SMART CONTRACTS REVIEW

zokyo

April 16th 2024 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that these smart contracts passed a security audit.

ZOKYO AUDIT SCORING ROLLIE

1. Severity of Issues:
    - Critical: Direct, immediate risks to funds or the integrity of the contract. Typically, these would have a very high weight.
    - High: Important issues that can compromise the contract in certain scenarios.
    - Medium: Issues that might not pose immediate threats but represent significant deviations from best practices.
    - Low: Smaller issues that might not pose security risks but are still noteworthy.
    - Informational: Generally, observations or suggestions that don't point to vulnerabilities but can be improvements or best practices.
2. Test Coverage: The percentage of the codebase that's covered by tests. High test coverage often suggests thorough testing practices and can increase the score.
3. Code Quality: This is more subjective, but contracts that follow best practices, are well-commented, and show good organization might receive higher scores.
4. Documentation: Comprehensive and clear documentation might improve the score, as it shows thoroughness.
5. Consistency: Consistency in coding patterns, naming, etc., can also factor into the score.
6. Response to Identified Issues: Some audits might consider how quickly and effectively the team responds to identified issues.

HYPOTHETICAL SCORING CALCULATION:

Let's assume each issue has a weight:
- Critical: -30 points
- High: -20 points
- Medium: -10 points
- Low: -5 points
- Informational: -1 point

Starting with a perfect score of 100:
- 0 Critical issues: 0 points deducted
- 0 High issues: 0 points deducted
- 4 Medium issues: 2 resolved and 2 acknowledged = - 8 points deducted
- 9 Low issues: 8 resolved and 1 acknowledged = - 2 points deducted
- 2 Informational issue: 2 resolved = 0 points deducted

Thus, 100 - 8 - 2 = 90

TECHNICAL SUMMARY

This document outlines the overall security of the Rollie smart contract/s evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the Rollie smart contract/s codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the review. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contract/s but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the Rollie team put in place a bug bounty program to encourage further active analysis of the smart contract/s.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the Rollie repository:
Repo: https://github.com/Rollie-Finance/rollie-contract/tree/core-trading

Last commit: fdfa146612f02a06ff02df4d9e606e4df194d67e

Within the scope of this audit, the team of auditors reviewed the following contract(s):

- ./TradingStorage.sol
- ./TradingVault.sol
- ./Trading.sol
- ./PythOracle.sol
- ./libs/VarConstant.sol
- ./libs/TradingHelper.sol
- ./libs/Types.sol
- ./PairInfos.sol
- ./FeeHelper.sol
- ./Referrals.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of Rollie smart contract/s. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01 Due diligence in assessing the overall code quality of the codebase.

02 Cross-comparison with other, similar smart contract/s by industry leaders.

03 Thorough manual review of the codebase line by line.

Executive Summary

The Zokyo team has performed a security audit of the provided codebase. The contracts submitted for auditing are well-crafted and organized. Detailed findings from the audit process are outlined in the "Complete Analysis" section.

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the Rollie team and the Rollie team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Vulnerability in updatePair Function Allows Creation of Duplicate pairName Entries	Medium	Resolved
2	userLastDepositTime can be enforced without user's consent	Medium	Acknowledged
3	Redundant Calculation of positionChange in updateMargin Function	Medium	Acknowledged
4	Potential Denial of Service (DoS) via Unchecked External Calls in FeeHelper Contract	Medium	Resolved
5	Single Point of Failure in Keeper Role	Low	Acknowledged
6	Redundant Expression in doneCloseRequst Function	Low	Resolved
7	Local Variable Shadowing in TradingVault & Referrals Contract	Low	Resolved
8	Precision Loss Due to Division Before Multiplication in closingFee Calculation	Low	Resolved
9	Inconsistent Attribute Validation in Modifiers	Low	Resolved
10	Potential inconsistent accounting due to rescueFunds Function	Low	Resolved
11	Collateral limit may prevent the position owner from updating their position and avoiding liquidation	Low	Resolved
12	Method distributeReward can increment currentBalance without sending tokens	Low	Resolved
13	No address(0) check	Low	Resolved
14	Confusing Function Naming in _checkTradeType Function	Informational	Resolved
15	Safe maths utilized in for-loop	Informational	Resolved

Vulnerability in updatePair Function Allows Creation of Duplicate pairName Entries

Location: PairInfos.sol

In PairInfos.sol, the `updatePair` function is susceptible to a critical vulnerability that allows for the creation of duplicate `pairName` entries for pairs. This vulnerability arises from the scenario where the function is called with a Pair object containing a `pairName` that matches an existing pair's `pairName`. The vulnerability can lead to inconsistencies in the contract's state that may be challenging to mitigate.

