Cyberscope

Audit Report

Rabbits Network

September 2024

Network	BSC
Address	0x5899B9aEB098bC3835F1Dc2e34D6813b79cee9A9
Audited by	© cyberscope

Analysis

Critical Medium Minor / Informative Pass

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

Diagnostics

Critical Medium Minor / Informative

Severity	Code	Description	Status
Critical	USF	Unlocked Swap Functionality	Unresolved
Medium	IRF	Insufficient Reward Funds	Unresolved
Medium	POE	Potential Out-of-Gas Error	Unresolved
Medium	RV	Randomization Vulnerability	Unresolved
Minor / Informative	AOI	Arithmetic Operations Inconsistency	Unresolved
Minor / Informative	CCR	Contract Centralization Risk	Unresolved
Minor / Informative	DDP	Decimal Division Precision	Unresolved
Minor / Informative	IDI	Immutable Declaration Improvement	Unresolved
Minor / Informative	MC	Missing Check	Unresolved
Minor / Informative	MEE	Missing Events Emission	Unresolved
Minor / Informative	PBV	Percentage Boundaries Validation	Unresolved
Minor / Informative	PLPI	Potential Liquidity Provision Inadequacy	Unresolved
Minor / Informative	PNRP	Potential nftContract Revert Propagation	Unresolved
Minor / Informative	PVC	Price Volatility Concern	Unresolved

●	RRA	Redundant Repeated Approvals	Unresolved
●	RSML	Redundant SafeMath Library	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
●	L16	Validate Variable Setters	Unresolved
●	L19	Stable Compiler Version	Unresolved

Table of Contents

Risk Classification

The criticality of findings in Cyberscope's smart contract audits is determined by evaluating multiple variables. The two primary variables are:

1. **Likelihood of Exploitation**: This considers how easily an attack can be executed, including the economic feasibility for an attacker.
2. **Impact of Exploitation**: This assesses the potential consequences of an attack, particularly in terms of the loss of funds or disruption to the contract's functionality.

Based on these variables, findings are categorized into the following severity levels:

1. **Critical**: Indicates a vulnerability that is both highly likely to be exploited and can result in significant fund loss or severe disruption. Immediate action is required to address these issues.
2. **Medium**: Refers to vulnerabilities that are either less likely to be exploited or would have a moderate impact if exploited. These issues should be addressed in due course to ensure overall contract security.
3. **Minor**: Involves vulnerabilities that are unlikely to be exploited and would have a minor impact. These findings should still be considered for resolution to maintain best practices in security.
4. **Informative**: Points out potential improvements or informational notes that do not pose an immediate risk. Addressing these can enhance the overall quality and robustness of the contract.

Severity	Likelihood / Impact of Exploitation
Critical	Highly Likely / High Impact
Medium	Less Likely / High Impact or Highly Likely/ Lower Impact
Minor / Informative	Unlikely / Low to no Impact

Review

Contract Name	RabbitNetwork
Compiler Version	v0.8.21+commit.d9974bed
Optimization	200 runs
Explorer	https://bscscan.com/address/0x5899b9aeb098bc3835f1dc2e34d6813b79cee9a9
Address	0x5899b9aeb098bc3835f1dc2e34d6813b79cee9a9
Network	BSC
Symbol	RBT.network
Decimals	18
Total Supply	100,000,000,000
Badge Eligibility	Must Fix Criticals

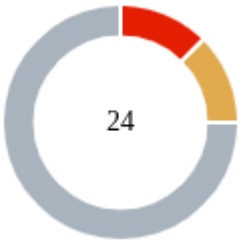
Audit Updates

Initial Audit	17 Sep 2024

Source Files

Filename	SHA256
RabbitNetwork.sol	825a0b26cb749e909d3b3acf1cdc8fcb24eacc4b87e421c91af149b89734f41a

Findings Breakdown



Critical	3
Medium	3
Minor / Informative	18

Severity	Unresolved	Acknowledged	Resolved	Other
Critical	3	0	0	0
Medium	3	0	0	0
Minor / Informative	18	0	0	0

ST - Stops Transactions

Criticality	Critical
Location	RabbitNetwork.sol#L1856,1869,1795
Status	Unresolved

Description

The contract owner has the authority to stop transactions, as described in detail in sections `ELFM` and `PNRP`. As a result, the contract might operate as a honeypot.

Recommendation

The team is advised to follow the recommendations outlined in the `ELFM` and `PNPM` findings and implement the necessary steps to mitigate the identified risks, ensuring that the contract does not operate as a honeypot. Renouncing ownership will effectively eliminate the threats, but it is non-reversible.

ELFM - Exceeds Fees Limit

Criticality	Critical
Location	RabbitNetwork.sol#L1856,1869
Status	Unresolved

Description

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

```
    // Update the total buy tax and allocations
    function updateBuyTax(
        uint256 buyTaxToHolders,
        uint256 buyTaxToMarketing
    ) external onlyOwner {
        uint256 buyTax = buyTaxToHolders + buyTaxToMarketing;
        _buyTax = buyTax;
        _buyTaxToHolders = buyTaxToHolders;
        _buyTaxToMarketing = buyTaxToMarketing;

        emit BuyTaxUpdated(buyTax, buyTaxToHolders,
buyTaxToMarketing);
    }

    // Update the total sell tax and allocations
    function updateSellTax(
        uint256 sellTaxToHolders,
        uint256 sellTaxToMarketing,
        uint256 sellTaxToBuyback
    ) external onlyOwner {
        uint256 sellTax = sellTaxToHolders + sellTaxToMarketing
+ sellTaxToBuyback;

        _sellTax = sellTax;
        _sellTaxToHolders = sellTaxToHolders;
        _sellTaxToMarketing = sellTaxToMarketing;
        _sellTaxToBuyback = sellTaxToBuyback;

        emit SellTaxUpdated(sellTax, sellTaxToHolders,
sellTaxToMarketing, sellTaxToBuyback);
    }
```

Recommendation

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

Temporary Solutions:

These measurements do not decrease the severity of the finding

- Introduce a time-locker mechanism with a reasonable delay.

- Introduce a multi-signature wallet so that many addresses will confirm the action.
- Introduce a governance model where users will vote about the actions.

Permanent Solution:

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

USF - Unlocked Swap Functionality

Criticality	Critical
Location	RabbitNetwork.sol#L1710,1755
Status	Unresolved

Description

The contract contains functionality within the `_swapAndDistribute` function to swap tokens for ETH and then for USDT on each transfer transaction. However, this swap function is not locked, which creates a reentrancy vulnerability, as it triggers a reentrant call to the token contract. Additionally, the swap is currently triggered on both buy and sell transactions. This design increases the risk of exploitation and inefficiency, particularly due to the lack of proper restriction on when the swap should occur.

```
 if (!_isExcludedFromFee[sender] &&
!_isExcludedFromFee[recipient]) {
            ...

                if (recipient == uniswapV2Pair) {
                    // Sell tax allocation
                    taxForHolders =
taxAmount.mul(_sellTaxToHolders).div(_sellTax);
                    taxForMarketing =
taxAmount.mul(_sellTaxToMarketing).div(_sellTax);
                    taxForBuyback =
taxAmount.mul(_sellTaxToBuyback).div(_sellTax);
                } else if (sender == uniswapV2Pair) {
                    // Buy tax allocation
                    taxForHolders =
taxAmount.mul(_buyTaxToHolders).div(_buyTax);
                    taxForMarketing =
taxAmount.mul(_buyTaxToMarketing).div(_buyTax);
                }

                if (taxForHolders > 0) {
                    _swapAndDistribute(taxForHolders);
                }
        ...
   function _swapAndDistribute(uint256 amount) private {
        address[] memory path = new address[](3);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();
        path[2] = USDT;

        _approve(address(this), address(uniswapV2Router),
amount);


uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
            amount,
            0,
            path,
            address(this),
            block.timestamp
        );

        uint256 usdtBalance =
IERC20(USDT).balanceOf(address(this));
        _distributeRewards(usdtBalance);
    }
```

Recommendation

It is recommended to implement a modifier that locks the swap functionality, ensuring it can only be executed once per transaction, thus preventing reentrancy attacks. Additionally, the swap functionality should be restricted to trigger only on sell transactions. This would enhance both the security and efficiency of the contract.

IRF - Insufficient Reward Funds

Criticality	Medium
Location	RabbitNetwork.sol#L1711
Status	Unresolved

Description

The contract is utilizing the `_distributeRewards` function to allocate USDT rewards among holders based on their token balances. This function applies a reward multiplier if the holder owns an NFT from a specific collection. However, the multiplier is determined by a random factor generated by the `getRandomMultiplier` function, which introduces variability in the reward calculation. This randomness results in a total reward amount that exceeds the available USDT balance in the contract. Consequently, holders will be credited with USDT rewards that the contract does not have the funds to cover, leading to potential issues where holders cannot claim their full rewards.

```
        for (uint256 i = 0; i < holderCount; i++) {
            address holder = _holders[i];
            uint256 holderBalance = balanceOf(holder);

            if (holderBalance > 0) {
                uint256 holderReward =
usdtAmount.mul(holderBalance).div(totalTokenSupply);

                // Apply reward multiplier if the holder has an
NFT
                if (nftContract.balanceOf(holder) > 0) {
                    holderReward =
holderReward.mul(getRandomMultiplier(holder));
                }

                _rewards[holder] =
_rewards[holder].add(holderReward);
```

Recommendation

It is recommended to reconsider the application of the multiplier rewards to ensure that the contract maintains a sufficient USDT balance to cover all potential rewards, even when

multipliers are applied. This can be achieved by implementing checks or constraints that guarantee the total distributed rewards do not exceed the available USDT balance in the contract. Additionally, consider using a more deterministic approach to reward distribution that aligns with the actual available funds.

POE - Potential Out-of-Gas Error

Criticality	Medium
Location	RabbitNetwork.sol#L1775,1819,1829
Status	Unresolved

Description

There are code segments that could be optimized. A segment may be optimized so that it becomes a smaller size, consumes less memory, executes more rapidly, or performs fewer operations.

The contract is currently designed with two separate `for` loops within the `_distributeRewards` function, which increases the overall transaction cost as the size of the `_holders` array grows. The first loop calculates the total token supply held by all holders, while the second loop distributes the rewards based on this total supply. As the `_holders` array grows, this approach leads to increased gas costs, impacting the contract's efficiency and scalability. Additionally, the `_removeHolder` function, which is called when updating holders, contains a call to `_getHolderIndex`, introducing yet another iteration over the array. The combination of multiple loops could result in high gas consumption and potential out-of-gas errors, especially in scenarios with a large number of token holders.

```
    function _distributeRewards(uint256 usdtAmount) private {
        uint256 holderCount = _holders.length;
        if (holderCount == 0 || usdtAmount == 0) return;

        uint256 totalTokenSupply = 0;
        for (uint256 i = 0; i < holderCount; i++) {
            address holder = _holders[i];
            totalTokenSupply =
totalTokenSupply.add(balanceOf(holder));
        }

        if (totalTokenSupply == 0) return;

        for (uint256 i = 0; i < holderCount; i++) {
            address holder = _holders[i];
            uint256 holderBalance = balanceOf(holder);

            if (holderBalance > 0) {
                uint256 holderReward =
usdtAmount.mul(holderBalance).div(totalTokenSupply);

                // Apply reward multiplier if the holder has an
NFT
                if (nftContract.balanceOf(holder) > 0) {
                    holderReward =
holderReward.mul(getRandomMultiplier(holder));
                }

                _rewards[holder] =
_rewards[holder].add(holderReward);
            }
        }
        ...
        }
    function _updateHolder(address holder) private {
        if (!_isHolder[holder] && balanceOf(holder) > 0) {
            _isHolder[holder] = true;
            _holders.push(holder);
        } else if (_isHolder[holder] && balanceOf(holder) == 0)
{
            _isHolder[holder] = false;
            _removeHolder(holder);
        }
    }

    function _removeHolder(address holder) private {
        uint256 holderIndex = _getHolderIndex(holder);
        _holders[holderIndex] = _holders[_holders.length - 1];
        _holders.pop();
    }
```

Recommendation

The team is advised to take these segments into consideration and rewrite them so the runtime will be more performant. That way it will improve the efficiency and performance of the source code and reduce the cost of executing it.

To improve the contract's efficiency and scalability, it is recommended to implement a gas limit check when distributing rewards to holders. Currently, the distribution process uses two separate loops, which can lead to increased gas consumption as the number of holders grows. By modifying the distribution logic to process rewards in batches, based on the available gas, the contract can avoid out-of-gas errors and high transaction costs. This method ensures that even with a large number of token holders, the rewards can be distributed incrementally across multiple transactions, maintaining the contract's performance and reducing the risk of gas exhaustion.

Additionally, it is recommended to utilize a single `for` loop within the `_distributeRewards` function while updating the `totalTokenSupply` dynamically during the addition or removal of holder addresses. By maintaining the total token supply in real-time, the function can avoid recalculating it during every reward distribution, thus reducing the number of iterations needed and optimizing the gas consumption. This approach will lead to a more efficient and cost-effective execution, especially for contracts with a large number of token holders.

RV - Randomization Vulnerability

Criticality	Medium
Location	RabbitNetwork.sol#L1806
Status	Unresolved

Description

The contract is using an on-chain technique in order to determine random numbers. The blockchain runtime environment is fully deterministic, as a result, the pseudo-random numbers could be predicted.

```
function getRandomMultiplier(address holder) private view
returns (uint256) {
    uint256 random =
uint256(keccak256(abi.encodePacked(block.timestamp, holder))) %
100;
    if (random < x4MultiplierChance) {
        return 4;
    } else if (random < x3MultiplierChance +
x4MultiplierChance) {
        return 3;
    } else if (random < x2MultiplierChance + x3MultiplierChance
+ x4MultiplierChance) {
        return 2;
    } else {
        return baseMultiplier;
    }
}
```

Recommendation

The contract could use an advanced randomization technique that guarantees an acceptable randomization factor. For instance, the Chainlink VRF (Verifiable Random Function). https://docs.chain.link/docs/chainlink-vrf

AOI - Arithmetic Operations Inconsistency

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1690,1726,1860,1874
Status	Unresolved

Description

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

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

```
_mint(msg.sender, 100_000_000_000 * 10**decimals());
...
taxForHolders = taxAmount.mul(_sellTaxToHolders).div(_sellTax);
...
_rewards[holder] = _rewards[holder].add(holderReward);
...
uint256 buyTax = buyTaxToHolders + buyTaxToMarketing;
...
uint256 sellTax = sellTaxToHolders + sellTaxToMarketing +
sellTaxToBuyback;
```

Recommendation

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

CCR - Contract Centralization Risk

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L2144
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.

The contract owner has the authority to exclude any account from paying fees, change the address of the Uniswap V2 pair, and set or modify the address of the NFT contract, granting them significant control over key contract functionalities.

```
  function excludeFromFee(address account, bool excluded)
external onlyOwner {
        _isExcludedFromFee[account] = excluded;
    }

    function updateUniswapV2Pair(address _uniswapV2Pair)
external onlyOwner {
        uniswapV2Pair = _uniswapV2Pair;
    }

    // make a function to set nft contract address
    function setNftcontract(address _nftcontract) external
onlyOwner {
        nftContract = IERC721(_nftcontract);
    }
```

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.

DDP - Decimal Division Precision

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1724
Status	Unresolved

Description

Division of decimal (fixed point) numbers can result in rounding errors due to the way that division is implemented in Solidity. Thus, it may produce issues with precise calculations with decimal numbers.

Solidity represents decimal numbers as integers, with the decimal point implied by the number of decimal places specified in the type (e.g. decimal with 18 decimal places). When a division is performed with decimal numbers, the result is also represented as an integer, with the decimal point implied by the number of decimal places in the type. This can lead to rounding errors, as the result may not be able to be accurately represented as an integer with the specified number of decimal places.

Hence, the splitted shares will not have the exact precision and some funds may not be calculated as expected.

```
if (recipient == uniswapV2Pair) {
    // Sell tax allocation
    taxForHolders =
taxAmount.mul(_sellTaxToHolders).div(_sellTax);
    taxForMarketing =
taxAmount.mul(_sellTaxToMarketing).div(_sellTax);
    taxForBuyback =
taxAmount.mul(_sellTaxToBuyback).div(_sellTax);
} else if (sender == uniswapV2Pair) {
    // Buy tax allocation
    taxForHolders =
taxAmount.mul(_buyTaxToHolders).div(_buyTax);
    taxForMarketing =
taxAmount.mul(_buyTaxToMarketing).div(_buyTax);
}
```

Recommendation

The team is advised to take into consideration the rounding results that are produced from the solidity calculations. The contract could calculate the subtraction of the divided funds in the last calculation in order to avoid the division rounding issue.

IDI - Immutable Declaration Improvement

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1693,1694,1695
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.

```
USDT
marketingWallet
buybackWallet
```

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.

MC - Missing Check

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1905,1910
Status	Unresolved

Description

The contract is processing variables that have not been properly sanitized and checked that they form the proper shape. These variables may produce vulnerability issues.

Specifically, the contract is missing checks to verify that the `uniswapV2Pair` and the `nftContract` address is not set to zero address.

```
   function updateUniswapV2Pair(address _uniswapV2Pair)
external onlyOwner {
        uniswapV2Pair = _uniswapV2Pair;
    }

    // make a function to set nft contract address
    function setNftcontract(address _nftcontract) external
onlyOwner {
        nftContract = IERC721(_nftcontract);
    }
```

Recommendation

The team is advised to properly check the variables according to the required specifications.

MEE - Missing Events Emission

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1987
Status	Unresolved

Description

The contract performs actions and state mutations from external methods that do not result in the emission of events. Emitting events for significant actions is important as it allows external parties, such as wallets or dApps, to track and monitor the activity on the contract. Without these events, it may be difficult for external parties to accurately determine the current state of the contract.

```
   function excludeFromFee(address account, bool excluded)
external onlyOwner {
        _isExcludedFromFee[account] = excluded;
    }

    function updateUniswapV2Pair(address _uniswapV2Pair)
external onlyOwner {
        uniswapV2Pair = _uniswapV2Pair;
    }

    // make a function to set nft contract address
    function setNftcontract(address _nftcontract) external
onlyOwner {
        nftContract = IERC721(_nftcontract);
    }
```

Recommendation

It is recommended to include events in the code that are triggered each time a significant action is taking place within the contract. These events should include relevant details such as the user's address and the nature of the action taken. By doing so, the contract will be more transparent and easily auditable by external parties. It will also help prevent potential issues or disputes that may arise in the future.

