BLOCKSEC

Security Audit Report for Lotus Protocol

Date: July 30, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Mandala Labs
Target	Lotus Protocol

Version History

Version	Date	Description
1.0	July 30, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-07-30 11:54:04+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1] of Lotus Protocol of Mandala Labs.

The Lotus protocol of Mandala Labs is a decentralized exchange (i.e., DEX) protocol forked from Uniswap V2 and V3.

Note this audit focuses on the smart contracts located in the lotus-v2-core/contracts directory, lotus-v2-periphery/contracts directory, lotus-v3-core/contracts directory and lotus-v3-periphery/contracts directory, excluding the following directories/files:

- lotus-v2-core/contracts/test/*
- lotus-v2-periphery/contracts/test/*
- lotus-v3-core/contracts/test/*
- lotus-v3-periphery/contracts/test/*

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
lotus-v2-core	Version 0	ee547b17853e71ed4e0101ccfd52e70d5acded58
	Version 1	afb7e550a8930c322babc78b7d37ee817b6ad708
	Version 2	e42957d78a0d7278d2eca33f336c892a8dffb613
lotus-v2-periphery	Version 0	0335e8f7e1bd1e8d8329fd300aea2ef2f36dd19f
	Version 1	45ec8a9bbf87d5eb2cc46be96d18bd2cb0bb4eaa
lotus-v3-core	Version 0	d8b1c635c275d2a9450bd6a78f3fa2484fef73eb
	Version 1	7b4b45556ecb2247c9ea19f228124f08e5a453a9
	Version 2	e667af8ba78679b3aa0b9d009cec5f919f90aaa1
lotus-v3-periphery	Version 0	0682387198a24c7cd63566a2c58398533860a5d1
	Version 1	536f9792c7c372dab0f261d947fc68610ed11340
	Version 2	66b2d07f9027a438c969dba20d65876b56d7a93a

[1] https://github.com/mandala-labs-org/lotus-v2-core, https://github.com/mandala-labs-org/lotus-v2-core, https://github.com/mandala-labs-org/lotus-v2-periphery, https://github.com/mandala-labs-org/lotus-v3-core, https://github.com/mandala-labs-org/lotus-v3-periphery

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness

- Oracle security
- Business logic correctness
- Semantic and functional consistency
- Emergency mechanism
- Economic and incentive impact

1.3.2 Additional Recommendation

- Gas optimization
- Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [2] and Common Weakness Enumeration [3]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **one** potential security issue. Besides, we have **two** recommendations and **five** notes.

- Low Risk: 1
- Recommendation: 2
- Note: 5

ID	Severity	Description	Category	Status
1	Low	Potential signature verification failures and replay attacks due to hard forks	Security Issue	Fixed
2	-	Add checks for the input `_owner`	Recommendation	Confirmed
3	-	Add existence checks for the input `token` in the function `safeTransfer()`	Recommendation	Fixed
4	-	Potential centralization risks	Note	-
5	-	Token integration issues	Note	-
6	-	Hardcoded addresses and the contract deployment chains	Note	-
7	-	Ensure proper bounds in the function `swap()` to prevent potential DoS	Note	-
8	-	Instant invocation of the function `Initialize()` after pool creations	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Potential signature verification failures and replay attacks due to hard forks

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `LotusV2ERC20`, the constructor initializes the variable `DOMAIN_SEPARATOR` using the value `chainid`. However, the design is improper due to potential blockchain hard forks which may alter the value `chainid`. Specifically, this could invalidate the variable `DOMAIN_SEPARATOR` and affect signature verification in the function `permit()`. As a result, this design may lead to verification failures and potential replay attacks.

```
24    constructor() public {
25        uint chainId;
26        assembly {
27            chainId := chainid
28        }
29        DOMAIN_SEPARATOR = keccak256(
30            abi.encode(
31                keccak256('EIP712Domain(string name,string version,uint256 chainId,address
                      verifyingContract)'),
```

```
32              keccak256(bytes(name)),
33              keccak256(bytes('1')),
34              chainId,
35              address(this)
36          )
37      );
38  }
```

Listing 2.1: lotus-v2-core/contracts/LotusV2ERC20.sol

```
81  function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r,
        bytes32 s) external {
82      require(deadline >= block.timestamp, 'LotusV2: EXPIRED');
83      bytes32 digest = keccak256(
84          abi.encodePacked(
85              '\x19\x01',
86              DOMAIN_SEPARATOR,
87              keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++,
                    deadline))
88          )
89      );
90      address recoveredAddress = ecrecover(digest, v, r, s);
```

Listing 2.2: lotus-v2-core/contracts/LotusV2ERC20.sol

Impact Potential verification failures and replay attacks due to hard forks.