Scenario:
Consider the following scenario:

- A PairInfos Contract named target is deployed, and pairs are added to it using the `addPairs` function. Among these pairs, there are two pairs with `pairName` values of "ETHUSDT" and "WBTCUSDT", respectively.
- The `updatePair` function is called with an index of 1 and a Pair object that has its `pairName` set to "ETHUSDT".
- Due to a coding error, the `updatePair` function fails to properly check for duplicate pairName entries.
- As a result, the contract allows the creation of a duplicate `pairName` entry for the pair with the `pairName` "ETHUSDT".
- Subsequent operations on the pairs may encounter inconsistencies, as the contract now contains two pairs with the same `pairName` value.

Impact:
This vulnerability can have severe implications for the integrity and reliability of the smart contract:

- Inconsistencies in State: The presence of duplicate `pairName` entries can lead to inconsistencies in the contract's state, making it difficult to determine the correct data associated with each pair.
- Data Integrity Compromised: The incorrect data associated with pairs can compromise the integrity of the contract's data, potentially resulting in erroneous calculations or operations.
- Mitigation Challenges: Mitigating the effects of this vulnerability may be challenging, as identifying and resolving inconsistencies in the contract's state can be complex and resource-intensive

Recommendation:

Duplicate Entry Checking: Enhance the `updatePair` function to include thorough checks for duplicate `pairName` entries before allowing updates to pairs. This ensures that each pair has a unique `pairName` value.

Fix - Issue addressed in commit fdfa146 , by adding this at line 218 `require(!isPairListed[_pair.pairName], "PAIR_ALREADY_LISTED");`

The error message though confuses in case of debugging as it confuses with the error in the line before it “PAIR_NOT_LISTED”. It is recommended to change the error message to “NAME_ALREADY_USED”.

userLastDepositTime can be enforced without user's consent

Location: TradingVault.sol

In function `deposit(uint256 _amount, address _user)`, caller can deposit `_amount` of standardToken for `_user` and update `userLastDepositTime` for that user. The mapping `userLastDepositTime` limits the user from withdrawing the deposits.

```
userLastDepositTime[user] = block.timestamp;
```

Caller `msg.sender` can deposit dust amounts (i.e. 1 wei) in order to block user from withdrawing which can be considered a denial of service being imposed on that user from doing withdrawal operations.

Recommendation:

The following strategies can be implemented to address the issue:

- Implementing a minimum deposit requirement can serve as a deterrent to potential attackers.
- It is important to carefully select and vet allowed callers, particularly those listed in the allowed mapping (i.e. no code change required).

Fix - Finding is addressed in commit fdfa146 , the instruction: `userLastDepositTime[user] = block.timestamp;` is only executed when caller is not among the list of privileged allowed depositors. Therefore the timestamp update can not be imposed on the user. However the list of allowed depositors still require careful vetting to avoid a scenario in which allowed depositor conspires with a user to give advantage to the user to deposit without any time constraint to stay locked in the protocol.

An important issue arises when the user have no deposit and the allowed depositor is depositing for them for the first time. In this case we have `userLastDepositTime[_user] = 0;` which gives a misleading piece of information about the user's deposit. If rewards are distributed based on the locking time here from another protocol this can be critical.

In response to the identified issue, the client provided clarification regarding the purpose of the specified timeframe `userLastDepositTime`. They affirmed that this timeframe is not directly associated with rewarding users but rather serves as a mechanism to prevent malicious staking and withdrawal activities. Additionally, the client committed to conducting a comprehensive review of each permission granted for staking, emphasizing the importance of understanding and validating these permissions thoroughly.

MEDIUM-3 | ACKNOWLEDGED

Redundant Calculation of positionChange in updateMargin Function

Location: Trading.sol

Description:
The updateMargin function within Trading contract intended for trading operations contains a logical inconsistency that results in the positionChange variable always evaluating to zero. This issue arises from the method of recalculating leverage and subsequently calculating positionChange. The specific lines of code in question are:

```
uint256 position = ot.base.margin * ot.base.leverage;
...
uint256 leverage = position / newMargin;
...
uint positionChange = position / LEV_DENOMINATOR - (newMargin * leverage) / LEV_DENOMINATOR;
```

Given that leverage is recalculated as position / newMargin, and position is initially calculated as ot.base.margin * ot.base.leverage, the recalculation of position using the new leverage and new Margin essentially reverses the initial calculation, leading to positionChange being zero. This redundancy does not align with the intended functionality of capturing the change in position due to margin updates.

