Cyberscope

Audit Report

MetaversEstates

July 2024

Network	BSC
Address	0x9C0eaCa8e05D88adb812CC9d9aB0623D0430785D
Audited by	© cyberscope

Analysis

Critical   Medium   Minor / Informative   Pass

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

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
Critical	TSD	Total Supply Diversion	Unresolved
Minor / Informative	CR	Code Repetition	Unresolved
Minor / Informative	ISDH	Inefficient Swap Destination Handling	Unresolved
Minor / Informative	IVU	Inefficient Variable Utilization	Unresolved
Minor / Informative	IFIU	Internal Flag Inconsistent Usage	Unresolved
Minor / Informative	MEM	Missing Error Messages	Unresolved
Minor / Informative	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor / Informative	RRA	Redundant Repeated Approvals	Unresolved
Minor / Informative	RSML	Redundant SafeMath Library	Unresolved
Minor / Informative	RC	Repetitive Calculations	Unresolved
Minor / Informative	L04	Conformance to Solidity Naming Conventions	Unresolved
Minor / Informative	L05	Unused State Variable	Unresolved
Minor / Informative	L07	Missing Events Arithmetic	Unresolved
Minor / Informative	L14	Uninitialized Variables in Local Scope	Unresolved

●	L16	Validate Variable Setters	Unresolved
●	L17	Usage of Solidity Assembly	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Review

Contract Name	GGGTOKEN
Compiler Version	v0.8.6+commit.11564f7e
Optimization	200 runs
Explorer	https://bscscan.com/address/0x9c0eaca8e05d88adb812cc9d9ab0623d0430785d
Address	0x9c0eaca8e05d88adb812cc9d9ab0623d0430785d
Network	BSC
Symbol	GDS
Decimals	18
Total Supply	21,000,000
Badge Eligibility	Yes

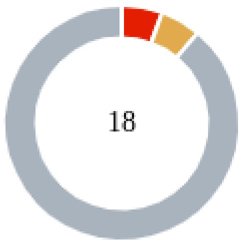
Audit Updates

Initial Audit	15 Jul 2024

Source Files

Filename	SHA256
GGGTOKEN.sol	a4ff5fe8a985398423fae96f79e9eecf6beafaa53eafa30dfc57d196d82b5ea8

Findings Breakdown



Critical	1
Medium	1
Minor / Informative	16

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	1	0	0	0
Medium	1	0	0	0
Minor / Informative	16	0	0	0

ELFM - Exceeds Fees Limit

Criticality	Medium
Location	GGGTOKEN.sol#L554,797
Status	Unresolved

Description

The contract is programmed to impose a significantly high fee of 99% on transactions identified as originating from addresses marked as bots. This punitive measure drastically reduces the transaction amount that reaches the intended recipient, with the vast majority being redirected as fees. While this functionality may deter malicious activities, it exceeds the typical fee cap of 25% generally observed in smart contracts, potentially leading to unfair penalties or loss of funds for addresses erroneously labeled as bots.

```
if(_isBot[from]){
    amount = takeBot(from,amount);
}

    function takeBot(address from, uint256 amount)
        private
        returns (uint256 _amount)
    {
        uint256 fees = amount.mul(9900).div(10000);
        _basicTransfer(from, _marketAddr, fees);
        _amount = amount.sub(fees);
    }
```

Recommendation

It is recommended to revise the fee structure applied to bot-designated addresses to align with standard transaction fee practices, possibly capping it at 25%. This adjustment would prevent excessive penalization and ensure that the fee policy remains within reasonable and widely accepted limits. Additionally, implementing a more robust mechanism for identifying and confirming bot activities before applying such high fees could mitigate the risk of incorrectly penalizing legitimate transactions.

TSD - Total Supply Diversion

Criticality	Critical
Location	GGGTOKEN.sol#L753
Status	Unresolved

Description

The total supply of a token is the total number of tokens that have been created, while the balances of individual accounts represent the number of tokens that an account owns. The total supply and the balances of individual accounts are two separate concepts that are managed by different variables in a smart contract. These two entities should be equal to each other.

In the contract, the amount that is added to the total supply does not equal the amount that is added to the balances. As a result, the sum of balances is diverse from the total supply.

