Cyberscope

A TAC Security Company

Audit Report

Irosh

May 2025

Network BSC

Address 0xcF228a6C603E96821027A663bd4D74f9b645EBB5

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
Medium	UDF	Unused Donation Funds	Unresolved
Minor	CR	Code Repetition	Unresolved
Minor	CCS	Commented Code Segments	Unresolved
Minor	DDP	Decimal Division Precision	Unresolved
Minor	IEA	Incorrect ETH Allocation	Unresolved
Minor	MSC	Misleading Supply Comments	Unresolved
Minor	MEE	Missing Events Emission	Unresolved
Minor	RAC	Redundant Address Check	Unresolved
Minor	RED	Redundant Event Declaration	Unresolved
Minor	RRA	Redundant Repeated Approvals	Unresolved
Minor	RSML	Redundant SafeMath Library	Unresolved
Minor	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor	L09	Dead Code Elimination	Unresolved
Minor	L13	Divide before Multiply Operation	Unresolved

●	L15	Local Scope Variable Shadowing	Unresolved
●	L16	Validate Variable Setters	Unresolved

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	Irosh
Compiler Version	v0.8.26+commit.8a97fa7a
Optimization	200 runs
Explorer	https://bscscan.com/address/0xcf228a6c603e96821027a663bd4d74f9b645ebb5
Address	0xcf228a6c603e96821027a663bd4d74f9b645ebb5
Network	BSC
Symbol	IRO
Decimals	18
Total Supply	1,000,000,000
Badge Eligibility	Yes

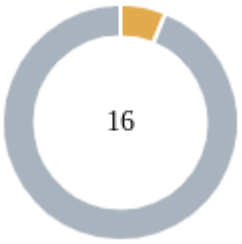
Audit Updates

Initial Audit	02 Jun 2025

Source Files

Filename	SHA256
Irosh.sol	e69adfeef3a2c8992f9e899b3f1cb32f8bc80e439424be24fc46f8bac17e8ed6

Findings Breakdown



Critical	0
Medium	1
Minor / Informative	15

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	0	0	0	0
Medium	1	0	0	0
Minor / Informative	15	0	0	0

UDF - Unused Donation Funds

Criticality	Medium
Location	Irosh.sol#L1185,1224
Status	Unresolved

Description

The contract is designed to allocate portions of transaction fees into three categories: liquidity, marketing, and donation. During the `swapBack` operation, the tokens assigned for liquidity are properly used to add liquidity, and the marketing tokens are swapped for ETH and sent to the designated marketing wallet. However, the portion allocated to `tokensForDonation` is also swapped for ETH but remains unutilized within the contract. This occurs because the code responsible for transferring the resulting ETH to the donation wallet is commented out. As a result, ETH accumulates in the contract balance without a defined purpose or automatic mechanism for utilization. The only remaining pathway to use these accumulated ETH funds is through manual execution of the `buyBackTokens` function by the contract owner. This behaviour introduces inefficiency in the contract's fee allocation logic and may create misleading expectations regarding the donation functionality.

```
 function swapBack() private {
        uint256 contractBalance = balanceOf(address(this));
        uint256 totalTokensToSwap = tokensForLiquidity +
tokensForMarketing + tokensForDonation;

        if(contractBalance == 0 || totalTokensToSwap == 0) {return;}

        // Halve the amount of liquidity tokens
        uint256 liquidityTokens = contractBalance * tokensForLiquidity /
totalTokensToSwap / 2;
        ...

        // if(address(this).balance > 0) {
        //     (success,) = address(donationWallet).call{value:
address(this).balance}("");
        // }
    }

    function buyBackTokens(uint256 bnbAmountInWei) external onlyOwner {
        // generate the uniswap pair path of weth -> eth
        address[] memory path = new address[](2);
        path[0] = uniswapV2Router.WETH();
        path[1] = address(this);

        // make the swap

uniswapV2Router.swapExactETHForTokensSupportingFeeOnTransferTokens{value:
bnbAmountInWei}(
            0, // accept any amount of Ethereum
            path,
            address(0xdead),
            block.timestamp
        );
        emit BuyBackTriggered(bnbAmountInWei);
    }
```

Recommendation

It is recommended to reconsider the implementation and intended use of `tokensForDonation`. One option is to reinstate and properly configure the code that transfers the ETH to the designated donation wallet, ensuring that funds are used as expected. Alternatively, if donations are no longer a supported feature, the related logic and token allocations should be removed to prevent misleading behaviour. If accumulation of ETH is intended solely for buybacks, it may be clearer to directly allocate those tokens to a buyback mechanism rather than labeling them as donations.

CR - Code Repetition

Criticality	Minor / Informative
Location	Irosh.sol#L1074
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.

```
if (automatedMarketMakerPairs[from] &&
!_isExcludedMaxTransactionAmount[to]) {
    require(amount <= maxTransactionAmount, "Buy transfer amount exceeds
the maxTransactionAmount.");
    }

//when sell
else if (automatedMarketMakerPairs[to] &&
!_isExcludedMaxTransactionAmount[from]) {
    require(amount <= maxTransactionAmount, "Sell transfer amount exceeds
the maxTransactionAmount.");
        }
```

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.

CCS - Commented Code Segments

Criticality	Minor / Informative
Location	Irosh.sol#L1041,1218
Status	Unresolved

Description

The contract contains several code segments that are commented out. Blocks of code, including important operations and validation checks, are present but commented out. Commented code can be a source of confusion, as it's unclear whether these segments are meant for future use, are remnants of previous iterations, or are temporarily disabled for testing purposes. Moreover, commented out code can clutter the contract, making it more challenging to read and understand the actual functioning code.

```
// function updatedonationWallet(address newWallet) external onlyOwner {
//     emit donationWalletUpdated(newWallet, donationWallet);
//     donationWallet = newWallet;
// }
...
// if(address(this).balance > 0) {
//     (success,) = address(donationWallet).call{value:
address(this).balance}("");
// }
```

Recommendation

It is recommended to either remove the parts of the code that are not intended to be used or to declare and code the appropriate segments properly if they are meant for future implementation. If the intention is to preserve these segments for historical or reference purposes, it would be beneficial to move them to documentation outside of the active codebase. This approach helps maintain the clarity and cleanliness of the contract's code, ensuring that it accurately reflects its current functionality and intended use.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	Irosh.sol#L1117
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.

```
if (automatedMarketMakerPairs[to] && sellTotalFees > 0){
    fees = amount.mul(sellTotalFees).div(100);
    tokensForLiquidity += fees * sellLiquidityFee / sellTotalFees;
    tokensForDonation += fees * sellDonationFee / sellTotalFees;
    tokensForMarketing += fees * sellMarketingFee / sellTotalFees;
    tokensForBurn += fees * sellBurnFee / sellTotalFees;
}
// on buy
else if(automatedMarketMakerPairs[from] && buyTotalFees > 0) {
    fees = amount.mul(buyTotalFees).div(100);
    tokensForLiquidity += fees * buyLiquidityFee / buyTotalFees;
    tokensForDonation += fees * buyDonationFee / buyTotalFees;
    tokensForMarketing += fees * buyMarketingFee / buyTotalFees;
    tokensForBurn += fees * buyBurnFee / buyTotalFees;
}
```

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.

