Cyberscope

# Audit Report
# MEMECOIF

October 2024

| | |
|---|---|
| Network | BSC |
| Address | 0x063d2d1f52f4B8C12475D05c351EFBDc14d78aF0 |
| Audited by | © cyberscope |

# 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** | MemecoifContract |
| **Compiler Version** | v0.8.17+commit.8df45f5f |
| **Optimization** | 200 runs |
| **Explorer** | https://bscscan.com/address/0x063d2d1f52f4b8c12475d05c351efbdc14d78af0 |
| **Address** | 0x063d2d1f52f4b8c12475d05c351efbdc14d78af0 |
| **Network** | BSC |
| **Symbol** | MEMECOIF |
| **Decimals** | 18 |
| **Total Supply** | 100,000,000 |
| **Badge Eligibility** | Yes |

## Audit Updates

| | |
|---|---|
| **Initial Audit** | 14 Oct 2024<br>https://github.com/cyberscope-io/audits/blob/main/memecoif/v1/audit.pdf |
| **Corrected Phase 2** | 17 Oct 2024 |

## Source Files

| **Filename** | SHA256 |
|---|---|
| **MemecoifContract.sol** | 8fa247a184435043c211d1b091e28afaea4166dd968210b28f1c58b643f25c16 |

## Analysis

● Critical ● Medium ● Minor / Informative ● Pass

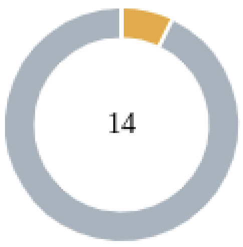
| Severity | Code | Description | Status |
|---|---|---|---|
| ● | ST | Stops Transactions | Passed |
| ● | OTUT | Transfers User's Tokens | Passed |
| ● | ELFM | Exceeds Fees Limit | Passed |
| ● | MT | Mints Tokens | Passed |
| ● | BT | Burns Tokens | Passed |
| ● | BC | Blacklists Addresses | Passed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

## Diagnostics

● Critical ● Medium ● Minor / Informative

| Severity | Code | Description | Status |
|---|---|---|---|
| ● | OCTD | Transfers Contract's Tokens | Acknowledged |
| ● | CR | Code Repetition | Acknowledged |
| ● | MMN | Misleading Method Naming | Acknowledged |
| ● | PMRM | Potential Mocked Router Manipulation | Acknowledged |
| ● | PVC | Price Volatility Concern | Acknowledged |
| ● | RRA | Redundant Repeated Approvals | Acknowledged |
| ● | RSD | Redundant Swap Duplication | Acknowledged |
| ● | L02 | State Variables could be Declared Constant | Acknowledged |
| ● | L04 | Conformance to Solidity Naming Conventions | Acknowledged |
| ● | L07 | Missing Events Arithmetic | Acknowledged |
| ● | L13 | Divide before Multiply Operation | Acknowledged |
| ● | L18 | Multiple Pragma Directives | Acknowledged |
| ● | L19 | Stable Compiler Version | Acknowledged |
| ● | L20 | Succeeded Transfer Check | Acknowledged |

## Findings Breakdown



| | | |
|---|---|---|
| 🔴 | Critical | 0 |
| 🟠 | Medium | 1 |
| ⚪ | Minor / Informative | 13 |

| | Severity | Unresolved | Acknowledged | Resolved | Other |
|---|---|---|---|---|---|
| 🔴 | Critical | 0 | 0 | 0 | 0 |
| 🟠 | Medium | 0 | 1 | 0 | 0 |
| ⚪ | Minor / Informative | 0 | 13 | 0 | 0 |

## OCTD - Transfers Contract's Tokens

| | |
|---|---|
| **Criticality** | Medium |
| **Location** | MemecoifContract.sol#L1320,1326 |
| **Status** | Acknowledged |

### Description

The contract owner has the authority to claim all the balance of the contract by transferring it to pool1Wallet wallet. The owner may take advantage of it by calling the `transferERC20TokenFromContractAddressToPool1` and `transferBNBFromContractAddressToPool1` functions. Additionally, an inconsistency may occur between the contract balance and the accumulated fees. The variables `collectedAmountLiquidityFee`, `collectedAmountMarketingFee`, `collectedAmountPool1Fee`, `collectedAmountPool2Fee`, and `collectedAmountPool3Fee` are designed to accumulate tokens from fees. However, the token contract can be withdrawn from the contract and in this case, the accumulated fee variables are not initialized causing `distributeCollectedFees` to fail.

```
function transferERC20TokenFromContractAddressToPool1(address
_tokenERC20) public onlyOwner {
	ERC20 tokenERC20 = ERC20(_tokenERC20);
	uint256 amount = tokenERC20.balanceOf(address(this));
	tokenERC20.transfer(pool1Wallet, amount);
}

function transferBNBFromContractAddressToPool1() public onlyOwner {
	address payable pool1WalletBNB = payable(pool1Wallet);
	pool1WalletBNB.transfer(address(this).balance);
}
```

## Recommendation

It is recommended to implement a mechanism that updates the balance when tokens are withdrawn from the contract. This can be achieved by adjusting the accumulated fee variables whenever a withdrawal is made. Alternatively, the contract could be modified to disallow the withdrawal of the token from the contract's address. This would ensure that the balance of the contract and the accumulated fees remain consistent, reducing the potential for exploitation and improving the overall security and reliability of the contract. Additionally, 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. Some suggestions are:

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

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

## Team Update

The two mentioned functions will only be used in emergency situations and NOT for MEMECOIF tokens. If a user accidentally sends an ERC20 token or BNB to the smart contract address, the team will withdraw these from the contract and return them to the user. The accumulated fees remain consistent, forever.

## CR - Code Repetition

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1672 |
| **Status** | Acknowledged |

### 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 contract is using repetitive code segments in the segment where `_processPool1Active` or `_processPool2Active` is true regarding the pool1 and pool2 operations. The code segments for `_processPool1Active` and `_processPool2Active` are almost identical, differing only in the specific pool they are processing.

```
if (_processPool1Active){
    if (shareholderIndexes[_shareholder] <
_payoutPool1ShareholderCount) {
        if (currentIndexPool1 < shareholderIndexes[_shareholder]) {
            if(shares[_shareholder].amount < _amountNew){
                shares[_shareholder].amountExcludedBuyPool1 =
(_amountNew - shares[_shareholder].amount) +
shares[_shareholder].amountExcludedBuyPool1;
            }
        }
    }
}

if (_processPool2Active){
    if (shareholderIndexes[_shareholder] <
_payoutPool2ShareholderCount) {
        if (currentIndexPool2 < shareholderIndexes[_shareholder]) {
            if(shares[_shareholder].amount < _amountNew) {
                shares[_shareholder].amountExcludedBuyPool2 =
(_amountNew - shares[_shareholder].amount) +
shares[_shareholder].amountExcludedBuyPool2;
            }
        }
    }
}
```

## 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.

## Team Update

The team has acknowledged that this is not a security issue and states:

"It will become significantly more complex if we were to extract this portion of the code into an internal function, as it would require passing several parameters (for Pool 1 or Pool 2) as well: _processPool1Active, _payoutPool1ShareholderCount, currentIndexPool1, amountExcludedBuyPool1"

## MMN - Misleading Method Naming

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1044,2007 |
| **Status** | Acknowledged |

### Description

Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand. Methods can have misleading names if their names do not accurately reflect the functionality they contain or the purpose they serve. The contract uses some method names that are too generic or do not clearly convey the underneath functionality. Misleading method names can lead to confusion, making the code more difficult to read and understand.

Specifically, the contract is utilizing the `pool3BurnAddress` to distribute tokens. This address is initially set to the zero address, but the contract owner has the ability to change this address by calling the `setPool3BurnAddress` function. This could potentially lead to confusion or misuse, as the `pool3BurnAddress` may no longer represent a burn address but could be set to any arbitrary address. This is particularly concerning because the function names `setPool3BurnAddress` and `updatePool3BurnAddress` suggest that these addresses are specifically for burning tokens, but the implementation allows for a broader range of functionality.

