Cyberscope

Audit Report

BlaroThings

January 2025

SHA256 f0262e99c8deca3c4ada1ab0442d5a892845fd569411bf729fdb8b369daa6b51

Audited by © cyberscope

Table of Contents

Table of Contents 1
Risk Classification 3
Review 4
Audit Updates 4
Source Files 4
Overview 5
Findings Breakdown 6
Diagnostics 7
CCR - Contract Centralization Risk 8
Description 8
Recommendation 9
EIS - Excessively Integer Size 10
Description 10
Recommendation 10
IRC - Incorrect Reward Calculation 11
Description 11
Recommendation 12
MTSC - Misleading Total Stakers Count 13
Description 13
Recommendation 13
MEE - Missing Events Emission 14
Description 14
Recommendation 14
PTAI - Potential Transfer Amount Inconsistency 15
Description 15
Recommendation 16
L04 - Conformance to Solidity Naming Conventions 17
Description 17
Recommendation 18
L09 - Dead Code Elimination 19
Description 19
Recommendation 19
L13 - Divide before Multiply Operation 20
Description 20
Recommendation 20
L18 - Multiple Pragma Directives 21
Description 21
Recommendation 21
L19 - Stable Compiler Version 22

Description 22
Recommendation 22
L20 - Succeeded Transfer Check 23
Description 23
Recommendation 23
Functions Analysis **24**
Inheritance Graph **25**
Flow Graph **26**
Summary **27**
Disclaimer **28**
About Cyberscope **29**

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Audit Updates

Initial Audit	13 Jan 2025
Corrected Phase 2	23 Jan 2025

Source Files

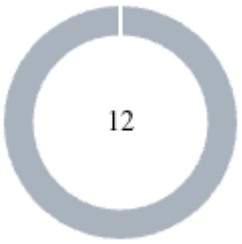
Filename	SHA256
staking.sol	f0262e99c8deca3c4ada1ab0442d5a892845fd569411bf729fdb8b369daa6b51

Overview

The BLRStaking contract is a staking platform designed for the BLR token. It enables users to stake their BLR tokens into predefined pools with varying lock-up durations and associated annual percentage yields (APYs). The pools range from 30 days to 1095 days, each offering different rewards and multipliers. Users can opt for either locked or unlocked staking, where locked staking provides higher rewards through multipliers but restricts withdrawals until the lock-up period ends. The contract employs mechanisms to calculate pending rewards based on the staked amount, pool-specific APY, and time elapsed since staking. Additionally, early withdrawals for locked stakes are subject to a penalty.

The contract uses OpenZeppelin's Ownable and ReentrancyGuard contracts for secure and efficient management. The onlyOwner modifier restricts administrative functions such as toggling staking and withdrawing extra tokens to the contract owner. It incorporates non-reentrant modifiers to prevent vulnerabilities like reentrancy attacks. Furthermore, the contract allows users to claim rewards, withdraw their stake, and view their staking history. The withdrawExtraTokens function ensures that surplus tokens beyond the current staked amount can be securely retrieved by the owner, maintaining a clean contract balance.

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	12

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
	CCR	Contract Centralization Risk	Unresolved
	EIS	Excessively Integer Size	Unresolved
	IRC	Incorrect Reward Calculation	Unresolved
	MTSC	Misleading Total Stakers Count	Unresolved
	MEE	Missing Events Emission	Unresolved
	PTAI	Potential Transfer Amount Inconsistency	Unresolved
	L04	Conformance to Solidity Naming Conventions	Unresolved
	L09	Dead Code Elimination	Unresolved
	L13	Divide before Multiply Operation	Unresolved
	L18	Multiple Pragma Directives	Unresolved
	L19	Stable Compiler Version	Unresolved
	L20	Succeeded Transfer Check	Unresolved

