Solido Money

Audit Report

contact@bitslab.xyz

https://twitter.com/movebit_

Tue Apr 08 2025

Solido Money Audit Report

1 Executive Summary

1.1 Project Information

Description	Solido is a CDP which enables users to borrow funds using multiple forms of yield-generating collateral, including $SUPRA and yield-amplified or auto-compounding yield tokens
Type	DeFi
Auditors	MoveBit
Timeline	Tue Mar 11 2025 - Tue Apr 08 2025
Languages	Move
Platform	Others
Methods	Architecture Review, Unit Testing, Manual Review
Source Code	https://github.com/solido-money/cdp-contracts
Commits	ab42ba766ac4634eb6f1efe3dbc3f1624071bec4 b473a56ad6631dc8b47d97778b3f809dd309c853 587b9192d2aaba814ad9eba2a9820d58fa2c65ea

1.2 Files in Scope

The following are the SHA1 hashes of the original reviewed files.

ID	File	SHA-1 Hash
MOV	Move.toml	51eb6fbe1485c0368fa84eb2fcd1d7da17d00eaf
EVE	sources/contracts/events.move	79783a533fd8c2a7e1a21d69d4bf8234471a5226
CMU	sources/contracts/cdp_multi.move	c3b9f82fe13e2b160e723ec076aaefe0de724488
POS	sources/contracts/positions.move	b4d1a55074ba2298a83a5c08e08cb2f042337dbe
POR	sources/contracts/price_oracle.move	fced12c19b560b9529842eadefd8d2d58716fb5f
CON	sources/contracts/config.move	c2264e4b94e8d902cb0aa25f7ecb7816ebb08091

1.3 Issue Statistic

Item	Count	Fixed	Acknowledged
Total	20	19	1
Informational	1	1	0
Minor	3	3	0
Medium	8	7	1
Major	7	7	0
Critical	1	1	0

1.4 MoveBit Audit Breakdown

MoveBit aims to assess repositories for security-related issues, code quality, and compliance with specifications and best practices. Possible issues our team looked for included (but are not limited to):

- Transaction-ordering dependence
- Timestamp dependence
- Integer overflow/underflow by bit operations
- Number of rounding errors
- Denial of service / logical oversights
- Access control
- Centralization of power
- Business logic contradicting the specification
- Code clones, functionality duplication
- Gas usage
- Arbitrary token minting
- Unchecked CALL Return Values
- The flow of capability
- Witness Type

1.5 Methodology

The security team adopted the **"Testing and Automated Analysis"**, **"Code Review"** and **"Formal Verification"** strategy to perform a complete security test on the code in a way that is closest to the real attack. The main entrance and scope of security testing are stated in the conventions in the "Audit Objective", which can expand to contexts beyond the scope according to the actual testing needs. The main types of this security audit include:

(1) Testing and Automated Analysis

Items to check: state consistency / failure rollback / unit testing / value overflows / parameter verification / unhandled errors / boundary checking / coding specifications.

(2) Code Review

The code scope is illustrated in section 1.2.

(3) Formal Verification(Optional)

Perform formal verification for key functions with the Move Prover.

(4) Audit Process

- Carry out relevant security tests on the testnet or the mainnet;
- If there are any questions during the audit process, communicate with the code owner in time. The code owners should actively cooperate (this might include providing the latest stable source code, relevant deployment scripts or methods, transaction signature scripts, exchange docking schemes, etc.);
- The necessary information during the audit process will be well documented for both the audit team and the code owner in a timely manner.

2 Summary

This report has been commissioned by Solido to identify any potential issues and vulnerabilities in the source code of the Solido Money smart contract, as well as any contract dependencies that were not part of an officially recognized library. In this audit, we have utilized various techniques, including manual code review and static analysis, to identify potential vulnerabilities and security issues.

During the audit, we identified 20 issues of varying severity, listed below.

