Cyberscope

Audit Report

Genesis Staking

June 2024

Repository	https://github.com/hrawi20/genesis-contracts
Commit	3723cca78a258c497fba1db4f8f3d705cabfa9aa
Audited by	© cyberscope

Table of Contents

Review

Repository	https://github.com/hrawi20/genesis-contracts
Commit	3723cca78a258c497fba1db4f8f3d705cabfa9aa

Audit Updates

Initial Audit	20 Jun 2024

Source Files

Filename	SHA256
GenesisStaking.sol	ae384014e9f31967a70974619cd853de17f935df18db183722caa788aef14b14

Overview

The GenesisStaking contract is designed to provide a comprehensive staking mechanism with rebase functionality, which adjusts token supply based on predefined APR parameters. This contract allows users to stake and unstake tokens, and includes various administrative functions for the contract owner to manage and configure the staking and rebase parameters.

Stake

Users have the ability to stake their tokens by calling the `stake` function. When staking, the contract first performs a rebase to adjust the total supply of tokens. The staked amount is then transferred from the user to the contract, after which a stake tax is deducted and the net amount is burned. The user is credited with staking tokens equivalent to the net amount after tax. This process helps manage the supply of the base token and incentivizes participation in the staking process.

Unstake

The `unstake` function allows users to withdraw their staked tokens. Similar to the staking process, the contract performs a rebase before proceeding with the unstake operation. The function checks the user's balance, burns the staking tokens, and mints the equivalent amount of base tokens to the contract. An unstake tax is deducted, and the net amount is transferred back to the user. This function ensures that the supply and value dynamics remain consistent with the staking mechanism.

Rebase Functionality

The rebase functionality adjusts the total supply of tokens periodically based on the fixed and dynamic APR values. It calculates the supply delta, which is the change in supply needed to achieve the desired APR. The contract updates the maximum supply and recalculates the `gonsPerFragment` value to reflect the new supply. This mechanism helps maintain the token's economic stability and aligns with the intended APR targets, ensuring a balanced token supply over time.

Owner Functionality

The owner has the ability to set several crucial parameters of the contract that affect the tokenomics and rebase mechanism. The `setAprParameters` function allows the owner to configure the fixed APR, dynamic APR caps, and dynamic APR constant. The owner can also set the rebase interval, stake tax, and unstake tax using the respective setter functions. Additionally, the owner can set the base token, liquidity pool, treasury address, and stake start time. These functions provide the owner with control over the key economic aspects of the contract, enabling adjustments to be made in response to changing market conditions or strategic goals.

By carefully managing these parameters, the owner can influence the overall supply and value dynamics, ensuring the contract operates effectively and maintains user confidence in the staking mechanism.

Roles

Owner

The owner can interact with the following functions:

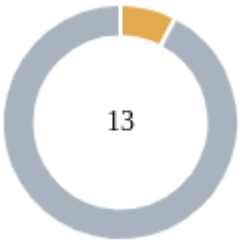
- function setBaseToken
- function setLiquidityPool
- function setAprParameters
- function setRebaseInterval
- function setStakeTax
- function setUnstakeTax
- function setTreasury
- function setStakeStartTime
- function emergencyWithdraw
- function emergencyEthWithdraw

Users

The users can interact with the following functions:

- function rebase
- function stake
- function unstake

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	12

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Medium	PFLRM	Potential Flash Loan Rebase Manipulation	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	CSC	Contradictory Sender Check	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	MTEE	Missing Transfer Event Emission	Unresolved
Minor / Informative	RRF	Rebase Reusable Functionality	Unresolved
Minor / Informative	STV	Start Time Validation	Unresolved
Minor / Informative	L02	State Variables could be Declared Constant	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L13	Divide before Multiply Operation	Unresolved
Minor / Informative	L16	Validate Variable Setters	Unresolved
Minor / Informative	L19	Stable Compiler Version	Unresolved
Minor / Informative	L20	Succeeded Transfer Check	Unresolved

PFLRM - Potential Flash Loan Rebase Manipulation

Criticality	Medium
Location	GenesisStaking.sol#L179,222,241
Status	Unresolved

Description

The contract includes mechanisms for staking and unstaking tokens as well as a rebase function that adjusts the token's supply based on specific economic triggers. However, a vulnerability exists due to how these functionalities interact and the conditions under which they operate.

The problem arises with the interplay between staking, unstaking, and rebasing. Users can call the `stake` and `unstake` functions, which in turn invoke the rebase function, thus recalibrating the total supply just before any tokens are staked or unstaked. Within the staking function, a tax is deducted from the staked amount, which is emitted as a `Taxed` event, and the net amount is minted to the user's account. Similarly, in unstaking, the tokens are returned to the user after deducting a potential tax, which is then transferred to a treasury.

The vulnerability is that users have the option to not participate to staking if the rebased amount, after considering the tax, results in no positive net gain for them. Moreover, the ability to frequently invoke these functions without stringent checks allows users to artificially manipulate the supply since minting and burning functionalities are called repeatedly. This opens the door to potential flashloan attacks, where a user could borrow a large number of tokens, influence the rebase outcome by strategically timing staking and unstaking actions, and thereby exploit the supply mechanism for personal gain. Such activities could destabilize the token's economy by artificially inflating or deflating its supply in short periods, leading to significant financial discrepancies and undermining the token's intended economic model.

```
function rebase() public {
        if (block.timestamp < stakeStartTime) return;

        ...
        if (supplyDelta > maxSupply) {
            supplyDelta = maxSupply;
        }

        maxSupply += supplyDelta;
        gonsPerFragment = maxGons / maxSupply;

        totalSupplyTokens = totalGons / gonsPerFragment;
        lastRebaseTime = timeToUse + rebaseInterval * rebaseCount;

        ...
        );
    }

    function stake(uint256 amount) external {
        rebase();
        ...
        uint256 taxAmount = (amount * stakeTax) / PERCENTAGE_BASE;
        emit Taxed(taxAmount);
        uint256 netAmount = amount - taxAmount;
        ...

        _mint(msg.sender, netAmount);
       ...
    }

    function unstake(uint256 amount) external {
        rebase();

        ...

        IERC20(baseToken).transfer(msg.sender, netAmount);
        if (taxAmount > 0) {
            IERC20(baseToken).transfer(treasury, taxAmount);
        }

        emit Unstake(msg.sender, amount, taxAmount, netAmount);
    }
```

Recommendation

It is recommended to implement safeguards against such exploitative behavior. Measures could include limiting the frequency of rebase calls, implementing stricter checks on `stake` and `unstake` operations, and monitoring large or suspicious transactions to

detect and prevent flash loan attacks. Enhancing the transparency of rebase operations and requiring a minimum staking period before unstaking can also mitigate these risks.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	GenesisStaking.sol#L110,429,434
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 owner have the ability to set crucial parameters that directly impact the tokenomics and the rebase mechanism variables. Specifically, the owner can adjust parameters such as the fixed APR, dynamic APR minimum and maximum caps, and the dynamic APR constant. These adjustments can significantly influence the supply and value dynamics of the token. If not managed carefully, such centralized control could lead to adverse effects on the token's stability and market trust.

```
    function setAprParameters(
        uint256 _fixedAPR,
        uint256 _dynamicAPRMinCap,
        uint256 _dynamicAPRMaxCap,
        uint256 _dynamicAPRConstant
    ) external onlyOwner {
        require(_dynamicAPRMinCap <= _dynamicAPRMaxCap, "Min
cap higher than max cap");
        fixedAPR = _fixedAPR;
        dynamicAPRMinCap = _dynamicAPRMinCap;
        dynamicAPRMaxCap = _dynamicAPRMaxCap;
        dynamicAPRConstant = _dynamicAPRConstant;

        emit APRParamsSet(_fixedAPR, _dynamicAPRMinCap,
_dynamicAPRMaxCap, _dynamicAPRConstant);
    }
```

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. The contract owner should carefully examine the tokenomics and the broader impact of any parameter changes. Additionally, consider implementing multi-signature authorization or community voting to approve significant changes to the parameters. This will ensure that changes are made transparently and with broader consensus, enhancing the trust and stability of the token ecosystem.

