Cyberscope

Audit Report

Galaxy Fox Staking

May 2024

Repository https://github.com/humanshield89/galaxy-fox-token

Commit f0fc44d6a19a2cdfffc2fc86b696389ad0b88fca

Audited by © cyberscope

Table of Contents

Review

Repository	https://github.com/humanshield89/galaxy-fox-token
Commit	f0fc44d6a19a2cdfffc2fc86b696389ad0b88fca

Audit Updates

Initial Audit	14 May 2024

Source Files

Filename	SHA256
GFoxStakingStorage.sol	512c005c9fd84abe5f85fc24b4f63d7002b51ba5c48391eb3a653497ccfc8601

Overview

The GFOXStakeStorage contract implements a comprehensive staking system designed to manage user investments in cryptocurrency pools. It provides functionalities for admins to add, update, and remove pools, while allowing users to stake, claim rewards, and unstake their funds, including an option for early unstaking with a penalty. This contract uses a variety of parameters such as APY, lock periods, and staking limits to cater to diverse user preferences, ensuring flexibility and control in investment strategies while maintaining security and transparency in transactions.

Owner Functionalities

The admin has the authority to manage the operational aspects of the staking pools, ensuring flexibility and control over the staking environment. This includes the ability to add new pools with specific parameters such as APY (Annual Percentage Yield) expressed in basis points, lock period durations, and minimum or maximum stake limits. Additionally, admins can update existing pools to adjust these parameters or remove pools if necessary. These functionalities are crucial for maintaining the relevance and attractiveness of the staking options available to users.

Stake Functionality

Users can engage with the platform by staking their tokens into specific pools. When staking, users can select the pool that best fits their investment strategy based on the APY and lock period. The system calculates pending rewards based on the time the tokens are staked and the specific APY of the chosen pool. This allows users to potentially increase their earnings over time by benefiting from interest on their staked tokens.

Unstake Functionality

The unstake functionality provides users with the option to withdraw their staked tokens once the lock period has expired. This process includes calculating any earned rewards up to the point of lock period time, which are then added to the principal amount for withdrawal. This functionality is integral to giving users access to their funds and rewards after fulfilling the conditions of the lock period.

Early Unstake Functionality

For users who need to access their staked tokens before the end of the lock period, the early unstake functionality allows this action but with a penalty. The penalty, which is deducted from the total return, compensates for the shorter staking period and is transferred to a specified treasury account. This mechanism ensures that the integrity of the staking agreement is maintained, while still offering flexibility for users in urgent scenarios.

Claim Pending Rewards Functionality

Users can claim their accumulated rewards without unstaking their principal investment through the claim pending rewards functionality. This feature calculates the rewards based on the staked amount, the duration of the stake relative to the lock period, and the pool's APY. Users can claim these rewards at any point, providing them with liquidity and ongoing returns on their investment while still participating in the staking program.

Roles

onlyAdmin

The onlyAdmin can interact with the following functions:

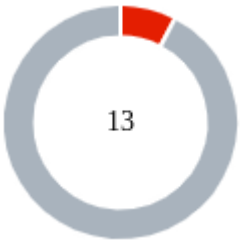
- function addPool
- function updatePool
- function setTreasury
- function stakeFor
- function recoverTokens
- function setPaused

Users

The users can interact with the following functions:

- function stake
- function unstake
- function earlyUnstake
- function claimPendingRewards

Findings Breakdown



Critical	1
Medium	0
Minor / Informative	12

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	0	0	0	0
Minor / Informative	12	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	IRM	Increased Rewards Miscalculation	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	MPV	Missing PoolID Validation	Unresolved
Minor / Informative	MU	Modifiers Usage	Unresolved
Minor / Informative	PBV	Percentage Boundaries Validation	Unresolved
Minor / Informative	PTAI	Potential Transfer Amount Inconsistency	Unresolved
Minor / Informative	RCO	Redundant Calculation Overhead	Unresolved
Minor / Informative	TUU	Time Units Usage	Unresolved
Minor / Informative	TSI	Tokens Sufficiency Insurance	Unresolved
Minor / Informative	OCTD	Transfers Contract's Tokens	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L19	Stable Compiler Version	Unresolved