ID	Title	Severity	Status
CMU-1	Positions is Flagged as Liquidatable immediately after Creation	Critical	Fixed
CMU-2	Lack of Slippage Protection in `redeem()` Function	Major	Fixed
CMU-3	Bad Debt in `liquidate()` Due to Excessive Penalty Payouts	Major	Fixed
CMU-4	'total_collateral' Miscalculated	Major	Fixed
CMU-5	Missing Access Control for `initialize()`	Major	Fixed
CMU-6	No Partial Liquidation Prevents Whale Liquidation	Major	Fixed
CMU-7	Unable to Repay Fully	Major	Fixed
CMU-8	The Repayer is Charged a Liquidation Fee	Major	Fixed
CMU-9	Partial Clearing of Batches with Loss of Precision	Medium	Fixed

EVE-1	Set the Event Functions to `friend`	Minor	Fixed
POR-1	The Supra Oracle could Return a Stale Price	Medium	Fixed
CMU-10	User Lacks Buffer Space	Medium	Acknowledged
CMU-11	Users Can Self-liquidate	Medium	Fixed
CMU-12	`set_price` not Marked as `test-only`	Medium	Fixed
CMU-13	The User's Position may not be Liquidated	Medium	Fixed
CMU-14	The Verification of Total Distribution is Vulnerable to Precision Loss	Medium	Fixed
CMU-15	Missing a bad Debt Position Process	Medium	Fixed
CMU-16	Missing Fee Validation in `redeem()` Function	Minor	Fixed
CMU-17	Lack of Event	Minor	Fixed
CMU-18	`supra_oracle_storage` is Not Available	Informational	Fixed

3 Participant Process

Here are the relevant actors with their respective abilities within the Solido Money Smart Contract :

Admin

- `initialize` : Initializes core contract components.
- `add_collateral` : Adds new collateral type with parameters.
- `set_config` : Updates collateral configuration parameters.
- `set_oracle_id` : Updates oracle ID for a collateral type.
- `set_operation_status` : Controls operational flags for collateral.
- `set_fee_collector` : Updates fee collector address.

User

- `register_circle_coin` : Registers CIRCLECoin for an account.
- `register_collateral_coin` : Registers collateral coin type for an account.
- `register_as_redemption_provider` : Opt-in/out as redemption provider.
- `open_trove` : Creates new collateralized debt position.
- `deposit_or_mint` : Adjusts collateral/debt in existing position.
- `repay_or_withdraw` : Repays debt and/or withdraws collateral.
- `close_trove` : Closes debt position with full repayment.
- `liquidate` : Liquidates undercollateralized positions.
- `partial_liquidate` : Liquidators can perform partial liquidations.
- `redeem` : Redeems collateral for CIRCLE tokens.
- `redeem_multiple` : Batch redemption from multiple providers.

4 Findings

CMU-1 Positions is Flagged as Liquidatable immediately after Creation

Severity: Critical

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#407-416

Descriptions:

In the `liquidate()` function, the protocol calculates the `current_ratio` as follows:

```
// Calculate ICR
let price = price_oracle::get_price<CoinType>();
let collateral_value = fixed_point32::multiply_u64(collateral_amount, price);
let current_ratio = (collateral_value * 10000) / debt_amount;
```

The function then checks whether the position is liquidatable.

```
// Verify position is liquidatable (ICR < liquidation threshold)
    assert!(current_ratio < liquidation_threshold, events::err_cannot_liquidate());
```

However, during initialization, the debt token is assumed to have 8 decimals.

```
// Initialize CIRCLE coin
    let (burn_cap, freeze_cap, mint_cap) = coin::initialize<CIRCLECoin>(
        admin,
        string::utf8(b"SOLIDO STABLECOIN"),
        string::utf8(b"CIRCLE"),
        8,
        true
    );
```

On Aptos, not all tokens follow this standard—for example, **USDT (6 decimals)** and **USDC (6 decimals)**.

https://aptoscan.com/fungible-asset/0x357b0b74bc833e95a115ad22604854d6b0fca151cecd94111770e5d6ffc9dc2b

If such tokens (with fewer than 8 decimals) are used as collateral, the `current_ratio` calculation becomes incorrect. Specifically, the ratio will be **100 times smaller** than the actual value (since `debt_amount` is treated as having 8 decimals, while collateral may have fewer). As a result, positions may be flagged as liquidatable immediately after creation.

