Cyberscope

Audit Report

Marvin Token

June 2024

SHA256	b1d86a05eb4174349e30e18680bfd3542623aeb8559ac4ca8b2dfdbac1691006
Audited by	© cyberscope

Analysis

● Critical ● Medium ● Minor / Informative ● Pass

Severity	Code	Description	Status
●	ST	Stops Transactions	Unresolved
●	OTUT	Transfers User's Tokens	Passed
●	ELFM	Exceeds Fees Limit	Unresolved
●	MT	Mints Tokens	Passed
●	BT	Burns Tokens	Passed
●	BC	Blacklists Addresses	Unresolved

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	CDF	Contract Deployment Failure	Unresolved
●	SRF	Swap Reentrance Failure	Unresolved
●	HNV	Hardcoded Numeric Values	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	MEE	Missing Events Emission	Unresolved
●	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
●	PTRP	Potential Transfer Revert Propagation	Unresolved
●	PVC	Price Volatility Concern	Unresolved
●	RSW	Redundant Storage Writes	Unresolved
●	L02	State Variables could be Declared Constant	Unresolved
●	L04	Conformance to Solidity Naming Conventions	Unresolved
●	L05	Unused State Variable	Unresolved
●	L09	Dead Code Elimination	Unresolved
●	L14	Uninitialized Variables in Local Scope	Unresolved

●	L19	Stable Compiler Version	Unresolved

Table of Contents

Review

Badge Eligibility	Must Fix Criticals

Testing Deploy	https://testnet.bscscan.com/address/0x487f80C0e243e417be66e1B2A68225320Fb35DD8

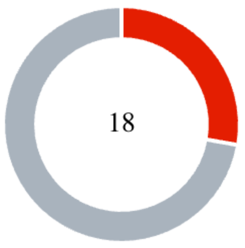
Audit Updates

Initial Audit	04 Jun 2024

Source Files

Filename	SHA256
marvin.sol	b1d86a05eb4174349e30e18680bfd3542623aeb8559ac4ca8b2dfdbac1691006

Findings Breakdown



●	Critical	5
●	Medium	0
●	Minor / Informative	13

	Severity	Unresolved	Acknowledged	Resolved	Other
●	Critical	5	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	13	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	marvin.sol
Status	Unresolved

Description

The contract owner has the authority to stop the sales for all users excluding the owner. The owner may take advantage of it by setting the `MARKETING_WALLET` to contract that does not accept ether.

See PTRP for more details.

Recommendation

The contract should not be able to revert on `_transfer` .

See PTRP for more details.

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	marvin.sol#L1099
Status	Unresolved

Description

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

```
function setSellTaxRate(uint256 _sellTaxRate) external onlyOwner {
    require(_sellTaxRate <= 9000, "Sell tax cannot exceed 90%");
    sellTaxRate = _sellTaxRate;
    emit TaxRateChanged("sell", _sellTaxRate);
}
```

Recommendation

The contract could embody a check for the maximum acceptable value. The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

BC - Blacklists Addresses

Criticality	Critical
Location	marvin.sol#L1047,1129
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `blacklistAddress` function.

```
require(!_isBlacklisted[from], "Blacklisted address");
...
function blacklistAddress(address account, bool value) external
onlyOwner {
    _isBlacklisted[account] = value;
    emit AddressBlacklisted(account, value);
}
```

Recommendation

The team should carefully manage the private keys of the owner's account. We strongly recommend a powerful security mechanism that will prevent a single user from accessing the contract admin functions.

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.
- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

- Renouncing the ownership, which will eliminate the threats but it is non-reversible.

CDF - Contract Deployment Failure

Criticality	Critical
Location	marvin.sol#L1029,1043
Status	Unresolved

Description

The contract is designed to check if trading is open or if the sender is an authorized address before allowing token transfers. This logic, however, leads to a critical issue during the deployment process. Specifically, the variable `tradingOpen` is initialized to `false` , and there is a `require` statement in the `_update` function that ensures transfers are only allowed when trading is open or the sender is an authorized address (owner, marketing wallet, or the contract itself).

In the contract constructor, the `_mint` function is called to mint the initial supply of tokens to the deployer's address (`msg.sender`). However, during this minting process, the `from` address is the zero address (`0x0000000000000000000000000000000000000000`), which is standard for minting operations. This zero address is not included in the list of authorized addresses that can perform transfers when `tradingOpen` is `false` .