Suggestion Revise the logic accordingly.

2.2 Recommendation

2.2.1 Add checks for the input `_owner`

Status Confirmed

Introduced by `Version 1`

Description In the contract `LotusV3Factory`, both the constructor and the function `setOwner()` fail to implement proper non-zero address checks for the input `_owner`. Furthermore, the function `setOwner()` should include a duplicate check for the input `_owner` to prevent redundant owner assignments. It is recommended to implement these checks to prevent unintended behaviors.

```
22  constructor() {
23      owner = msg.sender;
24      emit OwnerChanged(address(0), msg.sender);
```

Listing 2.3: lotus-v3-core/contracts/LotusV3Factory.sol

```
54  function setOwner(address _owner) external override {
55      require(msg.sender == owner);
56      emit OwnerChanged(owner, _owner);
57      owner = _owner;
```

```
58    }
```

Listing 2.4: lotus-v3-core/contracts/LotusV3Factory.sol

Suggestion Add proper checks for the input `_owner` in both the constructor and the function `setOwner()`.

2.2.2 Add existence checks for the input `token` in the function `safeTransfer()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TransferHelper`, the function `safeTransfer()` lacks a check to verify whether the input `token` represents a valid contract. Specifically, the function `safeTransfer()` performs a low-level call for token transfer, which will return success (i.e., true) even when the target contract (i.e., `token`) does not exist. It is recommended to add proper existence checks for the input `token` in the function `safeTransfer()`.

```
19        (bool success, bytes memory data) =
20            token.call(abi.encodeWithSelector(IERC20Minimal.transfer.selector, to, value));
21        require(success && (data.length == 0 || abi.decode(data, (bool))), 'TF');
```

Listing 2.5: lotus-v3-core/contracts/libraries/TransferHelper.sol

Suggestion Add existence checks for the input `token` in the function `safeTransfer()`.

2.3 Note

2.3.1 Potential centralization risks

Introduced by `Version 1`

Description In this project, several privileged roles (e.g., the variable `owner` in the contract `LotusV3Factory`) can conduct sensitive operations, which introduces potential centralization risks. If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project acknowledged the potential centralization risks and decided to use multisig for the owner address.

2.3.2 Token integration issues

Introduced by `Version 1`

Description The project exhibits integration challenges with certain token types (e.g., fee-on-transfer, rebasing, and reflection tokens). Users should be explicitly informed about these limitations and advised to carefully consider token selection when creating pools to ensure proper functionality.

Feedback from the project The project stated that they will inform users which token should be used in the protocol. The project will not support token types like fee-on-transfer, rebasing, reflection tokens, etc.

2.3.3 Hardcoded addresses and the contract deployment chains

Introduced by Version 1

Description In the project, some hardcoded addresses are used in some contracts (e.g., the contract `NonfungibleTokenPositionDescriptor`). Therefore, the project must ensure the protocols are deployed on the proper chains for use.

Feedback from the project The project stated that they will hardcoded correct addresses for specific chains.

2.3.4 Ensure proper bounds in the function `swap()` to prevent potential DoS

Introduced by Version 1

Description In the contract `LotusV3Pool`, the function `swap()` contains a while loop that processes swaps by iterating ticks. The project must ensure proper bounds (e.g., using a proper `sqrtPriceLimitX96` in the frontend) to avoid a gas exhaustion DoS issue happening during the tick finding process.

Feedback from the project The project stated that their front-end will use proper SDK functions to avoid gas exhaustion issue during the tick finding process.

2.3.5 Instant invocation of the function `Initialize()` after pool creations

Introduced by Version 1

Description In the contract `UniswapV3Pool`, an attacker is allowed to front-run the invocation of the function `initialize()` to set a malicious price for created pools. The project must notify users to follow secure pool creation procedures by invoking the function `initialize()` properly.

Feedback from the project The project stated that their front-end will create pools securely via the function `createAndInitializePoolIfNecessary()`.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS