Cyberscope

Audit Report

Skyren

December 2024

Network POLYGON

Address 0xBa74014e2A8ab23b14f7D6d067494A0Bf1567bB2

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
●	L02	State Variables could be Declared Constant	Acknowledged
●	L03	Redundant Statements	Acknowledged
●	L04	Conformance to Solidity Naming Conventions	Acknowledged
●	L09	Dead Code Elimination	Acknowledged
●	L11	Unnecessary Boolean equality	Acknowledged
●	L13	Divide before Multiply Operation	Acknowledged
●	L14	Uninitialized Variables in Local Scope	Acknowledged
●	L15	Local Scope Variable Shadowing	Acknowledged
●	L16	Validate Variable Setters	Acknowledged
●	L17	Usage of Solidity Assembly	Acknowledged
●	L20	Succeeded Transfer Check	Acknowledged

Table of Contents

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

Contract Name	TaxToken
Compiler Version	v0.8.17+commit.8df45f5f
Optimization	1 runs
Explorer	https://polygonscan.com/address/0xba74014e2a8ab23b14f7d6d067494a0bf1567bb2
Address	0xba74014e2a8ab23b14f7d6d067494a0bf1567bb2
Network	POLYGON
Symbol	SKYRN
Decimals	18
Total Supply	190,000,000

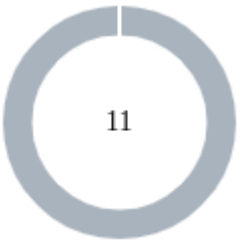
Audit Updates

Initial Audit	02 Dec 2024

Source Files

Filename	SHA256
TaxToken.sol	981442cb631ca707c1b8a1214395b8ccc42efc2065a0d34449c3002fb0fd39ca

Findings Breakdown



Critical	0
Medium	0
Minor / Informative	11

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

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	TaxToken.sol#L4178
Status	Acknowledged

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.

```
uint256 public CONTRACT_VERSION = 1;
```

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.

L03 - Redundant Statements

Criticality	Minor / Informative
Location	TaxToken.sol#L2119
Status	Acknowledged

Description

Redundant statements are statements that are unnecessary or have no effect on the contract's behavior. These can include declarations of variables or functions that are not used, or assignments to variables that are never used.

As a result, it can make the contract's code harder to read and maintain, and can also increase the contract's size and gas consumption, potentially making it more expensive to deploy and execute.

```
contract TaxFacet is Ownable {}
```

Recommendation

To avoid redundant statements, it's important to carefully review the contract's code and remove any statements that are unnecessary or not used. This can help to improve the clarity and efficiency of the contract's code.

By removing unnecessary or redundant statements from the contract's code, the clarity and efficiency of the contract will be improved. Additionally, the size and gas consumption will be reduced.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	TaxToken.sol#L331,430,444,449,456,473,495,518,529,561,631,640,649,658,667,676,685,886,891,897,902,927,933,939,951,955,979,985,991,1003,1007,1014,1241,1363,1721,1772,1777,1819,1827,1832,1847,1851,1866,1881,1888,1916,1944,1951,1956,1961,1966,2136,2244,2473,2479,2485,2490,2515,2536,2540,2548,2559,3535,3536,3537,3549,3558,3569,3575,3579,3586,3597,3606,3620,3627,3634,3649,3659,3669,3679,3755,3771,3819,3835,3847,3851,3855,3859,3863,3867,3960,3965,3998,4178,4181,4182,4243,4244,4261,4312,4339,4381,4386,4428,4455,4494,4499
Status	Acknowledged

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 _newThreshold
address _facet
bytes4 _functionSelector
bytes4 _interfaceId
FacetCut[] memory _diamondCut
bytes4[] memory _functionSelectors
address _facetAddress
bytes4 _selector
string memory _errorMessage
address _contract
address _newSettingsAddress
address _newLosslessAddress
address _newTaxAddress
address _newConstructorAddress

...
```

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	TaxToken.sol#L2981,3002,3127,3139
Status	Acknowledged

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 _mint(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: mint to the zero
address");

        _beforeTokenTransfer(address(0), account, amount);

        _totalSupply += amount;
        _balances[account] += amount;
        emit Transfer(address(0), account, amount);
    }
...
```

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.

L11 - Unnecessary Boolean equality

Criticality	Minor / Informative
Location	TaxToken.sol#L1806,3762
Status	Acknowledged

Description

Boolean equality is unnecessary when comparing two boolean values. This is because a boolean value is either true or false, and there is no need to compare two values that are already known to be either true or false.

it's important to be aware of the types of variables and expressions that are being used in the contract's code, as this can affect the contract's behavior and performance. The comparison to boolean constants is redundant. Boolean constants can be used directly and do not need to be compared to true or false.

```
if(s.taxSettings.canPause == false) {
            return false;
}
```

Recommendation

Using the boolean value itself is clearer and more concise, and it is generally considered good practice to avoid unnecessary boolean equalities in Solidity code.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	TaxToken.sol#L2048,2053,2068,2072,2073,2074,2075,2076,2077,2085
Status	Acknowledged

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.

```
assembly {
        denominator := div(denominator, twos)
    }

    // Divide [prod1 prod0] by the factors of two
    assembly {
        prod0 := div(prod0, twos)
    }
    // Shift in bits from prod1 into prod0. For this we need
    // to flip `twos` such that it is 2**256 / twos.
    // If twos is zero, then it becomes one
    assembly {
        twos := add(div(sub(0, twos), twos), 1)
    }
    prod0 |= prod1 * twos;
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	TaxToken.sol#L1340,1584,1783,2180,3560
Status	Acknowledged

Description

Using an uninitialized local variable can lead to unpredictable behavior and potentially cause errors in the contract. It's important to always initialize local variables with appropriate values before using them.