As a result, the `require` statement in the `_update` function reverts the transaction during deployment because the zero address is not authorized to transfer tokens while `tradingOpen` is `false` .

```
constructor(address initialOwner) ERC20("Marvin", "MOB")
Ownable(initialOwner) {
    isExcludedFromTax[msg.sender] = true; // Exclude deployer from tax
    isExcludedFromTax[MARKETING_WALLET] = true; // Exclude marketing
wallet from tax
    isExcludedFromTax[address(this)] = true; // Exclude contract address
from tax

    IUniswapV2Router02 _uniswapV2Router =
IUniswapV2Router02(0x8909Dc15e40173Ff4699343b6eB8132c65e18eC6); //
Replace with Base chain router address
    uniswapV2Pair =
IUniswapV2Factory(_uniswapV2Router.factory()).createPair(address(this),
_uniswapV2Router.WETH());
    uniswapV2Router = _uniswapV2Router;

    _mint(msg.sender, INITIAL_SUPPLY);
}
...
function _update(address from, address to, uint256 amount) internal
override {
    require(
        tradingOpen || from == owner() || from == MARKETING_WALLET ||
from == address(this),
        "Trading is not open yet"
    );
}
```

Recommendation

To resolve this issue, modify the `require` statement in the `_update` function to allow the zero address (`0x0000000000000000000000000000000000000000`) to mint tokens during deployment. This can be achieved by including an additional condition to check if the `from` address is the zero address.

SRF - Swap Reentrance Failure

Criticality	Critical
Location	marvin.sol#L1073
Status	Unresolved

Description

The contract is using the `nonReentrant` modifier to prevent the reentrance during the swap and liquify process. In the case of reentrance, the transfer will revert since the `nonReentrant` modifier reverts the execution.

```
    function swapAndSendToMarketing(uint256 contractTokenBalance) private
nonReentrant {
        swapTokensForEth(contractTokenBalance);

        uint256 contractETHBalance = address(this).balance;
        if (contractETHBalance > 0) {
            payable(MARKETING_WALLET).transfer(contractETHBalance);
            emit SwapAndSendToMarketing(contractTokenBalance,
contractETHBalance);
        }
    }
```

Recommendation

The team is advised to prevent the reentrance issue in a more tolerant manner so it will not revert the execution. A recommended way is to not execute the `swapAndSendToMarketing` method during the reentrance phase.

HNV - Hardcoded Numeric Values

Criticality	Minor / Informative
Location	marvin.sol#L1052,1054,1059
Status	Unresolved

Description

The contract incorporates fees on buy, sell or transfer. These fees are calculated using the corresponding tax variables `buyTaxRate` , `sellTaxRate` and `transferTaxRate` divided by a hardcoded value `10000` . Using hardcoded values is a bad practice and may introduce bugs in the future.

```
taxAmount = amount * buyTaxRate / 10000;
...
taxAmount = amount * sellTaxRate / 10000;
...
taxAmount = amount * transferTaxRate / 10000;
```

Recommendation

The team is advised to replace the hardcoded value with a constant in order to have a cleaner and maintainable code base.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	marvin.sol#L1020
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.

```
uniswapV2Pair
```

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.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	marvin.sol#L1121,1125
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 excludeFromTax(address account) external onlyOwner
function includeInTax(address account) 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.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	marvin.sol#L1042,1071,1081
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 _update(address from, address to, uint256 amount) internal
override {
    ...
    uint256 taxAmount;
    if (!isExcludedFromTax[from] && !isExcludedFromTax[to]) {
        if (from == uniswapV2Pair) {
            taxAmount = amount * buyTaxRate / 10000;
            /// @audit-ok hard coded integers division
        } else if (to == uniswapV2Pair) {
            taxAmount = amount * sellTaxRate / 10000;
            if (swapAndLiquifyEnabled && balanceOf(address(this)) >=
swapAndLiquifyThreshold) {
                swapAndSendToMarketing(swapAndLiquifyThreshold);
            }
        } else {
            taxAmount = amount * transferTaxRate / 10000;
        }
    }
    ...
}
...
function swapAndSendToMarketing(uint256 contractTokenBalance) private
nonReentrant {
    swapTokensForEth(contractTokenBalance);

    uint256 contractETHBalance = address(this).balance;
    if (contractETHBalance > 0) {
        payable(MARKETING_WALLET).transfer(contractETHBalance);
        emit SwapAndSendToMarketing(contractTokenBalance,
contractETHBalance);
    }
}
...
function swapTokensForEth(uint256 tokenAmount) private {
    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.

