verichains

SECURITY AUDIT OF

KULADAO SMART CONTRACTS

KULA

Public Report

Mar 18, 2024

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Mar 18, 2024. We would like to thank the KulaDAO for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the KulaDAO Smart Contracts. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About KulaDAO Smart Contracts

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the KulaDAO Smart Contracts. It was conducted on commit `bd7bba05f0b88f8950685737f0e443da94ec1fcb` from git repository link: *https://github.com/KulaDao/kula-dao-contracts*

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

KulaDAO acknowledges that the security services provided by Verichains, are conducted to the best of their professional abilities but cannot guarantee 100% coverage of all security vulnerabilities. KulaDAO understands and accepts that despite rigorous auditing, certain vulnerabilities may remain undetected. Therefore, KulaDAO agrees that Verichains shall not be held responsible or liable, and shall not be charged for any hacking incidents that occur due to security vulnerabilities not identified during the audit process.

1.5. Acceptance Minute

This final report served by Verichains to the KulaDAO will be considered an Acceptance Minute. Within 7 days, if no any further responses or reports is received from the KulaDAO, the final report will be considered fully accepted by the KulaDAO without the signature.

2. AUDIT RESULT

2.1. Overview

The KulaDAO Smart Contracts was written in `Solidity` language, with the required version to be `^0.8.8`. The source code was written based on OpenZeppelin's library.

2.1.1. KulaDaoToken.sol, RegionalDaoToken.sol

These two token contracts extend the `ERC20Votes`, `Pausable`, and `Ownable` contracts. With `Ownable`, by default, the contract owner is the contract deployer, but he can transfer ownership to another address at any time.

The `KulaDaoToken` is `ERC20` implementation that has some properties (as of the report writing time):

PROPERTY	VALUE
Name	UNKNOWN
Symbol	UNKNOWN
Decimals	18
Total Supply	1,000,000 (x10^{18}) Note: the number of decimals is 18, so the total representation token will be 1,000,000 or 1 million.

Table 2. The KulaDaoToken properties

For the ERC20 token, the security audit team has the list of centralization issues below:

Checklist	Status	Passed
Upgradeable	No	Yes
Fee modifiable	No	Yes
Mintable	No	Yes
Burnable	No	Yes
Pausable	No	Yes
Trading cooldown	No	Yes
Has blacklist	No	Yes
Has whitelist	No	Yes

Table 3. The decentralization checklist

The `RegionalDaoToken` is `ERC20` implementation that has some properties (as of the report writing time):

PROPERTY	VALUE
Name	RegionalDaoToken
Symbol	RDT
Decimals	18
Total Supply	1,000,000 (x10^{18}) Note: the number of decimals is 18, so the total representation token will be 1,000,000 or 1 million.

Table 4. The RegionalDaoToken properties

For the ERC20 token, the security audit team has the list of centralization issues below:

Checklist	Status	Passed
Upgradeable	No	Yes
Fee modifiable	No	Yes
Mintable	No	Yes
Burnable	No	Yes
Pausable	No	Yes
Trading cooldown	No	Yes
Has blacklist	No	Yes
Has whitelist	No	Yes

Table 5. The decentralization checklist

2.1.2. KulaDaoGovernor.sol, RegionalDaoGovernor.sol

These two contracts both extend the following contracts: `Governor`, `GovernorSettings`, `GovernorCountingSimple`, `GovernorVotes`, `GovernorVotesQuorumFraction`, `GovernorTimelockControl`, `ReentrancyGuard`, and `Ownable`. With `Ownable`, by default, the contract owner is the contract deployer, but he can transfer ownership to another address at any time.

- `Governor`: The core contract that contains all the logic and primitives. It is abstract and requires choosing one of each of the modules below, or custom ones.
- `GovernorSettings`: Manages some of the settings (voting delay, voting period duration, and proposal threshold) in a way that can be updated through a governance proposal, without requiring an upgrade.
- `GovernorCountingSimple`: Simple voting mechanism with 3 voting options: Against, For and Abstain.
- `GovernorVotes`: Extracts voting weight from an `ERC20Votes`.
- `GovernorVotesQuorumFraction`: Combines with GovernorVotes to set the quorum as a fraction of the total token supply.
- `GovernorTimelockControl`: Connects with an instance of `TimelockController`. Allows multiple proposers and executors, in addition to the Governor itself.

2.1.3. TimeLock.sol

The `TimeLock` contract extends `TimelockController` contract. In a governance system, the `TimelockController` contract is in charge of introducing a delay between a proposal and its execution. It can be used with or without a Governor.

2.1.4. Registry.sol

The Registry contract extends `Ownable` contract. With `Ownable`, by default, the contract owner is the contract deployer, but he can transfer ownership to another address at any time. The contract owner has the ability to add regional DAOs and DAO actions.

