Cyberscope

Audit Report

Dogita Staking

September 2024

Networks ETH, BSC

Addresses 0xC633174D8B937927Bfd1650EEa928d509b368554,

0xC633174D8B937927Bfd1650EEa928d509b368554

Audited by © cyberscope

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Explorer	https://etherscan.io/address/0xc633174d8b937927bfd1650eea928d509b368554 https://bscscan.com/address/0xc633174d8b937927bfd1650eea928d509b368554

Audit Updates

Initial Audit	03 Sep 2024

Source Files

Filename	SHA256
Dogita_Staking.sol	8d29327fed71402d32d65cfd3c30c25419af0c2030abd26eb7b00ace5b02a1ad

Overview

The Dogita_Staking contract is deployed in the ETH and BSC networks and implements a staking mechanism.

Functionality:

- Allows users to stake tokens in return for rewards.
- Manages the deposit and withdrawal processes.
- Defines multiple staking pools and their respective characteristics.
- Calculates and distributes staking rewards based on predefined rules.

Key Functions:

- deposit(uint256 _amount, uint256 _lockupDuration)

  Allows users to stake a specified amount of tokens for a selected lockup period.

- withdraw(uint256 orderId)

  Allows users to withdraw the staked amounts and the staking rewards after the lockup period.

- emergencyWithdraw(uint256 orderId)

  Allows users to withdraw the staked amounts and the staking rewards before the lockup period for a high taxation fee.

- claimRewards(uint256 orderId)

  Allows users to claim their earned rewards at any time during or after the lockup period.

- withdrawERC20(address _tokenAddress)

  Allows the owner to transfer any ERC20 token accumulated in the staking contract, including the token used for the staking process.

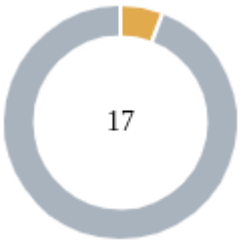
Roles:

- Owner:

  The contract's owner can interact with the following functions:

  1. function renounceOwnership() external onlyOwner { }
  2. function transferOwnership(address newOwner) external onlyOwner { }
  3. function setPlansApy( ) external onlyOwner { }
  4. function setEmergencyWithdrawFees( ) external onlyOwner { }
  5. function setWithdrawFees( ) external onlyOwner { }
  6. function setMinStake( ) external onlyOwner { }
  7. function setMaxStake( ) external onlyOwner { }
  8. function setEmergencyFeeReceiver( ) external onlyOwner { }
  9. function setFeeReceiver( ) external onlyOwner { }
  10. function toggleStaking( ) external onlyOwner { }
  11. function withdrawERC20( ) external onlyOwner { }
  12. function setClaimInterval() external onlyOwner { }

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	WAI	Withdrawn Amount Inconsistency	Unresolved
Minor / Informative	CO	Code Optimization	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	ISI	Inconsistent Sender Identification	Unresolved
Minor / Informative	MEM	Misleading Error Messages	Unresolved
Minor / Informative	MVN	Misleading Variables Naming	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	OCTD	Transfers Contract's Tokens	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	TSI	Tokens Sufficiency Insurance	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L07	Missing Events Arithmetic	Unresolved

●	L09	Dead Code Elimination	Unresolved
●	L13	Divide before Multiply Operation	Unresolved
●	L16	Validate Variable Setters	Unresolved

WAI - Withdrawn Amount Inconsistency

Criticality	Medium
Location	Dogita_Staking.sol#L236
Status	Unresolved

Description

The contract's withdrawal mechanism calculates an unstaking fee, which is deducted from the withdrawn amount. The remaining balance is sent to the staker's address, and the fee is intended to be sent to the `withdrawFeeReceiver`. However, the contract incorrectly charges the staker twice in the withdrawal process. Initially, it calculates the unstaking fee and deducts it from the withdrawn amount, sending the remaining balance to the staker. Then, it requires the staker to pay the fee again from their own balance while withholding the deducted amount. This effectively results in the staker paying the fees twice.

```
function withdraw(uint256 orderId) external nonReentrant {
    ...
    uint256 fees = (orderInfo.amount * withdrawFees) / 100;
    uint256 depAmount = total - fees;
    ...
    require(token.transfer(address(_msgSender()), depAmount),
            "TokenStaking: token transfer via withdraw not
succeeded");
    require(token.transferFrom(_msgSender(),
withdrawFeeReceiver, fees),
            "TokenStaking: token transferFrom via deposit not
succeeded");
    ...
    }
```

Recommendation

The team is advised to review and revise the implementation of the `withdraw` function to ensure proper handling of the withdrawn amounts.

CO - Code Optimization

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L216
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations. In particular the state of `hasStaked[msg.sender][_lockupDuration]` could be updated within the previous conditional expression to prevent redundant write operations when the variable is already `true` .

```
if (!hasStaked[msg.sender][_lockupDuration]) {
    stakersPlan[_lockupDuration] =
stakersPlan[_lockupDuration] + 1;
    totalStakers = totalStakers + 1;
    }
    //updating staking status
    hasStaked[msg.sender][_lockupDuration] = true;
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L376,386,390,394,398,402,406,410,424,433
Status	Unresolved

Description

The contract's functionality and behaviour are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
function setPlansApy( ) external onlyOwner {...}
function setEmergencyWithdrawFees( ) external onlyOwner {...}
function setWithdrawFees( ) external onlyOwner {...}
function setMinStake( ) external onlyOwner {...}
function setMaxStake( ) external onlyOwner {...}
function setEmergencyFeeReceiver( ) external onlyOwner {...}
function setFeeReceiver( ) external onlyOwner {...}
function toggleStaking( ) external onlyOwner {...}
function withdrawERC20( ) external onlyOwner {...}
function setClaimInterval() external onlyOwner {...}
```

In particular the owner is able to modify the pool characteristics, to increase the withdrawal fees to values greater than 100%, to stop new deposits and to withdraw all the staked tokens from the pools.

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

ISI - Inconsistent Sender Identification

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L196,200,205,216,223,226,227,229,244,259,262,266,270,276,287,297,299,304,314,325,335,341,348,425
Status	Unresolved

Description

The contract contains instances of both `msg.sender` and `_msgSender()`. The latter is introduced to support meta-transactions. Meta-transactions are a unique type of blockchain transaction that delegate transaction submission, processing, and fee payment to a third party. In such cases, `msg.sender` and `_msgSender()` may return different values, leading to potential inconsistencies in the contract. This discrepancy could result in unintended behaviour or introduce future errors.

```
hasStaked[msg.sender][_lockupDuration] = true;
```

```
balanceOf[_msgSender()] += depAmount;
```

Recommendation

The team is advised to adopt a consistent method for sender identification throughout the contract. This will enhance both the consistency and readability of the contract, while reducing the risk of unintended behaviour or future errors, particularly in the context of meta-transactions.

MEM - Misleading Error Messages

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L269,307
Status	Unresolved

Description

The contract contains misleading error messages. Specifically, there are error messages with misleading descriptions that do not accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(token.transferFrom(_msgSender(), withdrawFeeReceiver,
fees),
        "TokenStaking: token transferFrom via deposit not
succeeded");
```

```
require(token.transfer(feeReceiver, fees),
        "TokenStaking: token transfer via claim rewards not
succeeded");
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

MVN - Misleading Variables Naming

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L120
Status	Unresolved

Description

Variables can have misleading names if their names do not accurately reflect the value they contain or the purpose they serve. The contract uses some variable names that are too generic or do not clearly convey the information stored in the variable. Misleading variable names can lead to confusion, making the code more difficult to read and understand.

```
uint256 private constant _1Year = 431 days;
uint256 private constant _2Year = 631 days;
uint256 private constant _3Year = 731 days;
```

The use of these inaccurately represented time variables can result in a misleading APY calculation. As a result, the APY reported by the rewards pool may be lower than the nominal value indicated in the `PoolInfo` structure.

```
uint256 APY = (orderInfo.amount * orderInfo.returnPer) / 100;
uint256 reward = (APY * stakeTime) / _days365;
```

```
struct PoolInfo {
        uint256 lockupDuration;
        uint256 returnPer;
}
```

Recommendation

It's always a good practice for the contract to contain variable names that are specific and descriptive. The team is advised to keep in mind the readability of the code.