Furthermore, the `updatePool3BurnAddress` function emits an event `Pool3BurnAddressUpdated` whenever the burn address is updated. This event could be misleading if the updated address is not actually a burn address. This discrepancy between the function and variable names and their actual functionality can lead to confusion and misinterpretation of the contract's behavior, making the code more difficult to read and understand.

```
function setPool3BurnAddress(address _newPool3BurnAddress) public
onlyOwner validateAddressNotZero(_newPool3BurnAddress) {

poolDistributor.updatePool3BurnAddress(_newPool3BurnAddress);
}
```

```
function updatePool3BurnAddress(address _newPool3BurnAddress)
external onlyOwner {
        require(_newPool3BurnAddress != pool3BurnAddress, "Error:
The pool3BurnAddress is already this address");
        emit Pool3BurnAddressUpdated(_newPool3BurnAddress,
pool3BurnAddress);
        pool3BurnAddress = _newPool3BurnAddress;
 }
```

## Recommendation

It's always a good practice for the contract to contain method names that are specific and descriptive. It is recommended to rename both the function and the variable to more accurately reflect their actual implementation.This would make it clear that these addresses are used for distribution, not necessarily for burning. Additionally, the event `Pool3BurnAddressUpdated` could be renamed to avoid any confusion. By doing so, the contract would become more self-explanatory and easier to understand, reducing the risk of misuse or misinterpretation.

## Team Update

The team has acknowledged that this is not a security issue and states:
"In this case, pool3BurnAddress is a vesting contract with lockDuration with max value. pool3BurnAddress is used as a "special" burn address as it is not possible to release the tokens once they have been sent to this vesting contract. After deployment of COIF contract pool3BurnAddress will be updated to this special vesting contract. This is done only once.

Please see an example for old deployed COIF contract:
https://bscscan.com/address/0xb2ca5c351242c2e526fc5d89656848ce3511a79e#code
contract Pool3BurnForever is TokenVesting { //lockDuration = near to max value of uint256, vestingDuration = 1 (zero not possible) uint256 private pool3LockDuration = type(uint256).max - block.timestamp 2;* constructor ()

TokenVesting(address(0xfD92637A67cfCbd7eE0c79056325e3701123d5A2), block.timestamp, pool3LockDuration, 1) { } }

For sure, there are other possibilities to implement special burn functionality.But there is no inconsistency between the function and variable names and their actual functionality in this case."

## PMRM - Potential Mocked Router Manipulation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1079 |
| **Status** | Acknowledged |

### Description

The contract includes a method that allows the owner to modify the router address and create a new pair. While this feature provides flexibility, it introduces a security threat. The owner could set the router address to any contract that implements the router's interface, potentially containing malicious code. In the event of a transaction triggering the swap functionality with such a malicious contract as the router, the transaction may be manipulated.

```
function updateUniswapV2Router(address _newAddress) public onlyOwner {
        require(_newAddress != address(uniswapV2Router), "Error: The router
already has that address");
        emit UpdateUniswapV2Router(_newAddress, address(uniswapV2Router));
        uniswapV2Router = IUniswapV2Router02(_newAddress);
        excludeFromDividends(_newAddress,true);
}
```

### 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.

## Team Update

The team has acknowledged that this is not a security issue and states:
“It is not intended to call this function if it is not required for certain PancakeSwap updates. Additionally, we will transfer ownership of the contract to multi-signature wallet in the future.”

## PVC - Price Volatility Concern

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1072,1363 |
| **Status** | Acknowledged |

