BLOCKHAT
SECURITY

WealthClubCoin

Smart Contract Security Audit

Prepared by BlockHat

November 23rd, 2022 - December 3rd, 2022

BlockHat.io

contact@blockhat.io

Document Properties

Client	hecrod
Version	1.0
Classification	Public

Scope

Files	MD5 Hash
WCC.sol	42427c61d2b7e1c5ee3e149ce7cd3217

Link	File
https://bscscan.com/address/0x5daee2a93bfbe08c645e0edc63c1ff1ce565600c	WCC.sol

Contacts

COMPANY	CONTACT
BlockHat	contact@blockhat.io

Contents

1 Introduction

hecrod engaged BlockHat to conduct a security assessment on the WealthClubCoin beginning on November 23rd, 2022 and ending December 3rd, 2022. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About WealthClubCoin

WCC is a cryptocurrency born within the Wealth Club community from the idea of facilitating access to cryptocurrencies by educating the masses about blockchain technology's benefits. It is a cryptocurrency evolving from a 4-year financial development plan, beginning Latin America.

Issuer	hecrod
Website	`www.wealthclub.org`
Type	Solidity Smart Contract
Audit Method	Whitebox

1.2 Approach & Methodology

BlockHat used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

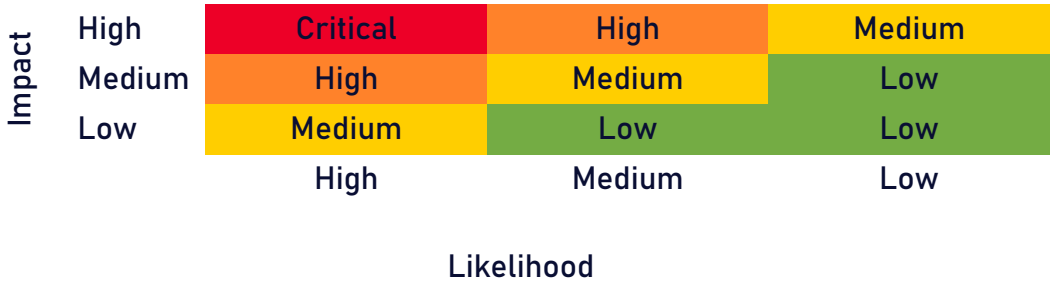
1.2.1 Risk Methodology

Vulnerabilities or bugs identified by BlockHat are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.



2 Findings Overview

2.1 Summary

The following is a synopsis of our conclusions from our analysis of the WealthClubCoin implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.2 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 1 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
Floating Pragma	LOW	Fixed

3 Finding Details

A WCCToken.sol

A.1 Floating Pragma [LOW]

Description:

The contract makes use of the floating-point pragma 0.8.17. Contracts should be deployed using the same compiler version. Locking the pragma helps ensure that contracts will not unintentionally be deployed using another pragma, which in some cases may be an obsolete version, that may introduce issues to the contract system.

Code:

Listing 1: WCCToken.sol

```
15 pragma solidity ^0.8.17;
```

Risk Level:

Likelihood – 1
Impact - 2

Recommendation:

Consider locking the pragma version. It is advised that floating pragma should not be used in production. Both truffle-config.js and hardhat.config.js support locking the pragma version.

Status - Fixed

The dev team fixed the issue

4 Best Practices

BP.1 Presence of unused code

Description:

The program contains code that is not essential for execution, i.e, makes no state changes and has no side effects that alter data or control flow, such that removal of the code would have no impact on functionality or correctness , Context._msgData() is never used and should be removed

Code:

Listing 2: WCCToken.sol

```
136     function _msgData() internal view virtual returns (bytes calldata) {
137         return msg.data;
138     }
```

Status - Fixed

The dev team fixed the issue by removing the unused code

BP.2 Public Function Can Be Called External

Description:

Functions with a public scope that are not called inside the contract should be declared external to reduce the gas fees

Code:

Listing 3: WCCToken.sol

```
194     function name() public view virtual override returns (string memory)
            ↪ {
```

```
195         return _name;
196     }
```

Listing 4: WCCToken.sol

```
202     function symbol() public view virtual override returns (string
            ↪ memory) {
203         return _symbol;
204     }
```

Listing 5: WCCToken.sol

```
226     function totalSupply() public view virtual override returns (uint256
            ↪ ) {
227         return _totalSupply;
228     }
```

Listing 6: WCCToken.sol

```
233     function balanceOf(address account) public view virtual override
            ↪ returns (uint256) {
234         return _balances[account];
235     }
```

Listing 7: WCCToken.sol

```
245     function transfer(address to, uint256 amount) public virtual
            ↪ override returns (bool) {
246         address owner = _msgSender();
247         _transfer(owner, to, amount);
248         return true;
249     }
```

