Guild Audits

JANUARY
16TH, 2025

# LODE PROTOCOL FINAL AUDIT REPORT

By Guild Audits

# TABLE OF CONTENTS

## DISCLAIMER

The Guild Audit team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

## EXECUTIVE SUMMARY

This section will represent the summary of the whole audit.

## PROJECT SUMMARY

**Project Name:** Lode Protocol

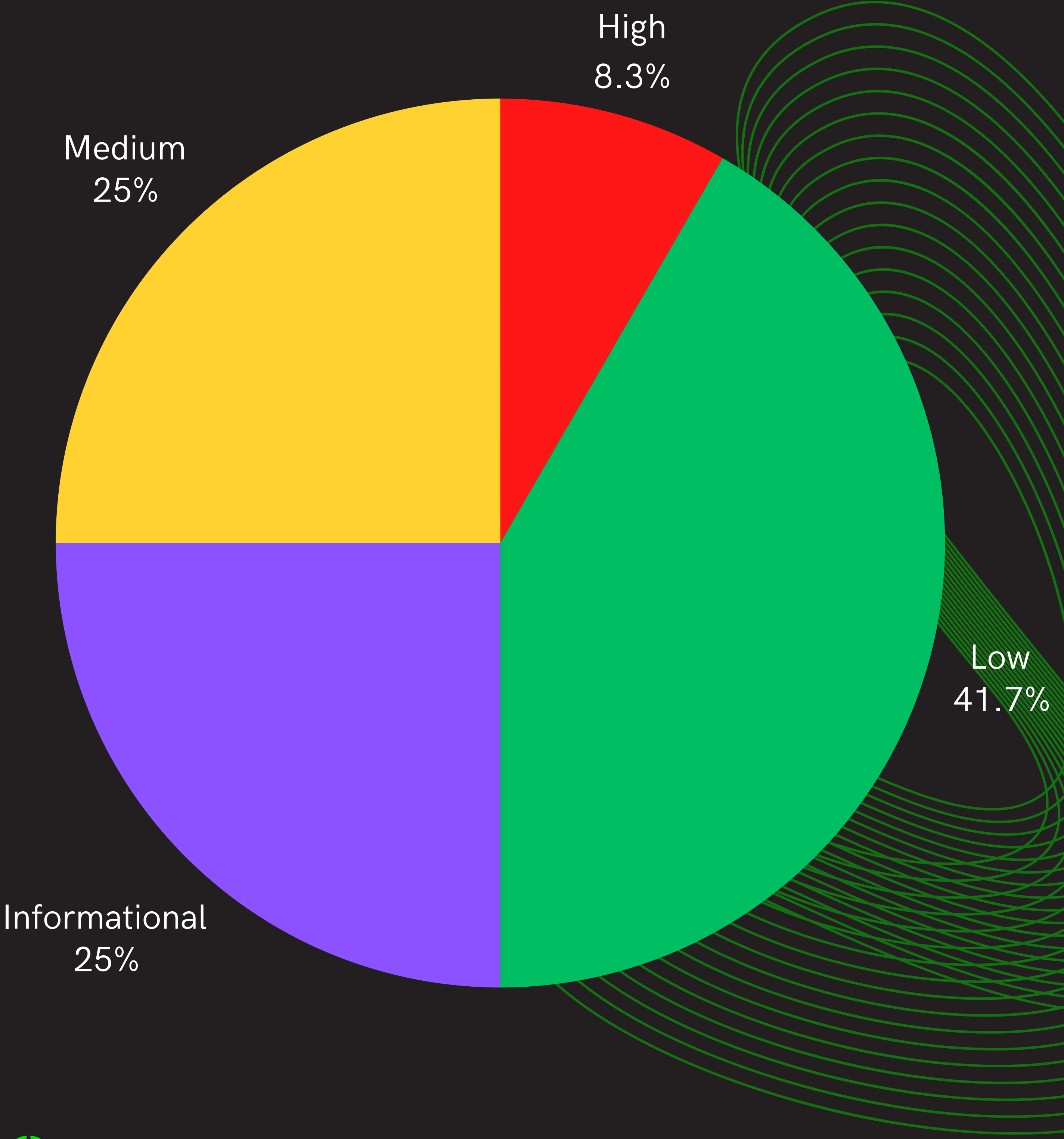
**Description**: The Lode Protocol has two different contracts,the first contract handles KYC verification and delegation using Merkle trees and the second contract handles payment collection to a multisig contract. Payment can be done with ether or any approved ERC20 token.
**Codebase**: https://github.com/Lode-Protocol/contracts
**Commit**:https://github.com/Lode-Protocol/contracts/commit/8f76052fe46897edcc150d67c91e109cb53b16ec