```
uint256 totalTaxes;
```

Recommendation

By initializing local variables before using them, the contract ensures that the functions behave as expected and avoid potential issues.

L15 - Local Scope Variable Shadowing

Criticality	Minor / Informative
Location	TaxToken.sol#L3998,4048
Status	Acknowledged

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.

```
uint256 currentAllowance = s._allowances[sender][_msgSender()];
```

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	TaxToken.sol#L3730,4362,4478
Status	Acknowledged

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.

```
(bool success, bytes memory result) =
constructorFacetAddress.delegatecall(abi.encodeWithSignature("constructorHandl
er((string,string,uint8,address,uint256,uint256,(bool,bool,bool,bool,bool,bool
,bool,bool),(bool,bool,bool,bool,bool,bool,bool,bool),((uint256,uint256),uint2
56,uint256,uint256),address,(string,(uint256,uint256),address,bool)[],uint256,
uint256,uint256,address,address,bool,(uint256,uint256,uint256,uint256,bool),ui
nt256,(uint256,bool)),address)", params, _factory));
```

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.

L17 - Usage of Solidity Assembly

Criticality	Minor / Informative
Location	TaxToken.sol#L563,2006,3425,3498,4135
Status	Acknowledged

Description

Using assembly can be useful for optimizing code, but it can also be error-prone. It's important to carefully test and debug assembly code to ensure that it is correct and does not contain any errors.

Some common types of errors that can occur when using assembly in Solidity include Syntax, Type, Out-of-bounds, Stack, and Revert.

```
function enforceHasContractCode(address _contract, string memory
_errorMessage) internal view {
        uint256 contractSize;
        assembly {
            contractSize := extcodesize(_contract)
        }
        require(contractSize > 0, _errorMessage);
}
...
```

Recommendation

It is recommended to use assembly sparingly and only when necessary, as it can be difficult to read and understand compared to Solidity code.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	TaxToken.sol#L1717,4347,4367,4463,4483
Status	Acknowledged

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.

```
function transferBalanceToTaxHelper() external {
        uint index = token.taxHelperIndex();
        require(msg.sender == factory.getTaxHelperAddress(index));
        uint256 tokenBalance = token.balanceOf(address(this));
        token.transfer(msg.sender, tokenBalance);
        emit TransferBalancetoTaxHelper(tokenBalance);
    }
```

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**
TaxToken	Implementation	Ownable		
		Public	✓	-
	transferOutBlacklistedFunds	External	✓	-
	isBlacklisted	Public		-
	paused	Public		-
	buyBackBurn	External	✓	-
	handleTaxes	Internal	✓	
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	CONTRACT_VERSION	Public		-
	taxSettings	Public		-
	isLocked	Public		-
	fees	Public		-
	customTaxes	Public		-
	transactionTaxWallet	Public		-
	customTaxLength	Public		-
	MaxTax	Public		-

	MaxCustom	Public		-
	_allowances	Public		-
	_isExcluded	Public		-
	_tFeeTotal	Public		-
	lpTokens	Public		-
	factory	Public		-
	buyBackWallet	Public		-
	lpWallet	Public		-
	pairAddress	Public		-
	taxHelperIndex	Public		-
	marketInit	Public		-
	marketInitBlockTime	Public		-
	antiBotSettings	Public		-
	maxBalanceAfterBuy	Public		-
	swapWhitelistingSettings	Public		-
	recoveryAdmin	Public		-
	admin	Public		-
	timelockPeriod	Public		-
	losslessTurnOffTimestamp	Public		-
	isLosslessTurnOffProposed	Public		-
	isLosslessOn	Public		-
	lossless	Public		-
	balanceOf	Public		-

	tokenFromReflection	Public		-
	_getRate	Public		-
	_getCurrentSupply	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_approve	Private	✓	
	_transfer	Private	✓	
	_beforeTokenTransfer	Internal	✓	
	mint	Public	✓	onlyOwner
	_mint	Internal	✓	
	burn	Public	✓	-
		External	✓	-

Inheritance Graph

For the detailed inheritance graph please refer to the following link.

Flow Graph

For the detailed flow graph please refer to the following link.

Summary

Skyren contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. Skyren is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. The contract Owner can access some admin functions that can not be used in a malicious way to disturb the users' transactions. The team has acknowledged the findings.

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