MC - Missing Check

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L390,394,398
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues. In particular the contract is not checking whether `minStake < maxStake` and if the fees exceed an upper limit.

```
function setWithdrawFees(uint256 _fee) external onlyOwner {
        withdrawFees = _fee;
}
```

```
function setMinStake(uint256 _value) external onlyOwner {
        minStake = _value;
}
```

```
function setMaxStake(uint256 _value) external onlyOwner {
        maxStake = _value;
}
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L238,281,319,353
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(orderId <= latestOrderId,
"TokenStaking: INVALID orderId, orderId greater than
latestOrderId"
);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L424
Status	Unresolved

Description

The contract owner has the authority to claim all the balance of the contract. The owner may take advantage of it by calling the `withdrawERC20` function.

```
function withdrawERC20(address _tokenAddress) external
onlyOwner {
        IERC20 withdrawtoken = IERC20(_tokenAddress);
        uint256 balance =
withdrawtoken.balanceOf(address(this));
        require(
            withdrawtoken.transfer(msg.sender, balance),
            "Token transfer failed"
        );}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L177
Status	Unresolved

Description

The `transferFrom()` function is used to transfer a specified amount of tokens to the address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount if the token implements a fee mechanism. This may produce inconsistency between the expected and the actual behaviour.

```
function deposit(uint256 _amount, uint256 _lockupDuration)
external {


    uint256 fees = (_amount * withdrawFees) / 100;
    uint256 depAmount = _amount - fees;

    require(token.transferFrom(_msgSender(), address(this),
depAmount),
            "TokenStaking: token transferFrom via deposit not
succeeded");
    require(token.transferFrom(_msgSender(),
withdrawFeeReceiver, fees),
            "TokenStaking: token transferFrom via deposit not
succeeded");

    orders[++latestOrderId] = OrderInfo(
        _msgSender(),
        depAmount,
        pool.lockupDuration,
        pool.returnPer,
        block.timestamp,
        block.timestamp + pool.lockupDuration,
        0,
        false,
        block.timestamp
    );
```

In the current implementation, a `depAmount` is transferred from the user to the contract and the same amount is stored in the `orders[++latestOrderId]` mapping as the deposited amount. However, if the token implements a fee mechanism the transferred amount may be less than the expected amount. This divergence may yield inconsistencies in the staking contract and its mechanisms.

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected. It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call. `Actual Transferred Amount = Balance After Transfer - Balance Before Transfer`

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L236
Status	Unresolved

Description

The tokens are not held within the contract itself. Instead, the contract is designed to provide the tokens from an external administrator. While external administration can provide flexibility, it introduces a dependency on the administrator's actions, which can lead to various issues and centralization risks.

```
function withdraw(uint256 orderId) external nonReentrant {
    ...
    require(
        token.transfer(address(_msgSender()), depAmount),
        "TokenStaking: token transfer via withdraw not
succeeded"
    );
    ...
}
```

Recommendation

It is recommended to consider implementing a more decentralized and automated approach for handling the contract tokens. One possible solution is to hold the tokens within the contract itself. If the contract guarantees the process it can enhance its reliability, security, and participant trust, ultimately leading to a more successful and efficient process.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L122
Status	Unresolved

Description

State variables can be declared as constant using the constant keyword. This means that the value of the state variable cannot be changed after it has been set. Additionally, the constant variables decrease gas consumption of the corresponding transaction.

```
IERC20 public token =
IERC20(0x488542C2320F20D65405a1C03DA769Bc124F9A28)
```

