#HashEx
BLOCKCHAIN SECURITY

CleverMinu

smart contracts
final audit report

October 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 CleverMinu team to perform an audit of their smart contract. The audit was conducted between 29/09/2022 and 03/10/2022.

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 was provided in .sol file with MD5 hash sum of c86fe39c705f6dfee9457c2e3349c1e9.

Update. The CleverMinu team has responded to this report. The updated code is located in the GitHub repository after the commit 4bf5066. The same code was deployed to Ethereum, Binance Smart Chain, and Polygon networks at address 0x155AB9Cd3655Aa6174E1e743a6DA1E208762b03d.

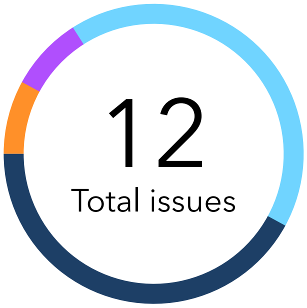
2.1 Summary

Project name	CleverMinu
URL	https://www.cleverminu.com/
Platform	Polygon Network, Binance Smart Chain, Ethereum
Language	Solidity

2.2 Contracts

Name	Address
SafeMath	
ERC20Interface, ApproveAndCallFallBack, IHoldingContract	
Owned	
HoldingContract	`0xc8C97055eB90E3A762ec67B5362B3E94a4D4f487`
CleverMinu	`0x155AB9Cd3655Aa6174E1e743a6DA1E208762b03d`

3. Found issues



● High	1 (8%)
● Medium	1 (8%)
● Low	5 (42%)
● Info	5 (42%)

C1. SafeMath

ID	Severity	Title	Status
C1-01	● Low	Gas optimizations	✓ Acknowledged
C1-02	● Info	Lack of error messages	✓ Resolved

C3. Owned

ID	Severity	Title	Status
C3-01	● Info	Lack of error messages	✓ Resolved

C4. HoldingContract

ID	Severity	Title	Status
C4-01	● Info	Tokens are transferred without confirmation	✓ Resolved

C5. CleverMinu

ID	Severity	Title	Status
C5-01	High	Exaggerated owner's rights	Resolved
C5-02	Medium	Holding bonus is not guaranteed	Acknowledged
C5-03	Low	Possible overflow	Resolved
C5-04	Low	Division before multiplication	Resolved
C5-05	Low	Approve before IMO ends	Acknowledged
C5-06	Low	Gas optimizations	Partially fixed
C5-07	Info	No visibility is set	Resolved
C5-08	Info	Lack of events	Resolved

4. Contracts

C1. SafeMath

Overview

Safe arithmetical operations library.

Issues

C1-01 Gas optimizations ● Low ✓ Acknowledged

1. In versions of Solidity >=0.8, over- and underflow checks are built-in when using math operations. To save gas remove, one can remove the SafeMath contract or rewrite the implementation of functions more simply, for example:

```
function safeAdd(uint256 a, uint256 b) internal pure returns (uint256) {
    return a + b;
}
```

C1-02 Lack of error messages ● Info ✓ Resolved

`require()` conditions return no error messages, which may confuse users and complicate the debugging.

C2. ERC20Interface, ApproveAndCallFallBack, IHoldingContract

Overview

Interfaces for standard compliance. No issues were found.

C3. Owned

Overview

A version of the Ownable authorization model without a simple possibility of renouncing.

Issues

C3-01 Lack of error messages ● Info ✓ Resolved

`require()` conditions return no error messages, which may confuse users and complicate the debugging.

C4. HoldingContract

Overview

A simple contract to be deployed inside CleverMinu's constructor section. The only function of HoldingContract is to transfer part of its balance upon the request from the CleverMinu token contract.

Issues

C4-01 Tokens are transferred without confirmation ● Info ✓ Resolved

The `initiate()` function doesn't return any values, i.e., it doesn't signal if tokens have been transferred successfully or not.

```
    function initiate(address receiver,uint256 tokens) public {
        require(msg.sender == MAINCONTRACT, "Forbidden");
        uint balance = ERC20Interface(MAINCONTRACT).balanceOf(this);
        if(balance<tokens) return;
            ERC20Interface(MAINCONTRACT).transferinternal(this,receiver, tokens);
    }
```

C5. CleverMinu

Overview

An implementation of the ERC-20 token standard. Transfers are susceptible to a fixed fee of USER_BURNRATIO value (10% default), which is moved to the HoldingContract and paid later to the token holders after the 1-hour delay (default bonus value is 0.01% per day).