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	staking.sol#L212
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 contract does not ensure that it holds sufficient tokens to cover all users’ staked amounts and their accumulated rewards. This creates a risk where users may be unable to claim their full rewards or even their initial staked amounts if the contract’s balance is insufficient. It is implicitly the owner’s responsibility to ensure the contract is adequately funded, but there is no mechanism to enforce or verify this requirement, exposing users to potential losses.

```
function claim(uint256 orderId) public nonReentrant {
    require(orderId <= latestOrderId, "BLRStaking: INVALID orderId");

    OrderInfo storage order = orders[orderId];
    require(order.beneficiary == msg.sender, "BLRStaking: Not your order");
    require(!order.claimed, "BLRStaking: Already claimed");
    if (order.locked) {
        require(block.timestamp >= order.endtime, "BLRStaking: Locked staking cannot
claim before the lockup period ends");
    }

    uint256 pendingReward = pendingRewards(orderId);
    require(pendingReward > 0, "BLRStaking: No pending rewards");

    totalRewardEarn[msg.sender] += pendingReward;
    order.claimedReward += pendingReward;

    // Transfer rewards
    BLR.transfer(order.beneficiary, pendingReward);

    emit RewardClaimed(msg.sender, pendingReward);
}
```

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.

EIS - Excessively Integer Size

Criticality	Minor / Informative
Location	staking.sol#L104,112
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.

Since the `lockupDuration` should be less than or equal to 1095 days (94608000 seconds), then it could be stored in a `Math.log2(94608000) = 26.49 -> uint32` variable.

```
uint256 lockupDuration;
```

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.

IRC - Incorrect Reward Calculation

Criticality	Minor / Informative
Location	staking.sol#L278
Status	Unresolved

Description

The `pendingRewards` function applies the multiplier for locked staking orders incorrectly. Instead of increasing the calculated reward amount (reward) by the multiplier, the function reassigns the `reward` variable to the multiplier-adjusted amount. This results in users receiving only the bonus portion as their reward, rather than the full reward amount inclusive of the bonus. Consequently, users will receive fewer rewards than they are entitled to, particularly for locked staking orders.

```
function pendingRewards(uint256 orderId) public view returns (uint256) {
    require(orderId <= latestOrderId, "BLRStaking: INVALID orderId");

    OrderInfo storage orderInfo = orders[orderId];
    if (!orderInfo.claimed) {
        uint256 APY = (orderInfo.amount * orderInfo.returnPer) / PRECISION;

        // Calculate the actual time elapsed since staking started
        uint256 timeElapsed = block.timestamp > orderInfo.endtime
            ? orderInfo.endtime - orderInfo.starttime
            : block.timestamp - orderInfo.starttime;

        // Proportionally calculate rewards based on the time elapsed
        uint256 reward = (APY * timeElapsed) / 365 days;

        // Apply the multiplier for locked staking
        if (orderInfo.locked) {
            reward = (reward * orderInfo.multiplier) / PRECISION;
        }

        uint256 claimAvailable = reward > orderInfo.claimedReward
            ? reward - orderInfo.claimedReward
            : 0;

        return claimAvailable;
    } else {
        return 0;
    }
}
```

Recommendation

To mitigate this issue, it is recommended to correct the logic in the `pendingRewards` function to ensure the multiplier is applied to the full reward amount. This will ensure that the multiplier increases the total reward amount for locked staking orders while maintaining proper deduction for already claimed rewards. This will provide users with the correct reward amounts based on the lockup duration and multiplier.

MTSC - Misleading Total Stakers Count

Criticality	Minor / Informative
Location	staking.sol#L198
Status	Unresolved

Description

The `totalStakers` variable is incremented when a user stakes in a pool they have not previously interacted with. However, the condition only checks if the user has staked in the specific `lockupDuration` pool (`hasStaked[msg.sender][_lockupDuration]`) and does not verify if the user has staked in other pools. This behavior can result in an inflated or misleading totalStakers count, as users staking in multiple pools will be incorrectly counted as new stakers for each pool.

```
if (!hasStaked[msg.sender][_lockupDuration]) {
    stakersPlan[_lockupDuration] += 1;
    totalStakers += 1;
}
```

Recommendation

The team is advised to modify the logic to ensure that `totalStakers` is incremented only when the user is staking for the first time across all pools. The team could introduce a separate mapping to track whether a user has staked in any pool. This ensures that `totalStakers` accurately reflects the number of unique users participating in staking, regardless of the number of pools they interact with.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	staking.sol#L293
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.

```
started = _start;
```

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.