### Description

The contract accumulates tokens from the taxes to swap them for ETH. The variable `swapFeeTokensMinAmount` 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 setSwapFeeTokensMinAmount(uint256 _swapMinAmount)
public onlyOwner {
        require(_swapMinAmount <= (10**18), "Error: use the value
without 10**18, e.g. 10000 for 10000 tokens");
        require(_swapMinAmount <= 100_000_000, "Error: token amount
exceeds total supply");
        swapFeeTokensMinAmount = _swapMinAmount * 10**18;
        emit SetSwapFeeTokensMinAmount(swapFeeTokensMinAmount);
    }
...
  if(
            tradingIsEnabled &&
            (balanceOf(address(this))>=swapFeeTokensMinAmount) &&
            !feesSwapping &&
            !automatedMarketMakerPairs[from] &&
            !excludedFromFees[from] &&
            !excludedFromFees[to]
        ) {
            feesSwapping = true;
            distributeCollectedFees(
                collectedAmountLiquidityFee,
                collectedAmountMarketingFee,
                collectedAmountPool1Fee,
                collectedAmountPool2Fee,
                collectedAmountPool3Fee
            );
            feesSwapping = false;
        }
```

## 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 total supply. Hence, the contract will guarantee that it cannot accumulate a huge amount of tokens in order to sell them.

## Team Update

The team has acknowledged that this is not a security issue and states:
"The default value of swapFeeTokensMinAmount is set to 10,000 COIF. This corresponds to 0.01% of the total supply (100,000,000). The team will further reduce the value as the price increases. There's no need to make the code more complex."

## RRA - Redundant Repeated Approvals

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1542,1557,1573 |
| **Status** | Acknowledged |

### 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 swapTokensForBNB(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

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.

## Team Update

The team has acknowledged that this is not a security issue and states:
“These parts of the code were taken from the COIF contract:
https://bscscan.com/address/0xb2a575d2c78c3ca53ab43e0cab5340f1dcf5b7f1#code
The contract runs without errors on the blockchain. To avoid potential issues when modifying the contract, this section of the code will not be changed. The contract will be deployed with Compiler Version v0.8.17.”

## RSD - Redundant Swap Duplication

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1496 |
| **Status** | Acknowledged |

### Description

The contract contains multiple swap methods that individually perform token swaps and transfer promotional amounts to specific addresses and features. This redundant duplication of code introduces unnecessary complexity and increases dramatically the gas consumption. By consolidating these operations into a single swap method, the contract can achieve better code readability, reduce gas costs, and improve overall efficiency.

```
     if(_collectedAmountMarketingFeeDist > 0) {

            swapAndSendFeeWBNB(_collectedAmountMarketingFeeDist,
marketingWallet);
        }

        if(_collectedAmountPool1FeeDist > 0) {
            swapAndSendFeeWBNB(_collectedAmountPool1FeeDist,
pool1Wallet);
        }

        if(_collectedAmountPool2FeeDist > 0) {
            swapAndSendFeeWBNB(_collectedAmountPool2FeeDist,
poolDistributorAddress);
        }
```

### Recommendation

A more optimized approach could be adopted to perform the token swap operation once for the total amount of tokens and distribute the proportional amounts to the corresponding addresses, eliminating the need for separate swaps.

## Team Update

The team has acknowledged that this is not a security issue and states:
"If we were to do the token swap operation once, we would need to convert WBNB to the individual fee parts. The code readability will be significantly worsened. In this particular case, the gas savings will not be significant. And since this part of the code isn't executed on every transfer, but only when a certain threshold of accumulated fees is reached, the code can remain in the more readable form. In addition, it was planned that the MarketingWallet would get a better conversion rate compared to Pool 1, Pool 2 would get the worst rate. This is also guaranteed with the current code because the rate deteriorates with each swap."

## L02 - State Variables could be Declared Constant

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L867 |
| **Status** | Acknowledged |

### 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 public totalSupplyInit = 100_000_000 * (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.

### Team Update

The team has acknowledged that this is not a security issue and states:
“The status variable totalSupplyInit is only used in the removeLimits() function.
This function is called once to remove limits (24 hours after the start) and then never again.
The change is not necessary.”

## L04 - Conformance to Solidity Naming Conventions

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L8,165,166,183,862,938,949,950,951,955,1004,1011,1018,1025,1032,1039,1044,1048,1052,1056,1060,1064,1068,1072,1079,1086,1091,1095,1102,1118,1130,1135,1136,1137,1138,1139,1150,1151,1152,1153,1154,1165,1166,1167,1168,1169,1179,1184,1191,1200,1208,1213,1218,1277,1290,1310,1316,1320,1480,1481,1482,1483,1484,1529,1542,1557,1573,1660,1661,1662,1663,1664,1665,1704,1709,1719,1720,1721,1729,1730,1731,1732,1733,1734,1785,1786,1787,1788,1819,1820,1821,1822,1861,1862,1863,1902,1914,1937,1959,1975,1981,1991,2001,2007,2013,2019,2025,2031,2037,2043 |
| **Status** | Acknowledged |

### 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);
function DOMAIN_SEPARATOR() external view returns (bytes32);
function PERMIT_TYPEHASH() external pure returns (bytes32);
function MINIMUM_LIQUIDITY() external pure returns (uint);
uint8 private constant _decimals = 18
event isExcludeFromDividends(address indexed account, bool
isExcluded);
event updatedBuyFees(uint256 newBuyLiquidityFee, uint256
newBuyMarketingFee, uint256 newBuyPool1Fee, uint256 newBuyPool2Fee,
uint256 newBuyPool3Fee);
event updatedSellFees(uint256 newSellLiquidityFee, uint256
newSellMarketingFee, uint256 newSellPool1Fee, uint256
newSellPool2Fee, uint256 newSellPool3Fee);
event updatedTxFees(uint256 newTxLiquidityFee, uint256
newTxMarketingFee, uint256 newTxPool1Fee, uint256 newTxPool2Fee,
uint256 newTxPool3Fee);
event triggeredPool1Payout(bool indexed newProcessPool1Trigger,
address indexed newProcessPool1Token, uint256 indexed
newProcessPool1StartTime);
address _newLiquidityWallet
address _newMarketingWallet
address _newPool1Wallet
address _newIdoWallet1

...
```

