Cyberscope

Audit Report

GIVR BEAR

June 2024

Network	BASE
Address	0x97Ee603f51Fb9d180a4077bbae9c8d4dD4B3DE41
Audited by	© cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Passed
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CR	Code Repetition	Unresolved
	DDP	Decimal Division Precision	Unresolved
	EIS	Excessively Integer Size	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	MTEE	Missing Transfer Event Emission	Unresolved
	RFEA	Redundant Fee Exclusion Array	Unresolved
	RNM	Redundant Non-Reentrant Modifier	Unresolved
	RRF	Redundant Reward Functionality	Unresolved
	UTF	Unused Transfer Functionality	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L09	Dead Code Elimination	Unresolved
	L13	Divide before Multiply Operation	Unresolved
	L16	Validate Variable Setters	Unresolved
	L18	Multiple Pragma Directives	Unresolved

●	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	GIVR
Testing Deploy	https://basescan.org/address/0x97ee603f51fb9d180a4077bbae9c8d4dd4b3de41
Address	0x97Ee603f51Fb9d180a4077bbae9c8d4dD4B3DE41
Network	BASE
Symbol	GIVR
Decimals	18
Total Supply	280,000,000,000,000
Badge Eligibility	Yes

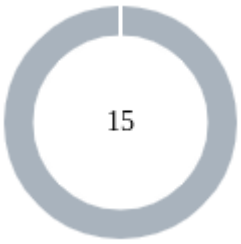
Audit Updates

Initial Audit	21 May 2024 https://github.com/cyberscope-io/audits/blob/main/givr/v1/audit.pdf
Corrected Phase 2	05 Jun 2024

Source Files

Filename	SHA256
contracts/GIVR.sol	91784c79411721fa77340775ac36c5d2472e2d45ffe86704fad5c462c4a4c9ff

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

CR - Code Repetition

Criticality	Minor / Informative
Location	GIVR.sol#L1122,1181
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 `_transfer` and `_transferFrom` share similar code segments.

```
   function _transfer(address from, address to, uint256 amount)
internal {
        require(from != address(0), "ERC20: transfer from the zero
address");
        require(to != address(0), "ERC20: transfer to the zero
address");
        uint256 senderBalance = balances[from];
        require(senderBalance >= amount, "ERC20: insufficient
balance");


        uint256 amountAfterFee = amount;
        if (!_isExcludedFromFee[from] && !_isExcludedFromFee[to]) {
            uint256 totalTaxRate = 350;
            uint256 fee = amount * totalTaxRate / 10000;
            amountAfterFee = amount - fee;

            uint256 liquidityFee = fee * liquidityTaxRate /
totalTaxRate;
            uint256 maintenanceFee = fee * maintenanceTaxRate /
totalTaxRate;
            uint256 charityFee = fee * charityTaxRate / totalTaxRate;
            uint256 burnFee = fee * burnTaxRate / totalTaxRate;
            uint256 developerFee = fee * developerTaxRate /
totalTaxRate; // Each developer gets an equal share

            balances[liquidityWallet] += liquidityFee;
            balances[maintenanceWallet] += maintenanceFee;
            balances[charityWallet] += charityFee;
            balances[burnWallet] += burnFee;
            balances[developerWallet1] += developerFee;
            balances[developerWallet2] += developerFee;
            balances[developerWallet3] += developerFee;
            balances[developerWallet4] += developerFee;

            emit Transfer(from, liquidityWallet, liquidityFee);
            emit Transfer(from, maintenanceWallet, maintenanceFee);
            emit Transfer(from, charityWallet, charityFee);
            emit Transfer(from, burnWallet, burnFee);
            emit Transfer(from, developerWallet1, developerFee);
            emit Transfer(from, developerWallet2, developerFee);
            emit Transfer(from, developerWallet3, developerFee);
            emit Transfer(from, developerWallet4, developerFee);
            emit TaxDistributed(developerWallet1, developerFee);
            emit TaxDistributed(developerWallet2, developerFee);
            emit TaxDistributed(developerWallet3, developerFee);
            emit TaxDistributed(developerWallet4, developerFee);
        }

    function _transferFrom(address from, address to, uint256 amount)
```

```
internal {
        require(to != address(0), "ERC20: transfer to the zero
address");
        uint256 senderBalance = balances[from];
        require(senderBalance >= amount, "ERC20: insufficient
balance");
        ...
        }
```

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	GIVR.sol#L1135,1194
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
uint256 liquidityFee = (fee * liquidityTaxRate) / totalTaxRate;
uint256 maintenanceFee = (fee * maintenanceTaxRate) / totalTaxRate;
uint256 charityFee = (fee * charityTaxRate) / totalTaxRate;
uint256 burnFee = (fee * burnTaxRate) / totalTaxRate;
uint256 developerFee = (fee * developerTaxRate) / totalTaxRate; // Each
developer gets an equal share
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	GIVR.sol#L1075
Status	Unresolved

Description

The contract is using a bigger unsigned integer data type that the maximum size that is required. By using an unsigned integer data type larger than necessary, the smart contract consumes more storage space and requires additional computational resources for calculations and operations involving these variables. This can result in higher transaction costs, longer execution times, and potential scalability bottlenecks.