CSC - Contradictory Sender Check

Criticality	Minor / Informative
Location	GenesisStaking.sol#L266
Status	Unresolved

Description

The contract is implementing a contradictory check within the `_transfer` function. Specifically, there is a requirement that the `sender` address must not be the zero address, which is a standard safeguard in ERC20 contracts to prevent invalid transfers. However, the contract also includes a contradictory check that requires the sender address to be the zero address in a certain condition. This conflicting logic can lead to confusion about the actual intent and behavior of the function, potentially causing unexpected outcomes or failures in transfer operations.

```
    function _transfer(address sender, address recipient,
uint256 amount) internal {
        require(sender != address(0), "ERC20: transfer from the
zero address");
        ...
        require(sender == address(0) || recipient !=
address(this), "ERC20: transfer to staking contract");
        ...
```

Recommendation

It is recommended to review and clarify the logic within the `_transfer` function to remove any contradictory requirements. Ensure that the checks are logically consistent and aligned with the intended behavior of the function. Specifically, the condition that requires `sender == address(0)` should be re-evaluated or removed if it contradicts the initial check that `sender != address(0)`. This will improve the clarity and reliability of the transfer process, preventing potential issues arising from ambiguous or conflicting conditions.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	GenesisStaking.sol#L89,90
Status	Unresolved

Description

The contract declares state variables that their value is initialized once in the constructor and are not modified afterwards. The `immutable` is a special declaration for this kind of state variables that saves gas when it is defined.

```
initialMaxSupply
maxGons
```

Recommendation

By declaring a variable as immutable, the Solidity compiler is able to make certain optimizations. This can reduce the amount of storage and computation required by the contract, and make it more gas-efficient.

MTEE - Missing Transfer Event Emission

Criticality	Minor / Informative
Location	GenesisStaking.sol#L279
Status	Unresolved

Description

The contract does not emit an event when portions of the main amount are transferred during the transfer process. This lack of event emission results in decreased transparency and traceability regarding the flow of tokens, and hinders the ability of decentralized applications (dApps), such as blockchain explorers, to accurately track and analyze these transactions.

Specifically, the contract is missing an event to emit the trasnfer to the treasury address.

```
if (gonsTaxAmount > 0) {
    _gonBalances[treasury] += gonsTaxAmount;
}
```

Recommendation

It is advisable to incorporate the emission of detailed event logs following each asset transfer. These logs should encapsulate key transaction details, including the identities of the sender and receiver, and the quantity of assets transferred. Implementing this practice will enhance the reliability and transparency of transaction tracking systems, ensuring accurate data availability for ecosystem participants.

RRF - Rebase Reusable Functionality

Criticality	Minor / Informative
Location	GenesisStaking.sol#L179,391
Status	Unresolved

