IPOR

protocol core

by Ackee Blockchain

25.9.2023

Contents

1. Document Revisions

0.1	Draft report	1.8.2023
1.0, 1.1	Final report	1.8.2023
1.2	Revision 1.2 + Fix review	30.8.2023
1.3	Revision 1.3 + Fix review	22.9.2023
1.4	Fix review 1.4	25.9.2023

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain

Ackee Blockchain is an auditing company based in Prague, Czech Republic, specializing in audits and security assessments. Our mission is to build a stronger blockchain community by sharing knowledge – we run free certification courses School of Solana, Summer School of Solidity and teach at the Czech Technical University in Prague. Ackee Blockchain is backed by the largest VC fund focused on blockchain and DeFi in Europe, RockawayX.

2.2. Audit Methodology

1. **Technical specification/documentation** - a brief overview of the system is requested from the client and the scope of the audit is defined.
2. **Tool-based analysis** - deep check with automated Solidity analysis tools and Woke is performed.
3. **Manual code review** - the code is checked line by line for common vulnerabilities, code duplication, best practices and the code architecture is reviewed.
4. **Local deployment + hacking** - the contracts are deployed locally and we try to attack the system and break it.
5. **Unit and fuzz testing** - run unit tests to ensure that the system works as expected, potentially write missing unit or fuzz tests.

2.3. Finding classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**-**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or configuration (such as deployment scripts, compiler configuration, use of multi-signature wallets for owners, etc.), but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or configuration (see above) was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review team

Member's Name	Position
Lukáš Böhm	Lead Auditor
Michal Převrátil	Auditor
Jan Kalivoda	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We've put our best effort to find all vulnerabilities in the system, however our findings shouldn't be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

IPOR protocol (Inter Protocol Over-block Rate) consists of two parts. The first one is the IPOR index.

The index functions as an interest rate benchmark, aggregating the lending and borrowing interest rates from multiple DeFi lending platforms to provide an objective view of the current market status. The logic lies off-chain. To access the index, a user must call the IPOR Oracle contract.

The second part is the IPOR automated market maker. Unlike the standard AMM, the IPOR AMM allows users to open/close swaps and speculate on interest rates. Swaps can be opened in two different directions:

- pay a fixed interest rate and receive a floating interest rate,
- pay a floating interest rate and receive a fixed interest rate.

The vital role is a liquidity provider. Liquidity providers provide liquidity to the AMM and receive a share of the swap fees. Liquidity in the pool is automatically rebalanced between the protocol treasury and the asset management, which deposits assets into Aave or Compound to earn interest.

Revision 1.0

IPOR engaged Ackee Blockchain to perform a security review of the IPOR protocol with a total time donation of 52 engineering days in a period between June 1 and July 28, 2023, and the lead auditor was Lukáš Böhm.

The audit was initially performed on the commit `680c80f`. The list of all audited contracts can be found in Appendix B.

We began our review using static analysis tools Woke. We then took a deep dive into the logic of the contracts and performed a manual code review. In parallel with the manual review, we created comprehensive fuzz tests for the most critical parts of the system, such as opening/closing swaps and providing liquidity. For testing and fuzzing, we have involved Woke testing framework. The test simulates the real-world behavior with a complete deployment of the protocol and with a focus on unexpected call sequences and edge case values. For more information about fuzz testing, see Appendix C.

During the fuzz testing, we found several issues (see the list of issues found by fuzz test in the last paragraph of Appendix C) that were reported to the client and immediately fixed. The cooperation with the team and their ability to quickly react was crucial to fuzz testing, where a correctly working protocol is necessary as it allows us to find more issues and edge cases. Fuzz testing at its final stage ran on newer commits (with bug fixes) than the initial one. This commit will be mentioned in revision 1.1 of this report with a more detailed description of the fixes and code changes. During the review, we paid particular attention to:

- ensuring the arithmetic of the system is correct,
- detecting possible financial attacks such as flash loans,
- ensuring the protocol's behavior stays consistent in edge case scenarios,
- checking access control mechanisms,
- detecting possible reentrancies in the code,
- checking proper storage handling,
- comparing the code logic to the documentation,
- looking for common issues such as data validation.

IPOR provides good documentation; however, some parts were outdated and described the previous protocol version. During the audit process, as part of the updated codebase, the client provided updated interfaces with detailed NatSpec documentation, which provided a good overview of the functions. Nevertheless, some more complex functions should have included more detailed descriptions of specific lines of code.

The overall protocol structure is well-designed. The entry point of the protocol is the *router* contract that forwards calls to *service* contracts to perform state-changing actions or to *lens* contracts to read the storage. The math behind the protocol is complex, and we found many issues which caused the protocol not to behave as expected. Even though these issues do not directly threaten the protocol, they can cause unexpected behavior and financial losses. Fixing some of the issues is non-trivial. For those reasons, we cannot recommend deployment, and we advise the client to perform another audit revision after the issues are fixed.

Our review resulted in 28 findings, ranging from Info to Critical severity. The number of findings takes into account Revision 1.0 and Revision 1.1 as they were performed during one review.

Ackee Blockchain recommends IPOR:

- update the test suite to prevent mathematical issues such as overflow and underflow,
- update the documentation to reflect the current state of the protocol,
- address all other reported issues.

See Revision 1.0 for the system overview of the codebase.

Revision 1.1

As mentioned in Revision 1.0, we found several issues that were reported to the client and immediately fixed during the audit. The review then continued on the last provided commit: `87c4d345`. Revisions 1.0 and 1.1 were performed during one review.

See Revision 1.1 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.2

IPOR provided an updated codebase with fixes for several reported issues on the date 22nd of August, 2023. However, not all the issues were fixed. See the table for the current status of issues. Together with fixes, new contracts for staked ETH pool were introduced. The review was performed on the commit `1847f3e6`.

In the first part of the review, we reviewed all the fixes and ensured they corresponded with our recommendations.

In the second part of the review, we manually reviewed the new codebase containing four new contracts and changes in several older contracts. The new codebase allows providing liquidity with Ether, Wrapped Ether, and Staked Ether. During the review, we identified four findings, ranging from info to low severity. The quality of the code is good and it is easy to read. Interface contracts contain NatSpec In-code documentation.

Before the commit for Revision 1.2 was provided, IPOR discovered a low issue in the contract AssetManagement in the function `withdrawAll()`. The variable `ivTokenTotalSupply` is not updated correctly, which results in incorrect data in the event.

See Revision 1.2 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.3

IPOR provided an updated codebase with fixes and a few code changes on the date 18th of September, 2023.

The review was performed on the commit `553e1c7` with a time donation of 3 MD.

We began the review with a focus on fixes of previously discovered issues. Then, we focused on the new changes in the codebase. After the manual review, we updated the fuzz tests to be relevant to the new codebase.

During the fuzzing process, we discovered several scenarios that ended up with different errors in the protocol. We identified root cases for some of the errors. However, we did not have enough time donation to debug all of them. Thus we decided to report all the error messages that the protocol is returning without further investigating the root causes.

During the manual review, we identified three findings, ranging from warning to medium severity. Two higher-severity findings M6 and L6 were discovered during the fuzzing process.

See Revision 1.3 for the review of the updated codebase and additional information we consider essential for the current scope.

Revision 1.4

IPOR provided an updated codebase with fixes on the commit: `3d99b22`.

No new changes were introduced except for updating the values of the spread slope.

See the summary of the findings for the current status of issues.

See Revision 1.4 for the review of the updated codebase and additional information we consider essential for the current scope.

Final note

In the given time donation and after all reported issues were fixed, the auditing team doesn't see any issue that would lead to a loss of funds or any other catastrophic consequences. The confidence of the auditing team is based on a manual review and a fuzz testing model.

The IPOR team sticks with good practices. The code quality is high, the code contains NatSpec documentation, and the general documentation is comprehensive. IPOR team also provided many diagrams and mathematical equations, which made the audit process more effective.

We cannot rule out the chance of DoS of the protocol caused by some edge case conditions. But it is not directly related to security but rather the mathematical and architectural complexity of the protocol.

The next step to enhance confidence in the protocol is to extend the fuzz test, and model all the parts of the protocol that are not included in the current fuzz test (liquidity mining, ETH staking, governance, DSR).

4. Summary of Findings

The following table summarizes the findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- a `Description`,
- an `Exploit scenario`,
- a `Recommendation` and if applicable
- a `Fix`.

There might often be multiple ways to solve or alleviate the issue, with varying requirements regarding the necessary changes to the codebase. In that case, we will try to enumerate them all, clarifying which solves the underlying issue better (albeit possibly only with architectural changes) than others.

	Severity	Reported	Status
C1: Profit & loss accounted twice when unwinding	Critical	1.0	Fixed
H1: Unwinding formula	High	1.0	Fixed
H2: Broken reentrancy lock	High	1.0	Fixed
M1: `INTEREST_FROM_STRATEGY_BELOW_ZERO` reverts	Medium	1.0	Fixed
M2: Inaccurate hypothetical interest formula	Medium	1.0	Fixed
M3: Pool contribution is not updated when liquidity is redeemed	Medium	1.0	Fixed

	Severity	Reported	Status
M4: Incorrect event data	Medium	1.0	Fixed
L1: Value in incorrect decimals	Low	1.0	Fixed
L2: Liquidation deposit accounted twice in rebalancing logic	Low	1.0	Fixed
L3: Aave incorrect APY formula	Low	1.0	Fixed
M5: Close swap and redeem transaction reverts	Low	1.0	Fixed
W1: Usage of `solc` optimizer	Warning	1.0	Acknowledged
W2: `SoapIndicatorRebalanceLogic` underflow	Warning	1.0	Fixed
W3: Insufficient data validation in the constructor	Warning	1.0	Acknowledged
W4: Missing array length check in the initialize function	Warning	1.0	Fixed
W5: `_calculateRedeemedCollateralRatio` underflow	Warning	1.0	Fixed
W6: Constant block production relied on	Warning	1.0	Acknowledged
W7: Github secrets leak	Warning	1.0	Fixed
W8: Infinite approval	Warning	1.0	Fixed

	Severity	Reported	Status
I1: Unreachable code	Info	1.0	Fixed
I2: Use `type(uint256).max` instead of integer literal	Info	1.0	Fixed
I3: Duplicated code	Info	1.0	Fixed
I4: Redundant require	Info	1.0	Fixed
I5: Using magic numbers	Info	1.0	Fixed
H3: Unwinding fee accounted twice in `liquidityPool` balance	High	1.1	Fixed
M6: Unwinding fee normalization	Medium	1.1	Fixed
W9: Missing swap direction validation	Warning	1.1	Fixed
I6: Use `forceApprove` instead of `safeApprove`	Info	1.1	Fixed
M7: No data validation while setting `redeemFeeRateEth`	Low	1.2	Fixed
I7: User can lose funds if the protocol is used incorrectly	Info	1.2	Fixed
I8: Mixing `_msgSender()` and `msg.sender` across the codebase	Info	1.2	Fixed
I9: Redundant logging of `block.timestamp`	Info	1.2	Fixed
M8: `IPOR_508` reverts during deposit	Medium	1.3	Fixed

	Severity	Reported	Status
L4: Close swap insufficient balance revert	Low	1.3	Fixed
W10: Setting array max index in constructor	Warning	1.3	Fixed

Table 2. Table of Findings

5. Report revision 1.0

5.1. System Overview

This section contains an outline of the audited contracts. Note that this is meant for understandability purposes and does not replace project documentation.

amm/*

The folder contains contracts with swapping logic, pools, treasury, and storage. There are two types of logic contracts:

- Services - entry points for the protocol
- lenses - used for reading data from the storage

AmmCloseSwapService

The logic contract for closing opened swaps for pairs of fixed/floating interest rates in DAI, USDT, and USDC.

AmmOpenSwapService

The logic contract for opening interest rates swaps in two possible directions fixed/floating and floating/fixed in DAI, USDT, and USDC.

