QuillAudits

Audit Report
October, 2022

For

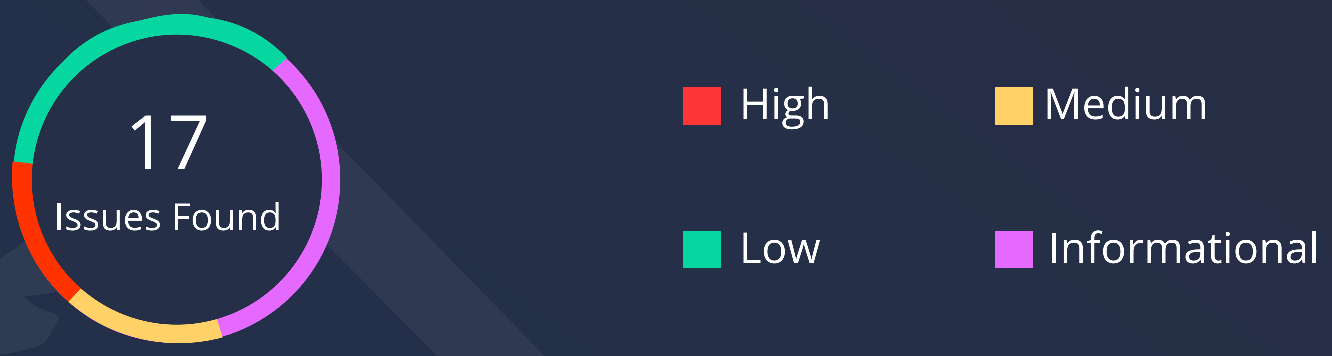
Cleverminu

Table of Content

Table of Content

Executive Summary

Project Name	CleverMinu
Timeline	30 september,2022 to 5 october,2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse CleverMinu codebase for quality, security, and correctness. *https://github.com/cleverminu/Contract/blob/main/TokenContract.sol* Commit hash: 4a7cac0615c0f265544c2aa974900041d59d61e6
Branch	Main
Fixed In	4bf506698fa0f1e49f74d07588d44a1b90327d6d **Mainnet address:** *https://polygonscan.com/* *address/0x155AB9Cd3655Aa6174E1e743a6DA1E208762b03d*



	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	3	3	0	1
Partially Resolved Issues	0	0	0	0
Resolved Issues	1	0	4	5

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- ✓ Re-entrancy
- ✓ Timestamp Dependence
- ✓ Gas Limit and Loops
- ✓ Exception Disorder
- ✓ Gasless Send
- ✓ Use of tx.origin
- ✓ Compiler version not fixed
- ✓ Address hardcoded
- ✓ Divide before multiply
- ✓ Integer overflow/underflow
- ✓ Dangerous strict equalities
- ✓ Tautology or contradiction
- ✓ Return values of low-level calls
- ✓ Missing Zero Address Validation
- ✓ Private modifier
- ✓ Revert/require functions
- ✓ Using block.timestamp
- ✓ Multiple Sends
- ✓ Using SHA3
- ✓ Using suicide
- ✓ Using throw
- ✓ Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

Manual Testing

A. Contract - CleverMinu

High Severity Issues

A1. All Funds can be drained before sale via IMOsale()

Description

IMOsale has no access control. This could result in the whitelisted addresses in draining the entire contract during the sale.

Remediation

It is advised to add sufficient access controls and require checks to mitigate this issue.

Status

Acknowledged

A2. All Funds can be drained before sale via IMOreferral()

Description

IMOreferral has no access control and can be used to drain the complete contract by whitelisted users.

Remediation

It is advised to add sufficient access controls and require checks to mitigate this issue

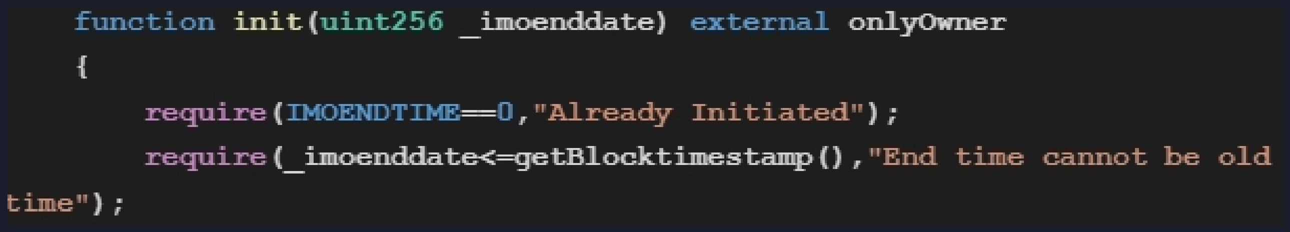
Status

Acknowledged

A3. Incorrect require statement in init()