Recommendation

Constant state variables can be useful when the contract wants to ensure that the value of a state variable cannot be changed by any function in the contract. This can be useful for storing values that are important to the contract's behavior, such as the contract's address or the maximum number of times a certain function can be called. The team is advised to add the constant keyword to state variables that never change.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L100,116,117,118,119,121,173,382,386,390,394,398,402,406,420
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
contract Dogita_Staking is Ownable, ReentrancyGuard {
    struct PoolInfo {
        uint256 lockupDuration;
        uint256 returnPer;
    }
    struct OrderInfo {
...
    }

    function setClaimInterval(uint256 newInterval) external
onlyOwner {
        require(newInterval > 0, "Claim interval must be
greater than 0");
        _claimInterval = newInterval;
    }
}

...
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation https://docs.soliditylang.org/en/stable/style-guide.html#naming-conventions.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L383,387,391,395,431
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
emergencyWithdrawFees = _fee
withdrawFees = _fee
minStake = _value
maxStake = _value
_claimInterval = newInterval
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L95
Status	Unresolved

Description

In Solidity, dead code is code that is written in the contract, but is never executed or reached during normal contract execution. Dead code can occur for a variety of reasons, such as:

- Conditional statements that are always false.
- Functions that are never called.
- Unreachable code (e.g., code that follows a return statement).

Dead code can make a contract more difficult to understand and maintain, and can also increase the size of the contract and the cost of deploying and interacting with it.

```
function _reentrancyGuardEntered() private view returns (bool)
{
        return _status == _ENTERED;
    }
```

Recommendation

To avoid creating dead code, it's important to carefully consider the logic and flow of the contract and to remove any code that is not needed or that is never executed. This can help improve the clarity and efficiency of the contract.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L356,357,362,363
Status	Unresolved

Description

It is important to be aware of the order of operations when performing arithmetic calculations. This is especially important when working with large numbers, as the order of operations can affect the final result of the calculation. Performing divisions before multiplications may cause loss of prediction.

```
uint256 APY = (orderInfo.amount * orderInfo.returnPer) / 100
uint256 reward = (APY * stakeTime) / _days365
```

Recommendation

To avoid this issue, it is recommended to carefully consider the order of operations when performing arithmetic calculations in Solidity. It's generally a good idea to use parentheses to specify the order of operations. The basic rule is that the multiplications should be prior to the divisions.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	Dogita_Staking.sol#L399,403
Status	Unresolved

Description

The contract performs operations on variables that have been configured on user-supplied input. These variables are missing of proper check for the case where a value is zero. This can lead to problems when the contract is executed, as certain actions may not be properly handled when the value is zero.

```
feeReceiver = _address
withdrawFeeReceiver = _address
```

Recommendation

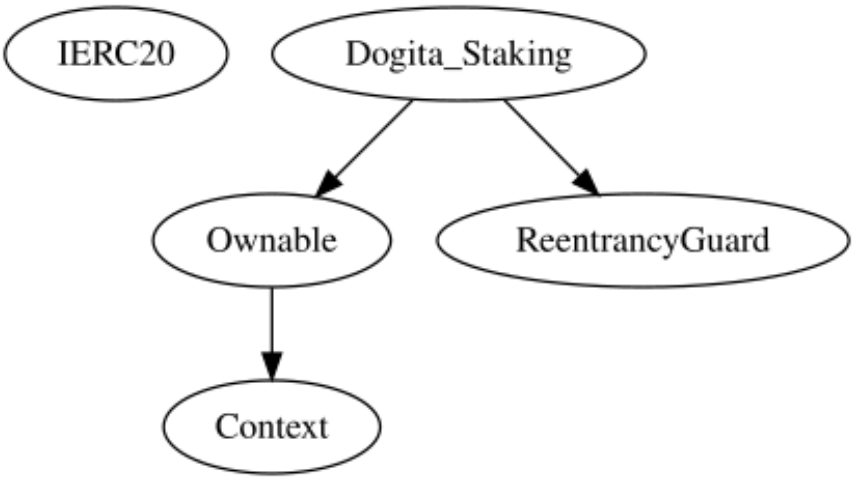
By adding the proper check, the contract will not allow the variables to be configured with zero value. This will ensure that the contract can handle all possible input values and avoid unexpected behavior or errors. Hence, it can help to prevent the contract from being exploited or operating unexpectedly.