PTRP - Potential Transfer Revert Propagation

Criticality	Minor / Informative
Location	marvin.sol#L1076
Status	Unresolved

Description

The contract sends funds to a `MARKETING_WALLET` as part of the transfer flow. This address can either be a wallet address or a contract. If the address belongs to a contract then it may revert from incoming payment. As a result, the error will propagate to the token's contract and revert the transfer.

```
if (contractETHBalance > 0) {
    payable(MARKETING_WALLET).transfer(contractETHBalance);
    emit SwapAndSendToMarketing(contractTokenBalance,
contractETHBalance);
}
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	marvin.sol#L1081
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapAndLiquifyThreshold` sets a threshold where the contract will trigger the swap functionality. If the variable is set to a big number, then the contract will swap a huge amount of tokens for ETH.

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

```
function swapTokensForEth(uint256 tokenAmount) private {
    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 contract could ensure that it will not sell more than a reasonable amount of tokens in a single transaction. A suggested implementation could check that the maximum amount should be less than a fixed percentage of the exchange reserves. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

RSW - Redundant Storage Writes

Criticality	Minor / Informative
Location	marvin.sol#L1111,1121,1125,1129
Status	Unresolved

Description

The contract modifies the state of the following variables without checking if their current value is the same as the one given as an argument. As a result, the contract performs redundant storage writes, when the provided parameter matches the current state of the variables, leading to unnecessary gas consumption and inefficiencies in contract execution.

```
function setSwapAndLiquifyEnabled(bool _enabled) external onlyOwner
function excludeFromTax(address account) external onlyOwner
function includeInTax(address account) external onlyOwner
function blacklistAddress(address account, bool value) external
onlyOwner
```

Recommendation

The team is advised to implement additional checks within to prevent redundant storage writes when the provided argument matches the current state of the variables. By incorporating statements to compare the new values with the existing values before proceeding with any state modification, the contract can avoid unnecessary storage operations, thereby optimizing gas usage.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	marvin.sol#L999
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.

```
rivate minted = false; // F
```

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	marvin.sol#L28,1088,1094,1100,1106,1111
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);
6 _buyTaxRate) exte
6 _sellTaxRate) exte
6 _transferTaxRate) exte
enabled) exte
6 _threshold) exte
```

Recommendation

By following the Solidity naming convention guidelines, the codebase increased the readability, maintainability, and makes it easier to work with.

Find more information on the Solidity documentation
https://docs.soliditylang.org/en/v0.8.17/style-guide.html#naming-convention.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	marvin.sol#L999
Status	Unresolved

Description

An unused state variable is a state variable that is declared in the contract, but is never used in any of the contract's functions. This can happen if the state variable was originally intended to be used, but was later removed or never used.

Unused state variables can create clutter in the contract and make it more difficult to understand and maintain. They can also increase the size of the contract and the cost of deploying and interacting with it.

```
rivate minted = false; // F
```

Recommendation

To avoid creating unused state variables, it's important to carefully consider the state variables that are needed for the contract's functionality, and to remove any that are no longer needed. This can help improve the clarity and efficiency of the contract.

L09 - Dead Code Elimination

Criticality	Minor / Informative
Location	marvin.sol#L241,436,894
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() internal view returns (bool) {
        return _status == _ENTERED;
    }

on _contextSuffixLength() internal view virtual returns (uint256) {
        return 0;
    }
}

//

...
```

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.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	marvin.sol#L1040
Status	Unresolved

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.

```
6 taxAmount;
```

Recommendation

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

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	marvin.sol#L982
Status	Unresolved

Description

The `^` symbol indicates that any version of Solidity that is compatible with the specified version (i.e., any version that is a higher minor or patch version) can be used to compile the contract. The version lock is a mechanism that allows the author to specify a minimum version of the Solidity compiler that must be used to compile the contract code. This is useful because it ensures that the contract will be compiled using a version of the compiler that is known to be compatible with the code.

```
pragma solidity ^0.8.20;
```

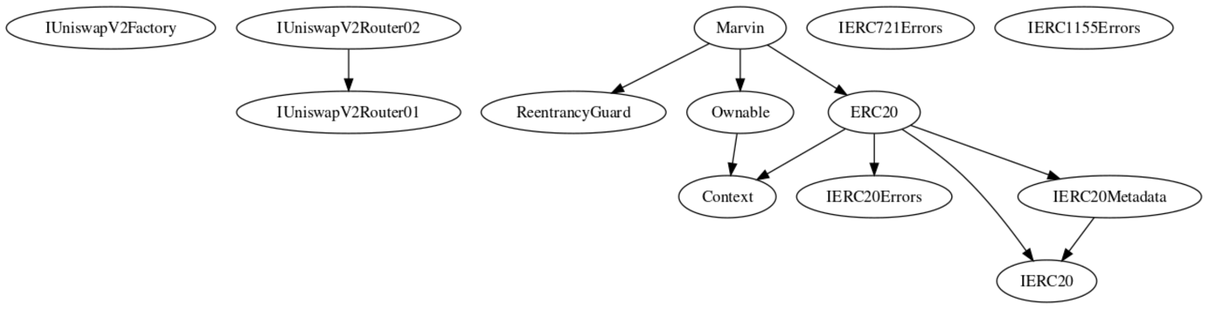
Recommendation

The team is advised to lock the pragma to ensure the stability of the codebase. The locked pragma version ensures that the contract will not be deployed with an unexpected version. An unexpected version may produce vulnerabilities and undiscovered bugs. The compiler should be configured to the lowest version that provides all the required functionality for the codebase. As a result, the project will be compiled in a well-tested LTS (Long Term Support) environment.

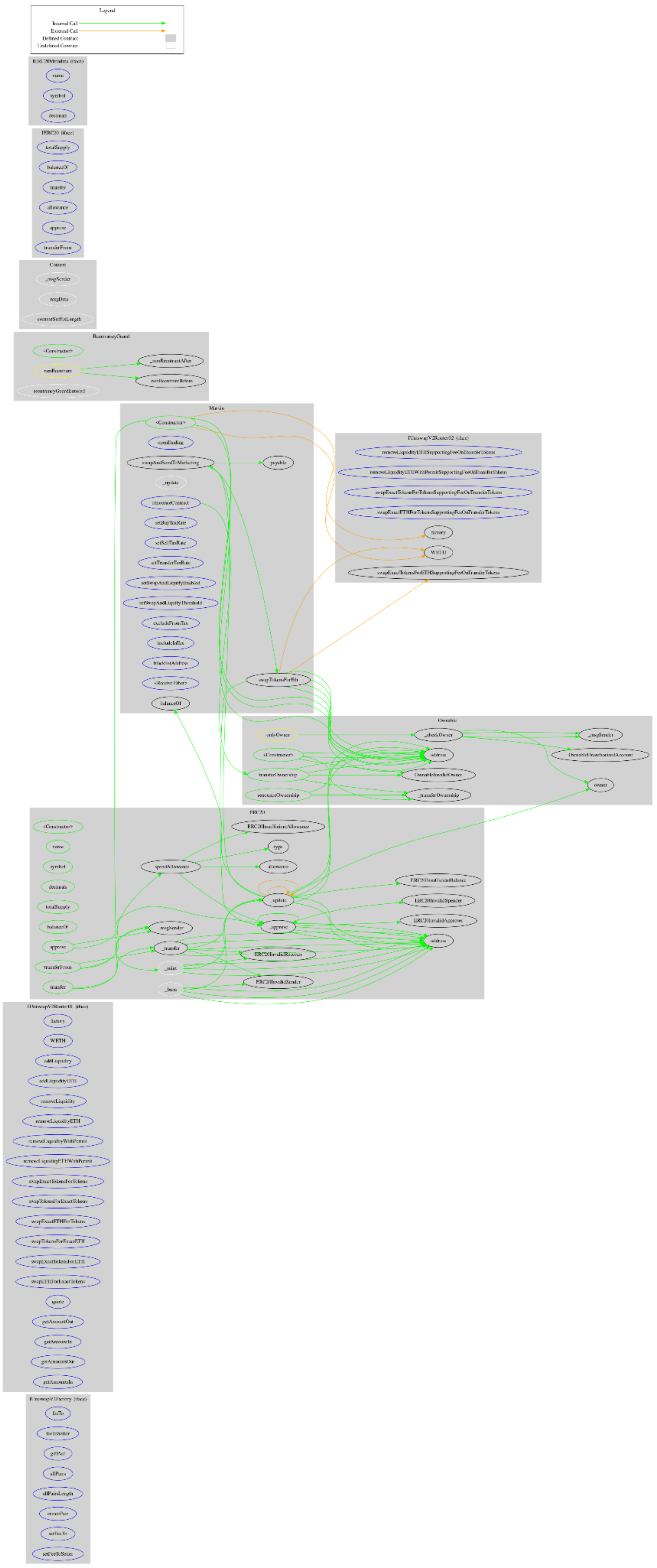
Functions Analysis

Marvin	Implementation	ERC20, Ownable, ReentrancyGuard		
		Public	✓	ERC20 Ownable
	openTrading	External	✓	onlyOwner
	renounceContract	External	✓	onlyOwner
	_update	Internal	✓	
	swapAndSendToMarketing	Private	✓	nonReentrant
	swapTokensForEth	Private	✓	
	setBuyTaxRate	External	✓	onlyOwner
	setSellTaxRate	External	✓	onlyOwner
	setTransferTaxRate	External	✓	onlyOwner
	setSwapAndLiquifyEnabled	External	✓	onlyOwner
	setSwapAndLiquifyThreshold	External	✓	onlyOwner
	excludeFromTax	External	✓	onlyOwner
	includeInTax	External	✓	onlyOwner
	blacklistAddress	External	✓	onlyOwner
		External	Payable	-

Inheritance Graph



Flow Graph



Summary

Marvin Token is an interesting project that has a friendly and growing community. The maximum fee percentage that can be set is 90%. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats. 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

https://www.cyberscope.io