BLOCKHAT
SECURITY

Deencoin

Smart Contract Security Audit

Prepared by BlockHat

August 1st, 2024 - August 3rd, 2024

BlockHat.io

contact@blockhat.io

Document Properties

Client	Deencoin
Version	0.1
Classification	Private

Scope

The Deencoin Contract in the Deencoin Repository

File	MD5
Deencoin	08b16336cb36531405afe26c4a8511ad

Contacts

COMPANY	CONTACT
BlockHat	contact@blockhat.io

Contents

1 Introduction 4
1.1 About Deencoin . . . 4
1.2 Approach & Methodology . . . 4
1.2.1 Risk Methodology . . . 5

2 Findings Overview 6
2.1 Summary . . . 6
2.2 Key Findings . . . 6

3 Finding Details 7
A Deencoin.sol . . . 7
A.1 Use of Outdated ERC20 Implementation [LOW] . . . 7
A.2 Floating Pragma [LOW] . . . 7

4 Static Analysis (Slither) 9

5 Conclusion 11

1 Introduction

Deencoin engaged BlockHat to conduct a security assessment on the Deencoin beginning on August 1st, 2024 and ending August 3rd, 2024. 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 Deencoin

Issuer	Deencoin
Website	
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.

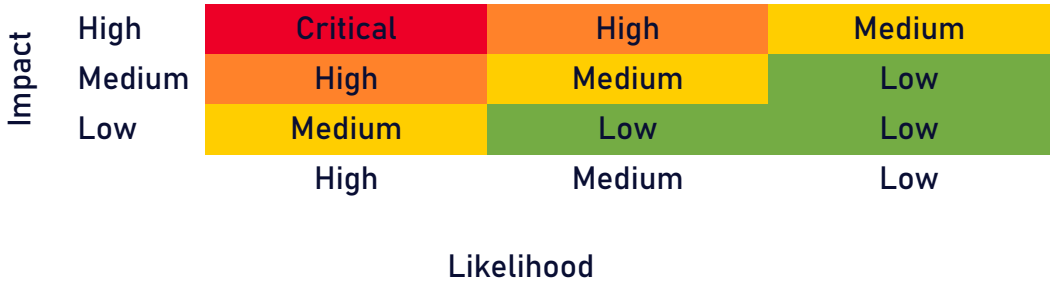
1.2.1 Risk Methodology

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 Deencoin 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 , 2 low-severity vulnerabilities.

Vulnerabilities	Severity	Status
Use of Outdated ERC20 Implementation	LOW	Not Fixed
Floating Pragma	LOW	Not Fixed

3 Finding Details

A Deencoin.sol

A.1 Use of Outdated ERC20 Implementation [LOW]

Description:

The contract implements its own version of the ERC20 standard, which may not include the latest security practices, optimizations, and features found in widely-used libraries such as OpenZeppelin's ERC20 implementation. Using an outdated or custom implementation can introduce risks and compatibility issues with other contracts and decentralized applications (dApps).

Recommendation:

Replace the custom ERC20 implementation with the latest version of the ERC20 token standard from a reputable library such as OpenZeppelin. This not only ensures compliance with the latest security practices but also benefits from the community's scrutiny, ongoing maintenance, and updates.

Status - Not Fixed

A.2 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: Deencoin.sol

```
670 pragma solidity ^0.8.9;;
```

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 - Not Fixed

4 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:

```
Different versions of Solidity are used:
        - Version used: ['^0.8.0', '^0.8.9']
        - ^0.8.0 (Deencoin.sol#8)
        - ^0.8.0 (Deencoin.sol#35)
        - ^0.8.0 (Deencoin.sol#120)
        - ^0.8.0 (Deencoin.sol#205)
        - ^0.8.0 (Deencoin.sol#235)
        - ^0.8.0 (Deencoin.sol#626)
        - ^0.8.9 (Deencoin.sol#665)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #different-pragma-directives-are-used
INFO:Detectors:
Context._msgData() (Deencoin.sol#25-27) is never used and should be
    ↪ removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #dead-code
INFO:Detectors:
Pragma version^0.8.0 (Deencoin.sol#8) allows old versions
Pragma version^0.8.0 (Deencoin.sol#35) allows old versions
Pragma version^0.8.0 (Deencoin.sol#120) allows old versions
Pragma version^0.8.0 (Deencoin.sol#205) allows old versions
```

```
Pragma version^0.8.0 (Deencoin.sol#235) allows old versions
Pragma version^0.8.0 (Deencoin.sol#626) allows old versions
Pragma version^0.8.9 (Deencoin.sol#665) allows old versions
solc-0.8.24 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #incorrect-versions-of-solidity
INFO:Detectors:
Deencoin.constructor() (Deencoin.sol#671-673) uses literals with too
    ↪ many digits:
        - _mint(msg.sender,10000000000 * 10 ** decimals()) (Deencoin.sol
            ↪ #672)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation
    ↪ #too-many-digits
```

Conclusion:

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

5 Conclusion

We examined the design and implementation of Deencoin in this audit and found several issues of various severities. We advise Deencoin team to implement the recommendations contained in all 2 of our findings to further enhance the code's security. It is of utmost priority to start by addressing the most severe exploit discovered by the auditors then followed by the remaining exploits, and finally we will be conducting a re-audit following the implementation of the remediation plan contained in this report.

We would much appreciate any constructive feedback or suggestions regarding our methodology, audit findings, or potential scope gaps in this report.

BLOCKHAT
SECURITY

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