Recommendation:

To address this issue, it is recommended to revise the calculation of positionChange to accurately reflect the change in position due to the margin update.

Potential Denial of Service (DoS) via Unchecked External Calls in FeeHelper Contract

Location: FeeHelper.sol
Description:
The dealReferralAward function in the FeeHelper contract makes external calls to the referral contract without validating the return values (rebase and discount). This oversight can lead to a situation where the sum of rebase and discount exceeds totalFee, resulting in a negative balance calculation. This scenario can potentially lock funds, as the contract attempts to transfer more tokens than available, leading to a Denial of Service (DoS) condition for subsequent operations.

Recommendation:
Implement a validation step in the dealReferralAward function to check if the sum of rebase and discount exceeds totalFee. If the condition is true, execute an alternative logic path that avoids a DoS scenario. For example:

```
function dealReferralAward(uint256 totalFee, address trader, uint256 position) private returns (uint256 left) {
    if (address(referral) == address(0)) return totalFee;
    (uint256 rebase, uint256 discount) = referral.distributeRefReward(trader, totalFee, position);

    // Validate the sum of rebase and discount does not exceed totalFee
    if (rebase + discount > totalFee) {
        // Implement alternative logic here
        // For example, log the incident and continue without transferring funds
        emit ReferralRewardExceedsTotalFee(trader, rebase, discount, totalFee);
        return totalFee;
    }

    if (discount != 0) {
        SafeERC20.safeTransfer(standardToken, trader, discount);
    }
    if (rebase != 0) {
        SafeERC20.safeTransfer(standardToken, address(referral), rebase);
    }
    left = totalFee - discount - rebase;
}
```

Single Point of Failure in Keeper Role

Location: PairInfos.sol

Description:
The setPairParams function in the PairInfos contract is protected by the onlyKeeper modifier, allowing only the designated keeper to update critical trading pair parameters. This function's role is crucial for the ongoing maintenance and adjustment of trading parameters to respond to market conditions or protocol needs. The reliance on a single keeper address for such a critical function introduces a single point of failure. If the keeper's private key is lost, there would be no way to update pair parameters, rendering parts of the protocol inoperable. If the key is compromised, an attacker could manipulate trading parameters to their advantage.

Impact:
The single point of failure associated with the keeper role can lead to several adverse outcomes:

- Operational Risk: Loss of the keeper's private key would prevent any further updates to pair parameters, potentially freezing aspects of the protocol's operations.
- Security Risk: Compromise of the keeper's private key could allow unauthorized manipulation of pair parameters, leading to financial loss for users and the protocol.

Redundancy Risk: Inability to update critical parameters could render the protocol obsolete over time, as it fails to adapt to evolving market conditions or protocol needs.

Recommendation:

- Implement Multi-Signature Mechanism: Transition from a single keeper to a multi-signature mechanism for critical functions like setPairParams. This would require multiple trusted parties to agree on updates, significantly reducing the risk of key loss or compromise.
- Decentralized Governance: Consider implementing a decentralized governance model, allowing token holders to vote on updates to pair parameters. This could enhance security and reduce reliance on a single keeper.

Redundant Expression in doneCloseRequst Function

Location: TradingStorage.sol

Description:
The function doneCloseRequst contains a redundant expression exec; which has no effect on the execution of the function. This expression does not perform any operation or change any state within the function, making it unnecessary and potentially confusing for readers of the code.

Impact:
While this issue does not directly affect the contract's security or functionality, it introduces unnecessary clutter and can lead to confusion. It may mislead developers or auditors into thinking that exec has some side effect or is checked later in the function, which is not the case.

Recommendation:
Remove the redundant exec; expression from the doneCloseRequst function. This will have no impact on the function's logic but will improve code clarity and maintainability.

Local Variable Shadowing in TradingVault & Referrals Contract

Location: Referrals.sol

Description:
In both the TradingVault and Referrals contracts, there is an issue with function parameter shadowing that could lead to confusion and potentially incorrect assumptions about the contract's state. Specifically, the parameter _status in the setAllowedToInteract and setAllowed functions shadows the _status state variable inherited from the ReentrancyGuard and ReentrancyGuardUpgradeable contracts, respectively. This shadowing can cause confusion regarding which _status variable is being referenced and manipulated within the function's scope.