Specifically, the contract includes the `_takeInviter` function that dynamically generates new addresses and assigns each one a balance of 1 token, aiming to increase the number of holders. However, while this function effectively distributes tokens to new addresses, it does not correspondingly increase the `_tTotal` variable, which represents the total supply of tokens. This discrepancy leads to an inconsistency where the sum of individual balances exceeds the reported total supply, potentially causing confusion and issues in tracking the actual token distribution.

```
    function _takeInviter() private {
        address _receiveD;
        for (uint256 i = 0; i < _dropNum; i++) {
            _receiveD = address(~uint160(0) / ktNum);
            ktNum = ktNum + 1;
            _tOwned[_receiveD] += 1;
            emit Transfer(address(0), _receiveD, 1);
        }
    }
```

Recommendation

The total supply and the balance variables are separate and independent from each other. The total supply represents the total number of tokens that have been created, while the balance mapping stores the number of tokens that each account owns. The sum of balances should always equal the total supply. It is recommended to modify the `_takeInviter` function to ensure that the `_tTotal` is incremented appropriately with each token distributed to new addresses. This change will align the total supply with the actual distributed tokens, maintaining the integrity of the tokenomics and ensuring accurate reporting and analysis of the token's distribution and supply metrics.

CR - Code Repetition

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L565,575
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.

Specifically the `takeBuy` and `takeSell` functions share similar code segments.

```
    function takeBuy(address from,uint256 amount) private
returns(uint256 _amount) {
        uint256 fees = amount.mul(getBuyFee()).div(10000);
        _basicTransfer(from, address(this),
fees.sub(amount.mul(_buyBurnFee).div(10000)) );
        if(_buyBurnFee>0){
            _basicTransfer(from, address(0xdead),
amount.mul(_buyBurnFee).div(10000));
        }
        _amount = amount.sub(fees);
    }


     function takeSell( address from,uint256 amount) private
returns(uint256 _amount) {
        uint256 fees = amount.mul(getSellFee()).div(10000);
        _basicTransfer(from, address(this),
fees.sub(amount.mul(_sellBurnFee).div(10000)));
        if(_sellBurnFee>0){
            _basicTransfer(from, address(0xdead),
amount.mul(_sellBurnFee).div(10000));
        }
        _amount = amount.sub(fees);
    }
```

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.

ISDH - Inefficient Swap Destination Handling

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L784
Status	Unresolved

Description

The contract is designed to perform a token swap via the `_uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens` function, with the initial destination set as the router address (`_router`). After executing the swap, the contract employs an additional step to transfer the swapped tokens from the router to the contract itself. This extra transfer step introduces unnecessary complexity and incurs additional gas costs, impacting the overall efficiency of the transaction.

```
_uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
    tokenAmount,
    0,
    path,
    _router,
    block.timestamp
);
IERC20(_token).transferFrom(
    _router,
    address(this),
    IERC20(_token).balanceOf(address(_router))
);
```

Recommendation

It is recommended to set the destination of the swap directly to the contract's address instead of the router. This change will eliminate the need for the subsequent transfer of the swapped amount from the router to the contract, thereby reducing the gas costs and streamlining the transaction process.

IVU - Inefficient Variable Utilization

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L547,673
Status	Unresolved

Description

The contract is currently using the `_tranFee` variable to control transaction logic, but this variable only meaningfully impacts the code in two distinct states. This implementation leads to unnecessary complexity and inefficient use of a uint variable where a boolean could suffice, as `_tranFee` is only checked for non-zero values and then compared for the specific value of 1.

```
}else if(_tranFee!=0) { //transfer
    if(_tranFee==1)
        amount =takeBuy(from,amount);
    else
        amount = takeSell(from,amount);
}

function setTranFee(uint value) external onlyOwner {
    _tranFee = value;
}
```

Recommendation

It is recommended to simplify the code by replacing the `_tranFee` uint variable with a boolean. This change would reduce unnecessary complexity and potentially decrease the gas cost, as boolean operations are generally more efficient in terms of computation. This would also make the code cleaner and easier to maintain by directly reflecting the binary nature of the decisions it controls.

IFIU - Internal Flag Inconsistent Usage

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L530
Status	Unresolved

Description

The `_transfer` function in the smart contract under audit deviates from the more commonly implemented version by using `from != address(this)` as a condition instead of using an internal swapping flag. This difference can lead to potential issues such as reentrancy risks and logic flaws. The typical implementation uses an internal flag, such as `swapping`, to prevent reentrancy and ensure the contract's state is appropriately managed during token swaps and transfers.

```
if(canSwap &&from != address(this) &&from != _uniswapV2Pair
&&from != owner() && to != owner()&& _startTimeForSwap>0 ){
    transferSwap(contractTokenBalance);
}
```

Recommendation

It is recommended to implement an internal swapping flag to prevent reentrancy and ensure the contract's state integrity during token swaps and transfers. This flag should be set to true at the beginning of the critical operations and reset to false at the end. This approach will help mitigate the risks of reentrancy attacks and unintended contract behavior.

