QuillAudits

Audit Report, March, 2024

For

ZOTH

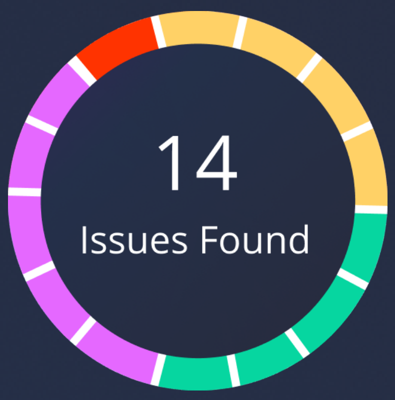
Table of Content

Table of Content

Executive Summary

Project Name	Zoth Pool
Project URL	https://zoth.io/
Overview	Zoth is an ecosystem that connects liquidity across Traditional Finance and Onchain Finance, effectively expediting the influx of assets and capital between these two sectors.
Audit Scope	*https://github.com/0xZothio/contracts/tree/main/contracts/V3*
Contracts in Scope	Branch: Main Contracts:- -ZothPool.sol -IV3ZothPool.sol -IWhitelistManager.sol
Commit Hash	2510f3df
Language	Solidity
Blockchain	Ethereum
Method	Manual Review, Automated tools,Functional testing
Review 1	02nd Feb 2024 - 13th Feb 2024
Updated Code Received	16th Feb 2024
Review 2	17th Feb 2024
Fixed In	ec2a8d373663bace86914d3b4b147c7d5a732e9e

Number of Issues per Severity



🟥 High	🟨 Medium
🟩 Low	🟪 Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	1	0	2
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	3	4	3

Checked Vulnerabilities

- ✓ Access Management
- ✓ Arbitrary write to storage
- ✓ Centralization of control
- ✓ Ether theft
- ✓ Improper or missing events
- ✓ Logical issues and flaws
- ✓ Arithmetic Correctness
- ✓ Race conditions/front running
- ✓ SWC Registry
- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Malicious libraries
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ ERC's conformance
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Multiple Sends
- ✓ Using suicide
- ✓ Using delegatecall
- ✓ Upgradeable safety
- ✓ Using throw

Checked Vulnerabilities

✓ Using inline assembly	✓ Unsafe type inference
✓ Style guide violation	✓ Implicit visibility level

Techniques and Methods

Throughout the audit of smart contracts, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments, match logic and expected behavior.
- Token distribution and calculations are as per the intended behavior mentioned in the whitepaper.
- Implementation of ERC standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods, and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analyzed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

A static Analysis of Smart Contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual Analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analyzed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behavior of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle,Solhint, Mythril, Slither, Solidity static analysis.

Types of Severity

Every issue in this report has been assigned to a severity level. There are four levels of severity, and each of them has been explained below.

High Severity Issues

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium Severity Issues

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low Severity Issues

Low-level severity issues can cause minor impact and are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are four severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

A. Contract - ZothPool.sol

High Severity Issues

A.1 Emergency Withdrawal Logic Flaw in emergencyWithdrawal

Line 225-247

Function - emergencyWithdraw

```
function emergencyWithdraw(uint256 id) external {
        Deposit memory depositData = lenders[msg.sender].deposits[id];

        require(depositData.amount != 0, "You have nothing with this ID");
        require(
        block.timestamp <
        depositData.startDate + depositData.lockingDuration,
        "You can not emergency withdraw"
        );
        uint256 depositedAmount = depositData.amount;

        uint256 withdrawFee = (depositedAmount * _withdrawPenaltyPercent) /
1E2;
        uint256 refundAmount = depositedAmount - withdrawFee;
        delete lenders[msg.sender].deposits[id];
        _totalWithdrawFee = _totalWithdrawFee + depositedAmount -
refundAmount;

        _updateId(msg.sender);

        IERC20(tokenAddresses[depositData.tokenId]).transfer(
        msg.sender,
        refundAmount
        );
        }
```

