RHO PROTOCOL SMART CONTRACTS SECURITY AUDIT REPORT

OXORIO

APRIL 10, 2024

CONTENTS

1 INTRO

1.1 DISCLAIMER

The audit makes no statements or warranties about the utility of the code, safety of the code, suitability of the business model, investment advice, endorsement of the platform or its products, regulatory regime for the business model, or any other statements about the fitness of the contracts to purpose, or their bug free status. The audit documentation is for discussion purposes only.

1.2 ABOUT OXORIO

Oxorio is a young but rapidly growing audit and consulting company in the field of the blockchain industry, providing consulting and security audits for organizations from all over the world. Oxorio has participated in multiple blockchain projects during which smart contract systems were designed and deployed by the company.

Oxorio is the creator, maintainer, and major contributor of several blockchain projects and employs more than 5 blockchain specialists to analyze and develop smart contracts.

Our contacts:

- oxor.io
- ping@oxor.io
- Github
- Linkedin
- Twitter

1.3 SECURITY ASSESSMENT METHODOLOGY

A group of auditors is involved in the work on this audit. Each of them checks the provided source code independently of each other in accordance with the security assessment methodology described below:

1. Project architecture review

Study the source code manually to find errors and bugs.

2. Check the code for known vulnerabilities from the list

Conduct a verification process of the code against the constantly updated list of already known vulnerabilities maintained by the company.

3. Architecture and structure check of the security model

Study the project documentation and its comparison against the code including the study of the comments and other technical papers.

4. Result's cross-check by different auditors

Normally the research of the project is done by more than two auditors. This is followed by a step of mutual cross-check process of the audit results between different task performers.

5. Report consolidation

Consolidation of the audited report from multiple auditors.

6. Reaudit of new editions

After the provided review and fixes from the client, the found issues are being double-checked. The results are provided in the new version of the audit.

7. Final audit report publication

The final audit version is provided to the client and also published on the official website of the company.

1.4 FINDINGS CLASSIFICATION

1.4.1 Severity Level Reference

The following severity levels were assigned to the issues described in the report:

- **CRITICAL**: A bug leading to the possibility of assets theft, locked fund access, or any other loss of funds due to transfer to unauthorized parties.
- **MAJOR**: A bug that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement.
- **WARNING**: A bug that can break the intended contract logic or expose it to DDoS attacks.
- **INFO**: Minor issue or recommendation reported to / acknowledged by the client's team.

1.4.2 Status Level Reference

Based on the feedback received from the client's team regarding the list of findings discovered by the contractor, the following statuses were assigned to the findings:

- **NEW**: Waiting for the project team's feedback.
- **FIXED**: Recommended fixes have been applied to the project code and the identified issue no longer affects the project's security.
- **ACKNOWLEDGED**: The project team is aware of this finding. Recommended fixes for this finding are planned to be made. This finding does not affect the overall security of the project.
- **NO ISSUE**: Finding does not affect the overall security of the project and does not violate the logic of its work.
- **DISMISSED**: The issue or recommendation was dismissed by the client.

1.5 PROJECT OVERVIEW

Rho Protocol is a novel decentralized marketplace allowing professional traders to efficiently exchange rate risk through on-chain swaps and futures. Rho Protocol brings interest rate derivatives to the DeFi ecosystem. It is designed for institutions and provides features uniquely catering to this client base, including innovative market pricing (`vAMM`), risk management, and compliance (permissioned market sections) solutions. By becoming a pre-eminent digital asset rates market, Rho Protocol is well-positioned to offer on-chain rates trading and management for the arrival of major institutions and securities tokenization.

1.6 AUDIT SCOPE

- `access-control`
- `collateral`
- `configuration`
- `future`
- `index-oracles`
- `issuer`
- `libraries`
- `market`
- `router`
- `utils`
- `vamm`

The audited commit identifier is `18561a80c5199fd7dbfbe53dbb6516f17aa3ddbc`.

2 FINDINGS REPORT

OXORIO

2.1 CRITICAL

C-01	Tick handling in `VAMM`, `LiquidityLogic` can lead to losses in certain cases when the market rate falls on the interval boundary
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
VAMM.sol	contract `VAMM`	386
SwapLogic.sol	contract `SwapLogic`	133
SwapLogic.sol	contract `SwapLogic`	174
LiquidationLogic.sol	contract `LiquidationLogic`	158
VAMM.sol	contract `VAMM`	439
LiquidationLogic.sol	contract `LiquidationLogic`	524

Description

In the function `provideLiquidity` in the `VAMM` contract, it's possible to add liquidity twice to the same interval by setting the `VAMM` state when the rate is sitting right on the boundary of the interval. For example:

1. The initial rate is `0.1`.
2. Maker adds liquidity to the `0.099`-`0.1` interval, and the notional amount is `1`. The `intervalLiquidity` variable, calculated based on density, is `10583236255`.
3. Taker opens a short position with a notional of `1`. During the `swap` function execution, there will be two crosses between ticks. The first cross will happen from the `0.1`-`0.101` interval (since `0.1` was the initial) to start the swap. However, since the swap is executing for the amount of 1 notional (which is all the amount on the interval), the resulting rate will become `0.099`. This rate is equal to the upper boundary of the next interval, and one more cross will occur:

```
result.needToCross = result.rate == result.intervalInfo.nextRate;
```

After that, the current rate is still `0.099`, but the current interval is `0.098-0.099`.
4. Maker adds liquidity one more time to the `0.099-0.1` interval, and the notional amount is `1`. During the liquidity provision, we fall into the `if` statement since both the `0.099-0.1` interval is within the `0.099` current tick:

```
if (currentRate >= bounds.lower && currentRate < bounds.upper) {
  _storage.setCurrentIntervalNotionalDensity(_storage.getCurrentIntervalNotionalDensity() +
notionalDensityDelta);
}
```

However, the `getCurrentIntervalNotionalDensity` view function will return the notional density of the `0.098-0.099` interval, since the state of the `VAMM` was updated during the previous step. The `getCurrentIntervalNotionalDensity` function will return `0` value of the `0.098-0.099` interval liquidity, while it should have returned `1000000` value of the `0.099-0.1` interval liquidity.
5. Taker opens the long position for `3` notional, and the first step of the `swap` is executed with the `intervalLiquidity` value of `10572652009`, and after that one more cross is executed, with additional `intervalLiquidity` of `31749742022` generated from nowhere. Under normal conditions, the `intervalLiquidity` should be equal to the `21166490801` value, which is the actual provided liquidity to the `VAMM`. This notional is created from nowhere, allowing the creation of an infinite amount of liquidity.

The Proof of Concept of the double add vulnerability is here.

Similar to the incorrect tick handling in the `VAMM` contract, in the function `liquidatePositions` in the `LiquidationLogic` contract, liquidations can be frontrun by users by setting the VAMM state when the rate is on the boundary of the interval. This leads to a revert during the liquidation because, in the function `removeLiquidity` in the `VAMM` contract, the removal of liquidity will revert. The `liquidatePositions` function will also revert since the `removeLiquidity` function is called with the `_liquidateAllFutureProvisions` function call.

For example:

1. A liquidator starts liquidating an underwater position with the `liquidatePositions` call.
2. An attacker frontruns the liquidation by setting the VAMM state when the rate is on the boundary of the interval.

3. The liquidation function is executed and reverted during the `removeLiquidity` function call. Liquidity cannot be removed from the interval since, during the liquidity removal, the code falls into the `if` statement:

```
    if (currentRate >= bounds.lower && currentRate < bounds.upper) {

_storage.setCurrentIntervalNotionalDensity(_storage.getCurrentIntervalNotionalDensity() -
notionalDensityDelta);
    }
```

The `getCurrentIntervalNotionalDensity` view function returns the notional density of the interval previous to the interval from which liquidity is attempted to be removed. The remove liquidity function returns `0` value for the previous interval liquidity (if, for example, there is no liquidity on the previous interval), while it should have returned the amount of notional value on the current interval liquidity. This makes it impossible to liquidate users until the state of the VAMM is changed to the correct tick.
4. When the liquidation transaction fails, the hacker hedges his previous position.

The Proof of Concept of frontrunning case is here.

Recommendation

We recommend reviewing existing logic, adding additional checks for situations when the rate is located on the lower boundary of one interval and the higher boundary of the other interval. This could be done by adding additional flags to the storage of the `VAMM` contract after swaps and validating the state in case `currentRate >= bounds.lower`. So the case `currentRate > bounds.lower && currentRate < bounds.upper` can be handled in the way it is handled right now, while the case `currentRate == bounds.lower` should be handled differently. These changes must be applied to the `provideLiquidity`, `removeLiquidity` functions, and functions in the `RatePoint` library. We also recommend adding fuzzing tests to the contracts to ensure that all calculations work correctly and as expected.

Update

Fixed in commits `4b4d43b3aafe5655bb248f4a659cf54974420250`, `5bc610348e86ef1933fd7f2a0fed254be85b6363`.

Rh0's response

Added logic to handle the exception when the current market rate is equal to the lower bound of the interval.

Oxorio's response

Issue is fixed in the `4b4d43b3aafe5655bb248f4a659cf54974420250`, `5bc610348e86ef1933fd7f2a0fed254be85b6363` commits.