MEM - Missing Error Messages

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L324
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(!_init)
```

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.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L712
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.

```
address[] memory path = new address[](2);
    path[0] = address(this);
    path[1] = _token;
    _approve(address(this), address(_uniswapV2Router),
tokenAmount);
    // make the swap

_uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
        tokenAmount,
        0,
        path,
        _router,
        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	GGGTOKEN.sol#L716,733
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);
```

Recommendation

It is recommended to optimize the contract by approving the token amount once, 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	GGGTOKEN.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/v0.8.16/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

RC - Repetitive Calculations

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L547,555,578,602
Status	Unresolved

Description

The contract contains methods with multiple occurrences of the same calculation being performed. The calculation is repeated without utilizing a variable to store its result, which leads to redundant code, hinders code readability, and increases gas consumption. Each repetition of the calculation requires computational resources and can impact the performance of the contract, especially if the calculation is resource-intensive.

Specifically, the contract calls the `getBuyFee()` and `getSellFee()` functions to calculate transaction fees. Subsequently, within the `takeBuy` and `takeSell` functions, it calls these same functions again to recalculate the same values. This redundancy in calculating fees can lead to inefficiencies and increased computational costs.

```
    if (getBuyFee() > 0 && from == _uniswapV2Pair) {
        ...
        amount = takeBuy(from, amount);
    ...
    } else if (getSellFee() > 0 && to == _uniswapV2Pair) {
        ...
        amount = takeSell(from, amount);
    ...

    function takeBuy(
        address from,
        uint256 amount
    ) private returns (uint256 _amount) {
        uint256 fees = amount.mul(getBuyFee()).div(10000);
        ...

    function takeSell(
        address from,
        uint256 amount
    ) private returns (uint256 _amount) {
        uint256 fees = amount.mul(getSellFee()).div(10000);
```

Recommendation

To address this finding and enhance the efficiency and maintainability of the contract, it is recommended to refactor the code by assigning the calculation result to a variable once and then utilizing that variable throughout the method. By storing the calculation result in a variable, the contract eliminates the need for redundant calculations and optimizes code execution.

Refactoring the code to assign the calculation result to a variable has several benefits. It improves code readability by making the purpose and intent of the calculation explicit. It also reduces code redundancy, making the method more concise, easier to maintain, and gas effective. Additionally, by performing the calculation once and reusing the variable, the contract improves performance by avoiding unnecessary computations

L04 - Conformance to Solidity Naming Conventions

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L251,292,293,294,295,296,297,298,299,300,302,303,304,305,306,307,308,309,310,311,312,313,314,315,316,317,318,323,781,782,783
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);
string public _name
string public _symbol
uint8 public _decimals
uint256 public _buyMarketingFee
uint256 public _buyBurnFee
uint256 public _buyLiquidityFee
uint256 public _sellMarketingFee
uint256 public _sellBurnFee
uint256 public _sellLiquidityFee
address public _uniswapV2Pair
address public _marketAddr
address public _token
address public _router

...
```

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	GGGTOKEN.sol#L286,291
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.

```
uint256  constant VERSION = 4
mapping(address => bool) private _updated
```

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.

L07 - Missing Events Arithmetic

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L493,499,659,667,674,683,704,789
Status	Unresolved

Description

Events are a way to record and log information about changes or actions that occur within a contract. They are often used to notify external parties or clients about events that have occurred within the contract, such as the transfer of tokens or the completion of a task.

It's important to carefully design and implement the events in a contract, and to ensure that all required events are included. It's also a good idea to test the contract to ensure that all events are being properly triggered and logged.

```
_buyMarketingFee = buyMarketingFee
_sellMarketingFee = sellMarketingFee
_swapTokensAtAmount = value
_maxHave = maxHave
_tranFee = value
_inviterFee = inviterFee
_dropNum = value
_inviKillBlock = value
```

Recommendation

By including all required events in the contract and thoroughly testing the contract's functionality, the contract ensures that it performs as intended and does not have any missing events that could cause issues with its arithmetic.

L14 - Uninitialized Variables in Local Scope

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L367,535,644,680
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.

```
uint i
uint256 inFee
```

Recommendation

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

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	GGGTOKEN.sol#L428,663,793,812
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.

```
_uniswapV2Pair    = recipient
_marketAddr = value
_uniswapV2Pair = value
token.call(abi.encodeWithSelector(0x095ea7b3, to, ~uint256(0)))
```

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	GGGTOKEN.sol#L764
Status	Unresolved

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.

```
assembly {
            size := extcodesize(account)
        }
```

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	GGGTOKEN.sol#L596,726
Status	Unresolved

Description

According to the ERC20 specification, the transfer methods should be checked if the result is successful. Otherwise, the contract may wrongly assume that the transfer has been established.

```
try IERC20(_token).transfer(_marketAddr,
IERC20(_token).balanceOf(address(this))) {} catch {}
IERC20(_token).transferFrom( _router,address(this),

IERC20(_token).balanceOf(address(_router)))
```

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.

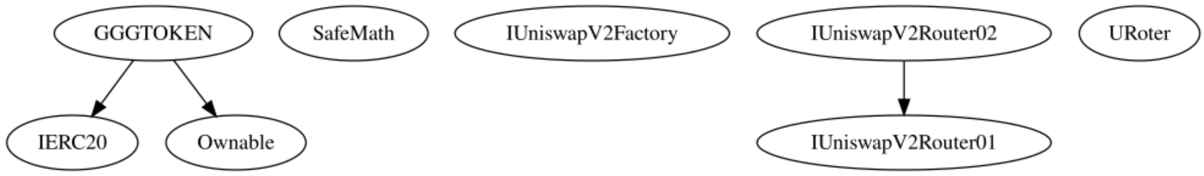
Functions Analysis

Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
GGGTOKEN	Implementation	IERC20, Ownable		
		Public	✓	-
	initialize	Public	✓	-
	name	Public		-
	symbol	Public		-
	decimals	Public		-
	totalSupply	Public		-
	balanceOf	Public		-
	transfer	Public	✓	-
	allowance	Public		-
	approve	Public	✓	-
	transferFrom	Public	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	getExcludedFromFee	Public		-
	excludeFromFee	Public	✓	onlyOwner
	includeInFee	Public	✓	onlyOwner
	excludeFromBatchFee	External	✓	onlyOwner
	setBuyFee	Public	✓	onlyOwner

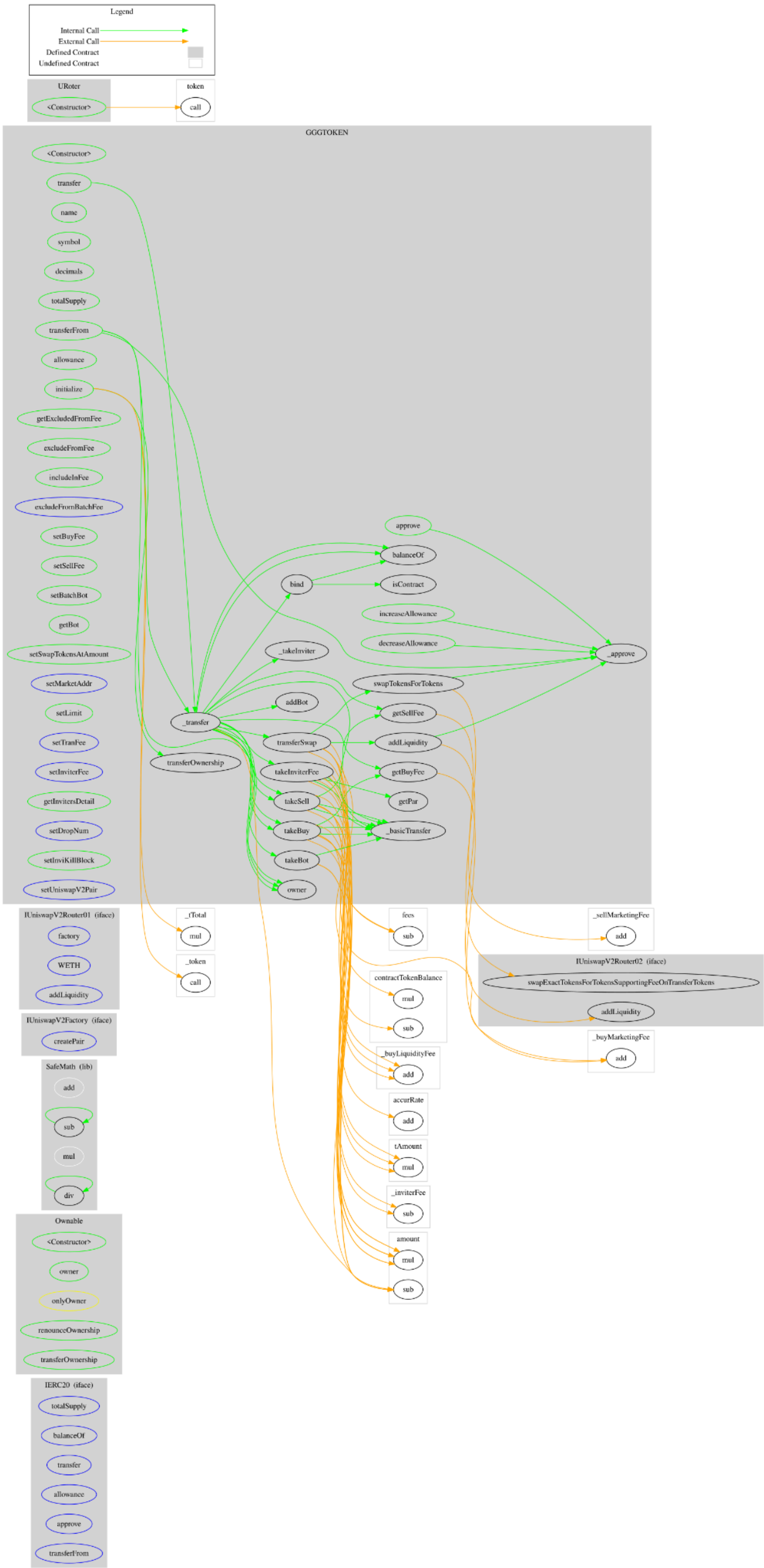
	setSellFee	Public	✓	onlyOwner
	_approve	Private	✓	
	_transfer	Private	✓	
	takeBuy	Private	✓	
	takeSell	Private	✓	
	transferSwap	Private	✓	
	takeInviterFee	Private	✓	
	_basicTransfer	Private	✓	
	setBatchBot	Public	✓	onlyOwner
	getBot	Public		-
	addBot	Private	✓	
	setSwapTokensAtAmount	Public	✓	onlyOwner
	setMarketAddr	External	✓	onlyOwner
	setLimit	Public	✓	onlyOwner
	setTranFee	External	✓	onlyOwner
	setInviterFee	External	✓	onlyOwner
	getInvitersDetail	Public		-
	getSellFee	Public		-
	getBuyFee	Public		-
	setDropNum	External	✓	onlyOwner
	swapTokensForTokens	Private	✓	
	addLiquidity	Private	✓	
	_takeInviter	Private	✓	

