Cyberscope

Audit Report

FROG

May 2023

Network	BSC
Address	0x5DC853039FC17EBe634513f1A1D9685be0E71a70
Audited by	© cyberscope

Table of Contents

Review

Explorer	https://bscscan.com/address/0x5dc853039fc17ebe634513f1a1d9685be0e71a70

Audit Updates

Initial Audit	18 Jul 2023

Source Files

Filename	SHA256
FrogStake.sol	7bbc4d09848fda796322cbf44c8c2937808d7611b14df16105f1e684da8d7364

Overview

The FrogStake contract is a robust staking system that allows users to stake tokens and earn rewards based on the amount staked and the duration of the stake. The contract is designed with a set of features that provide flexibility and control to both the contract owner and the users.

The contract owner has the authority to change the token address, transfer ownership, enable or disable staking, set pool fee (`poolFee`), set burn fee (`burnFee`), set stake limit (`minTxAmount`), set daily reward (`dailyReward`), and withdraw the remaining tokens from the contract. These functions provide the owner with the ability to manage the contract effectively and respond to changes in the token's ecosystem.

Additionally, the contract owner also has the obligation to add the reward tokens to the contract. This is a crucial aspect of the contract's operation, as it ensures that there are sufficient tokens in the contract to distribute as rewards to the users.

Stake

The FrogStake contract enables users to stake tokens by depositing a certain amount. This is done through the `stake` function, which takes the amount to be staked as an argument. The function first checks if staking is enabled and if the amount is greater than or equal to the minimum transaction amount. It then transfers the tokens from the user to the contract, updates the user's staking details, and increases the total amount staked in the contract. The rewards for the staked tokens start to accumulate from the moment of staking, based on the user's share of the total staked amount and the daily reward rate.

Claim

Users can claim their accumulated rewards by calling the `claim` function. This function first checks if the user has staked any tokens. It then calculates the rewards to be claimed by calling the `getRewards` function, which determines the rewards based on the time elapsed since the last reward claim and the user's share of the total staked amount. The function then updates the user's last reward timestamp, increases the total claimed rewards for the user, increases the total rewards distributed by the contract, and transfers the reward amount to the user.

Unstake

The `unstake` function provides users with the flexibility to `unstake` their tokens, including all of their unclaimed rewards, at any time. The function takes the amount to be unstaked as an argument. It first checks if the user has staked any tokens and if the unstake amount is below or equal to the user's staked balance. It then calculates the rewards to be claimed and the total amount to be transferred to the user. If the unstacking operation occurs before the end of the staking duration, a `burn fee` and a `pool fee` are deducted from the unstaked amount. The remaining amount, along with the unclaimed rewards, is then transferred to the user. The function also updates the user's staking details and decreases the total amount staked in the contract.

Rewards

The rewards calculation in the FrogStake contract is a linear process that is based on the user's staked amount, the user's share of the total staked amount in the pool, the daily reward rate which is set by the contract owner, and the time elapsed since the user's last reward claim.

The `getRewards` function is responsible for calculating the rewards. It first checks if the user has staked any tokens. If the user has a staked balance, the function calculates the stake amount and the time difference since the last reward claim.

Roles

Owner

The owner is responsible for setting up the token address in the constructor function during the deployment process. This token address will be used as the main token upon which the staking contract operates. All functionalities of the contract, including stake, unstake, claim, and other related processes, will be based on this token.

The Owner can interact with the following functions:

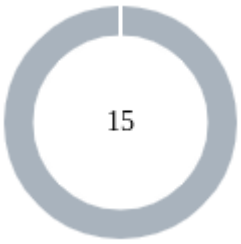
- `changeTokenAddress(address newTokenAddress)`
- `transferOwnership(address newOwner)`
- `disableStaking()`
- `enableStaking()`
- `function setPoolFee(uint256 _poolFee)`
- `function setBurnFee(uint256 _burnFee)`
- `function setStakeLimit(uint256 stakeLimit)`
- `function setDailyReward(uint256 _dailyReward)`
- `function transferTokens()`

User

The users can interact with the following functions:

- `stake(uint256 _amount)`
- `getRewards(address account)`
- `claim()`
- `unstake(uint256 _amount)`
- `function getUserDetails(address account)`

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	0	0	0	0
Minor / Informative	15	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	TAM	Token Address Modification	Unresolved
	FC	Fees Ckeck	Unresolved
	DKO	Delete Keyword Optimization	Unresolved
	MEE	Missing Events Emission	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	TUU	Time Units Usage	Unresolved
	RSML	Redundant SafeMath Library	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L07	Missing Events Arithmetic	Unresolved
	L09	Dead Code Elimination	Unresolved
	L13	Divide before Multiply Operation	Unresolved
	L18	Multiple Pragma Directives	Unresolved
	L19	Stable Compiler Version	Unresolved

