Zellic

Move Dollar

Smart Contract Security Assessment

February 28, 2023

Prepared for:

Adam Cader

Thala Labs

Prepared by:

Daniel Lu and Oliver Murray

Zellic Inc.

Contents

About Zellic

Zellic was founded in 2020 by a team of blockchain specialists with more than a decade of combined industry experience. We are leading experts in smart contracts and Web3 development, cryptography, web security, and reverse engineering. Before Zellic, we founded perfect blue, the top competitive hacking team in the world. Since then, our team has won countless cybersecurity contests and blockchain security events.

Zellic aims to treat clients on a case-by-case basis and to consider their individual, unique concerns and business needs. Our goal is to see the long-term success of our partners rather than simply provide a list of present security issues. Similarly, we strive to adapt to our partners' timelines and to be as available as possible. To keep up with our latest endeavors and research, check out our website zellic.io or follow @zellic_io on Twitter. If you are interested in partnering with Zellic, please email us at hello@zellic.io or contact us on Telegram at https://t.me/zellic_io.

1 Executive Summary

Zellic conducted an audit for Thala Labs from September 26th to October 7th, 2022.

> It is worth noting that the project name has been changed from Aptos Dollar (APD) to Move Dollar (MOD). However, the commit reviewed in this report still uses the APD acronym in the codebase. As a result, we will also use Aptos Dollar and the APD acronym in this report to maintain consistency with the codebase.

Our general overview of the code is that it was very well-organized and structured. However, the code base is incomplete as it pertains to Thala's documentation and our interpretation of what is required to create a robust stablecoin protocol.

Unit tests are generally lacking, and integration testing should be expanded upon considerably. It is critical that algorithmic stablecoins such as this, which require sequences of competing user interactions to maintain the peg, have corresponding test suites that show this.

Zellic thoroughly reviewed the Aptos Dollar codebase to find protocol-breaking bugs as defined by the documentation and to find any technical issues outlined in the Methodology section (2.2) of this document.

Specifically, taking into account Aptos Dollar's threat model, we focused heavily on issues that would break core invariants such as calculating the positions in the pool as well as the states that the 'LendingStorageManager' handles for the liquidity providers.

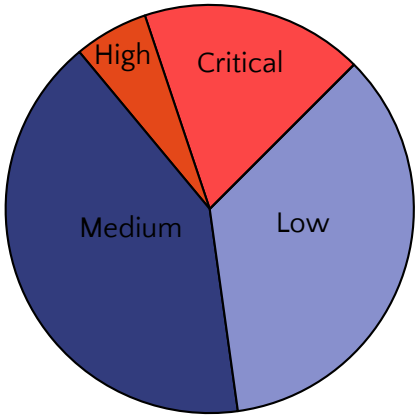
During our assessment on the scoped Aptos Dollar contracts, we discovered 17 findings, three of which were critical and are currently in the process of remediation. Of the remaining 14 findings, one was high severity, seven were medium severity, and six were low severity.

Additionally, Zellic summarized its notes and observations from the audit for Thala Labs's benefit in the Discussion section (5) at the end of the document.

Subsequent to reviewing our findings and commentary, Thala added considerable functionality to the protocol. Additional tests have been added to support that functionality. However, these changes were considerable in nature and require a separate review to verify their security and economic soundness.

Breakdown of Finding Impacts

Impact Level	Count
Critical	3
High	1
Medium	7
Low	6
Informational	0



2 Introduction

2.1 About Aptos Dollar

Thala is a decentralized finance (DeFi) protocol built on Aptos, with a collateralized debt position (CDP) stablecoin and an automated market maker (AMM). The two core products will revolve around Aptos Dollar (APD), the protocol's overcollateralized stablecoin.

As a native stablecoin to Aptos, APD can be used to transact, facilitate, and interact with various DeFi protocols within the ecosystem, while acting as a store of value, a medium of exchange, and a unit of account.

2.2 Methodology

During a security assessment, Zellic works through standard phases of security auditing including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of open-source tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. We analyze the scoped smart contract code using automated tools to quickly sieve out and catch these shallow bugs. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so forth as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We manually review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents. We also thoroughly examine the specifications and designs themselves for inconsistencies, flaws, and vulnerabilities. This involves use-cases that open the opportunity for abuse, such as flawed tokenomics or share pricing, arbitrage opportunities, and so forth.

Complex integration risks. Several high-profile exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. We perform a meticulous

review of all of the contract's possible external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so forth.

Code maturity. We review for possible improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradeability weaknesses, centralization risks, and so forth.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact; we assign it on a case-by-case basis based on our professional judgment and experience. As one would expect, both the severity and likelihood of an issue affect its impact; for instance, a highly severe issue's impact may be attenuated by a very low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Similarly, Zellic organizes its reports such that the most important findings come first in the document rather than being ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their importance may differ. This varies based on numerous soft factors, such as our clients' threat models, their business needs, their project timelines, and so forth. We aim to provide useful and actionable advice to our partners that consider their long-term goals rather than simply provide a list of security issues at present.

2.3 Scope

The engagement involved a review of the following targets:

Aptos Dollar Contracts

Repository https://github.com/ThalaLabs/thala-protocol-v1/

Versions `b1eb47e32794b5238b78a0470f931f684b565f1d`

Programs

- sources/math.move
- sources/sorted_vaults.move
- sources/apd_coin.move
- sources/stability_pool.move
- sources/scripts/vault_scripts.move
- sources/scripts/stability_pool_scripts.move
- sources/oracle.move
- sources/manager.move
- sources/params.move
- sources/fixed_point.move
- sources/thl_coin.move
- sources/vault.move

Type Move-Aptos

Platform Aptos

2.4 Project Overview

Zellic was contracted to perform a security assessment with three consultants for a total of four person-weeks. The assessment was conducted over the course of two calendar weeks.

Contact Information

The following project managers were associated with the engagement:

Jasraj Bedi, Co-founder
jazzy@zellic.io

Stephen Tong, Co-founder
stephen@zellic.io

The following consultants were engaged to conduct the assessment:

Daniel Lu, Engineer
daniel@zellic.io

Oliver Murray, Engineer
oliver@zellic.io

2.5 Project Timeline

The key dates of the engagement are detailed below.

September 26, 2022	Start of primary review period
October 7, 2022	End of primary review period

3 Detailed Findings

3.1 Stealing of liquidation rewards in `stability_pool`

- **Target**: `stability_pool`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

The share of liquidation rewards entitled to APD depositors in the stability pool depends on the user's deposited amount relative to the value of the pool at the time of the liquidation call.

```
// Compute share of the pool
let (deposit_amount, _, next)
    = iterable_table::borrow_iter_mut(stability_pool_deposits,
    depositor);
let share = ((*deposit_amount as u128) * SHARE_DECIMAL_CONSTANT)
    / (stability_pool_apd_amount as u128);
…
let collateral_share_amount = (((collateral_amount as u128) * share)
    / SHARE_DECIMAL_CONSTANT as u64);
*depositor_collateral_share = *depositor_collateral_share
    + collateral_share_amount;
```

There is nothing to enforce that depositors of APD who are compensated from profitable liquidation events actually had APD deposited prior to the profitable liquidation event and hence exposure to losses.

Impact

The above mechanism creates the following attack vector:

1. Identify profitable liquidation vaults (these are deterministic and can be determined from reviewing the liquidation compensation mechanism).

2. Deposit a large amount of APD into the liquidation pool to obtain a disproportionate share of the rewards.

3. Call liquidate, receive rewards, and withdraw them from the liquidity pool.

With access to sufficient amounts of APD, a malicious user could claim the vast majority of the rewards. Such attacks would lead to loss of confidence in the protocol. Users would likely remove their funds from the stability pool due to lack of compensation for risks taken.

Recommendations

The attack can be discouraged by enforcing timelocks on APD deposits into the stability pool. However, there is still the potential for gaming. For example, depending on market conditions, it could be economically rational to flood the pool with APD to steal liquidation rewards and ride out any subsequent exposure to losses in the stability pool. The use of timelocks would, however, prevent pool takeovers from flash loans.

A more involved fix would be to require that compensation to APD depositors depends on the amount of time they have been in the pool. Of course, this also needs careful consideration as it may discourage important sources of liquidity from supporting the pool if they are not going to be compensated for it.

Remediation

To mitigate risk-free profit from opportunistic deposits, the protocol now requires liquidity providers to hold funds in the pool for 24 hours or incur a linear fee. This solution would theoretically help significantly with the problem, but it would require separate review due to the presence of extensive architectural changes.

We believe the fix does not entirely mitigate the issue — depositors can still front-run profitable events to add funds and accept the 24 hours of risk. We encourage Thala Labs to evaluate further mitigations (such as a short delay on the deposit side) and consider whether they would be helpful for the protocol economics.

3.2 Riskless liquidation rewards in `stability_pool`

- **Target**: `stability_pool`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

The profits for stability pool depositors are initially increasing as the price of collateral assets relative to APD decreases. Profits continue to increase until they reach their maximum, after which they begin to decrease and eventually become losses.

The intercepts for profit and loss and the location of peak profit depend on system parameters. However, in general there is an optimal liquidation price at which maximum profit is realized for the liquidation. Below this price, profits are decreasing until they eventually cross a critical threshold and turn into losses.

Impact

Because APD depositors are able to freely deposit and withdraw funds from the stability pool, the incentive mechanism above creates free optionality for APD depositors. For example, a clever depositor can avoid losses in all cases by

1. Calling liquidate themselves when it optimizes the profit of the stability pool.

2. Front-run liquidation events that would result in losses by withdrawing APD prior to the liquidation call.

Furthermore, there are no economic incentives for anyone who is not a stability pool depositor to call `vault::liquidate<CoinType>`.

A malicious actor who follows this strategy can effectively steal other APD depositors from their compensation.

It is likely that word of this exploit would spread, resulting in other APD depositors following this strategy or, when unable to do so, removing their deposits from the protocol.

This would effectively break a critical mechanism of the design and the integrity of the stablecoin.

Recommendations

We recommend the following changes in order to remove the attacker vector:

1. Add timelocks for depositors.

2. Provide incentives for nonstability pool depositors to call `vault::liquidate<CoinType>`.

It is important to note, however, that the proper functioning of the stablecoin protocol requires that APD depositors have timely access to their funds. For example, it may be necessary to support other mechanisms in the protocol such as calls to `vault::redeem_collateral<Coin_Type>`.

Careful consideration should be made in determining the appropriate length of time for timelocks. It may even be advisable to actively manage the length of timelocks in response to market conditions.

Remediation

Thala Labs has incorporated a mitigation for this issue by enforcing a minimum deposit time with a linear fee. Due to extensive changes in the project, a separate review would be required to confirm its correctness. To further discourage liquidity providers from front-running negative events to leave the pool, we encourage Thala Labs to consider adding a short delay for all withdrawals — even those by depositors who have spent significant time in the pool.

3.3 Redemption mechanism allows undercollateralized vaults to escape liquidation penalization

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: Critical

Description

Undercollateralized vaults can have their debt paid off without incurring liquidation penalities when users make calls to `vault::redeem_collateral`.

The amount of debt paid off in a given vault during a call to liquidation is given by the following:

```
let redeemed_usd = fixed_point::min(
    fixed_point::min(collateral_usd, debt_usd),
    fixed_point::from_u64(coin::value(&remained_debt_coin)),
);
```

In the event that `collateral_usd < debt_usd` and `collateral_usd < remained_debt_coin` prior to the call to `repay_internal`, and a `remained_debt_coin > 0` after the call to `repay_internal`,

```
repay_internal<CoinType>(redeemee, coin::extract(&mut remained_debt_coin,
    redeemed_debt));
```

the full collateral of the vault will be removed and an amount of debt equal to the collateral amount will be paid. However, the vault will hold a debt equal to `debt_usd - collateral_usd`.

Additionally, the vault with zero collateral and nonzero debt will be reinserted into the sorted vault:

```
// update sorted_vaults
if (coin::value(&remained_debt_coin) != 0) {
    // all debt repayed, so should be inserted as head
    sorted_vaults::reinsert<CoinType>(
        redeemee,
        math::compute_nominal_cr(0, 0),
        option::none(),
```

```
        sorted_vaults::get_first<CoinType>(),
    );
} else {
```

Impact

The ability for undercollateralized positions to be exited without paying penalties to the stability pool disincentivizes users from supporting the stability of the protocol by depositing APD into the stability pool. Furthermore, it creates a way for undercollateralized vaults to redeem their collateral without incurring any penalty.

Vaults with zero collateral and nonzero debt should not exist in the system at all, let alone in the head of the `SortedVaults`, where it is assumed that positions have nonzero debt.

Furthermore, this APD would effectively be locked out from burning and would result in outstanding APD that is not backed by collateral.

While this might not immediately break the protocol, these unbacked APD positions could accumulate over time. Given one of the aims of the protocol is to ensure that all APD is not only backed by collateral but is overcollateralized, this could result in loss of confidence in the protocol.

Recommendations

The situation can be avoided if undercollateralized vaults cannot be redeemed but can only be liquidated.

An undercollateralization check should be included in the logic for `vault::redeem_collateral`. Thala should consider auto-liquidating these vaults when they are encountered during calls to `vault::redeem_collateral`.

Remediation

Thala Labs has implemented a fix in commit 9ac67d7c that skips redemption when vaults are undercollateralized. This mitigation addresses the issue pointed out, but its interactions with other significant protocol changes would require a separate review.

3.4 Public access to `register_collateral` can lock out collateral `CoinTypes` from APD

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: High
- **Impact**: High

Description

The function `stability_pool::register_collateral` is `public` when it should be `public(friend)`:

```
public entry fun register_collateral<CoinType>(account: &signer) {
    assert!(signer::address_of(account) == @thala_protocol_v1,
    ERR_UNAUTHORIZED);
    assert!(initialized(), ERR_UNINITIALIZED);

    if (!exists<DistributedCollateral<CoinType>>(@thala_protocol_v1)) {
        let collateral = coin::zero<CoinType>();
        let shares = table::new<address, u64>();
        move_to(account, DistributedCollateral { collateral, shares });
    }
}
```

Impact

A malicious actor can call `register_collateral` for any `CoinType` prior to this function being called from its intended control flow via an internal function call made by `vault::initialize`.

The assertion checks in `vault::initialize`:

```
public entry fun initialize<CoinType>(manager: &signer) {
    assert!(signer::address_of(manager) == @thala_protocol_v1,
    ERR_INVALID_MANAGER);
    assert!(manager::initialized(), ERR_UNINITIALIZED_MANAGER);
    assert!(!exists<CollateralState<CoinType>>(@thala_protocol_v1),
    ERR_INITIALIZED_COLLATERAL);

    stability_pool::register_collateral<CoinType>(manager);
    sorted_vaults::initialize<CoinType>(manager);
```

```
    move_to(manager, CollateralState<CoinType> {
        total_collateral: 0,
        total_debt: 0,
    });
}
```

This will prevent the protocol manager from being able to initialize vaults for the given `CoinType`.

In the worst case scenario, it would be possible for an attacker to completely prevent the deployment of vaults for any `CoinType`.

Recommendations

Modify the access to `stability_pool::register_collateral` from `public` to `public(friend)`.

Remediation

Thala Labs has followed our recommendation and changed `stability_pool::register_collateral` from `public` to `public(friend)` in commit fdba1010.

3.5 Partially filled APD redemptions always charge the full redemption fees

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Medium
- **Impact**: Medium

Description

Partially filled APD redemptions always charge the full redemption fee, even if some of the APD passed in the function call is not redeemed:

```
public fun redeem_collateral<CoinType>(
        debt: Coin<APD>,
        // TODO - take hints from the off-chain
        // prev: Option<address>,
        // next: Option<address>,
    ): (Coin<CoinType>, Coin<APD>) acquires VaultStore, CollateralState {
        let remained_debt_coin = debt;
        let redeemed_collateral_coin = coin::zero<CoinType>();

        let redemption_fee_amount = {
            let redemption_fee = get_redemption_fee<CoinType>();
            let remained_debt_amount
    = fixed_point::from_u64(coin::value(&remained_debt_coin));
            fixed_point::to_u64(fixed_point::mul(remained_debt_amount,
    redemption_fee))
        };
        let redemption_fee_coin = coin::extract(&mut remained_debt_coin,
    redemption_fee_amount);
        manager::charge_redemption_fee(redemption_fee_coin);
```

Impact

Because the variable `redemption_fee_coin` is not adjusted to account for partial redemptions, users who call `vault::redeem_collateral` are always charged the full redemption fee.

This could discourage users from calling `vault::redeem_collateral` and potentially alter the economics of interacting with the protocol to the point where users seek alternative stablecoin protocols.

Recommendations

Calculate `redemption_fee_coin` at the end of the `vault::redeem_collateral` based on the actual amount of APD redeemed.

Remediation

Thala Labs updated the function in commit 6de6e464 to charge the correct fee for redemption. Since other architectural changes have affected the function as well, additional review would be required to confirm the correctness of redemption mechanics.

3.6 Distribution mechanism for liquidation rewards susceptible to `max_gas`

- **Target**: `stability_pool`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

On the liquidation of undercollateralized vaults, control is passed from `vault::liquidate` to `stability_pool::distribute_collateral_and_request_apd`. This function uses a while loop to iterate over all of the addresses in an iterable table:

```
struct StabilityPool has key {
    apd: Coin<APD>,
    deposits: IterableTable<address, u64>,
    num_depositors: u64,
}

...
public(friend) fun distribute_collateral_and_request_apd<CoinType>(
        vault_addr: address,
        requested_apd: u64,
        collateral: Coin<CoinType>
    ): Coin<APD> acquires StabilityPool, StabilityPoolEvents,
    DistributedCollateral {
         ...
        let depositor_iter_option
    = iterable_table::head_key(stability_pool_deposits);
        while (option::is_some(&depositor_iter_option)) {
            let depositor = *option::borrow(&depositor_iter_option);
         ...
```

Impact

As the number of APD depositors grows, the gas costs of liquidation will steadily increase. Additionally, a malicious attacker could flood the `StabilityPool.deposits` iterable table with accounts with zero APD deposited. This could eventually lead to `max_gas` and the inability for stability pool depositors to be rewarded for risks taken in supporting the stability pool.

Recommendations

We suggest Thala Labs adopt the reward distribution mechanism central to the ERC 4626 token vault standard. Rather than looping over depositors when allocating rewards, it increments the redemption value of shares held by all depositors to reflect increases in claimable rewards.

Remediation

Thala Labs has overhauled the reward system and adopted a pull-based approach for distributions. These changes can be seen in commit 513f0736. Conceptually, this is a move in the right direction; however, verifying the security of these changes would require a separate review.

3.7 Low collateral positions can lead to `max_gas`

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The `vault::open_vault` function described previously enforces minimum collateralization rates but not minimum collateral.

The implementation of `sorted_vaults` maintains a list of vaults ordered by decreasing collateralization rate. The `redeem_collateral` function in `vault.move` iterates from the sorted list's tail to extract collateral for APD; this can be expensive. Consider this excerpt from its implementation:

```
while (option::is_some(&min_cr_address)
    && coin::value(&remained_debt_coin) > 0) {
    let redeemee = *option::borrow<address>(&min_cr_address);

    // [...]

    min_cr_address = sorted_vaults::get_prev<CoinType>(redeemee);

    // update sorted_vaults
    if (coin::value(&remained_debt_coin) != 0) {
        // all debt repayed, so should be inserted as head
        sorted_vaults::reinsert<CoinType>(
            redeemee,
            math::compute_nominal_cr(0, 0),
            option::none(),
            sorted_vaults::get_first<CoinType>(),
        );
    } else {
        let (vault_collateral, vault_debt)
    = collateral_and_debt_amount<CoinType>(redeemee);

        // not all debt repayed, so should be reinserted with hint
        sorted_vaults::reinsert<CoinType>(
            redeemee,
            math::compute_nominal_cr((vault_collateral as u128),
    (vault_debt as u128)),
```

```
            option::none(), // TODO - should be prev
            option::none(), // TODO - should be next
        );
    }
};
```

Essentially, this begins at the vault with the lowest collateralization rate and iterates towards the head. It extracts collateral from positions until all the given APD is exchanged. Each iteration reinserts the empty vault at the head, with the last requiring a traversal to find an insertion position.

Impact

Because traversal begins at the end of the sorted vaults and continues until collateral is fully redeemed, an abundance of low-collateral vaults at the list's tail will make `redeem_collateral` more expensive in gas.

An attacker could open many vaults with low collateral, setting the borrow amount to barely reach minimum collateralization rate. These low-collateral positions would be placed near the end of the sorted vaults where collateral redemption begins. This would increase gas costs and could lead to `max_gas` in `vault::redeem_collateral`, affecting the ability of users to exchange APD for collateral.

Recommendations

We recommend that `vault::open_vault` enforces a minimum collateral requirement. This would significantly lessen the impact of flooding the sorted vault, as the redemption of collateral would require fewer positions.

Remediation

Thala Labs has added logic to prevent the system from being flooded with zero- or low-collateral vaults. The additional checks can be found in commit 6de6e464. However, fully verifying the correctness would require a separate review of the extensive architectural changes.

3.8 Accumulation of vaults can lead to `max_gas` via insertion algorithm

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

There are no controls preventing the creation of vaults with zero collateral in the call to `vault::open_vault`.

Additionally, there are no processes in place to remove vaults with zero collateral. The complete liquidation of all collateral in a vault does not result in a function call to `vault::close_vault`. Furthermore, the current implementation of `vault::close_vault` does not actually remove the vault from the `SortedVaults` data structure.

The insertion and reinsertion algorithm of `sorted_vaults` uses the nominal collateralization ratio to determine the order placement of vaults inserted and reinserted into the `SortedVaults` data structure. In the current implementation, vaults with zero collateral (and hence zero debt) are placed at the front of the linked list.

```
public fun compute_nominal_cr(collateral: u128, debt: u128): u128 {
    if (debt > 0) {
        (collateral * NICR_PRECISION / debt)
    } else {
        // Return the maximal value for u128 if the Trove has a debt of 0.
    Represents “infinite” CR.
        MAX_U128
    }
}
```

Impact

In the current implementation, there are in general no hints provided for the insertion and reinsertion of vaults, whether they have zero or nonzero collateral or not. In the majority of cases, insertion or reinsertion require traversing the linked list from the head until the placement determined by the rank order of the nominal collateralization ratio is found.

Uncontrolled size of the linked list can result in increasing gas costs for interacting with the protocol and ultimately its failure due to `max_gas`.

There are two separate vectors contributing to reaching `max_gas`:

1. A malicious attacker can flood the system with zero-collateral vaults using calls to `vault::open_vault`.
2. Depending on the number of users in the protocol, its regular operation will result in the steady increase of zero-collateral vaults that are never removed either by calls to `vault::close_vault` or `vault::liquidate`.

A combination of the above is the most likely avenue to reaching `max_gas`.

Recommendations

There are several recommendations that should be followed in order to address the issue:

1. Ensure that vaults cannot be opened with zero collateral. Furthermore, it may be beneficial to enforce a minimum collateral amount in order to open a vault to reduce the economic feasibility of the attack mentioned above.
2. Ensure that calls to `vault::close_vault` result in removal of the vault from `SortedVaults`.
3. Ensure that the complete liquidation of vaults results in calls to the updated version of `vault::close_vault`.
4. Provide hints to the insertion and reinsertion algorithm to avoid traversing the linked list from head when it is not necessary.

Remediation

Thala Labs has added checks to ensure that zero-collateral vaults cannot be created. Specifically, in commit 6de6e464, vault creation is enforced against a minimum debt requirement. Fully validating these checks would require a separate review of new data structures and how they interact with the rest of the protocol.

3.9 Unable to unregister collateral `CoinTypess`

- **Target**: `stability_pool`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

There is currently no way to unregister collateral assets from the protocol. Furthermore, there is no mechanism to disincentivize borrowing against collateral assets that no longer meet Thala's risk framework.

Impact

During the evolution of the protocol, it is likely that some of the assets that were initially deemed suitable for inclusion in the APD stablecoin protocol no longer satisfy these conditions.

For example, the volatility of collateral assets is in no way guaranteed to remain within a range acceptable to the framework. In the event one of the collateral assets becomes too volatile, there would be no way to remove it from the system.

Because the stability pool supports all collateral `CoinTypes`, the inability to remove or discourage the use of the volatile assets could disincentivize APD depositors from supporting the stability pool. For example, it could increase the perceived likelihood of liquidation events that result in losses for stability pool depositors.

Recommendations

Thala Labs has identified the need for appropriate mechanisms to disincentivize the use of such collateral assets. The proposal references interest rates for debt borrowers. We recommend Thala Labs flesh out these mechanisms so that they can be reviewed.

Remediation

Among the considerable architectural changes that Thala Labs has made, one has been the incorporation of capabilities like asset freezing into the protocol. This functionality is present in commit 6de6e464. The update does address our concern, but the code changes are very extensive. Verifying its security and functionality would require a separate review.

3.10 Missing oracle stale price check

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

The oracle does not keep a time stamp, and there is no infrastructure in place to check for stale prices.

```
struct PriceStore<phantom CoinType> has key {
    numerator: u64,
    denominator: u64,
}
```

Impact

Even if there is a rigorous oracle-updating mechanism, stale price checks can prevent catastrophic outcomes in the event the oracle has issues.

During volatile markets or rapid price movements, the true market price could easily deviate from the price in the `PriceStore`. Allowing users to interact with the protocol using stale prices opens up the avenue for a multitude of exploits given the large number of ways users can interact with the protocol.

For example, they could avoid liquidation events, redeem collateral at favorable prices, borrow excess APD, and so forth.

Recommendations

Expand the oracle `PriceStore` to include time stamps reflecting calls made to `oracle::set_price<CointType>` by the `oracle_manager`.

Additionally, calls made to `oracle::price_of<CoinType>` by `get_oracle_price<CoinType>` should check time stamps against the current time to evaluate whether prices are stale or not.

We suggest the protocol managers incorporate a combination of statistical price analysis and market expectations to determine the appropriate time window since the last oracle update. It may also be advisable to incorporate some flexibility into the time window — for example, it is possible for prices to become increasingly stale during volatile markets with rapid price movements.

We further suggest that Thala Labs make available the processes for updating their price oracle so that we can assess its robustness.

Remediation

Thala Labs has made commendable efforts to mitigate issues due to stale oracle prices: for instance, the project now uses a tiered oracle system that considers factors like staleness. However, the oracle framework has been expanded considerably and would require a separate review to ensure the issue has been fixed.

The new oracle system exists in commit 6de6e464.

3.11 Centralization risk

- **Target**: Project Wide
- **Category**: Centralization Risk
- **Likelihood**: Medium
- **Severity**: Medium
- **Impact**: Medium

Description

There are several mechanisms in which the operators of the protocol can influence the protocol in material ways. Protocol managers can exert control over the following critical operations:

1. Control over the minimum collateralization ratio (MCR) and redemption fees.
2. Vault initialization and collateral `CoinTypes` used in the protocol.
3. Control over the price oracle.

Impact

In the most severe cases, control over the aforementioned mechanisms can lead to the following outcomes.

Calls to `params::set_params` can be used to reduce the value of the MCR such that certain vaults become immediately eligible for liquidation. With their knowledge of the profit and loss awarded to liquidators, a malicious actor with management access could set MCR such that a subsequent vault liquidation would maximize profit. They could combine this with a flash loan to take over the majority of the liquidation rewards and effectively rug the protocol.

Additionally, calls to `params::set_params` can be used to set redemption fees to excessively high values.

Calls to `vault::initialize` can be used to register assets that do not meet the criteria of Thala Labs's risk framework. Because all vaults are supported by one stability pool, this could severely disrupt the economics and incentives for other users to use the system.

Lastly, the manager can effectively take over the oracle to set prices as they please. When done maliciously, this could severely disrupt the operation of the protocol in all manners of mechanism.

Recommendations

While it is critical for protocol managers to be able to exert control over the parameters and variables mentioned above, this access should be controlled through a multi-signature wallet.

In particular, changes to the MCR on existing pools, if made at all, should be done in combination with announcements so that users have ample time to modify their collateralization ratios and avoid liquidation.

Most projects utilize multi-signature wallets to mitigate centralization risks and provide an additional layer of security. However, there are no security benefits if access control is not implemented correctly. The keys of a multi-signature wallet should always be stored independently of each other, on physically separate hardware. Should one of the systems be compromised, the damage will be isolated. Make use of hardware wallets if possible. Also consider trusted, industry-standard key custody providers.

Remediation

Thala Labs has indicated that a multi-signature wallet (MSafe) will be used to manage centralization risk. However, in the current implementation, it is possible for any address to be used as the manager account. One possible solution would be to check that the manager account has the MSafe resource. Additional checks could potentially be included to verify that the wallet satisfies some requirements for quorum and threshold.

3.12 Missing assertion checks for critical protocol parameters

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

There are no checks in place to enforce that `params::set_params<CoinType>` has been called for a given `CoinType` prior to calling `vault::initialize<CoinType>`:

```
public entry fun initialize<CoinType>(manager: &signer) {
    assert!(signer::address_of(manager) == @thala_protocol_v1,
    ERR_INVALID_MANAGER);
    assert!(manager::initialized(), ERR_UNINITIALIZED_MANAGER);
    assert!(!exists<CollateralState<CoinType>>(@thala_protocol_v1),
    ERR_INITIALIZED_COLLATERAL);

    stability_pool::register_collateral<CoinType>(manager);
    sorted_vaults::initialize<CoinType>(manager);
    move_to(manager, CollateralState<CoinType> {
        total_collateral: 0,
        total_debt: 0,
    });
}
```

Impact

If the `ParamStore<CoinType>` has not been initialized via a call to `params::set_params<CoinType>`, subsequent calls to vault functions will fail with unclear error messages.

Recommendations

Force the parameters to be set up prior to allowing calls to `vault::initialize<CoinType>` by including the following assertion check:

```
public entry fun initialize<CoinType>(manager: &signer) {
    assert!(signer::address_of(manager) == @thala_protocol_v1,
    ERR_INVALID_MANAGER);
    assert!(manager::initialized(), ERR_UNINITIALIZED_MANAGER);
```

```
    assert!(!exists<CollateralState<CoinType>>(@thala_protocol_v1),
    ERR_INITIALIZED_COLLATERAL);
    assert!(!exists<ParamStore<CoinType>>(@thala_protocol_v1),
    ERR_UNINITIALIZED_PARAMSTORE);

    stability_pool::register_collateral<CoinType>(manager);
    sorted_vaults::initialize<CoinType>(manager);
    move_to(manager, CollateralState<CoinType> {
        total_collateral: 0,
        total_debt: 0,
    });
}
```

Remediation

Thala Labs has made extensive changes to the initialization sequence, which would require a separate review to confirm that all protocol parameters are set prior to or during the initialization.

These changes are present in commit 6de6e464.

3.13 Missing validation checks in `set_params`

- **Target**: `manager`
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Currently there are no validation checks in `params::set_params` to ensure that the following critical protocol parameters are not set to values that break the protocol.

```
public entry fun set_params<CoinType>(
    manager: &signer,
    mcr_numerator: u64,
    mcr_denominator: u64,
    redeem_fee_numerator: u64,
    redeem_fee_denominator: u64,
) acquires ParamStore {
    assert!(
        signer::address_of(manager) == @thala_protocol_v1,
        error::invalid_argument(ERR_MANAGER_ADDRESS_MISMATCH),
    );

    if (!exists<ParamStore<CoinType>>(@thala_protocol_v1)) {
        move_to(manager, ParamStore<CoinType> {
            mcr_numerator,
            mcr_denominator,
            redeem_fee_numerator,
            redeem_fee_denominator,
        });
    } else {
        let param_store
    = borrow_global_mut<ParamStore<CoinType>>(@thala_protocol_v1);
        param_store.mcr_numerator = mcr_numerator;
        param_store.mcr_denominator = mcr_denominator;
        param_store.redeem_fee_numerator = redeem_fee_numerator;
        param_store.redeem_fee_denominator = redeem_fee_denominator;
    }
}
```

Impact

The rest of the protocol ultimately makes calls to `params::mcr_of` and `params::redeem_fee_of` under the assumption that the MCR numerator is greater than the denominator and that the redeem fee numerator is less than the denominator.

Because there are no checks on their end, this is likely to result in a combination of failures and, worse, potentially erroneous calculations.

Recommendations

Include the following validation checks to ensure numerators are less than denominators:

```
public entry fun set_params<CoinType>(
    manager: &signer,
    mcr_numerator: u64,
    mcr_denominator: u64,
    redeem_fee_numerator: u64,
    redeem_fee_denominator: u64,
) acquires ParamStore {
    assert!(
        signer::address_of(manager) == @thala_protocol_v1,
        error::invalid_argument(ERR_MANAGER_ADDRESS_MISMATCH),
    );
    assert!(
        (mcr_numerator ≥ mcr_denominator),
        error::invalid_argument(ERR_MCR_NUMR_LT_DENOM),
    );

    assert!(
        (redeem_fee_numerator ≤ redeem_fee_denominator),
        error::invalid_argument(ERR_FEE_NUMR_GT_DENOM),
    );

    if (!exists<ParamStore<CoinType>>(@thala_protocol_v1)) {
        move_to(manager, ParamStore<CoinType> {
            mcr_numerator,
            mcr_denominator,
            redeem_fee_numerator,
            redeem_fee_denominator,
        });
    } else {
```

```
        let param_store
    = borrow_global_mut<ParamStore<CoinType>>(@thala_protocol_v1);
        param_store.mcr_numerator = mcr_numerator;
        param_store.mcr_denominator = mcr_denominator;
        param_store.redeem_fee_numerator = redeem_fee_numerator;
        param_store.redeem_fee_denominator = redeem_fee_denominator;
    }
}
```

Remediation

The initialization changed considerably by commit 6de6e464. Confirming that all the assertions are secure would require a separate review. We note that the new sequence now sets vault parameters to default values; however, they can be later modified with setter functions, which each need to be reviewed for proper checks. Additionally, the setter functions allow the minimum collateralization ratios to be changed after a vault has been deployed, which could introduce a centralization risk where a protocol owner causes open vaults to be liquidated.

3.14 Locked redemption fees

- **Target**: `manager`
- **Category**: Business Logic
- **Likelihood**: High
- **Severity**: Low
- **Impact**: Low

Description

Currently there is no way for the `manager` to retrieve fees stored in the `FeeStore` from calls made to `manager::charge_redemption_fee;` in `vault::redeem_collateral;`.

Impact

The owners of the protocol would be unable to retrieve redemption fees from the `FeeStore`.

Recommendations

Add an access-controlled method to the `manager` to allow the protocol owners to retrieve redemption fees.

Remediation

Thala Labs has made considerable efforts to include the functionality to allow collateral withdrawals. In commit 6de6e464, the module `thala_protocol_v1::fees` was fleshed out with the withdrawal functionality. However, the changes are extensive and require a separate review.

3.15 The ascending insertion search fails to return the `tail`

- Target: `sorted_vaults`
- Category: Business Logic
- Severity: Low
- Likelihood: Low
- Impact: Low

Description

The `sorted_vaults::find_insert_position_ascending` search algorithm fails to return the `tail` position:

```
fun find_insert_position_ascending<CoinType>(
    nominal_cr: u128,
    start_id: Option<address>,
): (Option<address>, Option<address>) acquires SortedVaults {
    if (empty<CoinType>()) {
        return (option::none(), option::none())
    };

    // check if the insert position is after the tail
    let tail = get_last<CoinType>();
    if (option::is_none(&start_id)) {
        let tail_nominal_cr
    = get_nominal_cr<CoinType>(*option::borrow(&tail));
        if (tail_nominal_cr ≥ nominal_cr) {
            return (option::none(), tail)
        }
    };
     ...
```

Impact

The position returned by `sorted_vaults::find_insert_position_ascending` does not correspond with a valid insertion position.

Fortunately, however, in the current implementation this never happens because `find_insert_position_ascending` is never passed a `start_id` that is `none`.

Recommendations

We strongly advise that this coding mistake be fixed. If future iterations extend the current codebase and make calls to this function by passing `start_id` that is `none`, it could have material implications for the protocol.

Remediation

Thala Labs has made extensive changes to the sorted vaults implementation. The function containing this bug was removed by commit 6de6e464. It appears as though the issue has been resolved, but complete verification of the new mechanics would require a separate review.

3.16 Instances of `none` in `VaultStore.vault`

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

Calls to `vault::close_vault` leave the vault store with a `none` vault:

```
// clear resource
let vault_store = borrow_global_mut<VaultStore<CoinType>>(account_addr);
vault_store.vault = option::none();

withdrawn_collateral
```

Impact

Closing a vault can cause the following getters to fail with an unclear error message:

```
public fun max_borrow_amount<CoinType>(addr: address): u64 acquires
    VaultStore {
    assert_vault_store<CoinType>(addr);

    let vault_store = borrow_global<VaultStore<CoinType>>(addr);
    let vault = option::borrow(&vault_store.vault);
    max_borrow_amount_given_collateral<CoinType>(vault.collateral)
}

public fun collateral_amount<CoinType>(addr: address): u64 acquires
    VaultStore {
    assert_vault_store<CoinType>(addr);

    let vault_store = borrow_global<VaultStore<CoinType>>(addr);
    let vault = option::borrow(&vault_store.vault);
    (vault.collateral)
}

public fun debt_amount<CoinType>(addr: address): u64 acquires VaultStore {
    assert_vault_store<CoinType>(addr);
```

```
    let vault_store = borrow_global<VaultStore<CoinType>>(addr);
    let vault = option::borrow(&vault_store.vault);
    (vault.debt)
}
```

Closed vaults, and hence vault stores with `none` vaults, remain in the `SortedVaults` struct. It appears as though this should not cause an issue in the current implementation. It is strongly advised that Thala Labs avoid having fields with `none` values as this can lead to unexpected failures with unclear error messages.

Recommendations

Consider removing `VaultStores` with `none` vaults from the `SortedVaults` struct. Alternatively, remove the `VaultStore` for closed vaults entirely.

Additionally, inlcude assertion checks for vaults that are not `none` in the above functions.

Remediation

In commit 6de6e464, Thala Labs added a check to ensure that no vaults with zero collateral are added to the system. However, verifying that subsequent withdrawals cannot leave empty vaults in the system requires a separate review of architectural changes.