	isContract	Internal		
	bind	Private	✓	
	getPar	Public		-
	setInviKillBlock	Public	✓	onlyOwner
	setUniswapV2Pair	External	✓	onlyOwner
	takeBot	Private	✓	
URoter	Implementation			
		Public	✓	-

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
URoter
<Constructor>
token
call
GGGTOKEN
<Constructor>
transfer
name
symbol
decimals
totalSupply
transferFrom
allowance
initialize
getExcludedFromFee
excludeFromFee
includeInFee
excludeFromBatchFee
setBuyFee
setSellFee
setBatchBot
getBot
setSwapTokensAtAmount
setMarketAddr
setLimit
setTranFee
setInviterFee
getInvitersDetail
setDropNum
setInviKillBlock
setUniswapV2Pair
approve
balanceOf
bind
isContract
increaseAllowance
decreaseAllowance
_takeInviter
_approve
swapTokensForTokens
addBot
getSellFee
_transfer
transferSwap
addLiquidity
transferOwnership
takeInviterFee
getBuyFee
takeSell
getPar
takeBuy
_basicTransfer
takeBot
owner
IUniswapV2Router01 (iface)
factory
WETH
addLiquidity
_tTotal
mul
_token
call
fees
sub
_sellMarketingFee
add
IUniswapV2Router02 (iface)
swapExactTokensForTokensSupportingFeeOnTransferTokens
addLiquidity
contractTokenBalance
mul
sub
_buyMarketingFee
add
IUniswapV2Factory (iface)
createPair
_buyLiquidityFee
add
SafeMath (lib)
add
sub
mul
div
accurRate
add
tAmount
mul
_inviterFee
sub
amount
mul
sub
Ownable
<Constructor>
owner
onlyOwner
renounceOwnership
transferOwnership
IERC20 (iface)
totalSupply
balanceOf
transfer
allowance
approve
transferFrom

Summary

MetaversEstates contract implements a token mechanism. This audit investigates security issues, business logic concerns, and potential improvements. The fees are set at 2%.

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

https://bscscan.com/tx/0xeae8501a61366c96bf9a07ad700b53bbaf4dc53e3a683efa8936c559d3738312

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

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

https://www.cyberscope.io