Listing 8: WCCToken.sol

```
268     function approve(address spender, uint256 amount) public virtual
            ↪ override returns (bool) {
269         address owner = _msgSender();
270         _approve(owner, spender, amount);
```

```
271         return true;
272     }
```

Listing 9: WCCToken.sol

```
290     function transferFrom(
291         address from,
292         address to,
293         uint256 amount
294     ) public virtual override returns (bool) {
295         address spender = _msgSender();
296         _spendAllowance(from, spender, amount);
297         _transfer(from, to, amount);
298         return true;
299     }
```

Listing 10: WCCToken.sol

```
313     function increaseAllowance(address spender, uint256 addedValue)
            ↪ public virtual returns (bool) {
314         address owner = _msgSender();
315         _approve(owner, spender, allowance(owner, spender) + addedValue);
316         return true;
317     }
```

Listing 11: WCCToken.sol

```
333     function decreaseAllowance(address spender, uint256 subtractedValue)
            ↪  public virtual returns (bool) {
334         address owner = _msgSender();
335         uint256 currentAllowance = allowance(owner, spender);
336         require(currentAllowance >= subtractedValue, "BEP20: decreased
                ↪ allowance below zero");
337         unchecked {
338             _approve(owner, spender, currentAllowance - subtractedValue);
339         }
340
```

```
341         return true;
342     }
```

Listing 12: WCCToken.sol

```
530     function burn(uint256 amount) public virtual {
531         _burn(_msgSender(), amount);
532     }
```

Listing 13: WCCToken.sol

```
545     function burnFrom(address account, uint256 amount) public virtual {
546         _spendAllowance(account, _msgSender(), amount);
547         _burn(account, amount);
548     }
549 }
```

Listing 14: WCCToken.sol

```
605     function renounceOwnership() public virtual onlyOwner {
606         _transferOwnership(address(0));
607     }
```

Listing 15: WCCToken.sol

```
613     function transferOwnership(address newOwner) public virtual
            ↪ onlyOwner {
614         require(newOwner != address(0), "Ownable: new owner is the zero
                ↪ address");
615         _transferOwnership(newOwner);
616     }
```

Status - Fixed

The dev team fixed the issue

5 Static Analysis (Slither)

Description:

Block Hat expanded the coverage of the specific contract areas using automated testing methodologies. Slither, a Solidity static analysis framework, was one of the tools used. Slither was run on all-scoped contracts in both text and binary formats. This tool can be used to test mathematical relationships between Solidity instances statically and variables that allow for the detection of errors or inconsistent usage of the contracts' APIs throughout the entire codebase.

Results:

```
Context._msgData() (WCC.sol#136-138) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #dead-code

Pragma version^0.8.17 (WCC.sol#15) necessitates a version too recent to
    ↪ be trusted. Consider deploying with 0.6.12/0.7.6/0.8.7
solc-0.8.17 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #incorrect-versions-of-solidity

WCC.constructor() (WCC.sol#631-633) uses literals with too many digits:
        - _mint(msg.sender,2500000000 * 10 ** decimals()) (WCC.sol#632)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #too-many-digits

name() should be declared external:
        - BEP20.name() (WCC.sol#194-196)
symbol() should be declared external:
        - BEP20.symbol() (WCC.sol#202-204)
totalSupply() should be declared external:
        - BEP20.totalSupply() (WCC.sol#226-228)
```

```
balanceOf(address) should be declared external:
        - BEP20.balanceOf(address) (WCC.sol#233-235)
transfer(address,uint256) should be declared external:
        - BEP20.transfer(address,uint256) (WCC.sol#245-249)
approve(address,uint256) should be declared external:
        - BEP20.approve(address,uint256) (WCC.sol#268-272)
transferFrom(address,address,uint256) should be declared external:
        - BEP20.transferFrom(address,address,uint256) (WCC.sol#290-299)
increaseAllowance(address,uint256) should be declared external:
        - BEP20.increaseAllowance(address,uint256) (WCC.sol#313-317)
decreaseAllowance(address,uint256) should be declared external:
        - BEP20.decreaseAllowance(address,uint256) (WCC.sol#333-342)
burn(uint256) should be declared external:
        - BEP20Burnable.burn(uint256) (WCC.sol#530-532)
burnFrom(address,uint256) should be declared external:
        - BEP20Burnable.burnFrom(address,uint256) (WCC.sol#545-548)
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (WCC.sol#605-607)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (WCC.sol#613-616)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #public-function-that-could-be-declared-external
WCC.sol analyzed (7 contracts with 78 detectors), 17 result(s) found
```

Conclusion:

Most of the vulnerabilities found by the analysis have already been addressed by the smart contract code review.

6 Conclusion

In this audit, we examined the design and implementation of WealthClubCoin contract and discovered several issues of varying severity. hecrod team addressed all the issues raised in the initial report and implemented the necessary fixes.

The present code base is well-structured and ready for the mainnet.

BLOCKHAT
SECURITY

For a Smart Contract Audit, contact us at contact@blockhat.io