AmmPoolsService

The logic contract manages liquidity inside the specific pools (DAI, USDT, USDC). It allows to provide and redeem, and it also holds a logic for rebalancing between an asset management and a treasury.

AmmStorage

The storage contract for the AMM logic contracts.

AmmTreasury

The logic treasury contract holding tokens, depositing or withdrawing them from the asset management.

amm/spread/*

The folder contains contracts for calculating the spread for 28, 60, and 90 days. The logic is accessible through `SpreadRouter` contract and limited by `SpreadAccessControl` contract.

amm/libraries/*

The folder contains libraries for calculating the spread and the swap amount. It also implements custom data structures (structs) for swaps, spreads, risk indicators etc.

governance/*

AmmConfigurationManager

The library implements the logic for managing swap liquidators, addresses appointed to rebalance AMM, and setting pool parameters.

libraries/*

The folder contains libraries with implemented:

- custom errors
- constants
- storage management
- handling *wad* values
- mathematical functions for computing:
    - interest rates

- SOAP
- risk indicators
- spread

oracles/*

Data from oracles are used in the protocol components and third-party applications can also use it.

IporOracle

The contract is used to get updated index data for a given asset. The contract owner, manages listed assets, and the Updaters update asset indexes.

- pausable
- UUPS upgradeable

IporRiskManagementOracle

The contract is used to get data about spread, risk indicators, and rates. contract owner, manages listed assets, and the Updaters update spread, risk indicators, and rates.

- pausable
- UUPS upgradeable

OraclePublisher

The contract is used for interacting with IPOR Oracles. The mapping `updaters` contains addresses of the contracts which are allowed to update Oracle data by performing a `functionCall` (low-level call to a specific address with a data payload)

- pausable
- UUPS upgradeable

The folder **/oracles/libraries** contains libraries for calculating the quasi-IBT price and custom data storage structures.

router/*

AccesControl

The contract uses owner manager and pause manager contracts to handle roles. It also contains non-reentrant logic.

IporProtocolRouter

The contract inherits the logic from AccessControl contract. The contract delegates call using the **fallback** function to the specific protocol components (addresses) set in the constructor. Before an actual call, a function signature is checked to prevent calling a wrong function. The contract also allows batch calls.

- pausable
- UUPS upgradeable

security/*

IporOwnable (IporOwnableUpgradeable)

Abstract contracts for managing ownership of the system. Standard and upgradable version.

OwnerManager

The library is used for managing owners of the system. It uses a two-step ownership transfer.

PauseManager

The library for managing puase guardians of the system. They can be added or removed.

tokens/*

IporToken

The contract implements the native ERC20 token of the protocol. It is used for governance and staking to receive PowerIpor tokens.

IpToken

The token represents a provided liquidity amount into an AMM pool.

IvToken

The token represents a deposit to vault contracts.

vault/*

AssetManagement

The contract manages part of the assets in the system (the second part is in amm treasury contract) by staking and withdrawing from APY strategies. More specifically, Aave and Compound. The more profitable strategy is always chosen while depositing to earn the highest interest. The contract is also capable of rebalancing between strategies.

When the token is deposited/withdrawn, the contract mints/burns the corresponding amount (depends on the current exchange rate, it does not have to be 1:1) of the vault shares, which is represented by IvToken.

- pausable
- UUPS upgradeable

strategy/StrategyCore

The core contract for strategies. It implements modifiers, getters, and setters.

- pausable
- UUPS upgradeable

strategy/StrategyAave

The contract inherits from the core contract. It calculates the current APY of Aave, depositing and withdrawing from the Aave lending pool.

strategy/StrategyCompound

The contract inherits from the core contract. It calculates the current APY of Compound, depositing, and withdrawing from the Compound lending pool.

Actors

This part describes the system’s actors, roles, and permissions.

Owner

The main role in the system. It can manage other roles, unpausing and upgrading contracts, setting up the system parameters and adding/removing assets in the IporOracle contract etc. It is the most powerful role in the system.

Treasury Manager

The role is responsible for setting the treasury address in the StrategyCore contract.

Asset manager

The role can deposit and withdraw assets from strategies. It is managed by the owner of the contract. This role is assigned to the address of the contract AssetManagement

Pause guardian

The role has the privilege to pause a contract. It is managed by the << owner, owner>> of the contract. It is managed by the owner of the contract.

Updaters

The role is responsible for updating data in IPOR Oracle contracts. More than one address can be assigned to this role. The role can update spread, risk indicators, and rates and publish indexes. It is managed by the owner of the contract.

5.2. Trust Model

The trust model is evident by the system's centralized nature, where the system's central part - IPOR index is computed off-chain. Users have to trust IPOR team as they are the owners of the system and the index Providers as they feed the Oreacle with off-chain data.

C1: Profit & loss accounted twice when unwinding

Critical severity issue

Impact:	High	Likelihood:	High
Target:	AmmCloseSwapService	Type:	Logic error

Description

A buyer can always close his swap. Swaps closed before maturity or when 100% profit or loss is not reached are subject to swap unwinding. Unwinding is when a swap in the opposite direction is closed and accounted into the final profit & loss.

The function `calculateSwapUnwindValue` in the `IporSwapLogic` library calculates the unwind value based on the opposite swap.

Listing 1. IporSwapLogic.calculateSwapUnwindValue

```
swapUnwindValue =
    swapPayoffToDate +

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
        (oppositeLegFixedRate * time).toInt256()
    ) -

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
        (swap.fixedInterestRate * time).toInt256()
    );
```

The value returned from this function is later used in the `AmmCloseSwapService` contract as the `swapUnwindAmount` variable in the following way:

Listing 2. AmmCloseSwapService._calculatePayoff

```
    payoff = swapPayoffToDate + swapUnwindValue -
swapUnwindOpeningFeeAmount.toInt256();
```

As a result, the swap payoff to date value is accounted for twice in the final profit & loss.

Vulnerability scenario

A swap with collateral of 1000 USDC is opened. Later, the swap is in 500 USDC profit, and the buyer decides to close the swap. The swap is closed before its maturity, so it is unwinded. Due to the bug, the 500 USDC profit is accounted for twice in the final profit & loss. The buyer receives double the profit he should have received (minus the opposite swap value and an unwinding fee).

Recommendation

Do not add the `swapPayoffToDate` value to `swapUnwindAmount` in the `calculateSwapUnwindValue` function.

Fix 1.1

Both impacted functions underwent significant refactoring, but the `swapPayoffToDate` value is no longer used in the `calculateSwapUnwindValue` function, and the final profit & loss is calculated correctly.

Go back to Findings Summary

H1: Unwinding formula

High severity issue

Impact:	Medium	Likelihood:	High
Target:	IporSwapLogic	Type:	Logic error

Description

A swap unwinding occurs when closing a swap before maturity or 100% profit or loss is reached. A virtual swap in the opposite direction is opened. A difference between interest from the virtual swap and the swap being closed is computed for the remaining time period (until the swap maturity is reached). The difference is then added to the final profit & loss value. In the difference formula, the pay fixed swap interest should always come with the minus sign, while the receive fixed interest should always be with the plus sign. However, the formula in the `IporSwapLogic` library does not differentiate between pay fixed and receive fixed swap kinds. The signs are used so that the opposite direction rate is with the plus sign, and the current swap interest is with the minus sign.

Listing 3. IporSwapLogic.calculateSwapUnwindValue

```
swapUnwindValue =
    swapPayoffToDate +

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
        (oppositeLegFixedRate * time).toInt256()
    ) -

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
        (swap.fixedInterestRate * time).toInt256()
    );
```

This is an issue when a receive-fixed swap is closed because the signs are used reversely. The issue was discovered in parallel with the client.

Vulnerability scenario

A receive-fixed swap is being closed before maturity, and 100% profit or loss is not reached. The swap has a positive profit. Swap unwinding is triggered, and because of the issue, the swap unwinding value is also positive, and the swap buyer receives more money than he should.

Recommendation

Compute the `swapUnwindValue` value so that the pay fixed interest always comes with the minus sign and the receive fixed interest comes with the plus sign.

Fix 1.1

The issue was fixed in the following way:

Listing 4. IporSwapLogic.calculateSwapUnwindValue

```
if (direction == AmmTypes.SwapDirection.PAY_FIXED_RECEIVE_FLOATING) {
    swapUnwindPnlValue =

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
            (oppositeLegFixedRate * time).toInt256()
        ) -

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
            (swap.fixedInterestRate * time).toInt256()
        );
} else if (direction == AmmTypes.SwapDirection.PAY_FLOATING_RECEIVE_FIXED)
{
    swapUnwindPnlValue =
swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
            (swap.fixedInterestRate * time).toInt256()
        ) -

swap.notional.toInt256().calculateContinuousCompoundInterestUsingRatePeriod
MultiplicationInt(
            (oppositeLegFixedRate * time).toInt256()
        );
} else {
    revert(AmmErrors.UNSUPPORTED_DIRECTION);
}
```

Go back to Findings Summary

H2: Broken reentrancy lock

High severity issue

Impact:	Medium	Likelihood:	High
Target:	IporProtocolRouter	Type:	Logic error

Description

The contract IporProtocolRouter checks in the `getRouterImplementation` function if the call is batch operation via the function argument (batchOperation). The `_nonReentrantBefore()` function should be called if the value equals zero. There is only one issue now, but it is potentially more severe.

The code contains only `_nonReentrantBefore` instead of `_nonReentrantBefore()` (missing brackets), so no call is performed. That means there is no reentrancy protection.

The second issue that would be present if the call were correctly written (with brackets) is that the `getRouterImplementation` function is publicly accessible. It means anyone can call it and set a lock without releasing it. This can be done repeatedly and cause a denial of service.

Vulnerable scenario

If a malicious user finds a way to exploit the protocol with reentrancy, he can use this vulnerability to bypass the reentrancy protection.

Recommendation

Add brackets `_nonReentrantBefore` → `_nonReentrantBefore()`. Set the visibility of the function to `internal`.

Fix 1.1

Brackets were added to the `_nonReentrantBefore()` function call. The visibility of the function was set to `internal`.

Go back to Findings Summary

M1: INTEREST_FROM_STRATEGY_BELOW_ZERO reverts

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AmmStorage	Type:	Denial of service

Description

The functions updateStorageWhenWithdrawFromAssetManagement and updateStorageWhenDepositToAssetManagement in the AmmStorage contract are responsible for updating the liquidity pool and asset management balances when withdrawing from asset management or depositing to asset management. Whenever calling one of the functions, the balances are also updated with the interest received from Aave and Compound. In both functions, there is a check that the interest earned since the last update is non-negative.

Listing 5. AmmStorage.updateStorageWhenWithdrawFromAssetManagement

```
uint256 currentVaultBalance = _balances.vault;
// We need this because for compound if we deposit and withdraw we could
get negative intrest based on rounds
require(vaultBalance + withdrawnAmount >= currentVaultBalance,
AmmErrors.INTEREST_FROM_STRATEGY_BELOW_ZERO);

uint256 interest = vaultBalance + withdrawnAmount - currentVaultBalance;
```

Listing 6. AmmStorage.updateStorageWhenDepositToAssetManagement

```
uint256 currentVaultBalance = _balances.vault;

require(currentVaultBalance <= (vaultBalance - depositAmount),
AmmErrors.INTEREST_FROM_STRATEGY_BELOW_ZERO);

uint256 interest = currentVaultBalance > 0 ? (vaultBalance -
currentVaultBalance - depositAmount) : 0;
```

Due to different decimals of Compound tokens, underlying tokens, and other math errors (including rounding errors), the interest may be computed as negative for a short time. This causes the current transaction to revert with the error `AmmErrors.INTEREST_FROM_STRATEGY_BELOW_ZERO`.

Vulnerability scenario

A liquidity provider wants to provide liquidity to the pool. The deposit is large enough to trigger rebalance. Compound currently offers higher APY than Aave, and the deposit is made to Compound. In the `AmmStorage` contract, the interest is computed as negative due to rounding errors, and the transaction reverts. The liquidity provider cannot deposit to the pool.

Recommendation

Allow for negative interest in the `updateStorageWhenWithdrawFromAssetManagement` and `updateStorageWhenDepositToAssetManagement` functions. Check that `_balances.liquidityPool + interest` is non-negative instead. For security reasons, check that the computed interest is not lower than a certain negative threshold.

Fix 1.2

The logic of functions was updated and the issue was mitigated.

Go back to Findings Summary

M2: Inaccurate hypothetical interest formula

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	SoapIndicatorRebalanceLogic	Type:	Logic error

Description

The IPOR protocol uses continuous compounding when computing an interest. The protocol needs to know the SOAP (sum of all payoffs) value when updating risk indicators and when computing ip token exchange rates. Computing SOAP using iteration over all currently opened swaps would be inefficient, so a hypothetical interest value is computed from an average interest rate. However, this approach does not count that the real interest is computed using continuous compounding, which uses the exponential formula. This leads to an insignificant error between real and computed hypothetical interests. The error would be acceptable for swaps opened over 28, 60, or 90 days, but the same error may also cause underflow transaction reverts in the following line:

Listing 7. SoapIndicatorRebalanceLogic.rebalanceWhenCloseSwap

```
uint256 hypotheticalInterestTotal = currentHypoteticalInterestTotal -
interestPaidOut;
```

Vulnerability scenario

Over time, the error accumulates and may lead to the case when `currentHypoteticalInterestTotal < interestPaidOut`. This causes an underflow error, and the close swap transaction reverts. The user is unable to close the swap and withdraw their funds.

Recommendation

Correct the formula for computing the hypothetical interest. Alternatively, since the error is insignificant, the current formula can be used, but ensure the underflow error does not occur.

Fix 1.1

The original formula is still used, but a self-healing mechanism was implemented to prevent the underflow error.

Go back to Findings Summary

M3: Pool contribution is not updated when liquidity is redeemed

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AmmPoolService	Type:	Logic error

Description

When providing liquidity in `AmmPoolService` by calling the function `provideLiquidityDai` (Dai/Usdt/Usdc), the function `addLiquidityInternal` is called to update amm storage:

Listing 8. AmmPoolService._provideLiquidity

```
IAmmStorage(poolCfg.ammStorage).addLiquidityInternal(
    beneficiary,
    wadAssetAmount,
    uint256(ammPoolsParamsCfg.maxLiquidityPoolBalance) * 1e18,
    uint256(ammPoolsParamsCfg.maxLpAccountContribution) * 1e18
);
```

Inside the storage contract, the following mapping is updated: `_liquidityPoolAccountContribution[account]`

However, the mapping is not updated when liquidity is withdrawn by calling the `redeem` functions.

Vulnerability scenario

There are two problematic points:

- calling `getLiquidityPoolAccountContribution` in the contract `AmppPoolsLens` will return the sum of all the LP deposits an account made in the past, even if they have been withdrawn

- The limit of a max contribution is set in every pool. Checking if the limit is exceeded is done in the function `addLiquidityInternal` in the following way:

Listing 9. AmmStorage.addLiquidityInternal

```
uint128 newLiquidityPoolAccountContribution =
_liquidityPoolAccountContribution[account] + assetAmount.toUint128();
require(newLiquidityPoolAccountContribution <= cfgMaxLpAccountContribution,
AmmErrors.LP_ACCOUNT_CONTRIBUTION_IS_TOO_HIGH);
```

Because `_liquidityPoolAccountContribution` is not updated when liquidity is redeemed, a specific account is allowed to deposit less and less liquidity after every redeem till no deposit is allowed.

Recommendation

Update mapping `_liquidityPoolAccountContribution[account]` when liquidity is redeemed. Alternatively, add proper documentation to inform users about the limitation.

Fix 1.1

The client responded that it is a design solution. However, the client decided to remove this limitation.

Go back to Findings Summary

M4: Incorrect event data

Medium severity issue

Impact:	Low	Likelihood:	High
Target:	AmmPoolService	Type:	Logic error

Description

The interface `IAmmPoolsService` defines the following event, where the second element is the address `from` (the address that provides liquidity):

```
event ProvideLiquidity(
    /// @notice moment when liquidity is provided by `from` account
    uint256 timestamp,
    /// @notice address that provides liquidity
    address from,
    ...
    );
```

The event is emitted in the contract `AmmPoolService` in the function `_provideLiquidity`.

Listing 10. AmmPoolService._provideLiquidity

```
function _provideLiquidity(
        address asset,
        address beneficiary,
        uint256 assetAmount
    ) internal {
```

Nevertheless, as we can see in the following three code pieces, the address `beneficiary` is not the address that provides liquidity, but it is the address that receives `ipToken`. The liquidity is provided and transferred from the address of `msg.sender`.

Listing 11. AmmPoolService._provideLiquidity

```
IERC20Upgradeable(poolCfg.asset).safeTransferFrom(msg.sender,
poolCfg.ammTreasury, assetAmount);

uint256 ipTokenAmount = IporMath.division(wadAssetAmount * 1e18,
exchangeRate);

IIpToken(poolCfg.ipToken).mint(beneficiary, ipTokenAmount);

/// @dev Order of the following two functions is important, first
safeTransferFrom, then rebalanceIfNeededAfterProvideLiquidity.
_rebalanceIfNeededAfterProvideLiquidity(poolCfg, ammPoolsParamsCfg,
balance.vault, wadAssetAmount);

emit ProvideLiquidity(
    block.timestamp,
    beneficiary,
    poolCfg.ammTreasury,
    exchangeRate,
    wadAssetAmount,
    ipTokenAmount
);
```

Vulnerable scenario

- Some services can rely on the event data to track the liquidity providers. If the event data is incorrect, the service will not be able to track the liquidity providers correctly.
- Event data are necessary for the postmortem hack analysis. The incorrect data can make tracing more complicated and misleading.

Recommendation

The function should emit an event where `from == msg.sender` and use an additional parameter for the `beneficiary` address.

Fix 1.3

The event was updated to emit the correct data.

Go back to Findings Summary

L1: Value in incorrect decimals

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	AmmCloseSwapService	Type:	Logic error

Description

The value of the variable `wadAmmTreasuryErc20BalanceBeforeRedeem` is acquired using `IERC20.balanceOf`, and so the value is in asset-specific decimals. However, the name of the variables implies the value to be in wads.

Furthermore, the variable `wadAmmTreasuryErc20BalanceBeforeRedeem` is later used as an argument of the `calculateRebalanceAmountBeforeWithdraw` function, but this function expects all parameters in wads. This results in more tokens withdrawn from the asset management than intended when rebalancing between asset management and treasury.

Listing 12. AmmCloseSwapService._transferDerivativeAmount

```
uint256 wadAmmTreasuryErc20BalanceBeforeRedeem =
IERC20Upgradeable(poolCfg.asset).balanceOf(
    poolCfg.ammTreasury
);

if (wadAmmTreasuryErc20BalanceBeforeRedeem <= transferAmountAssetDecimals)
{
    AmmTypes.AmmPoolCoreModel memory model;

    model.ammStorage = poolCfg.ammStorage;
    model.ammTreasury = poolCfg.ammTreasury;
    model.assetManagement = poolCfg.assetManagement;

    IporTypes.AmmBalancesMemory memory balance = model.getAccruedBalance();

    StorageLib.AmmPoolsParamsValue memory ammPoolsParamsCfg =
AmmConfigurationManager.getAmmPoolsParams(
            poolCfg.asset
      );

      int256 rebalanceAmount =
AssetManagementLogic.calculateRebalanceAmountBeforeWithdraw(
            wadAmmTreasuryErc20BalanceBeforeRedeem,
            balance.vault,
            transferAmount + liquidationDepositAmount,
            uint256(ammPoolsParamsCfg.ammTreasuryAndAssetManagementRatio) *
1e14
      );
```

Vulnerability scenario

Due to a large number of liquidity redeems, the balance of the AMM treasury is low. A swap is being closed, and there is insufficient liquidity in the AMM treasury, so the rebalance logic is triggered. The swap was opened in USDT, which has six decimals. The rebalance amount is evaluated higher than intended, resulting in more tokens withdrawn from the asset management. This causes an indirect loss to the liquidity pool because the extra tokens withdrawn from the asset management are no longer utilized in Aave and Compound.

Recommendation

Rename the variable `wadAmmTreasuryErc20BalanceBeforeRedeem` to `ammTreasuryErc20BalanceBeforeRedeem` and convert the value to wads when calling the `calculateRebalanceAmountBeforeWithdraw` function.

Fix 1.1

The variable was renamed to `ammTreasuryErc20BalanceBeforeRedeem`, and the value is now converted to wads when calling the `calculateRebalanceAmountBeforeWithdraw` function.

Go back to Findings Summary

L2: Liquidation deposit accounted twice in rebalancing logic

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	AmmCloseSwapService	Type:	Logic error

Description

The value of the `wadTransferAmount` variable in the following code listing is an amount to be transferred to a swap buyer. The transfer amount is the swap collateral plus the profit and loss value. If the swap buyer is the one who closes the swap, the liquidation deposit is added to the final transfer value.

Listing 13. AmmCloseSwapService._transferDerivativeAmount

```
if (beneficiary == buyer) {
    transferAmount = transferAmount + liquidationDepositAmount;
} else {
    //transfer liquidation deposit amount from AmmTreasury to Liquidator
address (beneficiary),
    // transfer to be made outside this function, to avoid multiple
transfers
    payoutForLiquidator = liquidationDepositAmount;
}
```

If the transfer amount is higher than the current balance of the AMM treasury, a rebalance must be performed. The rebalance value is calculated in the `calculateRebalanceAmountBeforeWithdraw` function. The function accepts the total value to be transferred (including the liquidation deposit). The value is passed as `transferAmount + liquidationDepositAmount`. However, in the case when the swap buyer is the one who closes the swap, the liquidation deposit is already included in the transfer amount.

Listing 14. AmmCloseSwapService._transferDerivativeAmount

```
int256 rebalanceAmount =
AssetManagementLogic.calculateRebalanceAmountBeforeWithdraw(
    wadAmmTreasuryErc20BalanceBeforeRedeem,
    balance.vault,
    transferAmount + liquidationDepositAmount,
    uint256(ammPoolsParamsCfg.ammTreasuryAndAssetManagementRatio) * 1e14
);

if (rebalanceAmount < 0) {
    IAmmTreasury(poolCfg.ammTreasury).withdrawFromAssetManagementInternal(
        (-rebalanceAmount).toUint256()
    );
}
```

Vulnerability scenario

Due to a large number of liquidity redeems, the balance of the AMM treasury is low. A swap is being closed, and there is insufficient liquidity in the AMM treasury, so the rebalance logic is triggered. The swap buyer is closing the swap. Because of the bug, slightly more tokens are withdrawn from the asset management than intended. This causes an indirect loss to the liquidity pool because the extra tokens withdrawn from the asset management are no longer utilized in Aave and Compound.

Recommendation

Replace `liquidationDepositAmount` with `payoutForLiquidator` in the `calculateRebalanceAmountBeforeWithdraw` function call.

Fix 1.1

The issue was fixed according to the recommendation.

Go back to Findings Summary

L3: Aave incorrect APY formula

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	StrategyAave	Type:	Logic error

Description

The IPOR liquidity pool balance is split into treasury and asset management. Assets in the treasury are directly accessible and can be used for swap payoffs and when redeeming liquidity by a liquidity provider. Assets in the asset management are deposited into Aave or Compound, depending on the current APY (annual percentage yield). The protocol with the higher APY is chosen. Both protocols perform interest compounding — the interest is credited after a given period and the new interest is computed from the total deposited amount, including the credited interest.

