Cyberscope

Audit Report

Genesis

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	19 Jun 2024

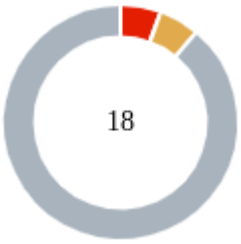
Source Files

Filename	SHA256
GenesisDAO.sol	96195b72608117aad05d03957957500e41d386cb8fb9cedd11c3d35d23f30e44

Overview

GenesisDAO is an ERC20 token designed for decentralized governance and finance management. It incorporates multiple functionalities such as ownership management through the Ownable contract and reentrancy protection via ReentrancyGuard. The token has built-in taxation mechanisms for buy and sell transactions, with respective taxes of up to 5% and 20%. A portion of the token tax is sent directly to the treasury address, while the remaining amount is swapped for native tokens (ETH) and sent to the treasury. The token contract integrates with the Uniswap V2 protocol for liquidity management, including automatic liquidity pool creation and tax liquidation. Additionally, the contract allows for the exclusion of certain addresses from taxes, emergency withdrawals, and includes functions for minting and burning tokens.

Findings Breakdown



Critical	1
Medium	1
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	16	0	0	0

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	PRR	Potential Reentrance Revert	Unresolved
	MT	Mints Tokens	Unresolved
	EIS	Excessively Integer Size	Unresolved
	IDI	Immutable Declaration Improvement	Unresolved
	MEM	Missing Error Messages	Unresolved
	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
	PVC	Price Volatility Concern	Unresolved
	RCC	Redundant Conditional Check	Unresolved
	RRA	Redundant Repeated Approvals	Unresolved
	OCTD	Transfers Contract's Tokens	Unresolved
	L02	State Variables could be Declared Constant	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L07	Missing Events Arithmetic	Unresolved
	L13	Divide before Multiply Operation	Unresolved

●	L15	Local Scope Variable Shadowing	Unresolved
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

PRR - Potential Reentrance Revert

Criticality	Critical
Location	GenesisDAO.sol#L130
Status	Unresolved

Description

The contract is using the `nonReentrant` modifier in the swap functionality. During the `swapExactTokensForETH` operation, the router will execute the transfer method as a recursive call. During the re-entrance phase, if the router address is not whitelisted, the transfer may revert since the `nonReentrant` guard will be triggered. As a result, this can cause the swap to fail, potentially leading to unexpected behavior or loss of funds during the token liquidation process.

```
    function liquidateTokenTax() internal nonReentrant
     {
          uint tokenAmount = balanceOf(address(this));
          if (tokenAmount > 0)
          {
               address[] memory path = new address[](2);
               path[0] = address(this);
               path[1] = router.WETH();
               _approve(address(this), address(_routerAddr), tokenAmount);

               router.swapExactTokensForETH(
               tokenAmount,
               0,
               path,
               treasury,
               block.timestamp
               );
          }
     }
```

Recommendation

It is recommended to modify the `reentrance` guard to prevent the swap during the recursive execution of the `swapExactTokensForETH` method. The team should ensure

that the contract can handle re-entrance scenarios properly, possibly by implementing a whitelist or role-based mechanism to allow certain trusted addresses to bypass the reentrancy guard during such operations.

MT - Mints Tokens

Criticality	Medium
Location	GenesisDAO.sol#L160,163
Status	Unresolved

Description

The contract owner has the authority to set any address as the `stakingContract`. This means that the `stakingContract` will have the authority to mint tokens. The owner of the `stakingContract` address may take advantage of it by calling the `mint` function. As a result, the contract tokens will be highly inflated.

```
    function mint(address to, uint256 amount) external {
        require(msg.sender == stakingContract);
        _mint(to, amount);
    }

    function setStakingContract(address stakingContract_) public
onlyOwner {
        stakingContract = stakingContract_;
    }
```

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, while setting the correct `stakingContract` address, which will eliminate the threats but it is non-reversible.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	GenesisDAO.sol#L21
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.

Specifically, the variables `buyTax`, `sellTax`, and `taxLiquidatePercentage` could be set to a `uint8` type since their value do not exceed 255.

```
uint256 public buyTax = 2;
uint256 public sellTax = 5;
uint256 public taxLiquidatePercentage = 40;
```

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	GenesisDAO.sol#L33,37,38
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.

```
treasury
_routerAddr
_pairAddress
```

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.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	GenesisDAO.sol#L161
Status	Unresolved

Description

The contract is missing error messages. Specifically, there are no error messages to accurately reflect the problem, making it difficult to identify and fix the issue. As a result, the users will not be able to find the root cause of the error.

```
require(msg.sender == stakingContract)
```

Recommendation

The team is suggested to provide a descriptive message to the errors. This message can be used to provide additional context about the error that occurred or to explain why the contract execution was halted. This can be useful for debugging and for providing more information to users that interact with the contract.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	GenesisDAO.sol#L135
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
address[] memory path = new address[](2);
path[0] = address(this);
path[1] = router.WETH();
_approve(address(this), address(_routerAddr), tokenAmount);

router.swapExactTokensForETH(
tokenAmount,
0,
path,
treasury,
block.timestamp
);
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	GenesisDAO.sol#L130
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `tokenAmount` sets the amount of the tokens for the swap. However if the swap amount is a big number, then the contract will swap a huge amount of tokens for ETH.