Affected Functions:

- TradingVault.setAllowedToInteract(address,bool)._status
- TradingVault.setAllowed(address,bool)._status
- Referrals.setAllowedToInteract(address,bool)._status

Recommendation:
To avoid confusion and ensure clarity in the contract's logic, it is recommended to rename the _status parameter in the setAllowedToInteract and setAllowed functions across both the TradingVault and Referrals contracts.

Precision Loss Due to Division Before Multiplication in closingFee Calculation

Location: Trading.sol

Description:
The calculation of closingFee within the _closeTrade function is susceptible to precision loss due to the order of operations where division occurs before multiplication. This issue stems from the calculation of closePosition where _closeMargin is divided by LEV_DENOMINATOR before being multiplied by _ot.base.leverage. This sequence of operations can lead to significant precision loss, especially when dealing with large numbers or small margins, affecting the accuracy of fee calculations and potentially leading to incorrect fee deductions and value transfers.

Recommendation:
To mitigate the precision loss, the calculation of closingFee should be adjusted to ensure multiplication occurs before division.

Inconsistent Attribute Validation in Modifiers

Location: PairInfos.sol

The smart contract includes four modifiers (`groupListed`, `feeListed`, `groupOk` and `feeOk`) that are responsible for validating attributes related to groups and fees. However, there is inconsistency in how these modifiers handle zero values for certain attributes.

```
 modifier groupListed(uint256 _groupIndex) {
        require(groups[_groupIndex].maxCollateralP > 0,
"GROUP_NOT_LISTED");
        _;
    }

    modifier feeListed(uint256 _feeIndex) {
        require(fees[_feeIndex].openFeeP > 0, "FEE_NOT_LISTED");
        _;
    }

    modifier groupOk(Group calldata _group) {
        require(_group.maxCollateralP < MILLAGE_DENOMINATOR,
"maxCollateralP wrong");
        _;
    }

    modifier feeOk(Fee calldata _fee) {
        require(_fee.openFeeP < MILLAGE_DENOMINATOR && _fee.closeFeeP <
MILLAGE_DENOMINATOR, "WRONG_FEES");
        _;
    }
```

The `groupListed` and `feeListed` modifiers are designed to validate the existence of a group and fee, respectively, by verifying that certain attributes are non-zero. For example, `groupListed` checks if the `maxCollateralP` attribute of the specified group is greater than zero, while `feeListed` checks if the `openFeeP` attribute of the specified fee is greater than zero.

On the other hand, the corresponding modifiers `groupOk` and `feeOk` are intended to ensure that the attributes of a group or fee are within acceptable ranges. However, these modifiers allow zero values for certain attributes. For instance, `groupOk` permits setting a zero value for the `maxCollateralP` attribute of a group, while `feeOk` allows zero values for both the `openFeeP` and `closeFeeP` attributes of a fee.

Impact:
This inconsistency in attribute validation can lead to unexpected behavior and potential vulnerabilities in the smart contract. Allowing zero values for certain attributes without proper validation may result in incorrect or unintended functionality, potentially exposing the contract to manipulation or exploitation by malicious actors.

Recommendation:
To address this inconsistency and enhance the security of the smart contract, it is recommended to ensure uniform validation of attributes across all modifiers. Specifically:

Consistent Attribute Validation: Review and revise the groupOk and feeOk modifiers to enforce non-zero values for attributes that are crucial for the integrity and functionality of the contract. This ensures consistency with the validation approach used in the groupListed and feeListed modifiers.

Potential inconsistent accounting due to rescueFunds Function

Location: FeeHelper.sol

The smart contract contains a function named `rescueFunds` that allows the owner to withdraw an arbitrary amount of an arbitrary ERC20 token (_token) and transfer it to a specified _receiver address. While this functionality provides flexibility for the owner to recover funds, there is a flaw in how it handles withdrawals of the contract's (`standardToken`).

```
   function rescueFunds(address _token, address _receiver, uint256
_amount) external onlyOwner {
        require(_receiver != address(0));
        SafeERC20.safeTransfer(IERC20(_token), _receiver, _amount);
    }
```

When the owner calls `rescueFunds` with _token set to `standardToken`, it triggers a transfer of the token from the contract to the specified _receiver address. However, since the contract's accounting is likely based on the standardToken, this withdrawal could lead to inconsistencies or incorrect accounting records.

Recommendation:
Restrict Standard Token Withdrawals: Modify the rescueFunds function to prevent the withdrawal of (standardToken). This can be achieved by implementing additional checks or by explicitly excluding the token from the list of tokens eligible for withdrawal in `rescueFunds`.