Description

The contract is currently calculating the supply delta directly within the `rebase` function, leading to repetitive calculations that could be optimized. By performing the supply delta calculation within the `rebase` function, the contract repeats several steps that are already encapsulated in the `totalTokensAtNextRebase` function. This approach not only introduces redundancy but also makes the code more complex and harder to maintain. Leveraging the `totalTokensAtNextRebase` function, which already computes the necessary values for determining the supply delta, could streamline the process. This change would enhance the clarity and efficiency of the code by centralizing the calculation logic in a single place, thus reducing the potential for errors and improving maintainability.

```
    function rebase() public {
        if (block.timestamp < stakeStartTime) return;

        uint256 timeToUse = lastRebaseTime > 0 ? lastRebaseTime :
stakeStartTime;
        uint256 rebaseCount = (block.timestamp - timeToUse) /
rebaseInterval;
        if (rebaseCount == 0) return;

        uint256 fixedApr = getFixedAPR();
        uint256 dynamicApr = getDynamicAPR();
        uint256 finalApr = fixedApr + dynamicApr;

        uint256 intervalsPerYear = ONE_YEAR / rebaseInterval;
        uint256 intervalAPR = finalApr / intervalsPerYear;
        uint256 supplyDelta = (maxSupply * intervalAPR) / SCALE / 100;
        ...
    }

    function totalTokensAtNextRebase() public view returns (uint256) {
        uint256 fixedApr = getFixedAPR();
        uint256 dynamicApr = getDynamicAPR();
        uint256 finalApr = fixedApr + dynamicApr;

        uint256 intervalsPerYear = ONE_YEAR / rebaseInterval;
        uint256 intervalAPR = finalApr / intervalsPerYear;
        uint256 supplyDelta = (totalSupplyTokens * intervalAPR) / SCALE
/ 100;

        return supplyDelta;
    }
```

Recommendation

It is recommended to refactor the `rebase` function to utilize the `totalTokensAtNextRebase` function for calculating the supply delta. This adjustment will consolidate the logic for supply delta computation, ensuring that the calculation is performed consistently and efficiently. By using the `totalTokensAtNextRebase` function, the contract will benefit from reduced code duplication, making it easier to read, understand, and maintain. This improvement will also help in minimizing the risk of introducing bugs during future updates or modifications to the rebase calculation logic.

STV - Start Time Validation

Criticality	Minor / Informative
Location	GenesisStaking.sol#L77
Status	Unresolved

Description

The contract is missing a check to verify that the `stakeStartTime` is greater than the current time during its initialization. This could allow the staking process to begin immediately, potentially leading to unexpected behavior or premature staking before the intended start time.

```
    constructor(
        address _baseToken,
        address _liquidityPool,
        address _treasury,
        uint256 _stakeStartTime
    ) Ownable(msg.sender) {
        baseToken = _baseToken;
        liquidityPool = _liquidityPool;
        treasury = _treasury;
        stakeStartTime = _stakeStartTime;
        ...
    }
```

Recommendation

It is recommended to implement a validation check during the contract's initialization to ensure that the `stakeStartTime` is set to a future time relative to the current blockchain timestamp. This will prevent the staking process from starting before the designated start time.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	GenesisStaking.sol#L21,22,23
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.

```
string public name = "Genesis Staking"
string public symbol = "sGEN"
uint8 public decimals = 18
```

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	GenesisStaking.sol#L100,105,111,112,113,114,125,130,136,142,147
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 _baseToken
address _liquidityPool
uint256 _fixedAPR
uint256 _dynamicAPRMinCap
uint256 _dynamicAPRMaxCap
uint256 _dynamicAPRConstant
uint256 _rebaseInterval
uint256 _stakeTax
uint256 _unstakeTax
address _treasury
uint256 _stakeStartTime
```

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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	GenesisStaking.sol#L183,191,192,195,207,270,274,300,306,332,336,397,398,406,408
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 ratio = (totalStaked * SCALE) / lpBalance
uint256 scaledRatio = (ratio * dynamicAPRConstant) / SCALE
```

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	GenesisStaking.sol#L83,84,85,101,106,143,435
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.

```
baseToken = _baseToken
liquidityPool = _liquidityPool
treasury = _treasury
payable(to).transfer(amount)
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	GenesisStaking.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 ^0.8.0;
```

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	GenesisStaking.sol#L233,254,256,431
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.

```
IERC20(baseToken).transfer(treasury, taxAmount)
IERC20(baseToken).transfer(msg.sender, netAmount)
IERC20(token).transfer(to, amount)
```

Recommendation

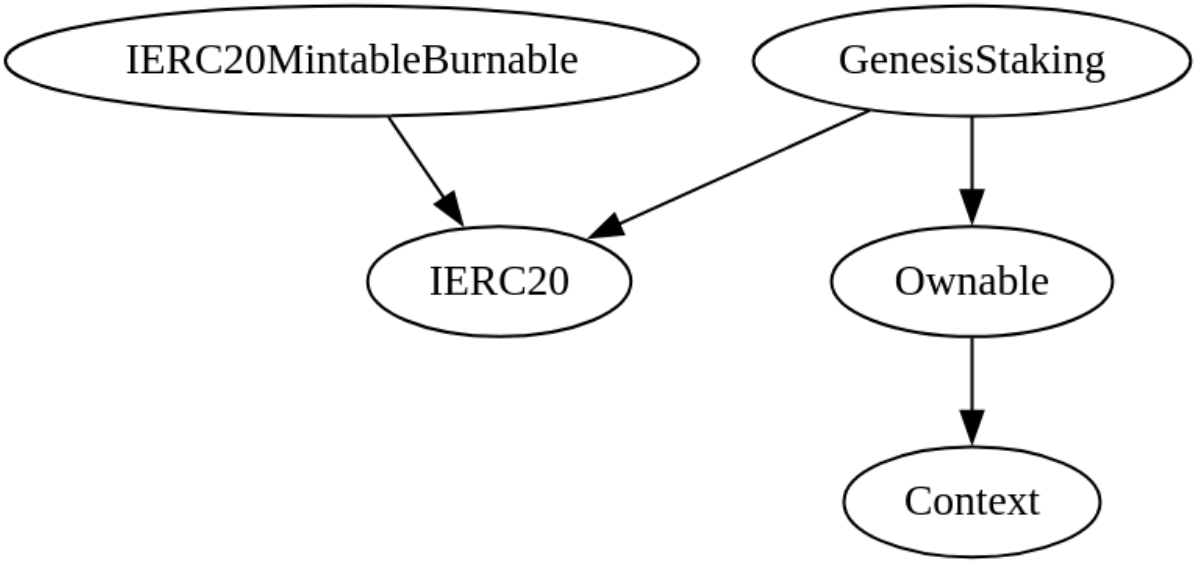
The contract should check if the result of the transfer methods is successful. The team is advised to check the SafeERC20 library from the Openzeppelin library.

