Cyberscope

Audit Report

Fuku-Kun

August 2024

Network	ETH
Address	0x1001271083C249bd771E1bb76C22D935809a61Ee
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	Unresolved
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Passed

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	AOI	Arithmetic Operations Inconsistency	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MEM	Missing Error Messages	Unresolved
●	MU	Modifiers Usage	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	RRA	Redundant Repeated Approvals	Unresolved
●	RSML	Redundant SafeMath Library	Unresolved
●	RSRS	Redundant SafeMath Require Statement	Unresolved
●	UAR	Unexcluded Address Restrictions	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	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	FUKU
Compiler Version	v0.8.23+commit.f704f362
Optimization	200 runs
Explorer	https://etherscan.io/address/0x1001271083c249bd771e1bb76c22d935809a61ee
Address	0x1001271083c249bd771e1bb76c22d935809a61ee
Network	ETH
Symbol	FUKU
Decimals	9
Total Supply	420,690,000,000

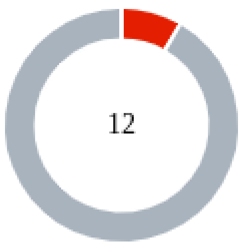
Audit Updates

Initial Audit	20 Aug 2024

Source Files

Filename	SHA256
FUKU.sol	0bf7eeee62a67a49f2f3fc9b0caec336b287ef6e31c9f530e808877950640dda

Findings Breakdown



🟥	Critical	1
🟧	Medium	0
●	Minor / Informative	11

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

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	FUKU.sol#L319
Status	Unresolved

Description

The contract deployer has the authority to increase fees over the allowed limit of 25%. The deployer may take advantage of it by calling the `reduceFee` function with a high percentage value.

```
uint256 private _finalBuyTax=0;
uint256 private _finalSellTax=0;
```

```
function _transfer(address from, address to, uint256 amount)
private {
        ...
        taxAmount =
amount.mul((_buyCount>_reduceBuyTaxAt)?_finalBuyTax:_initialBuy
Tax).div(100);
        ...
        taxAmount =
amount.mul((_buyCount>_reduceSellTaxAt)?_finalSellTax:_initialS
ellTax).div(100);
        ...
}
```

```
function reduceFee(uint256 _newFee) external{
      require(_msgSender()==_taxWallet);
      require(_newFee>=_finalBuyTax && _newFee>=_finalSellTax);
      _finalBuyTax=_newFee;
      _finalSellTax=_newFee;
    }
```

The name of the `reduceFee` function is misleading as it permits the deployer to increase the buy and sale fees more than the current `_finalBuyTax` and `_finalSellTax` fees. Both variables are however initialized with zero values. The deployer can thus set the tax to any arbitrary number, i.e. 100, turning the token in a honeypot.

Recommendation

The contract should employ a check for the maximum acceptable fee percentage. The team should therefore revise the implementation of the `reduceFee` function to reflect its use. In the current contract, the deployer has the unilateral ability to increase fees beyond the accepted limits.

AOI - Arithmetic Operations Inconsistency

Criticality	Minor / Informative
Location	FUKU.sol#L228,257
Status	Unresolved

Description

The contract uses both the SafeMath library and native arithmetic operations. The SafeMath library is commonly used to mitigate vulnerabilities related to integer overflow and underflow issues. However, it was observed that the contract also employs native arithmetic operators (such as +, -, *, /) in certain sections of the code.

The combination of SafeMath library and native arithmetic operations can introduce inconsistencies and undermine the intended safety measures. This discrepancy creates an inconsistency in the contract's arithmetic operations, increasing the risk of unintended consequences such as inconsistency in error handling, or unexpected behavior.

```
require(amount <= _maxTxAmount, "Exceeds the _maxTxAmount.");
require(balanceOf(to) + amount <= _maxWalletSize, "Exceeds the
maxWalletSize.");
```

```
_balances[from]=_balances[from].sub(amount);
_balances[to]=_balances[to].add(amount.sub(taxAmount));
```

Recommendation

To address this finding and ensure consistency in arithmetic operations, it is recommended to standardize the usage of arithmetic operations throughout the contract. The contract should be modified to either exclusively use SafeMath library functions or entirely rely on native arithmetic operations, depending on the specific requirements and design considerations. This consistency will help maintain the contract's integrity and mitigate potential vulnerabilities arising from inconsistent arithmetic operations.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	FUKU.sol#L158
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.

```
_taxWallet
```

