Cyberscope

Audit Report

GIVR BEAR

May 2024

SHA256 1cf3c31b0e38d32bc0f2b0fcad165959be74d9486de82331c901b8cd66e7e0e1

Audited by © cyberscope

Analysis

Critical Medium Minor / Informative Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Passed
●	OTUT	Transfers User's Tokens	Unresolved
●	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
Medium	UPA	Unexcluded Pinksale Address	Unresolved
Minor	CR	Code Repetition	Unresolved
Minor	DDP	Decimal Division Precision	Unresolved
Minor	EIS	Excessively Integer Size	Unresolved
Minor	IDI	Immutable Declaration Improvement	Unresolved
Minor	MTEE	Missing Transfer Event Emission	Unresolved
Minor	L02	State Variables could be Declared Constant	Unresolved
Minor	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor	L09	Dead Code Elimination	Unresolved
Minor	L16	Validate Variable Setters	Unresolved
Minor	L18	Multiple Pragma Directives	Unresolved
Minor	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Contract Name	GIVR
Testing Deploy	https://testnet.bscscan.com/address/0xb20d9f9b135ecf19403d18c1d00d880dc1426b77
Symbol	GIVR
Decimals	18
Total Supply	280,000,000,000,000
Badge Eligibility	Must Fix Criticals

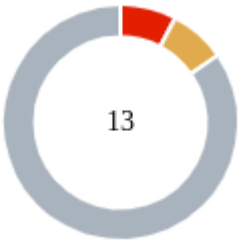
Audit Updates

Initial Audit	21 May 2024

Source Files

Filename	SHA256
contracts/GIVR.sol	1cf3c31b0e38d32bc0f2b0fcad165959be74d9486de82331c901b8cd66e7e0e1

Findings Breakdown



Critical	1
Medium	1
Minor / Informative	11

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	11	0	0	0

OTUT - Transfers User's Tokens

Criticality	Critical
Status	Unresolved

Description

The contract includes the `transferFrom` function which does not implement an allowance mechanism. As a result, any user has the authority to transfer tokens from another account without the account holder's consent. This vulnerability can lead to unauthorized transfers and potential loss of funds for the users of the contract.

```
    function transferFrom(
        address from,
        address to,
        uint256 amount
    ) external override returns (bool) {
        _transferFrom(from, to, amount);
        return true;
    }

    function _transferFrom(address from, address to, uint256
amount) internal {
    ...
    }
```

Recommendation

It is recommended to implement an allowance mechanism to prevent unauthorized transfers. This can be achieved by incorporating checks within the `transferFrom` function to ensure that the spender is allowed to transfer the specified amount of tokens on behalf of the token owner. This will safeguard against unauthorized transfers and enhance the security of the contract.

UPA - Unexcluded Pinksale Address

Criticality	Medium
Location	contracts/GIVR.sol#L117\0,1222
Status	Unresolved

Description

The contract incorporates operational restrictions on transactions, which can hinder seamless interaction with decentralized applications (dApps) such as launchpads, presales, lockers, or staking platforms. In scenarios where an external contract, such as a launchpad factory, needs to integrate with the contract, it should be exempt from the limitations to ensure uninterrupted service and functionality. Failure to provide such exemptions can block the successful process and operation of services reliant on this contract.

```
   function _transfer(address from, address to, uint256 amount)
internal {
        ...
        uint256 totalTaxRate = 350; // Corrected total tax rate
calculation
        uint256 fee = (amount * totalTaxRate) / 10000; //
Corrected fee calculation to match the total tax rate
        uint256 amountAfterFee = amount - fee;
        // Distribute fees
        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[from] = senderBalance - amount;
        balances[to] += amountAfterFee;
        ...
    }

    function _transferFrom(address from, address to, uint256
amount) internal {
        ...
        uint256 totalTaxRate = 350; // Corrected total tax rate
calculation
        uint256 fee = (amount * totalTaxRate) / 10000; //
Corrected fee calculation to match the total tax rate
        uint256 amountAfterFee = amount - fee;

        // Distribute fees
        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[from] = senderBalance - amount;
        balances[to] += amountAfterFee;
        ...
    }
```

Recommendation

It is advisable to modify the contract by incorporating functionality that enables the exclusion of designated addresses from transactional restrictions. This enhancement will allow specific addresses, such as those associated with decentralized applications (dApps) and service platforms, to operate without being hindered by the standard constraints imposed on other users. Implementing this feature will ensure smoother integration and functionality with external systems, thereby expanding the contract's versatility and effectiveness in diverse operational environments.