PTAI - Potential Transfer Amount Inconsistency

Criticality	Minor / Informative
Location	staking.sol#L179
Status	Unresolved

Description

The `transfer()` and `transferFrom()` functions are used to transfer a specified amount of tokens to an address. The fee or tax is an amount that is charged to the sender of an ERC20 token when tokens are transferred to another address. According to the specification, the transferred amount could potentially be less than the expected amount. This may produce inconsistency between the expected and the actual behavior.

The following example depicts the diversion between the expected and actual amount.

Tax	Amount	Expected	Actual
No Tax	100	100	100
10% Tax	100	100	90

```
require(BLR.transferFrom(msg.sender, address(this), _amount), "BLRStaking: BLR transferFrom failed");
```

Recommendation

The team is advised to take into consideration the actual amount that has been transferred instead of the expected.

It is important to note that an ERC20 transfer tax is not a standard feature of the ERC20 specification, and it is not universally implemented by all ERC20 contracts. Therefore, the contract could produce the actual amount by calculating the difference between the transfer call.

```
 Actual Transferred Amount = Balance After Transfer - Balance Before Transfer
```

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	staking.sol#L122,123,124,125,126,132,174,292,306
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 private constant _1Pool = 30 days
uint256 private constant _2Pool = 180 days
uint256 private constant _3Pool = 365 days
uint256 private constant _4Pool = 730 days
uint256 private constant _5Pool = 1095 days
IERC20 public BLR
uint256 _lockupDuration
uint256 _amount
bool _locked
bool _start
address _token
```

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/stable/style-guide.html#naming-conventions.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	staking.sol#L95
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() private view returns (bool) {
        return _status == _ENTERED;
    }
```

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	staking.sol#L266,274,278
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 APY = (orderInfo.amount * orderInfo.returnPer) / PRECISION
uint256 reward = (APY * timeElapsed) / 365 days
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	staking.sol#L3,100
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 0.8.20;
pragma solidity ^0.8.0;
```

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	staking.sol#L100
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	staking.sol#L229,256
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.