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	FUKU.sol#L220,316,317,325,337
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(!bots[from] && !bots[to])
require(_msgSender()==_taxWallet)
require(_newFee>=_finalBuyTax && _newFee>=_finalSellTax)
```

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.

MU - Modifiers Usage

Criticality	Minor / Informative
Location	FUKU.sol#L316,325,337
Status	Unresolved

Description

The contract is using repetitive statements on some methods to validate some preconditions. In Solidity, the form of preconditions is usually represented by the modifiers. Modifiers allow you to define a piece of code that can be reused across multiple functions within a contract. This can be particularly useful when you have several functions that require the same checks to be performed before executing the logic within the function.

```
require(_msgSender()==_taxWallet);
```

Recommendation

The team is advised to use modifiers since it is a useful tool for reducing code duplication and improving the readability of smart contracts. By using modifiers to perform these checks, it reduces the amount of code that is needed to write, which can make the smart contract more efficient and easier to maintain.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	FUKU.sol#L267
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.

```
function swapTokensForEth(uint256 tokenAmount) private lockTheSwap {
    address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = uniswapV2Router.WETH();
    _approve(address(this), address(uniswapV2Router), tokenAmount);
    uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
        tokenAmount,
        0,
        path,
        address(this),
        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.

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	FUKU.sol#L271
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.

```
function swapTokensForEth(uint256 tokenAmount) private lockTheSwap {
        ...
        _approve(address(this), address(uniswapV2Router), tokenAmount);
        ...
    }
```

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	FUKU.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.

RSRS - Redundant SafeMath Require Statement

Criticality	Minor / Informative
Location	FUKU.sol#L34
Status	Unresolved

Description

The contract utilizes a `require` statement within the `add` function aiming to prevent overflow errors. This function is designed based on the SafeMath library's principles. In Solidity version 0.8.0 and later, arithmetic operations revert on overflow and underflow, making the overflow check within the function redundant. This redundancy could lead to extra gas costs and increased complexity without providing additional security.

```
function add(uint256 a, uint256 b) internal pure returns
(uint256) {
        uint256 c = a + b;
        require(c >= a, "SafeMath: addition overflow");
        return c;
}
```

Recommendation

It is recommended to remove the `require` statement from the add function since the contract is using a Solidity pragma version equal to or greater than 0.8.0. By doing so, the contract will leverage the built-in overflow and underflow checks provided by the Solidity language itself, simplifying the code and reducing gas consumption. This change will uphold the contract's integrity in handling arithmetic operations while optimizing for efficiency and cost-effectiveness.

UAR - Unexcluded Address Restrictions

Criticality	Minor / Informative
Location	FUKU.sol#L223
Status	Unresolved

Description

The contract incorporates operational restrictions on transactions, which can hinder seamless interaction with decentralized applications (dApps) such as launchpads, presales, lockers, or staking platforms. In scenarios where an external contract, such as a launchpad factory, needs to integrate with the contract, it should be exempt from the limitations to ensure uninterrupted service and functionality. Failure to provide such exemptions can block the successful process and operation of services reliant on this contract. In this case, it is not possible to exclude certain decentralized applications from the implementation of fees.

```
if (from == uniswapV2Pair && to != address(uniswapV2Router) && !
_isExcludedFromFee[to] ) { ... }
```

Recommendation

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

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	FUKU.sol#L125,126,129,130,131,140,141
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.

```
uint256 private _initialBuyTax=20
uint256 private _initialSellTax=20
uint256 private _reduceBuyTaxAt=20
uint256 private _reduceSellTaxAt=20
uint256 private _preventSwapBefore=20
uint256 public _taxSwapThreshold= 4200000000 * 10**_decimals
uint256 public _maxTaxSwap= 4200000000 * 10**_decimals
```

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	FUKU.sol#L106,134,135,136,137,138,139,140,141,315
Status	Unresolved

Description

The Solidity style guide is a set of guidelines for writing clean and consistent Solidity code. Adhering to a style guide can help improve the readability and maintainability of the Solidity code, making it easier for others to understand and work with.