Functions Analysis

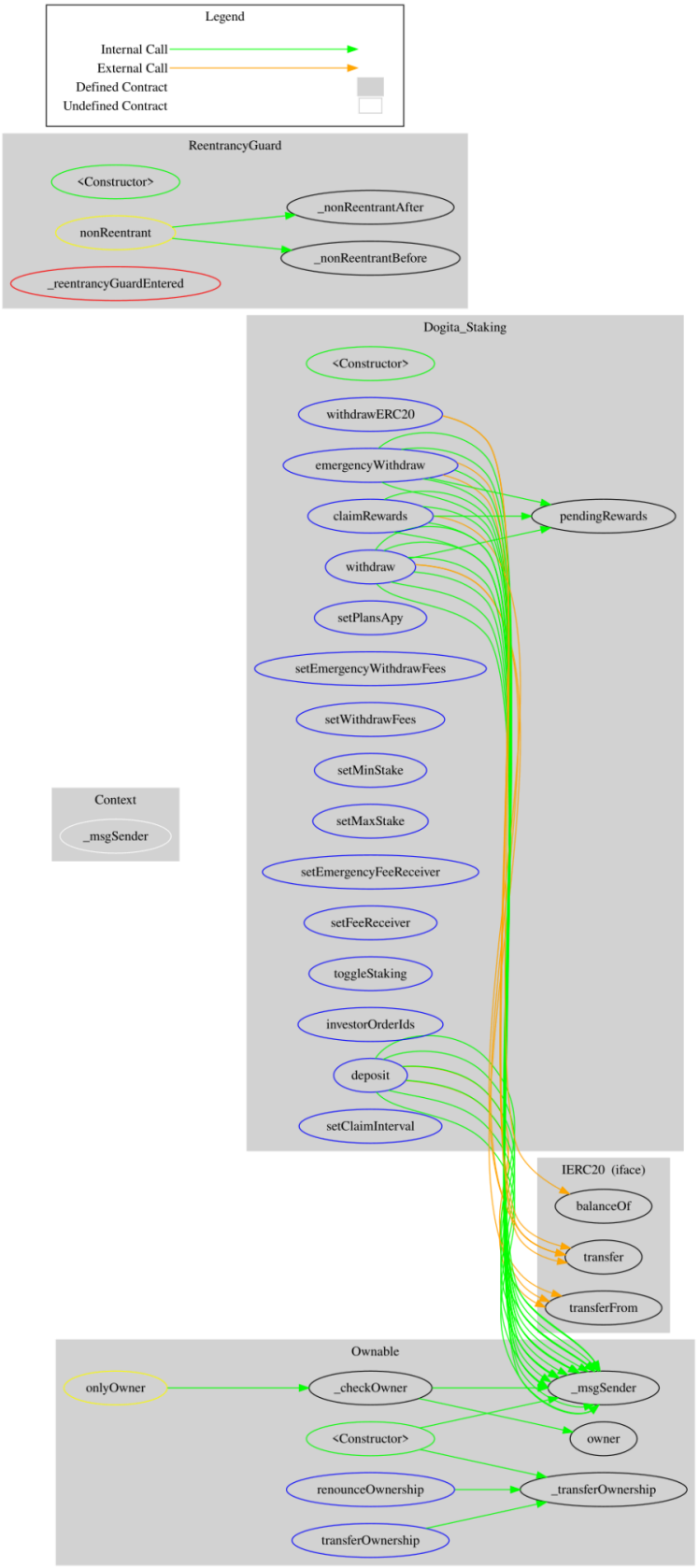
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
IERC20	Interface			
	balanceOf	External		-
	transfer	External	✓	-
	transferFrom	External	✓	-
Context	Implementation			
	_msgSender	Internal		
Ownable	Implementation	Context		
		Public	✓	-
	owner	Public		-
	_checkOwner	Private		
	renounceOwnership	External	✓	onlyOwner
	transferOwnership	External	✓	onlyOwner
	_transferOwnership	Private	✓	
ReentrancyGuard	Implementation			
		Public	✓	-
	_nonReentrantBefore	Private	✓	
	_nonReentrantAfter	Private	✓	