Description

The emergencyWithdraw function allows users to withdraw their deposits before the maturity date, imposing a penalty percentage (_withdrawPenaltyPercent) on the withdrawn amount. However, this mechanism has a critical flaw due to the absence of a check to ensure _withdrawPenaltyPercent is set. If _withdrawPenaltyPercent is not properly initialized, users might exploit this oversight to withdraw funds without incurring the expected penalty. This could lead to several adverse outcomes, including:

- **Loss to the Protocol:** Users could deposit funds, watch for a more favorable interest rate or opportunity, and then perform an emergency withdrawal without penalty, subsequently redepositing at the higher rate. This behavior could exploit the protocol's financial mechanisms and lead to unintended losses or imbalances.
- **Incentive Misalignment:** The lack of a penalty disincentivizes long-term holding, undermining the protocol's economic stability and its ability to manage liquidity effectively.

Remediation

1. **Ensure Penalty Initialization:** Implement a mechanism to guarantee that _withdrawPenaltyPercent is set to a sensible default upon contract deployment or initialization. This could involve setting a non-zero default value or requiring that this variable be initialized through a constructor parameter or an initialization function.
2. **Validation Check:** Add a require statement to confirm that _withdrawPenaltyPercent is greater than 0 before calculating and applying the withdrawal fee. This ensures that the penalty mechanism cannot be bypassed due to configuration oversights.

Status

Resolved (Intended functionality)

Medium Severity Issues

A.2 Inconsistency in Rate setting and validation in **changeBaseRates** and **setWithdrawRate** functions

Line **changeBaseRates & setWithdrawRate**

253-264

```
function changeBaseRates(uint256 baseStableApr) external {
        require(
        whitelistManager.isPoolManager(msg.sender),
        " USER_IS_NOT_POOL_MANAGER "
        );
        require(baseStableApr < 10_001, "Invalid Stable Apr");
        uint256 newStableApr = baseStableApr;
        unchecked {
        ++_currentRateRound;
        }
        rateRounds[_currentRateRound] = RateInfo(newStableApr,
block.timestamp);
        }
```

270-274

```
function setWithdrawRate(uint256 newRate) external {
        require(whitelistManager.isOwner(msg.sender), "
USER_IS_NOT_OWNER ");
        require(newRate < 10_000, "Rate can not be more than 100%");
        _withdrawPenaltyPercent = newRate;
        }
```

Description

The code snippets for changeBaseRates and setWithdrawRate functions contain logic intended to update financial rates within the contract. However, there's a critical inconsistency in how the conditions for these rates are checked against their intended use, particularly regarding basis points and percentage values. This inconsistency could lead to confusion, potential Denial of Service (DoS) conditions, or financial loss due to misconfiguration.

1. **Ambiguity in Rate Representation:** The conditions check in both functions seems to assume rates are represented in basis points (where 10,000 basis points equal 100%). However, the comments and error messages suggest a percentage-based understanding, which can confuse administrators and lead to incorrect rate settings.

2. **Inadequate Validation:** The changeBaseRates function allows a maximum value of 10_001 for baseStableApr, which exceeds the logical maximum for percentage-based calculations in basis points (10,000 for 100%). Similarly, setWithdrawRate caps newRate at 10_000, aligning with the maximum for basis points but potentially conflicting with the expected percentage-based input due to ambiguous documentation or comments.

Remediation

1. **Clarify Documentation:** Ensure that the documentation, comments, and error messages clearly state whether rates are expected in basis points or as percentages. This clarity will help prevent misconfiguration.

2. **Standardize Rate Representation:** Choose a consistent representation for all rate values within the contract (either basis points or percentages) and adjust the validation logic to reflect this choice. This standardization simplifies understanding and reduces the risk of errors.

3. **Adjust Validation Logic:** If rates are intended to be in basis points, validate inputs accordingly and ensure that error messages accurately reflect this. For example, a maximum baseStableApr of 10_000 (for 100%) might be more appropriate, and similar logic should apply to setWithdrawRate.