C-02	Excess number of maker provisions in a single market may lead to a gas bomb in `LiquidationLogic`, `FuturLogic`, `Future`
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
LiquidationLogic.sol	contract `LiquidationLogic`	158
LiquidationLogic.sol	contract `LiquidationLogic`	71
LiquidationLogic.sol	contract `LiquidationLogic`	267
FutureLogic.sol	contract `FutureLogic`	112
Future.sol	contract `Future`	212
FutureLogic.sol	contract `FutureLogic`	78
FutureLogic.sol	contract `FutureLogic`	102

Description

In the `liquidatePositions`, `cancelProvisions`, and `transferPositionsOwnership` functions of the `LiquidationLogic` contract, there are numerous loops over the `_makerProvisions` mapping, as well as several loops over the `unsettledFutures` mapping, and other gas-heavy logic that can lead to the out-of-gas error. This problem can be exploited by a malicious user to block liquidations, resulting in bad debt for the project.

A possible scenario for a malicious user to block liquidation functions:

1. The malicious user creates a large number of provisions. By providing liquidity to different bounds and immediately removing liquidity, the user generates numerous provisions with different bounds. For instance, if a VAMM has `200` intervals with `0.001` tick spacing, the user can create `200` provisions with liquidity distributed on `1` tick, `199` provisions with liquidity distributed on `2` ticks, and so on. This method of providing liquidity doesn't require substantial funds for the attacker, as the amount of liquidity provided is minimal and removed immediately, with no fees acquired by the protocol besides the blockchain gas fee.
2. The user establishes a position with high leverage. The collateral at risk is a minimal amount, while the position size and potential profit are substantial.

3. When the position goes underwater due to rate movement in the opposite direction, the user avoids liquidation since the liquidation function reverts with an out-of-gas error. Consequently, the protocol incurs losses, and bad debt increases.
4. After some time, the rate changes, the position emerges from underwater, and when the future expires, the user can settle the position and withdraw funds. Other functions in the protocol also require a significant amount of gas for execution; however, these functions are more gas-efficient compared to the liquidation functions, allowing the user to execute them while the liquidation functions are blocked.

The Proof of Concept for the described case is here.

Also, in the functions of the the `FutureLogic` and `Future` contracts:

- `makerProvisionsInfo`
- `isEmptyProvision`
- `openMakerPosition`
- `makerLiquidityDistribution`

There is a loop over the `_makerProvisions` mapping, which can lead to the out-of-gas error. Any user can create a large number of provisions by providing liquidity over a wide range of bounds or managing existing liquidity by removing some of the liquidity to other boundaries, which is an expected behavior. However, after reaching the gas limit, the functions mentioned above will revert with out-of-gas errors. The problem is that functions such as `withdraw`, `executeTrade`, `settleMaturedPositions`, `margin`, `profitAndLoss`, and many others use these functions and will also revert with out-of-gas errors. A significant amount of maker provisions will block all functions of the protocol for a user, without any possibility to reduce the number of maker provisions or withdraw funds from the protocol. The user's balance will be stuck forever in the protocol.

Recommendation

We recommend reviewing the existing logic, limiting the number of provisions that can be created by the user, adding functions for deleting unused maker provisions, optimizing the gas usage of the liquidation functions, and reducing the number of loops over the mappings.

Update

Fixed in commits `a35700574ae1457d07f8d8c1b82b0cc6ba72afd9`, `97a431bff30794a89867fadae4536bfdf70e8a70`, `44c7687fd5484f9a65265b25ea5e448aef9e9935`, `5376c01512aded2ce208807c063546d99703264f`, `71bee2649fa29f54248c354aa41eed61e83f1f99`.

Rh0's response

A limit for possible maker positions has been introduced to mitigate this risk. Also provisions that were reduced to 0 get cleaned up and are no longer processed in AMM.

Rh0's response

C-03 - requesting a merge with C-02. We want to request a merge into a single issue with C-02, describing both sets of implications for Futures and Liquidity. The same logical fix has been applied to address both issues.

Oxorio's response

The C-02 and C-03 findings were merged into one, the problem was fixed, however the fixes of the initial problem created new issues.
In the `97a431bff30794a89867fadae4536bfdf70e8a70` commit in the `SettlingLogic` library the `UNSETTLED_FUTURES_LIMIT` is a `constant` variable, which can't be changed with the regular flow of the contract. If in future, the gas cost of the opcodes will change, the protocol will not have an ability to update the `UNSETTLED_FUTURES_LIMIT` variable.
At the same time, the protocol uses UUPS proxy pattern, which means that the `UNSETTLED_FUTURES_LIMIT` `constant` variable is stored in the bytecode of the implementation, which makes it possible to change the `UNSETTLED_FUTURES_LIMIT` variable by migrating to another implementation.
We recommend clarifying whether the `UNSETTLED_FUTURES_LIMIT` must be configurable or not, adding functions for changing the `UNSETTLED_FUTURES_LIMIT` variable, storing the variable in the storage of the contract rather than in the bytecode of the implementation.
In the `97a431bff30794a89867fadae4536bfdf70e8a70` commit in the `ProvisionsLimitChecker` library there is a `ERC165Checker.supportsInterface` check to the `futureStorage` contract in the `if` statement, which doesn't implement the else condition. At the same time in the `71bee2649fa29f54248c354aa41eed61e83f1f99` commit in the `removeLiquidity` function of `FutureLogic` library there is the same `if` statement with the `ERC165Checker.supportsInterface` check to the `futureStorage` contract and a separate `else` condition. If the `futureStorage` doesn't implement new interface, the

```
result += IFutureStorageExtensionV1(address(futureStorage)).getMakerProvisionsSize(user)
```

operation won't occur. We recommend clarifying the use of the `else` condition for cases, when the `futureStorage` doesn't inherit correct interface as it is done in other parts of the code, adding comments to this section.

Rh0's response

Regarding the C-02, I added a couple of comments to the code and handled the `else` branch, which was not there previously: https://github.com/RhoLabs/rho-contracts-v1/commit/616c9dae592d04737a0b4d34ea341376584a0ff1

Originally, the `else` branch was not introduced because new features are released with new code. However, for transparency and based on your recommendation, we decided to add it after all.

Updating limits on the number of positions is considered purely a technical necessity. Therefore, executing this through replacing the implementation of contracts seems fine. We currently do not plan to add any specific role responsible for setting these limits through contract functions. Also, updates to the limits are likely to be associated with changes/optimizations in the code. This way, limits and code can be updated atomically within a single transaction.

Oxorio's response

The problem was fixed in the `616c9dae592d04737a0b4d34ea341376584a0ff1` commit.

C-03	Multiple unsettled futures lead to gas bomb in `LiquidationLogic`
Severity	CRITICAL
Status	• FIXED

Description

Creating multiple unsettled positions can lead to the blocking of liquidations, withdrawals, and other operations. Over the protocol's lifetime, markets will have many different futures, both active and finished. Since only a user can settle their futures, it's possible to create numerous unsettled futures with the `provideLiquidity` or `transferPositionsOwnership` functions. A large quantity of unsettled futures for a user poses a significant problem for the protocol. For example, it becomes unprofitable to liquidate small positions of the user with many unsettled futures because the `transferPositionsOwnership` and `liquidatePositions` functions are gas-heavy. Therefore, positions might not be liquidated even if the liquidation transaction gas fee is 1 ETH, and the amount for liquidation is 0.5 ETH.

Furthermore, having multiple loops over unsettled futures in gas-heavy functions exacerbates the situation:
In the `liquidatePositions` function, there is a for loop of unsettled futures in the `margin` function with multiple external calls, calculating the `profitAndLoss` of the user. After that, in the `initialMarginThreshold` function, there is a for loop of unsettled futures. Following that, there is a loop in the `lpMarginThreshold` function, and after that, there is a for loop in the `liquidationMarginThreshold` function with multiple external calls. After that, there is another for loop in the `_performLiquidationTrades` function with gas-heavy logic and external calls. Right after that, there is one more `margin` and `initialMarginThreshold` calls with for loops. In the very end, in the `MarginUpdate` event, there is one more `margin` call.

In total, there are at least 8 for loops for the length of the unsettled futures of the user in the `liquidatePositions` function. This implies that, to liquidate a user with only 3 active futures, it's needed to loop over each of these futures 24 times, executing multiple external calls and other gas-heavy operations. This architecture creates risks with a gas bomb attack, where the user creates multiple futures to revert the liquidation function.

It's worth mentioning that in multiple other functions with unsettled futures, this problem exists. In the worst-case scenario, the loop over unsettled futures will revert the `withdraw` function in the `Router` contract with an out-of-gas error, blocking all user funds.

Recommendation

We recommend refactoring the current architecture, reviewing the usage of unsettled futures in other functions, and limiting the number of unsettled futures for users.

Update

Fixed in commits `a35700574ae1457d07f8d8c1b82b0cc6ba72afd9`, `97a431bff30794a89867fadae4536bfdf70e8a70`.

Rh0's response

Introduced a limit for a number of unsettled futures. Settlement is now forced if the number of processed futures exceeds the limit.

Oxorio's response

The issue has been fixed in the `a35700574ae1457d07f8d8c1b82b0cc6ba72afd9`, `97a431bff30794a89867fadae4536bfdf70e8a70` commits.

C-04	Current rate manipulation in `SwapLogic` can lead to misappropriation of collateral from `CollateralManager` by malicious actor.
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
SwapLogic.sol	contract `SwapLogic` > function `calcSwapStepState`	164

Description

In the `calcSwapStepState` function of the `SwapLogic` contract, a malicious user can steal all the collateral from the `collateralManager` by manipulating the current rate.

When calculating a swap on the last interval, a new rate is determined based on a new price. The calculation of the new price, in turn, depends on the current price in the interval, and the current price is derived from the current rate.