Suggestion:

It is recommended to ensure the collateral_value and debt_amount are compared at the same decimal precision.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-2 Lack of Slippage Protection in `redeem()` Function

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#653-763

Descriptions:

In the `redeem()` function, the protocol calculates the `collateral_to_redeem` based on the current oracle price, deducts fees, and transfers the collateral to the user.

```
 // Calculate collateral amounts
    let price = price_oracle::get_price<CoinType>();
    let collateral_to_redeem = fixed_point32::divide_u64(actual_redemption_amount,
price);
    let redemption_fee = (collateral_to_redeem * redemption_fee) / 10000;
    let user_gratuity_fee = (collateral_to_redeem * redemption_fee_gratuity) / 10000;
    let collateral_after_fee = collateral_to_redeem - redemption_fee - user_gratuity_fee;

    coin::transfer<CoinType>(&resource_signer, signer::address_of(redeemer),
collateral_after_fee);
```

However, **there is no slippage protection mechanism**, meaning users have no guarantee on the minimum amount of collateral they will receive.

Suggestion:

It is recommended to introduce a slippage tolerance parameter (e.g., min_collateral_out) in the redeem() function:

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-3 Bad Debt in `liquidate()` Due to Excessive Penalty Payouts

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#391-516

Descriptions:

In the liquidation process, the protocol enforces the following logic:

1. If `current_ratio <= 100%` (i.e., `current_ratio <= 10000` in basis points), the liquidator receives **all collateral** from the position.

```
// Transfer all collateral to liquidator since debt > collateral value
        coin::transfer<CoinType>(&resource_signer, signer::address_of(liquidator),
```

2. If `current_ratio > 100%`, the protocol imposes a liquidation penalty (`total_penalty_in_collateral`) on the borrower.
    - **Problem:** If `total_penalty_in_collateral > collateral_amount`, the protocol **pays the excess penalty (`total_penalty_in_collateral - collateral_amount`)** from its own reserves.

```
    let liquidator_reward_in_collateral=liquidator_penalty_in_collateral+
fixed_point32::divide_u64(debt_amount, price) ;
        let user_refund = if (total_penalty_in_collateral < collateral_amount) {
            collateral_amount - total_penalty_in_collateral
        } else {
            0
        };
```

- This creates a **bad debt** scenario where the protocol covers losses beyond the available collateral.

Suggestion:

It is recommended to ensure `total_penalty_in_collateral` never exceeds `collateral_amount`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-4 'total_collateral' Miscalculated

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#482-591

Descriptions:

The problem appears in the redeem function of cdp_multi. When a user redeemes part of the collateral, the user reward fee `(user_gratuity_fee)` is not directly rewarded to the user account through token, but is directly added to the user's total collateral. This amount of reward collateral is not added to the total amount of contract collateral in the logic of partial withdrawal, which may result in the user being unable to withdraw the reward collateral

```
public entry fun redeem<CoinType>(
        redeemer: &signer,
        provider_addr: address,
        circle_amount: u64
    ) acquires  TroveManager, SignerCapability, LRCollectorCapability {

        ......
        let collateral_to_redeem = fixed_point32::divide_u64(actual_redemption_amount, price); // 2000
        let redemption_fee = (collateral_to_redeem * redemption_fee) / 10000; // 2000 * 0.5% = 10
        let user_gratuity_fee = (collateral_to_redeem * redemption_fee_gratuity) / 10000; // 2000 * 1% = 20
        let collateral_after_fee = collateral_to_redeem - redemption_fee - user_gratuity_fee; // 2000 - 10 - 20 = 1970

        ......
        coin::transfer<CoinType>(&resource_signer, signer::address_of(redeemer), collateral_after_fee); // 1970 --> USER

        ......
        let is_closing = actual_redemption_amount == max_redeemable;
        let new_collateral = collateral_amount - collateral_to_redeem + user_gratuity_fee; // 4000 - 2000 + 20 = 2020
```

```
        let new_debt = debt_amount - actual_redemption_amount;

        if (is_closing) {
            ......
        } else {
            *total_collateral = *total_collateral - collateral_to_redeem; // One: 4000 - 2000 =
2000 Two: 2000 - 2019
        };
        *total_debt = *total_debt - actual_redemption_amount;
                ......
        if (new_collateral == 0 && new_debt == 0) {
            positions::remove_position<CoinType>(provider_addr);
        } else {
            positions::update_position<CoinType>(
                provider_addr,
                new_collateral,
                new_debt,
                timestamp::now_seconds()
            );
        };
    }
```

Examples are as follows:

The total amount of collateral of the user is 4000, and then the user makes a partial redemption for the first time, and the redemption amount is 2000