## 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.

## Team Update

The team has acknowledged that this is not a security issue and states:
"Modifying the current contract would require too many changes. The risk of introducing errors in the process is too high."

## L07 - Missing Events Arithmetic

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1692 |
| **Status** | Acknowledged |

### 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.

```
totalShares = totalShares - shares[_shareholder].amount +
_amountNew
```

### 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.

### Team Update

The team has acknowledged that this is not a security issue and states:
"No event necessary because this function is updated after every transfer (buy/sell/tx): function setShare"

## L13 - Divide before Multiply Operation

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1458,1460,1977,1978 |
| **Status** | Acknowledged |

### 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.

```
payoutPool2CurrentWBNB = (pool2BalanceWBNB * payoutPool2Percent) /
100
payoutPool2DividendsPerShare = (payoutPool2CurrentWBNB *
poolDistributor.dividendsPerShareAccuracyFactor()) /
poolDistributor.totalShares()
```

### 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.

### Team Update

The team has acknowledged that this is not a security issue and states:
"First multiply then divide is valid for all cases in this finding:
1. payoutPool2CurrentWBNB , payoutPool2DividendsPerShare
2. function getUnpaidDividendsFromPool2
The rule is being followed, because the first calculated value is used in the second."

## L18 - Multiple Pragma Directives

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L4,102,148,203,226,253,338,419,449,814 |
| **Status** | Acknowledged |

### 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 >=0.6.2;
pragma solidity >=0.5.0;
pragma solidity ^0.8.0;
pragma solidity 0.8.17;
```

### 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.

### Team Update

The team has acknowledged that this is not a security issue and states:
“These parts of the code were taken from the COIF contract:
https://bscscan.com/address/0xb2a575d2c78c3ca53ab43e0cab5340f1dcf5b7f1#codeThe contract runs without errors on the blockchain. To avoid potential issues when modifying the contract, this section of the code will not be changed. The contract will be deployed with Compiler Version v0.8.17.”

## L19 - Stable Compiler Version

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L253 |
| **Status** | Acknowledged |

### 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 ^0.8.0;
```

### 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.

### Team Update

The team has acknowledged that this is not a security issue and states:
“These parts of the code were taken from the COIF contract:
https://bscscan.com/address/0xb2a575d2c78c3ca53ab43e0cab5340f1dcf5b7f1#code
The contract runs without errors on the blockchain. To avoid potential issues when modifying the contract, this section of the code will not be changed. The contract will be deployed with Compiler Version v0.8.17.”

## L20 - Succeeded Transfer Check

| | |
|---|---|
| **Criticality** | Minor / Informative |
| **Location** | MemecoifContract.sol#L1323,1725,1841,1844,1847,1850,1853,1882,1885,1888,1891,1894 |
| **Status** | Acknowledged |

### 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.

```
tokenERC20.transfer(pool1Wallet, amount)
pool1TokenERC20.transfer(_pool1Wallet, amount)
processPool1TokenERC20.transfer(pool3Wallet, amount)
processPool1TokenERC20.transfer(teamWallet, amount)
processPool1TokenERC20.transfer(longTermGrowthWallet, amount)
processPool1TokenERC20.transfer(ecosystemWallet, amount)
processPool1TokenERC20.transfer(_shareholder, amount)
WBNB.transfer(pool3Wallet, amount)
WBNB.transfer(teamWallet, amount)
WBNB.transfer(longTermGrowthWallet, amount)
WBNB.transfer(ecosystemWallet, amount)
WBNB.transfer(_shareholder, amount)
```

