RD AUDITORS

Moonbase Alpha, Smart Contract, Code Review and Security Analysis Report

Customer: Moonbase Alpha
Prepared on: 15th May, 2023
Platform: Arbitrum
Language: Solidity

rdauditors.com

Table of Contents

Disclaimer

This document may contain confidential information about its systems and intellectual property of the customer as well as information about potential vulnerabilities and methods of their exploitation.

The report containing confidential information can be used internally by the customer or it can be disclosed publicly after all vulnerabilities are fixed - upon the decision of the customer.

Documentation

Name	Smart Contract Code Review and Security Analysis Report of Moonbase Alpha
Platform	Arbitrum/ Solidity
File 1	MoonbaseTokenLocker.sol
MD5 hash	5dfdffb6391b51300eb49c960aed9f45
SHA256 hash	0c90e6e8c591ed9b63476d9cbb0e28e21edc9c59ab106816124423fff9b1b8c9
Date	15/05/2023

Introduction

RD Auditors (Consultant) were contracted by Moonbase Alpha (Customer) to conduct a Smart Contract Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contract and its code review conducted between 08th- 15th May, 2023.

This contract consists of one file.

Project Scope

The scope of the project is a smart contract. We have scanned this smart contract for commonly known and more specific vulnerabilities, below are those considered (the full list includes but is not limited to):

- Reentrancy
- Timestamp Dependence
- Gas Limit and Loops
- DoS with (Unexpected) Throw
- DoS with Block Gas Limit
- Transaction-Ordering Dependence
- Byte array vulnerabilities
- Style guide violation
- Transfer forwards all gas
- ERC20 API violation
- Malicious libraries
- Compiler version not fixed
- Unchecked external call - Unchecked math
- Unsafe type inference
- Implicit visibility level

Executive Summary

According to the assessment, the customer's solidity smart contract is now **Well-Secured.**

Automated checks are with smartDec, Mythril, Slither and remix IDE. All issues were performed by our team, which included the analysis of code functionality, the manual audit found during automated analysis were manually reviewed and applicable vulnerabilities are presented in the audit overview section. The general overview is presented in the AS-IS section and all issues found are located in the audit overview section.

We found the following;

Total Issues	2
Critical	0
High	0
Medium	0
Low	0
Very Low	2

Code Quality

The libraries within this smart contract are part of a logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned to a specific address and its properties/methods can be reused many times by other contracts.

The Moonbase Alpha team has not provided scenario and unit test scripts, which helped to determine the integrity of the code in an automated way.

Documentation

We were given a Moonbase Alpha smart contract code in the form of a source code. The hash of that code is mentioned above in the table. As mentioned above, It's recommended to write comments in the smart contract code, so anyone can quickly understand the programming flow as well as complex code logic.

Comments are very helpful in understanding the overall architecture of the protocol. It also provides a clear overview of the system components, including helpful details, like the lifetime of the background script.

Use of Dependencies

As per our observation, the libraries are used in this smart contract infrastructure. Those were based on well known industry standard open source projects and even core code blocks that are written well and systematically.

