Audit Report

I'm Meta Trader

April 2024

Network	BSC
Address	0xcccac22232db4b252a50c3ff82482909c62ddcce
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	Unresolved

Diagnostics

● Critical ● Medium ● Minor / Informative

Severity	Code	Description	Status
●	IR	Infinite Recursion	Unresolved
●	TSD	Total Supply Diversion	Unresolved
●	CCS	Clean Code Standard	Unresolved
●	CCR	Contract Centralization Risk	Unresolved
●	DOV	Duplicate Ownership Verification	Unresolved
●	IDI	Immutable Declaration Improvement	Unresolved
●	LMI	Locking Mechanism Inconsistency	Unresolved
●	RCS	Redundant Comment Segment	Unresolved
●	RF	Redundant Function	Unresolved
●	TUU	Time Units Usage	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

●	L13	Divide before Multiply Operation	Unresolved
●	L18	Multiple Pragma Directives	Unresolved
●	L20	Succeeded Transfer Check	Unresolved

Table of Contents

Review

Contract Name	BEP20Token
Compiler Version	v0.5.16+commit.9c3226ce
Optimization	200 runs
Explorer	https://bscscan.com/address/0xcccac22232db4b252a50c3ff82482909c62ddcce
Address	0xcccac22232db4b252a50c3ff82482909c62ddcce
Network	BSC
Symbol	IMMT
Decimals	8
Total Supply	1,000,000,000
Badge Eligibility	Must Fix Criticals

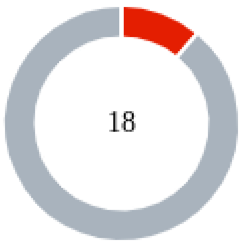
Audit Updates

Initial Audit	17 Apr 2024

Source Files

Filename	SHA256
BEP20Token.sol	a1d7cd880303507cea3364a6ec9875c4fe881a9273c37f4cbccac928d5b9886b

Findings Breakdown



●	Critical	2
●	Medium	0
●	Minor / Informative	16

Severity		Unresolved	Acknowledged	Resolved	Other
●	Critical	2	0	0	0
●	Medium	0	0	0	0
●	Minor / Informative	16	0	0	0

BC - Blacklists Addresses

Criticality	Critical
Location	BEP20Token.sol#L781,800
Status	Unresolved

Description

The contract owner has the authority to stop addresses from transactions. The owner may take advantage of it by calling the `lock` and `lockAfter` functions.

```
function lock(address _holder, uint256 _value, uint256 _releaseTime)
public onlyOwner {
    require(_value > 0, "Invalid lock value");
    require(_releaseTime > block.timestamp, "Token release time must be
after the current time.");
    _releaseLock(_holder);
    require(LockbalanceOf(_holder) >= _value, "Insufficient balance");
    _balances[_holder] = _balances[_holder].sub(_value);
    lockInfo[_holder].push(
        LockInfo(_releaseTime, 1, _value, 0)
    );
    emit Lock(_holder, _value, _releaseTime);
}
...
function lockAfter(address _holder, uint256 _value, uint256 _afterTime)
public onlyOwner {
    lock(_holder, _value, block.timestamp.add(_afterTime));
}
```

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.

IR - Infinite Recursion

Criticality	Critical
Location	BEP20Token.sol#L673
Status	Unresolved

Description

It was observed that the function `LockbalanceOf` exhibits a critical flaw due to infinite recursion. The function is designed to calculate the total locked balance of a given address `_holder`. However, the implementation leads to an infinite loop, resulting in unexpected behavior.

The issue arises from the fact that within the `LockbalanceOf` function, the last statement recursively calls itself without any termination condition.

```
function LockbalanceOf(address _holder) public view returns
(uint256) {
    uint256 lockedBalance = 0;
    uint256 length = lockInfo[_holder].length;
    for (uint256 i = 0; i < length; i++ ) {
        LockInfo memory acc = lockInfo[_holder][i];
        lockedBalance = lockedBalance.add(

acc.unitValue.mul(acc.releaseDays).add(acc.extraValue)
        );
    }
    return LockbalanceOf(_holder).add(lockedBalance);
}
```

Recommendation

To address this critical vulnerability, it is imperative to refactor the `LockbalanceOf` function to eliminate the infinite recursion. This can be achieved by redesigning the function logic to ensure that it terminates properly.

TSD - Total Supply Diversion

Criticality	Critical
Location	BEP20Token.sol#L786,834
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.

```
_balances[_holder] = _balances[_holder].sub(_value);
...
_balances[_holder] = _balances[_holder].sub(_totalValue);
```

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.

CCS - Clean Code Standard

Criticality	Minor / Informative
Location	BEP20Token.sol#L653
Status	Unresolved

Description

It is observed that the struct `LockInfo`, the mapping `lockInfo`, and the associated events are declared within the contract body, interspersed between function definitions. This departure from convention deviates from established clean code standards and can lead to decreased readability, maintainability, and potential confusion for developers.

The struct `LockInfo` and the mapping `lockInfo` are fundamental components of the contract logic, defining the structure for locking mechanisms and storing associated data. Placing these declarations at the top of the contract's code facilitates better code organization and clarity, making it easier for developers to understand the contract's data structure and its intended functionality at a glance.

Similarly, events play a crucial role in smart contract development by providing a transparent and immutable record of contract interactions. By scattering the event declarations throughout the contract body, the code loses cohesion and becomes less intuitive to follow, hindering debugging efforts.

```
uint256 private constant UNIX_DAY_TIME = 86400;

struct LockInfo {
    uint256 releaseStartTime;
    uint256 releaseDays;
    uint256 unitValue;
    uint256 extraValue;
}

mapping(address => LockInfo[]) internal lockInfo;

event Lock(address indexed holder, uint256 value, uint256
releaseTime);
event Lock(
    address indexed holder,
    uint256 totalValue,
    uint256 releaseDays,
    uint256 releaseStartTime
);
event Unlock(address indexed holder, uint256 value);
```

Recommendation

To adhere to clean code standards and improve code readability and maintainability, it is recommended to relocate the struct `LockInfo` , the mapping `lockInfo` , and all associated events to the top of the contract's code, preceding any function declarations. This restructuring aligns with industry best practices and promotes a more organized and comprehensible codebase.

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	BEP20Token.sol#L315,321,327,332,339,345,781,800,815,851,865
Status	Unresolved

Description

The contract's functionality and behavior are heavily dependent on external parameters or configurations. While external configuration can offer flexibility, it also poses several centralization risks that warrant attention. Centralization risks arising from the dependence on external configuration include Single Point of Control, Vulnerability to Attacks, Operational Delays, Trust Dependencies, and Decentralization Erosion.

```
lockToken() external onlyOwner
unlockToken() external onlyOwner
lockTokensUntil(address _holder, uint256 time) external
onlyOwner
unlockTokensUntil(address _holder, uint256 time) external
onlyOwner
lockWallet(address _wallet) external onlyOwner
unlockWallet(address _wallet) external onlyOwner
lock(address _holder, uint256 _value, uint256 _releaseTime)
public onlyOwner
lockAfter(address _holder, uint256 _value, uint256 _afterTime)
public onlyOwner
dailyLock(address _holder, uint256 _totalValue, uint256
_releaseDays, uint256 _releaseStartTime)
    public
    onlyOwner
dailyLockAfter(address _holder, uint256 _totalValue, uint256
_releaseDays, uint256 _afterTime)
    public
    onlyOwner
unlock(address _holder, uint256 i) public onlyOwner
```

Recommendation

To address this finding and mitigate centralization risks, it is recommended to evaluate the feasibility of migrating critical configurations and functionality into the contract's codebase itself. This approach would reduce external dependencies and enhance the contract's self-sufficiency. It is essential to carefully weigh the trade-offs between external configuration flexibility and the risks associated with centralization.

DOV - Duplicate Ownership Verification

Criticality	Minor / Informative
Location	BEP20Token.sol#L327,332
Status	Unresolved

Description

It has been discovered a redundancy in ownership verification within the functions `lockTokensUntil` and `unlockTokensUntil`. Both functions implement ownership verification in two ways: through a modifier named `onlyOwner` and with a `require` statement that checks if `msg.sender` is equal to the `_owner`.

Redundant checks can not only increase gas costs but also introduce unnecessary complexity.

Recommendation

To improve the efficiency and clarity of the code, the team is advised to remove the redundant ownership check within the function body, since the `onlyOwner` modifier already enforces ownership.

By eliminating redundant checks, the code becomes more concise and easier to understand.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	BEP20Token.sol#L416,417,418
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.

```
uint8 private _decimals;
string private _symbol;
string private _name;
```

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.

LMI - Locking Mechanism Inconsistency

Criticality	Minor / Informative
Location	BEP20Token.sol#L781,815
Status	Unresolved

Description

The functions `lock` and `dailyLock` within the smart contract implement the logic to lock tokens for a specified duration. However, there's an inconsistency in the implementation regarding the validation of the locked balance and the subtraction of the locked amount from the available balances.

In both functions, the contract first checks if the locked balance of the `_holder` is more than or equal to the value to be locked. If this condition fails, the contract reverts with an "Insufficient balance" error. However, immediately after this check, the contract subtracts the value to be locked from the `_balances` of the `_holder`, regardless of whether the balance is locked or not.

This implementation presents a discrepancy in logic.

```
function lock(address _holder, uint256 _value, uint256 _releaseTime)
public onlyOwner {
    require(_value > 0, "Invalid lock value");
    require(_releaseTime > block.timestamp, "Token release time must be
after the current time.");
    _releaseLock(_holder);
    require(LockbalanceOf(_holder) >= _value, "Insufficient balance");
    _balances[_holder] = _balances[_holder].sub(_value);
    lockInfo[_holder].push(
        LockInfo(_releaseTime, 1, _value, 0)
    );
    emit Lock(_holder, _value, _releaseTime);
}
...
function dailyLock(
    address _holder,
    uint256 _totalValue,
    uint256 _releaseDays,
    uint256 _releaseStartTime
) public onlyOwner{
    require(_totalValue > 0, "Invalid lock totalValue");
    require(
        _releaseDays > 0 && _releaseDays <= 1000,
        "Invalid releaseDays (0 < releaseDays <= 1000"
    );
    require(_releaseStartTime > block.timestamp, "Token release start
time must be after the current time.");

    _releaseLock(_holder);
    require(_totalValue <= LockbalanceOf(_holder), "Insufficient
balance");

    uint256 unitValue = _totalValue.div(_releaseDays);
    uint256 extraValue = _totalValue.sub(unitValue.mul(_releaseDays));

    _balances[_holder] = _balances[_holder].sub(_totalValue);

    lockInfo[_holder].push(
        LockInfo(_releaseStartTime, _releaseDays, unitValue, extraValue)
    );

    emit Lock(_holder, _totalValue, _releaseDays, _releaseStartTime);
}
```

Recommendation

To address the inconsistency observed in the locking mechanism, the team is advised to review and refine the validation logic within the `lock` and `dailyLock` functions. Specifically, ensure that the deduction of the locked value from the `_balances` is aligned with the intended behavior of the contract. This entails revising the validation conditions to accurately reflect the requirement for sufficient locked balance before deducting the value.

RCS - Redundant Comment Segment

Criticality	Minor / Informative
Location	BEP20Token.sol#L648
Status	Unresolved

Description

The contract contains a redundant comment in the codebase. While comments are crucial for enhancing code readability and facilitating understanding, they should be concise and informative. Redundant comments can clutter the codebase and may lead to confusion or misinformation if they diverge from the actual functionality of the code.

The comment preceding the definition of the `UNIX_DAY_TIME` constant appears to be redundant. It reiterates the purpose of the function to destroy tokens and deduct the amount from the caller's allowance, which is unrelated to the `UNIX_DAY_TIME` constant.

```
/**
   * @dev Destroys `amount` tokens from `account`.`amount` is then
deducted
   * from the caller's allowance.
   *
   * See {_burn} and {_approve}.
   */
uint256 private constant UNIX_DAY_TIME = 86400;
```

Recommendation

Consider reviewing and refining the comments throughout the codebase to ensure they accurately describe the associated functionality without redundancy. Remove the redundant comment preceding the `UNIX_DAY_TIME` constant to maintain code clarity and readability. Additionally, ensure that comments focus on explaining the purpose, behavior, and usage of the corresponding code elements to aid developers in understanding the smart contract's logic.

RF - Redundant Function

Criticality	Minor / Informative
Location	BEP20Token.sol#L355,465
Status	Unresolved

Description

The contract contains a redundant function `getOwner()` that mirrors the functionality of the `owner()` function. Both functions return the same value - the address of the contract owner. While redundancy itself may not necessarily pose a security risk, it can introduce unnecessary complexity.

Redundant code segments like `getOwner()` may confuse developers or users interacting with the contract. Additionally, redundant functions can increase gas costs and bloating of the contract size without providing any tangible benefit.

```
function owner() public view returns (address) {
    return _owner;
}
...
function getOwner() external view returns (address) {
    return owner();
}
```

Recommendation

Consider removing the redundant function `getOwner()` and encouraging users to utilize the existing `owner()` function directly. Simplifying the contract interface by eliminating unnecessary duplication enhances readability and improves overall contract efficiency.

TUU - Time Units Usage

Criticality	Minor / Informative
Location	BEP20Token.sol#L653
Status	Unresolved

Description

The contract is using arbitrary numbers to form time-related values. As a result, it decreases the readability of the codebase and prevents the compiler to optimize the source code.

```
uint256 private constant UNIX_DAY_TIME = 86400;
```

Recommendation

It is a good practice to use the time units reserved keywords like `seconds` , `minutes` , `hours` , `days` and `weeks` to process time-related calculations.

It's important to note that these time units are simply a shorthand notation for representing time in seconds, and do not have any effect on the actual passage of time or the execution of the contract. The time units are simply a convenience for expressing time in a more human-readable form.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	BEP20Token.sol#L425
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.

```
IBEP20 public token
```

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	BEP20Token.sol#L327,332,339,345,447,673,689,725,742,759,781,800,816,817,818,819,852,853,854,855,865
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.

```
address _holder
address _wallet
uint _tokenAmount
address[] calldata _path
uint _expectedMinOut

...
```

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	BEP20Token.sol#L311,312,313,315,321,327,332,339,345
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.

```
bool public tokenLocked;
mapping(address => bool) public walletLocked;
mapping(address => uint256) public lockUntil;

function lockToken() external onlyOwner {
    require(!tokenLocked, "Token is already locked");
    tokenLocked = true;
    emit TokenLock();
}

function unlockToken() external onlyOwner {
    require(tokenLocked, "Token is already unlocked");
    tokenLocked = false;
    emit TokenUnlock();
}

function lockTokensUntil(address _holder, uint256 time) external
onlyOwner {
    require(msg.sender == _owner, "Only the owner can lock tokens");
    lockUntil[_holder] = time;
}

    function unlockTokensUntil(address _holder, uint256 time) external
onlyOwner {
    require(msg.sender == _owner, "Only the owner can unlock tokens");
    require(time < lockUntil[_holder], "Unlock time must be earlier than
current lock time");
    lockUntil[_holder] = time;
}


function lockWallet(address _wallet) external onlyOwner {
    require(!walletLocked[_wallet], "Wallet is already locked");
    walletLocked[_wallet] = true;
    emit WalletLock(_wallet);
}

function unlockWallet(address _wallet) external onlyOwner {
    require(walletLocked[_wallet], "Wallet is already unlocked");
    walletLocked[_wallet] = false;
    emit WalletUnlock(_wallet);
}
```

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	BEP20Token.sol#L618
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 {
    require(account != address(0), "BEP20: burn from the zero
address");

    _balances[account] = _balances[account].sub(amount, "BEP20:
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	BEP20Token.sol#L831,832
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.

```
uint256 unitValue = _totalValue.div(_releaseDays)
uint256 extraValue =
_totalValue.sub(unitValue.mul(_releaseDays))
```

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.

L18 - Multiple Pragma Directives

Criticality	Minor / Informative
Location	BEP20Token.sol#L1,153,246
Status	Unresolved

Description

If the contract includes multiple conflicting pragma directives, it may produce unexpected errors. To avoid this, it's important to include the correct pragma directive at the top of the contract and to ensure that it is the only pragma directive included in the contract.

```
pragma solidity 0.5.16;
```

Recommendation

It is important to include only one pragma directive at the top of the contract and to ensure that it accurately reflects the version of Solidity that the contract is written in.