Description

Incorrect require statement due to which init reverts on setting IMO end date in any time in future.



```
    function init(uint256 _imoenddate) external onlyOwner
    {
        require(IMOENDTIME==0,"Already Initiated");
        require(_imoenddate<=getBlocktimestamp(),"End time cannot be old
time");
```

Instead his should be require(_imoenddate > getBlocktimestamp(),"End time cannot be old time");

Remediation

It is advised to make the changes as suggested above

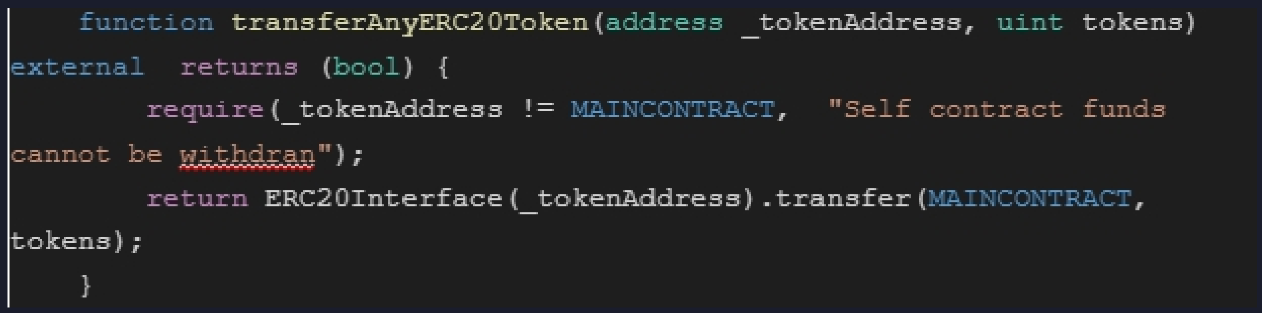
Status

Resolved

A4. Incorrect require statement in transferAnyERC20Token() of HoldingContract

Description

transferAnyERC20Token can be used to drain all the CLEVERMINU tokens from the holding contract by a malicious admin at any point of time. The require statement is not required as it checks for _tokenAddress to be of a MAINCONTRACT which is not required.



```
    function transferAnyERC20Token(address _tokenAddress, uint tokens)
external  returns (bool) {
        require(_tokenAddress != MAINCONTRACT,  "Self contract funds
cannot be withdran");
        return ERC20Interface(_tokenAddress).transfer(MAINCONTRACT,
tokens);
    }
```

Remediation

It is advised to make the changes as suggested above

Status

Acknowledged

Medium Severity Issues

A5. Approve race condition

Description

There exists a race condition for approve which allows the approved address to spend more tokens than expected. For example if Alice has approved Eve to spend n of her tokens, then Alice decides to change Eve's approval to m tokens. Alice submits a function call to approve with the value n for Eve. Eve runs an Ethereum node so knows that Alice is going to change her approval to m. Eve then submits a tranferFrom request sending n of Alice's tokens to herself, but gives it a much higher gas price than Alice's transaction. The transferFrom executes first so gives Eve n tokens and sets Eve's approval to zero. Then Alice's transaction executes and sets Eve's approval to m. Eve then sends those m tokens to herself as well. Thus Eve gets n + m tokens even though she should have gotten at most max(n,m).

Remediation

It is advised to use safeIncreaseAllowance and safeDecreaseAllowance such as that from Open Zeppelin instead

Status

Resolved

A6. Centralization of setUSERBurnRatio (Medium)

Description

setUSERBurnRatio() can be used to change USER_BURNRATIO anytime and increased by a malicious owner at the detriment of users. For example, a malicious owner can change the User burn ratio from 20 to say 80 without user’s notice and due to which all the users will need to comply with. There should be a mechanism for user’s to reject this change or opt out if they want to.

Remediation

It is advised to use multisig wallet for increasing decentralization and safety of private keys. A DAO is recommended to be used so that the burn ratio for users is not unfairly set without user’s notice

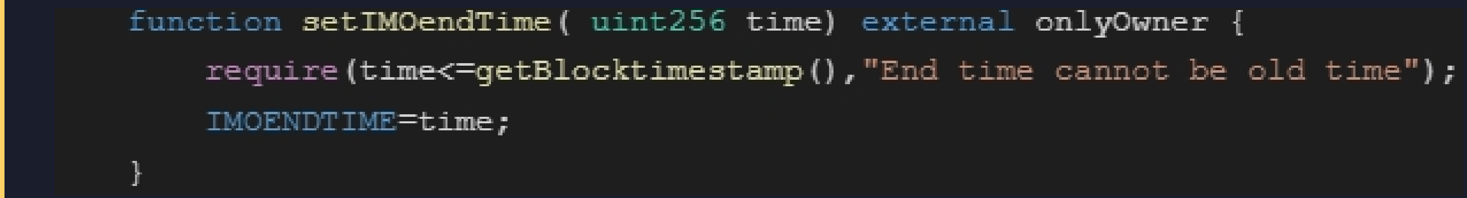
Status

Resolved

A7. Incorrect require statement in setIMOendTime()

