#HashEx
Blockchain Security

Omicron Rocket

smart contracts
final audit report

February 2022

hashex.org

contact@hashex.org

Contents

1. Disclaimer

This is a limited report on our findings based on our analysis, in accordance with good industry practice at the date of this report, in relation to cybersecurity vulnerabilities and issues in the framework and algorithms based on smart contracts, the details of which are set out in this report. In order to get a full view of our analysis, it is crucial for you to read the full report. While we have done our best in conducting our analysis and producing this report, it is important to note that you should not rely on this report and cannot claim against us on the basis of what it says or doesn't say, or how we produced it, and it is important for you to conduct your own independent investigations before making any decisions. We go into more detail on this in the disclaimer below – please make sure to read it in full.

By reading this report or any part of it, you agree to the terms of this disclaimer. If you do not agree to the terms, then please immediately cease reading this report, and delete and destroy any and all copies of this report downloaded and/or printed by you. This report is provided for information purposes only and on a non-reliance basis and does not constitute investment advice. No one shall have any right to rely on the report or its contents, and HashEx and its affiliates (including holding companies, shareholders, subsidiaries, employees, directors, officers, and other representatives) (HashEx) owe no duty of care towards you or any other person, nor does HashEx make any warranty or representation to any person on the accuracy or completeness of the report. The report is provided "as is", without any conditions, warranties, or other terms of any kind except as set out in this disclaimer, and HashEx hereby excludes all representations, warranties, conditions, and other terms (including, without limitation, the warranties implied by law of satisfactory quality, fitness for purpose and the use of reasonable care and skill) which, but for this clause, might have effect in relation to the report. Except and only to the extent that it is prohibited by law, HashEx hereby excludes all liability and responsibility, and neither you nor any other person shall have any claim against HashEx, for any amount or kind of loss or damage that may result to you or any other person (including without limitation, any direct, indirect, special, punitive, consequential or pure economic loss or damages, or any loss of income, profits, goodwill, data, contracts, use of money, or business interruption, and whether in delict, tort (including without limitation negligence), contract, breach of statutory duty, misrepresentation (whether innocent or negligent) or otherwise under any claim of any nature whatsoever in any jurisdiction) in any way arising from or connected with this report and the use, inability to use or the results of the use of this report, and any reliance on this report. The analysis of the security is purely based on the smart contracts alone. No applications or operations were reviewed for security. No product code has been reviewed. HashEx owns all copyright rights to the text, images, photographs, and other content provided in the following document. When using or sharing partly or in full, third parties must provide a direct link to the original document mentioning the author (hashex.org).

2. Overview

HashEx was commissioned by the Omicron token team to perform an audit of their smart contract. The audit was conducted between 24/02/2022 and 28/02/2022.

The purpose of this audit was to achieve the following:

- Identify potential security issues with smart contracts
- Formally check the logic behind given smart contracts.

Information in this report should be used for understanding the risk exposure of smart contracts, and as a guide to improving the security posture of smart contracts by remediating the issues that were identified.

The code is available at omicronrocket/omicron_token Github repository and was audited after commit 6a20ca5. The updated code was rechecked in the dc8d9da commit. The same code except changed LOOTBOX address was deployed to the BSC network at 0x90e4aaB26024e4C3405787bd7B01edafDDaBA70D address.

2.1 Summary

Project name	Omicron Rocket
URL	https://omicron-rocket.io/
Platform	Binance Smart Chain
Language	Solidity

2.2 Contracts

Name	Address
Ownable	
OmicronRocket	0x90e4aaB26024e4C3405787bd7B01edafDDaBA70D

3. Found issues

Critical	1 (6%)
High	4 (24%)
Medium	4 (24%)
Low	4 (24%)
Info	4 (22%)

C1. Ownable

ID	Severity	Title	Status
C1-01	Critical	Temprorary ownership renounce	Resolved

C2. OmicronRocket

ID	Severity	Title	Status
C2-01	High	The excluded[] array length problem	Resolved
C2-02	High	No safeguards for fees and maxTxAmount	Partially fixed
C2-03	High	excludeFromReward() abuse	Resolved
C2-04	High	Tokens could be transferred back from the burn address	Resolved
C2-05	Medium	addLiquidity() recipient	Resolved
C2-06	Medium	ERC20 standard violation	Acknowledged

C2-07	Medium	Locked ether	Resolved
C2-08	Medium	Hardcoded liquidity swap threshold is bigger than total supply	Resolved
C2-09	Low	Incorrect error message	Resolved
C2-10	Low	Hardcoded team wallet addresses	Partially fixed
C2-11	Low	Gas savings	Resolved
C2-12	Low	Lack of events on important value changes	Partially fixed
C2-13	Info	Typos	Resolved
C2-14	Info	Price calculation via pair's reserves	Acknowledged
C2-15	Info	Excluding from fee does not exlude from the fee tax	Resolved
C2-16	Info	Swaps are made with 100% slippage and infinite deadline	Acknowledged

4. Contracts

C1. Ownable

Overview

A modified version of the OpenZeppelin's Ownable contract. Compared to the original contract the current version has the functionality to temporarily renounce ownership by calling the `lock()` function.

Issues

C1-01 Temprorary ownership renounce

Critical Resolved

The Ownable contract which is inherited by the OmicronRocket token contract is a modified version of OpenZeppelin's `Ownable.sol`. It has the additional functionality of renouncing the ownership for a specified amount of time and then getting the ownership back to the previous owner. Moreover, if the lock function was once called, already renounced ownership can be returned to the owner by calling the unlock function. This can mislead users who check that owner of the contract is a zero address, they might believe that the ownership is actually renounced. We identify this behavior as fraudulent and strongly recommend locking for the maximum possible amount of time immediately after.

Recommendation

Use the original OpenZeppelin contract which is widely used and well-tested.

C2. OmicronRocket

Overview

OmicronRocket token is a SafeMoon fork with an added functionality of presale and vesting.

Issues

C2-01 The excluded[] array length problem

High Resolved

The mechanism of removing addresses from auto-yielding implies a loop over excluded addresses for every transfer operation or balance inquiry. This may lead to extreme gas costs up to the block gas limit and may be avoided only by the owner restricting the number of excluded addresses. In an extreme situation with a large number of excluded addresses transaction gas may exceed the maximum block gas size and all transfers will be effectively blocked. If the owner's account gets compromised the attacker can make the token completely unusable for all users. Moreover, the function`includeInReward()` relies on the same `for()` loop which may lead to irreversible contract malfunction.

Recommendation

The code may be refactored with no usage of the for() loops by using aggregated values. An example of implementation can be seen here.

C2-02 No safeguards for fees and maxTxAmount

High Partially fixed

The Omicron token contract contains external `onlyOwner` functions that set `_taxFee`, `_liquidityFee`, and `_maxTxAmount` to any value of uint256. This behavior is dangerous as at the time of the audit the contract owner is an EOA (externally owned account) and if it is compromised or the owner acts maliciously it can lead to devastating consequences for the token making it completely unusable.

```
    function setTaxFeePercent(uint256 taxFee) external onlyOwner() {
```

```
        _taxFee = taxFee;
    }

    function setMaxTxPercent(uint256 maxTxPercent) external onlyOwner() {
        _maxTxAmount = _tTotal.mul(maxTxPercent).div(
            10**2
        );
    }

    function setLiquidityFeePercent(uint256 liquidityFee) external onlyOwner() {
        _liquidityFee = liquidityFee;
    }
```

Recommendation

We recommend setting reasonable caps for the specified parameters.

Update

An upper limit of 5% for `taxFee`, `liquidityFee` and `maxTxPercent` parameters were added. The `maxTxPercent` should have a lower limit, not an upper, otherwise, the owner has the possibility to block users' transfers.

C2-03 excludeFromReward() abuse

High Resolved

The owner of the token contract can redistribute part of the tokens from users to a specific account. For this owner can exclude an account from the reward and include it back later. This will redistribute part of the tokens from holders in profit of the included account.

Recommendation

The `_rOwned()` must be updated with the current rate in the `includeInReward()` function.

C2-04 Tokens could be transferred back from the burn address

High Resolved

The burnt tokens (tokens that are transferred to the burn address) are not actually burnt and are used as tokens to be paid for vesting, airdrop, and presale. This may confuse users as

tokens on zero and dead addresses are commonly supposed to be locked forever on these addresses.

Recommendation

Refactor the code elimination token transfers from the burn address as it may confuse uses.

Update

After the update, 40% of minted tokens are transferred to BURN_ADDRESS address, which is actually an EOA address.

C2-05 addLiquidity() recipient

Medium Resolved

The `addLiquidity()` function in OmicronRocket L1098 calls for the `uniswapV2Router.addLiquidityETH()` function with the parameter of lp tokens recipient set to owner address. With time, the owner address may accumulate a significant amount of LP tokens which may be dangerous for token economics if an owner acts maliciously or their account gets compromised. This issue can be fixed by changing the recipient address to the OmicronRocket contract or by renouncing ownership which will effectively lock the generated LP tokens.

Update

The liquidity recipient was set to the token itself, which means that the liquidity will be locked forever.

C2-06 ERC20 standard violation

Medium Acknowledged

Implementation of the `transfer()` function does not allow to input zero amount as it's demanded in ERC20 and BEP20 standards. This issue may break the interaction with smart contracts that rely on full ERC20 support. Also, transfer functions of the reviewed contract don't throw error messages for the amounts bigger than the sender's balance (like `ERC20: transfer amount exceeds allowance` in OpenZeppelin's ERC20 implementation) which may confuse users.