Thus, the transformation chain is as follows:

```
currentRate->currentPrice -> targetPrice->targetRate
```

Here's how it happens in the code:

```
// Conversion of currentPrice from currentRate
UD60x18 currentPriceSqrt = uSqrt(rateToPrice(params.futureRate, params.indexWithMaturity,
params.rateMath));

// Calculation of targetPrice
targetPriceWithSqrt.priceSqrt = params.direction == RiskDirection.Value.PAYER
    ? calcTargetPriceSqrtByYAmount(params.targetFloatTokenAmount, intervalLiquidity,
currentPriceSqrt, false)
    : calcTargetPriceSqrtByYAmount(params.targetFloatTokenAmount, intervalLiquidity,
currentPriceSqrt, true);

targetPriceWithSqrt.price = targetPriceWithSqrt.priceSqrt * targetPriceWithSqrt.priceSqrt;
```

```
// Conversion of targetRate from targetPrice
result.rate = params.rateMath.priceToRate(
    params.rateMath.rebaseFloatTokenToNotional(targetPriceWithSqrt.price,
params.indexWithMaturity.floatIndex),
    params.indexWithMaturity.timeToMaturity
);
```

During these transformations, there are rounding errors in the calculations. Because of this, in the case of a small change in price, inaccuracies occur. For example, if a `payer` creates a long position and the rate was supposed to increase (`targetRate > currentRate`), a situation may arise where after the swap, the rate decreases (`targetRate < currentRate`).

Let's take the following VAMM state as an example and assume that we are swapping as a `payer`, creating a long position. Get the `currentPrice` from the `currentRate`:

```
currentRate = 101000000000000000
timeToMaturity = 299591870000000000000000000
floatIndex = 1100000000000000000

price = rateToPrice(currentRate, timeToMaturity) = 1/(1 + currentRate)^t = 912645389562917100
currentPrice = rebaseNotionalToFloatToken(price, floatIndex) = price/floatIndex =
829677626875379181

sqrtCurrentPrice = sqrt(currentPrice) = 910866415494269489
```

Since we are a payer, we take float tokens from the pool and bring in fixed tokens. Accordingly, the price - the ratio of float tokens to fixed tokens - will decrease. Suppose we made a very small change in price - by one unit of `sqrtCurrentPrice`, then we determine `targetRate` from the new price - `targetPrice`:

```
sqrtTargetPrice = sqrtCurrentPrice - 1 = 910866415494269488
timeToMaturity = 299591870000000000000000000
floatIndex = 1100000000000000000

targetPrice = sqrtTargetPrice * sqrtTargetPrice = 829677626875379178
price = rebaseFloatTokenToNotional(targetPrice, floatIndex) = targetPrice * floatIndex =
912645389562917095
targetRate = priceToRate(price, timeToMaturity) = 1/(price**(1/t)) - 1 = 100999999999999987

targetRate < currentRate
```

Thus, with a slight change in price during a swap due to a long position, the rate of the future shifts not upwards but downwards. And, because the current rate = `101000000000000000` is the interval boundary, then the new future rate = `100999999999999987` is in the adjacent interval.

Such a slight change in price can be achieved if there is a large amount of liquidity in the interval, while the swap is done with a very small notional amount.

This leads to the ability to steal collateral from `collateralManager`, because of incorrect calculation of LP fee for liquidity providers. Let's consider an attack scenario:

- The current rate of the future is set at `0.1`, `intervalLength` is `0.001`, `floatIndex` is `1.1`, and the token `decimal` is `6`.
- The user deposits `1000000002000003` tokens to the `collateralManager`.
- The user provides liquidity in two intervals:
  - `1000000` token for the interval `[0.1; 0.101]`
  - `1000000000000000` tokens for the interval `[0.101; 0.102]`
- The user creates a long position by swapping a notional of `1000003` tokens:
  - He exchanges `1000000` tokens in the interval `[0.1; 0.101]`
  - and another `3` tokens in the interval `[0.101; 0.102]`
- As a result of this swap, the new rate is set to `100999999999999987` - below the boundary of the interval `[0.101; 0.102]`. In this case, the logic of handling boundary crossing did not work; the `cross` function from the `RatePoint` library was not called, and as a result, the parameter `cumulativeAccruedLPFeeOutside` was not updated. Also, `_currentIntervalNotionalDensity` from the `SwapLogic` library did not update.
- As a result, the `balance` function for the user returns an LP fee value of `10000000000000` tokens, and the user's margin will be significantly more than his entire deposit:
  - total margin before the swap - `1000000002000003`
  - total margin after the swap - `1009527260560516`
- Next, the user can remove liquidity from the `[0.101; 0.102]` interval and create a long position on the same notional amount, reducing their `initialMarginThreshold` to a level where he can withdraw collateral before the future execution is complete. And he can withdraw more collateral than he deposited in total - `1007079623059571 > 1000000002000003`.

Here is a Proof of Concept of the case described above.

Recommendation

We recommend reviewing the logic to prevent setting the rate outside the current interval during a swap. We also recommend adding fuzzing tests to the contracts to ensure that all calculations work correctly and as expected.

Update

Fixed in commits `071c43897fa3f05ae34b581d3a0b0ae94581f145`, `221efde3110cadc678dfffecd2c2d87cbb4ea934`.

Rh0's response

We have added restrictions for the target price so that it cannot go beyond the interval.

Oxorio's response

In the `071c43897fa3f05ae34b581d3a0b0ae94581f145` commit in the `calcSwapStepState` function of `SwapLogic` library was added logic which "caps" the price to the current interval. In the `221efde3110cadc678dfffecd2c2d87cbb4ea934` commit there is a test case scenario, where after the swap the rate is "capped" from the `100999999999999964` value to the `101000000000000000` value in order to move the rate to the correct tick. However, because of this "cap", small amount of fixed tokens is not exchanged to the float tokens. This leads to the situation, when the exchange curve is moving higher on the ordinate axis, creating an inflation problem between the fixed and float tokens rates. Considering the amounts from the `221..934` commits test case, the user by swapping `1.000003` of notional tokes has spent `1019348` fixed tokens in order to shift the rate to the `101000000000000000` with the "cap" operation, but without the "cap" operation the user would need to swap of `1.000047` of notional tokes, spending `1019395` of fixed tokens in order to drive the rate to the `101000000000000005`, so because of the "cap" operation the disbalance in the pool is about `47` tokens, and the rate between the fixed and float tokens is lightly inflated.
Despite the fact that the inflation is very small, it's worth considering that this can happen unlimited times in the pool, and the amount of the inflation highly depends on the configuration of the pool and amount of the decimals of notional token. This inflation can lead to the full halt of the trading, because of the calculations of `result.tradeRate` variable in `SwapLogic` library:

```
    // With the inflated rate, tradeRate variable is also inflated, trading stops
    // because of the `TradeRateLimitIsExceeded` error
    result.tradeRate = params.rateMath.priceToRate(
      params.rateMath.rebaseFloatTokenToNotional(
        params.floatTokenDelta / swapState.fixedTokenDelta, // uses the rate between fixed
and float tokens
        params.indexWithMaturity.floatIndex
      ),
      params.indexWithMaturity.timeToMaturity
    );
```

As a Proof of Concept, it will be easier to review the case when the amount of fixed and float tokens after the swap is very small and not precise because of this, since with the current configuration of the pool from the test case it will take an enormous amount of steps:

```
  // minNotional is changed to toBn('0.000001', 6) in order to show the case
  it.only('Check interval cap when the liquidity amount is large and the notional amount is small', async function () {
    const { router, erc20Token, future1Id, oraclePackage, marketId } =
      await loadFixture(deployCompoundingVersion)
    const [, maker1, taker1] = await ethers.getSigners()
    const decimals = await erc20Token.decimals()
    // Mint tokens
    const makerDepositAmount = toBn('10000001.000003', decimals)
    await mintAndApprove(router, erc20Token, [maker1], makerDepositAmount)
    const takerDepositAmount = toBn('10', decimals)
    await mintAndApprove(router, erc20Token, [taker1], takerDepositAmount)
    const currentTs = await getCurrentBlockTimestamp()
    const deadline = BigNumber.from(currentTs + 1000 * 1000)
    // Deposit tokens for maker and taker. Taker just deposits and does nothing with his collateral.
    await router.connect(maker1).deposit(marketId, maker1.getAddress(), makerDepositAmount, true, [oraclePackage])
    await router.connect(taker1).deposit(marketId, taker1.getAddress(), takerDepositAmount, true, [oraclePackage])
    // Maker provides liquidity in two intervals:
    // - Liquidity is set to 1 token for the interval [0.1; 0.101]
    // - Liquidity is set to 100000000000000 tokens for the interval [0.101; 0.102]
    // The initial rate is 0.1
    await router
      .connect(maker1)
      .provideLiquidity(
        future1Id,
        toBn('0.000001', decimals),
        toBn('0', decimals),
        toBn((100 / 1000).toString()),
        toBn((101 / 1000).toString()),
        deadline,
        false,
        [oraclePackage]
      )
    await router
      .connect(maker1)
      .provideLiquidity(
        future1Id,
```

```
        toBn('100000000', decimals),
        toBn('0', decimals),
        toBn((101 / 1000).toString()),
        toBn((102 / 1000).toString()),
        deadline,
        false,
        [oraclePackage]
      )
    // Maker swaps 0.000003 tokens of notional.
    // Such small change of notional results in the small change of fixed token and float
tokens in the pool
    // The rate between fixed and float tokens inflates, resulting in not precise
calculation of tradeRate
    // Because of the inflated fixed and float tokens deltas, the tradeRate is also
inflated, trade reverts with
    // TradeRateLimitIsExceeded error, despite the fact that we have exchanged very small
amount of notional
    // In this case, fixed token delta is 3, float token delta is 2, the rate is
666666666666666666
    // Which results in 3715134049587793631 rate impact
    await expect(router
      .connect(maker1)
      .executeTrade(
        future1Id,
        riskDirection.PAYER,
        toBn('0.000003', decimals),
        toBn('1'),
        toBn('0', decimals),
        deadline,
        false,
        [oraclePackage]
      )).to.be.revertedWithCustomError(router, "TradeRateLimitIsExceeded");

      // Under normal conditions with the swap of 1 token of notional
      // Fixed token delta is 1019430, float token delta is 909090, the rate is
891763044054030193
      // Which results in 101005024060487136 rate impact
      // So, with the larger of notional the final rate is considered to be smaller
      // This means that with the inflated rate between fixed and float tokens, trading
becomes unaccessible
      await router
      .connect(maker1)
      .executeTrade(
        future1Id,
        riskDirection.PAYER,
```

```
        toBn('1', decimals),
        toBn('1'),
        toBn('0', decimals),
        deadline,
        false,
        [oraclePackage]
      )
  })
```

In the following PoC, the swap reverts with the `TradeRateLimitIsExceeded` error because of the inflation problem, even though we are exchanging a very small amount of tokens.
We recommend reviewing this scenario, and adding the limitation on the max liquidity of the tick (Uniswap V3 as an example) in order to minimize the risks.

Rh0's response

Regarding the finding in C-04, we didn't find any ways to exploit it. We agree that there is negligibly small error under certain conditions, but considering it as accumulative with each trade is not deemed correct, as I described yesterday: After each trade, the market operates based on the future rate that arises after that trade, and further calculations are based on this new future rate, rather than some implied, albeit more accurate rate. It seems incorrect to compute an exact rate based on the entire history and rely on it instead of the rate set in the pool. Even if we disregard a specific cap operation, the error in future rate calculations will accumulate with each operation. If we perform historical calculations of all trades with higher precision than currently available, we will obtain a different rate, but that does not mean that the market considers this rate correct, as each new trade is based on the rate that preceded it.

The `TradeRateLimitIsExceeded` error mentioned in the test occurs when setting the minimum allowable notional to `0.000001` for the USDT analogue. We introduced this restriction precisely to avoid such errors and accuracy issues with small trade sizes. In the mainnet, we use a value of `1.0`, so the case mentioned is not possible.

Oxorio's response

We recommend implementing fuzzing tests for this specific case with various pool configurations, in order to make sure that there is no exploitations cases.

C-05	Rounding in `SwapLogic` may lead to discrepancies in the amounts of fixed and floating tokens being exchanged in a trade
Severity	CRITICAL
Status	• FIXED

Location

File	Location	Line
SwapLogic.sol	contract `SwapLogic` > function `calcSwapStepState`	159

Description

In the `calcSwapStepState` function of the `SwapLogic` contract, after determining the new price, the calculation of `fixedTokenDelta` occurs on the current interval. It depends on the difference between the current and new prices:

```
result.fixedTokenDelta = nextPointFloatAmount == PrbMath.UNSIGNED_ZERO
  ? PrbMath.UNSIGNED_ZERO
  : calcXAmountSqrt(intervalLiquidity, currentPriceSqrt, targetPriceWithSqrt.priceSqrt);
```

The formula can be simplified as follows:

```
fixedTokenDelta = L * (1/sqrt(targetPrice) - 1/sqrt(currentPrice))
```

That is, if the current and new prices are equal, then `fixedTokenDelta` will be `0`. However, in the code, there is a check for a zero `fixedTokenDelta`, but it occurs after the results of the entire swap, not at each interval.

All of this leads to the fact that a `payer` can receive `floatTokenAmount` during a swap without spending `fixedTokenDelta`.

A situation with equal prices is possible if we exchange a very small value of `floatTokenAmount` with very high liquidity in the interval:

```
targetPriceWithSqrt.priceSqrt = params.direction == RiskDirection.Value.PAYER
    ? calcTargetPriceSqrtByYAmount(params.targetFloatTokenAmount, intervalLiquidity,
```

```
currentPriceSqrt, false)
    : calcTargetPriceSqrtByYAmount(params.targetFloatTokenAmount, intervalLiquidity,
currentPriceSqrt, true);
```

The formula can be simplified as follows:

```
sqrtTargetPrice = sqrtCurrentPrice ± (floatTokenAmount/liquidity)
```

So, with high `liquidity` and low `floatTokenAmount`, their ratio can be equal to `0` due to rounding.

Consider such a scenario:

- The current rate of the future is set at `0.1`, `intervalLength` is `0.001`, `floatIndex` is `1.1`, and the token `decimal` is `6`
- The maker provides liquidity in two intervals:
  - `1000000` tokens for the interval `[0.1; 0.101]`
  - `10000000000000000000` tokens for the interval `[0.101; 0.102]`
- The taker creates a long position by swapping a notional of `2` tokens:
  - They exchange `1000000` tokens in the interval `[0.1; 0.101]`
  - and another `1000000` tokens in the interval `[0.101; 0.102]`
- During the swap, on the second interval, the taker will exchange `1000000` tokens, i.e., `floatTokenAmount = 1_token / floatIndex = 909091`. At the same time, the liquidity on the interval will be equal to `L = 2314927140438373688847623`
- As a result of dividing `floatTokenAmount/L = 909091/2314927140438373688847623 = 0`, we get `0` due to rounding a very small value, and in turn: `sqrtTargetPrice == sqrtCurrentPrice`

In the end, the taker received `floatTokenAmount = 909091`, while spending `fixedTokenDelta = 0`.

This can be exploited by performing such exchanges several times in a row. For this, the taker needs to make a short position to roll the price back to the `[0.1; 0.101]` interval and make a long position again. Thus, he will bypass the `ZeroFixedTokenAmount` error since he will spend some amount of `fixedTokenAmount` in the first interval - `[0.1; 0.101]`. So, the following condition will not trigger:

```
if (swapState.fixedTokenDelta == PrbMath.UNSIGNED_ZERO) revert
IVAMMErrors.ZeroFixedTokenAmount();
```

Here is a Proof of Concept of the case described above.

Recommendation

We recommend revising the logic of the code to avoid a situation where, as a result of a swap on an interval, `floatTokenAmount` is exchanged with a zero `fixedTokenDelta`. We also suggest reviewing the methods for rounding numbers after arithmetic operations and considering increasing the level of precision.

Update

Fixed in commits `230d987089c6140f9d22a58081c7b8a211f2b02c`, `24b962156a67c70f7254d4495c72cab84f36b280`.

Rh0's response

We have changed the logic for calculating the target price for the payer case.
If the price moves down (payer shifts the rate up), we use rounding up to maximize the delta and the number of fixed tokens.
If the price moves up (receiver shifts the rate down), we use rounding down to minimize the delta and the number of fixed tokens.

Oxorio's response

The initial issue has been fixed, and now the output token amount is always minimized.
This fix has created a "feature" - based on the payer notional input it's possible to receive twice the amount of float token, while exchanging the exact same amount of fixed tokens:

```
  it.only('fixedTokenAmount is not zero when the liquidity amount is large and the notional amount is small', async function () {
    const {
      router,
      erc20Token,
      future1Id,
      future2Id,
      oraclePackage,
      marketId,
      contractProvider,
    } = await loadFixture(deployCompoundingVersion)
    const [, maker1, maker2, taker1, taker2] = await ethers.getSigners()
    const decimals = await erc20Token.decimals()
    // Mint tokens for users
    const initialBalance = 2000000000000000
    const convertedInitialBalance = toBn(initialBalance.toString(), decimals)
    await mintAndApprove(router, erc20Token, [maker1, maker2, taker1, taker2], convertedInitialBalance)
    const currentTs = await getCurrentBlockTimestamp()
```

```
    const deadline = BigNumber.from(currentTs + 1000 * 1000)
    const depositAmount = toBn('250', decimals)
    // Get the VAMM object
    const future2 = await ethers.getContractAt('Future', await
contractProvider.getFutureAddressById(future2Id))
    const vammAddress = await future2.getVAMM()
    // - Liquidity is set to 1000000 token for the interval from 0.1 to 0.101
    // - Liquidity is set to 100000000000000000000 tokens for the interval from 0.101 to
0.102
    // Then, the taker creates a long position.
    // The initial rate is 0.1.
    await router
      .connect(maker1)
      .provideLiquidity(
        future1Id,
        toBn('1', decimals),
        toBn('1', decimals),
        toBn((100 / 1000).toString()),
        toBn((101 / 1000).toString()),
        deadline,
        false,
        [oraclePackage]
      )
    await router
      .connect(maker1)
      .provideLiquidity(
        future1Id,
        toBn('1000000000000000', decimals),
        toBn('1000000000000000', decimals),
        toBn((101 / 1000).toString()),
        toBn((102 / 1000).toString()),
        deadline,
        false,
        [oraclePackage]
      )
    // Deposit tokens for a taker
    await router
      .connect(taker1)
      .deposit(marketId, taker1.getAddress(), 0, true, [oraclePackage], { value:
depositAmount })

    // notional amount is 1
    // we spend 1019336 fixed tokens
    // we receive 909090 float tokens
    const [firstTrade] = await router
```

```
      .connect(taker1)
      .callStatic.executeTrade(
        future1Id,
        riskDirection.PAYER,
        toBn('1', decimals),
        toBn('1000000'),
        toBn('0', decimals),
        deadline,
        false,
        [oraclePackage]
    )
    // notional amount is 1.000001
    // we spend 12910862 fixed tokens
    // we receive 909091 float tokens
    // compared with the previous trade we spend much more fixed tokens
    // however, due to the minimalization we receive only one extra float token for that
    const [secondTrade] = await router
      .connect(taker1)
      .callStatic.executeTrade(
        future1Id,
        riskDirection.PAYER,
        toBn('1.000001', decimals),
        toBn('1000000'),
        toBn('0', decimals),
        deadline,
        false,
        [oraclePackage]
      )
    // notional amount is 2
    // we spend 12910862 fixed tokens, same as at the previous trade
    // we receive 1818181 float tokens, which is 2 times bigger compared to the previous
trade
    // so, by exchanging the same amount of fixed tokens as in the previous trade we receive
    // twice the amount of float tokens
    const [thirdTrade] = await router
      .connect(taker1)
      .callStatic.executeTrade(
        future1Id,
        riskDirection.PAYER,
        toBn('2', decimals),
        toBn('1000000'),
        toBn('0', decimals),
        deadline,
        false,
        [oraclePackage]
```

```
        )
    })
```