	User collateral balance	Total collateral of contract
Initial state	4000	4000
First	4000 - 2000 + 20 = 2020	2000
Second	2020 - 2019 + user_gratuity_fee	2000 <= 2019 (Bad debts occur)

```
let collateral_to_redeem = fixed_point32::divide_u64(actual_redemption_amount, price); // 2000
let redemption_fee = (collateral_to_redeem * redemption_fee) / 10000; // 2000 * 0.5% = 10
let user_gratuity_fee = (collateral_to_redeem * redemption_fee_gratuity) / 10000; // 2000 * 1% = 20
let collateral_after_fee = collateral_to_redeem - redemption_fee - user_gratuity_fee; // 2000 - 10 - 20 = 1970

...
coin::transfer<CoinType>(&resource_signer, signer::address_of(redeemer), collateral_after_fee); // 1970 --> USER
```

According to the code logic, the redemption fee `(redemption_fee)` and the user reward fee `(user_gratuity_fee)` are respectively 10 and 20. The user actually receives 1970 collateral quantity


```
let new_collateral = collateral_amount - collateral_to_redeem + user_gratuity_fee; // 4000 - 2000 + 20 = 2020
```

```
let new_debt = debt_amount - actual_redemption_amount;
```

The user's total amount of new collateral `(new_collateral)` plus the user reward fee is 2020, and the total amount of contract collateral `(total_collateral)` is 2000

```
*total_collateral = *total_collateral - collateral_to_redeem;  // 4000 - 2000 = 2000
```

When the user's second redemption amount is 2019, the user cannot receive the reward fee because the total collateral amount is 2000 less than the user's redemption amount 2019 `(total_collateral)` resulting in bad debts

Suggestion:

User reward fees are awarded through other separate reserves to ensure that the total amount of collateral is calculated correctly

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-5 Missing Access Control for `initialize()`

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#37-65

Descriptions:

The `initialize()` function is designed to set up the initial state and configuration of a module. However, an issue with this function is the lack of access control, which means that anyone can call this function and initialize the module.

```
public entry fun initialize(admin: &signer, fee_collector: address) {
        price_oracle::initialize(admin);
        config::initialize(admin, fee_collector);
        positions::initialize(admin);
        // Initialize CIRCLE coin
        let (burn_cap, freeze_cap, mint_cap) = coin::initialize<CIRCLECoin>(
            admin,
            string::utf8(b"SOLIDO STABLECOIN"),
            string::utf8(b"CIRCLE"),
            8,
            true
        );
        // Create resource account for CDP pool
        let (_resource_signer, signer_cap) = account::create_resource_account(admin,
b"cdp_pool");
        move_to(admin, SignerCapability { cap: signer_cap });
        // Create LR_COLLECTOR account and capability
        let (lr_collector_signer, lr_collector_cap) = account::create_resource_account(admin,
b"lr_collector");
        if (!coin::is_account_registered<CIRCLECoin>
(signer::address_of(&lr_collector_signer))) {
            coin::register<CIRCLECoin>(&lr_collector_signer);
        };
        move_to(admin, LRCollectorCapability { cap: lr_collector_cap });
        move_to(admin, TroveManager {
            circle_mint_cap: mint_cap,
            circle_burn_cap: burn_cap,
```

```
        circle_freeze_cap: freeze_cap,
        total_collateral: table::new(),
        total_debt: table::new(),
    });
}
```

This can lead to serious security vulnerabilities and unintended behavior in the system.

Suggestion:

It is recommended to include access control mechanisms.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-6 No Partial Liquidation Prevents Whale Liquidation

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#441

Descriptions:

The `liquidate()` function currently allows only full liquidation of a user's position.However, it introduces a significant issue: large positions (e.g., "whale" positions) may not be liquidated because most users lack the funds to fully liquidate them.

```
// Transfer and burn CIRCLE from liquidator
 let vault_manager = borrow_global_mut<TroveManager>(@cdp);
 let circle_coins = coin::withdraw<CIRCLECoin>(liquidator, debt_amount);
 coin::burn(circle_coins, &vault_manager.circle_burn_cap);
```

Suggestion:

It is recommended to implement partial liquidation.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-7 Unable to Repay Fully

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#324

Descriptions:

Assuming that the user calls `open_trove` to use 1000 circle_mint, the user will get 1000 through `coin::mint`, and the total_debt_amount will increase the fee.