The `StrategyAave` and `StrategyCompound` contracts are responsible for computing the estimated APY. However, `StrategyCompound` does take into account interest compounding while `StrategyAave` does not. It makes the two APY values incomparable.

Listing 15. StrategyCompound

```
function getApr() external view override returns (uint256 apr) {
    uint256 cRate = CErc20(_shareToken).supplyRatePerBlock(); // interest %
per block
    uint256 ratePerDay = cRate * _blocksPerDay + 1e18;


    uint256 ratePerDay4 = IporMath.division(ratePerDay * ratePerDay *
ratePerDay * ratePerDay, 1e54);
    uint256 ratePerDay8 = IporMath.division(ratePerDay4 * ratePerDay4,
1e18);
```

```
    uint256 ratePerDay32 = IporMath.division(ratePerDay8 * ratePerDay8 *
ratePerDay8 * ratePerDay8, 1e54);
    uint256 ratePerDay64 = IporMath.division(ratePerDay32 * ratePerDay32,
1e18);
    uint256 ratePerDay256 = IporMath.division(ratePerDay64 * ratePerDay64 *
ratePerDay64 * ratePerDay64, 1e54);
    uint256 ratePerDay360 = IporMath.division(ratePerDay256 * ratePerDay64
* ratePerDay32 * ratePerDay8, 1e54);
    uint256 ratePerDay365 = IporMath.division(ratePerDay360 * ratePerDay4 *
ratePerDay, 1e36);


    apr = ratePerDay365 - 1e18;
}
```

Listing 16. StrategyAave

```
function getApr() external view override returns (uint256 apr) {
    address lendingPoolAddress = _provider.getLendingPool();
    require(lendingPoolAddress != address(0), IporErrors.WRONG_ADDRESS);
    AaveLendingPoolV2 lendingPool = AaveLendingPoolV2(lendingPoolAddress);


    DataTypesContract.ReserveData memory reserveData =
lendingPool.getReserveData(_asset);
    apr = IporMath.division(reserveData.currentLiquidityRate, (10**9));
}
```

Vulnerability scenario

A new significant liquidity deposit is performed, which triggers a rebalance. The rebalance logic is triggered, and the APY is computed for Aave and Compound. The APY values are very close, but the Aave APY value is slightly lower because it does not take into account interest compounding. Assets are deposited into Compound, although Aave actually has a higher APY. This causes an indirect loss to the liquidity pool because the assets are not deposited into the protocol with the highest APY.

Recommendation

Compute the APY for Aave with interest compounding using the same logic as Compound.

Fix 1.1

The functions were renamed to `getApy` to reflect their purpose better, and the APY for Aave is now computed with interest compounding.

Go back to Findings Summary

M5: Close swap and redeem transaction reverts

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	AmmCloseSwapService, AmmPoolsService, AssetManagement	Type:	Denial of service

Description

Multiple edge case scenarios can cause a close swap or liquidity redeem transactions to revert. These scenarios assume insufficient liquidity in the AMM treasury and a rebalance must be performed.

Liquidation deposit not accounted into transfer amount when closing a swap.

If the close swap beneficiary is not the swap buyer, the liquidation deposit is not transferred when closing the given swap. However, all liquidation deposits are later transferred at once. The logic of the `_transferDerivativeAmount` function does not consider the liquidation deposit when the beneficiary is not the swap buyer. Suppose there is enough liquidity for the swap payoff to the buyer but not enough liquidity for the swap payoff plus the liquidator payoff. In that case, the rebalance is not performed, and the close swap transaction reverts because of insufficient balance when transferring the liquidation deposit.

Listing 17. AmmCloseSwapService._transferDerivativeAmount

```
if (beneficiary == buyer) {
    transferAmount = transferAmount + liquidationDepositAmount;
} else {
    //transfer liquidation deposit amount from AmmTreasury to Liquidator
address (beneficiary),
    // transfer to be made outside this function, to avoid multiple
```

```
transfers
    payoutForLiquidator = liquidationDepositAmount;
}


if (transferAmount > 0) {
    uint256 transferAmountAssetDecimals =
IporMath.convertWadToAssetDecimals(transferAmount, poolCfg.decimals);
    uint256 wadAmmTreasuryErc20BalanceBeforeRedeem =
IERC20Upgradeable(poolCfg.asset).balanceOf(
        poolCfg.ammTreasury
    );


    if (wadAmmTreasuryErc20BalanceBeforeRedeem <=
transferAmountAssetDecimals) {
```

Withdrawal from the asset management may withdraw fewer tokens than requested.

Due to different decimals of Aave/Compound tokens than the underlying tokens and due to rounding errors, the amount of tokens withdrawn from the asset management may be less than the requested amount. This can be an issue if the requested amount of tokens is precisely needed to satisfy all transfers in the current transaction. The transaction will revert due to insufficient balance in this case.

Assets are withdrawn only from a single strategy when performing a withdrawal.

The logic of the `withdraw` function in the `AssetManagement` contract is designed to withdraw assets from a single strategy only. If the requested amount of tokens is large enough and the tokens are spread in both Aave and Compound, an insufficient amount of tokens may be withdrawn, and the transaction may revert due to insufficient balance of the AMM treasury.

Listing 18. AssetManagement.withdraw

```
    assetBalanceAaveStrategy,
    assetBalanceCompoundStrategy
);

if (selectedWithdrawAmount > 0) {
    //Transfer from Strategy to AssetManagement
    uint256 ivTokenWithdrawnAmount;
    (ivTokenWithdrawnAmount, vaultBalance) = _withdrawFromStrategy(
        selectedStrategy,
        selectedWithdrawAmount,
        ivTokenTotalSupply,
        strategyAave,
        strategyCompound
    );

    if (ivTokenWithdrawnAmount > senderIvTokens) {
```

Vulnerability scenario

All of the described scenarios lead to the same vulnerability scenario when there are not enough tokens in the AMM treasury needed to complete the current transaction. This is the swap payoff amount plus the liquidation deposit in the case of a close swap transaction and the liquidity redemption amount in the case of a liquidity redemption transaction. From the end user's perspective, the transaction revert is unexpected, and the user cannot withdraw their funds from the AMM or close their swap.

Recommendation

Liquidation deposit not accounted into transfer amount when closing a swap.

Compute `totalTransferAmountAssetDecimals` in the following way:

```
uint256 totalTransferAmountAssetDecimals = transferAmountAssetDecimals +
IporMath.convertWadToAssetDecimals(wadPayoutForLiquidator,
poolCfg.decimals);
```

Use this value in comparison with `wadAmmTreasuryErc20BalanceBeforeRedeem`.

Withdrawal from the asset management may withdraw fewer tokens than requested.

Fix any rounding errors in the `withdraw` function in the `AssetManagement` contract or withdraw a larger amount of tokens from the asset management to ensure that at least the requested amount of tokens is always withdrawn.

Assets are withdrawn only from a single strategy when performing a withdrawal.

Consider changing the logic of the `withdraw` function in the `AssetManagement` contract to withdraw assets from multiple strategies if needed.

Fix 1.1

The first issue was fixed using `totalTransferAmountAssetDecimals` as described in the recommendation.

Fix 1.3

All the scenarios were fixed.

Go back to Findings Summary

W1: Usage of `solc` optimizer

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Compiler configuration

Description

The project uses `solc` optimizer. Enabling `solc` optimizer may lead to unexpected bugs.

The Solidity compiler was audited in November 2018, and the audit concluded that the optimizer may not be safe.

Vulnerability scenario

A few months after deployment, a vulnerability is discovered in the optimizer. As a result, it is possible to attack the protocol.

Recommendation

Until the `solc` optimizer undergoes more stringent security analysis, opt-out using it. This will ensure the protocol is resilient to any existing bugs in the optimizer.

Go back to Findings Summary

W2: SoapIndicatorRebalanceLogic underflow

Impact:	Warning	Likelihood:	N/A
Target:	SoapIndicatorRebalanceLogic	Type:	Denial of service

Description

The `SoapIndicatorRebalanceLogic` re-calculates the average interest rate when opening and closing a swap. A simple weighted average with the swap notional is used for this.

Listing 19. SoapIndicatorRebalanceLogic

```
function calculateAverageInterestRateWhenOpenSwap(
    uint256 totalNotional,
    uint256 averageInterestRate,
    uint256 derivativeNotional,
    uint256 swapFixedInterestRate
) internal pure returns (uint256) {
    return
        IporMath.division(
            (totalNotional * averageInterestRate + derivativeNotional *
swapFixedInterestRate),
            (totalNotional + derivativeNotional)
        );
}
```

Opening a swap with a zero interest rate decreases the new average interest rate and can even result in the average rate being zero. As a consequence, the expression `totalNotional * averageInterestRate - swapNotional * swapFixedInterestRate` in the following function may be subject to underflow if there were previously opened swaps with zero interest rates.

Listing 20. SoapIndicatorRebalanceLogic

```
function calculateAverageInterestRateWhenCloseSwap(
    uint256 totalNotional,
    uint256 averageInterestRate,
    uint256 derivativeNotional,
    uint256 swapFixedInterestRate
) internal pure returns (uint256) {
    require(derivativeNotional <= totalNotional,
AmmErrors.SWAP_NOTIONAL_HIGHER_THAN_TOTAL_NOTIONAL);
    if (derivativeNotional == totalNotional) {
        return 0;
    } else {
        return
            IporMath.division(
                (totalNotional * averageInterestRate - derivativeNotional *
swapFixedInterestRate),
                (totalNotional - derivativeNotional)
            );
    }
}
```

Recommendation

Ensure that a swap with a zero interest rate cannot be opened and that the average interest rate can never be set to zero (except for the initial value).

Fix 1.1

Require statements were added to the `AmmOpenSwapService` and `SoapIndicatorRebalanceLogic` contracts to prevent opening a swap with a zero interest rate and setting the average interest rate to zero.

Go back to Findings Summary

W3: Insufficient data validation in the constructor

Impact:	Warning	Likelihood:	N/A
Target:	AmmPoolService	Type:	Data validation

Description

The constructor of the contract AmmPoolService lacks data validation for following parameters:

- decimals
- redeemFeeRate
- redeemLpMaxCollateralRatio

The rest of the parameters, such as: `ipToken`, `ammStorage`, `ammTreasury`, `assetManagement`, and `iporOracle` perform a zero address check. Although it is better than no validation, there are more sufficient ways, such as contract ID check implemented in IporToken.

Listing 21. AmmPoolsService.constructor

```
constructor(
        AmmPoolsServicePoolConfiguration memory usdtPoolCfg,
        AmmPoolsServicePoolConfiguration memory usdcPoolCfg,
        AmmPoolsServicePoolConfiguration memory daiPoolCfg,
        address iporOracle
    ) {
        _usdt = usdtPoolCfg.asset.checkAddress();
        _usdtDecimals = usdtPoolCfg.decimals;
        _usdtIpToken = usdtPoolCfg.ipToken.checkAddress();
        _usdtAmmStorage = usdtPoolCfg.ammStorage.checkAddress();
        _usdtAmmTreasury = usdtPoolCfg.ammTreasury.checkAddress();
        _usdtAssetManagement = usdtPoolCfg.assetManagement.checkAddress();
        _usdtRedeemFeeRate = usdtPoolCfg.redeemFeeRate;
        _usdtRedeemLpMaxCollateralRatio =
usdtPoolCfg.redeemLpMaxCollateralRatio;

        _usdc = usdcPoolCfg.asset.checkAddress();
        _usdcDecimals = usdcPoolCfg.decimals;
        _usdcIpToken = usdcPoolCfg.ipToken.checkAddress();
        _usdcAmmStorage = usdcPoolCfg.ammStorage.checkAddress();
        _usdcAmmTreasury = usdcPoolCfg.ammTreasury.checkAddress();
        _usdcAssetManagement = usdcPoolCfg.assetManagement.checkAddress();
        _usdcRedeemFeeRate = usdcPoolCfg.redeemFeeRate;
        _usdcRedeemLpMaxCollateralRatio =
usdcPoolCfg.redeemLpMaxCollateralRatio;

        _dai = daiPoolCfg.asset.checkAddress();
        _daiDecimals = daiPoolCfg.decimals;
        _daiIpToken = daiPoolCfg.ipToken.checkAddress();
        _daiAmmStorage = daiPoolCfg.ammStorage.checkAddress();
        _daiAmmTreasury = daiPoolCfg.ammTreasury.checkAddress();
        _daiAssetManagement = daiPoolCfg.assetManagement.checkAddress();
        _daiRedeemFeeRate = daiPoolCfg.redeemFeeRate;
        _daiRedeemLpMaxCollateralRatio =
daiPoolCfg.redeemLpMaxCollateralRatio;

        _iporOracle = iporOracle.checkAddress();
    }
```

Recommendation

Implement sufficient data validation for mentioned parameters. Consider adding contract ID checks across the protocol for constructor address parameters under the control of IPOR.