## PROJECT AUDIT SCOPE AND FINDINGS

The motive of this audit is to review the codebase of Lode Protocol contracts for the purpose of achieving secured, correctness and quality smart contracts. Number of Contracts in Scope: All the contract (145 SLOC)
**Duration for Audit:** 7 days

**Vulnerability Summary:**
Total issues: 12
Total High: 1
Total Medium: 3
Total Low: 5
Total Informational: 3

**Chart Illustration**



## FINDINGS

| Index | Title | Severity | Statusof Issues |
|---|---|---|---|
| 01 | [H-1]Timestamp in the LodeProtocolKYC::verifyKYC will cause the function to always fail. | High | Resolved |
| 02 | [M-1]EIP712 not implemented in LodeProtocolKYC | Medium | Resolved |
| 03 | [M-2]Transactions done with PaymentContract::payToken will always fail for fee on transfer token | Medium | Resolved |
| 04 | [M-3]PaymentContract::isERC20Token does not have adequate check to validate an ERC20 token | Medium | Partially Resolved |
| 05 | [L-1]LodeProtocolKYC::verifyKYC does not have a signature deadline check | Low | Resolved |
| 06 | [L-2]The `ecrecover` in recoverSigner function is susceptible to signature malleability | Low | Resolved |
| 07 | [L-3]PayEther function uses transfer instead of call | Low | Resolved |
| 09 | [L-5]Insufficient check for multisig contract in setMultisigAddress function | Low | Resolved |
| 10 | [I-1]Current delegate can be added again in modifyDelegate function | Info | Resolved |
| 11 | [I-2]No zero amount check in PayToken | Info | Resolved |
| 12 | [I-3]PayEther emits address(0) instead of the standard native ether Address | Info | Resolved |

# MODE OF AUDIT AND METHODOLOGIES

The mode of audit carried out in this audit process is as follows:
**Manual Review:** This is the first and principal step carried out to understand the business logic behind a project. At this point, it involves research, discussion and establishment of familiarity with contracts. Manual review is critical to understand the nitty-gritty of the contracts.
**Automated Testing:** This is the running of tests with audit tools to further cement discoveries from the manual review. After a manual review of a contract, the audit tools used are Slither, Echidna and others.
Functional Testing: Functional testing involves manually running unit, static, and dynamic testing. This is done to find out possible exploit scenarios that could be used to exploit the contracts.

**The methodologies for establishing severity issues:**

- High Level Severity Issues ⏫
- Medium Level Severity Issues ^
- Low Level Severity Issues ⌄
- Informational Level Severity Issues ●

## TYPES OF SEVERITY

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

**High Severity Issues:** These are critical issues that can lead to a significant loss of funds, compromise of the contract's integrity, or core function of the contract not working. Exploitation of such vulnerabilities could result in immediate and catastrophic consequences, such as:

- Complete depletion of funds.
- Permanent denial of service (DoS) to contract functionality.
- Unauthorized access or control over the contract.

**Medium Severity Issues:** These issues pose a significant risk but require certain preconditions or complex setups to exploit. They may not result in immediate financial loss but can degrade contract functionality or pave the way for further exploitation. Exploitation of such vulnerabilities could result in partial denial of service for certain users, leakage of sensitive data or unintended contract behaviour under specific circumstances.

**Low Severity Issues:** These are minor issues that have a negligible impact on the contract or its users. They may affect efficiency, readability, or clarity but do not compromise security or lead to financial loss. Impacts are minor degradation in performance, confusion for developers or users interacting with the contract and low risk of exploitation with limited consequences.

**Informational:** These are not vulnerabilities in the strict sense but observations or suggestions for improving the contract's code quality, readability, and adherence to best practices. There is no direct impact on the contract's functionality or security, it is aimed at improving code standards and developer understanding.

# TYPES OF ISSUES

Open

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

Resolved

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

Acknowledged

Vulnerabilities that 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.