We recommend reviewing this scenario, and adding the limitation on the max liquidity of the tick (Uniswap V3 as an example) in order to minimize the risks of this "feature" occurring.

Rh0's response

We don't use unchecked math like Uniswap, so the maximum limit already exists in the types from the PRB math library. The described feature leads to a position with a larger number of fixed tokens (rounded up), in a certain case. Since this case relates to the payer direction, the number of fixed tokens determines how much payer will pay, this feature leads to increased loss but no income. Thus it cannot be considered as a vulnerability.

2.2 MAJOR

M-01	Positions during `maturityLockout` continue to affect margin calculations in `MarketLogic`
Severity	MAJOR
Status	• FIXED

Location

File	Location	Line
MarketLogic.sol	contract `MarketLogic` > function `dv01AndRiskDirection`	168
MarketLogic.sol	contract `MarketLogic` > function `initialMarginThreshold`	220
MarketLogic.sol	contract `MarketLogic` > function `initialMarginThresholdWithPosition`	266
MarketLogic.sol	contract `MarketLogic` > function `initialMarginThresholdWithPositions`	315
MarketLogic.sol	contract `MarketLogic` > function `lpMarginThreshold`	371
MarketLogic.sol	contract `MarketLogic` > function `lpMarginThresholdWithProvision`	406
MarketLogic.sol	contract `MarketLogic` > function `liquidationMarginThreshold`	445

Description

In these locations within the functions:

- `dv01AndRiskDirection`
- `initialMarginThreshold`
- `initialMarginThresholdWithPosition`
- `initialMarginThresholdWithPositions`
- `lpMarginThreshold`
- `lpMarginThresholdWithProvision`
- `liquidationMarginThreshold`

While iterating through the `unsettledFutures` mapping, the following construction is used in the code block:

```
if (future.timeIsOutOfTerm(0)) continue;
```

This construction does not consider `maturityLockout`, as it only skips finalized futures. Consequently, positions in these futures will continue to impact the user's margin, even though these positions are no longer active.

During liquidation in the `transferPositionsOwnership` and `cancelProvisions` functions, while iterating through the `unsettledFutures` mapping, `maturityLockoutSeconds` is taken into account, opening opportunities for manipulation by a hacker.

Consider the following scenario:

1. A user opens two positions in different futures in different directions - Long and Short. Positions hedge each other.
2. The index rate changes, the Short position incurs losses, approaching liquidation in Future 1. The Long position is healthy and has a profit.
3. Future 1 enters maturity lockout; however, the Short position still affects the margin.
4. In Future 2, the index rate changes, or the trading rate changes (for example, a dishonest user notices this situation and makes a trade to influence the position). The user's margin falls below the liquidation threshold, and the user's position can be liquidated.
5. The user or liquidator starts the user's liquidation process using the `transferPositionsOwnership` function. In this function, due to the check that the position in Future 1 is already locked - it starts to liquidate the position in Future 2, which is profitable.
6. The hacker takes the user's collateral and reward without adding his funds to the position, taking the user's funds during liquidation.

Here is a Proof of Concept of the case described above.

Recommendation

We recommend revising the logic for calculating margin on positions, taking into account possible locking in a `maturityLockout`. The start of the `maturityLockout` should be handled identically both for the `margin` and liquidation calculations.

Update

Fixed in commits `2863b4b8711762c19437f8b90132a5418f77f8a3`, `94bf86548eaedb198bd1cb33ad1f831a59f6e59a`, `9d721ec74e851782a6e6c29572cedc6ba10b45fc`.

Rh0's response

Positions currently in maturity lockout do affect margin requirements, they are considered active all the way to the point of maturity. Positions in the maturity lockout have now also been made transferrable via novation. Calculations of the transferring amount were revised accordingly.

Oxorio's response

This finding has been downgraded from Critical to Major, and the problem has been fixed.

M-02	Market may be subject to manipulation by an attacker, given sufficient resources and favorable market conditions at the time
Severity	MAJOR
Status	• ACKNOWLEDGED

Description

In the `Issuer` contract in the `createFuture` function, the future is created with the creation of its own VAMM module, initializing the `currentInterestRate` rate from which the trading will occur, and the start of the future `termStartTimestamp`.

Right after the pool creation, trading is still unavailable, as well as liquidity providing to the pool. During the VAMM initialization, there is also no liquidity provision. This means that at the `termStartTimestamp` timestamp, there is no pool liquidity, and the pool is very vulnerable to manipulations.

Let's dive deep into the possible scenario of the attack when there is no liquidity in the pool, but keep in mind that the same attack can be performed when there is liquidity in the pool, but it will be more expensive:

1. A future with the `termStartTimestamp` in the future. The pool has no liquidity. VAMM has `200` intervals with `0.001` tick spacing. The lower bound of the pool is `0`, the upper bound is `0.2`. The `currentInterestRate` is `0.1`.
2. The time of `termStartTimestamp` comes, and a malicious user starts a manipulation in the pool right in the starting block, packing all the attack transactions in `1` transaction and frontrunning all other trades to the pool in this block.
3. For the manipulation, several addresses are used:
    - `maker1` - the address that will provide liquidity to the pool, allowing takers to open trades.
    - `taker1` - the address that will execute the manipulation itself in the pool, crossing the liquidity intervals and moving the rate to the desired level. Positions of this user will be hedged, and this address will not suffer losses from manipulation itself.
    - `taker2` - the address that will execute the main profitable short that can never be liquidated.
4. The `maker1` address provides liquidity to the pool in the following way:
   `provideLiquidity` with the bounds `0.125` - `0.126` and the notional amount `1`.
   `provideLiquidity` with the bounds `0.150` - `0.151` and the notional amount `1`.
   `provideLiquidity` with the bounds `0.175` - `0.176` and the notional amount `1`.
   `provideLiquidity` with the bounds `0.199` - `0.2` and the notional amount `1`.

By doing these "liquidity steps" the `maker1` allows driving the rate to a higher level on the curve right to the upper bound of the curve. Since the notional amount is only `1`, the fees to the protocol and the liquidity provider are minimal.

5. The `taker1` address executes 4 trades as a `PAYER`. This is done in 4 separate transactions to pass the `TradeRateImpactLimitExceeded` check. The test is executed with the `maxRateImpactPerTrade` of `50000000000000000`, the value is taken from the `constants` file. Depending on this variable, the number of steps to drive the liquidity to the upper bound will differ. By executing 4 trades, the rate is moved from `0.1` to `0.126`, then to `0.151`, then to `0.176`, and finally to `0.2`. Since all `taker1` positions were as a `PAYER` and the rate went up, the `taker1` address hasn't suffered any losses except the trading fees.

6. The `maker1` address provides liquidity to the pool with the bounds `0.199` - `0.2` and the notional amount `1000`.

7. The `taker2` address executes a trade as a `RECEIVER` with the notional amount `1001`. This trade is executed in 1 transaction, since we are executing the trade within one tick and the rate changes to `0.199`.

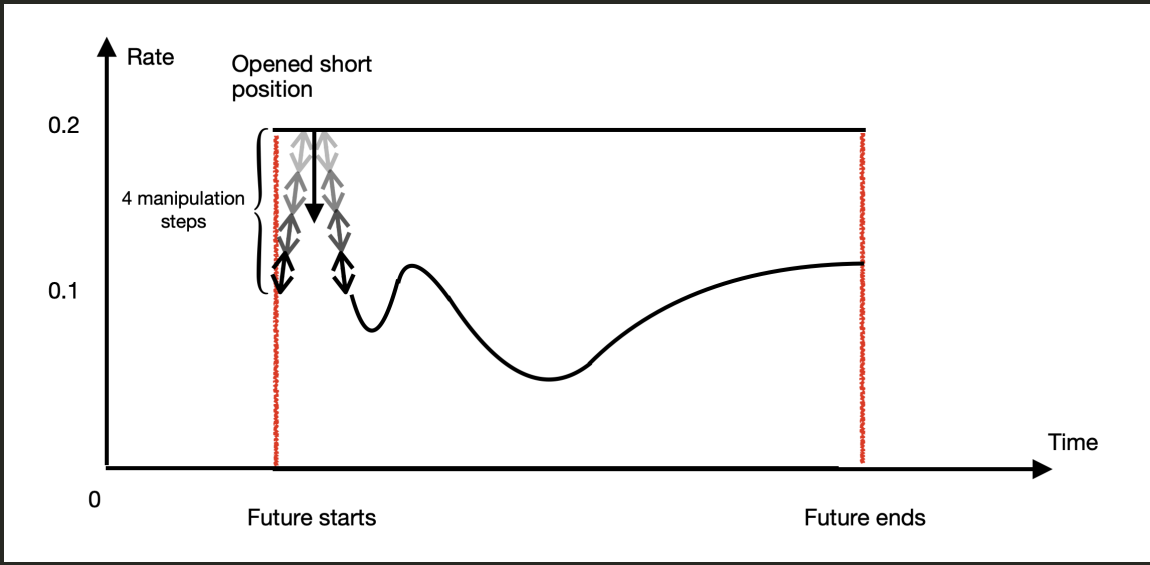
8. The `maker1` adds liquidity to the pool with the bounds `0.1` - `0.101` and the notional amount `1` as an additional liquidity step, since in the previous time we started from `0.1`, and there was no liquidity in that range.