C2-07 Locked ether

Medium Resolved

The payable `receive()` function makes it possible for the contract to receive ether or bnb. Moreover, `addLiquidityETH()` from UniswapV2Router returns any ETH/BNB leftovers back to the sender. There's no implemented mechanism for handling this contract's ETH/BNB balance.

```
    receive() external payable {}
```

C2-08 Hardcoded liquidity swap threshold is bigger than total supply

Medium Resolved

The variable `numTokensSellToAddToLiquidity` cannot be changed and exceeds the total supply.

```
    uint256 private _tTotal = 100000000000  * 10**18;
    uint256 private numTokensSellToAddToLiquidity = 500000 * 10**6 * 10**18;
```

This means that the auto-add liquidity functionality of the token won't work.

```
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        ...
        uint256 contractTokenBalance = balanceOf(address(this));

        if(contractTokenBalance >= _maxTxAmount)
        {
            contractTokenBalance = _maxTxAmount;
        }

        bool overMinTokenBalance = contractTokenBalance >= numTokensSellToAddToLiquidity;
        if (
            overMinTokenBalance &&
            !inSwapAndLiquify &&
```

```
            from != uniswapV2Pair &&
            swapAndLiquifyEnabled
        ) {
            contractTokenBalance = numTokensSellToAddToLiquidity;
            //add liquidity
            swapAndLiquify(contractTokenBalance);
        }
    ...
}
```

It should also be noted that the initial value of the maximum transfer amount `_maxTxAmount` also amounts to a value higher than the token total supply.

C2-09 Incorrect error message

Low Resolved

Incorrect error message in the function `includeInReward()`: it must be “Account is already included”.

```
function includeInReward(address account) external onlyOwner() {
        require(_isExcluded[account], "Account is already excluded");
        ...
}
```

C2-10 Hardcoded team wallet addresses

Low Partially fixed

The addresses in L719-724 are hardcoded and cannot be changed. This imposes additional risk if any of them is compromised. The owner would not be able to change the address to the one that's not compromised.

```
    // Team wallets
    address public EXCHANGE_WALLET = 0xfbcB5A5a96155d1A9229cac651884AB2730012f2;
    address public MARKETING_WALLET = 0x78BfcA0C53Ac8e6eFDE1979079B516AC3B8B8CFb;
    address public TEAM_WALLET = 0xB7EF0e728a9b4714153c049c58d19f8a4fd0db9c;
    address public BURN_WALLET = 0xf081a6bcC50079E122a387043BBda1C50F8A9810;
    address public BURN_ADDR = 0x000000000000000000000000000000000000dEaD;
    address private FUND_WALLET = 0x01096559F1595Fad92646A2fED58048f9F611699;
```

Recommendation

Make setters for the team wallet variables callable only by the owner.

Update

Setters for the `MARKETING_WALLET`, `TEAM_WALLET`, `FUND_WALLET` were added. It should be noted that the variables that can be changed should be written in camelCase form.

C2-11 Gas savings

Low Resolved

- The following functions can be declared external instead of public. This will save gas on calling them:

`isExcludedFromFee(), setSwapAndLiquifyEnabled(), includeInFee(), excludeFromFee(), reflectionFromToken(), deliver(), totalFees(), isExcludedFromReward(), decreaseAllowance(), increaseAllowance(), transferFrom(), approve(), allowance(), transfer(), totalSupply(), decimal(), symbol(), name()`

- The following variables can be declared constant:

`BURN_ADDR, BURN_WALLET, EXCHANGE_WALLET, FUND_WALLET, MARKETING_WALLET, PRESALE_CLIAM_START, PRESALE_END, PRESALE_LIMIT, PRESALE_START, TEAM_WALLET, _decimals, _name, _symbol, _tTotal, numTokensSellToAddToLiquidity`

- Unreachable code in the `_tokenTransfer()` function. The last `else` statement is never reached and can be deleted.

```
    function _tokenTransfer(address sender, address recipient, uint256 amount,bool
takeFee) private {
         ...
        if (_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferFromExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && _isExcluded[recipient]) {
            _transferToExcluded(sender, recipient, amount);
        } else if (!_isExcluded[sender] && !_isExcluded[recipient]) {
            _transferStandard(sender, recipient, amount);
        } else if (_isExcluded[sender] && _isExcluded[recipient]) {
            _transferBothExcluded(sender, recipient, amount);
```

```
        } else {
            _transferStandard(sender, recipient, amount);
        }
        ...
    }
```

C2-12 Lack of events on important value changes

Low Partially fixed

The functions that change important values in the contract should emit appropriate events. For example, the function `excludeFromReward()` should emit an `Excluded` event to notify users.