●	L20	Succeeded Transfer Check	Unresolved

TAM - Token Address Modification

Criticality	Minor / Informative
Location	FrogStake.sol#L785,864,894,898
Status	Unresolved

Description

The contract owner has the authority to change the token address of the token by calling the `changeTokenAddress` function of the contract. While this function does have checks to prevent the new token address from being zero or the same as the current one, it does not account for the potential loss of user balances. If the token address is changed, users will not be able to call the `claim` or `unstake` function using the older token address. Since the contract will point to a new token address after the change, any token balance of the users associated with the previous token will be lost. This is because the `claim` and `unstake` functions transfer tokens from the contract to the user, and after the token address change, the contract's balance of the old token becomes irrelevant.

```
   function changeTokenAddress(address newTokenAddress)
        public
        onlyOwner
        returns (bool)
    {
        require(newTokenAddress != address(0), "Token address
cannot be 0");
        require(
            newTokenAddress != address(token),
            "Token address cannot be the same as the current
one"
        );
        emit TokenAddressUpdated(address(token),
newTokenAddress);
        token = ERC20(newTokenAddress);
        return true;
    }

    function claim() public returns (bool) {
        ...
        token.transfer(msg.sender, rewardToClaim);
        return true;
    }

    function unstake(uint256 _amount) public returns (bool) {
        ...
            token.transfer(burnAddress, burn);
        }

        //send % to user
        token.transfer(msg.sender, users);

        }else{

            token.transfer(msg.sender, amount);

        }
        ...
        return true;
    }
```

Recommendation

It is recommended to implement a mechanism which will ensure that if the token address changes to a new token, then the staked token or the reward amount of the contract will be able to be transferred back to users if there is need to. This could be achieved by creating a

function that allows users to withdraw their staked or reward amount of the old token after the token address has been changed. This function should only be callable by users who have a staked balance of the old token, and it should transfer the correct amount of the old token from the contract to the calling user. This will ensure that user balances are not lost when the token address is changed.

FC - Fees Ckeck

Criticality	Minor / Informative
Location	FrogStake.sol#L886,925,930
Status	Unresolved

Description

The contract is calculating the number of `users` amount based on the code: `users = amount * (100 - burnFee - poolFee) / 100`. This implies that the combined value of `burnFee` and `poolFee` should always be less than `100` to ensure that the user's value is correctly computed. However, the current implementation does not have any checks in place to validate that the sum of `burnFee` and `poolFee` remains below `100` when they are set. This can lead to unintended consequences and potential vulnerabilities in the contract's logic.

```
uint256 users = amount * (100 - burnFee - poolFee) / 100;
...
    function setPoolFee(uint256 _poolFee) public onlyOwner returns
(bool) {
        poolFee = _poolFee;
        return true;
    }

    function setBurnFee(uint256 _burnFee) public onlyOwner returns
(bool) {
       burnFee = _burnFee;
        return true;
    }
```

Recommendation

It is recommended to add an additional check in the `setPoolFee` and `setBurnFee` functions to ensure that the combined value of `burnFee` and `poolFee` does not exceed `100`. This can be achieved by adding a condition like require statement in both functions. This will ensure that the fee values will stay within the correct boundaries, preventing any miscalculations.

DKO - Delete Keyword Optimization

Criticality	Minor / Informative
Location	FrogStake.sol#L902
Status	Unresolved

Description

The contract resets variables to the default state by setting the initial values. Specifically within the `unstake` function, the contract sets the `unStake` variable of a `user` to zero when their `amount` is zero. While this approach is functional, it is not the most gas-efficient way to reset state variables. Setting values to state variables directly can increase the gas cost.

```
    if(user[msg.sender].amount == 0){
         user[msg.sender].unStake = 0;
     }
```

Recommendation

The team is advised to use the `delete` keyword instead of setting variables. Instead of setting `user[msg.sender].unStake` to zero, the team is advised to use `delete user[msg.sender]`. This can be more efficient than setting the variable to a new value, using delete can reduce the gas cost associated with storing data on the blockchain.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	FrogStake.sol#L813,858,869,925,930,940
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function stake(uint256 _amount) public returns (bool) {
        require(isStakingEnabled, "Staking is disabled");
        ...
        return true;
    }

    function claim() public returns (bool) {
        require(user[msg.sender].amount > 0, "user not exist");
        ...
        return true;

    }

    function unstake(uint256 _amount) public returns (bool) {
        uint256 tokenDecimals = token.decimals();
        uint256 unstakeAmount = _amount * 10**tokenDecimals;
        ...
        return true;
    }

    function setPoolFee(uint256 _poolFee) public onlyOwner
returns (bool) {
        poolFee = _poolFee;
        return true;
    }

    function setBurnFee(uint256 _burnFee) public onlyOwner
returns (bool) {
       burnFee = _burnFee;
        return true;
    }

    function setDailyReward(uint256 _dailyReward) public
onlyOwner returns (bool) {
        dailyReward = _dailyReward;
        return true;
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be

more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	FrogStake.sol#L820
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an 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. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
    function stake(uint256 _amount) public returns (bool) {
        require(isStakingEnabled, "Staking is disabled");
        ...
        token.transferFrom(msg.sender, address(this), amount);

        user[msg.sender].amount += amount;
        ...
    }
```

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
```

TUU - Time Units Usage

Criticality	Minor / Informative
Location	FrogStake.sol#L750,838,845
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 stakeDuration = 15778476; //6months
...
uint256 timeDiff = (block.timestamp - lastReward) / 60;
...
uint256 contractReward = (dailyReward * 10**tokenDecimals) /
1440;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	FrogStake.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	FrogStake.sol#L746,747
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.

```
uint256 stakeDuration = 15778476
address burnAddress =
0x000000000000000000000000000000000000dEaD
```

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	FrogStake.sol#L751,813,869,925,930,940
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.

```
struct userDetails {
        uint256 amount;
        uint256 lastReward;
        uint256 totalClaimed;
        uint256 unStake;
    }
uint256 _amount
uint256 _poolFee
uint256 _burnFee
uint256 _dailyReward
```

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/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	FrogStake.sol#L883,941
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.

```
amountStaked -= unstakeAmount
dailyReward = _dailyReward
```

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	FrogStake.sol#L623,646
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 _mint(address account, uint256 amount) internal
virtual {
        require(account != address(0), "ERC20: mint to the zero
address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);

        _afterTokenTransfer(address(0), account, amount);
    }

...
```

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	FrogStake.sol#L838,842,844,846,848
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 timeDiff = (block.timestamp - lastReward) / 60
uint256 rewardPerMinute =  (percentage * contractReward) /
10**tokenDecimals
uint256 rewardAmount = rewardPerMinute * timeDiff
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	FrogStake.sol#L8,237,263,347,376,729
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.4;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	FrogStake.sol#L8,237,263,347,376,729
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.0;
pragma solidity ^0.8.4;
```

Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	FrogStake.sol#L820,864,889,894,898,946
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
token.transferFrom(msg.sender, address(this), amount)
token.transfer(msg.sender, rewardToClaim)
token.transfer(burnAddress, burn)
token.transfer(msg.sender, users)
token.transfer(msg.sender, amount)
token.transfer(msg.sender, token.balanceOf(address(this)))
```

Recommendation

The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

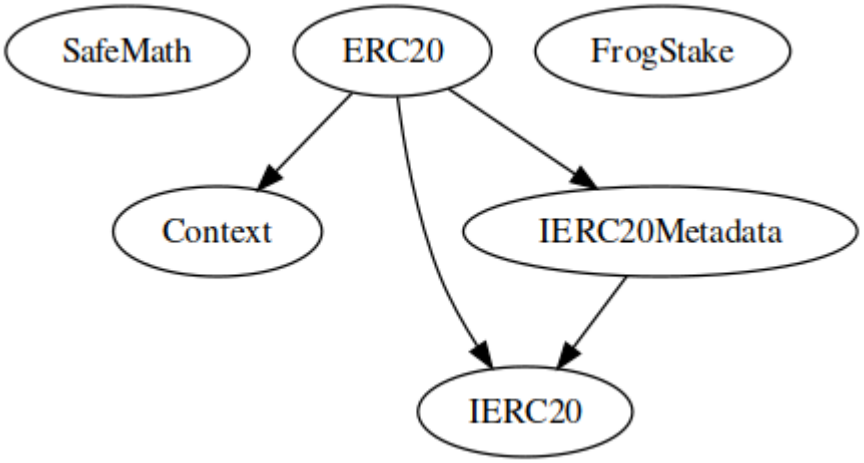
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
SafeMath	Library			
	tryAdd	Internal		
	trySub	Internal		
	tryMul	Internal		
	tryDiv	Internal		
	tryMod	Internal		
	add	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	mod	Internal		
	sub	Internal		
	div	Internal		
	mod	Internal		
Context	Implementation			
	_msgSender	Internal		
	_msgData	Internal		

IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
IERC20Metadata	Interface	IERC20		
	name	External		-
	symbol	External		-
	decimals	External		-
ERC20	Implementation	Context, IERC20, IERC20Metadata		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-

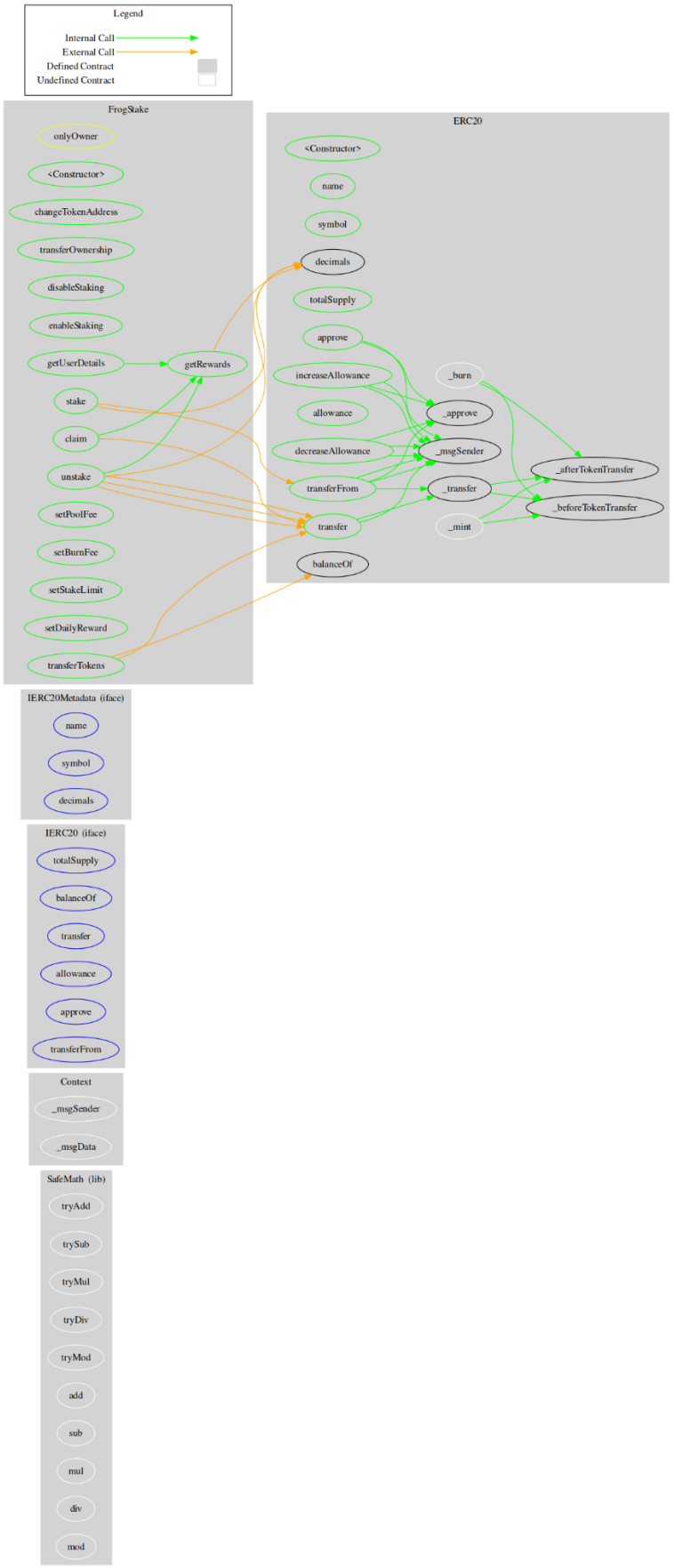
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_mint	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	
	_beforeTokenTransfer	Internal	✓	
	_afterTokenTransfer	Internal	✓	
FrogStake	Implementation			
		Public	✓	-
	changeTokenAddress	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	disableStaking	Public	✓	onlyOwner
	enableStaking	Public	✓	onlyOwner
	stake	Public	✓	-
	getRewards	Public		-
	claim	Public	✓	-
	unstake	Public	✓	-
	getUserDetails	Public		-

	setPoolFee	Public	✓	onlyOwner
	setBurnFee	Public	✓	onlyOwner
	setStakeLimit	Public	✓	onlyOwner
	setDailyReward	Public	✓	onlyOwner
	transferTokens	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
FrogStake
onlyOwner
<Constructor>
changeTokenAddress
transferOwnership
disableStaking
enableStaking
getUserDetails
getRewards
stake
claim
unstake
setPoolFee
setBurnFee
setStakeLimit
setDailyReward
transferTokens
ERC20
<Constructor>
name
symbol
decimals
totalSupply
approve
increaseAllowance
allowance
decreaseAllowance
transferFrom
transfer
balanceOf
_burn
_approve
_msgSender
_transfer
_mint
_afterTokenTransfer
_beforeTokenTransfer
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Context
_msgSender
_msgData
SafeMath (lib)
tryAdd
trySub
tryMul
tryDiv
tryMod
add
sub
mul
div
mod

Summary

FROG contract implements a staking and rewards mechanism. This audit investigates security issues, business logic concerns and potential improvements.

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

https://www.cyberscope.io