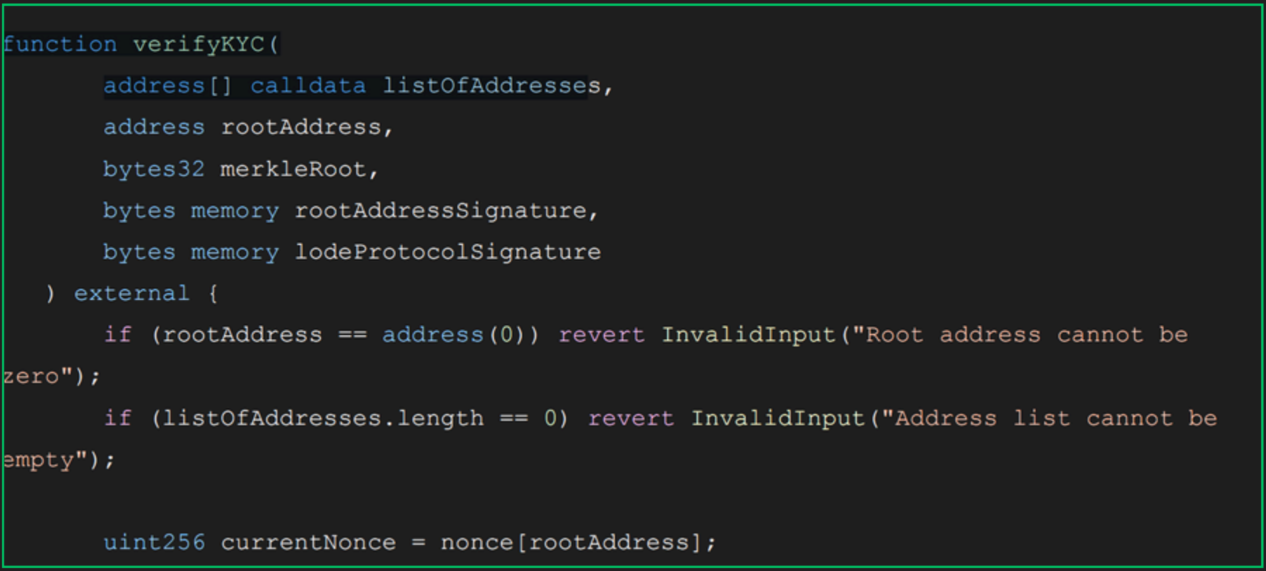
# REPORT OF FINDINGS

## HIGH SEVERITY ISSUES

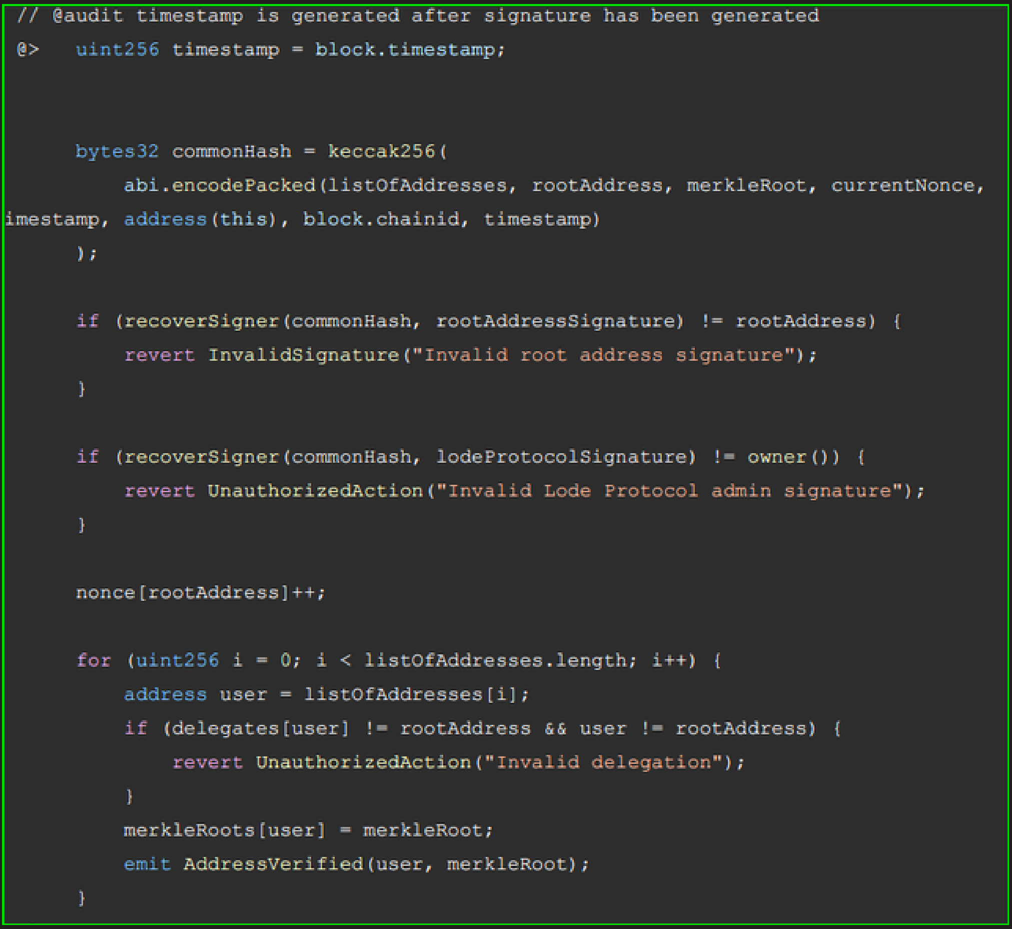
- **Issue**: [H-1] Timestamp in the LodeProtocolKYC::verifyKYC will cause the function to always fail.

- **Description**:

The LodeProtocolKYC::verifyKYC accepts two signatures, the rootAddressSignature and lodeProtocolSignature. These signatures are generated using the listOfAddresses, rootAddress, merkleRoot,nonce,timestamp,address of the contract and the chainId. The rootAddress user and the lodeProtocol admin create their signatures using the above information. The problem is that they need to guess the timestamp that the verifyKYC will generate when it is called, which is nearly impossible, causing the verifyKYC to always fail. function verifyKYC( address[] calldata listOfAddresse



```
function verifyKYC(
        address[] calldata listOfAddresses,
        address rootAddress,
        bytes32 merkleRoot,
        bytes memory rootAddressSignature,
        bytes memory lodeProtocolSignature
    ) external {
        if (rootAddress == address(0)) revert InvalidInput("Root address cannot be
zero");
        if (listOfAddresses.length == 0) revert InvalidInput("Address list cannot be
empty");

        uint256 currentNonce = nonce[rootAddress];
```



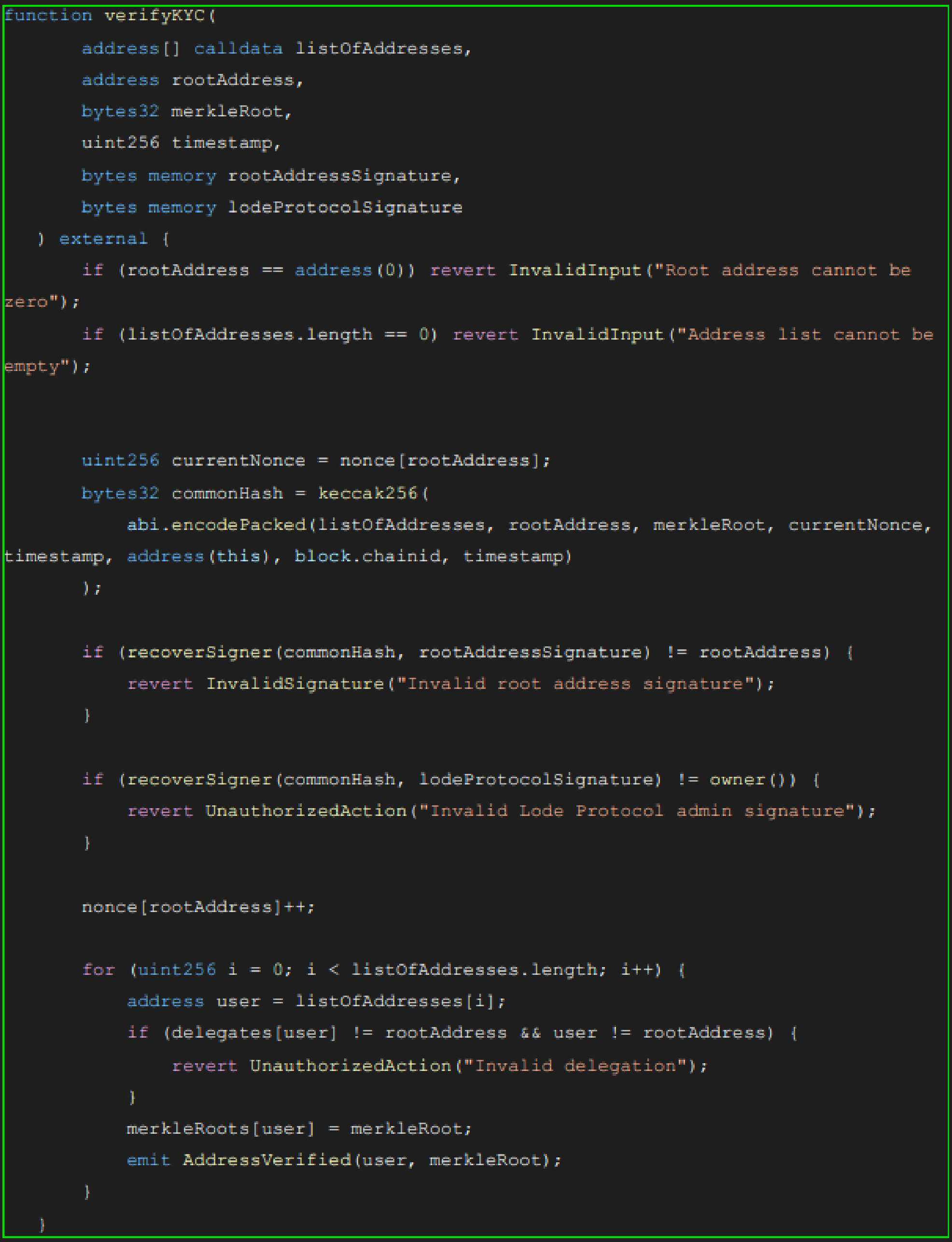
```
 // @audit timestamp is generated after signature has been generated
 @>    uint256 timestamp = block.timestamp;


       bytes32 commonHash = keccak256(
           abi.encodePacked(listOfAddresses, rootAddress, merkleRoot, currentNonce,
imestamp, address(this), block.chainid, timestamp)
       );

       if (recoverSigner(commonHash, rootAddressSignature) != rootAddress) {
           revert InvalidSignature("Invalid root address signature");
       }

       if (recoverSigner(commonHash, lodeProtocolSignature) != owner()) {
           revert UnauthorizedAction("Invalid Lode Protocol admin signature");
       }

       nonce[rootAddress]++;

       for (uint256 i = 0; i < listOfAddresses.length; i++) {
           address user = listOfAddresses[i];
           if (delegates[user] != rootAddress && user != rootAddress) {
               revert UnauthorizedAction("Invalid delegation");
           }
           merkleRoots[user] = merkleRoot;