IRM - Increased Rewards Miscalculation

Criticality	Critical
Location	GFoxStakingStorage.sol#L243,267
Status	Unresolved

Description

The contract contains the `_claimPendingRewards` function that enables users to claim rewards based on the time their funds have been staked. However, there is a significant oversight in the `_unstake` and `_unstakeWithPenalty` functions where both recalculate and disburse rewards for the entire staking period again without deducting any rewards previously claimed by the user through the `_claimPendingRewards` function. This flaw leads to the possibility of users claiming more than their rightful share of rewards, as each claim does not update the total rewards to reflect amounts previously claimed. Consequently, this could potentially result in the depletion of pool reserves faster than anticipated, undermining the financial stability and trust in the smart contract.

```
    function _claimPendingRewards(uint256 _poolId) internal {
        Pool storage pool = pools[_poolId];
        UserStake storage userStake = userStakes[_poolId][_msgSender()];

        require(userStake.amount > 0, "GFOXStake: No stake");

        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        uint256 rewards = (userStake.amount * userStake.apy *
timePassed) /
            (BPS_DENOMINATOR * ONE_YEAR_SECONDS);

        uint256 rewardsToTransfer = rewards - userStake.rewardsClaimed;

        userStake.rewardsClaimed = rewards;

        gfoxToken.safeTransfer(_msgSender(), rewardsToTransfer);

        emit Claimed(_msgSender(), _poolId, rewardsToTransfer);

        pool.reservedRewards -= rewardsToTransfer;
    }

    function _unstake(uint256 _poolId) internal {
        Pool storage pool = pools[_poolId];
        UserStake storage userStake = userStakes[_poolId][_msgSender()];

        require(
            userStake.stakedOn + pool.lockPeriod < block.timestamp,
            "GFOXStake: Locked"
        );

        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        uint256 rewards = (userStake.amount * userStake.apy *
timePassed) /
            (BPS_DENOMINATOR * ONE_YEAR_SECONDS);

        pool.totalStaked -= userStake.amount;
        pool.reservedRewards -= userStake.rewardsReserved;

        emit Unstaked(_msgSender(), _poolId, userStake.amount);
        emit Claimed(_msgSender(), _poolId, rewards);
```

```
        uint256 transferAmount = userStake.amount + rewards;

        delete userStakes[_poolId][_msgSender()];

        gfoxToken.safeTransfer(_msgSender(), transferAmount);
    }

    function _unstakeWithPenalty(uint256 _poolId) internal {
    ...
    }
```

Recommendation

It is recommended to refactor the `_unstake` and `_unstakeWithPenalty` function to correctly account for any rewards already claimed through the `_claimPendingRewards` function. Specifically, the calculation of rewards during unstaking or unstaking with penalty should subtract any rewards previously claimed by the user. Implementing this change will ensure that rewards are not incorrectly disbursed multiple times for the same staking period, thereby protecting the integrity of the reward pool and maintaining equitable reward distribution among all participants. Additionally, enhancing the tracking of claimed rewards and integrating checks within the unstaking process will bolster the contract's robustness against potential abuses and errors.

CR - Code Repetition

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L
Status	Unresolved

Description

The contract contains repetitive code segments. There are potential issues that can arise when using code segments in Solidity. Some of them can lead to issues like gas efficiency, complexity, readability, security, and maintainability of the source code. It is generally a good idea to try to minimize code repetition where possible.