9. The `taker1` returns the rate back by executing 4 trades as a `RECEIVER` with the notional amount `1`. The rate is moved to `0.1`, and the `taker1` address hasn't suffered any losses except the trading fees; all the previous longs are hedged. The `taker1` with the `maker1` also can drop the rate even lower if needed.

10. After that, the `maker1` address provides liquidity to the pool with the bounds `0.95` - `0.105` and the notional amount `2000`, or bigger. We don't want to give other users a chance to drive the rate back to the upper limit, and by providing a lot of liquidity, we are making it harder for other users to manipulate the rate. The amount of liquidity and the range can be higher and wider, if necessary, depending on the amount of funds that the `maker1` address has, pool volatility, and liquidity of other users.

11. On the provided rate close to `0.1` range, other users provide more liquidity, create trades. As for the `taker2` position, it is not interesting, since his position sits on the very upper corner of the rate curve; his position can never be liquidated. The more time passes, the less likely that the rate will go back to the upper bound, decreasing the profits of the attacker. This attack can also be performed in a more complex way when after the creation of the short position on the upper bound, the attacker creates a long position on the lower bound from the other address. By doing this, the attacker's profits will be maximized, and at the same time, the pool will remain balanced for the other users.

12. The trading stops, the future expires, and the `taker2` address settles his position, gaining profits from the attack.



The Proof of Concept to the attack is here.

Recommendation

We recommend reviewing existing logic, changing the price impact validation by storing the impact rate in the contract and validating the change across not only one transaction but across several blocks, adding liquidity before the start of the trading. Allow other users to add liquidity before the start of the future. Implement an arbitrage bot, which will be balancing the rates across different pools.

Update

Rh0's response

Looking at this example, it seems to me taker 2 and maker 1 are accounts that belong to the same owner. The result of the manipulation is that taker 2 receives rate at favorable price, making profits when the rates are normalized. At the same time, maker 1 pays fixed rate at unfavorable price for him, losing the same amount of money in the process. Although the taker 2 has a profitable position, the owner of both account doesn't benefit from the price manipulation.

It is worth noting that wash trades are a common problem across exchanges. During the ramp-up period of the protocol, this issue is mitigated by our whitelisting and KYC requirements. Before making the protocol public, we'll add incentives for more market participants to provide liquidity at the market's initiation.

It doesn't seem to be a bug, it's basically how markets work — having a lot of funds one can manipulate the market, be it TradFi or DeFi. Since our protocol is at an early stage, we will have to mitigate those risks manually. For future we'll think about a way to give some window where Market Makers could place their liquidity before opening the Market for trades. In the nearest future this would be made for Takers in form of an auction, that discovers the starting price/rate.

Rh0's response

Market manipulation in TradFi is basically prevented by inability of a single actor to amass enough dry powder to impact an efficient market in a meaningful way on their own. And even then, these things happen (see Soros vs Bank of England). In crypto, all Sybil-like attacks are difficult to deal with and are mainly limited by their economic cost/benefit rather than technical restrictions. While we plan to implement further price impact controls, including those covering multiple blocks and longer time periods, all of those can be inefficient in certain market scenarios OR will limit the market efficiency in “normal” scenarios.

Oxorio's response

The problem was acknowledged by the Rh0 team, and we have downgraded the finding from Critical to Major. According to our severity level reference:

MAJOR: A bug that could cause a contract failure, with recovery possible only through manual modification of the contract state or replacement.

In this case, we consider a consistently profitable position without the possibility of liquidation as a system failure, which can be rectified only through contract replacement. This perpetually profitable position not only diminishes the profits of other users but also creates unfair rates in the pool. We cannot downgrade the problem to a Warning due to the potential harm that may result from exploitation.

As the client has pointed out, this finding is a common issue across exchanges, and the risks of price manipulations with a low level of liquidity in the protocol at the beginning of the future must not be underestimated. While fixes related to price impact control over multiple blocks may limit market efficiency, implementing logic that initiates the future only after achieving sufficient liquidity across multiple owners will not harm market efficiency but rather improve it. Whitelisting users with KYC does not eliminate the problem but does reduce the risk of exploiting this vulnerability in the near future.

Additionally, regarding Rh0's observations, we would like to add that the profitability of price manipulation heavily depends on market conditions, liquidity amounts, rate limits, and fee configurations. A rate manipulation attack can be executed with a minimal notional amount, and, at the same time, the manipulator will hedge all positions as a payer with the same positions as a receiver at the same rate levels. This implies that the only expenses for the manipulation are blockchain gas fees and protocol fees.

M-03	In certain cases, when the current rate falls on the LP interval boundary, LP fee delta may become negative
Severity	MAJOR
Status	• FIXED

Location

File	Location	Line
RatePoint.sol	contract `RatePoint`	121

Description

In the function `_calcIntervalAccruedLPFeeCumulativeDelta` in the `RatePoint` contract, the returned amount of the function can be a negative value after the swaps that set the VAMM state when the rate is sitting on the boundary of the interval. This blocks all liquidity provisions until the state of the VAMM is changed to the correct tick.
For example:

1. The initial rate is `0.1`.
2. Maker adds liquidity to the `0.1-0.101`, `0.075-0.076`, `0.05-0.051`, `0.025-0.026`, `0.000-0.001` intervals, and the amount of notional is 1.
3. Taker drops the rate to `0.000` by making the short position for `1` notional as a `Receiver`.
4. During the last short position, one additional cross occurs to the additional previous interval before the `0.000-0.001` interval. This makes the `_calcIntervalAccruedLPFeeCumulativeDelta` function return a negative value since the `signedOverallCumulativeAccruedLPFee` is `50000000000000000`, and the accrued LP fee on the `0.001` tick is `40000000000000000`, which is correct. However, the accrued LP fee on the `0.000` tick of the lower boundary of the interval equals the `cumulativeAccruedLPFeeOutside` variable instead of `signedOverallCumulativeAccruedLPFee - cumulativeAccruedLPFeeOutside` and is `50000000000000000`.

This makes it impossible to add liquidity to any other intervals for all users until the state of the VAMM is changed to the correct tick.

The Proof of Concept for the finding is here.

Recommendation

We recommend adding more validations to the `swap` function prohibiting crossing to the incorrect tick.

Update

Fixed in commits `f3fdc8670b1a249c700fe87c7291a167854b5bdf`, `0083ecc1ba9c103f15cc0477fa4185316eb0e522`.

Rh0's response

We have added separate logic to work with the case when the current rate is equal to the lower bound of the interval.

Oxorio's response

The issue has been fixed in the `f3fdc8670b1a249c700fe87c7291a167854b5bdf`, `0083ecc1ba9c103f15cc0477fa4185316eb0e522` commits.

M-04	If the market rate is set incorrectly at market initiation, trading in such market will be impossible
Severity	MAJOR
Status	• NO ISSUE

Location

File	Location	Line
VAMM.sol	contract `VAMM` > `constructor`	113

Description

In the `constructor` of the `VAMM` contract, the `currentRate` variable is set during deployment; however, with the initialization of the `currentRate` variable, there is no liquidity provision to the pool. This means that when the `currentRate` is set to a value far away from the real interval rates, trading will be impossible for users. This is because liquidity will be provided on intervals far away from the `currentRate`, and when executing a swap, the rate will go to the nearest tick with liquidity, causing the swap to revert due to the high rate impact. To enable trading, someone should provide liquidity to the pool closer to the `currentRate` value, where trading doesn't usually happen. After this, several trades for the provided liquidity amount should be executed to move the rate to the desired level. It's unlikely that simple users will handle this case by themselves, and the future will be unable for trading since all users will receive the `TradeRateImpactLimitExceeded` error.

Recommendation

We recommend reviewing existing logic and adding liquidity to the pool during the `constructor` execution. With enough liquidity across the curve, users will be able to drive the rate to the appropriate levels by themselves.

Update

Rh0's response

At the current stage we set the starting rate for a new Future manually. We accept the risk since there is no other way at this point, but we are already designing a solution to this problem that is the auctions that have to take place to discover the price point at which the Future should be started.

Rh0's response

While the findings are valid (and known to us), we feel that they do not relate to the protocol code quality (but rather to our management of off-chain infrastructure, which is out of scope for this particular audit), and may not be relevant to the current implementation:
For M-03, at the current implementation, we set the initial rate at the initiation of the pool using independent market parameters (say, looking at the futures curve structure on Binance), and we deploy initial liquidity. If the set rate is wrong, the trading will not be possible, but since the market is new, there will be no client funds at risk unless we fix the error. So it feels like a Warning to us.

Oxorio's response

The problem was marked as "no issue" by the Rh0 team, but we haven't downgraded this finding. According to our severity level reference:

MAJOR: A bug that could cause a contract failure, with recovery possible only through manual modification of the contract state or replacement.

In this case, because of the incorrect configuration of the pools rate and rate impact limitations the trading will not be possible in the Future (as you also marked in the response). This is a contract failure, which can be recovered with the manual modification or contract state replacement. If clients funds were at risk, the problem would have Critical severity. Our severity level reference doesn't take into the consideration the likelihood of the exploitation, amounts of funds or quantities of clients at risk.

M-05	Trading is halted if the floating rate oracle packages do not contain a correct cryptographic signature
Severity	MAJOR
Status	• NO ISSUE

Location

File	Location	Line
BaseFloatIndexOracle.sol	contract `BaseFloatIndexOracle` > function `_checkExpiration`	216

Description

In the function `_checkExpiration` of the `BaseFloatIndexOracle` contract, the oracle signature is checked, and if the signature is outdated, the function reverts. If the oracle address stops signing the data due to server malfunction, power outage on the machine with the private key, or any other reason, the oracle will stop working, and the protocol will cease to function. There are no emergency functions to give users an opportunity to continue trading, and the protocol will be stuck in the current state. With the volatility of the crypto market, oracle downtime will lead to the inability of users to add collateral in time, as the `deposit` function also requires a fresh oracle signature. After the oracle downtime, all users will be immediately liquidated.

Recommendation

We recommend reviewing existing logic, adding an emergency state to the protocol that will allow users to continue trading in case of oracle downtime for a period until the new signature is produced, or adding a separate `deposit` function, which will work only in case of an outdated float index, allowing users to add collateral to their positions and avoid liquidation.

Update

Rh0's response

Trading is not possible without fresh oracle data.
When the oracle becomes available again, the trading resumes with fresh oracle data. So nothing manually have to be done on the protocol side.
A significant oracle downtime is a special case. Then the trading should be stopped and continued after an auction is held to discover new market price. The auctions feature is to be introduced later on, until then such cases must be handled manually. But their likelihood is low.

Rh0's response

The trading would halt without fresh oracle-signed data, which is actually an intended behavior. The trading would restart as normal without any manual interventions once the oracle data becomes available, so it may not fall within definition of Major. There are certain risks associated with a sharp market rate shifts at the trading restart in case of a prologued downtime (that we are working on), which probably justifies having this as a Warning.

Oxorio's response

The problem was marked as "no issue" by the Rh0 team, but we haven't downgraded this finding. According to our severity level reference:

MAJOR: A bug that could cause a contract failure, with recovery possible only through manual modification of the contract state or replacement.

In this case, when a significant oracle downtime happens - the protocol halts and all the state becomes outdated, which we can determine as a protocol failure. The recovery from this situation is only possible with the manual modifications of the contracts state, or, for example, it can be an auction process. However, in this repository the auction logic is missing.

Regarding your first response, with the small downtime without the market volatility the protocol will be able to work correctly with small consequences, however oracle downtime of any duration with the market volatility will lead to massive liquidations without possibility of adding collateral. The likelihood of the finding is not used in our severity level reference.

Regarding your second response, the finding is related mainly to the audited on-chain scope, since this scope is missing logic for this emergency case, as well as to the `deposit` function. This problems relates mainly to the downtime with any duration, but during the market volatility, resulting in permanent rate shifts, possible bad debt of the protocol because of the not working liquidations.

Rh0's response

Regarding M-05, there is a small note that in our functionality, there is a granular configuration for stopping available operations in the protocol. Thus, in the event of prolonged issues with the oracle, we can run the market with the liquidation operation stopped to give users time to replenish their margin.

2.3 WARNING

W-01	Validate decimals in `CollateralManager`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
CollateralManager.sol	contract `CollateralManager` > `constructor`	70

Description

In the `constructor` of the `CollateralManager`, there is no validation of the `_underlyingToken` `decimals` value. The math in the protocol doesn't work correctly when the `decimals` value is smaller than `6`. For example, a token with 2 decimals, representing a Euro or USD token, which has only 2 decimals for cents, won't work correctly and is not supported; the math will be incorrect. Test cases:

- Successfully handle a simple positive case according to the lightpaper with compounding math.
- Successfully handle a simple positive case according to the lightpaper with linear math.

These cases will revert, as well as other tests will revert with the `NotEnoughMargin` error.

Recommendation

We recommend validating the `decimals` variable.

Update

Fixed in commit `1f28e0cb8d917a975f66e03011a4ff4aad7b8d0d`.

Rh0's response

Checks were added.

Rh0's response

This is a good one, but we don't work with fiat. In a crypto world, I don't know of assets with less than 6 decimals. i.e. happy to take it and fix it, but it's not a major problem for the existing protocol's shape.

Oxorio's response

This finding has been downgraded from Major to Warning, and the problem has been fixed.

W-02	No functions to delete maker provisions in `FutureStorage`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
FutureStorage.sol	contract `FutureStorage`	36
FutureStorage.sol	contract `FutureStorage`	102
FutureStorage.sol	contract `FutureStorage`	110

Description

In the `FutureStorage` contract, `deleteMakerProvision` and `cleanMakerProvisions` functions for deleting maker provisions have an `onlyViaRelatedFuture` modifier. However, there is no such call from the futures contract, which makes it impossible to delete a maker provision from the `FutureStorage` contract.

Recommendation

We recommend adding functions to the `Future` contract that call the `FutureStorage` contract to remove `makerProvisions`.

Update

Fixed in commit `71bee2649fa29f54248c354aa41eed61e83f1f99`.

Rh0's response

We have implemented a mechanism for archiving LP positions when liquidity is deleted. Positions are cleared in AMM and aggregated in a separate storage.

Oxorio's response

The issue has been fixed in the `71bee2649fa29f54248c354aa41eed61e83f1f99` commit.

W-03	Missing `amount` validation in `CollateralManager`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
CollateralManager.sol	contract `CollateralManager` > function `collectProtocolFee`	167

Description

In the function `collectProtocolFee` of the `CollateralManager` contract, there is a validation that the `amount` is not greater than the collected `protocolFee`; however, there is no validation that the `amount` is not equal to `0`.

Recommendation

We recommend adding validation for zero values.

Update

Fixed in commit `e3f9117705aff7177e59c1c329953781da3a7115`.

Rh0's response

We changed the behaviour of this function so amount=0 means to collect all protocol fee available.
Added validation so transaction reverts if protocolFee to be collected is 0.

Oxorio's response

The issue has been fixed in the `e3f9117705aff7177e59c1c329953781da3a7115` commit.

W-04	Withdraw function can be called by a user without `unsettledFutures` in `RouterLogic`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
RouterLogic.sol	contract `RouterLogic`	307

Description

In the `RouterLogic` contract, a user can call the `withdraw` function without having any unsettled futures by providing the `amount` variable as zero, which can lead to a DDOS of `MarginUpdate` events.

Recommendation

We recommend adding a check for the `amount` variable to prevent this issue.

Update

Fixed in commit `dc809cf689fe28281dda80ea0e7369d180911aa5`.

Rh0's response

When `withdraw` is called with `amount=0` it means to withdraw all withdrawable margin of a user.
Added check to revert if user withdrawable amount is 0.

Oxorio's response

The issue has been fixed in the `dc809cf689fe28281dda80ea0e7369d180911aa5` commit.

W-05	Negative bounds for rate in `VAMM`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
VAMM.sol	contract `VAMM` > `constructor`	117
CompoundingRateMath.sol	contract `VAMM` > `constructor`	41
LinearRateMath.sol	contract `VAMM` > `constructor`	33

Description

In the `constructor` of the `VAMM` contract, it is possible to set bounds less than `-100%`. This leads to underflow in the functions function `rateToPrice`, function `rateToPrice`.

Recommendation

We recommend adding a validation for the lower bound rate to prevent this issue.

Update

Fixed in commit `235977b90639bc153903adc9d82d80d8b43874cd`.

Rh0's response

We now validate that the lowest bound rate in VAMM is `>= -100%`.

Oxorio's response

This commit fixes the initial problem in the `235977b90639bc153903adc9d82d80d8b43874cd`.

W-06	Deposits are allowed when there is no ongoing futures in `Router`
Severity	WARNING
Status	• NO ISSUE

Location

File	Location	Line
Router.sol	contract `Router`	218

Description

In the `Router` contract, deposits to the protocol are allowed when there are no ongoing futures or delayed futures. In the `deposit` function, there is a `whenNotPaused` modifier that stops all protocol operations, including pausing the `withdraw` operation. If the protocol decides to stop creating futures and close, users will still be able to deposit funds into the protocol.

Recommendation

We recommend adding a separate pause function for the `deposit` function.

Update

Rh0's response

This doesn't seem to be an issue or a vulnerability. Users could withdraw all their funds back if they deposit the funds by mistake.

W-07	`persistIndexAtMaturity` can be called before maturity in `Router`
Severity	WARNING
Status	• FIXED

Location

File	Location	Line
Router.sol	contract `Router` > function `persistIndexAtMaturity`	336

Description

In the `persistIndexAtMaturity` function of the `Router` contract, there is no check to ensure that the `futureId` has already reached maturity.