It is important to note that the price of the token representing it, can be highly volatile. This means that the value of a price volatility swap involving Ether could fluctuate significantly at the triggered point, potentially leading to significant price volatility for the parties involved.

```
    function liquidateTokenTax() internal nonReentrant
     {
         uint tokenAmount = balanceOf(address(this));
         if (tokenAmount > 0)
         {
             address[] memory path = new address[](2);
             path[0] = address(this);
             path[1] = router.WETH();
             _approve(address(this), address(_routerAddr), tokenAmount);

             router.swapExactTokensForETH(
             tokenAmount,
             0,
             path,
             treasury,
             block.timestamp
             );
         }
     }
```

Recommendation

The contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RCC - Redundant Conditional Check

Criticality	Minor / Informative
Location	GenesisDAO.sol#L115
Status	Unresolved

Description

The contract is performing a conditional check `if (ethTax > 0)` after calculating `ethTax` as the remainder of taxes after subtracting tokenTax. Since `taxLiquidatePercentage` can be a maximum of `40%`, `ethTax` will always be positive, making the condition always `true`. This makes the check redundant.

```
uint256 tokenTax = (taxes * (100 - taxLiquidatePercentage)) / 100;
super._update(from, treasury, tokenTax);

// send remainder to token contract.
uint256 ethTax = taxes - tokenTax;
if (ethTax > 0)
{
    super._update(from, address(this), ethTax);
}
```

Recommendation

It is recommended to remove the redundant `if (ethTax > 0)` check to streamline the code and improve its efficiency, ensuring that unnecessary conditions are not evaluated.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	GenesisDAO.sol#L138
Status	Unresolved

Description

The contract is designed to approve token transfers during the contract's operation by calling the `_approve` function before specific operations. This approach results in additional gas costs since the approval process is repeated for every operation execution, leading to inefficiencies and increased transaction expenses.

```
address[] memory path = new address[](2);
path[0] = address(this);
path[1] = router.WETH();

_approve(address(this), address(_routerAddr), tokenAmount);

router.swapExactTokensForETH(
tokenAmount,
0,
path,
treasury,
block.timestamp
```

Recommendation

It is recommended to optimize the contract by approving the token amount once, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

OCTD - Transfers Contract's Tokens

Criticality	Minor / Informative
Location	GenesisDAO.sol#L179
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 `emergencyWithdraw` function.

```
    function emergencyWithdraw(address token, address to, uint256 amount)
public onlyOwner {
        IERC20(token).transfer(to, 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.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	GenesisDAO.sol#L13
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.

```
address public treasury
```

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	GenesisDAO.sol#L56,61,66,71
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.

```
uint256 _buyTax
uint256 _sellTax
uint256 _taxLiqPercentage
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.
Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	GenesisDAO.sol#L58,63,68,74,157,176
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
buyTax = _buyTax
sellTax = _sellTax
taxLiquidatePercentage = _taxLiqPercentage
_liquidityPools[pair] = isLiquidityPool;
_isExcludedFromTaxes[account] = excluded;
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	GenesisDAO.sol#L100,110
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.

```
taxes = (amount * buyTax) / 100
uint256 tokenTax = (taxes * (100 - taxLiquidatePercentage)) / 100
```

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.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	GenesisDAO.sol#L29
Status	Unresolved

Description

Local scope variable shadowing occurs when a local variable with the same name as a variable in an outer scope is declared within a function or code block. When this happens, the local variable "shadows" the outer variable, meaning that it takes precedence over the outer variable within the scope in which it is declared.

```
address treasury
```

Recommendation

It's important to be aware of shadowing when working with local variables, as it can lead to confusion and unintended consequences if not used correctly. It's generally a good idea to choose unique names for local variables to avoid shadowing outer variables and causing confusion.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	GenesisDAO.sol#L37,166,184
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.

```
_routerAddr = uniswapRouterAddress
stakingContract = stakingContract_
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	GenesisDAO.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 ^0.8.20;
```