### 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.

### Team Update

The team has acknowledged that this is not a security issue and states:
"It's possible that a _shareholder is a smart contract (not an EOA) that doesn't allow transfers of WBNB or specific ERC20 tokens. In this case the method “safeTransfer” would stop processPool1 and processPool2 by revert of transaction. To ensure the process continues, we would need to manually exclude the problematic account from receiving dividends. However, we would like to avoid doing that. To ensure that processPool1 and

processPool2 don't fail (no revert!), we have to keep using „transfer“ method.
In addition, safeTransfer from SafeERC20 typically consumes more gas than the regular transfer from ERC20. This is because safeTransfer performs additional checks to ensure the transfer is successful before executing. These additional checks increase the gas consumption cost compared to directly using transfer. And we should try to use less gas during processPool1 and processPool2 execution."

## Functions Analysis

| Contract | Type | Bases | | |
|---|---|---|---|---|
| | **Function Name** | **Visibility** | **Mutability** | **Modifiers** |
| | | | | |
| **MemecoifContract** | Implementation | ERC20, Ownable | | |
| | | Public | ✓ | ERC20 |
| | | External | Payable | - |
| | setLiquidityWallet | Public | ✓ | onlyOwner validateAddressNotZero |
| | setMarketingWallet | Public | ✓ | onlyOwner validateAddressNotZero |
| | setPool1Wallet | Public | ✓ | onlyOwner validateAddressNotZero |
| | setIdoWallet1 | Public | ✓ | onlyOwner validateAddressNotZero |
| | setIdoWallet2 | Public | ✓ | onlyOwner validateAddressNotZero |
| | setPool3Wallet | Public | ✓ | onlyOwner validateAddressNotZero |
| | setPool3BurnAddress | Public | ✓ | onlyOwner validateAddressNotZero |
| | setTeamWallet | Public | ✓ | onlyOwner validateAddressNotZero |
| | setLongTermGrowthWallet | Public | ✓ | onlyOwner validateAddressNotZero |

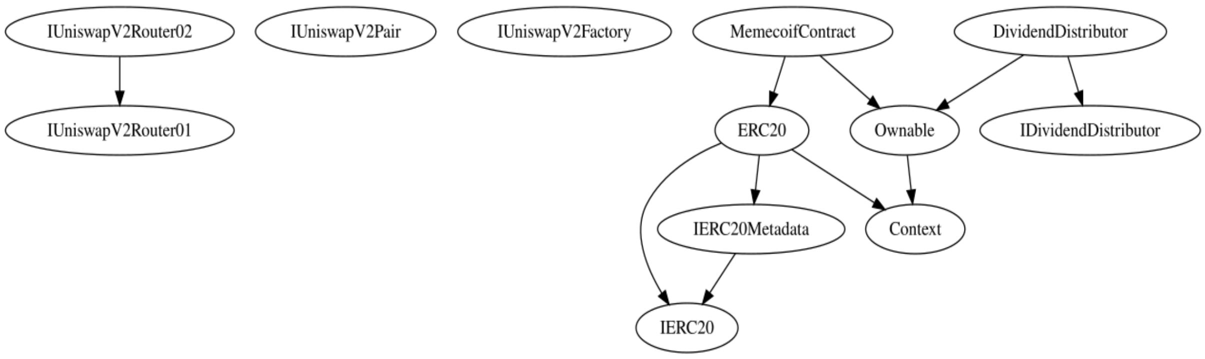
| | | | | |
|---|---|---|---|---|
| | setEcosystemWallet | Public | ✓ | onlyOwner validateAddress NotZero |
| | setTeamLockAddress | Public | ✓ | onlyOwner validateAddress NotZero |
| | setLongTermGrowthLockAddress | Public | ✓ | onlyOwner validateAddress NotZero |
| | setEcosystemLockAddress | Public | ✓ | onlyOwner validateAddress NotZero |
| | setSwapFeeTokensMinAmount | Public | ✓ | onlyOwner |
| | updateUniswapV2Router | Public | ✓ | onlyOwner |
| | excludeFromFees | Public | ✓ | onlyOwner |
| | isExcludedFromFees | Public | | - |
| | excludeMultipleAccountsFromFees | Public | ✓ | onlyOwner |
| | setAutomatedMarketMakerPair | Public | ✓ | onlyOwner |
| | _setAutomatedMarketMakerPair | Private | ✓ | |
| | excludeFromDividends | Public | ✓ | onlyOwner |
| | isExcludedFromDividends | Public | | - |
| | updateBuyFees | Public | ✓ | onlyOwner |
| | updateSellFees | Public | ✓ | onlyOwner |
| | updateTxFees | Public | ✓ | onlyOwner |
| | setTradeFeeStatus | Public | ✓ | onlyOwner |
| | setPayoutGas | Public | ✓ | onlyOwner |
| | setPayoutPool2Percent | Public | ✓ | onlyOwner |
| | setPayoutPool2MinAmountWBNB | Public | ✓ | onlyOwner |

| | | | | |
|---|---|---|---|---|
| | setMinimumBalanceForDividends | Public | ✓ | onlyOwner |
| | setPayoutPool2FrequencySec | Public | ✓ | onlyOwner |
| | triggerPool1Payout | Public | ✓ | onlyOwner |
| | getCurrentInfoAboutPool1 | Public | | - |
| | getCurrentInfoAboutPool2 | Public | | - |
| | getAccountDividendsInfoForPool2 | Public | | - |
| | getAccountDividendsInfoForPool2AtIndex | Public | | - |
| | launch | Public | ✓ | onlyOwner |
| | setCanTransferBeforeTradingIsEnabled | Public | ✓ | onlyOwner |
| | transferERC20TokenFromPool2ToPool1 | Public | ✓ | onlyOwner |
| | transferERC20TokenFromContractAddressToPool1 | Public | ✓ | onlyOwner |
| | transferBNBFromContractAddressToPool1 | Public | ✓ | onlyOwner |
| | _transfer | Internal | ✓ | |
| | removeLimits | External | ✓ | onlyOwner |
| | distributeCollectedFees | Private | ✓ | |
| | swapAndLiquify | Private | ✓ | |
| | swapTokensForBNB | Private | ✓ | |
| | swapAndSendFeeWBNB | Private | ✓ | |
| | addLiquidity | Private | ✓ | |
| | getCollectedFeeAmounts | Public | | - |
| | | | | |