Fix 1.2

Issue was partially fixed with a following client's response:

" We will not add contractId validation concept because this one not solve following scenario: Contract A use Contract B in constructor param Contract A validate contractId in Contract B Contract A use asset USDT Contract B during construction use asset DAI. Validation contractId does not protect against mismatch between asset in Contract A and Contract B Analogical situation can be in any contract which is not from the same “family” associated with the same asset. Instead we will prepare bunch of tests which validate smart contracts parameters/configuration on fork after deployment but before we open protocol for further use by users in V2. ”

Go back to Findings Summary

W4: Missing array length check in the initialize function

Impact:	Warning	Likelihood:	N/A
Target:	IporOracle	Type:	Data validation

Description

The `initialize` function is not checking if `assets` and `updatedTimestamp` arrays have equal lengths.

Listing 22. IporOracle.initialize

```
function initialize(address[] memory assets, uint32[] memory
updateTimestamps) public initializer {
```

If a bigger array of timestamps is passed by accident, it can cause the timestamps to be poorly matched to the relevant assets.

Recommendation

Perform an array lengths check.

Fix 1.2

Fixed.

Go back to Findings Summary

W5: `_calculateRedeemedCollateralRatio` underflow

Impact:	Warning	Likelihood:	N/A
Target:	AmmPoolsService	Type:	Underflow

Description

The `_calculateRedeemedCollateralRatio` function calculates a ratio of total collateral balance and total liquidity balance after a liquidity redemption.

Listing 23. AmmPoolsService

```
function _calculateRedeemedCollateralRatio(
    uint256 totalLiquidityPoolBalance,
    uint256 totalCollateralBalance,
    uint256 redeemedAmount
) internal pure returns (uint256) {
    uint256 denominator = totalLiquidityPoolBalance - redeemedAmount;
    if (denominator > 0) {
        return IporMath.division(totalCollateralBalance * 1e18,
totalLiquidityPoolBalance - redeemedAmount);
    } else {
        return Constants.MAX_VALUE;
    }
}
```

The denominator is computed as a difference between `totalLiquidityPoolBalance` and `redeemedAmount`. However, since the `redeemedAmount` is user-supplied, the expression can underflow and raise a revert error. This is correct because the transaction would revert if the `Constants.MAX_VALUE` had been returned instead. However, the underflow error does not provide a user-friendly error message.

Recommendation

Add an `if` condition checking that `redeemedAmount` is less than `totalLiquidityPoolBalance` and return `Constants.MAX_VALUE` if it is not. Otherwise, compute the `denominator` value and proceed with the calculation.

Update (Revision 1.1)

The function was rewritten in the following way:

Listing 24. AmmPoolsService

```
function _calculateRedeemedCollateralRatio(
    uint256 totalLiquidityPoolBalance,
    uint256 totalCollateralBalance,
    uint256 redeemedAmount
) internal pure returns (uint256) {
    uint256 denominator = totalLiquidityPoolBalance - redeemedAmount;
    if (denominator > 0) {
        if (totalLiquidityPoolBalance <= redeemedAmount) {
            return Constants.MAX_VALUE;
        } else {
            return IporMath.division(totalCollateralBalance * 1e18,
totalLiquidityPoolBalance - redeemedAmount);
        }
    } else {
        return Constants.MAX_VALUE;
    }
}
```

The `totalLiquidityPoolBalance <= redeemedAmount` condition was added, but this does not solve the underflow issue. The `denominator` value is computed as the first statement in the function, and the evaluation of the `denominator` value may cause underflows.

Fix 1.2

Fixed. The function was rewritten in the following way:

Listing 25. AmmPoolsService

```
function _calculateRedeemedCollateralRatio(
      uint256 totalLiquidityPoolBalance,
      uint256 totalCollateralBalance,
      uint256 redeemedAmount
) internal pure returns (uint256) {
      if (totalLiquidityPoolBalance <= redeemedAmount) {
          return Constants.MAX_VALUE;
      }

      return IporMath.division(totalCollateralBalance * 1e18,
totalLiquidityPoolBalance - redeemedAmount);
ic}
```

Go back to Findings Summary

W6: Constant block production relied on

Impact:	Warning	Likelihood:	N/A
Target:	StrategyCompound	Type:	Blockchain parameter assumptions

Description

The `StrategyCompound` contract uses the `getApr` function to estimate the current APY (annual percentage yield) of the Compound protocol. Compound uses interest compounding based on block numbers. However, the `getApr` function computes the APY using the number of days. There is a configurable parameter defining the number of blocks mined per day.

Listing 26. StrategyCompound

```
function getApr() external view override returns (uint256 apr) {
    uint256 cRate = CErc20(_shareToken).supplyRatePerBlock(); // interest %
per block
    uint256 ratePerDay = cRate * _blocksPerDay + 1e18;


    uint256 ratePerDay4 = IporMath.division(ratePerDay * ratePerDay *
ratePerDay * ratePerDay, 1e54);
    uint256 ratePerDay8 = IporMath.division(ratePerDay4 * ratePerDay4,
1e18);
    uint256 ratePerDay32 = IporMath.division(ratePerDay8 * ratePerDay8 *
ratePerDay8 * ratePerDay8, 1e54);
    uint256 ratePerDay64 = IporMath.division(ratePerDay32 * ratePerDay32,
1e18);
    uint256 ratePerDay256 = IporMath.division(ratePerDay64 * ratePerDay64 *
ratePerDay64 * ratePerDay64, 1e54);
    uint256 ratePerDay360 = IporMath.division(ratePerDay256 * ratePerDay64
* ratePerDay32 * ratePerDay8, 1e54);
    uint256 ratePerDay365 = IporMath.division(ratePerDay360 * ratePerDay4 *
ratePerDay, 1e36);
```

```
    apr = ratePerDay365 - 1e18;
}
```

The block production rate may not be constant, and the estimated number of blocks mined daily may be inaccurate. This is especially true in the case of L2 chains, for example, Optimism:

> On Ethereum, the NUMBER opcode (block.number in Solidity) corresponds to the current Ethereum block number. Similarly, in Optimism, block.number corresponds to the current L2 block number. However, as of the OVM 2.0 release of Optimism (Nov. 2021), each transaction on L2 is placed in a separate block, and blocks are NOT produced at a constant rate.
>
> — Optimism documentation

Recommendation

Remember that the `StrategyCompound` contract relies on constant block production. When considering deployment to new chains (especially L2), research the block production parameters of the chain. When possible (after DAI and USDT pools opened), consider migrating to Compound v3.

Go back to Findings Summary

W7: Github secrets leak

Impact:	Warning	Likelihood:	N/A
Target:	`**/*`	Type:	Security practices

Description

We used `$ gitleaks -v detect` and found several private keys in the repository. Keys may be used for local testing or non-relevant anymore. However, the repository should not contain any sensitive data.

Recommendation

Clean the repository from sensitive data

Fix 1.2

Fixed.

Go back to Findings Summary

W8: Infinite approval

Impact:	Warning	Likelihood:	N/A
Target:	StrategyAave, StrategyCompound	Type:	Security practices

Description

In the `initialize` function, unlimited approval is performed to the `lendingPoolAddress`. This approach saves some gas and has better UX, but for the cost of the security.

```
IERC20Upgradeable(_asset).safeApprove(lendingPoolAddress, type(
uint256).max);
```

Recommendation

Do approvals based on the needed amount in the specific transaction.

Fix 1.3

The contracts are now using the function `.forceApprove()` in the `deposit()` function.

Go back to Findings Summary

I1: Unreachable code

Impact:	Info	Likelihood:	N/A
Target:	AmmCloseSwapService	Type:	Code quality

Description

A swap buyer can always close the swap. However, the function `_validateAllowanceToCloseSwap` in the `AmmCloseSwapService` contract raises an error of type `AmmErrors.CANNOT_CLOSE_SWAP_CLOSING_IS_TOO_EARLY_FOR_BUYER`. The corresponding `if` branch of the code is unreachable.

Listing 27. AmmCloseSwapService

```
uint256 closableStatus = _getClosableStatusForSwap(owner, payoff,
closeTimestamp, swap, poolCfg);


if (closableStatus == 1) revert(AmmErrors.INCORRECT_SWAP_STATUS);
if (closableStatus == 2)
revert(AmmErrors.CANNOT_CLOSE_SWAP_SENDER_IS_NOT_BUYER_NOR_LIQUIDATOR);


if (closableStatus == 3 || closableStatus == 4) {
    if (msg.sender == swap.buyer) {
        swapUnwindRequired = true;
    } else {
        if (closableStatus == 3)
revert(AmmErrors.CANNOT_CLOSE_SWAP_CLOSING_IS_TOO_EARLY_FOR_BUYER);
        if (closableStatus == 4)
revert(AmmErrors.CANNOT_CLOSE_SWAP_CLOSING_IS_TOO_EARLY);
    }
}
```

Recommendation

Remove the unreachable `if` branch and remove the error

CANNOT_CLOSE_SWAP_CLOSING_IS_TOO_EARLY_FOR_BUYER or name it to a better status code describing that the swap can be closed.

Partial solution (Revision 1.1)

The code was significantly refactored. The logic of the _validateAllowanceToCloseSwap was simplified:

Listing 28. AmmCloseSwapService

```
function _validateAllowanceToCloseSwap(
    AmmTypes.SwapClosableStatus closableStatus,
    bool swapUnwindRequired
) internal pure {
    if (closableStatus == AmmTypes.SwapClosableStatus.SWAP_ALREADY_CLOSED)
{
        revert(AmmErrors.INCORRECT_SWAP_STATUS);
    }
    if (closableStatus ==
AmmTypes.SwapClosableStatus.SWAP_REQUIRED_BUYER_OR_LIQUIDATOR_TO_CLOSE) {

revert(AmmErrors.CANNOT_CLOSE_SWAP_SENDER_IS_NOT_BUYER_NOR_LIQUIDATOR);
    }
    if (closableStatus ==
AmmTypes.SwapClosableStatus.SWAP_CANNOT_CLOSE_CLOSING_TOO_EARLY_FOR_COMMUNI
TY) {
        revert(AmmErrors.CANNOT_CLOSE_SWAP_CLOSING_IS_TOO_EARLY);
    }
}
```

Still, the name of the error remained inappropriate (SWAP_CANNOT_CLOSE_CLOSING_TOO_EARLY_FOR_BUYER).

```
return
(AmmTypes.SwapClosableStatus.SWAP_CANNOT_CLOSE_CLOSING_TOO_EARLY_FOR_BUYER,
true);
```

Fix 1.2

Fixed. Error string changed to (`SWAP_IS_CLOSABLE`).

Go back to Findings Summary

I2: Use `type(uint256).max` instead of integer literal

Impact:	Info	Likelihood:	N/A
Target:	Constants	Type:	Code quality

Description

The `Constants` library defines the `MAX_VALUE` constant in the following way:

Listing 29. Constants

```
uint256 public constant MAX_VALUE =
115792089237316195423570985008687907853269984665640564039457584007913129639
935;
```

At first sight, it is not clear this value represents the maximum value of the `uint256` type.