3.17 Missing assertion checks for oracle initialization

- **Target**: `vault`
- **Category**: Business Logic
- **Likelihood**: Low
- **Severity**: Low
- **Impact**: Low

Description

There are no checks in place to enforce that `oracle::set_price<CoinType>` has been called for a given `CoinType` prior to calling `vault::initialize<CoinType>`:

```
public entry fun initialize<CoinType>(manager: &signer) {
    assert!(signer::address_of(manager) == @thala_protocol_v1,
    ERR_INVALID_MANAGER);
    assert!(manager::initialized(), ERR_UNINITIALIZED_MANAGER);
    assert!(!exists<CollateralState<CoinType>>(@thala_protocol_v1),
    ERR_INITIALIZED_COLLATERAL);

    stability_pool::register_collateral<CoinType>(manager);
    sorted_vaults::initialize<CoinType>(manager);
    move_to(manager, CollateralState<CoinType> {
        total_collateral: 0,
        total_debt: 0,
    });
}
```

Impact

If the `PriceStore<CoinType>` has not been initialized via a call to `oracle::set_price<CoinType>`, calls to vault functions will fail with unclear error messages.

Recommendations

Force the oracle to be set up prior to allowing calls to `vault::initialize<CoinType>` by including the following assertion check:

```
public entry fun initialize<CoinType>(manager: &signer) {
    assert!(signer::address_of(manager) == @thala_protocol_v1,
    ERR_INVALID_MANAGER);
    assert!(manager::initialized(), ERR_UNINITIALIZED_MANAGER);
```

```
    assert!(!exists<CollateralState<CoinType>>(@thala_protocol_v1),
    ERR_INITIALIZED_COLLATERAL);
    assert!(!exists<PriceStore<CoinType>>(@thala_protocol_v1),
    ERR_UNINITIALIZED_PRICESTORE);

    stability_pool::register_collateral<CoinType>(manager);
    sorted_vaults::initialize<CoinType>(manager);
    move_to(manager, CollateralState<CoinType> {
        total_collateral: 0,
        total_debt: 0,
    });
}
```

Remediation

Thala Labs added checks in commit 853c1f03 to ensure that the oracle is initialized before vault initialization. However, verifying that these checks are secure would require a separate review of extensive changes to both the oracle and the new initialization sequence.

4 Formal Verification

The Move language is developed alongside the Move specification language, which allows for formal specifications to be written and verified by the Move prover. The project did not include any such specifications, so we provided Thala Labs with some ourselves.

Writing specifications against this project has a number of obstacles. First, the dependencies were fairly out of date, which presented problems for verification. Specifically, the version of the Aptos framework used was incompatible with the current state of the prover, so running the tool required an upgrade. Additionally, the U256 module uses bitwise operators, which are unsupported by the Move prover. In older versions, this module would prevent the prover from being run at all; it now includes specifications that mark problematic functions as opaque.

This issue with bitwise operators presented another challenge. The source of this protocol also utilized bitwise operators in a number of places. For instance,

```
/// returns a to the power of b.
public fun exp(a: u64, b: u8): u64 {
    let c = 1;

    while (b > 0) {
        if (b & 1 > 0) c = c * a;
        b = b >> 1;
        a = a * a;
    };
    c
}
```

We recommend that Thala Labs use the modulo operator over `& 1` in this and other instances.

Finally, the state of the prover and the Aptos framework are not quite robust; they are not without bugs. In order to let the prover run on `stability_pool.move`, it is necessary to make minor changes to framework specifications. Additionally, the prover will not work on `vault.move` or `sorted_vaults.move` at all, as it consumes far too much memory.

Despite these challenges, the prover still presents a powerful way to verify the behavior of certain functions. The following is a sample of some specifications we have

provided; we strongly recommend that the Thala Labs team add more as well.

4.1 thala_protocol_v1::apd_coin

This module is fairly simple. Here is a basic specification that checks the behavior of `mint`:

```
spec mint {
    /// Only aborts if uninitialized.
    aborts_if !has_capabilities();

    /// Minted value must equal amount.
    ensures result.value == amount;
}
```

For this module, we provided specifications for all functions: `initialization`, `mint`, `burn`, and `initialized`.

4.2 thala_protocol_v1::oracle

The `oracle` module is also straightforward to prove. We can show that each of its functions performs necessary checks and changes prices correctly.

```
spec fun has_store<CoinType>(): bool {
    exists<PriceStore<CoinType>>(@thala_protocol_v1)
}

spec fun get_store<CoinType>(): PriceStore<CoinType> {
    global<PriceStore<CoinType>>(@thala_protocol_v1)
}

spec set_price {
    /// Can only be called by @thala_protocol_v1.
    aborts_if signer::address_of(oracle_manager) ≠ @thala_protocol_v1;

    /// Even if the resource did not exist before, it should exist after.
    ensures has_store<CoinType>();

    /// Prices should be properly set.
```

```
    ensures get_store<CoinType>().numerator == numerator;
    ensures get_store<CoinType>().denominator == denominator;
}

spec price_of {
    /// Should abort if and only if the resource does not exist.
    aborts_if !has_store<CoinType>();

    /// Returned prices should reflect stored values.
    ensures result_1 == get_store<CoinType>().numerator;
    ensures result_2 == get_store<CoinType>().denominator;
}
```

4.3 `thala_protocol_v1::params`

This module is similar to `thala_protocol_v1::oracle`. Here is what one of the function specifications looks like:

```
spec set_params {
    /// Only be called by @thala_protocol_v1 can set params.
    aborts_if signer::address_of(manager) ≠ @thala_protocol_v1;

    /// Param store should be created if it did not exist.
    ensures has_store<CoinType>();

    /// Parameters should be properly set.
    ensures get_store<CoinType>().mcr_numerator == mcr_numerator;
    ensures get_store<CoinType>().mcr_denominator == mcr_denominator;
    ensures get_store<CoinType>().redeem_fee_numerator
    == redeem_fee_numerator;
    ensures get_store<CoinType>().redeem_fee_denominator
    == redeem_fee_denominator;
}
```

The others are also closely analogous to those in the `oracle` module.

4.4 `thala_protocol_v1::stability_pool`

This specification verifies that `register_collateral` is idempotent; that is, after the first initialization, additional calls should not mutate relevant values in global state.

```
spec register_collateral {
    /// Can be called only by @thala_protocol_v1.
    aborts_if signer::address_of(account) ≠ @thala_protocol_v1;

    /// Must abort if stability pool does not exist.
    aborts_if !has_pool();

    /// After this is called, `DistributedCollateral<CoinType>` must exist
    /// if it did not already.
    ensures has_collateral<CoinType>();

    /// If the resource already exists, then its collateral and shares
    /// should not change.
    ensures old(has_collateral<CoinType>()) ⟹ (
        get_collateral<CoinType>().collateral ==
        old(get_collateral<CoinType>().collateral)
    );

    ensures old(has_collateral<CoinType>()) ⟹ (
        get_collateral<CoinType>().shares ==
        old(get_collateral<CoinType>().shares)
    );
}
```

The following specification verifies that `deposit_apd` correctly changes the APD in the stability pool. A similar specification was provided for the withdrawal as well.

```
spec deposit_apd {
    /// One possible abort happens when `deposited_apd_amount + amount`
    /// exceeds `MAX_U64`. However, `iterable_table` does not provide
    /// enough primitives for actually accessing the previous value from
    /// the spec.
    pragma aborts_if_is_partial = true;

    /// Aborts if the stability pool does not exist.
    aborts_if !has_pool();
```

```
    aborts_if !has_pool_events();

    // The `iterable_table` implementation from Aptos does not yet provide
    // sufficient primitives for confirming that accounting is done
    // correctly in `stability_pool.deposits`.

    /// The pool's APD should increase by the deposited amount.
    ensures coin::value(get_pool().apd) == (
        old(coin::value(get_pool().apd)) + coin::value(apd)
    );
}
```

Because `distribute_collateral_and_request_apd` features fairly complex looping behavior, fully specifying it was out of scope for the audit. However, we are still able to make certain guarantees about how the pool APD changes and what APD is returned.

```
spec distribute_collateral_and_request_apd {
    pragma aborts_if_is_partial = true;

    /// Aborts if not properly initialized.
    aborts_if !has_pool();
    aborts_if !has_collateral<CoinType>();

    /// Should abort if there is insufficient APD to give.
    aborts_if coin::value(get_pool().apd) < requested_apd;

    /// Pool APD should decrease by requested APD.
    ensures coin::value(get_pool().apd) == (
        old(coin::value(get_pool().apd)) - requested_apd
    );

    /// Requested APD should be returned.
    ensures result.value == requested_apd;
}
```

5 Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment.