AS-IS Overview

MoonbaseTokenLocker.sol

File And Function Level Report

File : MoonbaseTokenLocker.sol

Contract: MoonbaseTokenLocker

Inherit: IMoonbaseLock, Ownable

Observation: Passed

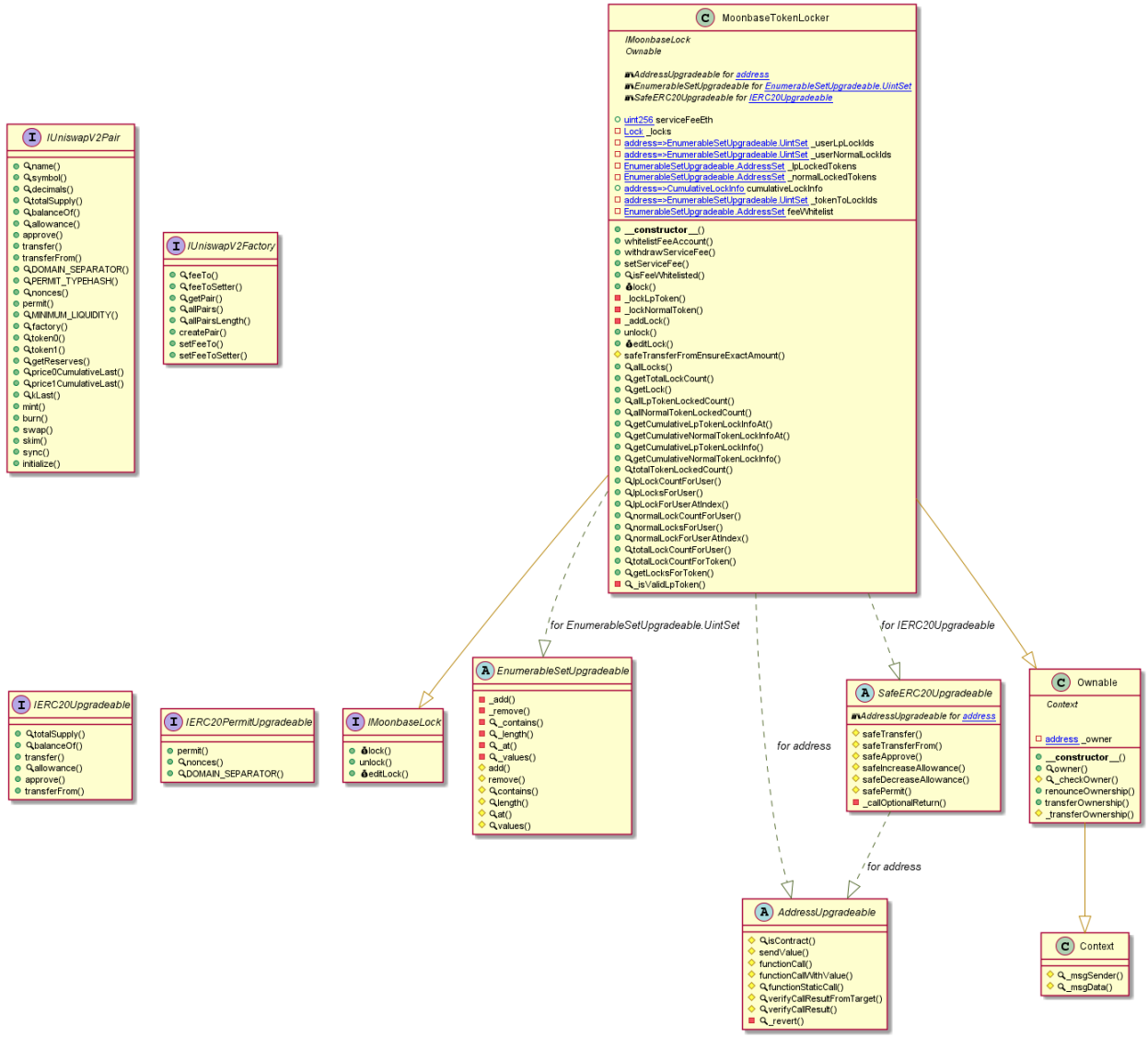
Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	whitelistFeeAccount	public	Passed	All Passed	No Issue	Passed
2	withdrawServiceFee	public	Passed	All Passed	No Issue	Passed
3	setServiceFee	public	Passed	All Passed	No Issue	Passed
4	isFeeWhitelisted	public	Passed	All Passed	No Issue	Passed
5	Lock	public	Passed	All Passed	No Issue	Passed
6	_lockLpToken	private	Passed	All Passed	No Issue	Passed
7	_lockNormalToken	private	Passed	All Passed	No Issue	Passed
8	_addLock	private	Passed	All Passed	No Issue	Passed
9	unlock	external	Passed	All Passed	No Issue	Passed
10	editLock	external	Passed	All Passed	No Issue	Passed

11	safeTransferFromEnsureExactAmount	internal	Passed	All Passed	No Issue	Passed
12	allLocks	public	Passed	All Passed	No Issue	Passed
13	getTotalLockCount	public	Passed	All Passed	No Issue	Passed
14	getLock	public	Passed	All Passed	No Issue	Passed
15	allLpTokenLockedCount	public	Passed	All Passed	No Issue	Passed
16	allNormalTokenLockedCount	public	Passed	All Passed	No Issue	Passed
17	getCumulativeLpTokenLockInfoAt	public	Passed	All Passed	No Issue	Passed
18	getCumulativeNormalTokenLockInfoAt	public	Passed	All Passed	No Issue	Passed
19	getCumulativeLpTokenLockInfo	public	Passed	All Passed	No Issue	Passed
20	getCumulativeNormalTokenLockInfo	public	Passed	All Passed	No Issue	Passed
21	totalTokenLockedCount	public	Passed	All Passed	No Issue	Passed
22	lpLockCountForUser	public	Passed	All Passed	No Issue	Passed
23	lpLocksForUser	public	Passed	All Passed	No Issue	Passed
24	lpLockForUserAtIndex	public	Passed	All Passed	No Issue	Passed
25	normalLockCountForUser	public	Passed	All Passed	No Issue	Passed
26	normalLocksForUser	public	Passed	All Passed	No Issue	Passed