Status

Acknowledged (false positive)

A.3 Inadequate Locking Duration and End Date Checks

Line	Function - depositByLockingPeriod
146-154	see code below

```
lenderData.deposits[currentId] = Deposit(
    _amount,
    apr,
    lockingPeriod,
    block.timestamp,
    block.timestamp + lockingPeriod, // @audit the endDate should not exceed the
tenure time period
    block.timestamp,
    _tokenId
);
```

Description

The code responsible for creating a new deposit entry for lenders lacks proper validation for the locking duration and the calculation of the end date. Specifically, the endDate is set by simply adding the lockingPeriod to the current block timestamp. This approach may not account for the overall tenure time period of the deposit or ensure that the endDate is logically consistent with the contract's operational parameters.

1. **End Date Calculation Overlooks Tenure Limits:** By setting the endDate as block.timestamp + lockingPeriod without further checks, there's a risk that the end date could extend beyond an allowed tenure time period or create inconsistencies with the expected lifecycle of a deposit.

Remediation

1. **Implement Locking Duration Checks:** Introduce validation logic to ensure the lockingPeriod complies with the contract's rules regarding minimum and maximum locking durations. This step ensures that all deposits adhere to predefined constraints, enhancing predictability and security.

2. **Validate End Date Against Tenure Limits:** Ensure that the calculated endDate does not exceed any set tenure limits or fall outside expected operational parameters. This might involve checking against a maximum allowed deposit duration or ensuring the end date does not extend beyond certain contract milestones or expiration dates.

Status

Resolved

A.4 Reward Calculation During Inactive Pool Period

Line **Functions - depositByLockingPeriod**

155-166

```
function depositByLockingPeriod(
          uint256 _amount,
          uint256 _lockingDuration,
          uint256 _tokenId
          ) public returns (uint256) {}
```

Description

The current implementation of _calculateFormula calculates rewards based solely on the deposit amount, the duration of the deposit, and the rate. It does not take into account whether the pool is active or has already reached its tenure. This oversight means that rewards continue to be calculated and potentially distributed for deposits extending beyond the pool's active tenure, leading to unwarranted reward payouts that could affect the pool's financial integrity and sustainability.

Remediation

Implement and Enforce Duration Limits: As part of broader contract safeguards, ensure that deposit durations do not exceed the pool's tenure through proper validation during the deposit process. This preemptive check will reduce the need for complex adjustments during reward calculations.

Status

Resolved

A.5 Unrestricted Modification of Critical Pool Variables

Line **Function - setContractVariables**

80
```
function setContractVariables(
          uint256 _tenure,
          uint256 _poolId,
          uint256 _hotPeriod,
          uint256 _coolDownPeriod,
          address[] memory _tokenAddresses
          ) external {}
```

Description

The setContractVariables function allows for the modification of critical variables related to the pool's operation, such as its tenure, pool ID, hot period, cooldown period, and associated token addresses. This function can be called multiple times, potentially altering the fundamental parameters of the pool post-deployment.

While the function is restricted to addresses authorized as Pool Managers via the whitelistManager.isPoolManager(msg.sender) check, the ability to change key operational parameters after the pool has been established and potentially after users have engaged with it introduces risks. These include confusion among participants, users tokens lost if pool tenure is decreased than the users locking duration, potential mismatches in user expectations versus pool behavior, and opportunities for manipulation if the changes are not transparent or understood by all stakeholders.

Remediation

To mitigate these risks, consider implementing one or more of the following controls:

- **Immutability:** If these variables are not intended to change over the lifetime of the contract, consider setting them once upon contract deployment (e.g., through the constructor) and making them immutable.

- **Governance Process:** If changes must be possible, introduce a governance process for making such changes. This could include timelocks (delays before changes take effect), multi-signature approval requirements, or requiring a consensus from a broader set of stakeholders (e.g., token holders).

- **Event Logging:** Ensure that any changes to these critical variables emit events that clearly log the old and new values. This transparency helps users and external systems track changes and understand the current state of the pool.
- **Change Limitations:** Implement limitations on how often or under what conditions these variables can be changed to prevent abuse or ensure stability.