Collateral limit may prevent the position owner from updating their position and avoiding liquidation

The updateMargin function in the Trading contract allows a position owner to either increase or decrease the margin of their open trade. The function retrieves the details of the open trade and performs checks to ensure the caller is the trader of the position and the amount to be added or subtracted is positive. If the margin is being increased, it checks if the new margin would exceed the collateral limit using pairInfo.isExceedGroupsCollateralLimit. If not, it transfers the additional amount from the trader.

However, in a rapidly declining market, the value of collateral can decrease quickly. If the collateral value approaches or exceeds the maximum allowed (as per isExceedGroupsCollateralLimit), traders who wish to add margin to their positions to avoid liquidation may be unable to do so. If traders cannot add margin to their positions, they face automatic liquidation if the margin falls below the maintenance margin.
The check for exceeding the group's collateral limit is not presented during the opening of a position. This inconsistency might allow positions to be opened that are already close to the collateral limit.

Recommendation:
Introduce checks similar to isExceedGroupsCollateralLimit during the position opening phase. This ensures that new positions are not opened too close to the collateral limit, thereby reducing the risk of immediate liquidations in volatile markets

LOW-8 | RESOLVED

Method distributeReward can increment currentBalance without sending tokens

In Contract TradingVault, method distributeReward(...) has a boolean value to check if standardToken needs to be transferred from msg.sender or not.
In case _send is false and StandardToken is not already transferred, currentBalance will be increased by _amount which can lead to wrong calculation of shares and assets.

Recommendation:
Add a check to ensure that currentBalance is equal to amount of Standard tokens in the contract.

LOW-9 | RESOLVED

No address(0) check

In Contract PythOracle.sol, the method rescueFunds() does not validate if _receiver is address(0) or not. If accidentally set to address(0), funds would be lost.

In Contract PairInfos.sol, the method initialize(...) transfers ownership to address _owner which is not validated to be address(0) or not.

In Contract Trading.sol, the method initialize(...) transfers ownership to address _owner which is not validated to be address(0) or not.

Recommendation:
Add a check to validate if the given parameter is address(0) or not.

Confusing Function Naming in _checkTradeType Function

Location: Trading.sol

In the smart contract, there exists a function named `_checkTradeType` with a misleading name that obscures its intended functionality. Despite its name suggesting a check for trade type consistency, the function's actual purpose is to assert that two trade types are not equal.

```
function _checkTradeType(TradeType c, TradeType a) private pure {
     require(c != a, "unsupport");
 }
```

The misleading naming of the `_checkTradeType` function can lead to confusion among developers and maintainers of the smart contract. This confusion may result in misinterpretation of the function's intended behavior, leading to errors or unintended consequences.

Recommendation:
To improve clarity and maintainability of the smart contract code, it is recommended to rename the `_checkTradeType` function to accurately reflect its purpose. A more descriptive name, such as `_assertTradeTypeInequality`, would better convey the function's role in asserting the inequality of trade types.

Additionally, consider adding comments or documentation to the function to provide further context on its intended behavior and usage. Clear and concise naming conventions, coupled with comprehensive documentation, contribute to the readability and understanding of smart contract code.

By adopting clearer naming conventions and providing sufficient documentation, developers can enhance the readability and maintainability of smart contracts, reducing the likelihood of errors and facilitating smoother development and auditing processes.

Safe maths utilized in for-loop

Location: PairInfos.sol

The default safe math operation in solidity versions ^0.8.x incurs extra gas overhead due to it requiring more computation. The following operations in functions (`setPairParams, addPairs and occupiedCollateral`), that are being carried out on the iterator of the for-loop, can be more gas-efficient by using the `unchecked` statement.

One example in PairInfos.sol:

```
function addPairs(Pair[] calldata _pairs) external onlyOwner {
    for (uint256 i = 0; i < _pairs.length; i++) {
        _addPair(_pairs[i]);
    }
}
```

Preferrable to implement it as such:

```
function addPairs(Pair[] calldata _pairs) external onlyOwner {
    for (uint256 i = 0; i < _pairs.length;) {
        _addPair(_pairs[i]);
        unchecked { i++; }
    }
}
```

Recommendation:
Utilize the use of `unchecked` whenever possible as long as arithmetic safety is ensured within the context of calculation

	./TradingStorage.sol ./TradingVault.sol ./Trading.sol ./PythOracle.sol ./libs/VarConstant.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	./libs/TradingHelper.sol ./libs/Types.sol ./PairInfos.sol ./FeeHelper.sol ./Referrals.sol
Reentrance	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the Rollie team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the Rollie team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo