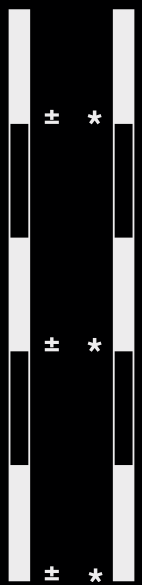
VERILOG

2024

# DODO STAKE CONTRACT: SECURITY REVIEW REPORT



Apr 12, 2024

# Contents

# 1 | Introduction


**Figure 1.1:** DODOchain Report Cover

This report presents our engineering engagement with the DODO team on their staking contract.

| **Project Name** | **DODOchain** |
|---|---|
| Repository Link | https://github.com/DODOchain/stake-contract |
| First Commit Hash | First: f954b24; |
| Final Commit Hash | Final: a0f5051; |
| Language | Solidity |
| Chain | Arbitrum Merlin |

## 2 | About Verilog Solutions

Founded by a group of cryptography researchers and smart contract engineers in North America, Verilog Solutions elevates the security standards for Web3 ecosystems by being a full-stack Web3 security firm covering smart contract security, consensus security, and operational security for Web3 projects.
Verilog Solutions team works closely with major ecosystems and Web3 projects and applies a quality-above-quantity approach with a continuous security model. Verilog Solutions onboards the best and most innovative projects and provides the best-in-class advisory services on security needs, including on-chain and off-chain components.

# 3 | Service Scope

## 3.1 | Service Stages

Our auditing service includes the following two stages:

- Smart Contract Auditing Service

### 3.1.1 | Smart Contract Auditing Service

The Verilog Solutions team analyzed the entire project using a detailed-oriented approach to capture the fundamental logic and suggested improvements to the existing code. Details can be found under Findings And Improvement Suggestions.

## 3.2 | Methodology

- **Code Assessment**
  - We evaluate the overall quality of the code and comments as well as the architecture of the repository.
  - We help the project dev team improve the overall quality of the repository by providing suggestions on refactorization to follow the best practices of Web3 software engineering.
- **Code Logic Analysis**
  - We dive into the data structures and algorithms in the repository and provide suggestions to improve the data structures and algorithms for the lower time and space complexities.
  - We analyze the hierarchy among multiple modules and the relations among the source code files in the repository and provide suggestions to improve the code architecture with better readability, reusability, and extensibility.
- **Business Logic Analysis**
  - We study the technical whitepaper and other documents of the project and compare its specifications with the functionality implemented in the code for any potential mismatch between them.
  - We analyze the risks and potential vulnerabilities in the business logic and make suggestions to improve the robustness of the project.
- **Access Control Analysis**
  - We perform a comprehensive assessment of the special roles of the project, including their authorities and privileges.
  - We provide suggestions regarding the best practice of privilege role management according to the standard operating procedures (SOP).
- **Off-Chain Components Analysis**
  - We analyze the off-chain modules that are interacting with the on-chain functionalities and provide suggestions according to the SOP.
  - We conduct a comprehensive investigation for potential risks and hacks that may happen on the off-chain components and provide suggestions for patches.

## 3.3 | Audit Scope

Our auditing for the DODOchain covered the Solidity smart contract 'TimeLockContract' in the following directory (https://github.com/DODOchain/stake-contract/contracts/stake1/TimeLockContract.sol) with commit hash **f954b24**.

# 4 | Findings and Improvement Suggestions

| Severity | Total | Acknowledged | Resolved |
|---|---|---|---|
| High | 0 | 0 | 0 |
| Medium | 3 | 3 | 3 |
| Low | 2 | 2 | 2 |
| Informational | 1 | 1 | 1 |

## 4.1 | Medium

### 4.1.1 | `<address>.call()` should be used instead of `<address>.transfer()` to transfer native tokens

| Severity | Medium |
|---|---|
| Source | contracts/stake1/TimeLockContract.sol#L50; contracts/stake1/TimeLockContract.sol#L54; |
| Commit | f954b24; |
| Status | Resolved in commit 341bc3c; |

- **Description**
  The current smart contract uses the transfer() function to transfer the native tokens:

  ```
  payable(beneficiary).transfer(amount);
  ```

  However, using this method has some limitations like the gas limit of 2,300.

- **Exploit Scenario**

  - Users stake ETH in the `TimeLockContract`;
  - After the `releaseTime` passes, the `beneficiary` calls the `withdrawETH()` function to withdraw the native token staked;
  - The transaction reverts because of the gas limit of the `transfer()` function.

- **Recommendations**
  Use the `call` method to transfer ETH and check the return value:

  ```
  (bool success, ) = beneficiary.call{value: amount}('');
  require(success, 'send value failed');
  ```

- **Results**
  Resolved in commit 341bc3c.

  The suggestion was implemented.

### 4.1.2 | Unchecked return values

| Severity | Medium |
|---|---|
| Source | contracts/stake1/TimeLockContract.sol.sol#L42;<br>contracts/stake1/TimeLockContract.sol.sol#L67; |
| Commit | f954b24; |
| Status | Resolved in commit a68a568; |

- **Description**
  The return value of an external `transfer`/`transferFrom` call is not checked. Several tokens do not revert in case of failure and return false, hence some transactions will actually fail but the transactions will still execute.
- **Exploit Scenario**
  - Users try to stake ERC20 tokens using the `lockERC20()` function;
  - The return value of the `transferFrom()` function is false but since the `transferFrom()` function doesn't check it, the transaction goes through;
  - Users think they have staked but in reality, nothing has changed.
- **Recommendations**
  Use `SafeERC20`, or ensure that the transfer/transferFrom return value is checked.
- **Results**
  Resolved in commit 341bc3c.

