracle: &PriceOracle,
        storage: &mut Storage,
        pool: &mut Pool<CoinType>,
        asset: u8,
        amount: u64, // e.g. 100USDT(1000000) -> 100 * 1e6
        to: address,
        ctx: &mut TxContext
    ) {
        when_not_paused(storage);
        let sender = tx_context::sender(ctx);

        // e.g. 100000000 -> 100000000000
        let normal_withdraw_amount = pool::normal_amount(pool, amount);
        validation::validate_withdraw(storage, asset, normal_withdraw_amount);

        let actual_amount = logic::execute_withdraw(clock, oracle, storage, asset,
    ↪   sender, (normal_withdraw_amount as u256));

        let normal_actual_amount = pool::unnormal_amount(pool, actual_amount);
        pool::withdraw(pool, normal_actual_amount, to, ctx);
        emit(WithdrawEvent {
            reserve: asset,
            sender: tx_context::sender(ctx),
            to: to,
            amount: normal_withdraw_amount
        })
    }
```

The function allows an attacker to supply an `asset` parameter indicating USD and a `pool` with Bitcoin as `CoinType`. By specifying an amount of `10^8`, the function deducts 100 USD from the USD `ReserveData` in storage. However, due to the attacker specifying a Bitcoin `pool`, they will receive `10^8` Bitcoin units, equivalent to one Bitcoin when accounting for decimals. The attacker acquires one Bitcoin for only 100 USD.

Note that within the module, similar issues may occur in the other functions as well.

Remediation

Implement input validation to ensure that the provided `asset` and `pool` parameters are valid against the `CoinType` supplied.

Patch

Resolved in a7ea49c by incorporating validation for the `CoinType` against the provided `asset`.

OS-NVI-ADV-01 [crit] | Flawed Validations Lead To Inaccuracies

Description

In `validator.move`, a set of functions validate various actions related to lending operations. These functions validate specific conditions before allowing the execution of the corresponding tasks. However, some of these functions use scaled balances `supply` and `borrow` in conjunction with unscaled amounts, which may lead to inaccuracies. In particular:

1. In `validate_deposit`, the calculation for `estimate_supply` combines a scaled supply balance with an unscaled amount to compare against the `supply_cap_ceiling`.
2. In `validate_withdraw`, the condition `supply_balance >= borrow_balance + amount` compares two values scaled using different indexes; this may result in the condition failing if the supply index is relatively small compared to the borrowing index.
3. In `validate_borrow`, the calculation for `current_borrow_ratio` involves scaled borrow and supply balances multiplied by different indexes.

Remediation

Insert the validation functions in `logic.move` and place them after the `update_state` call; this ensures that the indexes become updated before performing calculations, and the `supply` and `borrow` balances become unscaled by multiplying them with their respective indexes.

Patch

Fixed in e4557a5 and d8ec7e7 by relocating validation functions to `logic.move`, and making adjustments to utilize unscaled amounts for comparisons during checks.

OS-NVI-ADV-02 [high] | Erroneous Calculation Leads To Unfair Liquidation

Description

In `logic.move`, `calculate_max_liquidation` retrieves `max_liquidable_collateral` and `max_liquidable_debt`. During calculations done within the function, some inaccuracies undermine the accuracy of these values. The calculations create exploitable opportunities for attackers, enabling them to profit from improper liquidation scenarios.

sources/logic.move RUST

```
    public fun calculate_max_liquidation(
        storage: &mut Storage,
        oracle: &PriceOracle,
        liquidated_user: address,
        collateral_asset: u8,
        loan_asset: u8
    ): (u256, u256) {
        let (liquidation_ratio, liquidation_bonus, _) =
    ↪   storage::get_liquidation_factors(storage, collateral_asset);

        let _collateral_value = user_collateral_value(oracle, storage,
    ↪   collateral_asset, liquidated_user); // 100 u ETH
        let _loan_value = user_loan_value(oracle, storage, loan_asset,
    ↪   liquidated_user); // 1000u - 100u

        let max_liquidable_collateral_value = ray_math::ray_mul(_collateral_value,
    ↪   liquidation_ratio);
        let max_liquidable_debt_value = _loan_value;
        if (_loan_value > _collateral_value) {
            max_liquidable_collateral_value =
    ↪   ray_math::ray_mul(max_liquidable_collateral_value, (ray_math::ray() +
    ↪   liquidation_bonus));
            max_liquidable_debt_value = _collateral_value
        } else {
            max_liquidable_debt_value = ray_math::ray_mul(max_liquidable_debt_value,
    ↪   (ray_math::ray() + liquidation_bonus))
        };

        let max_liquidable_collateral = calculator::calculate_amount(oracle,
    ↪   max_liquidable_collateral_value, collateral_asset);
        let max_liquidable_debt = calculator::calculate_amount(oracle,
    ↪   max_liquidable_debt_value, loan_asset);