2.1.5. MultiSigWallet.sol

The proposal, once it reaches the `Succeeded` state, can be added to the queue and submitted to the multisig wallet for confirmation by the owners. After the proposal has received a sufficient number of confirmations, it can be executed by an owner

2.1.6. BuyBack.sol

The `BuyBack` contract support owners to buy back the tokens from the market and burn them.

2.2. Findings

During the audit process, the audit team found some vulnerabilities in the given version of KulaDAO Smart Contracts. The latest commit of the code at the time of the audit is `26a1b968cfefcac7b5c3d2e2bc03343adda605c3`.

Severity	Name	Status
CRITITCAL	Missing restrictions on the `claimTokens()` function	FIXED
CRITITCAL	Single owner exploits confirmation bypass by arbitrarily adding owners	ACK
HIGH	Precision loss error in `calculateBuybackAmount` function	FIXED
HIGH	Creating a new proposal fails if the value is greater than zero	FIXED
MEDIUM	Missing implementation of pause function `KulaDaoGovernor`, `RegionalDaoGovernor`	FIXED
MEDIUM	The multisig wallet does not have any impact	ACK

Severity	Name	Status
LOW	Tracking total votes variable can be manipulated	ACK
LOW	Redundancy of the `_timelock` variable	FIXED
INFO	The redundancy of the `Pausable` and `Ownable` contracts	FIXED
INFO	Hardcoding in the constants	FIXED
INFO	Missing event in the `addOwner()` function	ACK
INFO	Consider implementing a function to remove an owner	ACK
INFO	Using `immutable` state variable	ACK

2.2.1. [CRITITCAL] Missing restrictions on the `claimTokens()` function

Positions:

- `KulaDaoToken.sol`#L38
- `RegionalDaoToken.sol`#L39
- `RegionalDaoToken.sol`#L47

Description:

The `claimTokens()` and `claimTokens(amount)` functions allow users to claim any amount of tokens at any time. As both functions have no restrictions, malicious users can potentially claim the entire token balance within the contract.

```
// Function to allow users to claim a predefined amount of tokens.
function claimTokens() external {
    require(!isTokenClaimed[msg.sender], "Tokens already claimed");
    _transfer(address(this), msg.sender, TOKEN_CLAIM_AMOUNT);
    isTokenClaimed[msg.sender] = true;
    holders.push(msg.sender);
}

// Function to allow users to claim a specified amount of tokens.
function claimTokens(uint256 amount) external {
    require(!isTokenClaimed[msg.sender], "Tokens already claimed");
    _transfer(address(this), msg.sender, amount);
    isTokenClaimed[msg.sender] = true;
    holders.push(msg.sender);
}
```

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.2.2. [CRITITCAL] Single owner exploits confirmation bypass by arbitrarily adding owners

Positions:

- `MultiSigWallet.sol`#L189

Description:

An owner can add any address to become an owner of the multisig wallet. If that added address is a malicious owner, it can potentially bypass the `numConfirmationsRequired` condition when calling the `executeTransaction()` function.

```
function addOwner(address _owner) public onlyOwner {
    require(_owner != address(0), "invalid owner");
    require(!isOwner[_owner], "owner not unique");

    isOwner[_owner] = true;
    owners.push(_owner);
}
```

RECOMMENDATION

When a new owner is added, it should initially be a pending owner. It can only become a new owner after receiving sufficient confirmations from the existing owners.

UPDATES

Mar 18, 2024: The KulaDAO team has acknowledged this issue.

2.2.3. [HIGH] Precision loss error in `calculateBuybackAmount` function

Description: In the `calculateBuybackAmount` function, `stableCoinPerToken` is calculated by dividing `stableCoinBalance` by `totalSupply` . The result of this division is then multiplied by amount to calculate the buyback amount. However, the result of the division may have a precision loss error, which can lead to incorrect buyback amounts.

```
function calculateBuybackAmount(uint256 daoTokenAmount) public view returns (uint256) {
    uint256 stableCoinPerToken = stableCoin.balanceOf(treasuryAddress) / totalSupply;
    uint256 minimumBuybackAmount = 1e18; // Define a minimum threshold to ensure meaningful
buyback amounts
    uint256 calculatedAmount = (daoTokenAmount * stableCoinPerToken) / 1e18; // Adjust
based on decimals
    return calculatedAmount > minimumBuybackAmount ? calculatedAmount :
```

```
minimumBuybackAmount;
}
```

RECOMMENDATION

To avoid precision loss errors, we recommend multiplying `daoTokenAmount` by `stableCoin.balanceOf(treasuryAddress)` and then dividing by `totalSupply`. This will ensure that the result is accurate and does not have any precision loss errors.