           emit AddressVerified(user, merkleRoot);
       }
```

- **Impact of the vulnerability**

1. LodeProtocolKYC::verify will always fail
2. Users will not be able to verify their addresses

- **Recommendation**:

The timestamp, which is most likely a deadline, should be added as a parameter instead of being generated in the LodeProtocolKYC::verify function.



```
function verifyKYC(
        address[] calldata listOfAddresses,
        address rootAddress,
        bytes32 merkleRoot,
        uint256 timestamp,
        bytes memory rootAddressSignature,
        bytes memory lodeProtocolSignature
    ) external {
        if (rootAddress == address(0)) revert InvalidInput("Root address cannot be
zero");
        if (listOfAddresses.length == 0) revert InvalidInput("Address list cannot be
empty");


        uint256 currentNonce = nonce[rootAddress];
        bytes32 commonHash = keccak256(
            abi.encodePacked(listOfAddresses, rootAddress, merkleRoot, currentNonce,
timestamp, address(this), block.chainid, timestamp)
        );

        if (recoverSigner(commonHash, rootAddressSignature) != rootAddress) {
            revert InvalidSignature("Invalid root address signature");
        }

        if (recoverSigner(commonHash, lodeProtocolSignature) != owner()) {
            revert UnauthorizedAction("Invalid Lode Protocol admin signature");
        }

        nonce[rootAddress]++;

        for (uint256 i = 0; i < listOfAddresses.length; i++) {
            address user = listOfAddresses[i];
            if (delegates[user] != rootAddress && user != rootAddress) {
                revert UnauthorizedAction("Invalid delegation");
            }
            merkleRoots[user] = merkleRoot;
            emit AddressVerified(user, merkleRoot);
        }
    }
```

**Status: Resolved**

## MEDIUM SEVERITY ISSUES ^

- **Issue:** [M-1] EIP712 not implemented in LodeProtocolKYC

- **Description:**

The LodeProtocolKYC involves signing and verification of signatures but does not use the EIP712 standard. This standard is used for typed structured data hashing and signing.

- **Impact of the vulnerability**

1. The use of some of the signed data is not clear, i.e we have two timestamps in the common hash but we do not know the use or what they represent.

**Recommendations:**
Implement the EIP712 standard in the LodeProtocolKYC contract.

**Status:** Resolved

- **Issue:** [M-2] Transactions done with PaymentContract::payToken will always fail for fee on transfer token

- **Description:**

When a fee on transfer token is approved by the owner, any transaction done using the PaymentContract::payToken will always fail because the transfer is done in two fold, first the token is transferred to the contract and then it is transferred to the multisig address. After the first transfer the amount transferred will reduce due to the fee applied on the transfer and at the point of transfer to the multisig, there will be insufficient balance in the contract because the contract is trying to send the full amount.

```
function payToken(
        address tokenAddress,
        uint256 amount,
        string memory transactionId
    ) external {
        if (!allowedTokens[tokenAddress]) revert TokenNotAllowed(tokenAddress);
        IERC20 token = IERC20(tokenAddress);
  @>      if (!token.transferFrom(msg.sender, address(this), amount)) revert
TokenTransferFailed(tokenAddress);
  @>      if (!token.transfer(multiSigAddress, amount)) revert
TokenTransferFailed(tokenAddress);

        emit PaymentReceived(tokenAddress, amount, msg.sender, transactionId);
    }
```

- **Impact of the vulnerability**

1. Sending fee on transfer token will always fail

- **Recommendations:**

Send the funds directly to the multisig address.

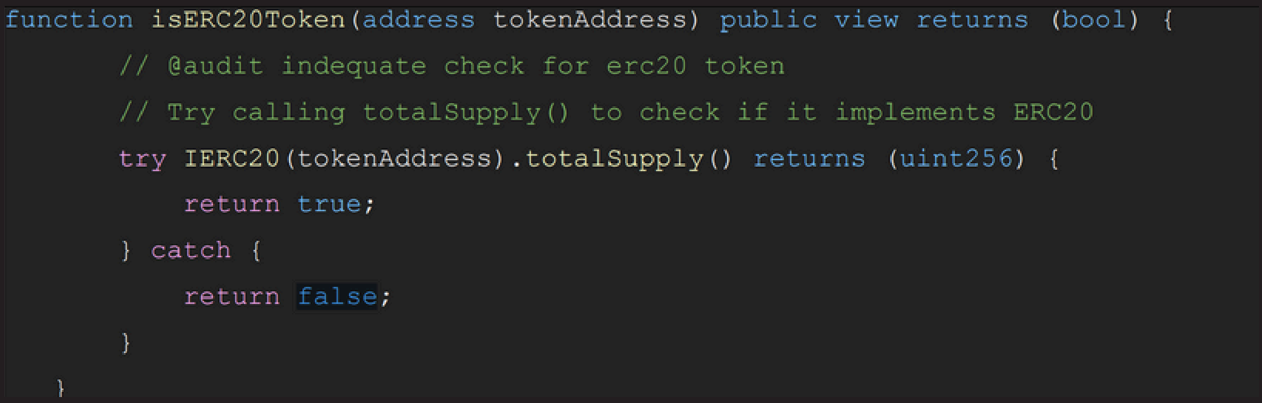
```
function payToken(
        address tokenAddress,
        uint256 amount,
        string memory transactionId
    ) external {
        if (!allowedTokens[tokenAddress]) revert TokenNotAllowed(tokenAddress);


        IERC20 token = IERC20(tokenAddress);
        if (!token.transferFrom(msg.sender, multiSigAddress, amount)) revert
TokenTransferFailed(tokenAddress);
        emit PaymentReceived(tokenAddress, amount, msg.sender, transactionId);
    }
```

**Status: Resolved**

**Issue:** [M-3] PaymentContract::isERC20Token does not have adequate checks to validate an ERC20 token.

- **Description:**

The PaymentContract::isERC20Token function only checks if the token has a totalSupply function. This check is not enough because ERC777 also has the totalSupply function and will pass the check.



```
function isERC20Token(address tokenAddress) public view returns (bool) {
        // @audit indequate check for erc20 token
        // Try calling totalSupply() to check if it implements ERC20
        try IERC20(tokenAddress).totalSupply() returns (uint256) {
            return true;
        } catch {
            return false;
        }
    }
```

**Impact:** 1. ERC777 will pass the isERC20Token check

- **Recommendation:**

1. Add a check for decimal function.
2. If you add an allowance() function check on isERC20Token() function an ERC777 contract will still pass because it also has an allowance() function. Instead add a check for send() or granularity() or revokeOperator(). These functions only exist in ERC777 so if it returns true then the token is not an ERC20.