Recommendation

Replace the literal with `type(uint256).max`.

Fix 1.2

Fixed.

Go back to Findings Summary

I3: Duplicated code

Impact:	Info	Likelihood:	N/A
Target:	AmmLib, RiskManagementLogic	Type:	Code quality

Description

Both libraries `AmmLib` and `RiskManagementLogic` define a function named `getRiskIndicators`. The functions return the same data, and the logic is almost identical.

Listing 30. AmmLib

```
function getRiskIndicators(
    AmmInternalTypes.RiskIndicatorsContext memory context,
    uint256 direction
) internal view returns (AmmTypes.OpenSwapRiskIndicators memory
riskIndicators) {
```

Listing 31. RiskManagementLogic

```
function getRiskIndicators(
    address asset,
    uint256 direction,
    IporTypes.SwapTenor tenor,
    uint256 liquidityPool,
    uint256 cfgMinLeverage,
    address cfgIporRiskManagementOracle
) internal view returns (AmmInternalTypes.OpenSwapRiskIndicators memory
riskIndicators) {
```

Recommendation

Consider refactoring the functions into a single function in either of the libraries. Duplicating the same code in multiple places is not a good coding practice.

Fix 1.3

The code was refactored.

Go back to Findings Summary

I4: Redundant require

Impact:	Info	Likelihood:	N/A
Target:	AmmOpenSwapService	Type:	Code quality

Description

The function `openSwapReceiveFixed60daysUsdt` in the contract `AmmOpenSwapService` contains the following requirement:

Listing 32. AmmOpenSwapService.openSwapReceiveFixed60daysUsdt

```
require(beneficiary != address(0), "AmmOpenSwapService: beneficiary is zero
address");
```

No other function for opening swaps has the same check. The beneficiary address is checked in the function `_beforeOpenSwap`.

Recommendation

Remove redundant code to make the code look more straightforward and gas efficient.

Fix 1.2

Fixed. Code was removed.

Go back to Findings Summary

I5: Using magic numbers

Impact:	Info	Likelihood:	N/A
Target:	`**/*`	Type:	Code quality

Description

Magic numbers are used across the codebase. Magic numbers usually work like a constant. However, they are not assigned to a variable but hardcoded in the code.

When calculating rate and 1e18 is used as a divider, it is obvious and easy to understand. But using other numbers in more unusual places makes the code harder to understand.

For example in the contract 5.1.4.2 the function `_getRiskIndicators` returns following:

Listing 33. IporRiskManagementOracle._getRiskIndicators

```
return (
    uint256(indicators.maxNotionalPayFixed) * 1e22, // 1 = 10k notional
    uint256(indicators.maxNotionalReceiveFixed) * 1e22,
    uint256(indicators.maxCollateralRatioPayFixed) * 1e14, // 1 = 0.01%
    uint256(indicators.maxCollateralRatioReceiveFixed) * 1e14,
    uint256(indicators.maxCollateralRatio) * 1e14,
    uint256(indicators.lastUpdateTimestamp)
);
```

Even though the code is commented, it is not clear why the numbers are multiplied by 1e22 and 1e14 and it takes some time to understand it.

Another example is in the contract StrategyAave in the function `getApr`:

```
apr = IporMath.division(reserveData.currentLiquidityRate, (10**9));
```

It should be clear from the first look why it is divided by 10**9.

There are many other occurrences of magic numbers in the codebase.

Recommendation

Add more explanatory comments to the code to make it easier to understand. Stick with one way of writing the numbers; either use 1e18 or 10**18.

Fix 1.3

The codebase now contains a lot of explanatory comments.

Go back to Findings Summary

6. Report revision 1.1

The logic changes in the codebase were performed as a response to found issues.

6.1. System Overview

The following methods were removed from the contract AmmCloseSwapService:

- closeSwapPayFixedUsdt
- closeSwapPayFixedUsdc
- closeSwapPayFixedDai
- closeSwapReceiveFixedUsdt
- closeSwapReceiveFixedUsdc
- closeSwapReceiveFixedDai
- emergencyCloseSwapPayFixedUsdt
- emergencyCloseSwapPayFixedUsdc
- emergencyCloseSwapPayFixedDai
- emergencyCloseSwapReceiveFixedUsdt
- emergencyCloseSwapReceiveFixedUsdc
- emergencyCloseSwapReceiveFixedDai
- emergencyCloseSwapsPayFixedUsdt
- emergencyCloseSwapsPayFixedUsdc
- emergencyCloseSwapsPayFixedDai
- emergencyCloseSwapsReceiveFixedUsdt

Instead, the following more generic methods and one view method were added:

- emergencyCloseSwapsUsdt
- emergencyCloseSwapsUsdc
- emergencyCloseSwapsDai
- getClosingSwapDetails

The generic methods for standard swap close very already included in the previous version, they are now used directly.

H3: Unwinding fee accounted twice in `liquidityPool` balance

High severity issue

Impact:	Medium	Likelihood:	High
Target:	AmmStorage	Type:	Logic error

Description

A buyer can always close his swap. However, closing the swap before its maturity or before 100% profit or loss is reached is subject to an unwinding fee. The unwinding fee is split into two parts:

- the part that belongs to the liquidity pool,
- the part that belongs to IPOR.

The fee is accounted into the final profit & loss value in the `_calculateSwapUnwindWhenUnwindRequired` function:

Listing 34. AmmCloseSwapService._calculateSwapUnwindWhenUnwindRequired

```
(swapUnwindOpeningFeeLPAmount, swapUnwindOpeningFeeTreasuryAmount) =
IporSwapLogic.splitOpeningFeeAmount(
    swapUnwindOpeningFeeAmount,
    poolCfg.unwindingFeeTreasuryPortionRate
);


swapPnlValue = swapPnlValueToDate + swapUnwindPnlValue -
swapUnwindOpeningFeeAmount.toInt256();
swapPnlValue = IporSwapLogic.normalizePnlValue(swap.collateral,
swapPnlValue);
```

The absolute value of the profit & loss value is then used to update the liquidityPool balance in the AmmStorage contract:

Listing 35. AmmStorage._updateBalancesWhenCloseSwap

```
if (pnlValue > 0) {
    /// @dev Buyer earns, AMM (LP) looses
    require(_balances.liquidityPool >= absPnlValue,
AmmErrors.CANNOT_CLOSE_SWAP_LP_IS_TOO_LOW);
    /// @dev When AMM (LP) looses, then  always substract all pnlValue
    _balances.liquidityPool =
        _balances.liquidityPool -
        absPnlValue.toUint128() +
        swapUnwindOpeningFeeLPAmount.toUint128();
} else {
    /// @dev AMM earns, Buyer looses,
    _balances.liquidityPool =
        _balances.liquidityPool +
        absPnlValue.toUint128() +
        swapUnwindOpeningFeeLPAmount.toUint128();
}
_balances.treasury = _balances.treasury +
swapUnwindOpeningFeeTreasuryAmount.toUint128();
```

Since the absPnlValue variable already has the unwinding fee accounted:

- swapUnwindOpeningFeeLPAmount is accounted for twice in the liquidityPool balance,
- swapUnwindOpeningFeeTreasuryAmount is accounted in the liquidityPool balance even though it should not.

Vulnerability scenario

Over time, the error in the liquidityPool balance accumulates, and the liquidityPool balance is higher than it should be. This has a significant impact on the overall protocol execution. Most importantly, the error influences:

- pay fixed/receive fixed offered rates,

- ip token exchange rates,
- maximum leverage,
- evaluation of protocol constraints and safety checks.

Recommendation

Replace the subexpression `+ swapUnwindOpeningFeeLPAmount.toUint128()` with `- swapUnwindOpeningFeeTreasuryAmount.toUint128()` in both `if` branches. This ensures that the liquidity pool balance is not increased for the part of the unwinding fee that belongs to IPOR and that the liquidity pool part is not accounted for twice.

Fix 1.3

The unwind fee is not accounted for twice anymore.

Go back to Findings Summary

M6: Unwinding fee normalization

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AmmCloseSwapService	Type:	Logic error

Description

When performing unwinding because of closing a swap before its maturity or 100% profit or loss reached, the total profit & loss value is computed from the current profit & loss, from the unwinding value and the unwinding fee. The final swap profit & loss value is normalized not to be less/more than the swap collateral.

Listing 36. AmmCloseSwapService

```
swapUnwindPnlValue = swap.calculateSwapUnwindPnlValue(direction,
closeTimestamp, oppositeLegFixedRate);


swapUnwindOpeningFeeAmount =
swap.calculateSwapUnwindOpeningFeeAmount(closeTimestamp,
poolCfg.unwindingFeeRate);


(swapUnwindOpeningFeeLPAmount, swapUnwindOpeningFeeTreasuryAmount) =
IporSwapLogic.splitOpeningFeeAmount(
    swapUnwindOpeningFeeAmount,
    poolCfg.unwindingFeeTreasuryPortionRate
);


swapPnlValue = swapPnlValueToDate + swapUnwindPnlValue -
swapUnwindOpeningFeeAmount.toInt256();
swapPnlValue = IporSwapLogic.normalizePnlValue(swap.collateral,
swapPnlValue);
```

The unwinding fee, especially parts of the fee (`swapUnwindOpeningFeeLPAmount` and `swapUnwindOpeningFeeTreasuryAmount`) are not updated after normalization. This can lead to one of the scenarios when `swapPnlValue - swapUnwindOpeningFeeAmount < -swap.collateral` or `swapPnlValue + swapUnwindOpeningFeeAmount > swap.collateral`.

Later, `swapPnlValue`, `swapUnwindOpeningFeeLPAmount`, and `swapUnwindOpeningFeeTreasuryAmount` are used to update the balances of the liquidity pool and IPOR treasury in the `AmmStorage`.

Listing 37. AmmStorage

```
function _updateBalancesWhenCloseSwap(
    int256 pnlValue,
    uint256 swapUnwindOpeningFeeLPAmount,
    uint256 swapUnwindOpeningFeeTreasuryAmount
) internal {
    uint256 absPnlValue = IporMath.absoluteValue(pnlValue);


    if (pnlValue > 0) {
        /// @dev Buyer earns, AMM (LP) looses
        require(_balances.liquidityPool >= absPnlValue,
AmmErrors.CANNOT_CLOSE_SWAP_LP_IS_TOO_LOW);
        /// @dev When AMM (LP) looses, then  always substract all pnlValue
        _balances.liquidityPool =
            _balances.liquidityPool -
            absPnlValue.toUint128() +
            swapUnwindOpeningFeeLPAmount.toUint128();
    } else {
        /// @dev AMM earns, Buyer looses,
        _balances.liquidityPool =
            _balances.liquidityPool +
            absPnlValue.toUint128() +
            swapUnwindOpeningFeeLPAmount.toUint128();
    }
    _balances.treasury = _balances.treasury +
swapUnwindOpeningFeeTreasuryAmount.toUint128();
}
```

However, more than `swap.collateral` cannot be taken from a user since the user only deposited the swap collateral plus initial fees not accounted into the collateral. In the second scenario, the original swap collateral plus a profit of additional swap collateral is transferred to the user, but the unwinding fee is not taken from the user. The liquidity pool and IPOR treasury balances are updated with unwinding fees in both scenarios. However, the fees are not actually taken from the user in the amount they are updated.

Vulnerability scenario

A swap is closed, and unwinding is performed. The final profit & loss value is less than the swap collateral. The unwinding fees and the liquidity pool, and IPOR treasury balances are not updated with more than the swap collateral. This creates an error in the computed balances being higher than the actual balances. The error can accumulate over time, significantly impacting the overall protocol execution. Most importantly, the error influences:

- pay fixed/receive fixed offered rates,
- ip token exchange rates,
- maximum leverage,
- evaluation of protocol constraints and safety checks.

Recommendation

At first, do not account for the unwinding fee in the `swapPnlValue` value. Check if `swapPnlValue - swapUnwindOpeningFeeAmount < -swap.collateral`, and if this is the case, lower the unwinding fee accordingly and then recompute both components of the fee. if `swapPnlValue + swapUnwindOpeningFeeAmount > swap.collateral`, set `swapPnlValue` so that the sum of `swapPnlValue` and `swapUnwindOpeningFeeAmount` is equal to `swap.collateral`.

```
swapPnlValue = swapPnlValueToDate + swapUnwindPnlValue;

if (swapPnlValue - swapUnwindOpeningFeeAmount < -swap.collateral) {
    swapPnlValue = IporSwapLogic.normalizePnlValue(swap.collateral,
swapPnlValue);
    swapUnwindOpeningFeeAmount = swap.collateral + swapPnlValue;
    (swapUnwindOpeningFeeLPAmount, swapUnwindOpeningFeeTreasuryAmount) =
IporSwapLogic.splitOpeningFeeAmount(
        swapUnwindOpeningFeeAmount,
        poolCfg.openingFeeTreasuryPortionRateForSwapUnwind
    );
} else if (swapPnlValue + swapUnwindOpeningFeeAmount > swap.collateral) {
    swapPnlValue = swap.collateral;
}

swapPnlValue = swapPnlValue - swapUnwindOpeningFeeAmount;
```

Fix 1.3

The code for fee normalization was rewritten.

Go back to Findings Summary

W9: Missing swap direction validation

Impact:	Warning	Likelihood:	N/A
Target:	AmmCloseSwapService, AmmSwapsLens	Type:	Data validation

Description

The functions `getPnlPayFixed` and `getPnlReceiveFixed` defined in the `AmmSwapsLens` and the function `getClosingSwapDetails` defined in the `AmmCloseSwapService` contract accept a swap ID. Swap IDs are distributed so that a pay fixed swap with a given asset cannot have the same ID as a receive fixed swap with the same asset. However, all mentioned functions do not check that a swap with a given ID exists in a given direction. If a pay fixed swap with ID 1 exists and the functions are used to query a receive fixed swap with ID 1, the functions do not revert and return default zeroed-out data instead.

Listing 38. AmmCloseSwapService

```
function getClosingSwapDetails(
    address asset,
    AmmTypes.SwapDirection direction,
    uint256 swapId,
    uint256 closeTimestamp
) external view override returns (AmmTypes.ClosingSwapDetails memory
closingSwapDetails) {
```

Listing 39. AmmSwapsLens

```
function getPnlPayFixed(address asset, uint256 swapId) external view
override returns (int256) {
```

Listing 40. AmmSwapsLens

```
function getPnlReceiveFixed(address asset, uint256 swapId) external view
override returns (int256) {
```

Recommendation

Check the direction of the swap returned by the `ammStorage.getSwap(direction, swapId)` function call and revert if the direction does not match the expected one.

Fix 1.2

Fixed. The code now verifies if an ID exists for the specified direction.

Go back to Findings Summary

I6: Use `forceApprove` instead of `safeApprove`

Impact:	Info	Likelihood:	N/A
Target:	AmmTreasury	Type:	Overflow

Description

The function `grandMaxAllowanceForSpender` in the `AmmTreasury` contract sets the max approval to the IPOR router and asset management contracts. The function uses `SafeERC20Upgradeable.safeApprove` internally. It is implemented such that the transaction reverts if the approval value is not zero or the current allowance is not zero. This can be a problem if the contract already has an allowance set and the set allowance is different from `type(uint256).max`. In this case, the allowance must be zeroed out first, and then the function `grandMaxAllowanceForSpender` may be called.

Listing 41. AmmTreasury

```
function grandMaxAllowanceForSpender(address spender) external override
onlyOwner {
    IERC20Upgradeable(_asset).safeApprove(spender, Constants.MAX_VALUE);
}
```

Recommendation

Consider using `SafeERC20Upgradeable.forceApprove` in the `grandMaxAllowanceForSpender` function instead of `SafeERC20Upgradeable.safeApprove`.

Fix 1.2

Fixed. The function is now implemented using `SafeERC20Upgradeable.forceApprove`.

Go back to Findings Summary

7. Report revision 1.2

The logic changes in the original codebase were performed as a response to found issues. Several original contracts contain changes for compatibility with newly introduced contracts.

7.1. System Overview

New contracts:

AmmLibEth

The library contract implements a single function for calculating the exchange ratio between `stEth` and `ipstEth`.

AmmPoolsLensEth

The contract implements a public view function for the `ipstEth` exchange rate. It uses the AmmLibEth library.

AmmPoolsServiceEth

The contract contains three functions for liquidity providing.

The function `provideLiquidityStEth` enables users to provide stEth and mint an IPOR version of the token - `ipstEth`

The function `provideLiquidityWEth` enables users to provide `wEth` with the following flow:

- The function calls `withdraw` on `wEth` contract, and Eth is received.
- The internal function `_depositEth` forwards Eth to Lido’s `stEth` contract using the `submit` function, resulting in received `stEth` tokens.
- The token `ipstEth` is minted and sent to the user.

The function `provideLiquidityEth` enables users to provide Eth with the following flow:

- The function receives ETH from the user.
- The internal function `_depositEth` forwards Eth to Lido's `stEth` contract using the `submit` function, resulting in received `stEth` tokens.
- The token `ipstEth` is minted and sent to the user.

The last function, `redeemFromAmmPoolStEth` burns the user's `ipstEth` tokens and sends `stEth` tokens to the user.

AmmTreasuryEth

The contract stores stEth tokens, pauses and unpauses the system, and provides several getters for the system state.

M7: No data validation while setting `redeemFeeRateEth`

Medium severity issue

Impact:	Medium	Likelihood:	Low
Target:	AmmPoolsServiceEth	Type:	Data validation

Description

During the contract AmmPoolsServiceEth creation, the parameter `ethRedeemFeeRateInput` is supplied to the constructor, and its value is assigned to `redeemFeeRateEth`. However, the parameter lacks any data validation.

The problem is that the immutable variable `redeemFeeRateEth` is used only at one place in the code, which is the function `redeemFromAmmPoolStEth`. The function will be called by users later in time because they first have to provide liquidity. Because of this, the potential wrong value in `redeemFeeRateEth` may be discovered too late.

Listing 42. AmmPoolsServiceEth

```
constructor(
    ...
    uint256 ethRedeemFeeRateInput
) {
    ...
    redeemFeeRateEth = ethRedeemFeeRateInput;
}
```

Listing 43. AmmPoolsServiceEth.redeemFromAmmPoolStEth

```
function redeemFromAmmPoolStEth(address beneficiary, uint256 ipTokenAmount)
external {

    ...
    uint256 exchangeRate = AmmLibEth.getExchangeRate(stEth, ipstEth,
ammTreasuryEth);

    uint256 stEthAmount = IporMath.division(ipTokenAmount * exchangeRate,
1e18);
    uint256 amountToRedeem = IporMath.division(stEthAmount * (1e18 -
redeemFeeRateEth), 1e18);

    require(amountToRedeem > 0,
AmmPoolsErrors.CANNOT_REDEEM_ASSET_AMOUNT_TOO_LOW);

    IIpToken(ipstEth).burn(msg.sender, ipTokenAmount);

    IStETH(stEth).safeTransferFrom(ammTreasuryEth, beneficiary,
amountToRedeem);
    ...

}
```

Vulnerability scenario

The contract's creator supplies the constructor with a wrong value or value in the wrong format. Because the variable `redeemFeeRateEth` is used only during liquidity redeem, the issue will be discovered too late when the pool already contains users' liquidity.

Recommendation

In general, data validations in the constructor are not that problematic when discovered soon or when the contract owner can reset wrongly supplied parameters. However, this mistake can stay hidden in this case because the value is used later in time. The owner also cannot change the variable's value (which is welcomed for a decentralized and trust model manner).

Perform data validation that allows the owner to set the fee rate in a specific range.

Go back to Findings Summary

I7: User can lose funds if the protocol is used incorrectly

Impact:	Info	Likelihood:	N/A
Target:	AmmPoolsServiceEth	Type:	Documentation

Description

The contract AmmPoolsServiceEth is meant to be used only through the protocol router contract IporProtocolRouter. However, at first sight, the contract seems to be usable directly, too. For most of the functions: `provideLiquidityStEth`, `provideLiquidityWEth`, and `redeemFromAmmPoolStEth`, the system will revert the transaction because the allowance is set to the router or mint functions are allowed to be performed by router only. But the last function, `provideLiquidityEth`, may be problematic if several **rare** conditions are met, and it can lead to the loss of users' funds.

Recommendation

Add NatSpec documentation to Service contracts with information that those contracts should be used only through the protocol router. If a well-informed user performs direct calls anyway, it can be considered the same as sending assets to zero address.

Fix 1.3

Comments were added to the contract to make it clear that the contract should be used only through the protocol router.

Go back to Findings Summary

I8: Mixing `_msgSender()` and `msg.sender` across the codebase

Impact:	Info	Likelihood:	N/A
Target:	AmmPoolsServiceEth, AmmTreasuryEth	Type:	Code quality

Description

Across the codebase, two different approaches for getting the message sender address are used. In the contract `AmmPoolsServiceEth`, the system variable `msg.sender` is used. In the contract `AmmTreasuryEth`, the function `_msgSender()` is used instead.

Recommendation

Choose one of the approaches and use it consistently.

Fix 1.3

The protocol is now using `msg.sender` only.

Go back to Findings Summary

I9: Redundant logging of `block.timestamp`

Impact:	Info	Likelihood:	N/A
Target:	AmmPoolsServiceEth	Type:	Code quality

Description

The contract `AmmPoolsServiceEth` is emitting `block.timestamp` in its events. It is redundant to emit `block.timestamp` as it can be read from a block's data.

Recommendation

If the back-end code is already implemented with a logic to read `block.timestamp` from a specific event, leave it as it is. Otherwise, use the block's data to read the timestamp.

Fix 1.3

The parameter `block.timestamp` was removed from events.

Go back to Findings Summary

8. Report revision 1.3

The updated codebase contains the following changes:

- The contract IporRiskManagementOracle was updated, and its functions are now working with additional parameters of demandSpreadFactor.
- The change of the logic for Spread calculation is reflected in contracts: AmmCloseSwapService, AmmOpenSwapService and contracts in `spread/` folder, where Time Weighted Notional calculation was modified.
- IvToken was removed and logic of AssetManagement is now working without the token
- Logic of pause pause guardians was updated inside the PauseManager contract

Additionally, the DSR strategy (which was previously audited) was added, and all other strategies were refactored together with AssetManagement con

M8: IPOR_508 reverts during deposit

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	AssetManagement	Type:	Math errors

Description

Due to rounding errors caused by different decimals of the underlying asset (DAI, for example) and the internal values used in the protocol (the values are typically in wads), asset management deposits frequently revert with the IPOR_508 error.

Listing 44. AssetManagement

```
require(
    tryDepositedAmount > 0 && tryDepositedAmount <= amount,
    AssetManagementErrors.STRATEGY_INCORRECT_DEPOSITED_AMOUNT
);
```

This issue was found using the Woke fuzzer. However, the source code of the fuzz test is not included in the report, as revert errors were occurring in the protocol while running the test for longer time periods. Due to insufficient time donation, finishing the fuzz test and finding the root causes of the revert errors was impossible.

Vulnerability scenario

A user wants to deposit tokens to the protocol. Due to the described issue, the transactions often fail, which causes losses to the user sending the transaction and reduces the credibility of the protocol.

Recommendation

Normalize the `amount` value describing the amount of tokens to be deposited to the protocol to the exact decimals as the returned `tryDepositedAmount` value.

Fix 1.4

The function now works with normalized value. The test is no longer returning the error `IPOR_508`.