27	normalLockForUserAtIndex	public	Passed	All Passed	No Issue	Passed
28	totalLockCountForUser	public	Passed	All Passed	No Issue	Passed
29	totalLockCountForToken	public	Passed	All Passed	No Issue	Passed
30	getLocksForToken	public	Passed	All Passed	No Issue	Passed
31	_isValidLpToken	private	Passed	All Passed	No Issue	Passed

Code Flow Diagram - MoonbaseTokenLocker.sol



Code Flow Diagram - Slither Results Log

MoonbaseTokenLocker.sol

```
MoonbaseTokenLocker.lock(address,address,bool,uint256,uint256).owner (MoonbaseTokenLocker.sol#790) shadows:
        - Ownable.owner() (MoonbaseTokenLocker.sol#694-696) (function)
MoonbaseTokenLocker._lockLpToken(address,address,address,uint256,uint256).owner (MoonbaseTokenLocker.sol#823) shadows:
        - Ownable.owner() (MoonbaseTokenLocker.sol#694-696) (function)
MoonbaseTokenLocker._lockNormalToken(address,address,uint256,uint256).owner (MoonbaseTokenLocker.sol#844) shadows:
        - Ownable.owner() (MoonbaseTokenLocker.sol#694-696) (function)
MoonbaseTokenLocker._addLock(address,address,uint256,uint256).owner (MoonbaseTokenLocker.sol#863) shadows:
        - Ownable.owner() (MoonbaseTokenLocker.sol#694-696) (function)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#local-variable-shadowing

Variable 'MoonbaseTokenLocker.lock(address,address,bool,uint256,uint256).factory (MoonbaseTokenLocker.sol#804)' in MoonbaseTok
enLocker.lock(address,address,bool,uint256,uint256) (MoonbaseTokenLocker.sol#789-820) potentially used before declaration: pos
sibleFactoryAddress = factory (MoonbaseTokenLocker.sol#805)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#pre-declaration-usage-of-local-variables

Reentrancy in MoonbaseTokenLocker.editLock(uint256,uint256,uint256) (MoonbaseTokenLocker.sol#915-947):
        External calls:
        - safeTransferFromEnsureExactAmount(userLock.token,msg.sender,address(this),diff) (MoonbaseTokenLocker.sol#937)
                - returndata = address(token).functionCall(data,SafeERC20: low-level call failed) (MoonbaseTokenLocker.sol#664
)
                - IERC20Upgradeable(token).safeTransferFrom(sender,recipient,amount) (MoonbaseTokenLocker.sol#956)
                - (success,returndata) = target.call{value: value}(data) (MoonbaseTokenLocker.sol#499)
        External calls sending eth:
        - safeTransferFromEnsureExactAmount(userLock.token,msg.sender,address(this),diff) (MoonbaseTokenLocker.sol#937)
                - (success,returndata) = target.call{value: value}(data) (MoonbaseTokenLocker.sol#499)
        State variables written after the call(s):
        - tokenInfo.amount = tokenInfo.amount + diff (MoonbaseTokenLocker.sol#942)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
```

```
Pragma version^0.8.9 (MoonbaseTokenLocker.sol#2) allows old versions
solc-0.8.9 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in AddressUpgradeable.sendValue(address,uint256) (MoonbaseTokenLocker.sol#465-470):
        - (success) = recipient.call{value: amount}() (MoonbaseTokenLocker.sol#468)
Low level call in AddressUpgradeable.functionCallWithValue(address,bytes,uint256,string) (MoonbaseTokenLocker.sol#492-501):
        - (success,returndata) = target.call{value: value}(data) (MoonbaseTokenLocker.sol#499)
Low level call in AddressUpgradeable.functionStaticCall(address,bytes,string) (MoonbaseTokenLocker.sol#507-514):
        - (success,returndata) = target.staticcall(data) (MoonbaseTokenLocker.sol#512)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Function IUniswapV2Pair.DOMAIN_SEPARATOR() (MoonbaseTokenLocker.sol#27) is not in mixedCase
Function IUniswapV2Pair.PERMIT_TYPEHASH() (MoonbaseTokenLocker.sol#29) is not in mixedCase
Function IUniswapV2Pair.MINIMUM_LIQUIDITY() (MoonbaseTokenLocker.sol#55) is not in mixedCase
Function IERC20PermitUpgradeable.DOMAIN_SEPARATOR() (MoonbaseTokenLocker.sol#590) is not in mixedCase
Parameter MoonbaseTokenLocker.whitelistFeeAccount(address,bool)._user (MoonbaseTokenLocker.sol#769) is not in mixedCase
Parameter MoonbaseTokenLocker.whitelistFeeAccount(address,bool)._add (MoonbaseTokenLocker.sol#769) is not in mixedCase
Parameter MoonbaseTokenLocker.setServiceFee(uint256)._serviceFeeEth (MoonbaseTokenLocker.sol#781) is not in mixedCase
Parameter MoonbaseTokenLocker.isFeeWhitelisted(address)._user (MoonbaseTokenLocker.sol#785) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions

Redundant expression "this (MoonbaseTokenLocker.sol#676)" inContext (MoonbaseTokenLocker.sol#671-679)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements

MoonbaseTokenLocker.lock(address,address,bool,uint256,uint256) (MoonbaseTokenLocker.sol#789-820) uses literals with too many d
igits:
        - require(bool,string)(unlockDate < 10000000000,TIMESTAMP INVALID) (MoonbaseTokenLocker.sol#796)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#too-many-digits
MoonbaseTokenLocker.sol analyzed (11 contracts with 84 detectors), 71 result(s) found
```