Status

Resolved

Low Severity Issues

A.6 Incorrect Token URI for NFT Metadata

Line **Function - _mintNFTAfterDeposit**

165
```
if (_amount <= 10000 * 10 ** 6) {
            // blue // (1, URL)
_setTokenURI(newTokenId, "https://gateway.pinata.cloud/ipfs/
QmeRhd2icJLyNbD9yzKoiJUvxtBw4u43JB25jzt73vMv28");
```

Description

The provided token URI points directly to the animation or a specific part of the NFT content, rather than the JSON metadata file that describes the NFT's attributes, image, and other properties according to the ERC-721 metadata standard. The metadata JSON is crucial for NFT platforms to correctly display and utilize the NFT, including its visual representation, attributes, and any related content. Incorrect or incomplete metadata can lead to a suboptimal experience for users, potentially affecting the NFT's visibility, utility, and value.

Remediation

1. **Correct the Token URI:** Ensure that the token URI points to a valid JSON metadata file hosted on a reliable and persistent storage service (like IPFS). The JSON file should conform to the ERC-721 metadata schema, including fields for name, description, image, and any additional attributes relevant to the NFT.

2. **Validate Metadata Content:** Before setting the token URI in the smart contract, verify the correctness and completeness of the metadata content. This step can prevent issues related to missing information or incorrect links within the metadata.

Status

Resolved

A.7 Unchecked Transfer Return Values

Line	Functions

220	// In withdrawUsingDepositId function
	IERC20(tokenAddresses[depositData.tokenId]).transfer(msg.sender, stableAmount);

240	// In emergencyWithdraw function
	IERC20(tokenAddresses[depositData.tokenId]).transfer(msg.sender, refundAmount);

Description

The ERC-20 transfer function, according to the ERC-20 standard, returns a boolean value indicating the success or failure of the operation. Ignoring this return value can lead to situations where a transfer fails (due to reasons like insufficient balance, or transfer to a contract that doesn't accept tokens), but the contract continues execution as if it had succeeded. This oversight could cause discrepancies in the contract's internal accounting, leading to potential vulnerabilities or loss of funds under specific conditions.

Remediation

1. Using require to Assert Successful Transfer

2. **Using SafeERC20 Library from OpenZeppelin:** The OpenZeppelin contracts library provides a SafeERC20 wrapper around the ERC20 interface, which automatically checks the return value of transfer, transferFrom, and other ERC-20 operations, and reverts the transaction if any operation fails.

Status

Resolved

A.8 All State-changing methods are missing event emissions

Line	Functions - All functions
NA	All State changing functions

Description

Critical functions within the contract lack the emission of events. Events play a vital role in providing transparency, enabling external systems to react to changes, and offering a way to track important contract activities. The absence of events can hinder monitoring and auditing efforts, making it difficult to detect and respond to critical contract actions.

Remediation

To address this issue and improve the transparency and auditability of the contract, you should add appropriate event emissions in critical functions. These events should capture essential information about the function's execution, including input parameters and outcomes. Additionally, consider emitting events both before and after critical state changes, where applicable.

Status

Resolved

A.9 Unused codes and Imports

Line **Function - transferFee**

```
        // ZothPool.sol

7       import "@openzeppelin/contracts/utils/math/SafeMath.sol"; // Not needed
        above versions 0.8.0
40      uint256 public hotPeriod;
41      uint256 public cooldownPeriod;

        // IV3ZothPool.sol
        struct Lender {
6          uint256 amount;
7          uint256 pendingStableReward;
        }
```

Description

The contract contains sections of code that are defined but not referenced or executed during the contract's operation. This unused code might include functions, variables, or entire code blocks that serve no functional purpose or are not part of the contract's intended behavior. Unused code can increase the contract's complexity, potentially introducing confusion for developers and auditors and leading to unnecessary gas consumption.

Remediation

To address this issue, carefully review the contract code and remove any sections that are not actively used or required for the contract's functionality. This helps streamline the codebase, making it more concise and easier to manage. If there are functions or variables that were intended for future use or testing, consider either implementing them or commenting on their purpose to provide clarity.

Status

Resolved

Informational Issues

A.10 Insufficient Balance Check Before Transfer

Line	Function - NA
163	// In withdrawUsingDepositId function
225	// In emergencyWithdraw function

Description

In both the withdrawUsingDepositId and emergencyWithdraw functions, the smart contract transfers a certain amount of tokens to the user's address. However, there is no explicit check to ensure that the contract itself has a sufficient balance of the specified token before attempting the transfer. While the ERC-20 token's transfer function should fail if the contract does not have enough tokens, preemptively checking the contract's balance can provide clearer error messages and avoid unnecessary gas expenditure for doomed transactions.