Go back to Findings Summary

L4: Close swap insufficient balance revert

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	AmmCloseSwapService	Type:	Logic error

Description

Under an edge case scenario, when the treasury does not own enough tokens (the tokens are deposited to the asset management) and a swap with 100% loss is being closed by a 3rd party liquidator, the close swap transaction reverts with an insufficient balance error. This is caused by the logic of the `AmmCloseSwapService` contract that performs a rebalance between the treasury and asset management only when there is a non-zero payoff to the swap buyer. However, a swap with 100% loss has zero payoff to the buyer. Still, the liquidation deposit must be transferred to the liquidator.

Listing 45. AmmCloseSwapService

```
if (beneficiary == buyer) {
    wadTransferAmount = wadTransferAmount + wadLiquidationDepositAmount;
} else {
    /// @dev transfer liquidation deposit amount from AmmTreasury to
Liquidator address (beneficiary),
    /// transfer to be made outside this function, to avoid multiple
transfers
    wadPayoutForLiquidator = wadLiquidationDepositAmount;
}

if (wadTransferAmount > 0) {
    // perform rebalance
}
```

This issue was found using the Woke fuzzer. However, the *source code of the fuzz test is not included in the report, as revert errors were occurring in the protocol while running the test for longer time periods. Due to insufficient time donation, finishing the fuzz test and finding the root causes of the revert errors was impossible.*

Vulnerability scenario

A liquidator wants to close a swap with 100% or close to 100% loss. The payoff to the swap buyer is zero, and the treasury does not own enough tokens to cover the liquidation deposit. The liquidator sends a close swap transaction. The transaction reverts with an insufficient balance error.

Recommendation

Rewrite the problematic if condition as follows:

Listing 46. AmmCloseSwapService

```
if (wadTransferAmount + wadPayoutForLiquidator > 0) {
    // perform rebalance
}
```

Fix 1.4

The function was fixed as recommended.

Go back to Findings Summary

W10: Setting array max index in constructor

Impact:	Warning	Likelihood:	N/A
Target:	AssetManagement	Type:	Code quality

Description

The contract AssetManagement implements the constructor in a following way:

```
constructor(
    address assetInput,
    address ammTreasuryInput,
    uint256 supportedStrategiesVolumeInput,
    uint256 highestApyStrategyArrayIndexInput
) {
    asset = assetInput.checkAddress();
    ammTreasury = ammTreasuryInput.checkAddress();
    supportedStrategiesVolume = supportedStrategiesVolumeInput;
    highestApyStrategyArrayIndex = highestApyStrategyArrayIndexInput;

    require(_getDecimals() ==
IERC20MetadataUpgradeable(assetInput).decimals(),
IporErrors.WRONG_DECIMALS);
}
```

The contract is marked as `abstract`, and contracts for specific stable coins are inheriting it. The contract `AssetManagementDai` implements three strategies: `Compound`, `Aave`, and `Dsr`. For accessing strategies, the following function is used:

```
function _getStrategiesData() internal view override returns
(StrategyData[] memory strategies) {
    strategies = new StrategyData[](supportedStrategiesVolume);
    strategies[0].strategy = strategyAave;
    strategies[0].balance = IStrategy(strategyAave).balanceOf();
    strategies[1].strategy = strategyCompound;
    strategies[1].balance = IStrategy(strategyCompound).balanceOf();
    strategies[2].strategy = strategyDsr;
    strategies[2].balance = IStrategy(strategyDsr).balanceOf();
}
```

During the deployment of the contract `AssetManagementDai`, we have to specify two values (`supportedStrategiesVolumeInput`, `highestApyStrategyArrayIndexInput`) that correspond with the number of strategies used. As we can see, the array of strategies is hardcoded in the contract. For this reason, having the ability to set the array size and max index manually is **error-prone**. Variables `supportedStrategiesVolumeInput` and `highestApyStrategyArrayIndexInput` are used for accessing array values in functions like `withdraw` and `deposit`. If wrong values are passed to the constructor, the contract will not work as expected, and it can, for example, access random values in memory.

Recommendation

Hardcode the values of `supportedStrategiesVolumeInput` and `highestApyStrategyArrayIndexInput` instead of manually setting them in the constructor.

Fix 1.4

Variables `supportedStrategiesVolumeInput` and `highestApyStrategyArrayIndexInput` were removed from the constructor. The function `_getNumberOfSupportedStrategies` was added to strategies, returning the hardcoded number of supported strategies. The highest index is calculated based on the number of supported strategies.

Go back to Findings Summary

9. Report revision 1.4

The current update contains fixes of findings reported in Revision 1.3.

Additionally, hardcoded values for a spread slope were updated.

ackee | blockchain security

Appendix A: How to cite

Please cite this document as:

Ackee Blockchain, IPOR: protocol core, 25.9.2023.

Appendix B: Audit scope

```
contracts/
 ├─── amm
 │    ├─── libraries
 │    │    ├─── types
 │    │    │    ├─── AmmInternalTypes.sol
 │    │    │    └─── StorageInternalTypes.sol
 │    │    ├─── IporSwapLogic.sol
 │    │    ├─── SoapIndicatorLogic.sol
 │    │    └─── SoapIndicatorRebalanceLogic.sol
 │    ├─── spread
 │    │    ├─── CalculateTimeWeightedNotionalLibs.sol
 │    │    ├─── DemandSpreadLibs.sol
 │    │    ├─── ISpread28Days.sol
 │    │    ├─── ISpread28DaysLens.sol
 │    │    ├─── ISpread60Days.sol
 │    │    ├─── ISpread60DaysLens.sol
 │    │    ├─── ISpread90Days.sol
 │    │    ├─── ISpread90DaysLens.sol
 │    │    ├─── ISpreadCloseSwapService.sol
 │    │    ├─── ISpreadStorageLens.sol
 │    │    ├─── OfferedRateCalculationLibs.sol
 │    │    ├─── Spread28Days.sol
 │    │    ├─── Spread60Days.sol
 │    │    ├─── Spread90Days.sol
 │    │    ├─── SpreadAccessControl.sol
 │    │    ├─── SpreadCloseSwapService.sol
 │    │    ├─── SpreadRouter.sol
 │    │    ├─── SpreadStorageLens.sol
 │    │    ├─── SpreadStorageLibs.sol
 │    │    └─── SpreadTypes.sol
 │    ├─── AmmCloseSwapService.sol
 │    ├─── AmmGovernanceService.sol
 │    ├─── AmmOpenSwapService.sol
 │    ├─── AmmPoolsLens.sol
 │    ├─── AmmPoolsService.sol
 │    ├─── AmmStorage.sol
 │    ├─── AmmSwapsLens.sol
 │    ├─── AmmTreasury.sol
 │    └─── AssetManagementLens.sol
```

```
├──── governance
│     └──── AmmConfigurationManager.sol
├──── libraries
│     ├──── errors
│     │     ├──── AmmErrors.sol
│     │     ├──── AmmPoolsErrors.sol
│     │     ├──── AssetManagementErrors.sol
│     │     ├──── IporErrors.sol
│     │     ├──── IporOracleErrors.sol
│     │     └──── IporRiskManagementOracleErrors.sol
│     ├──── math
│     │     ├──── InterestRates.sol
│     │     ├──── InterestRatesMock.sol
│     │     └──── IporMath.sol
│     ├──── AmmLib.sol
│     ├──── AssetManagementLogic.sol
│     ├──── Constants.sol
│     ├──── IporContractValidator.sol
│     ├──── PaginationUtils.sol
│     ├──── RiskManagementLogic.sol
│     └──── StorageLib.sol
├──── oracles
│     ├──── libraries
│     │     ├──── IporLogic.sol
│     │     └──── IporRiskManagementOracleStorageTypes.sol
│     ├──── IporOracle.sol
│     ├──── IporRiskManagementOracle.sol
│     └──── OraclePublisher.sol
├──── router
│     ├──── AccessControl.sol
│     └──── IporProtocolRouter.sol
├──── security
│     ├──── IporOwnable.sol
│     ├──── IporOwnableUpgradeable.sol
│     ├──── OwnerManager.sol
│     └──── PauseManager.sol
├──── tokens
│     ├──── IporToken.sol
│     ├──── IpToken.sol
│     └──── IvToken.sol
├──── vault
    │
    ├──── strategies
    │     ├──── StrategyAave.sol
    │     ├──── StrategyCompound.sol
    │     └──── StrategyCore.sol
    ├──── AssetManagement.sol
    ├──── AssetManagementDai.sol
    ├──── AssetManagementUsdc.sol
    └──── AssetManagementUsdt.sol
```

Appendix C: Woke outputs

As a part of the audit, tests in Woke development and testing framework were implemented.

Non-fuzz unit tests were implemented to cover the following contracts:

- `AaveStrategy` including deposits and withdrawals
- `CompoundStrategy` including deposits and withdrawals
- `AssetManagement`
- `IporOwnable` and `IporOwnableUpgradeable` that extend `Ownable` and `OwnableUpgradeable` contracts from OpenZeppelin
- `PauseManager`

With the use of fork testing, the upgradeability of already deployed contracts was tested to ensure the storage layout of the contracts was not changed in a non-compatible way. Upgradeability of the following contracts was tested:

- `AssetManagement`
- `AaveStrategy`
- `CompoundStrategy`
- `IporOracle`
- `PowerToken`
- `LiquidityMining`
- `AmmStorage`

To test math libraries and evaluate the accuracy of the results, fuzz tests were prepared to test the following functionalities:

- `InterestRates` library is used to compute continuous compounding interest rates; the library uses ABDKMathQuad library for fixed-point arithmetic,
- `division` and `divisionInt` functions in the `IporMath` library.

Finally, a complex fuzz test was implemented to cover the most common use cases of the protocol entirely. The test covers the following scenarios:

- deposits and withdrawals of liquidity to and from the protocol,
- opening swaps with all accepted parameters being set by the fuzzer,
- closing swaps both before their maturity (triggers swap unwinding) and after their maturity by swap buyers and community liquidators,
- rebalancing between AMM treasuries and asset management contracts by appointed accounts, including adding and removing accounts appointed to rebalance.

Most of the parameters are randomly generated. These include the value of the liquidation deposit, publication fee, opening fee rate, unwinding fee rate, or treasury/asset management ratio. The rest of the parameters are set with reasonable values inspired mainly by the current state of the protocol and the documentation. The initial value of IPOR indexes is retrieved from the Ethereum Mainnet at a given block number. The IPOR indexes are randomly updated to fluctuate between 0% and 10%. Each fuzzer run generates completely new market conditions.

The following properties are checked during the fuzz test:

- balances of all accounts (including AMM treasuries) exactly match the expected values for all supported tokens,
- interim payoffs of all opened swaps and the final payoff of a swap being closed are observed with a reasonable error margin,

- SOAP value calculated by the contracts is close to the SOAP value calculated in the fuzzer using the direct iterative approach,
- non-LP balances (swap collaterals, liquidation deposits, collected fees) match the expected values with a reasonable error margin,
- the total balance reported by asset management before withdrawal from Aave and Compound and after the withdrawal is the same or almost the same,
- AMM treasury balances are observed to be within the expected range,
- IPOR treasury balances describing the amount of collected fees belonging to the IPOR protocol match the expected values,
- ip token exchange rates are close to the values calculated by the fuzzer,
- the amount of tokens transferred to a liquidity provider when redeeming liquidity matches the value calculated by the fuzzer while accounting for any math errors that occurred when providing liquidity,
- contracts revert only in documented cases.

The complex fuzz test was gradually developed during the audit and caught several bugs in the protocol, including C1, H1, H3, M1, M2 and M4. After the bugs were fixed in the contracts, the fuzz test was run for dozens of hours with all the tested properties passing.

Thank You

Ackee Blockchain a.s.

Prague, Czech Republic

hello@ackeeblockchain.com

https://twitter.com/AckeeBlockchain