PBV - Percentage Boundaries Validation

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1711
Status	Unresolved

Description

The contract utilizes variables for percentage-based calculations that are required for its operations. These variables are involved in multiplication and division operations to determine proportions related to the contract's logic. If such variables are set to values beyond their logical or intended maximum limits, it could result in incorrect calculations. This misconfiguration has the potential to cause unintended behavior or financial discrepancies, affecting the contract's integrity and the accuracy of its calculations.

```
if (recipient == uniswapV2Pair) {
    // Sell transaction
    taxAmount = amount.mul(_sellTax).div(100);
} else if (sender == uniswapV2Pair) {
    // Buy transaction
    taxAmount = amount.mul(_buyTax).div(100);
```

Recommendation

To mitigate risks associated with boundary violations, it is important to implement validation checks for variables used in percentage-based calculations. Ensure that these variables do not exceed their maximum logical values. This can be accomplished by incorporating `require` statements or similar validation mechanisms whenever such variables are assigned or modified. These safeguards will enforce correct operational boundaries, preserving the contract's intended functionality and preventing computational errors.

PLPI - Potential Liquidity Provision Inadequacy

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1755
Status	Unresolved

Description

The contract operates under the assumption that liquidity is consistently provided to the pair between the contract's token and the native currency. However, there is a possibility that liquidity is provided to a different pair. This inadequacy in liquidity provision in the main pair could expose the contract to risks. Specifically, during eligible transactions, where the contract attempts to swap tokens with the main pair, a failure may occur if liquidity has been added to a pair other than the primary one. Consequently, transactions triggering the swap functionality will result in a revert.

```
    function _swapAndDistribute(uint256 amount) private {
        address[] memory path = new address[](3);
        path[0] = address(this);
        path[1] = uniswapV2Router.WETH();
        path[2] = USDT;

        _approve(address(this), address(uniswapV2Router),
amount);


uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTo
kens(
            amount,
            0,
            path,
            address(this),
            block.timestamp
        );
    ....
    }
```

Recommendation

The team is advised to implement a runtime mechanism to check if the pair has adequate liquidity provisions. This feature allows the contract to omit token swaps if the pair does not have adequate liquidity provisions, significantly minimizing the risk of potential failures.

Furthermore, the team could ensure the contract has the capability to switch its active pair in case liquidity is added to another pair.

Additionally, the contract could be designed to tolerate potential reverts from the swap functionality, especially when it is a part of the main transfer flow. This can be achieved by executing the contract's token swaps in a non-reversible manner, thereby ensuring a more resilient and predictable operation.

PNRP - Potential nftContract Revert Propagation

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1795
Status	Unresolved

Description

The contract relies on an external `nftContract` address to check the token balance of a holder using the `balanceOf` function. If the `nftContract` address is not a valid ERC721 contract, or if it is a non-contract address (like an EOA), calling this function will cause a revert since the `balanceOf` function may not exist or behave as expected. This revert will propagate through the contract, potentially causing the functionalities such as reward calculations or token transfers to fail unexpectedly. This issue arises because no validation is performed to ensure that the `nftContract` is a compliant ERC721 contract before invoking its functions.

```
if (nftContract.balanceOf(holder) > 0) {
    holderReward =
holderReward.mul(getRandomMultiplier(holder));
}
```

Recommendation

The contract should tolerate the potential revert from the underlying contracts when the interaction is part of the main transfer flow. This could be achieved by not allowing set contract addresses or by sending the funds in a non-revertable way. To prevent this issue, the contract should validate that the `nftContract` address implements the ERC721 standard before it is assigned. This can be done by checking whether the contract supports the ERC721 interface using the ERC165 standard. By enforcing this validation, the contract can ensure that only valid ERC721 contracts are used, thereby preventing potential reverts during execution and maintaining smooth functionality. Additionally, consider implementing mechanisms to handle failed interactions gracefully if external contracts behave unexpectedly.

PVC - Price Volatility Concern

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1951
Status	Unresolved

Description

The contract accumulates tokens from the taxes to swap them for USDT. The variable `taxForHolders` 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 USDT.

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.

```
if (taxForHolders > 0) {
    _swapAndDistribute(taxForHolders);
}
```

Recommendation

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

RRA - Redundant Repeated Approvals

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1977
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), amount);

uniswapV2Router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
    amount,
    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.

RSML - Redundant SafeMath Library

Criticality	Minor / Informative
Location	RabbitNetwork.sol
Status	Unresolved

Description

SafeMath is a popular Solidity library that provides a set of functions for performing common arithmetic operations in a way that is resistant to integer overflows and underflows.

Starting with Solidity versions that are greater than or equal to 0.8.0, the arithmetic operations revert to underflow and overflow. As a result, the native functionality of the Solidity operations replaces the SafeMath library. Hence, the usage of the SafeMath library adds complexity, overhead and increases gas consumption unnecessarily in cases where the explanatory error message is not used.

```
library SafeMath {...}
```

Recommendation

The team is advised to remove the SafeMath library in cases where the revert error message is not used. Since the version of the contract is greater than `0.8.0` then the pure Solidity arithmetic operations produce the same result.

If the previous functionality is required, then the contract could exploit the `unchecked { ... }` statement.

Read more about the breaking change on https://docs.soliditylang.org/en/stable/080-breaking-changes.html#solidity-v0-8-0-breaking-changes.

L02 - State Variables could be Declared Constant

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1649,1668,1676,1677,1678,1679
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.

```
address public burningWallet;
uint256 public rewardCycleInterval = 24 hours;

...
uint256 private x3MultiplierChance = 15;
uint256 private x4MultiplierChance = 5;
```

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	RabbitNetwork.sol#L1035,1491,1646,1901,1906
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 DOMAIN_SEPARATOR() external view returns (bytes32);
function WETH() external pure returns (address);
address public USDT;
_isExcludedFromFee[account]
address _uniswapV2Pair
address _nftcontract
```

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.

L05 - Unused State Variable

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1872
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.

```
mapping(address => uint256) private _rewardMultiplier;
```

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	RabbitNetwork.sol#L138,548,1080,1134,1143,1174,1244,1252,1261,1276,1310
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 _contextSuffixLength() internal view virtual returns (uint256) {
        return 0;
    }

function _burn(address account, uint256 value) internal {
        if (account == address(0)) {
            revert ERC20InvalidSender(address(0));
        }
        _update(account, address(0), value);
    }



...
```

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.

L16 - Validate Variable Setters

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L1693,1694,1695,1902
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.

```
_usdt;
marketingWallet = _marketingWallet;
buybackWallet = _buybackWallet;
uniswapV2Pair = _uniswapV2Pair;
```

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.

L19 - Stable Compiler Version

Criticality	Minor / Informative
Location	RabbitNetwork.sol#L7,89,117,147,312,630,732,950,1043,1205,1325,1353,1632
Status	Unresolved

Description

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

```
pragma solidity ^0.8.20;
...
```