CR - Code Repetition

Criticality	Minor / Informative
Location	contracts/GIVR.sol#L1170,1222
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(to != address(0), "ERC20: transfer to the zero address");
        uint256 senderBalance = balances[from];
        require(senderBalance >= amount, "ERC20: insufficient balance");

        uint256 totalTaxRate = 350; // Corrected total tax rate
calculation
        uint256 fee = (amount * totalTaxRate) / 10000; // Corrected fee
calculation to match the total tax rate
        uint256 amountAfterFee = amount - fee;

        // Distribute fees
        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;

        balances[from] = senderBalance - amount;
        balances[to] += amountAfterFee;

        emit Transfer(from, to, amountAfterFee);
        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	contracts/GIVR.sol#L1180,1232
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	contracts/GIVR.sol#L1099
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	contracts/GIVR.sol#L1148,1149,1150,1151
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	contracts/GIVR.sol#L1153
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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	contracts/GIVR.sol#L1077,1078,1079,1099,1100,1101,1102,1103
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;
public liquidityTaxRate = 100;
public maintenanceTaxRate = 50;
public charityTaxRate = 75;
public burnTaxRate = 25;
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.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	contracts/GIVR.sol#L48
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.

```
function WETH() external pure returns (address);
```

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.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	contracts/GIVR.sol#L446,967,982
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 _contextSuffixLength() internal view virtual returns
(uint256) {
        return 0;
    }
}

//
...
        if (account == address(0)) {
            revert ERC20InvalidReceiver(address(0));
        }
        _update(address(0), account, value);
    }

    ...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	contracts/GIVR.sol#L593
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	contracts/GIVR.sol#L8,43,196,248,425,455,558,622,714,740,1074
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	contracts/GIVR.sol#L248,425,455,558,622,714,740,1074
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 ^0.8.20;
...
```

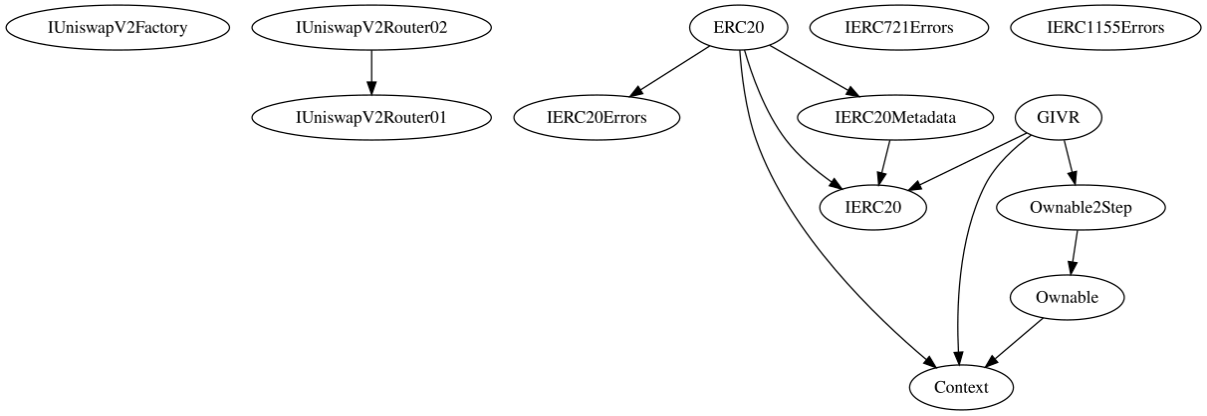
Recommendation

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.

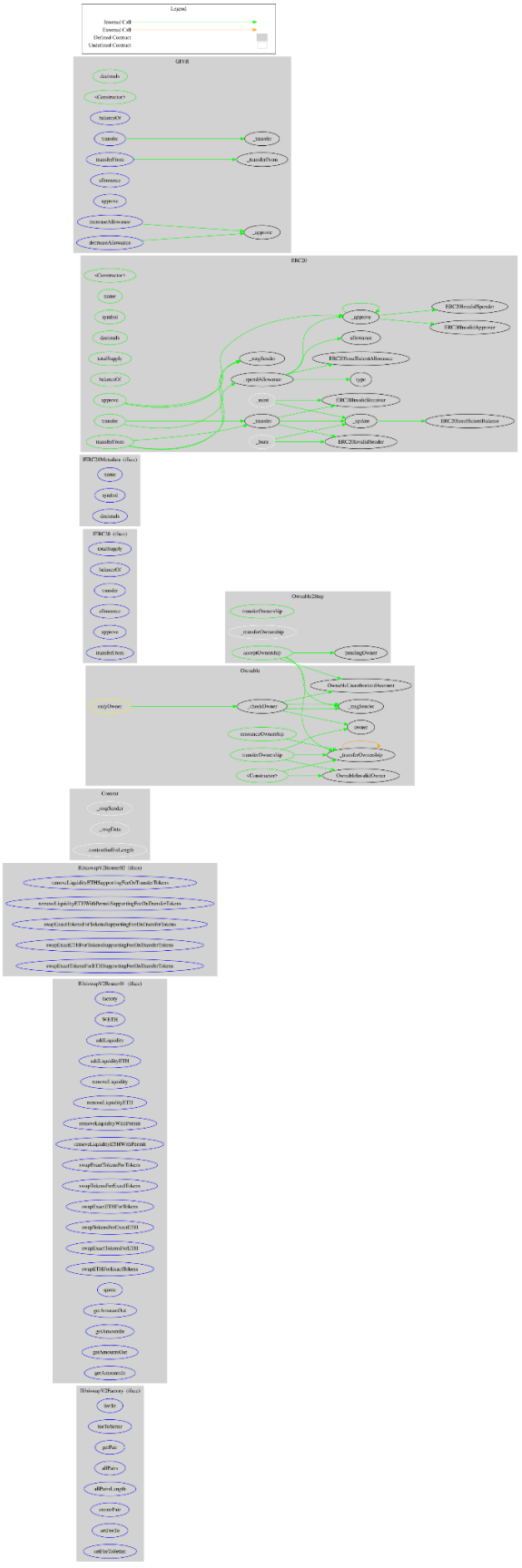
Functions Analysis

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

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
GIVR
decimals
<Constructor>
balanceOf
transfer
_transfer
transferFrom
_transferFrom
allowance
approve
_approve
increaseAllowance
decreaseAllowance
ERC20
<Constructor>
name
symbol
decimals
totalSupply
balanceOf
_msgSender
spendAllowance
approve
_approve
ERC20InvalidSpender
ERC20InvalidApprover
allowance
ERC20InsufficientAllowance
type
transfer
_transfer
mint
_update
ERC20InvalidReceiver
ERC20InsufficientBalance
transferFrom
_burn
ERC20InvalidSender
IERC20Metadata (iface)
name
symbol
decimals
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Ownable2Step
transferOwnership
_transferOwnership
acceptOwnership
pendingOwner
Ownable
OwnableUnauthorizedAccount
onlyOwner
_checkOwner
_msgSender
owner
renounceOwnership
transferOwnership
_transferOwnership
<Constructor>
OwnableInvalidOwner
Context
_msgSender
_msgData
contextSuffixLength
IUniswapV2Router02 (iface)
removeLiquidityETHSupportingFeeOnTransferTokens
removeLiquidityETHWithPermitSupportingFeeOnTransferTokens
swapExactTokensForTokensSupportingFeeOnTransferTokens
swapExactETHForTokensSupportingFeeOnTransferTokens
swapExactTokensForETHSupportingFeeOnTransferTokens
IUniswapV2Router01 (iface)
factory
WETH
addLiquidity
addLiquidityETH
removeLiquidity
removeLiquidityETH
removeLiquidityWithPermit
removeLiquidityETHWithPermit
swapExactTokensForTokens
swapTokensForExactTokens
swapExactETHForTokens
swapTokensForExactETH
swapExactTokensForETH
swapETHForExactTokens
quote
getAmountOut
getAmountIn
getAmountsOut
getAmountsIn
IUniswapV2Factory (iface)
feeTo
feeToSetter
getPair
allPairs
allPairsLength
createPair
setFeeTo
setFeeToSetter

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