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.2.4. [HIGH] Creating a new proposal fails if the value is greater than zero

Positions:

- `MultiSigWallet.sol`#L150

Description:

When the `executeTransaction()` function is called with a non-zero `values` parameter, the proposal cannot be executed because the call to the `execute()` function in the `regionalDAOGovernor` contract will fail due to not passing native tokens.

```
function executeTransaction(
    uint256 _txIndex
) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) {
    Transaction storage transaction = transactions[_txIndex];

    require(
        transaction.numConfirmations >= numConfirmationsRequired,
        "cannot execute tx"
    );

    transaction.executed = true;

    IGovernor regionalDAOGovernor = IGovernor(transaction.to);
    regionalDAOGovernor.execute( // <- @AUDIT - missing msg.value
        transaction.targets,
        transaction.values, // <- @AUDIT - values > 0 => fail
        transaction.calldatas,
        transaction.descriptionHash
    );

    emit ExecuteTransaction(msg.sender, _txIndex);
}
```

RECOMMENDATION

Pass msg.value when calling the regionalDAOGovernor.execute() function.

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.2.5. [MEDIUM] Missing implementation of pause function KulaDaoGovernor, RegionalDaoGovernor

Description: In KulaDaoGovernor and RegionalDaoGovernor contracts, there is no function to trigger the pause state. The pause function is a critical function that allows the contract owner to pause the contract in case of an emergency.

RECOMMENDATION

Implementing the pause function in the KulaDaoGovernor and RegionalDaoGovernor contracts.

2.2.6. [MEDIUM] The multisig wallet does not have any impact

Positions:

- MultiSigWallet.sol

Description:

When proposals reach the Succeeded state, the owners of the multisig wallet confirm those proposals. Once a proposal has enough confirmations, it can be executed by any owner through the executeTransaction() function. The executeTransaction() function calls the execute() function inside the regionalDAOGovernor contract. However, the execute() function can be called by anyone without requiring confirmations from the owners of the multisig wallet.

```
// MultiSigWallet.sol
function executeTransaction(
    uint256 _txIndex
) public onlyOwner txExists(_txIndex) notExecuted(_txIndex) {
    Transaction storage transaction = transactions[_txIndex];

    require(
        transaction.numConfirmations >= numConfirmationsRequired,
        "cannot execute tx"
    );

    transaction.executed = true;

    IGovernor regionalDAOGovernor = IGovernor(transaction.to);
    regionalDAOGovernor.execute(
        transaction.targets,
        transaction.values,
        transaction.calldatas,
        transaction.descriptionHash
    );

    emit ExecuteTransaction(msg.sender, _txIndex);
}

// Governor.sol
function execute(
    address[] memory targets,
    uint256[] memory values,
    bytes[] memory calldatas,
    bytes32 descriptionHash
) public payable virtual override returns (uint256) {
    uint256 proposalId = hashProposal(targets, values, calldatas, descriptionHash);

    ProposalState currentState = state(proposalId);
    require(
        currentState == ProposalState.Succeeded || currentState == ProposalState.Queued,
        "Governor: proposal not successful"
    );
    _proposals[proposalId].executed = true;

    emit ProposalExecuted(proposalId);

    _beforeExecute(proposalId, targets, values, calldatas, descriptionHash);
    _execute(proposalId, targets, values, calldatas, descriptionHash);
    _afterExecute(proposalId, targets, values, calldatas, descriptionHash);

    return proposalId;
}

// GovernorTimelockControl.sol
    function _execute(
        ...
    ) internal virtual override {
        _timelock.executeBatch{value: msg.value}(targets, values, calldatas, 0,
descriptionHash); // @AUDIT - msg.sender call to timelock will be DAO
    }
```

UPDATES

Mar 18, 2024: The KulaDAO team acknowledged this issue.

2.2.7. [LOW] Tracking total votes variable can be manipulated

Positions:

- KulaDaoGovernor.sol
- RegionalDaoGovernor.sol

Description:

The tracking variables, such as _usersTotalVoteInDao and _proposalTotalVote, increase by 1 each time users call the functions castVoteWithReason() and castVoteWithReasonAndParams(). However, these functions do not check whether the weight of the voting user is greater than 0, allowing anyone to arbitrarily increase these two variables.

```
//KulaDaoGovernor.sol
function castVoteWithReason(
    uint256 proposalId,
    uint8 support,
    string calldata reason
) public override(Governor, IGovernor) returns (uint256) {
    _usersTotalVoteInDao[msg.sender] += 1;
    if (support > 0) {
        _proposalTotalVote[proposalId].yes += 1;
    } else {
        _proposalTotalVote[proposalId].no += 1;
    }
    return super.castVoteWithReason(proposalId, support, reason);
}
//Governor.sol
function castVoteWithReason(
    uint256 proposalId,
    uint8 support,
    string calldata reason
) public virtual override returns (uint256) {
    address voter = _msgSender();
    return _castVote(proposalId, voter, support, reason);
}

function _castVote(
    uint256 proposalId,
    address account,
    uint8 support,
    string memory reason,
    bytes memory params
) internal virtual returns (uint256) {
    ProposalCore storage proposal = _proposals[proposalId];
    require(state(proposalId) == ProposalState.Active, "Governor: vote not currently
active");

    uint256 weight = _getVotes(account, proposal.voteStart, params); // <- @AUDIT - weight
can be equal to 0
    _countVote(proposalId, account, support, weight, params);

    if (params.length == 0) {
        emit VoteCast(account, proposalId, support, weight, reason);
    } else {
        emit VoteCastWithParams(account, proposalId, support, weight, reason, params);
    }

    return weight;
}
```

UPDATES

Mar 18, 2024: The KulaDAO team has acknowledged this issue.

2.2.8. [LOW] Redundancy of the `_timelock` variable

Positions:

- `KulaDaoGovernor.sol`#L57
- `RegionalDaoGovernor.sol`#L57

Description:

The variable `_timelock` has already been declared in the abstract contract `GovernorTimelockControl`, and it can change its value in the future. Therefore, declaring the variable `_timelock` again in the `KulaDaoGovernor` and `RegionalDaoGovernor` contracts is unnecessary and can lead to confusion.

```
constructor(
    IVotes _token,
    TimelockController timelock,
    ...
)
    ...
    GovernorTimelockControl(timelock)
{
    multiSigWallet = payable(_multiSigWallet);
    proposalCount = 0;
    _timelock = timelock; // <- @AUDIT - duplicate
}
```

RECOMMENDATION

Remove the `_timelock` variable from both the `KulaDaoGovernor` and `RegionalDaoGovernor` contracts.

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.3. Additional notes and recommendations

2.3.1. [INFO] The redundancy of the Pausable and Ownable contracts

Positions:

- KulaDaoToken.sol
- RegionalDaoToken.sol

Description: Both of these token contracts extend the Pausable and Ownable contracts, but no logic from these two contracts is utilized.

RECOMMENDATION

Utilize the logics within the two contracts or remove them if unnecessary for clearer readability.

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.3.2. [INFO] Hardcoding in the constants

Positions:

- KulaDaoToken.sol#L17
- KulaDaoToken.sol#L22
- RegionalDaoToken.sol#L18
- RegionalDaoToken.sol#L23

Description:

The constants are defined with a very long sequence of zeros at the end, making it difficult to read:

```
// Maximum token supply constant.
    uint256 constant MAX_SUPPLY = 1000000000000000000000000; // 1 million tokens
    // Constant amount for token claims.
    uint256 constant TOKEN_CLAIM_AMOUNT = 50000000000000000000000; // 50,000 tokens
```

RECOMMENDATION

To make the smart contract clearer, instead of writing long sequences of zeros, use the token's decimal:

```
// Maximum token supply constant.
    uint256 constant MAX_SUPPLY = 1_000_000 * 10**18; // 1 million tokens
    // Constant amount for token claims.
    uint256 constant TOKEN_CLAIM_AMOUNT = 50_000 * 10**18; // 50,000 tokens
```

UPDATES

Mar 18, 2024: The KulaDAO team has fixed this issue.

2.3.3. [INFO] Missing event in the `addOwner()` function

Positions:

- `MultiSigWallet.sol`#L189

Description:

Events should be added and emitted in state-changing functions. The `addOwner()` function modifies the list of owners but does not emit any events. An event should be added for the `addOwner()` function.

UPDATES

Mar 18, 2024: The KulaDAO team has acknowledged this issue.

2.3.4. [INFO] Consider implementing a function to remove an owner

Positions:

- `MultiSigWallet.sol`

Description:

Consider defining a function to remove an owner, allowing for the removal of a potentially malicious or compromised owner. The function should require enough confirmations before removing the owner from the list.

UPDATES

Mar 18, 2024: The KulaDAO team has acknowledged this issue.

2.3.5. [INFO] Using `immutable` state variable

Positions:

- `MultiSigWallet.sol`#L28

Description:

The variable `numConfirmationsRequired` cannot change its value, so it should be declared with the `immutable` keyword.

UPDATES

Mar 18, 2024: The KulaDAO team has acknowledged this issue.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Mar 18, 2024*	Public Report	Verichains Lab

Table 6. Report versions history