Specifically the `_claimPendingRewards` , `_unstake` and `_unstakeWithPenalty` functions share similar code segments.

```
    function _claimPendingRewards(uint256 _poolId) internal {
        ...
        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        uint256 rewards = (userStake.amount * userStake.apy *
timePassed) /
            (BPS_DENOMINATOR * ONE_YEAR_SECONDS);

        uint256 rewardsToTransfer = rewards - userStake.rewardsClaimed;

        userStake.rewardsClaimed = rewards;

        gfoxToken.safeTransfer(_msgSender(), rewardsToTransfer);

        emit Claimed(_msgSender(), _poolId, rewardsToTransfer);

        pool.reservedRewards -= rewardsToTransfer;
    }

    function _unstake(uint256 _poolId) internal {
        ...
        require(
            userStake.stakedOn + pool.lockPeriod < block.timestamp,
            "GFOXStake: Locked"
        );

        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        uint256 rewards = (userStake.amount * userStake.apy *
timePassed) /
            (BPS_DENOMINATOR * ONE_YEAR_SECONDS);

        pool.totalStaked -= userStake.amount;
        pool.reservedRewards -= userStake.rewardsReserved;

        emit Unstaked(_msgSender(), _poolId, userStake.amount);
        emit Claimed(_msgSender(), _poolId, rewards);

        uint256 transferAmount = userStake.amount + rewards;

        delete userStakes[_poolId][_msgSender()];

        gfoxToken.safeTransfer(_msgSender(), transferAmount);
    }
```

```
    function _unstakeWithPenalty(uint256 _poolId) internal {
        ...
        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        uint256 rewards = (userStake.amount * userStake.apy *
timePassed) /
            (BPS_DENOMINATOR * ONE_YEAR_SECONDS);

        uint256 penalty = (userStake.amount * pool.penalty) /
BPS_DENOMINATOR;

        pool.totalStaked -= userStake.amount;
        pool.reservedRewards -= userStake.rewardsReserved;

        uint256 transferAmount = (userStake.amount + rewards) - penalty;

        ...

        delete userStakes[_poolId][_msgSender()];

        gfoxToken.safeTransfer(_msgSender(), transferAmount);

        gfoxToken.safeTransfer(treasury, penalty);
    }
```

Recommendation

The team is advised to avoid repeating the same code in multiple places, which can make the contract easier to read and maintain. The authors could try to reuse code wherever possible, as this can help reduce the complexity and size of the contract. For instance, the contract could reuse the common code segments in an internal function in order to avoid repeating the same code in multiple places.

MPV - Missing PoolID Validation

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L156,161
Status	Unresolved

Description

The contract is currently structured to execute functions without first verifying the existence of the specified `poolId`. This oversight may lead to erroneous transactions where users attempt to interact with non-existent pool IDs, potentially causing disruptions in contract functionality and user experience. Such an issue can lead to failed transactions or unintended effects within the contract's logic, as the internal `_stake` and `_unstake` functions are called regardless of the validity of the poolId provided.

```
    function stake(uint256 _poolId, uint256 _amount) external {
        require(!paused, "GFOXStake: Paused");
        _stake(_msgSender(), _poolId, _amount);
    }

    function unstake(uint256 _poolId) external {
        require(!paused, "GFOXStake: Paused");
        _unstake(_poolId);
    }
```

Recommendation

It is recommended to include an additional check within the functions to verify if the poolId `exists` before proceeding with further functionalities. This validation can be implemented by adding a mapping to track existing pool IDs or by ensuring that the poolId corresponds to a valid and active pool within the system. This check should be designed to reject any transactions involving invalid pool IDs, thereby safeguarding the contract against potential misuse and enhancing its overall robustness.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L80,180
Status	Unresolved

Description

The contract's functionality and behavior 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.

The admin address has the authority to pause the contract, which consequently suspends the staking functionality. Additionally, the admin holds the power to add or update pool information concerning staking variables through functions like `addPool` and `updatePool`. As a result, it is crucial that the owner sets accurate values for these parameters to avoid disrupting the staking process. The ability to modify key parameters and control the operational state emphasizes the importance of diligent management by the admin to ensure the system functions smoothly and securely.

```
    function addPool(
        uint256 _apyBPS,
        uint256 _lockPeriod,
        uint256 _minStake,
        uint256 _maxStake,
        uint256 _penaltyBPS
    ) external onlyAdmin {
        _addPool(_apyBPS, _lockPeriod, _minStake, _maxStake,
_penaltyBPS);
    }

    function updatePool(
        uint256 _poolId,
        uint256 _apy,
        uint256 _lockPeriod,
        uint256 _minStake,
        uint256 _maxStake,
        uint256 _penalty
    ) external onlyAdmin {
        _updatePool(_poolId, _apy, _lockPeriod, _minStake,
_maxStake, _penalty);
    }

    function setPaused(bool _paused) external onlyAdmin {
        paused = _paused;
    }
```

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L152,157,162,167,172
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(!paused, "GFOXStake: Paused");
```

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.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L315
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
uint256 penalty = (userStake.amount * pool.penalty) /
BPS_DENOMINATOR;
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L193
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

```
gfoxToken.safeTransferFrom(_msgSender(), address(this),
_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`

RCO - Redundant Calculation Overhead

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L272,302
Status	Unresolved

Description

The contract is currently employing a `require` statement in both the `_unstake` and `_unstakeWithPenalty` functions to ensure that the staking period conditions are met before proceeding. This verification checks if the `lockPeriod` has passed or not, based on the `stakedOn` timestamp. Following this check, the contract uses the `_min` function to calculate the `timePassed`, comparing `block.timestamp - userStake.stakedOn` with `pool.lockPeriod`. However, since the eligibility for unstaking is already assured by the require statement, the use of `_min` function becomes unnecessary and introduces redundant computational overhead.

```
   function _unstake(uint256 _poolId) internal {
        ...
        require(
            userStake.stakedOn + pool.lockPeriod <
block.timestamp,
            "GFOXStake: Locked"
        );

        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );

        ...
    }

    function _unstakeWithPenalty(uint256 _poolId) internal {
        ...
        require(
            userStake.stakedOn + pool.lockPeriod >
block.timestamp,
            "GFOXStake: Unlocked"
        );

        uint256 timePassed = _min(
            block.timestamp - userStake.stakedOn,
            pool.lockPeriod
        );
        ...
        }
```

Recommendation

It is recommended to remove the `_min` function and directly return the `lockPeriod` for the `_unstake` function, and `block.timestamp - userStake.stakedOn` for the `_unstakeWithPenalty` function. This adjustment will streamline the code by eliminating superfluous calculations, thereby optimizing gas usage and reducing potential points of failure.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L8
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 constant ONE_YEAR_SECONDS = 365 * 24 * 60 * 60;
```

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.

TSI - Tokens Sufficiency Insurance

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L261
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.

```
gfoxToken.safeTransfer(_msgSender(), rewardsToTransfer);
```

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 presale 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.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L176
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 `recoverTokens` function.

```
    function recoverTokens(address _token, uint256 _amount)
external onlyOwner {
        IERC20(_token).safeTransfer(_msgSender(), _amount);
    }
```

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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L52,53,54,55,70,71,72,73,81,82,83,84,85,91,92,93,94,95,96,137,148,149,150,156,161,166,171,176,180
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.

```
address _gfoxToken
address _owner
address _treasury
PoolCreation[] calldata _pools

function __init_GFOXStakeStorage(
        address _gfoxToken,
        address _owner,
        address _treasury
    ) internal {
        gfoxToken = IERC20(_gfoxToken);
        treasury = _treasury;
        __GFAccessControlUpgradable_init(_owner);
    }
uint256 _apyBPS