Recommendation

We recommend validating if the `futureId` has already reached its maturity in the `persistIndexAtMaturity` function.

Update

Fixed in commit `cdfb81a7ad41e5b3e5e0622861c15a5908d39e43`.

Rh0's response

Added check so tx reverts if future has not matured yet.

Oxorio's response

The issue has been fixed in the `cdfb81a7ad41e5b3e5e0622861c15a5908d39e43` commit.

W-08	`tx.origin` is not checked in `Router`
Severity	WARNING
Status	• NO ISSUE

Location

File	Location	Line
Router.sol	contract `Router` > modifier `onlyProtocolUser`	60

Description

In the `onlyProtocolUser` modifier of the `Router` contract, the `tx.origin` of the transaction is not checked together with the `msg.sender`. This means that it's possible for any user to call the function from the whitelisted contract, passing the `msg.sender` check. It also allows calling all the functions from the whitelisted contract, even if this contract is a malicious one.

Recommendation

We recommend adding a check to the `_checkProtocolUser` function of the `Router` contract for the `tx.origin` together with the `msg.sender`.

Update

Rh0's response

We're not limiting the protocol calls to be done only by users, since there are some integrations to be expected soon.

W-09	`tradeRateImpactLimit` is used for one trade in `VAMM`
Severity	WARNING
Status	• NO ISSUE

Location

File	Location	Line
VAMM.sol	contract `VAMM` > function `swap`	463

Description

In the function `swap` of contract `VAMM`, the `tradeRateImpactLimit` is used only during one trade execution, allowing manipulation of the pool rate by executing several trades in one block. The `tradeRateImpactLimit` should be applied for a specific period, for example, several blocks, and not just for a single trade.

Recommendation

We recommend refactoring the `tradeRateImpactLimit` logic by validating the difference of rate impacts not only within one transaction but for a fixed period of time, for example, several blocks.

Update

Rh0's response

Trade rate impact limit does not cover all cases, in the future we plan to consider a circuit breaker to automatically stop trading and then start it after an auction.

W-10	Lack of validations on the number of intervals in `VAMM`
Severity	WARNING
Status	• NO ISSUE

Location

File	Location	Line
VAMM.sol	contract `VAMM` > `constructor`	118

Description

In the `constructor` of the `VAMM` contract, the values of `_intervalsCount` and `_intervalLength` are not validated. Simultaneously, when calling the `swap` function in the `VAMM` contract, it iterates over an unrestricted number of intervals.

For example, a situation is possible when a low level of liquidity was provided over a large number of intervals, and the length of the intervals themselves is very small. In such a scenario, the taker, when making a swap, will traverse a large number of intervals, spending gas on processing each of them but will not change the rate too much to trigger the `TradeRateImpactLimitExceeded` error. Such a swap will eventually result in an out-of-gas error.

Recommendation

We recommend implementing validation for the number and size of intervals to prevent potential gas-related issues.

Update

Rh0's response

There is already a restriction on the minimum number of intervals - 1.
The maximum must be selected by the issuer taking into account the proper value of trade rate impact limitations. Issuer at this point is a representative of the Team, who chooses reasonable parameters.
Iterations through all intervals are not required for current logic.

2.4 INFO

I-01	Freezing futures parameters during `maturityLockout`
Severity	INFO
Status	• NO ISSUE

Location

File	Location	Line
ViewDataProvider.sol	contract `ViewDataProvider` > function `makerLiquidityDistribution`	346
ViewDataProvider.sol	contract `ViewDataProvider` > function `poolLiquidityDistribution`	358

Description

When manipulating the rate in the last block before `maturityLockout`, the rate is frozen for the entire duration of `maturityLockout`. This effect provides an opportunity for rate manipulation since, in the next block, no one can return the rate back to its market value.

This impacts users who monitor the futures rate changes through view functions, such as `makerLiquidityDistribution` or `poolLiquidityDistribution`.

Recommendation

We recommend revising the mechanism for providing rate-related information via view functions to users during the `maturityLockout` period.

Update

Rh0's response

This manipulation doesn't make sense in terms of futures market. When the Future market is getting closer to maturity point, its rate should essentially conform to that of the underlying floating rate. So if whoever "manipulates" the rate just before the maturity lockout, he only does worse for himself.

I-02	Prb-math library not audited
Severity	INFO
Status	• FIXED

Description

The library `prb-math` documents have not been audited by a security researcher. With the use of this library, the protocol increases the risk of unexpected behavior during calculations.

Recommendation

We recommend updating the library to the latest version from `4.0.0` to `4.0.2`, monitoring security problems with the library, adding fuzzing tests to the smart contracts to ensure correct behavior in various cases.

Update

Fixed in commit `3e84a1af2ba7527a7188cd155f9f6f68302ce6d6`.

Rh0's response

Property-based (fuzzy) tests can be found in `test/Properties.ts`.
We have updated the library version to v4.0.2.

Oxorio's response

The issue has been fixed in the `3e84a1af2ba7527a7188cd155f9f6f68302ce6d6` commit.

I-03	Missing events on initialization of contracts
Severity	INFO
Status	• FIXED

Location

File	Location	Line
ContractProvider.sol	contract `ContractProvider` > `constructor`	76
MarketStorage.sol	contract `MarketStorage` > `constructor`	55
VAMMStorage.sol	contract `VAMMStorage` > `constructor`	52

Description

In the `constructor` of contracts:

- `ContractProvider#L76`
- `MarketStorage#L55`
- `VAMMStorage#L52`

the emission of events is missing. The absence of emissions in the constructor leads to an incorrect history of events.

Recommendation

We recommend emitting events in the constructor.

Update

Fixed in commit `40384fff338411801781d1424f485b1089134d81`.

Rh0's response

We have added events with contract initialization parameters.

Oxorio's response

The issue has been fixed in the `40384fff338411801781d1424f485b1089134d81` commit.

I-04	Add all interfaces to interface folders
Severity	INFO
Status	• NO ISSUE

Description

In the `contracts` folder, interfaces are mixed with the contracts and libraries in other folders.

Recommendation

We recommend removing interfaces from folders with contracts and libraries and moving them to the `interfaces` folder to keep the repository clean.

Update

Rh0's response

In our opinion, the current approach makes more sense

I-05	Missing validations in `CollateralManager`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
CollateralManager.sol	contract `CollateralManager` > function `collectProtocolFee`	176

Description

In the function `collectProtocolFee` of contract `CollateralManager`, it's possible to have a withdrawal of `_protocolFee` without transferring the tokens themselves due to a lack of check (`to != address(this)`). In such a case, `emergencyERC20TokenTransfer` would need to be called to manually transfer tokens from the contract.

Recommendation

We recommend adding a check to ensure that the recipient address is not equal to the address of the `CollateralManager` contract.

Update

Fixed in commit `6b9bec89a5744ddce38470b0730bcf8c09a594d1`.

Rh0's response

We have added validation.

Oxorio's response

The issue has been fixed in the `6b9bec89a5744ddce38470b0730bcf8c09a594d1` commit.

I-06	Redundant initialization in `ContractProvider`
Severity	INFO
Status	• FIXED

Description

In the `constructor` of contract `ContractProvider`, the default value of `_isEmergencyModeEnabled` is `false`. There is no need to initialize it in the constructor.

Recommendation

We recommend removing the variable from the constructor to keep the codebase clean.

Update

Fixed in commit `40384fff338411801781d1424f485b1089134d81`.

Rh0's response

Initialization has been removed.

Oxorio's response

The issue has been fixed in the `40384fff338411801781d1424f485b1089134d81` commit.

I-07	Similar events in `CollateralManager`, `Router`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
Router.sol	contract `Router` > modifier `collectProtocolFee`	455
CollateralManager.sol	contract `CollateralManager` > function `collectProtocolFee`	174

Description

In function `collectProtocolFee`, function `collectProtocolFee`, similar events with identical names are called consecutively in the functions `Router.collectProtocolFee` and `CollateralManager.collectProtocolFee`.

Recommendation

We recommend removing the emission of one of the events or changing its name to avoid creating confusion.

Update

Fixed in commit `3cd598e5595ac8b30dd88deed123034cbd8d2d8a`.

Rh0's response

One of the events has been renamed.

Oxorio's response

The issue has been fixed in the `3cd598e5595ac8b30dd88deed123034cbd8d2d8a` commit.

I-08	Mixing of type names `uint` and `uint256`
Severity	INFO
Status	• FIXED

Location

File	Location	Line
CollateralManager.sol	contract `CollateralManager` > function `emergencyERC20TokenTransfer`	187
FutureStorage.sol	contract `FutureStorage` > function `cleanMakerProvisions`	113

Description

In function `emergencyERC20TokenTransfer` and function `cleanMakerProvisions`, there is a mixing of type names `uint` and `uint256`.

Recommendation

We recommend using uniform variable type names to avoid creating confusion in the codebase.

Update

Fixed in commit `bb9b46d85afc389f4e0b1560f66a6b5bb4d1e843`.

Rh0's response

The uint typing has been unified.

Oxorio's response

The issue has been fixed in the `bb9b46d85afc389f4e0b1560f66a6b5bb4d1e843` commit.

3 CONCLUSION

The following table contains the total number of issues that were found during audit:

Severity	FIXED	ACKNOWLEDGED	NO ISSUE	Total
CRITICAL	5	0	0	5
MAJOR	2	1	2	5
WARNING	6	0	4	10
INFO	6	0	2	8
Total	19	1	8	28

THANK YOU FOR CHOOSING