DEFIMOON

be secure

Smart Contract Audit Report

August, 2023

Standard

DEFIMOON PROJECT

Audit and
Development

CONTACTS

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

DEFIMOON

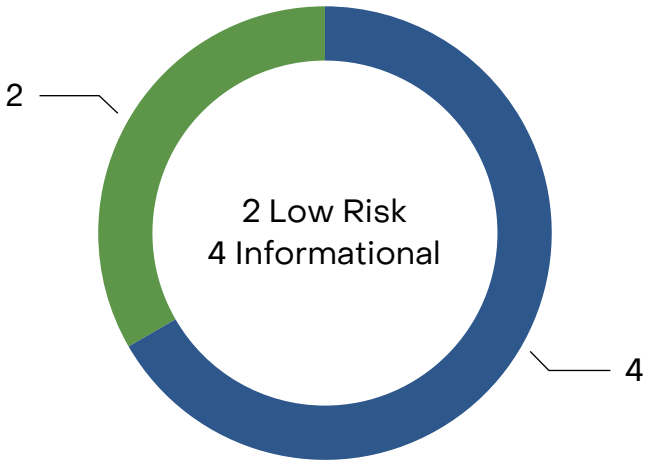
be secure

29 August 2023

This audit report was prepared by DefiMoon for Standard Protocol 2.0.

Audit information

Description	Decentralized order book
Audited files	MatchingEngine.sol
Timeline	21 August 2023 - 29 August 2023
Approved by	Artur Makhnach, Kirill Minyaev
Languages	Solidity
Methods	Architecture Review, Unit Testing, Functional Testing, Manual Review
WhitePaper	https://github.com/standardweb3/Whitepaper/blob/main/whitepaper_en.md
Source code	https://github.com/standardweb3/standard-2.0-contracts/tree/b1465efabf3a63880bda8585c63772331714957c
Network	EVM-like
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

Major vulnerabilities were not found.

The contract is well written, but has a lot of duplicate functionality that could be replaced with generic functions to make it easier to change functionality during fixes and updates, and to reduce deployment gas costs.

The contract also uses an optimistic approach to predicting the orberbook address, which may not always exist and interaction with which will return an unspecified error.

Summary of findings

ID	Description	Severity
DFM-1	Invalid use of nonReentrant	Low Risk
DFM-2	Using upgradeable versions of contracts	Low Risk
DFM-3	Disabling initializing	Informational
DFM-4	The existence of the contract is not checked	Informational
DFM-5	Conversion optimization	Informational
DFM-6	Field indexing in events	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 «Invalid use of nonReentrant»

Severity: Low Risk

Description: The cancelOrders and rematchOrder functions use the nonReentrant modifier and call functions that also use the nonReentrant modifier, resulting in a "ReentrancyGuard: reentrant call" error.

Recommendation: You can use a combination of private and public functions like this:

```
function f1() public nonReentrant {
    _f1();
}

function _f1() private {
    // your logic
}

function f2() external nonReentrant {
    // your logic
    _f1();
    // your logic
}
```

DFM-2 «Using upgradeable versions of contracts»

Severity: Low Risk

Description: The contract uses a proxy, but uses a non-upgradeable version of the inherited openzeppelin::ReentrancyGuard contract. As a result, with future updates or fixes to the logic of the openzeppelin::ReentrancyGuard contract, updating to a new version can be difficult due to differences in contract storage.

Recommendation: We recommend using the upgradeable implementation of openzeppelin::ReentrancyGuard to be able to easily upgrade to a new version, since the upgradeable implementation of openzeppelin::ReentrancyGuard reserves slots in storage for future upgrades.

DFM-3 «Disabling initializing»

Severity: Information

Description: Since the upgradeable version of the contract is used, the initialize function is not called on the implementation contract and can be called by anyone. It is better to block the call of this function on the implementation contract.

Recommendation: We recommend disabling the initialize function, as recommended by OpenZeppelin: «Locks the contract, preventing any future reinitialization. This cannot be part of an initializer call. Calling this in the constructor of a contract will prevent that contract from being initialized or reinitialized to any version. It is recommended to use this to lock implementation contracts that are designed to be called through proxies.»

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() {
    _disableInitializers();
}
```

DFM-4 «The existence of the contract is not checked»

Severity: Information

Description: The OrderbookFactory::getBookByPair function does not return the address of an existing orderbook, but predicts its address for a pair of base and quote tokens, as a result of which, if the orderbook has not been deployed, an unspecified error will be generated when trying to call its functions.

Recommendation: We recommend using mapping(base => mapping(quote => orderbook)) to store the addresses of deployed orderbooks, or explicitly handle cases where the contract doesn't exist and return a readable error.

DFM-5 «Conversion optimization»

Severity: Information

Description: The _convert function uses a check (base == quote), but the factory forbids the creation of such an orderbook.

The _convert function uses the check (orderbook == address(0)), but address(0) cannot be returned by the OrderbookFactory::getBookByPair function.

Recommendation: Extra checks can be removed:

```
function _convert(
    address base,
    address quote,
    uint256 price,
    uint256 amount,
    bool isBid
) internal view returns (uint256) {
    address orderbook = getBookByPair(base, quote);
    return IOrderbook(orderbook).convert(price, amount, isBid);
}
```

DFM-6 «Field indexing in events»

Severity: Information

Description: The contract uses events for all major operations, but does not use field indexing.

Recommendation: We recommend using the indexing of the main fields in events to simplify the search for them. Events can be an important part of the integration of smart contracts with the UI of the protocol, and can also be used to collect statistics and analyze data.

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