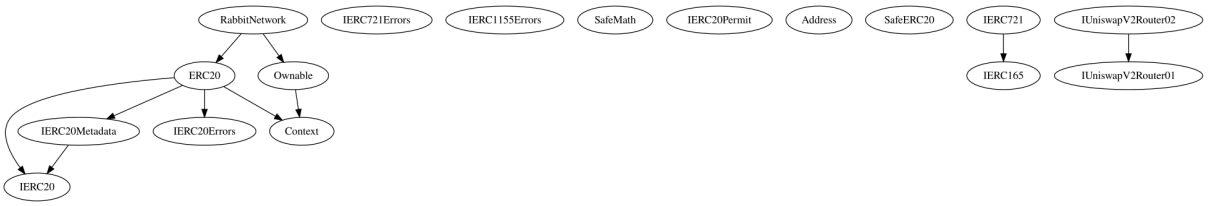
Recommendation

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

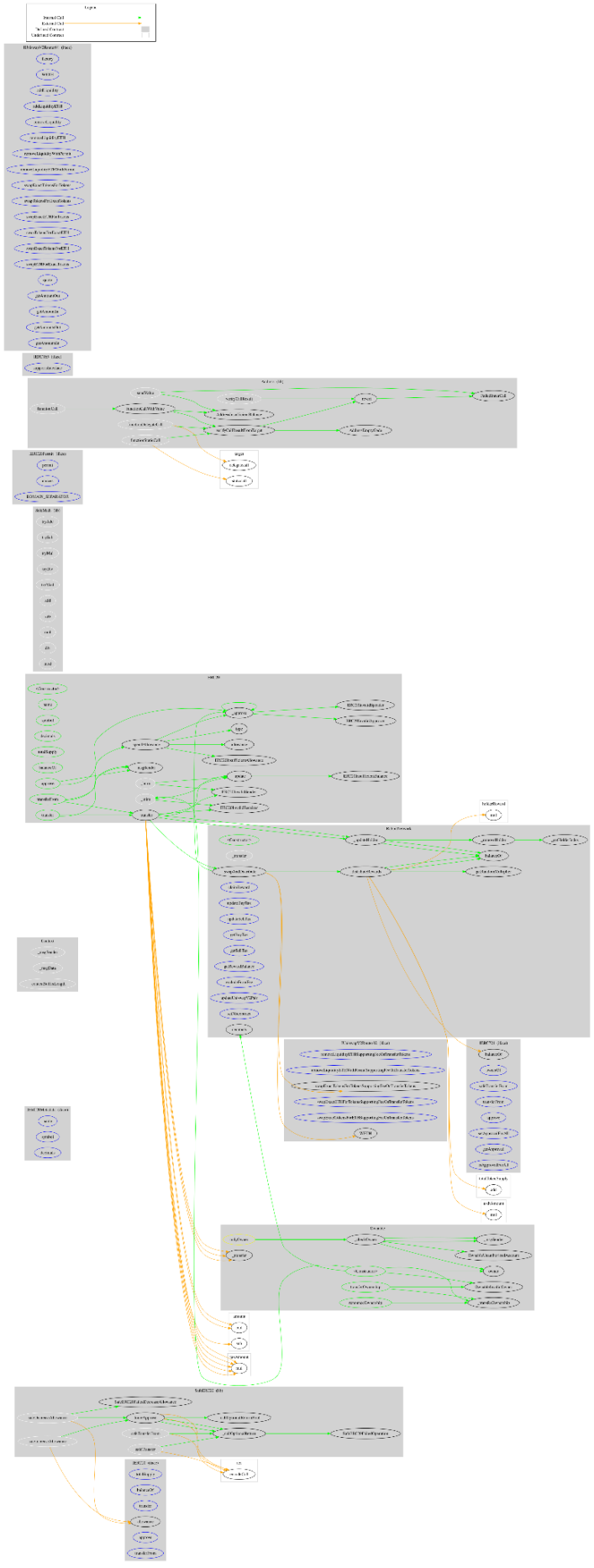
Functions Analysis

Contract	Type	Bases		
	Function Name	**Visibility**	**Mutability**	**Modifiers**
RabbitNetwork	Implementation	ERC20, Ownable		
		Public	✓	ERC20 Ownable
	_transfer	Internal	✓	
	_swapAndDistribute	Private	✓	
	_distributeRewards	Private	✓	
	getRandomMultiplier	Private		
	_updateHolder	Private	✓	
	_removeHolder	Private	✓	
	_getHolderIndex	Private		
	claimReward	External	✓	-
	updateBuyTax	External	✓	onlyOwner
	updateSellTax	External	✓	onlyOwner
	getBuyTax	External		-
	getSellTax	External		-
	getRewardBalance	External		-
	excludeFromFee	External	✓	onlyOwner
	updateUniswapV2Pair	External	✓	onlyOwner

Inheritance Graph



Flow Graph



Summary

Rabbits Network 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 manipulate the fees. 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

cyberscope.io