Description

On line: 354 the require statement is potentially incorrect



```
function setIMOendTime( uint256 time) external onlyOwner {
    require(time<=getBlocktimestamp(),"End time cannot be old time");
    IMOENDTIME=time;
}
```

There is mismatch between error message and require statement.

require(time > getBlocktimestamp(),"End time cannot be old time");
Should be used instead

Remediation

It is advised to use safeIncreaseAllowance and safeDecreaseAllowance such as that from Open Zeppelin instead.

Refer: https://swcregistry.io/docs/SWC-114

Status

Resolved

Low Severity Issues

A8. Old solidity version

Description

The contract is using solidity version 0.4.24. Using an old version prevents access to new Solidity security checks.

Remediation

Consider using the latest solidity version.

Refer: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Status

Resolved

A9. Lack of sanity checks

Description

No sanity value checks for IMO_BURNRATIO in setIMOBurnRatio function. It is possible to set setIMOBurnRatio greater than 100, which could result in more tokens being sent for burning than intended during the execution of IMOsale function.

Remediation

Consider using the latest solidity version.

Status

Resolved

A10. No sanity value checks for IMOENDTIME

Description

A non-epoch timestamp can also be set for this value accidentally resulting in IMOENDTIME being passed already by the currents timestamp

Status

Resolved

A11. Uncheck Return Value:

Description

HoldingContract.initiate(address,uint256) (contracts/TokenContract.sol#87-92) ignores return value by ERC20Interface(MAINCONTRACT).transferinternal(this,receiver,tokens) (contracts/TokenContract.sol#91)
CleverMinu.IMOreferral(address,uint256) (contracts/TokenContract.sol#156-164) ignores return value by ERC20Interface(this).transferinternal(this,to,amount) (contracts/TokenContract.sol#162)

Remediation

It is advised to add specific require checks on the return values

Refer: https://github.com/crytic/slither/wiki/Detector-Documentation#unused-return

Status

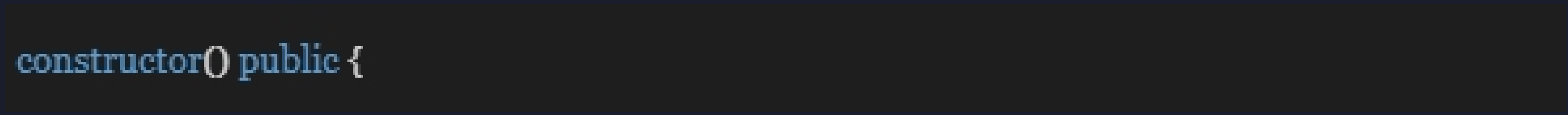
Resolved

Informational Issues

A12. Constructor does not need visibility specifier

Description

Solidity versions >0.5.0 do not require a visibility specifier for constructors.



```
constructor() public {
```

Remediation

It is recommended to follow the article’s recommendation and reduce the step-size in A to not more than 0.1% per block.

Status

Resolved

A13. Inaccurate type specified (Informational)

Description

uint type declaration has been used in the IERC20Interface and SafeMath.

Remediation

It is advised to specifically declare the type of the variable as uint256 instead of uint to adhere to best security practices

Status

Resolved

A14. Lack of error messages

Description

SafeMath and Owned does not have comments for errors of require statements.

Remediation

It is advised to add require statements for the same.

Status

Resolved

A15. Unnecessary usage of function (Informational)

Description

A separate safeDiv256() function is not needed as safeDiv() function works for uint256 values as well

Remediation

It is advised to remove the safeDiv() function from the contract

Status

Resolved

A16. Unnecessary usage of SafeMath

Description

SafeMath not required in Latest solidity compiler versions above 0.8.0 Usage of SafeMath makes code harder to read and increases code size.

Remediation

It is advised to remove the SafeMath as it is not required

Status

Acknowledged

A17. SafeMath functions can be made internal

Description

SafeMath functions can be made internal instead of public

Remediation

It is advised to make SafeMath functions internal instead of public

Status

Resolved

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

In this report, we have considered the security of the CleverMinu. We performed our audit according to the procedure described above.

Some issues of High, Medium, Low and informational severity were found, Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the cleverMinu Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the cleverMinuTeam put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

600+
Audits Completed

$15B
Secured

600K
Lines of Code Audited

Follow Our Journey

Audit Report
October, 2022

For

Cleverminu

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com