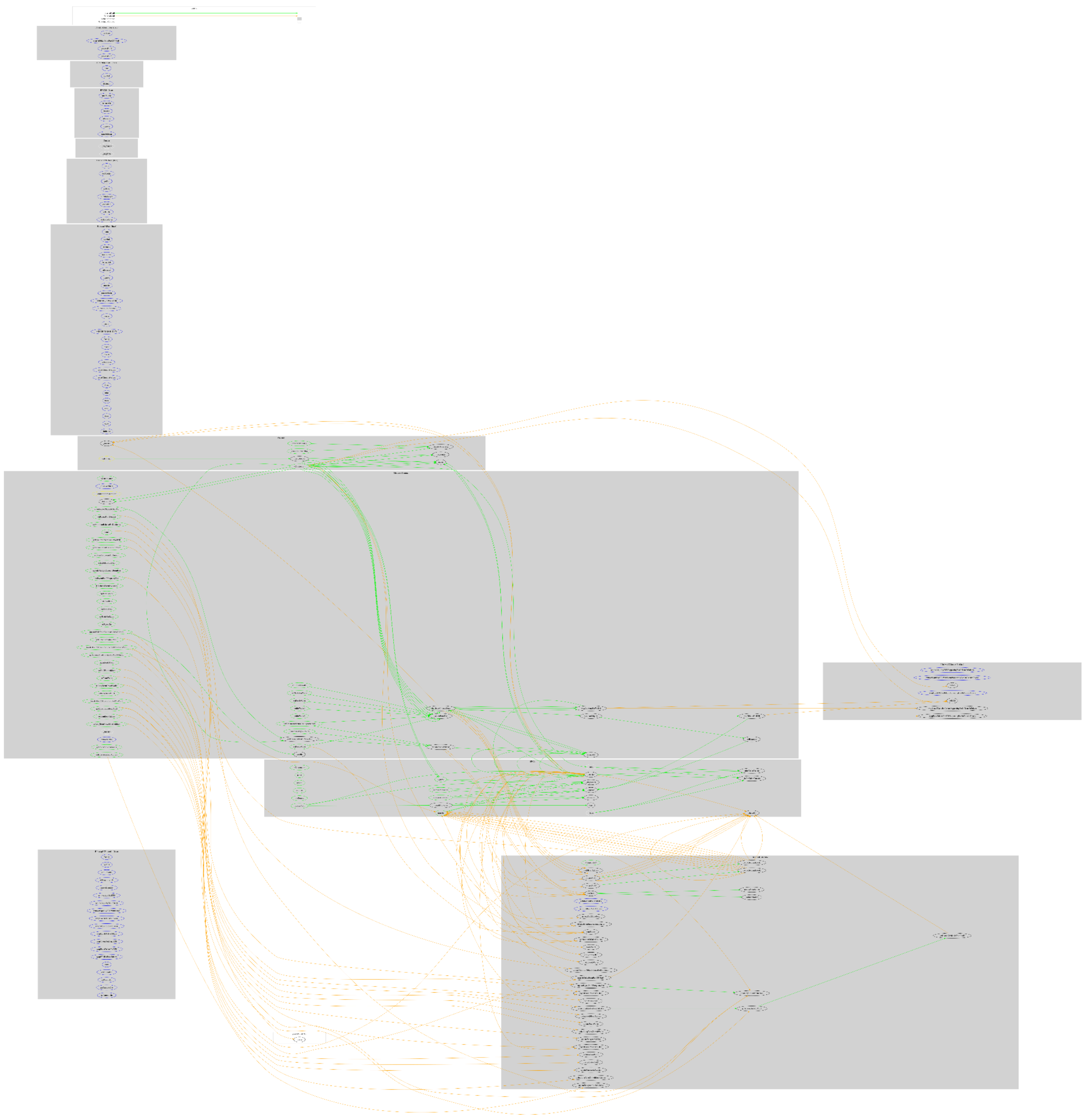
| | | | | |
|---|---|---|---|---|
| **DividendDistributor** | Implementation | IDividendDistributor, Ownable | | |
| | | Public | ✓ | - |
| | setShare | External | ✓ | onlyOwner |
| | addShareholder | Internal | ✓ | |
| | removeShareholder | Internal | ✓ | |
| | transferTokenFromPool2ToPool1 | External | ✓ | onlyOwner |
| | processPool1 | External | ✓ | onlyOwner |
| | processPool2 | External | ✓ | onlyOwner |
| | payoutDividendsPool1 | Internal | ✓ | |
| | payoutDividendsPool2 | Internal | ✓ | |
| | getInfoAboutPool1AtIndex | External | | - |
| | getInfoAboutPool1AtToken | External | | - |
| | getInfoAboutPool2 | External | | - |
| | getAccountInfoForPool2 | Public | | - |
| | getAccountInfoForPool2AtIndex | External | | - |
| | getUnpaidDividendsFromPool2 | Public | | - |
| | updatePayoutPool2FrequencySec | External | ✓ | onlyOwner |
| | updatePayoutPool2TimeNext | Public | ✓ | onlyOwner |
| | updateMinimumTokenBalanceForDividends | External | ✓ | onlyOwner |
| | getNumberOfTokenHolders | External | | - |
| | updatePool3Wallet | External | ✓ | onlyOwner |
| | updatePool3BurnAddress | External | ✓ | onlyOwner |

| | | | | |
|---|---|---|---|---|
| | updateTeamWallet | External | ✓ | onlyOwner |
| | updateLongTermGrowthWallet | External | ✓ | onlyOwner |
| | updateEcosystemWallet | External | ✓ | onlyOwner |
| | updateTeamLockAddress | External | ✓ | onlyOwner |
| | updateLongTermGrowthLockAddress | External | ✓ | onlyOwner |
| | updateEcosystemLockAddress | External | ✓ | onlyOwner |

## Inheritance Graph



## Flow Graph



## Summary

MEMECOIF contract implements a token mechanism. This audit investigates security issues, business logic concerns and potential improvements. MEMECOIF is an interesting project that has a friendly and growing community. The Smart Contract analysis reported no compiler error or critical issues. There is also a limit of max 10% fees.

## 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**

cyberscope.io