The contract is currently using `uint256` variables to declare and set the tax rate variables. However, since the maximum tax rate is 100, it would be more efficient to use `uint8` instead of `uint256`. This is because `uint8` can store values from 0 to 255, which is sufficient for the intended range of 0 to 100.

```
uint256 public liquidityTaxRate = 100; // Liquidity tax rate of 1%
uint256 public maintenanceTaxRate = 50; // Maintenance tax rate of 0.5%
uint256 public charityTaxRate = 75; // Charity tax rate of 0.75%
uint256 public burnTaxRate = 25; // Burn tax rate of 0.25%
uint256 public developerTaxRate = 25; // Developer tax rate of 0.25%
for each developer
```

Recommendation

To address the inefficiency associated with using an oversized unsigned integer data type, it is recommended to accurately determine the required size based on the range of values the variable needs to represent.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	GIVR.sol#L1105,1106,1107,1108
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.

```
developerWallet1
developerWallet2
developerWallet3
developerWallet4
```

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	GIVR.sol#L1110
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.

```
balances[msg.sender] = totalSupply;
```

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.

RFEA - Redundant Fee Exclusion Array

Criticality	Minor / Informative
Location	GIVR.sol#L1268,1275
Status	Unresolved

Description

The contract is currently utilizing both the `_isExcludedFromFee` mapping and the `_excludedFromFee` array to manage addresses that are excluded from fee calculations. The `_isExcludedFromFee` mapping directly fulfills the need for tracking whether an address is exempt from fees, allowing for efficient O(1) look-up times. However, the contract also maintains the `_excludedFromFee` array, which duplicates the functionality of the mapping by listing excluded addresses. This redundancy is unnecessary and increases the complexity of the code, as well as the potential for errors and higher gas costs associated with maintaining two separate data structures for the same purpose.

```
    function excludeFromFee(address account) external onlyOwner {
        if (!_isExcludedFromFee[account]) {
            _isExcludedFromFee[account] = true;
            _excludedFromFee.push(account);
        }
    }

    function includeInFee(address account) external onlyOwner {
        if (_isExcludedFromFee[account]) {
            _isExcludedFromFee[account] = false;
            for (uint256 i = 0; i < _excludedFromFee.length; i++) {
                if (_excludedFromFee[i] == account) {
                    _excludedFromFee[i] =
_excludedFromFee[_excludedFromFee.length - 1];
                    _excludedFromFee.pop();
                    break;
                }
            }
        }
    }
```

Recommendation

It is recommended to remove the `_excludedFromFee` array from the contract since the exclusion from fee functionality is effectively accomplished by the `_isExcludedFromFee` mapping. This simplification will reduce the contract's gas usage during transactions that involve updating fee exclusions and will decrease the overall complexity of the code, thereby reducing the likelihood of errors. Further, this change will streamline state management within the contract by eliminating redundant data storage.

RNM - Redundant Non-Reentrant Modifier

Criticality	Minor / Informative
Location	GIVR.sol#L1117,1171,1228,1239,1248
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract is currently utilizing the `nonReentrant` modifier across several functions. This modifier is typically used to prevent reentrancy attacks where external calls in a function could lead to unexpected behaviors by re-entering the function. However, there are no external calls within these functions that might lead to reentrancy issues. Therefore, the use of the `nonReentrant` modifier in these contexts is redundant and does not contribute to security but adds unnecessary complexity and gas costs to the transactions.

```
   function transfer(address to, uint256 amount) external override
nonReentrant returns (bool) {
        _transfer(msg.sender, to, amount);
        return true;
    }

    function transferFrom(address from, address to, uint256 amount)
external override nonReentrant returns (bool) {
    ...
    }

    function approve(
        address spender,
        uint256 amount
    ) external override nonReentrant returns (bool) {
    ...
    }

    function increaseAllowance(
        address spender,
        uint256 addedValue
    ) external nonReentrant returns (bool) {
        ...
    }

    function decreaseAllowance(
        address spender,
        uint256 subtractedValue
    ) external nonReentrant returns (bool) {
    ...
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it. It is recommended to remove the `nonReentrant` modifier from the functions where it is currently applied, as there are no reentrancy risks present within their implementations. This change will simplify the contract's code and reduce the gas consumption for these functions without compromising the contract's security.

RRF - Redundant Reward Functionality

Criticality	Minor / Informative
Location	GIVR.sol#L1288,1295
Status	Unresolved

Description

The contract includes the functions `excludeFromReward` and `includeInReward`, which are designed to manage the inclusion and exclusion of accounts from receiving rewards. However, the contract lacks any actual mechanisms or functionality for distributing rewards. This discrepancy indicates that the aforementioned functions are redundant as there are no reward distributions being controlled by these functionalities. The presence of these functions without associated reward mechanisms unnecessarily increases the complexity and size of the contract, which may lead to confusion and mismanagement.

```
    function excludeFromReward(address account) external onlyOwner {
        if (!_isExcludedFromReward[account]) {
            _isExcludedFromReward[account] = true;
            _excludedFromReward.push(account);
        }
    }

    function includeInReward(address account) external onlyOwner {
        if (_isExcludedFromReward[account]) {
            _isExcludedFromReward[account] = false;
            for (uint256 i = 0; i < _excludedFromReward.length; i++) {
                if (_excludedFromReward[i] == account) {
                    _excludedFromReward[i] =
_excludedFromReward[_excludedFromReward.length - 1];
                    _excludedFromReward.pop();
                    break;
                }
            }
        }
    }
```

Recommendation

It is recommended to remove the `excludeFromReward` and `includeInReward` functions from the contract since they serve no purpose in the absence of a reward distribution mechanism. Eliminating these functions will streamline the contract, reduce unnecessary overhead, and clarify the actual functionalities being provided, thereby optimizing both the performance and maintainability of the contract.

UTF - Unused Transfer Functionality

Criticality	Minor / Informative
Location	GIVR.sol#L1181
Status	Unresolved

Description

The contract is designed with an internal `_transferFrom` function intended to handle transfers of tokens between addresses along with the application of transaction fees. Despite its comprehensive setup, which includes calculating and distributing multiple fees to specific wallets, this function is not called anywhere within the contract's other functions. Additionally, the contract does not implement any allowance mechanism typically associated with ERC20 standard transfers, which is essential for managing and verifying third-party transfers. This oversight suggests that the `_transferFrom` function, is redundant as its functionality is not integrated into the contract's operational framework.

```
    function _transferFrom(address from, address to, uint256 amount)
internal {
        ...
        emit Transfer(from, to, amountAfterFee);
    }
```

Recommendation

It is recommended to remove the internal `_transferFrom` function from the contract if no future features will utilize it. This removal will help in reducing the contract's complexity and gas costs associated with maintaining unused code. Furthermore, if the intention is to adhere to the ERC20 standard practices, it is advised to implement an allowance mechanism that supports secure third-party transfers, ensuring the contract's compliance and functionality are aligned with common token standards.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	GIVR.sol#L1048,1049,1050,1074,1075,1076,1077,1078
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.

```
public name = "GIVR BEAR";
public symbol = "GIVR";
private _decimals = 18;
uint256 public liquidityTaxRate = 100;
uint256 public maintenanceTaxRate = 50;
uint256 public charityTaxRate = 75;
uint256 public burnTaxRate = 25;
uint256 public developerTaxRate = 25;
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	GIVR.sol#L80,448,951,966,1181
Status	Unresolved

Description

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

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

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

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

function _contextSuffixLength() internal view virtual returns (uint256)
{
        return 0;
    }
}

//

...
```

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	GIVR.sol#L1132,1135,1136,1137,1138,1139,1190,1193,1194,1195,1196,1197
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 fee = amount * totalTaxRate / 10000;
uint256 burnFee = fee * burnTaxRate / totalTaxRate;
```

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	GIVR.sol#L591
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.

```
_pendingOwner = newOwner;
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	GIVR.sol#L10,94,115,214,262,427,458,560,621,703,731,1045
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 >=0.5.0;
pragma solidity >=0.6.2;
pragma solidity ^0.8.19;
pragma solidity ^0.8.20;
...
```