  The `SafeERC20` library is now considered.

### 4.1.3 | Lack of a two-step process for contract ownership

| Severity | Medium |
|---|---|
| Source | contracts/stake1/TimeLockContract.sol.sol#L8; |
| Commit | e460e58; |
| Status | Resolved in commit a0f5051; |

- **Description**
  The current smart contract uses the `Ownable` library from Open Zeppelin. This library is known to not require a two-step process when changing ownership of the contract.
- **Exploit Scenario**
  - The `owner` tries to transfer ownership to a new address;
  - There is a mistake when inputting the address;
  - Ownership of the contract is lost.
- **Recommendations**
  Use the `Ownable2Step` abstract contract from Open Zeppelin.
- **Results**
  Resolved in commit a0f5051.

  The `Ownable2Step` abstract contract is now considered.

## 4.2 | Low

### 4.2.1 | Duplicate interface

| Severity | Low |
|---|---|
| Source | contracts/stake1/TimeLockContract.sol#L4-8; |
| Commit | f954b24; |
| Status | Resolved in commit 341bc3c; |

- **Description**
  OpenZeppelin is imported as a dependency in the `package.json` file, this means that the `IERC20` interface can already be found in the following directory `@openzeppelin/contracts/token/ERC20/IERC20.sol`.

  Therefore there's no need to define the interface in the `TimeLockContract` contract.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Remove the IERC20 interface and import it directly from OpenZeppelin.

- **Results**
  Resolved in commit 341bc3c.

  The suggestion was implemented.

### 4.2.2 | Lack of zero address checks

| Severity | Low |
|---|---|
| Source | contracts/stake1/TimeLockContract.sol#L24; |
| Commit | f954b24; |
| Status | Resolved in commit 341bc3c; |

- **Description**
  The `beneficiary` address is the only address that can unlock tokens and this address can only be defined on deployment. Therefore, it is crucial to perform a zero address check on the `constructor()`.

- **Exploit Scenario**
  N/A.

- **Recommendations**
  Add the following check to the `constructor()`:

```
require(beneficiary != address(0))
```

- **Results**
  Resolved in commit 341bc3c.

  The suggestion was implemented.

## 4.3 | Informational

### 4.3.1 | Variables can be defined as immutable

| Severity | Informational |
| --- | --- |
| Source | contracts/stake1/TimeLockContract.sol#L11-12; |
| Commit | e460e58; |
| Status | Resolved in commit a0f5051; |

- **Description**
  The `beneficiary` and the `_releaseTime` variables are only defined in the `constructor()`. Therefore they can be set as immutable to save some gas.
- **Exploit Scenario**
  N/A.
- **Recommendations**
  Set the `beneficiary` and `_releaseTime` variables as immutable.
- **Results**
  Resolved in commit a0f5051.

  The suggestion was implemented.

# 5 | Appendix

## 5.1 | Appendix I: Severity Categories

| Severity | Description |
|---|---|
| High | Issues that are highly exploitable security vulnerabilities. It may cause direct loss of funds / permanent freezing of funds. All high severity issues should be resolved. |
| Medium | Issues that are only exploitable under some conditions or with some privileged access to the system. Users' yields/rewards/information is at risk. All medium severity issues should be resolved unless there is a clear reason not to. |
| Low | Issues that are low risk. Not fixing those issues will not result in the failure of the system. A fix on low severity issues is recommended but subject to the clients' decisions. |
| Information | Issues that pose no risk to the system and are related to the security best practices. Not fixing those issues will not result in the failure of the system. A fix on informational issues or adoption of those security best practices-related suggestions is recommended but subject to clients' decision. |

## 5.2 | Appendix II: Status Categories

| Severity | Description |
|---|---|
| Unresolved | The issue is not acknowledged and not resolved. |
| Partially Resolved | The issue has been partially resolved |
| Acknowledged | The Finding / Suggestion is acknowledged but not fixed / not implemented. |
| Resolved | The issue has been sufficiently resolved |

# 6 | Disclaimer

Verilog Solutions receives compensation from one or more clients for performing the smart contract and auditing analysis contained in these reports. The report created is solely for Clients and published with their consent. As such, the scope of our audit is limited to a review of code, and only the code we note as being within the scope of our audit is detailed in this report. It is important to note that the Solidity code itself presents unique and unquantifiable risks since the Solidity language itself remains under current development and is subject to unknown risks and flaws. Our sole goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies. Thus, Verilog Solutions in no way claims any guarantee of security or functionality of the technology we agree to analyze.

In addition, Verilog Solutions reports do not provide any indication of the technology's proprietors, business, business model, or legal compliance. As such, reports do not provide investment advice and should not be used to make decisions about investment or involvement with any particular project. Verilog Solutions has the right to distribute the Report through other means, including via Verilog Solutions publications and other distributions. Verilog Solutions makes the reports available to parties other than the Clients (i.e., "third parties") – on its website in hopes that it can help the blockchain ecosystem develop technical best practices in this rapidly evolving area of innovation.