Recommendation

We recommend emitting events in all functions that make important state changes.

Update

Events were added on the `includeInReward()`, `excludeFromReward()` functions. Funtions excludeFromFee(), includeInFee(), setLiquidityFeePercent(), setMaxTxPercent(), setSwapAndLiquifyEnabled() still lack events.

C2-13 Typos

Info Resolved

There are several typos in the naming of variables: `PRESALE_CLIAM_START`, `intial`, `swaping` . Also, the `_uniswapV2Router` address should be written as a checksummed address.

C2-14 Price calculation via pair's reserves

Info Acknowledged

The function `participatePresale()` calculates price via pair's reserved. This price can be manipulated through flashloan attacks.

```
    function participatePresale() external payable{
        ...
        // get swap rate bnb to busd
        uint112 r1;
```

```
        uint112 r2;
        (r1, r2, ) = BNB2USDT.getReserves();
        uint256 rate = r2 / r1;

        uint256 tokenAmount = msg.value.mul(rate).mul(1000);
        ...
    }
```

It must be noted that the possibility of such an event is not big, because for BNB/USDT price manipulation an attacker must use a significant loan and pay commission for it. But the contract developers should be aware of the issue in case the pair is changed in future code updates.

C2-15 Excluding from fee does not exlude from the fee tax

● Info ✓ Resolved

The address can be excluded from fees but the sale tax is still applied.

```
    function _transfer(
        address from,
        address to,
        uint256 amount
    ) private {
        ...
        // apply 6% extra fee for sell
        if (to == address(uniswapV2Pair)) {
            uint256 saleTax = amount.mul(6).div(100);
            _tokenTransfer(from, BURN_ADDR, saleTax, false);
            amount = amount.sub(saleTax);
        }
        ...
}
```

C2-16 Swaps are made with 100% slippage and infinite deadline

Info Acknowledged

Swapping and adding to liquidity in the token contract are made with 100% slippage and an infinite deadline.

```
function swapTokensForEth(uint256 tokenAmount) private {
    ...

    // make the swap
    uniswapV2Router.swapExactTokensForETHSupportingFeeOnTransferTokens(
        tokenAmount,
        0, // accept any amount of ETH
        path,
        address(this),
        block.timestamp
    );
}
```

```
function addLiquidity(uint256 tokenAmount, uint256 ethAmount) private {
    ...
    // add the liquidity
    uniswapV2Router.addLiquidityETH{value: ethAmount}(
        address(this),
        tokenAmount,
        0, // slippage is unavoidable
        0, // slippage is unavoidable
        owner(),
        block.timestamp
    );
}
```

This opens a possibility for sandwich attacks.

5. Conclusion

1 critical, 4 high and 4 medium severity issues were found, most of them were resolved by the code update, including the critical one. The audited contracts are highly dependent on the owner's account. Users using the project have to trust the owner and that the owner's account is properly secured.

On token creation more than 60% of tokens are minted to EOA addresses, which may be considered as additional risks to the investors.

We recommend ownership with a proxy contract (e.g. Timelock) with multisig admin to secure the token. We also recommend adding unit tests with coverage of at least 90% to any introduced functionality.

This audit includes recommendations on improving the code and preventing potential attacks.

Appendix A. Issues severity classification

- **Critical.** Issues that may cause an unlimited loss of funds or entirely break the contract workflow. Malicious code (including malicious modification of libraries) is also treated as a critical severity issue. These issues must be fixed before deployments or fixed in already running projects as soon as possible.
- **High.** Issues that may lead to a limited loss of funds, break interaction with users, or other contracts under specific conditions. Also, issues in a smart contract, that allow a privileged account the ability to steal or block other users' funds.
- **Medium.** Issues that do not lead to a loss of funds directly, but break the contract logic. May lead to failures in contracts operation.
- **Low.** Issues that are of a non-optimal code character, for instance, gas optimization tips, unused variables, errors in messages.
- **Info.** Issues that do not impact the contract operation. Usually, info severity issues are related to code best practices, e.g. style guide.

Appendix B. List of examined issue types

- Business logic overview
- Functionality checks
- Following best practices
- Access control and authorization
- Reentrancy attacks
- Front-run attacks
- DoS with (unexpected) revert
- DoS with block gas limit
- Transaction-ordering dependence
- ERC/BEP and other standards violation
- Unchecked math
- Implicit visibility levels
- Excessive gas usage
- Timestamp dependence
- Forcibly sending ether to a contract
- Weak sources of randomness
- Shadowing state variables
- Usage of deprecated code

 contact@hashex.org

 @hashexbot

 blog.hashex.org

 linkedin

 github

 twitter

HashEx
Blockchain Security