	_reentrancyGuardEntered	Private		
Dogita_Staking	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	deposit	External	✓	-
	withdraw	External	✓	nonReentrant
	emergencyWithdraw	External	✓	nonReentrant
	claimRewards	External	✓	nonReentrant
	pendingRewards	Public		-
	setPlansApy	External	✓	onlyOwner
	setEmergencyWithdrawFees	External	✓	onlyOwner
	setWithdrawFees	External	✓	onlyOwner
	setMinStake	External	✓	onlyOwner
	setMaxStake	External	✓	onlyOwner
	setEmergencyFeeReceiver	External	✓	onlyOwner
	setFeeReceiver	External	✓	onlyOwner
	toggleStaking	External	✓	onlyOwner
	investorOrderIds	External		-
	withdrawERC20	External	✓	onlyOwner
	setClaimInterval	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
ReentrancyGuard
<Constructor>
nonReentrant
_reentrancyGuardEntered
_nonReentrantAfter
_nonReentrantBefore
Dogita_Staking
<Constructor>
withdrawERC20
emergencyWithdraw
claimRewards
withdraw
setPlansApy
setEmergencyWithdrawFees
setWithdrawFees
setMinStake
setMaxStake
setEmergencyFeeReceiver
setFeeReceiver
toggleStaking
investorOrderIds
deposit
setClaimInterval
pendingRewards
Context
_msgSender
IERC20 (iface)
balanceOf
transfer
transferFrom
Ownable
onlyOwner
_checkOwner
<Constructor>
renounceOwnership
transferOwnership
_msgSender
owner
_transferOwnership

Summary

The audited contract implements a staking mechanism. This audit investigated security issues, business logic concerns and potential improvements. The Smart Contract analysis reported no compiler error and a number of critical and medium severity issues. The contract Owner can access some admin functions that can be used in a malicious way to disturb the users' transactions. In particular, the contract owner has the ability to increase the withdrawal fees for the staking process, beyond the accessible limits and to transfer the staked amounts out of the contract. Other issues of moderate severity concern the consistent and robust execution of the code.

Disclaimer

The information provided in this report does not constitute investment, financial or trading advice and you should not treat any of the document's content as such. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes nor may copies be delivered to any other person other than the Company without Cyberscope's prior written consent. This report is not nor should be considered an "endorsement" or "disapproval" of any particular project or team. This report is not nor should be regarded as an indication of the economics or value of any "product" or "asset" created by any team or project that contracts Cyberscope to perform a security assessment. This document does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors' business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report represents an extensive assessment process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk Cyberscope's position is that each company and individual are responsible for their own due diligence and continuous security Cyberscope's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies and in no way claims any guarantee of security or functionality of the technology we agree to analyze. The assessment services provided by Cyberscope are subject to dependencies and are under continuing development. You agree that your access and/or use including but not limited to any services reports and materials will be at your sole risk on an as-is where-is and as-available basis Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives false negatives and other unpredictable results. The services may access and depend upon multiple layers of third parties.

About Cyberscope

Cyberscope is a blockchain cybersecurity company that was founded with the vision to make web3.0 a safer place for investors and developers. Since its launch, it has worked with thousands of projects and is estimated to have secured tens of millions of investors' funds.

Cyberscope is one of the leading smart contract audit firms in the crypto space and has built a high-profile network of clients and partners.

The Cyberscope team

cyberscope.io