Remediation

To enhance clarity and efficiency, it's advisable to add a balance check before executing token transfers.

Status

Resolved

A.11: Users can’t claim the reward until they withdraw the deposited amount.

Description

Users won’t be able to claim the rewards until or unless they withdraw the rewards after the duration is finished.

Status

Acknowledged

A.12: Bonus rewards are not being calculated, but mentioned in the comment

Line

163

```
 * @dev Calculates both the bonus reward and stable rewards for deposits with
locking period
function _calculateRewards(
        address _lender,
        uint256 _id,
        uint256 _endDate
        ) private view returns (uint256) {
```

Description

The bonus rewards calculation is not being done, but it is mentioned in the comments.

Status

Acknowledged

A.13: Hardcoded Token Decimal Values

Description

The smart contract uses hardcoded decimal values to validate token amounts, such as in conditional statements that check if an _amount is within a certain range. This approach can lead to issues if tokens with different decimal places are added to the system in the future. Since ERC-20 tokens can have a wide range of decimal values (although 18 decimals is standard for many tokens), assuming a specific decimal value in the contract logic can create inaccuracies in amount validations.

Status

Resolved

A.14: Consider using custom errors

Description

Custom errors reduce the contract size and can provide easier integration with a protocol. Consider using those instead of require statements with string error.

Status

Resolved

Functional Tests

Some of the tests performed are mentioned below:

ZothPool

- ✓ Should initiate the contract with provided USDC address
- ✓ Should initiate the contract with provided WhitelistManager address
- ✓ Should initiate the contract with provided owner address
- ✓ Should Set the contract Variables by Owner
- ✓ Should not Set the contract Variables by other than Pool Manager
- ✓ Should not deposit if not whitelisted
- ✓ Should add Whitelist
- ✓ Should deposit if whitelisted
- ✓ Should deposit with locking duration
- ✓ Should check Base APR
- ✓ Should check Active Deposits
- ✓ Should withdraw the funds related to deposit ID EMERGENCY
- ✓ Should withdraw by ID but Restriction of Time
- ✓ Should withdraw all funds
- ✓ Should transfer funds to another account
- ✓ Should Set the contract Variables by Owner
- ✓ Should not Set the contract Variables by other than Pool Manager
- ✓ Should not deposit if not whitelisted
- ✓ Should revert withdraw if deposit ID is invalid
- ✓ Should revert withdraw if locking duration is not over
- ✓ Should revert deposit if token address is invalid or user not whitelisted
- ✓ Should revert deposit if locking duration if user not whitelisted or amount 0
- ✓ Should set contract variables by owner

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the Zoth Pool. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits Smart contract security audit provides services to help identify and mitigate potential security risks in **Zoth Pool** smart contracts. However, it is important to understand that no security audit can guarantee complete protection against all possible security threats. QuillAudits audit reports are based on the information provided to us at the time of the audit, and we cannot guarantee the accuracy or completeness of this information. Additionally, the security landscape is constantly evolving, and new security threats may emerge after the audit has been completed.

Therefore, it is recommended that multiple audits and bug bounty programs be conducted to ensure the ongoing security of **Zoth Pool** smart contracts. One audit is not enough to guarantee complete protection against all possible security threats. It is important to implement proper risk management strategies and stay vigilant in monitoring your smart contracts for potential security risks.

QuillAudits cannot be held liable for any security breaches or losses that may occur subsequent to and despite using our audit services.. It is the responsibility of the **Zoth Pool** to implement the recommendations provided in our audit reports and to take appropriate steps to mitigate potential security risks.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

1000+
Audits Completed

$30B
Secured

1M+
Lines of Code Audited

Follow Our Journey

Audit Report
March, 2024

For

ZOTH

QuillAudits

Canada, India, Singapore, UAE, UK

www.quillaudits.com

audits@quillhash.com