        (max_liquidable_collateral, max_liquidable_debt)
    }
```

Proof of Concept

Consider the following scenario:

1. A victim has health less than one and two loan assets, one large and one small.
2. An attacker liquidates the small loan asset using the large collateral asset.
3. The attacker obtains the collateral for a minuscule loan repayment.
4. Note that `max_liquidable_collateral_value` is not appropriately reduced based on the `max_liquidable_debt_value` when the collateral exceeds the loan amount.

Remediation

Modify the code as follows, since `max_liquidable_collateral_value` should always be equal to `max_liquidable_debt_value * (1 + bonus)`:

sources/logic.move DIFF

```
@@ -487,11 +487,10 @@ module lending_core::logic {

         let max_liquidable_collateral_value = ray_math::ray_mul(_collateral_value,
    ↪  liquidation_ratio);
         let max_liquidable_debt_value = _loan_value;
-        if (_loan_value > _collateral_value) {
-            max_liquidable_collateral_value =
    ↪  ray_math::ray_mul(max_liquidable_collateral_value, (ray_math::ray() +
    ↪  liquidation_bonus));
-            max_liquidable_debt_value = _collateral_value
+        if (max_liquidable_debt_value > max_liquidable_collateral_value) {
+            max_liquidable_debt_value =
    ↪  ray_math::ray_div(max_liquidable_collateral_value, (ray_math::ray() +
    ↪  liquidation_bonus))
         } else {
-            max_liquidable_debt_value =
    ↪  ray_math::ray_mul(max_liquidable_debt_value, (ray_math::ray() +
    ↪  liquidation_bonus))
+            max_liquidable_collateral_value =
    ↪  ray_math::ray_mul(max_liquidable_debt_value, (ray_math::ray() +
    ↪  liquidation_bonus))
         };

         let max_liquidable_collateral = calculator::calculate_amount(oracle,
    ↪  max_liquidable_collateral_value, collateral_asset)
```

Patch

This modification has been implemented in 33fbb0f by deprecating `calculate_max_liquidation` and performing the calculation of liquidation using the updated `calculate_liquidation`.

OS-NVI-ADV-03 [med] | Health Check Performed On Outdated State

Description

The `is_health` assert in `execute_withdraw` and `execute_borrow` in `logic.move` depends on the user's collateral and loan balances. However, these balances are not updated with `update_state` during health validation, potentially causing inaccuracies. This issue is particularly impactful during the liquidation process, as outdated collateral asset states may lead to exclusion from liquidation.

sources/logic.move RUST

```
    public(friend) fun execute_borrow(clock: &Clock, oracle: &PriceOracle, storage: &mut
    ↪  Storage, asset: u8, user: address, amount: u256) {
        ////////////////////////////////////////////////////////////////
        // Update borrow_index, supply_index, last_timestamp, treasury  //
        ////////////////////////////////////////////////////////////////
        update_state(clock, storage, asset);


        ////////////////////////////////////////////////////////////////////////
        // Convert balances to actual balances using the latest exchange rates //
        ////////////////////////////////////////////////////////////////////////
        increase_borrow_balance(storage, asset, user, amount);

        ////////////////////////////////////////////////////
        // Add the asset to the user's list of loan assets //
        ////////////////////////////////////////////////////
        if (!is_loan(storage, asset, user)) {
            storage::update_user_loans(storage, asset, user)
        };

        ///////////////////////////////////
        // Checking user health factors //
        ///////////////////////////////////
        assert!(is_health(oracle, storage, user), LOGIC_USER_UN_HEALTH);

        update_interest_rate(storage, asset);
    }
```

Remediation

Update all asset states invoking `update_state` before performing the health check.

Patch

Addressed in c3dbdd3 by updating all asset states prior to conducting the health check.

OS-NVI-ADV-04 [low] | Inability To Withdraw Treasury Amount

Description

In `storage.move`, `increase_treasury_balance` and `update_state` collects the treasury amount by adding it to the `treasury_balance` of the coin's `ReserveData` in `Storage`. However, no functionality exists to collect this amount using the `withdraw_treasury`. As a result, this amount remains locked in the pool.

sources/storage.move RUST

```
public(friend) fun increase_treasury_balance(storage: &mut Storage, asset: u8,
↪  amount: u256) {
    let reserve = table::borrow_mut(&mut storage.reserves, asset);
    reserve.treasury_balance = reserve.treasury_balance + amount;
}
[...]

public(friend) fun update_state(
    storage: &mut Storage,
    asset: u8,
    new_borrow_index: u256,
    new_supply_index: u256,
    last_update_timestamp: u64,
    scaled_treasury_amount: u256
) {
    let reserve = table::borrow_mut(&mut storage.reserves, asset);

    reserve.current_borrow_index = new_borrow_index;
    reserve.current_supply_index = new_supply_index;
    reserve.last_update_timestamp = last_update_timestamp;

    reserve.treasury_balance = reserve.treasury_balance +
↪  scaled_treasury_amount;
}
```

Remediation

Create the necessary functionality to withdraw the treasury balance from the pool.

Patch

Fixed in commit 54dbd6e by implementing `withdraw_treasury` to allow for the withdrawal of the treasury balance.

05 | General Findings

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

ID	Description
OS-NVI-SUG-00	Checks to prevent the transfer of zero-value coins are absent.
OS-NVI-SUG-01	Improvement of computational efficiency in `reserve_validation` by optimizing variable initialization.
OS-NVI-SUG-02	Unused fields in objects are present throughout the application.
OS-NVI-SUG-03	Optimize `repay` and `liquidation_call` by removing a useless `withdraw` operation.
OS-NVI-SUG-04	Improve efficiency in `update_state` by eliminating redundant time difference calculations.
OS-NVI-SUG-05	Remove the possibility of storing `false` value in `table`.
OS-NVI-SUG-06	`Storage` contains useless field.
OS-NVI-SUG-07	Lack of a getter function for the `Storage` object.
OS-NVI-SUG-08	Improve the efficiency of `Storage` by enhancing the search process for existing users.
OS-NVI-SUG-09	`update_state` does not allocate the remaining interest, leading to the locking of funds.
OS-NVI-SUG-10	Rounding error during the decimal conversion creates an inconsistency in `Pool`.

OS-NVI-SUG-00 | Inappropriate Transfer Of Zero-Value Coins

Description

In `utils.move`, `split_coin` is used for splitting a specified `amount` from a `Coin` object. The function returns the extracted part while the remaining part transfers back to the user. The function fails to handle the scenario where the remaining coins after the split equal zero, leading to an unnecessary transfer of zero-value coins.

utils/utils.move RUST

```
public fun split_coin<CoinType>(split_coin: Coin<CoinType>, amount: u64, ctx:
↪  &mut TxContext): Coin<CoinType> {
    // get total value of coin
    let coin_value = coin::value(&split_coin);
    assert!(amount > 0, UTILS_AMOUNT_ZERO);
    assert!(coin_value >= amount, UTILS_INSUFFICIENT_FUNDS);

    // Split the specified amount of coin
    let split_ = coin::split(&mut split_coin, amount, ctx);

    // Transferring the unsegmented portion back to the user
    transfer::public_transfer(split_coin, tx_context::sender(ctx));
    split_
}
```

Remediation

Include a check to verify that after a coin splits, the remaining amount is above zero. If the remaining amount is zero, `destroy_zero` should be invoked instead of performing an unnecessary transfer of zero-value coins.

OS-NVI-SUG-01 | Optimizing Computation In Storage

Description

In `storage.move`, `reserve_validation` prevents the duplication of `reserve` by iterating through the `reserves` stored in the `Storage` object. Each reserve's `coin_type` is compared with the fixed `CoinType` name. To optimize computation within the function, the value of the `CoinType` name may initialize outside the loop as it remains constant throughout the iteration.

source/storage.move RUST

```
    public fun reserve_validation<CoinType>(storage: &Storage) {
        let count = storage.reserves_count;
        let i = 0;

        while (i < count) {
            let reserve = table::borrow(&storage.reserves, i);
            let name = type_name::into_string(type_name::get<CoinType>());
            assert!(reserve.coin_type != name, STORAGE_DUPLICATE_RESERVE);
            i = i + 1;
        }
    }
```

Remediation

Initialize `name` outside the loop.

OS-NVI-SUG-02 | Presence Of Unused Fields

Description

In both `storage.move` and `pool.move`, some fields within objects remain unused. The `ReserveData` object never utilizes the `is_isolated` field, while in the `PoolAdminCap` object, the `admins` and `owners` fields are similarly unused. These redundant fields serve no purpose within the current implementation, and removing them would simplify and improve the clarity of the code.

Remediation

Remove the following fields:

1. `is_isolated` from `ReserveData`.
2. `owners` from `PoolAdminCap`.
3. `admins` from `PoolAdminCap`.

OS-NVI-SUG-03 | Useless Withdraw Operation In Lending

Description

In `lending.move`, within `repay` and `liquidation_call`, the invocation of `withdraw` may be avoided. Subtracting the `excess_amount` before the `deposit` operation when calculating the amount deposited avoids the invocation. This optimization eliminates the need for a separate `withdraw` call, resulting in significant gas savings and improved efficiency.

source/lending.move RUST

```
public entry fun repay<CoinType>(
    clock: &Clock,
    oracle: &PriceOracle,
    storage: &mut Storage,
    pool: &mut Pool<CoinType>,
    asset: u8,
    repay_coin: Coin<CoinType>,
    amount: u64, // e.g. 100USDT(1000000) -> 100 * 1e6
    ctx: &mut TxContext
) {
    when_not_paused(storage);
    let sender = tx_context::sender(ctx);

    let split_coin = utils::split_coin(repay_coin, amount, ctx);
    let repay_value = coin::value(&split_coin);
    pool::deposit(pool, split_coin, ctx);

    let normal_repay_value = pool::normal_amount(pool, repay_value);// e.g.
↪  100USDT(100 * 1e6) -> 100 * 1e9

    validation::validate_repay(storage, asset, amount);
    let excess_amount = logic::execute_repay(clock, oracle, storage, asset,
↪  sender, (normal_repay_value as u256));
    if (excess_amount > 0) {
        pool::withdraw(pool, (excess_amount as u64), sender, ctx);
    };

    emit(RepayEvent {
        reserve: asset,
        sender: tx_context::sender(ctx),
        amount: repay_value
    })
}
```

Remediation

Subtract the `excess_amount` before the `deposit` operation and remove the `withdraw` invocation.

OS-NVI-SUG-04 | Removing Redundant Calculations

Description

In `logic.move`, during the execution of `update_state`, the time difference is calculated separately within the `calculate_compounded_interest` and `calculate_linear_interest`. Remove this redundancy by calculating the time difference once in `update_state` and then passing it as a parameter to both interest calculation functions. This optimization enhances code efficiency and minimizes redundant computations.

source/calculator.move RUST

```
    public fun calculate_compounded_interest(
        current_timestamp: u64,
        last_update_timestamp: u64,
        rate: u256
    ): u256 {
        // e.g. get the time difference of the last update --> (1685029315718 -
    ↪  1685029255718) / 1000 == 60s
        let timestamp_difference = (current_timestamp - last_update_timestamp as
    ↪  u256) / 1000;
        [...]
    }

    public fun calculate_linear_interest(
        current_timestamp: u64,
        last_update_timestamp: u64,
        rate: u256
    ): u256 {
        // e.g. get the time difference of the last update --> (1685029315718 -
    ↪  1685029255718) / 1000 == 60s
        let timestamp_difference = (current_timestamp - last_update_timestamp as
    ↪  u256) / 1000;
        [...]
```

Remediation

Calculate the time difference once at the beginning of `update_state` and store it in a variable. Then, pass this time difference as a parameter to the following functions:

1. `calculate_compounded_interest`
2. `calculate_linear_interest`

OS-NVI-SUG-05 | Useless Storing Of Boolean

Description

During the execution of `pool.move` and `storage.move`, when performing the `set_owner` and `set_admin` operations for the former and `set_configurator` for the latter, the `val` variable provided as a parameter helps populate the table. However, rather than storing the boolean value as `false`, it is more advantageous to remove it from the table altogether, as the tables represent the list of `owners`, `admins`, and `configurators`.

source/pool.move RUST

```
    public fun set_owner(pool_admin_cap: &mut PoolAdminCap, owner: u256, val: bool,
    ↪  ctx: &mut TxContext) {
        assert!(pool_admin_cap.creator == tx_context::sender(ctx),
    ↪  POOL_CALLER_NOT_OWNER);

        if (!table::contains(&pool_admin_cap.owners, owner)) {
            table::add(&mut pool_admin_cap.owners, owner, val)
        } else {
            let v = table::borrow_mut(&mut pool_admin_cap.owners, owner);
            *v = val
        };

        emit(PoolOwnerSetting {
            sender: tx_context::sender(ctx),
            owner: owner, value: val
        })
    }
```

Remediation

Remove the element from the table instead of storing it as false.

OS-NVI-SUG-06 | Unused Reserves List

Description

In `storage.move`, the `Storage` object contains `reserves_list` as a field representing a vector of reserve indexes. However, this field is removable since there are no delegations of reserve in the storage, and the vector may be derived using `reserves_count`.

source/storage.move RUST

```
struct Storage has key, store {
    // Use as index
    id: UID,
    // The create of the pool
    owner: address,
    // Whether the pool is paused
    paused: bool,
    // Reserve list. index -> reserve data
    reserves: Table<u8, ReserveData>,
    // List of reserve indexes
    reserves_list: vector<u8>,
    // Administrator of the protocol
    configurator: Table<address, bool>,
    // Total reserves count
    reserves_count: u8,
    users: vector<address>,
    user_info: Table<address, UserInfo>
}
```

Remediation

Remove the `reserves_list` field from `Storage` object.

OS-NVI-SUG-07 | Lack Of Functionalities In Storage

Description

In `storage.move`, there are two important fields in `ReserveData`: `is_isolated` and `treasury_balance`. These variables hold crucial information about the state of the system. However, there are no getter functions provided to access them.

source/storage.move RUST

```
struct ReserveData has store {
    [...]
    // isolated
    is_isolated: bool,
    [...]
    // fee balance
    treasury_balance: u256,
    [...]
}
```

Remediation

Implement getter functions to retrieve `is_isolated` and `treasury_balance` fields from `ReserveData`.

OS-NVI-SUG-08 | Optimizing Update Function

Description

In `update_user_loans` and `update_user_collaterals` of `storage.move`, the use of `vector::contains(&storage.users, &user)` for inserting items without duplicates can be computationally expensive. As `vector::contains` iterates over the entire vector, and considering the potentially large number of users, this operation consumes significant gas.

To improve performance, leverage the `Storage.user_info` table to check if the user exists. Since `Storage.user_info`'s implementation is as of a table, it provides a more efficient solution.

source/storage.move RUST

```
    public(friend) fun update_user_loans(storage: &mut Storage, asset: u8, user: address) {
        [...]
        if (!vector::contains(&storage.users, &user)) {
            vector::push_back(&mut storage.users, user)
        }
    }

    public(friend) fun update_user_collaterals(storage: &mut Storage, asset: u8, user:
    ↪  address) {
        [...]
        if (!vector::contains(&storage.users, &user)) {
            vector::push_back(&mut storage.users, user)
        }
    }
```

Remediation

Replace the current invocation of `vector::contains` over the `storage.users` vector with a direct lookup in the `Storage.user_info` table to improve performance and efficiency.

OS-NVI-SUG-09 | Unallocated Interest Remains Locked In Pool

Description

In `logic.move`, `update_state` is used to update the value of `borrow_index`, `supply_index`, `last_timestamp` and `treasury` in `Storage` object. During calculations, the compound interest is subtracted from the borrower's balances, while the linear interest is added to the supplier's balances. However, no explicit allocation for the remaining amount exists, which would represent the difference between compound and linear interests. As a result, this remaining amount becomes locked in the pool.

Remediation

Let protocol collect the remaining amount as a fee instead of leaving it locked in the pool.

OS-NVI-SUG-10 | Rounding Error Leads To Inconsistency In Pool

Description

In `pool.move`, `normal_amount` is utilized to convert the current decimal of `CoinType` to the `target_decimal` value employed in `Pool`. However, if the current decimal value is greater than nine, the converted amount is rounded down. This imprecision leads to inconsistency during the withdrawal operation by withdrawing the original value.

source/pool.move RUST

```
public fun convert_amount(amount: u64, cur_decimal: u8, target_decimal: u8):
↪  u64 {
    while (cur_decimal != target_decimal) {
        if (cur_decimal < target_decimal) {
            amount = amount * 10;
            cur_decimal = cur_decimal + 1;
        }else {
            amount = amount / 10;
            cur_decimal = cur_decimal - 1;
        };
    };
    amount
}

/// Normal coin amount in dola protocol
public fun normal_amount<CoinType>(pool: &Pool<CoinType>, amount: u64): u64 {
    let cur_decimal = get_coin_decimal<CoinType>(pool);
    let target_decimal = 9;
    convert_amount(amount, cur_decimal, target_decimal)
}
```

Consider the following scenario:

1. An attacker invokes `withdraw` with `amount` of ETH set to 1000000000999999999.
2. The amount rounds down to 1000000000 during `normal_amount`.
3. The attacker receives 1000000000999999999 ETH from the pool.

Remediation

Include a check to restrict users from including values beyond the ninth decimal point.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions
Informational	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 implementation of the program 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 sum, 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 get a comprehensive understanding of 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 the other 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.