By including all required compiler options and flags in a single pragma directive, the potential conflicts could be avoided and ensure that the contract can be compiled correctly.

L20 - Succeeded Transfer Check

Criticality	Minor / Informative
Location	BEP20Token.sol#L452
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.

```
token.transferFrom(msg.sender, address(this), _tokenAmount)
```

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

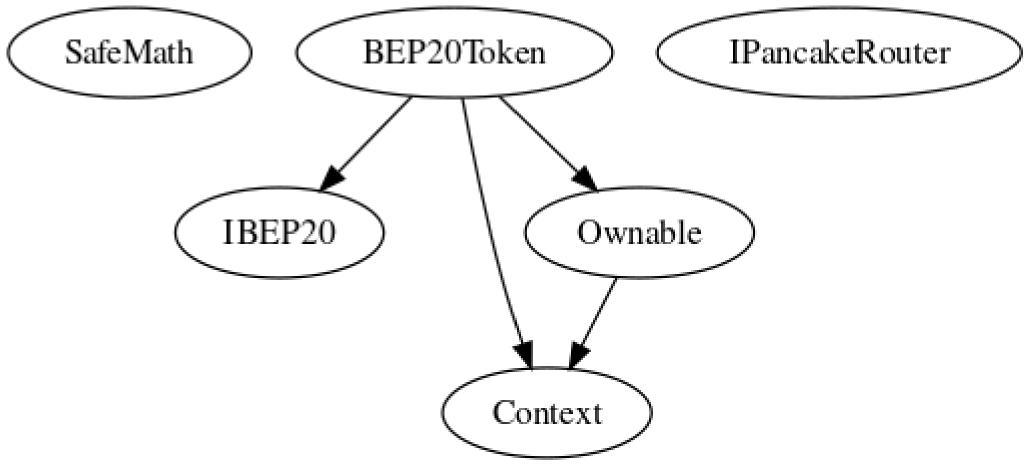
Contract	Type	Bases		
	Function Name	Visibility	Mutability	Modifiers
SafeMath	Library			
	add	Internal		
	sub	Internal		
	sub	Internal		
	mul	Internal		
	div	Internal		
	div	Internal		
	mod	Internal		
	mod	Internal		
IBEP20	Interface			
	totalSupply	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	getOwner	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-

	approve	External	✓	-
	transferFrom	External	✓	-
Context	Implementation			
		Internal	✓	
	_msgSender	Internal		
	_msgData	Internal		
Ownable	Implementation	Context		
		Internal	✓	
	lockToken	External	✓	onlyOwner
	unlockToken	External	✓	onlyOwner
	lockTokensUntil	External	✓	onlyOwner
	unlockTokensUntil	External	✓	onlyOwner
	lockWallet	External	✓	onlyOwner
	unlockWallet	External	✓	onlyOwner
	owner	Public		-
	renounceOwnership	Public	✓	onlyOwner
	transferOwnership	Public	✓	onlyOwner
	_transferOwnership	Internal	✓	
IPancakeRouter	Interface			
	getAmountsOut	External		-