Functions Analysis

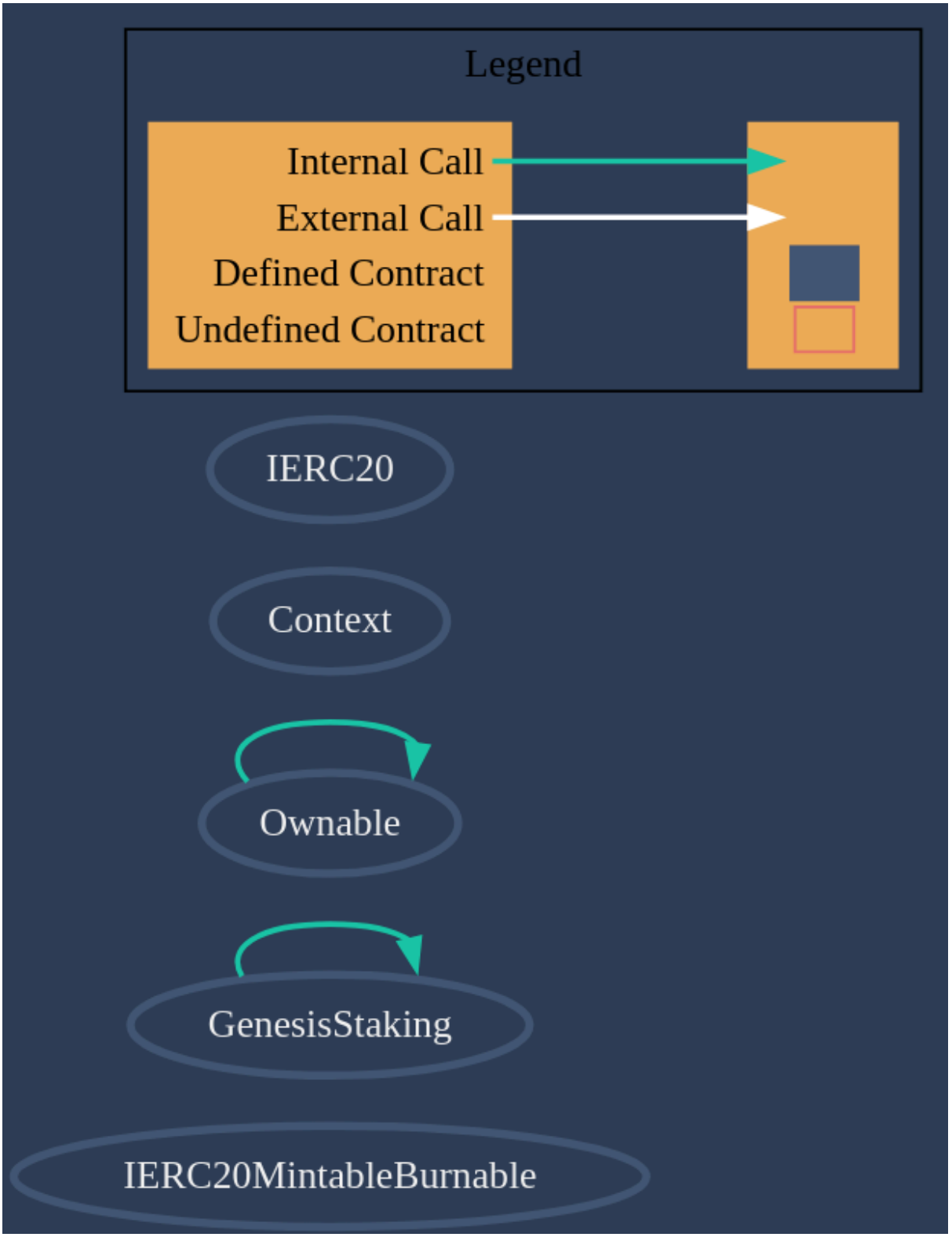
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
GenesisStaking	Implementation	IERC20, Ownable		
		Public	✓	Ownable
	setBaseToken	External	✓	onlyOwner
	setLiquidityPool	External	✓	onlyOwner
	setAprParameters	External	✓	onlyOwner
	setRebaseInterval	External	✓	onlyOwner
	setStakeTax	External	✓	onlyOwner
	setUnstakeTax	External	✓	onlyOwner
	setTreasury	External	✓	onlyOwner
	setStakeStartTime	External	✓	onlyOwner
	_mint	Internal	✓	
	_burn	Internal	✓	
	rebase	Public	✓	-
	stake	External	✓	-
	unstake	External	✓	-
	_transfer	Internal	✓	
	getFixedAPR	Public		-
	getDynamicAPR	Public		-
	getFinalAPR	Public		-

	log10	Internal		
	balanceOf	Public		-
	totalSupply	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	_approve	Internal	✓	
	timeTillNextRebase	Public		-
	totalTokensAtNextRebase	Public		-
	tokensForAddressAtNextRebase	Public		-
	tokensDeductedForUnstaking	Public		-
	tokensDeductedForStaking	Public		-
	index	Public		-
	totalTokensRewarded	Public		-
	emergencyWithdraw	External	✓	onlyOwner
	emergencyEthWithdraw	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IERC20
Context
Ownable
GenesisStaking
IERC20MintableBurnable

Summary

The GenesisStaking contract implements a dynamic staking and rebase mechanism designed to adjust the token supply based on predefined APR parameters and user staking activities. This audit investigates security issues, business logic concerns, and potential improvements to ensure the contract's robustness, efficiency, and alignment with intended economic models. Key areas of focus include the integrity of the rebase function, the proper handling of stake and unstake operations, the transparency of error handling, 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