The followings are a few key points from the Solidity style guide:

1. Use camelCase for function and variable names, with the first letter in lowercase (e.g., myVariable, updateCounter).
2. Use PascalCase for contract, struct, and enum names, with the first letter in uppercase (e.g., MyContract, UserStruct, ErrorEnum).
3. Use uppercase for constant variables and enums (e.g., MAX_VALUE, ERROR_CODE).
4. Use indentation to improve readability and structure.
5. Use spaces between operators and after commas.
6. Use comments to explain the purpose and behavior of the code.
7. Keep lines short (around 120 characters) to improve readability.

```
function WETH() external pure returns (address);
uint8 private constant _decimals = 9
uint256 private constant _tTotal = 420690000000 * 10**_decimals
string private constant _name = unicode"Fuku-Kun"
string private constant _symbol = unicode"FUKU"
uint256 public _maxTxAmount = 8400000000 * 10**_decimals
uint256 public _maxWalletSize = 8400000000 * 10**_decimals
uint256 public _taxSwapThreshold= 4200000000 * 10**_decimals
uint256 public _maxTaxSwap= 4200000000 * 10**_decimals
uint256 _newFee
```

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.

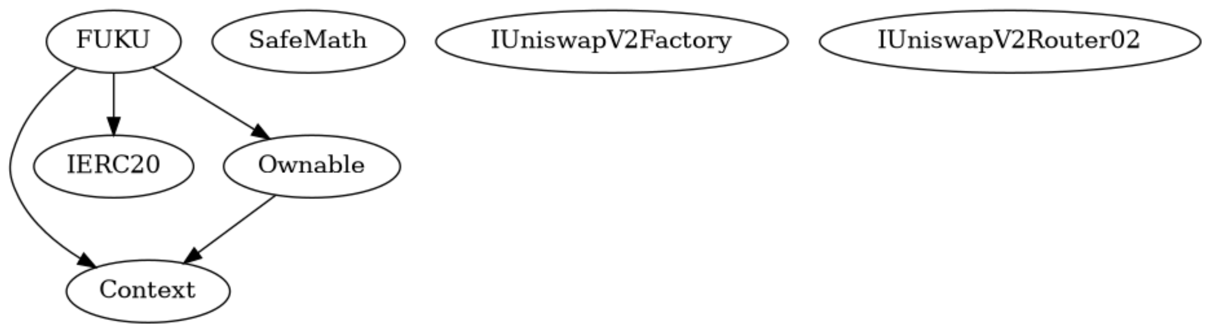
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
Context	Implementation			
	_msgSender	Internal		
IERC20	Interface			
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
SafeMath	Library			
	add	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	div	Internal		
Ownable	Implementation	Context		

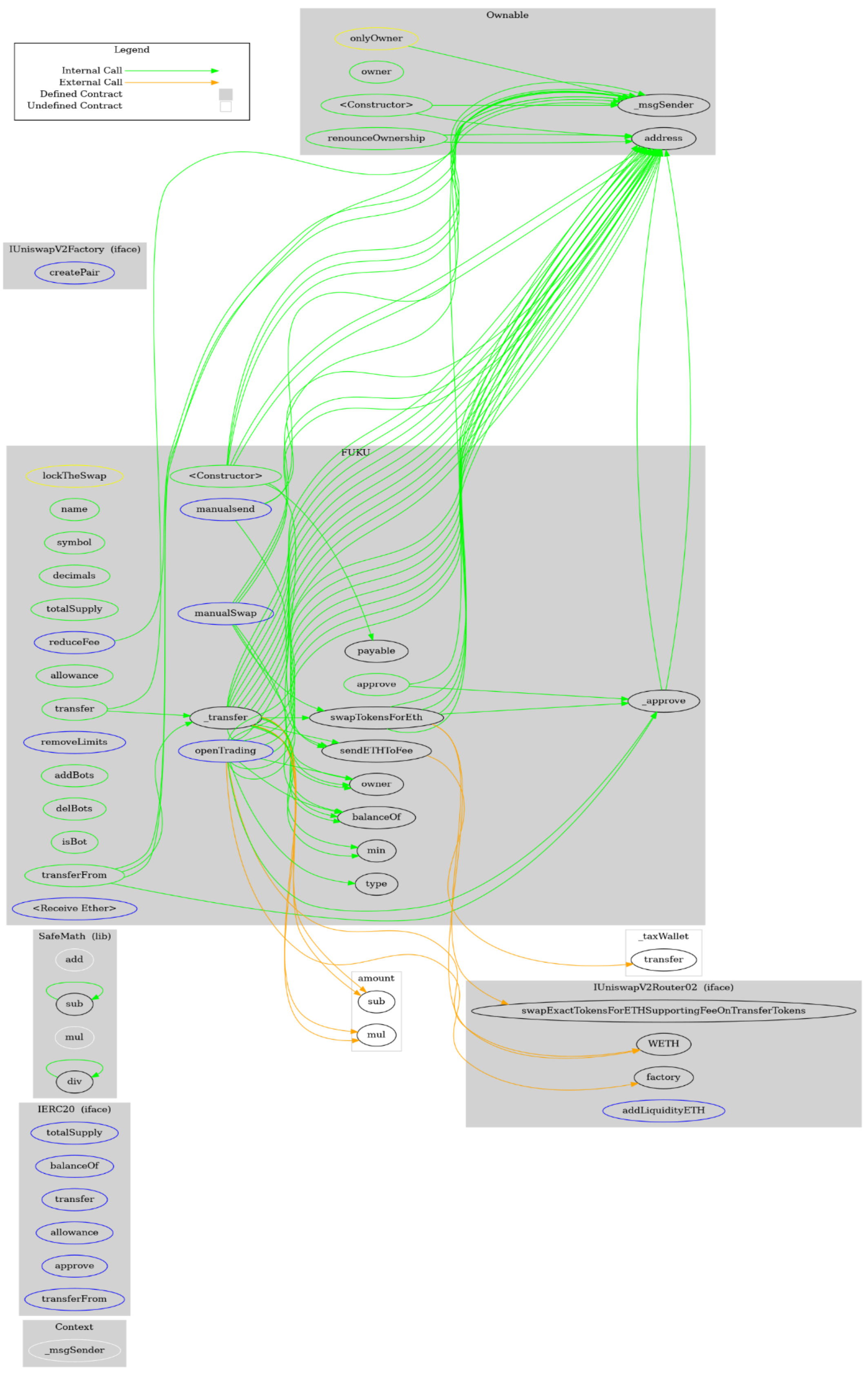
		Public	✓	-
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
IUniswapV2Factory	Interface			
	createPair	External	✓	-
IUniswapV2Router02	Interface			
	swapExactTokensForETHSupportingFeeOnTransferTokens	External	✓	-
	factory	External		-
	WETH	External		-
	addLiquidityETH	External	Payable	-
FUKU	Implementation	Context, IERC20, Ownable		
		Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-

	transferFrom	Public	✓	-
	_approve	Private	✓	
	_transfer	Private	✓	
	min	Private		
	swapTokensForEth	Private	✓	lockTheSwap
	removeLimits	External	✓	onlyOwner
	sendETHToFee	Private	✓	
	addBots	Public	✓	onlyOwner
	delBots	Public	✓	onlyOwner
	isBot	Public		-
	openTrading	External	✓	onlyOwner
	reduceFee	External	✓	-
		External	Payable	-
	manualSwap	External	✓	-
	manualsend	External	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
onlyOwner
owner
<Constructor>
_msgSender
renounceOwnership
address
IUniswapV2Factory (iface)
createPair
FUKU
lockTheSwap
<Constructor>
name
manualsend
symbol
decimals
totalSupply
manualSwap
reduceFee
payable
allowance
approve
transfer
_transfer
swapTokensForEth
_approve
removeLimits
openTrading
sendETHToFee
addBots
owner
delBots
balanceOf
isBot
min
transferFrom
type
<Receive Ether>
SafeMath (lib)
add
sub
mul
div
_taxWallet
transfer
amount
sub
mul
IUniswapV2Router02 (iface)
swapExactTokensForETHSupportingFeeOnTransferTokens
WETH
factory
addLiquidityETH
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom
Context
_msgSender

Summary

The Smart Contract analysis reported no compiler error and two critical issues. The contract deployer can access some admin functions that can be used in a malicious way to disturb the users' transactions.

There are some functions that can be abused by the deployer, like manipulating fees. The maximum fee percentage can be set arbitrarily high to prevent users from selling their tokens.

This audit investigates security issues, business logic concerns and potential improvements.

Disclaimer

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