```
BLR.transfer(order.beneficiary, pendingReward)
BLR.transfer(msg.sender, totalUnstake)
```

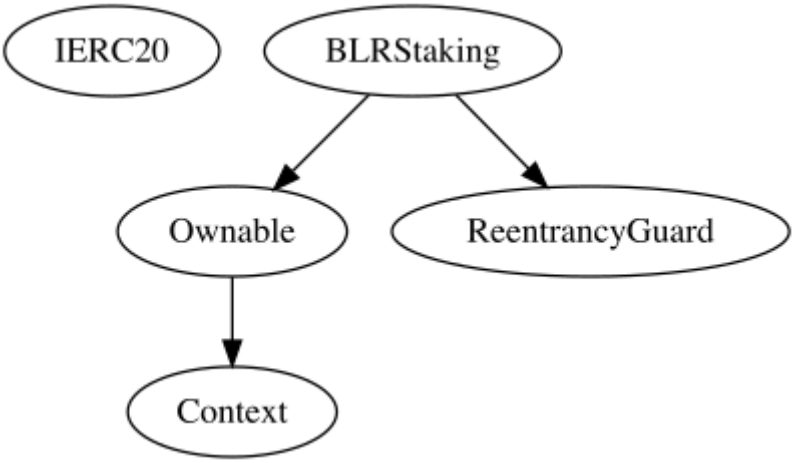
Recommendation

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.

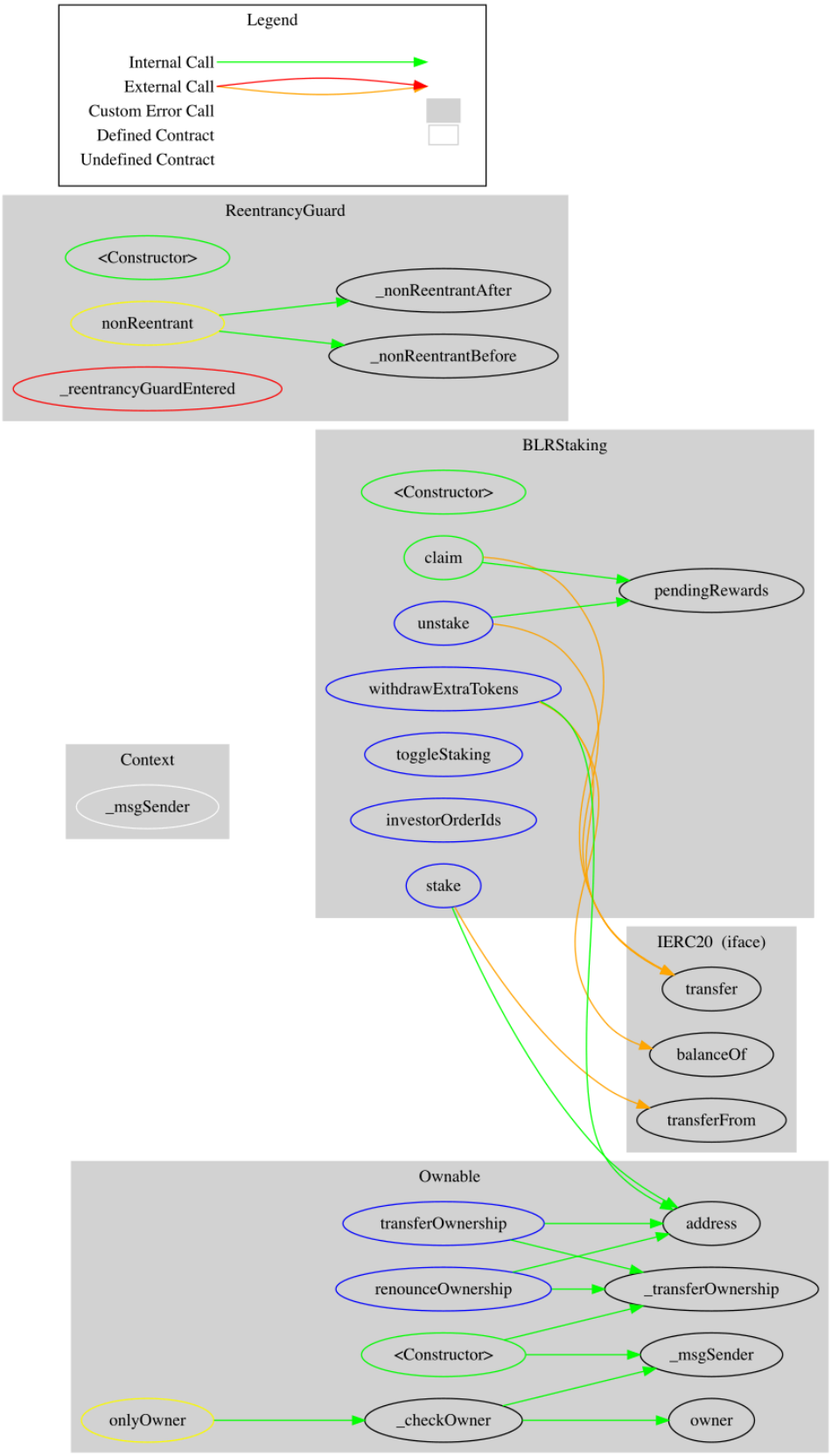
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
BLRStaking	Implementation	Ownable, ReentrancyGuard		
		Public	✓	-
	stake	External	✓	nonReentrant
	claim	Public	✓	nonReentrant
	unstake	External	✓	nonReentrant
	pendingRewards	Public		-
	toggleStaking	External	✓	onlyOwner
	investorOrderIds	External		-
	withdrawExtraTokens	External	✓	onlyOwner

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Custom Error Call
Defined Contract
Undefined Contract
ReentrancyGuard
<Constructor>
nonReentrant
_reentrancyGuardEntered
_nonReentrantAfter
_nonReentrantBefore
BLRStaking
<Constructor>
claim
unstake
withdrawExtraTokens
toggleStaking
investorOrderIds
stake
pendingRewards
Context
_msgSender
IERC20 (iface)
transfer
balanceOf
transferFrom
Ownable
transferOwnership
renounceOwnership
<Constructor>
onlyOwner
_checkOwner
address
_transferOwnership
_msgSender
owner

Summary

BlaroThings contract implements a staking and rewards mechanism. 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

cyberscope.io