5.1 Stablecoin algorithm robustness

Thala Labs's APD coin is an algorithmic stablecoin. The code reviewed implements several mechanisms intended to support the robustness of the APD peg:

- diversification across collateral asset types through multiple single-asset collateral vaults
- a liquidation mechanism to remove undercollateralized vaults from the protocol
- a stability pool providing APD liquidity for vault liquidations
- a redemption mechanism that allows APD holders to pay off debt on vaults in exchange for collateral (regardless of whether they are undercollateralized)

However, the current implementation is missing several critical features likely to contribute to the robustness of APD that are also covered in Thala Labs's documentation:

- multi-collateral vaults and priority liquidation mechanisms
- controlling for redemptions of specific collateral types
- blocking redemptions based on the system's total collateralization ratio
- emergency shutdown process
- use of interest rates to incentivize the addition and removal of specific collateral types
- governance mechanism to control important protocol parameters
- yield-bearing APD
- governance mechanisms

The following points pertain to the central topic of stablecoin robustness.

Thala Labs's documentation mentions that "APD is designed so that it will consistently trade at the $1 peg.... Users will only be able to mint ("borrow") a value of stablecoins less than the value of the collateral, such that the value of collateral in the system always exceeds the value of stablecoin in circulation." And also, "A liquidation process will occur when the collateralization of a user's vault falls under what is deemed safe for the backing of APD."

In the current implementation, this statement is not necessarily true because there are no incentives for users to call liquidate under all market scenarios. For example,

APD depositors in the stability pool are incentivized to call liquidate when it results in a profit. However, if the price of collateral falls to low, there are no incentives to call liquidate because this can result in a financial loss. Furthermore, if there are insufficient APDs to cover entire liquidations, the system may still be in an undercollateralized state after a liquidation call.

Regarding redemptions, it is not clear how controlling redemption on specific assets will be used to manage the relative ratios of collateral assets in the system. Furthermore, based on the description from the documentation, it creates an area of active management for the protocol managers and hence contributes to centralization risk. Lastly, it creates a challenging situation for auditing the robustness of the stablecoin protocol. We do not know what the process will be for managing collateral assets nor can make any assurances for the protocol managers to actually follow this process.

Currently there is no implementation of the emergency shutdown process. Based on compelling arguments in Thala Labs's documentation, this appears to be a critical component contributing to the overall robustness of the protocol. However, we are unable to comment on its potential impact.

We are unable to make any comments on multi-collateral vaults or including yield-generating mechanisms for APD as they are not implemented.

The mechanism outlined for collateral removal — "The reduction of a given collateral will be incentivized by a rapid increase in the interest rate, such that the debt (and thus the vault position) becomes unattractive to maintain. Thala will use all avenues to communicate with relevant stakeholders of potential changes to ensure all users have adequate time to close their positions" — is not implemented. As illustrated in the Findings section, this is a critical functionality of the protocol, which must be included in order to ensure robustness of the stablecoin.

In light of the above, there are several critical components contributing to the robustness and stability of the stablecoin that we are unable to comment on as they have not been implemented in the current code base. As it currently stands, the protocol is incomplete and there is insufficient functionality to determine whether the stablecoin will be robust. Additionally, it is not clear from the documentation or the test cases how all of the intended functionality will come together to ensure a robust stablecoin.

We would like to advise Thala Labs to exercise due care when incorporating governance mechanisms. For example, "Governance voting, or admin functionality, caps the amount of APD to be minted from XYK collateral at 100,000,000" indicates that governance mechanisms could influence critical components affecting the dynamics of the protocol. The rights of governance stakeholders should be carefully considered because they could unduly affect the robustness of the stablecoin. If token-weighted voting mechanisms are used, it could be possible for a malicious actor to manipulate the system for their gain and/or the loss of other users.

As it currently stands, the only way to mint APD is through the deposit of collateral. And the only way to stabilize the system for liquidation events is through the deposit of APD into the stability pool. It is unclear then how system liftoff will be achieved. It is currently assumed that a large LP will provide the seed collateral required to generate APD, but how the mechanics of all this will work out is unclear.

Ultimately, assessing the robustness of protocols like this requires complex and thorough economic simulation. For example, it is important to explore how the incentives for APD stability pool depositors change under volatile market conditions. In the current implementation, when there can be losses to stability pool depositors, it is not clear how APD depositors would behave.

In response, Thala Labs commented,

> A number of quality suggestions were made in this section to enhance the robustness of APD. It is worth noting that these suggestions have been taken to heart. Notably, the team is working on a governance module to control important thala protocol parameters. The team is also working on yield generation, and more advanced emergency shut down processes. In the interim, freezing collateral types is functional in the event a shut down of a collateral type is needed.

5.2 Future governance mechanisms

It is currently unclear how future governance structures using the `THL` token defined in `thl_coin` will interact with the protocol. Reviewing the current version of the documentation, it appears the intention is to allow holders of `THL` to vote on important aspects of the protocol such as which collateral assets to include.

We caution that the use of tokens for driving protocol governance be done with caution. For example, the use of token-weighted voting mechanisms can lead to governance takeovers and adverse implications for the protocol. If a user were able to get a majority voting stake, they could pass a measure to approve the including of a `CoinType` for which they have complete control. They could then manipulate the price and trigger liquidation events resulting in losses for stability pool depositors. Alternatively, they could manipulate the price and trigger profitable liquidation for themselves as APD depositors in the stability pool.

Thorough consideration should be given to the aspects of the protocol controlled by such governance mechanisms.

To this, Thala Labs responded,

> Valid concerns were made about the prospect for takeovers in a token-governed economy. Thala will not launch with governance support initially, however is keeping governance heavily in mind. Thala will introduce token governance gradually while iteratively improving its governance proceedings to minimize concerns around this issue.

5.3 Test case coverage

Unit tests

We highly recommend adding unit tests across the codebase. In general they are missing or have incomplete coverage for all modules. The only module with a well-structured suite of unit tests is the `sorted_vault`.

While we have strongly advised against using experimental data structures and unaudited libraries, we emphasize that it is an absolute must that if any of these are going to be used in the final codebase, they should be thoroughly testsed.

There should be tests of the scripts `vault_scripts` and `stability_pool_scripts`.

Integration tests

There are two tests on the liquidation function, one for a completely covered liquidation and one where the APD stability pool has insufficient APD. We, however, suggest the test suite be expanded to cover cases where there are multiple types of collateral assets being supported. And further, we suggest using tests that illustrate the mechanism and the incentives for liquidation that keep the value of the collateral above APD in the system across market scenarios. Integration tests that demonstrate this would go a long way in assuring APD holders and protocol investors that this critical requirement for protocol solvency is met.

It is strongly advised that Thala Labs develop integration test cases that show how the redemption mechanism supports the validity of the following statement from the Thala Labs documentation: "Users with an open vault will be given an option to redeem APD for $1 (minus redemption fee) worth of collateral, such that the effective price floor is $1." This can help identify cases in which this might not hold true and could therefore provide valuable insight into necessary protocol design changes.

In general, there are insufficient negative tests in the integration test suites.

For a complex protocol that relies on many mechanisms to be in place in order to keep APD stabilized, we highly encourage that Thala expand their test suite to demonstrate

that the desired dynamics exist.

Response

Thala Labs explained,

> Valid comments were made around the need for enhanced test coverage. Notably, we have created tickets for adding tests for vault and stability pool scripts. We have also added tickets for adding integration tests for vault profitability.

5.4 Use of experimental data structures

The current implementation relies on Aptos's `iterable_table`, a data structure that wraps `table_with_length` in the standard library to add an API for iteration. The version of the Aptos framework used in this protocol is fairly out of date; in that previous version, `iterable_table` was a member of `aptos_stdlib`, but it has since been moved to `move-examples/data_structures` for being experimental, exploratory work. See this pull request removing it from the framework and this pull request adding it into the examples.

We recommend that Thala Labs use nonexperimental data structures that have been approved as part of `aptos_stdlib`. When unable to do so, we suggest that Thala develop their own data structures and explicitly submit experimental data structures such as these for audit. It is important to emphasize that the `iterable_table` was not in scope for this audit and has not been reviewed.

In response, Thala Labs commented,

> `iterable_table` is no longer used in latest version of code.

These changes involve substantial modifications to the architecture, and Zellic has not reviewed them at this time.

5.5 Use of Pontem's U256

The project relies on Pontem Network's U256 implementation. This presents a challenge when analyzing the mathematics involved. First, it means that the security and robustness of the protocol is dependent on the correctness of U256 — the library has not yet been audited. Second, the use of this library makes using the prover difficult.

Bitwise operators are currently unsupported by the Move prover, so it is unable to reason about its correctness. Consider this very simple example:

```
use u256::u256::{Self, U256};

fun one_plus_one(): U256 {
    let one = u256::from_u64(1);
    u256::add(one, one)
}

spec one_plus_one {
    ensures u256::as_u64(result) == 2;
}
```

The prover is unable to verify the true claim.

At minimum we recommend that Thala Labs use a library for U256 that has been previously audited or submit their desired U256 library for audit. It is important to emphasize that this library was not in scope for this audit.

In response, Thala Labs commented,

> Pontem's U256 is no longer used. We switched to use Aptos native u256 after it's introduced in latest Aptos core.

These changes involve substantial modifications to the architecture, and Zellic has not reviewed them at this time.

5.6 Code maturity

In general the code is well-written and easy to understand. However, there are several areas that can be improved.

Use of `get_mcr`

For instance, consider the following computation in `vault::liquidate<CoinType>`:

```
let max_borrow_amount = max_borrow_amount<CoinType>(vault_addr);
let (collateral_amount, debt_amount)
    = collateral_and_debt_amount<CoinType>(vault_addr);
```

```
assert!(debt_amount > max_borrow_amount, ERR_MCR_MET);

let collateral_in_apd_unit =
    fixed_point::from_u64(convert_to_apd_unit<CoinType>(collateral_amount));

let max_borrow_amount = fixed_point::from_u64(max_borrow_amount);
let debt_amount = fixed_point::from_u64(debt_amount);

// SHARE_DECIMAL_CONSTANT is used to represent fixed point decimal number
// ex) SHARE_DECIMAL_CONSTANT = 1,000,000
// 1.01 ⟹ 1,010,000 in SHARE_DECIMAL_CONSTANT representation
//
// Calculate collateral ratio and min collateral ratio in fixed point
    decimal number
// with SHARE_DECIMAL_CONSTANT
let mcr = fixed_point::div(collateral_in_apd_unit, max_borrow_amount);
let cr = fixed_point::div(collateral_in_apd_unit, debt_amount);
```

Here, MCR is computed by dividing `collateral_in_apd_unit` by `max_borrow_amount`. The value of `max_borrow_amount` comes from the `vault::max_borrow_amount<CoinType>` function, which in turn calls `vault::max_borrow_amount_given_collateral<CoinType>`:

```
public fun
    max_borrow_amount_given_collateral<CoinType>(collateral: u64): u64 {
    let normalized_collateral
    = fixed_point::from_u64(scaled_by_apd<CoinType>(collateral));
    let price = get_oracle_price<CoinType>();
    let mcr = get_mcr<CoinType>();

    fixed_point::to_u64(fixed_point::div(fixed_point::mul(normalized_collateral,
    price), mcr))
}
```

This function simply computes `max_borrow_amount` as `normalized_collateral * price / mcr`. However, the numerator of this division is simply the collateral price in APD, so when `vault::liquidate<CoinType>` divides the collateral price in APD by the `max_borrow_amount`, the result should be the same as `vault::get_mcr<CoinType>`.

The only other place that `max_borrow_amount` is used in `vault::liquidate` is in checking that the given vault is undercollateralized. However, this can also be done by directly comparing MCR with CR, avoiding the computation of `max_borrow_amount` entirely.

This creates extra computation that is confusing and inefficient.

Copy and pasting of functions with partial symmetries

The function `sorted_vaults::find_insert_position_descending` has clearly been copied over and used for `sorted_vaults::find_insert_position_ascending`. This process was done without sufficient care and resulted in the issue covered in finding 3.15. The condition for the while loop also does not seem to have been adjusted, and there are still comments referring to a `descending` search. While this does not appear a security issue, we suggest Thala Labs review and clean this function where appropriate.

Summary

Overall, the codebase is in a state of development. For example, the governance mechanism is incomplete. Additionally, future commits in the repo point to changes to the vault structures, reward allocation mechanisms, powers of the manager, and operational state of the system.

We strongly advise that Thala Labs ensures that the final protocol deployed to mainnet is based on a commit that had reached code freeze at the time of the audit.

5.7 Scripts

The codebase includes two modules that expose a simpler interface for `thala_protocol_v1::vault` and `thala_protocol_v1::stability_pool`. These are in the files `scripts/vault_scripts.move` and `scripts/stability_pool_scripts.move` respectively. Among other things, these modules perform the work of withdrawing and depositing APD and collateral as well as executing necessary initialization steps.

We notice that these modules do not provide a complete interface for interacting with the protocol. For instance, if a user wanted to open a vault, they would still need to interact with the underlying `vault` module in order to determine how much APD to borrow for a given amount of collateral. We would expect that these scripts provide sufficient primitives for a standard user, but they do not at the moment.

From a security perspective, the correctness of these modules is less critical. They do not have additional permissions and cannot themselves introduce significant issues for the protocol. However, bugs in these scripts would still be a problem; they are published by the protocol, so users would somewhat trust that they work properly. Currently, not a single function in these interfaces is tested. We recommend that Thala Labs significantly improve the coverage of these modules.

Thala Labs noted this discussion point and commented:

> We have engineering tasks in place to increase the surface of our script interfaces, so that users can do most/all protocol operations via scripts

5.8 Missing validation checks

Thala Labs should make efforts to improve data validation checks across the codebase. For example, there are no validation checks to ensure a vault is open when calls are made to `vault::liquidate`. Additionally, there are no simple validation checks in `stability_pool::withdraw_apd` to ensure that the `account_addr` exists in `StabilityPool.deposits`. Validation checks should also be incorporated into nested functions, such as ensuring that `prev` and `next` do indeed exist in `sorted_vault.nodes` as shown in the code sample below:

```
else {
    let prev = *option::borrow(&prev);
    let next = *option::borrow(&next);

    let prev_node = table_with_length::borrow<address,
    Node>(&sorted_vaults.nodes, prev);
    let next_node = table_with_length::borrow<address,
    Node>(&sorted_vaults.nodes, next);

    (
        option::contains(&prev_node.next, &next) &&
        prev_node.nominal_cr ≥ nominal_cr &&
        nominal_cr ≥ next_node.nominal_cr
    )
}
```

6 Audit Results

At the time of our audit, the code was not deployed to mainnet Aptos.

During our audit, we discovered 17 findings. Of these, three were of critical risk, one was of high risk, seven of medium risk, and six of low risk. Thala Labs acknowledged all findings, and fixes have been made or are pending.

Thala Labs has made considerable architectural changes to the codebase based on our recommendations for addressing the findings identified in this report. In general, these changes they have made require an additional separate review in order to assure the security and integrity of the protocol.

6.1 Disclaimers

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any additional code added to the assessed project after the audit version of our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but it may not be tested or functional code.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.