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.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	GenesisDAO.sol#L180
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(token).transfer(to, amount)
```

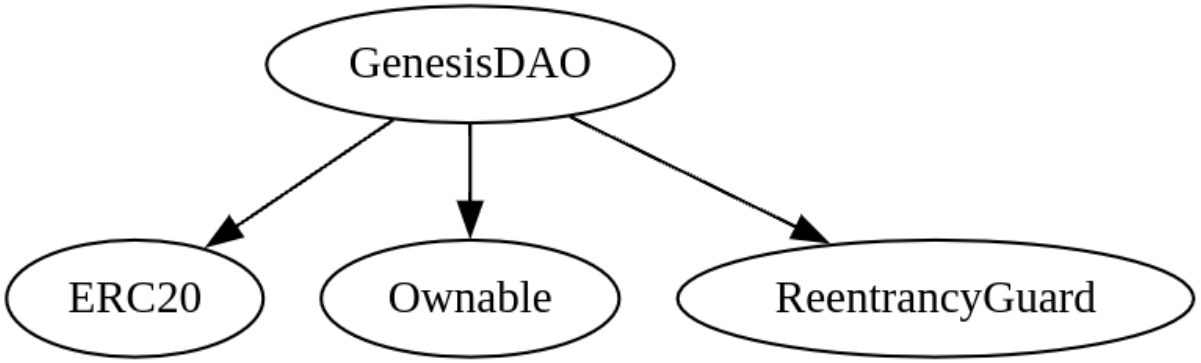
Recommendation

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.

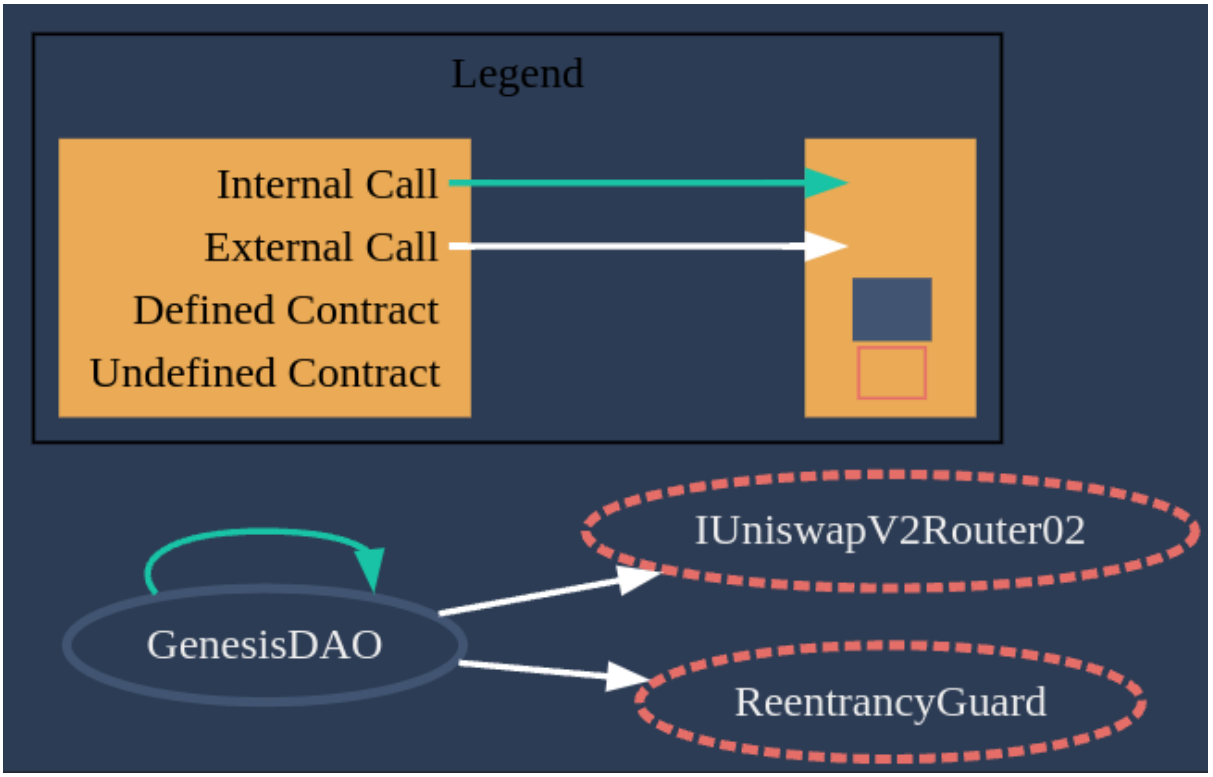
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
GenesisDAO	Implementation	ERC20, Ownable, ReentrancyGuard		
		External	Payable	-
		Public	✓	ERC20 Ownable
	updateBuyTax	External	✓	onlyOwner
	updateSellTax	External	✓	onlyOwner
	updateTaxLiquidationPercentage	External	✓	onlyOwner
	updateTaxes	External	✓	onlyOwner
	_update	Internal	✓	
	liquidateTokenTax	Internal	✓	nonReentrant
	setLiquidityPool	Public	✓	onlyOwner
	_setLiquidityPool	Private	✓	
	mint	External	✓	-
	setStakingContract	Public	✓	onlyOwner
	burn	Public	✓	-
	excludeFromTaxes	Public	✓	onlyOwner
	emergencyWithdraw	Public	✓	onlyOwner
	emergencyEthWithdraw	Public	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
IUniswapV2Router02
GenesisDAO
ReentrancyGuard

Summary

Genesis contract implements a token with built-in taxation and liquidity management mechanisms. 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