**Status:** Partially Resolved

## LOW SEVERITY ISSUES

**Issue**: [L-1] LodeProtocolKYC::verifyKYC does not have a signature deadline check

**Description**:
The LodeProtocol::verifyKYC does not have a check for the signature deadline. The signatures generated by rootAddress and LodeProtocol admin can be used at any time.

**Impact of the vulnerability :**
1. rootAddressSignature and lodeProtocolSignature can be used at any time.

**Recommendation:**
Add a signature deadline check to LodeProtocol::verifyKYC

**Status:** Resolved

- **Issue:** [L-2] The `ecrecover` in recoverSigner function is susceptible to signature malleability

- **Description:**

The recoverSigner currently uses the ecrecover() function to determine the signer of signatureData: This approach has the following issues:
**Error Handling:** A failed signature verification returns 0 rather than reverting with an error, potentially leading to incorrect error handling downstream.
**Signature Manipulation:** Direct use of **ecrecover**() is vulnerable to manipulation. For example, adjusting the signatureData.r parameter can yield a random address, allowing for incorrect or malicious signatures to be processed as valid.

- **Impact of the vulnerability**

1. Signature can be manipulated when ecrecover is used directly.

**Recommendation:**
To improve security and error handling, use the OpenZeppelin ECDSA library for signature verification. This library provides a secure and reliable implementation of ecrecover() that includes robust error handling and mitigates the risk of signature manipulation. Incorporating this library will ensure more accurate and secure signature verification in the contract.

**Status:** Resolved

- **Issue:** [L-3] PayEther function uses transfer instead of call

- **Description:**

The PayEther function uses the Solidity transfer method to send Ether. The transfer method has been known to introduce risks, particularly in the context of gas limit changes. Since the transfer method imposes a fixed 2300 gas stipend for the recipient, it can lead to reverts if the recipient is a smart contract with higher gas requirements for its fallback or receive function.

- **Impact of the vulnerability :**

1. If the recipient is a smart contract that requires more than 2300 gas to process the transfer, the transaction will fail, potentially halting the contract's functionality.

- **Recommendation**

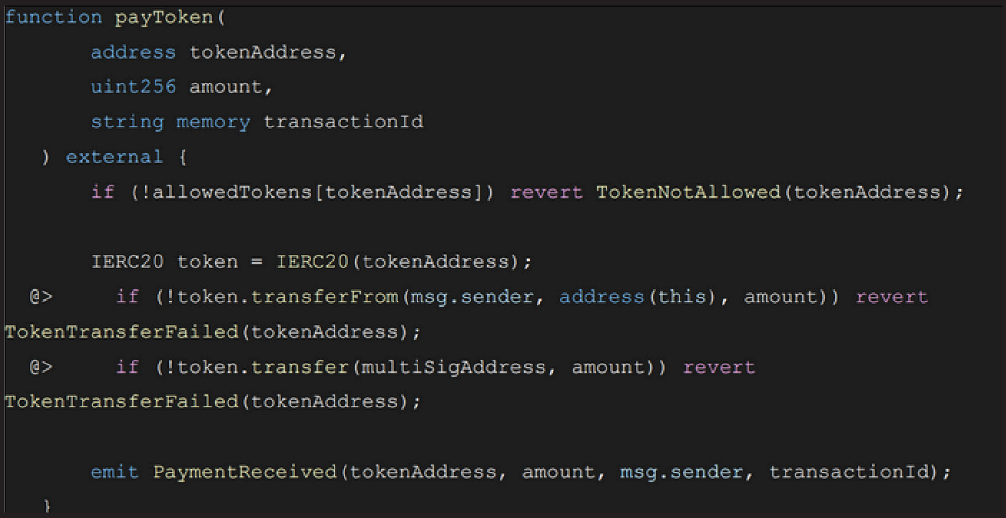
Replace the use of transfer with the call method to ensure greater flexibility and compatibility with recipient contracts. The call method allows specifying gas and handles the risks associated with the 2300 gas stipend limitation.

**Status:** **Resolved**

**Issue:** [L-4] PayToken function assumes all ERC20 returns bool values

**Description:**

The PayToken function assumes that all ERC-20 tokens conform to the ERC-20 standard by returning bool values for transfer and transferFrom functions. However, not all tokens strictly adhere to this standard. For example, tokens like Tether (USDT) and Binance-Pegged tokens do not return a bool and instead rely solely on transaction success or failure. This assumption can lead to compatibility issues, missed failures, or unexpected behavior.



```
function payToken(
        address tokenAddress,
        uint256 amount,
        string memory transactionId
    ) external {
        if (!allowedTokens[tokenAddress]) revert TokenNotAllowed(tokenAddress);

        IERC20 token = IERC20(tokenAddress);
  @>      if (!token.transferFrom(msg.sender, address(this), amount)) revert
TokenTransferFailed(tokenAddress);
  @>      if (!token.transfer(multiSigAddress, amount)) revert
TokenTransferFailed(tokenAddress);

        emit PaymentReceived(tokenAddress, amount, msg.sender, transactionId);
    }
```

- **Impact**:

1. Non-standard ERC-20 tokens will cause the function to fail, even when the transfer would otherwise succeed.
2. The contract could fail to interact with widely used tokens like USDT, limiting its usability in real-world scenarios.

- **Recommendation**

Use SafeERC20 from the OpenZeppelin library

**Status:** **Resolved**

**Issue:** [L-5] Insufficient check for multisig contract in setMultisigAddress function

**Description**:
The modifyDelegate function allows users to set or modify their delegate address. However, there is no check to prevent a user from setting the same delegate address that is already assigned to them. This creates unnecessary state changes and can result in redundant DelegateModified events being emitted, leading to higher gas costs and potential confusion.

**Impact of the vulnerability**

1. Unnecessary writes to the delegates mapping when the new delegate is the same as the current one.
2. Emitting the DelegateModified event for no actual modification can cause confusion during monitoring or debugging.

**Recommendation**
Add a check to ensure that the new delegate address is different from the current one before making any updates.

**Status:** **Resolved**

## INFORMATIONAL SEVERITY ISSUES

**Issue:** [I-1] Current delegate can be added again in modifyDelegate function

- **Description:**

The modifyDelegate function allows users to set or modify their delegate address. However, there is no check to prevent a user from setting the same delegate address that is already assigned to them. This creates unnecessary state changes and can result in redundant DelegateModified events being emitted, leading to higher gas costs and potential confusion.

- **Impact of the vulnerability**

1. Unnecessary writes to the delegates mapping when the new delegate is the same as the current one. 2. Emitting the DelegateModified event for no actual modification can cause confusion during monitoring or debugging.

- **Recommendation**

Add a check to ensure that the new delegate address is different from the current one before making any updates.

**Status:** **Resolved**

**Issue:** [I-2] No zero amount check in PayToken

**Description:**
The payToken function is used to transfer tokens to the contract and subsequently forward them to a multi-signature wallet. However, the function does not include a check to ensure that the amount being transferred is greater than zero. This can lead to unnecessary gas consumption and the emission of meaningless events when the amount is zero.

**Impact of the vulnerability**
1. Causes unnecessary gas usage and meaningless events when transferring zero tokens.

**Recommendation**
Ensure that the amount is greater than zero before proceeding with the token transfer.

**Status:** Resolved

- **Issue:** [I-3] PayEther emits address(0) instead of the standard native ether Address

- **Description:**

In the payEther function, the contract emits address(0) as the token address to represent native Ether payments. However, this approach deviates from the convention of using a standard placeholder, such as 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE, to represent native Ether in the context of smart contracts.

**Impact of the vulnerability**

1. External tools, wallets, or DApps may fail to recognize address(0) as a representation of Ether, leading to integration issues.

**Recommendation**

Use the standard placeholder address 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE to represent native Ether in the emitted event.

**Status:** **Resolved**

## Closing Summary

There were discoveries of some high, medium, low and informational issues after the audit. The audit team thereafter suggested some remediation to help remedy the issues found in the contract.

## Appendix

- Audit: The review and testing of contracts to find bugs.
- Issues: These are possible bugs that could lead exploits or help redefine the contracts better.
- Slither: A tool used to automatically find bugs in a contract.
- Severity: This explains the status of a bug.

## Guild Audits

Guild Audits is geared towards providing blockchain and smart contract security in the web3 world. The firm is passionate about remedying the constant hacks and exploits that deters the web3 motive.