IEA - Incorrect ETH Allocation

Criticality	Minor / Informative
Location	Irosh.sol#L1185
Status	Unresolved

Description

The contract is performing calculations to determine the ETH allocation for marketing and donations after a token swap. However, it incorrectly uses `totalTokensToSwap` as the divisor when distributing the resulting ETH balance. This leads to an inaccurate allocation of ETH—specifically, more ETH is assigned to liquidity than intended. Since liquidity tokens are halved prior to the swap, the correct divisor should account for only the tokens that were actually swapped for ETH, namely: half of the tokens reserved for liquidity plus the full amounts of tokens reserved for marketing and donations. Using the full `totalTokensToSwap` skews the distribution and may result in ETH being left unutilized in the contract if the ratios are not accurate.

```
    function swapBack() private {
        uint256 contractBalance = balanceOf(address(this));
        uint256 totalTokensToSwap = tokensForLiquidity +
tokensForMarketing + tokensForDonation;

        if(contractBalance == 0 || totalTokensToSwap == 0) {return;}

        // Halve the amount of liquidity tokens
        uint256 liquidityTokens = contractBalance * tokensForLiquidity /
totalTokensToSwap / 2;
        uint256 amountToSwapForETH =
contractBalance.sub(liquidityTokens);

        uint256 initialETHBalance = address(this).balance;

        swapTokensForEth(amountToSwapForETH);

        uint256 ethBalance =
address(this).balance.sub(initialETHBalance);

        uint256 ethForMarketing =
ethBalance.mul(tokensForMarketing).div(totalTokensToSwap);
```

```
        uint256 ethForDonation =
ethBalance.mul(tokensForDonation).div(totalTokensToSwap);
        ...
        }
```

Recommendation

It is recommended to revise the ETH allocation logic to use the sum of the actual swapped tokens: `(tokensForLiquidity / 2) + tokensForMarketing + tokensForDonation` instead of `totalTokensToSwap`. This will ensure that the ETH balance is accurately proportioned according to the real token-to-ETH conversion input, preventing excess ETH from remaining locked in the contract.

MSC - Misleading Supply Comments

Criticality	Minor / Informative
Location	Irosh.sol#L940
Status	Unresolved

Description

The contract is using inline comment messages to indicate specific portions of the total token supply when assigning values to transaction-related variables. However, these comments inaccurately describe the proportions being set. For instance, a variable initialized with 0.1% of the total supply is labeled as representing 1%, and vice versa. Such inconsistencies can mislead reviewers and integrators, potentially resulting in misunderstandings about the contract's intended behavior or incorrect assumptions during further development and audits. This may also cause confusion in parameter tuning or in the assessment of the contract's anti-whale and liquidity mechanisms.

```
maxTransactionAmount = totalSupply * 1 / 1000; // 1% of total supply
swapTokensAtAmount = totalSupply * 1 / 10000; // 0.1% of total supply
```

Recommendation

It is recommended to update the comments to accurately reflect the actual mathematical logic used in the variable assignments. Ensuring consistency between code logic and documentation improves clarity, reduces the chance of misinterpretation, and supports better maintainability and auditing of the smart contract.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	Irosh.sol#L980
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.

```
   function updateSwapTokensAtAmount(uint256 newAmount) external
onlyOwner returns (bool){
           ...
      }

    function updateMaxAmount(uint256 newNum) external onlyOwner {
        ...
    }

    function excludeFromMaxTransaction(address updAds, bool isEx) public
onlyOwner {
        _isExcludedMaxTransactionAmount[updAds] = isEx;
    }

    function updateSwapEnabled(bool enabled) external onlyOwner(){
        swapEnabled = enabled;
    }

    function updateBuyFees(uint256 _marketingFee, uint256 _liquidityFee,
uint256 _donationFee, uint256 _burnFee) external onlyOwner {
        ...
    }

    function updateSellFees(uint256 _marketingFee, uint256 _liquidityFee,
uint256 _donationFee, uint256 _burnFee) external onlyOwner {
    ...
    }
```

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.

RAC - Redundant Address Check

Criticality	Minor / Informative
Location	Irosh.sol#L1053
Status	Unresolved

Description

The contract is performing an unnecessary check within the `_transfer` function by verifying that the `to` address is not the zero address (`to != address(0)`) inside a conditional block. However, this check is redundant, as the function already includes a `require(to != address(0))` statement earlier in its execution. Since the `require` will revert the transaction if `to` is the zero address, the subsequent conditional `if (to != address(0))` will always evaluate to true for any code that follows it. This redundancy does not impact functionality but introduces unnecessary code complexity, which may affect readability and maintainability.

```
function _transfer(
        address from,
        address to,
        uint256 amount
    ) internal override {
        require(from != address(0), "ERC20: transfer from the zero
address");
        require(to != address(0), "ERC20: transfer to the zero
address");

         if(amount == 0) {
            super._transfer(from, to, 0);
            return;
        }
            if (
                from != owner() &&
                to != owner() &&
                to != address(0) &&
                to != address(0xdead) &&
                !swapping
            ){
            ...
```

Recommendation

It is recommended to remove the redundant `to != address(0)` check within the conditional logic following the initial `require` statements. Streamlining the conditional expressions improves code clarity, reduces gas usage marginally, and enhances auditability by eliminating misleading or unnecessary logic.

RED - Redundant Event Declaration

Criticality	Minor / Informative
Location	Irosh.sol#L889
Status	Unresolved

Description

The contract uses events that are not emitted within the contract's functions. As a result, these declared events are redundant and serve no purpose within the contract's current implementation.

```
event UpdateUniswapV2Router(address indexed newAddress, address indexed oldAddress);
```

Recommendation

To optimize contract performance and efficiency, it is advisable to regularly review and refactor the codebase, removing the unused event declarations. This proactive approach not only streamlines the contract, reducing deployment and execution costs but also enhances readability and maintainability.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	Irosh.sol#L1155,1172
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.

```
_approve(address(this), address(uniswapV2Router), tokenAmount);
    // make the swap

uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
        tokenAmount,
        0,
        path,
        address(this),
        block.timestamp
    );
...

_approve(address(this), address(uniswapV2Router), tokenAmount)
// add the liquidity
uniswapV2Router.addLiquidityETH{value: ethAmount}(
    address(this),
    tokenAmount,
    0, // slippage is unavoidable
    0, // slippage is unavoidable
    deadAddress,
    block.timestamp
);
```

Recommendation

Since the approved address is a trusted third-party source, it is recommended to optimize the contract by approving the maximum amount of tokens once in the initial set of the variable, rather than before each operation. This change will reduce the overall gas consumption and improve the efficiency of the contract.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	Irosh.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	Irosh.sol#L31,32,49,722,893,905,907,1001,1010
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 DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
function WETH() external pure returns (address);
mapping (address => bool) public _isExcludedMaxTransactionAmount
event marketingWalletUpdated(address indexed newWallet, address indexed
oldWallet);
event donationWalletUpdated(address indexed newWallet, address indexed
oldWallet);
uint256 _donationFee
uint256 _liquidityFee
uint256 _marketingFee
uint256 _burnFee
```

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	Irosh.sol#L398
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 _burn(address account, uint256 amount) internal virtual {
        require(account != address(0), "ERC20: burn from the zero
address");

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

        _balances[account] = _balances[account].sub(amount, "ERC20: burn
amount exceeds balance");
        _totalSupply = _totalSupply.sub(amount);
        emit Transfer(account, address(0), 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.

L13 - Divide before Multiply Operation

Criticality	Minor / Informative
Location	Irosh.sol#L1118,1119,1120,1121,1122,1126,1127,1128,1129,1130
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.

```
fees = amount.mul(buyTotalFees).div(100)
tokensForDonation += fees * buyDonationFee / buyTotalFees
```

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	Irosh.sol#L937
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.

```
uint256 totalSupply = 1_000_000_000 * 1e18
```

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	Irosh.sol#L1038
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.

```
marketingWallet = newMarketingWallet
```

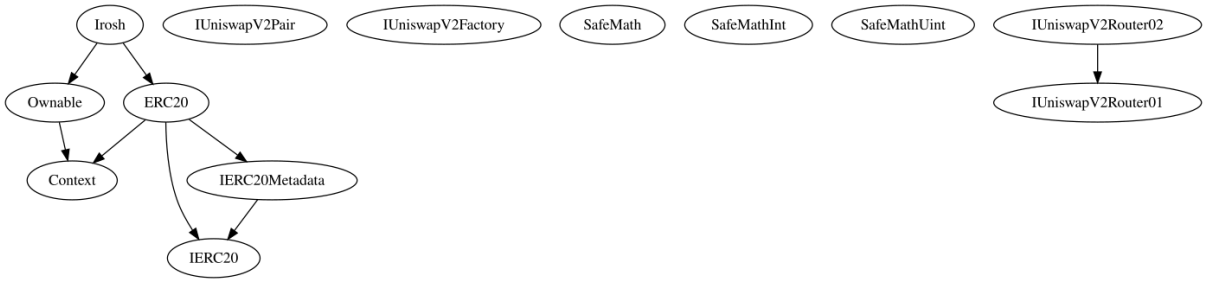
Recommendation

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.

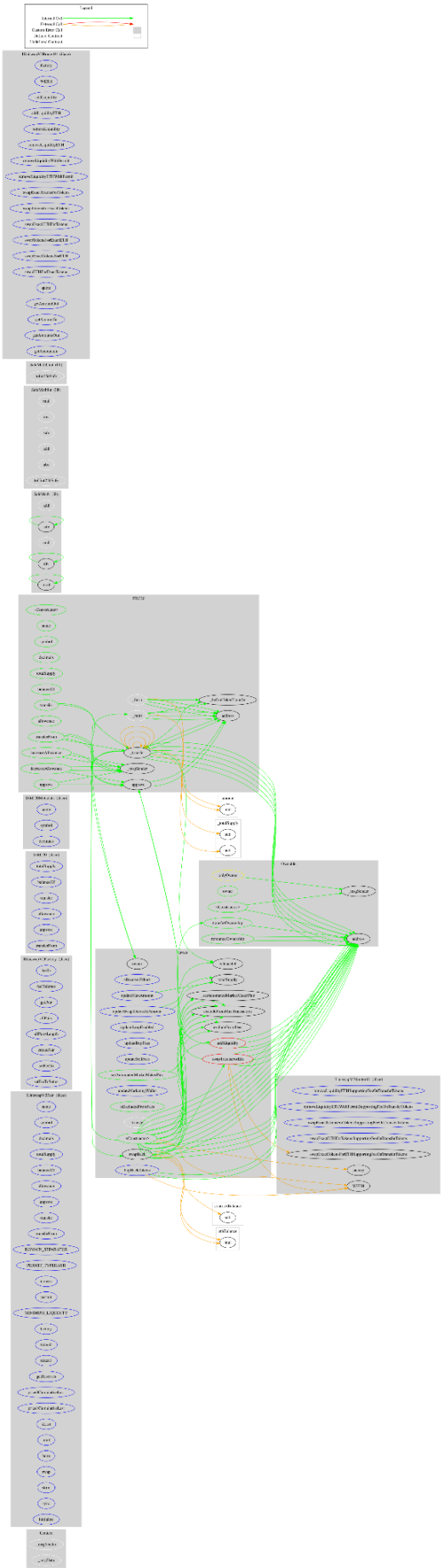
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Irosh	Implementation	ERC20, Ownable		
		Public	✓	ERC20
		External	Payable	-
	updateSwapTokensAtAmount	External	✓	onlyOwner
	updateMaxAmount	External	✓	onlyOwner
	excludeFromMaxTransaction	Public	✓	onlyOwner
	updateSwapEnabled	External	✓	onlyOwner
	updateBuyFees	External	✓	onlyOwner
	updateSellFees	External	✓	onlyOwner
	excludeFromFees	Public	✓	onlyOwner
	setAutomatedMarketMakerPair	Public	✓	onlyOwner
	_setAutomatedMarketMakerPair	Private	✓	
	updateMarketingWallet	External	✓	onlyOwner
	isExcludedFromFees	Public		-
	_transfer	Internal	✓	
	swapTokensForEth	Private	✓	
	addLiquidity	Private	✓	
	swapBack	Private	✓	

Inheritance Graph



Flow Graph



Summary

Irosh contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. There are some functions that can be abused by the owner, like stopping transactions. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. There is also a limit of a max 15% fee.

The contract's ownership has been renounced. The information regarding the transaction can be accessed through the following link:

https://bscscan.com/tx/0x636ec83cd4bc7949b2e73eee93894fa205fdd343a8fffda6db48a53c337732d2

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.

A **TAC Security** Company

The Cyberscope team

cyberscope.io