Issues

C5-01 Exaggerated owner's rights ● High ✓ Resolved

The contract owner is capable of halting the transfers by setting `USER_BURNRATIO` to over 100% and causing the underflow error or updating `IMOENDTIME` any time after the sale.

```
    function setIMOendTime( uint256 time) public onlyOwner {
        IMOENDTIME=time;
    }

    function setUSERBurnRatio( uint256 _ratio) public onlyOwner {
        USER_BURNRATIO=_ratio;
    }
```

Another problem is open minting for the owner: setting `IMOENDTIME = 0` allows the owner to re-init the contract, effectively minting another `_totalSupply` (1e21 tokens) to the CleverMinu contract's balance. After that, the owner can transfer these minted tokens to any address by calling `IMOreferral()`.

```
    function init(uint256 _imoenddate) public onlyOwner
    {
        require(IMOENDTIME==0,"Already Initiated");
        //IMOENDTIME = block.timestamp;
        IMOENDTIME=_imoenddate;
        balances[this] = _totalSupply;
        emit Transfer(address(0), this, _totalSupply);
        ContractAddress=this;
        addtoWhiteList(this);
    }
```

Recommendation

Add safety checks into `setIMOendTime()` and `setUSERBurnRatio()`. Consider adding input data validation to other set functions, i.e., `setHoldBonusRatio()` and `setIMOBurnRatio()`.

It is also acceptable to renounce ownership completely, although, it's not simple in case of used Owned contract.

C5-02 Holding bonus is not guaranteed ● Medium ✓ Acknowledged

The bonus for holders is paid in the `creditholdingbonus()` function, which is called in every transfer. This function checks the time period after the last user's balance update. It calls the HoldingContract to transfer the bonus amount from its balance to the user, but if HoldingContract doesn't hold enough tokens, nothing is transferred, and the bonus is cleared, meaning a loss for the user.

Moreover, anyone can trigger this kind of losing the bonus tokens by sending any amount to the whale holders, whose bonus exceeds the HoldingContract's balance at the moment.

Recommendation

Consider not updating `lastbalances[_address]` unless bonus tokens are successfully delivered. It's also possible to store unpaid bonuses into mapping `address => uint256`.

C5-03 Possible overflow ● Low ✓ Resolved

SafeMath is not used in the `getburntokencount()` function, since there are no limits in the `setUSERBurnRatio()`, see the 'Exaggerated owner's rights' issue.

```
    function getburntokencount(uint amount) public constant returns (uint balance) {
        return ((amount*IMO_BURNRATIO)/100);
    }
```

C5-04 Division before multiplication ● Low ✓ Resolved

Division before multiplication can, in some rare cases, cause calculation errors in integer math. In this contract, division before multiplication is performed in the `creditholdingbonus()` function on L235.

C5-05 Approve before IMO ends ● Low ✓ Acknowledged

Using the `increaseAllowance()` and `decreaseAllowance()` functions, you can change the allowance before the end of the sale. This is most likely a logical error, since the `approve()` function, on the contrary, checks the time.

C5-06 Gas optimizations ● Low ✓ Partially fixed

1. Multiple reads from storage are performed in `init()`: `_totalSupply`; `transfer()`: `IMOENDTIME`, `USER_BURNRATIO`, `Holding_CONTRACT`; `creditholdingbonus()`: `Holding_CONTRACT`; `transferFrom()`: `IMOENDTIME`, `USER_BURNRATIO`, `Holding_CONTRACT`; `transferAnyERC20Token()`: `owner`;
2. Reverted fallback is not needed as of Solidity 0.4.0.

3. External calls to itself in `IMOsale()` and `IMOreferral()` could be avoided by splitting transfer functions into internal and external separate functions.
4. Upgrading to recent pragma versions might save gas both for deployment and interaction with the contract. For example, in 0.6.5 the `immutable` keyword was introduced, post-0.8.0 versions include internal safe math checks, post-0.8.4 have custom errors to replace `require()` revert reasons.

C5-07 No visibility is set ● Info ✓ Resolved

Mappings `balances[]` and `allowed[]` have no default visibility set; `internal` is used by default.

C5-08 Lack of events ● Info ✓ Resolved

There are no custom events besides standard ERC-20 ones. We recommend adding such events, especially for governance functions, e.g., updating whitelist and ratios.

5. Conclusion

1 high, 1 medium, 5 low severity issues were found during the audit. 1 high, 2 low issues were resolved in the update.

We recommend adding documentation as well as unit and functional tests for all contracts. We also recommend upgrading the Solidity compiler version and importing the OpenZeppelin library.

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.
- **Informational.** Issues that do not impact the contract operation. Usually, informational 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

@hashex_manager

blog.hashex.org

linkedin

github

twitter

HashEx
BLOCKCHAIN SECURITY