...
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	GFoxStakingStorage.sol#L2
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 ^0.8.23;
```

Recommendation

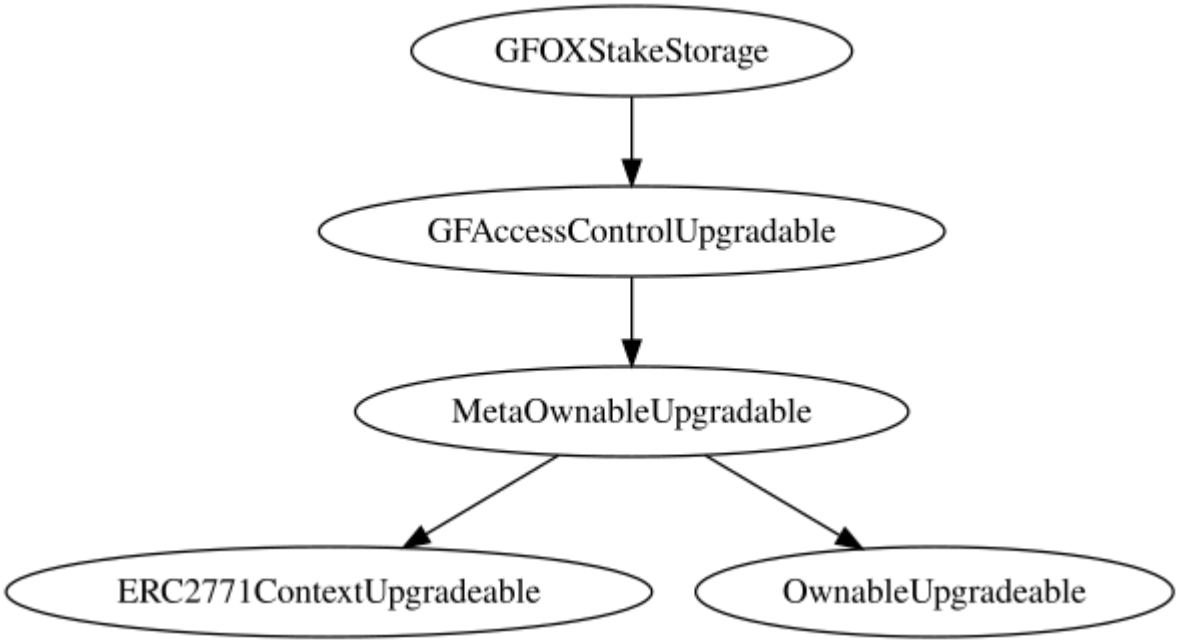
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest 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.

Functions Analysis

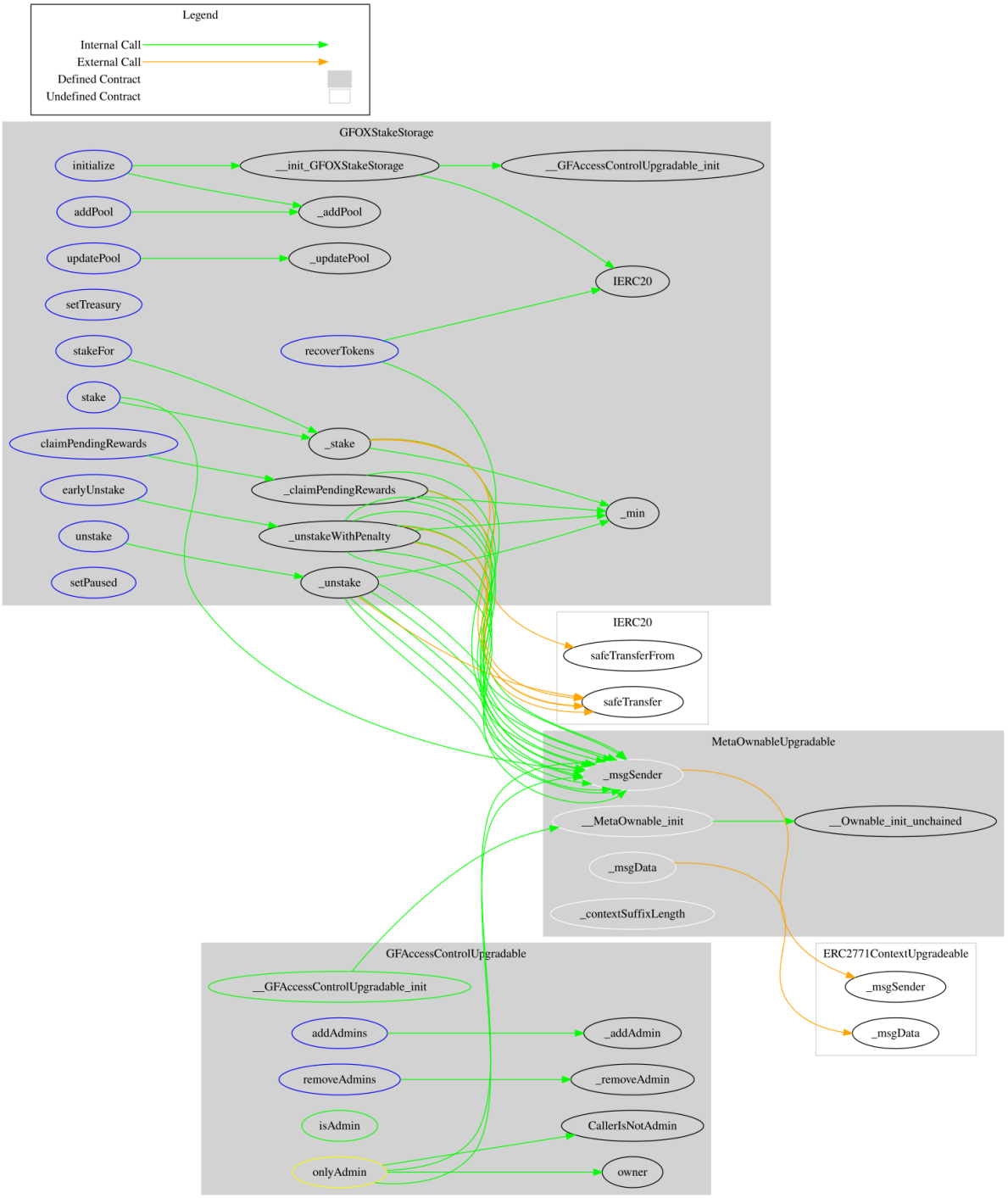
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
GFOXStakeStorage	Implementation	GFAccessControlUpgradable		
	initialize	External	✓	initializer
	__init_GFOXStakeStorage	Internal	✓	
	addPool	External	✓	onlyAdmin
	updatePool	External	✓	onlyAdmin
	_addPool	Internal	✓	
	_updatePool	Internal	✓	
	setTreasury	External	✓	onlyAdmin
	stakeFor	External	✓	onlyAdmin
	stake	External	✓	-
	unstake	External	✓	-
	earlyUnstake	External	✓	-
	claimPendingRewards	External	✓	-
	recoverTokens	External	✓	onlyOwner
	setPaused	External	✓	onlyAdmin
	_min	Internal		
	_stake	Internal	✓	
	_claimPendingRewards	Internal	✓	

	_unstake	Internal	✓	
	_unstakeWithPenalty	Internal	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
GFOXStakeStorage
initialize
__init_GFOXStakeStorage
__GFAccessControlUpgradable_init
addPool
_addPool
updatePool
_updatePool
IERC20
setTreasury
stakeFor
recoverTokens
stake
_stake
claimPendingRewards
earlyUnstake
_claimPendingRewards
_min
unstake
_unstakeWithPenalty
setPaused
_unstake
IERC20
safeTransferFrom
safeTransfer
MetaOwnableUpgradable
_msgSender
__MetaOwnable_init
__Ownable_init_unchained
_msgData
_contextSuffixLength
GFAccessControlUpgradable
__GFAccessControlUpgradable_init
ERC2771ContextUpgradeable
_msgSender
addAdmins
_addAdmin
_msgData
removeAdmins
_removeAdmin
isAdmin
CallerIsNotAdmin
onlyAdmin
owner

Summary

The Galaxy Fox Stake contract implements a staking and rewards distribution mechanism designed to allow users to deposit tokens into designated pools, earn interest based on the time and amount staked, and manage their investments through functions like claiming rewards, unstaking, and unstaking with penalty. This audit investigates various aspects of the contract, including security vulnerabilities, business logic errors, and potential improvements to ensure that the contract operates efficiently and securely. Our review focuses on identifying issues that could affect the integrity of the staking process, the accuracy of reward calculations, and the overall user experience, aiming to enhance the contract's robustness and reliability in a live environment. Overall, 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