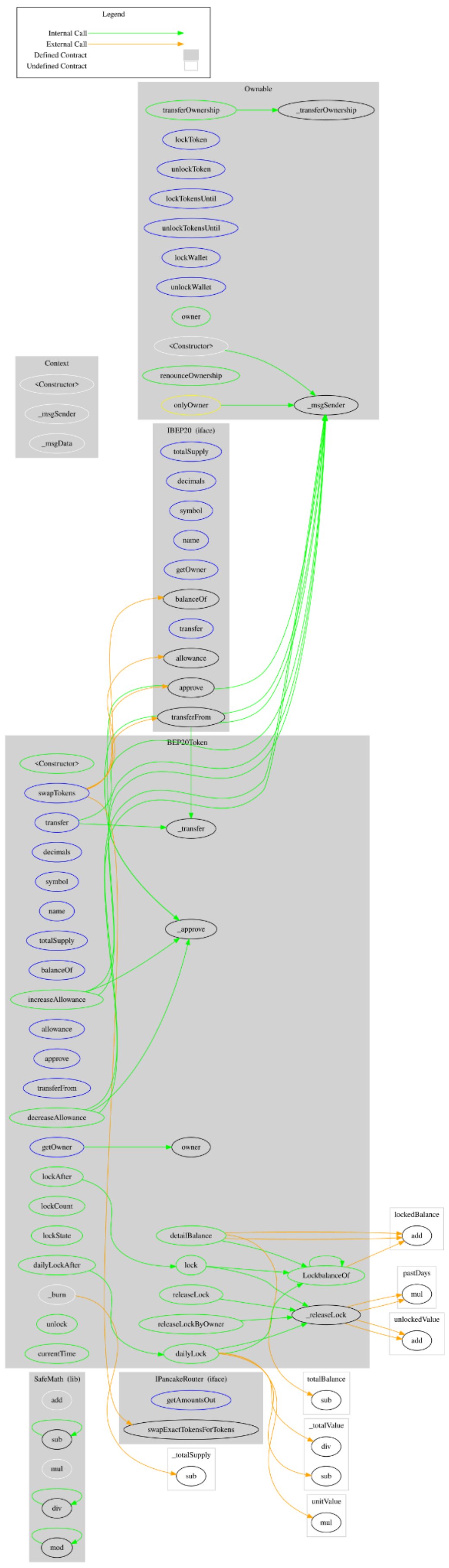
	swapExactTokensForTokens	External	✓	-
BEP20Token	Implementation	Context, IBEP20, Ownable		
		Public	✓	-
	swapTokens	External	✓	-
	getOwner	External		-
	decimals	External		-
	symbol	External		-
	name	External		-
	totalSupply	External		-
	balanceOf	External		-
	transfer	External	✓	-
	allowance	External		-
	approve	External	✓	-
	transferFrom	External	✓	-
	increaseAllowance	Public	✓	-
	decreaseAllowance	Public	✓	-
	_transfer	Internal	✓	
	_burn	Internal	✓	
	_approve	Internal	✓	
	LockbalanceOf	Public		-
	detailBalance	Public		-
	releaseLockByOwner	Public	✓	onlyOwner

	releaseLock	Public	✓	-
	lockCount	Public		-
	lockState	Public		-
	lock	Public	✓	onlyOwner
	lockAfter	Public	✓	onlyOwner
	dailyLock	Public	✓	onlyOwner
	dailyLockAfter	Public	✓	onlyOwner
	unlock	Public	✓	onlyOwner
	currentTime	Public		-
	_releaseLock	Internal	✓	

Inheritance Graph



Flow Graph


Legend
Internal Call
External Call
Defined Contract
Undefined Contract
Ownable
transferOwnership
_transferOwnership
lockToken
unlockToken
lockTokensUntil
unlockTokensUntil
lockWallet
unlockWallet
owner
<Constructor>
renounceOwnership
onlyOwner
_msgSender
Context
<Constructor>
_msgSender
_msgData
IBEP20 (iface)
totalSupply
decimals
symbol
name
getOwner
balanceOf
transfer
allowance
approve
transferFrom
BEP20Token
<Constructor>
swapTokens
transfer
_transfer
decimals
symbol
name
_approve
totalSupply
balanceOf
increaseAllowance
allowance
approve
transferFrom
decreaseAllowance
getOwner
owner
lockAfter
lockCount
lockState
detailBalance
dailyLockAfter
lock
LockbalanceOf
_burn
releaseLock
_releaseLock
unlock
releaseLockByOwner
currentTime
dailyLock
lockedBalance
add
pastDays
mul
unlockedValue
add
SafeMath (lib)
add
sub
mul
div
mod
IPancakeRouter (iface)
getAmountsOut
swapExactTokensForTokens
_totalSupply
sub
totalBalance
sub
_totalValue
div
sub
unitValue
mul

Summary

I'm Meta Trader 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 massively blacklist addresses. A multi-wallet signing pattern will provide security against potential hacks. Temporarily locking the contract or renouncing ownership will eliminate all the contract threats.

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