Solidity Static Analysis

MoonbaseTokenLocker.sol

Security

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in MoonbaseTokenLocker.lock(address,address,bool,uint256,uint256): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 84:4:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.

more

Pos: 220:72:

Gas & Economy

Gas costs:

Gas requirement of function MoonbaseTokenLocker.whitelistFeeAccount is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)

Pos: 64:4:

Gas costs:

Gas requirement of function MoonbaseTokenLocker.lpLocksForUser is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 326:4:

Gas costs:

Gas requirement of function MoonbaseTokenLocker.normalLocksForUser is infinite: If the gas requirement of a function is higher than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 344:4:

Miscellaneous

Constant/View/Pure functions:

IMoonbaseLock.unlock(uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 13:4:

Guard conditions:

Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external component.
more
Pos: 354:8:

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to lost tokens etc.
High	High level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g. public access to crucial functions.
Medium	Medium level vulnerabilities are important to fix; however, they cannot lead to lost tokens.
Low	Low level vulnerabilities are most related to outdated, unused etc. These code snippets cannot have a significant impact on execution.
Lowest Code Style/ Best Practice	Lowest level vulnerabilities, code style violations and information statements cannot affect smart contract execution and can be ignored.

Audit Findings

Critical:

No critical severity vulnerabilities were found.

High:

No high severity vulnerabilities were found.

Medium:

No medium severity vulnerabilities were found.

Low:

No low severity vulnerabilities were found.

Very Low:

(1) _lpLockedTokens.add(token); the storage will increase as this keeps a record of the same address multiple times.

```
function _lockLpToken(
    address owner,
    address token,
    address factory,
    uint256 amount,
    uint256 unlockDate
) private returns (uint256 id) {
    id = _addLock(owner, token, amount, unlockDate);
    _userLpLockIds[owner].add(id);
    _lpLockedTokens.add(token);

    CumulativeLockInfo storage tokenInfo = cumulativeLockInfo[token];
    if (tokenInfo.token == address(0)) {
        tokenInfo.token = token;
        tokenInfo.factory = factory;
    }
    tokenInfo.amount = tokenInfo.amount + amount;

    _tokenToLockIds[token].add(id);
}
```

(2)_normalLockedTokens.add(token); the storage will increase as this keeps a record of the same address multiple times.

```
function _lockNormalToken(
    address owner,
    address token,
    uint256 amount,
    uint256 unlockDate
) private returns (uint256 id) {
    id = _addLock(owner, token, amount, unlockDate);
    _userNormalLockIds[owner].add(id);
    _normalLockedTokens.add(token);

    CumulativeLockInfo storage tokenInfo = cumulativeLockInfo[token];
    if (tokenInfo.token == address(0)) {
        tokenInfo.token = token;
        tokenInfo.factory = address(0);
    }
    tokenInfo.amount = tokenInfo.amount + amount;

    _tokenToLockIds[token].add(id);
}
```

Conclusion

We were given a contract code in the form of a source code and have used all possible tests based on the given object. So it is ready to go for production. We have used all the latest static tools and manual observations to cover maximum possible test cases to scan everything.

The security state of the reviewed contract is “**well-Secured**”.

Note For Contract Users

Technical auditing does not guarantee the project's ethical side.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are 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.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included 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, skim open issue tickets, and generally 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 generally 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 analyse the feasibility of an attack in a live system.

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 scrutinised 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.

Disclaimers

RD Auditors Disclaimer

The smart contracts given for audit have been analysed in accordance with the best industry practices at the date of this report, in relation to: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

Because the total number of test cases are unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only - we recommend proceeding with several independent audits and a public bug bounty program to ensure security of smart contracts.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Email: info@rdauditors.com

Website: www.rdauditors.com