At this time, the total_debt_amount to be repaid = 1000 + borrow_fee + liquidation_reserve

Then the user calls the `repay_or_withdraw` function to repay all, and the user holds `circle_mint < total_debt_amount`, and the user cannot fully repay it only through the current protocol.

```
// Handle CIRCLE repayment
        if (circle_repay > 0) {
            let vault_manager = borrow_global_mut<TroveManager>(@cdp);
            let total_debt = table::borrow_mut(&mut vault_manager.total_debt,
collateral_type);
            *total_debt = *total_debt - circle_repay;

            let circle_coins = coin::withdraw<CIRCLECoin>(user, circle_repay);
            coin::burn(circle_coins, &vault_manager.circle_burn_cap);

            events::emit_debt_repaid_event(user_addr, collateral_type, circle_repay,
timestamp::now_seconds());
        };
        events::emit_vessel_updated_event(user_addr, collateral_type, new_collateral,
new_debt, timestamp::now_seconds(),
block::get_current_block_height(),events::trove_action_adjust());

         // Update or remove position
        if (new_collateral == 0 && new_debt == 0) {
            positions::remove_position<CoinType>(user_addr);
        } else {
            positions::update_position<CoinType>(
                user_addr,
```

```
            new_collateral,
            new_debt,
            timestamp::now_seconds()
        );
    };
```

Suggestion:

Fix and sync `liquidation_reserve` and `borrow_fee` which need to burn.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-8 The Repayer is Charged a Liquidation Fee

Severity: Major

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#283

Descriptions:

In the `redeem` and `repay_or_withdraw` functions, the repayer is charged a liquidation fee. When a position is opened, a liquidation fee is charged as a debt. Usually, the liquidation incentive is only obtained by the liquidator. In the current contract, the fee is charged by the contract when the position is not liquidated.

```
let new_debt = current_debt - circle_repay;
...
let actual_redemption_amount = if (circle_amount >= max_redeemable) {
        max_redeemable
    } else {
        let remaining_debt = debt_amount - circle_amount;
        if (remaining_debt < minimum_debt + liquidation_reserve) {
            // Truncate redemption amount to maintain minimum debt
            debt_amount - (minimum_debt + liquidation_reserve)
        } else {
            circle_amount
        }
    };
```

Suggestion:

Users who legally fully redeem or close their positions should return the liquidation incentive.

Resolution:

This issue has been fixed. The client achieves the expected reserve of tokens by burning liquidation reserves.

CMU-9 Partial Clearing of Batches with Loss of Precision

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move

Descriptions:

In order to liquidate the positions of whales, there are two main restrictions when setting the minimum ratio limit:

1. debt_to_liquidate >= debt_amount The current liquidation cannot exceed the total debt
2. remaining_debt >= minimum_debt + liquidation_reserve The remaining debt has a minimum value

The multi-stage execution of `partial_liquidate` gradually increases the imbalance difference between collateral and debt as the precision loss ratio increases, so there will be a situation where the liability is 0, the collateral is not 0, and the collateral remains in the contract.

For example

```
collateral_to_liquidate = 8000 * 1 / 10000 = 0

8000*333/10000 = 266(0.4)

7734*333/9667 = 266(0.413)

7468*333/9334 = 266(0.428)

7202*333/9001 = 266(0.4443)
```

```
// If full liquidation, call liquidate function
if (debt_to_liquidate >= debt_amount) {
    liquidate<CoinType>(liquidator, user_addr);
    return
};  //  remaining_debt = debt_amount-debt_to_liquidate>0       89    debt 100 50-40   50
25-20
// remaining_debt >  minimum_debt + liquidation_reserve = 10  debt 11

// Get parameters
let minimum_debt = config::get_minimum_debt<CoinType>();
let liquidation_reserve = config::get_liquidation_reserve<CoinType>();
```

```
let liquidation_threshold = config::get_liquidation_threshold<CoinType>();
let liquidation_penalty = config::get_liquidation_penalty<CoinType>();
let liquidation_fee_protocol = config::get_liquidation_fee_protocol<CoinType>();

// Calculate ICR
let price = price_oracle::get_price<CoinType>();
let collateral_value = fixed_point32::multiply_u64(collateral_amount, price);
let current_ratio = (collateral_value * 10000) / debt_amount;
// Verify position is liquidatable
assert!(current_ratio < liquidation_threshold, events::err_cannot_liquidate());

// Verify remaining debt will be above minimum
let remaining_debt = debt_amount - debt_to_liquidate;
assert!(remaining_debt >= minimum_debt + liquidation_reserve,
events::err_invalid_debt_amount());
// Calculate proportional collateral to liquidate
let collateral_to_liquidate = ((collateral_amount as u128) * (debt_to_liquidate as u128) /
(debt_amount as u128) as u64);
```

Suggestion:

Limiting the remainder and minimum value can alleviate the impact of this problem and prevent small amounts of collateral from being locked in the contract forever.

Resolution:

The customer was aware of this problem and set a minimum lower limit for partial clearing. The small loss of accuracy was within the expected design range.

EVE-1 Set the Event Functions to `friend`

Severity: Minor

Status: Fixed

Code Location:

sources/contracts/events.move#160

Descriptions:

The event functions in the current module are public functions. Anyone can call these functions to emit logs, which could cause log records to become disorganized.

```
public fun emit_trove_closed(
        user: address,
        collateral_type: TypeInfo,
        collateral_returned: u64,
        debt_repaid: u64,
        timestamp: u64
    ) {
        event::emit(TroveClosedEvent {
            user,
            collateral_type,
            collateral_returned,
            debt_repaid,
            timestamp
        });
    }
```

Suggestion:

It is recommended to change to friend permission.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

POR-1 The Supra Oracle could Return a Stale Price

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/price_oracle.move#105-126

Descriptions:

In the `get_price_from_supra()` function, the protocol fetches the price of an asset from the Supra Oracle.

```
fun get_price_from_supra(pair_id: u32): FixedPoint32 {
    let (price, decimals, _, _) = supra_oracle_storage::get_price(pair_id);
    let decimals_u8 = (decimals as u8);
```

However, there is a issue: the protocol does not verify the threshold or validity of the returned price. This means that the Supra Oracle could return a stale, outdated, or manipulated price, which could lead to incorrect calculations and potentially severe consequences for the protocol.

Suggestion:

It is recommended to ensure that the price returned by the oracle is recent by comparing its timestamp with the current time and verifying that the price is greater than 0.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-10 User Lacks Buffer Space

Severity: Medium

Status: Acknowledged

Code Location:

sources/contracts/cdp_multi.move

Descriptions:

Administrators can use `set_operation_status` to handle some special situations to suspend the function, but there is no buffer time in the current situation, which may cause users who are preparing to increase their ICR or other users to be liquidated after resuming the status.

Suggestion:

The client will consider adding time locks to management functions at a later stage.

CMU-11 Users Can Self-liquidate

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#386

Descriptions:

The `liquidate()` function allows users to **self-liquidate**, which is not permitted in most similar protocols.

```
public entry fun liquidate(
        liquidator: &signer,
        user_addr: address
    ) acquires ConfigParams, TroveManager, UserPositionsTable, SignerCapability,
PriceOracle {
    ...
    assert_trove_active(user_addr);
    ...
    assert!(current_ratio < config.liquidation_threshold, ERR_CANNOT_LIQUIDATE);
    }
```

Suggestion:

It is recommended to check `liquidator != user_addr`.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-12 `set_price` not Marked as `test-only`

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#687

Descriptions:

According to the comment, the `set_price()` function should be **test-only** for mainnet deployment. However, the admin can call this function arbitrarily.

```
// This function should be test-only for mainnet deployment
 // Price updates should come from oracle feeds in production
 public entry fun set_price<CoinType>(
    admin: &signer,
    price: u64
 ) {
    assert!(signer::address_of(admin) == @cdp, events::err_not_admin());
    price_oracle::set_price<CoinType>( price);
 }
```

Suggestion:

It is recommended to mark `set_price` as test-only.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-13 The User's Position may not be Liquidated

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#413

Descriptions:

The `repay_or_withdraw()` function allows users to withdraw collateral, while the `liquidate()` function requires that the **current collateralization ratio (** `current_ratio` **)** be **greater than 100% (10000 in basis points)** to initiate liquidation.

```
    public entry fun repay_or_withdraw<CoinType>(
        user: &signer,
        collateral_withdraw: u64,
        circle_repay: u64
    ) acquires   TroveManager, SignerCapability {
        let user_addr = signer::address_of(user);
        positions::assert_position_exists<CoinType>(user_addr);
        let collateral_type = type_info::type_of<CoinType>();

        // let position = table::borrow_mut(user_positions, collateral_type);
        let (current_collateral, current_debt, _, _) = positions::get_position<CoinType>
(user_addr);

        // Verify balances
        assert!(current_collateral >= collateral_withdraw,
events::err_insufficient_collateral_balance());
        assert!(current_debt >= circle_repay, events::err_insufficient_debt_balance());
```

```
    // Verify position is liquidatable (ICR < liquidation threshold)
        assert!(current_ratio < liquidation_threshold, events::err_cannot_liquidate());

        // If ICR <= 100%, revert transaction
        assert!(current_ratio > 10000, events::err_invalid_liquidation());
```

However, there is a **attack path** that exploits this logic, allowing a user to manipulate their position to avoid liquidation. Here's a detailed explanation of the issue and how it can be addressed:

The Attack Path

1. **Initial Deposit**:
    - Alice deposits **10 units of collateral** when the collateral price is **12**. The total collateral value is `10 * 12 = 120`.
    - Alice mints **120 debt tokens** against this collateral. At this point, the collateralization ratio is:

```
collateralization_ratio = (collateral_value * 10000 / debt)
                        = (120*10000 / 120)
                        = 10000 (100%)
```

2. **Price Increase and Withdrawal**:
    - The price of the collateral increases to **24**. Alice calls `repay_or_withdraw()` to withdraw **5 units of collateral**.
    - After withdrawal, Alice has **5 units of collateral** left, with a total collateral value of `5 * 24 = 120`.
    - The debt remains **120**, so the collateralization ratio is still:

```
collateralization_ratio = (120*10000 / 120)
                        = 10000 (100%)
```

3. **Price Drop and Liquidation Check**:
    - The price of the collateral drops back to **10**. The value of the remaining collateral is now `5 * 10 = 50`.
    - The collateralization ratio becomes:

```
collateralization_ratio = (50 *10000/ 120)
                        ≈ 4166 (41.66%)
```

    - The position is now **undercollateralized** and should be liquidated. However, the `liquidate()` function requires `current_ratio > 10000` to initiate liquidation. Since

`4166 < 10000`, the position **cannot be liquidated**.

Suggestion:

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-14 The Verification of Total Distribution is Vulnerable to Precision Loss

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#468

Descriptions:

In the `liquidate()` function, the protocol requires that the sum of the liquidator reward, protocol fee, and user refund equals the total collateral amount being liquidated.

```
assert!(liquidator_reward_in_collateral + protocol_fee_in_collateral + user_refund == collateral_amount, events::err_invalid_liquidation());
```

However, there is a issue with this requirement: precision loss in the calculations of `liquidator_reward_in_collateral` and `protocol_fee_in_collateral` can cause the condition to fail, even when the logic is correct.

```
 let penalty_amount_in_collateral = fixed_point32::divide_u64(penalty_amount, price);
        let protocol_fee_in_collateral = (penalty_amount_in_collateral * liquidation_fee_protocol) / 10000;
        let liquidator_penalty_in_collateral = penalty_amount_in_collateral - protocol_fee_in_collateral;

        // // Calculate total penalty and possible refund
        let total_penalty_in_collateral = penalty_amount_in_collateral + fixed_point32::divide_u64((debt_amount), price);

        let liquidator_reward_in_collateral=liquidator_penalty_in_collateral+ fixed_point32::divide_u64(debt_amount, price) ;
        let user_refund = if (total_penalty_in_collateral < collateral_amount) {
            collateral_amount - total_penalty_in_collateral
        } else {
            0
        };
```

Suggestion:

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-15 Missing a bad Debt Position Process

Severity: Medium

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#386-480

Descriptions:

In a DeFi protocol, the liquidation mechanism is designed to resolve undercollateralized positions by selling the collateral to repay the debt. However, there are scenarios where liquidation may not be profitable or some positions may not be liquidatable. Hence, implementing a bad debt position process is essential for managing undercollateralized positions that cannot be liquidated profitably or at all.

Suggestion:

It is recommended to implement a bad debt position process.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-16 Missing Fee Validation in `redeem()` Function

Severity: Minor

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#691-693

Descriptions:

In the `redeem()` function, the protocol deducts a redemption fee and a user gratuity fee before transferring collateral to the user. However, there is no validation to ensure these fees are greater than zero.

```
        let (collateral_amount, debt_amount, _, _) = positions::get_position<CoinType>
(provider_addr);
        let redemption_fee = config::get_redemption_fee<CoinType>();
        let redemption_fee_gratuity = config::get_redemption_fee_gratuity<CoinType>();
        let minimum_debt = config::get_minimum_debt<CoinType>();
        let liquidation_reserve = config::get_liquidation_reserve<CoinType>();
```

A bad actor can user small amount to avoid the fee to close the trove.

Suggestion:

It is recommended to ensure the fee is greater than 0.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-17 Lack of Event

Severity: Minor

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#802

Descriptions:

We should add event monitoring for privileged operations and important functions modified by administrators, because these functions are very important and are usually difficult to track directly through transactions on the aptos/sui chain.``` js public entry fun set_oracle(admin: &signer, new_oracle_id: u32, new_price_age: u64) { // Verify admin assert! (signer::address_of(admin) == @cdp, events::err_not_admin()); price_oracle::update_oracle(new_oracle_id, new_price_age); } public entry fun set_operation_status(admin: &signer, open_trove: bool, borrow: bool, deposit: bool, redeem: bool) { assert! (signer::address_of(admin) == @cdp, events::err_not_admin());

config::set_operation_status<CoinType>(open_trove, borrow, deposit, rede

} public entry fun set_fee_collector(admin: &signer, new_fee_collector: address) { assert! (signer::address_of(admin) == @cdp, events::err_not_admin());

config::set_fee_collector(new_fee_collector); }

Suggestion:

Recommend adding events

Resolution:

This issue has been fixed. The client has adopted our suggestions.

CMU-18 `supra_oracle_storage` is Not Available

Severity: Informational

Status: Fixed

Code Location:

sources/contracts/cdp_multi.move#1

Descriptions:

The contract address pushed by `supra_oracle_storage` is not as expected. The implementation may be used incorrectly. The `storage` contract on Aptos is marked as `N/A`.

```
use supra_oracle::supra_oracle_storage;
```

https://docs.supra.com/oracles/data-feeds/pull-oracle/networks

No	Mainnet	Pull Contract	Storage Contract
1	Aptos Mainnet	**Contract Address:** 0x4463d20beee3387cd17389efd6d779de69ec4959b6bc794eca2a217d93189533 **Oracle Holder:** 0x3d28f3142854ce60f387df9762886d27d2b58c0a5bd237f596f5bf6d5c9fad31	N/A

Suggestion:

The confirmed oracle pull address will be used in the official deployment.

Resolution:

This issue has been fixed. The client has adopted our suggestions.

Appendix 1

Issue Level

- **Informational** issues are often recommendations to improve the style of the code or to optimize code that does not affect the overall functionality.
- **Minor** issues are general suggestions relevant to best practices and readability. They don't post any direct risk. Developers are encouraged to fix them.
- **Medium** issues are non-exploitable problems and not security vulnerabilities. They should be fixed unless there is a specific reason not to.
- **Major** issues are security vulnerabilities. They put a portion of users' sensitive information at risk, and often are not directly exploitable. All major issues should be fixed.
- **Critical** issues are directly exploitable security vulnerabilities. They put users' sensitive information at risk. All critical issues should be fixed.

Issue Status

- **Fixed:** The issue has been resolved.
- **Partially Fixed:** The issue has been partially resolved.
- **Acknowledged:** The issue has been acknowledged by the code owner, and the code owner confirms it's as designed, and decides to keep it.

Appendix 2

Disclaimer

This report is based on the scope of materials and documents provided, with a limited review at the time provided. Results may not be complete and do not include all vulnerabilities. The review and this report are provided on an as-is, where-is, and as-available basis. You agree that your access and/or use, including but not limited to any associated services, products, protocols, platforms, content, and materials, will be at your own risk. A report does not imply an endorsement of any particular project or team, nor does it guarantee its security. These reports should not be relied upon in any way by any third party, including for the purpose of making any decision to buy or sell products, services, or any other assets. TO THE FULLEST EXTENT PERMITTED BY LAW, WE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, RELATED SERVICES AND PRODUCTS, AND YOUR USE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NOT INFRINGEMENT.