DEFIMOON

be secure

Smart Contract Audit Report

May, 2023

Stasha

CONTACTS

DEFIMOON PROJECT

Audit and Development

defimoon.org
audit@defimoon.org
defimoon_org
defimoonorg
defimoon
defimoonorg

DEFIMOON

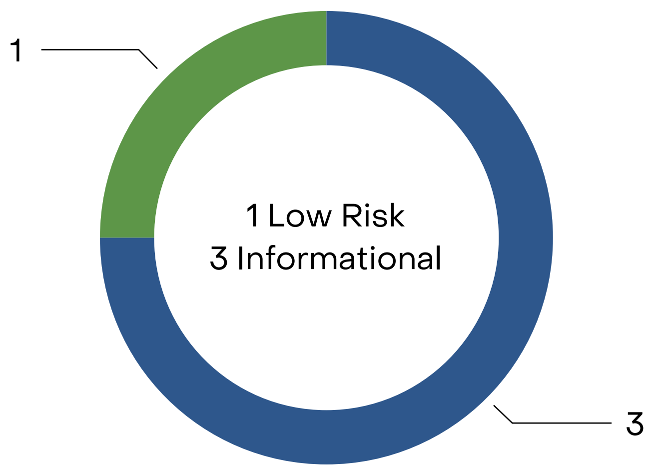
be secure

25 May 2023

This audit report was prepared by DefiMoon for Stasha.

Audit information

Description	Default ERC20 (BEP20) token smart contract
Site	https://Stasha.io
Audited files	StandardToken
Timeline	24 May 2023 - 25 May 2023
Audited by	Ilya Vaganov
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
Source code	https://github.com/stashaHq/STC-Coin/tree/e12859fe4899830f26531a1b6606bba2b33c8d2a
Chain	Binance Smart Chain
Status	Passed



	High Risk	A fatal vulnerability that can cause the loss of all Tokens / Funds.
	Medium Risk	A vulnerability that can cause the loss of some Tokens / Funds.
	Low Risk	A vulnerability which can cause the loss of protocol functionality.
	Informational	Non-security issues such as functionality, style, and convention.

Disclaimer

This audit is not financial, investment, or any other kind of advice and could be used for informational purposes only. This report is not a substitute for doing your own research and due diligence should always be paid in full to any project. Defimoon is not responsible or liable for any loss, damage, or otherwise caused by reliance on this report for any purpose. Defimoon has based this audit report solely on the information provided by the audited party and on facts that existed before or during the audit being conducted. Defimoon is not responsible for any outcome, including changes done to the contract/contracts after the audit was published. This audit is fully objective and only discerns what the contract is saying without adding any opinion to it. Defimoon has no connection to the project other than the conduction of this audit and has no obligations other than to publish an objective report. Defimoon will always publish its findings regardless of the outcome of the findings. The audit only covers the subject areas detailed in this report and unless specifically stated, nothing else has been audited. Defimoon assumes that the provided information and materials were not altered, suppressed, or misleading. This report is published by Defimoon, and Defimoon has sole ownership of this report. Use of this report for any reason other than for informational purposes on the subjects reviewed in this report including the use of any part of this report is prohibited without the express written consent of Defimoon. In instances where an auditor or team member has a personal connection with the audited project, that auditor or team member will be excluded from viewing or impacting any internal communication regarding the specific audit.

Audit Information

Defimoon utilizes both manual and automated auditing approach to cover the most ground possible. We begin with generic static analysis automated tools to quickly assess the overall state of the contract. We then move to a comprehensive manual code analysis, which enables us to find security flaws that automated tools would miss. Finally, we conduct an extensive unit testing to make sure contract behaves as expected under stress conditions.

In our decision making process we rely on finding located via the manual code inspection and testing. If an automated tool raises a possible vulnerability, we always investigate it further manually to make a final verdict. All our tests are run in a special test environment which matches the "real world" situations and we utilize exact copies of the published or provided contracts.

While conducting the audit, the Defimoon security team uses best practices to ensure that the reviewed contracts are thoroughly examined against all angles of attack. This is done by evaluating the codebase and whether it gives rise to significant risks. During the audit, Defimoon assesses the risks and assigns a risk level to each section together with an explanatory comment.

Audit overview

No major issues were found.

The token contract contains no major issues or vulnerabilities, but uses an outdated compiler version.
We recommend updating the compiler version to at least 0.8.0 and not using SafeMath.

In addition, the contract is a basic version of an ERC-20 contract, whereby it might look like this:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.0;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
import "@openzeppelin/contracts/access/Ownable.sol";

contract StashaToken is ERC20, Ownable {

   constructor() ERC20("Stasha Coin", "STC") {
      _mint(_msgSender(100000000e18), _amount);
   }

   function mint(uint256 amount) public onlyOwner returns (bool) {
      _mint(_msgSender(), amount);
      return true;
   }

   function burn(uint256 amount) external returns (bool) {
      _burn(_msgSender(), amount);
      return true;
   }

   function renounceOwnership() public override onlyOwner {
      revert("Safety: Is not allowed");
   }
}
```

This approach allows you to reduce the number of potential issues and vulnerabilities, and also helps you follow the best programming practices and use the latest functionality and interfaces by using the latest implementations from OpenZeppelin.

Summary of findings

ID	**Description**	**Severity**
DFM-1	Possible loss of rights to the contract	Low Risk
DFM-2	Solidity version too old	Informational
DFM-3	Unused function	Informational
DFM-4	Repetitive functions	Informational

Application security checklist

Compiler errors	Passed
Possible delays in data delivery	Passed
Timestamp dependence	Passed
Integer Overflow and Underflow	Passed
Race Conditions and Reentrancy	Passed
DoS with Revert	Passed
DoS with block gas limit	Passed
Methods execution permissions	Passed
Private user data leaks	Passed
Malicious Events Log	Passed
Scoping and Declarations	Passed
Uninitialized storage pointers	Passed
Arithmetic accuracy	Passed
Design Logic	Passed
Cross-function race conditions	Passed

Detailed Audit Information

Contract Programming

Solidity version not specified	Passed
Solidity version too old	Passed
Integer overflow/underflow	Passed
Function input parameters lack of check	Passed
Function input parameters check bypass	Passed
Function access control lacks management	Passed
Critical operation lacks event log	Passed
Human/contract checks bypass	Passed
Random number generation/use vulnerability	Passed
Fallback function misuse	Passed
Race condition	Passed
Logical vulnerability	Passed
Other programming issues	Passed

Code Specification

Visibility not explicitly declared	Passed
Variable storage location not explicitly declared	Passed
Use keywords/functions to be deprecated	Passed
Other code specification issues	Passed

Gas Optimization

Assert () misuse	Passed
High consumption ‘for/while’ loop	Passed
High consumption ‘storage’ storage	Passed
“Out of Gas” Attack	Passed

Findings

DFM-1 «Possible loss of rights to the contract»

Severity: Low Risk

Description: The inherited Ownable contract from OpenZeppelin has a renounceOwnership() function that removes the owner of the contract. This function can be accidentally called, leaving the contract without an owner.

Recommendation: In your case, the contract does not contain any functionality available only to the owner, so there are no visible vulnerabilities. But if other contracts in your ecosystem will have logic in which the owner of this contract participates, then you should be careful.

It can be fixed like this:

```
function renounceOwnership() public override onlyOwner {
    revert("Safety: Is not allowed");
}
```

DFM-2 «Solidity version too old»

Severity: Informational

Description: The rather old version of Solidity 0.5.16 is used, although the minimum recommended version is 0.8.0.

Recommendation: Try to use the latest version of the compiler to follow the best development practices. Also, since version 0.8.0 Solidity no longer needs to use the SafeMath library as it is built in.

DFM-3 «Unused function»

Severity: Informational

Description: The _burnFrom function is declared internal but is not used in other functions.

Recommendation: If this function is needed, then it must implement a public or external version, thanks to which it can be called. Otherwise, delete it.

DFM-4 «Repetitive functions»

Severity: Informational

Description: The Ownable::owner() and StashaToken::getOwner() functions do the same thing. Moreover, the StashaToken::getOwner() function returns the result of the Ownable::owner() function.

Recommendation: Make sure the StashaToken::getOwner() function is needed or remove it and use only Ownable::owner().

Automated Analyses

Slither

Slither's automatic analysis not found vulnerabilities, or these false positives results .

Methodology

Manual Code Review

We prefer to work with a transparent process and make our reviews a collaborative effort. The goal of our security audits is to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Vulnerability Analysis

Our audit techniques include manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, review open issue tickets, and investigate details other than the implementation.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system to make a final decision.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Appendix A — Finding Statuses

Resolved	Contracts were modified to permanently resolve the finding
Mitigated	The finding was resolved by other methods such as revoking contract ownership or updating the code to minimize the effect of the finding
Acknowledged	Project team is made aware of the finding
Open	The finding was not addressed