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	GIVR.sol#L10,262,427,458,560,621,703,731,1045
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 ^0.8.19;
...
```

Recommendation

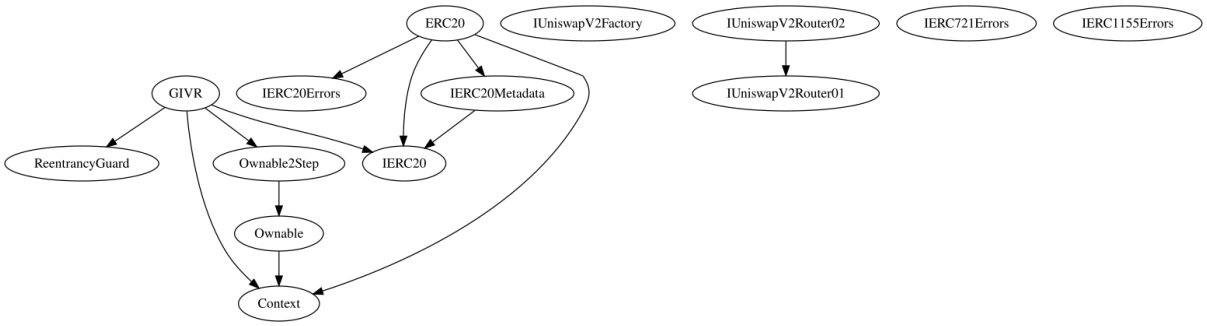
The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest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

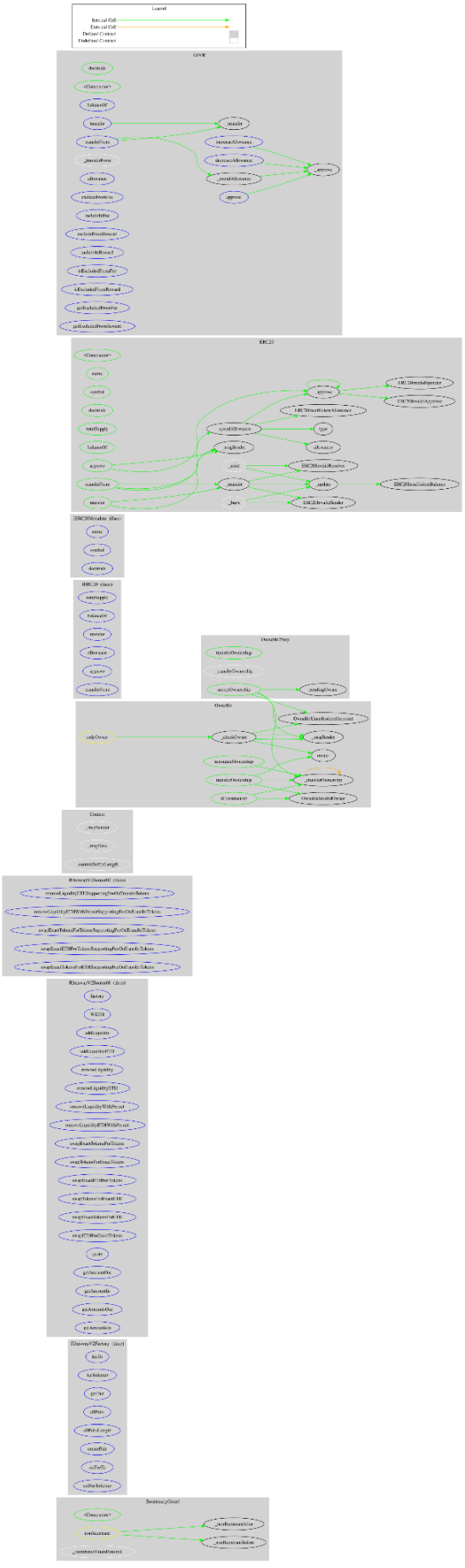
Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
GIVR	Implementation	Context, IERC20, Ownable2Step, ReentrancyGuard		
	decimals	Public		-
		Public	✓	Ownable
	balanceOf	External		-
	transfer	External	✓	nonReentrant
	_transfer	Internal	✓	
	transferFrom	External	✓	nonReentrant
	_spendAllowance	Internal	✓	
	_transferFrom	Internal	✓	
	approve	External	✓	nonReentrant
	increaseAllowance	External	✓	nonReentrant
	decreaseAllowance	External	✓	nonReentrant
	_approve	Internal	✓	
	allowance	External		-
	excludeFromFee	External	✓	onlyOwner
	includeInFee	External	✓	onlyOwner
	excludeFromReward	External	✓	onlyOwner

	includeInReward	External	✓	onlyOwner
	isExcludedFromFee	External		-
	isExcludedFromReward	External		-
	getExcludedFromFee	External		-
	getExcludedFromReward	External		-

Inheritance Graph



Flow Graph



Summary

GIVR BEAR contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. The audit revealed one critical issue. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. The fee is set at 3.5%.

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