RD AUDITORS

Moonscore Smart Contract, Code Review and Security Analysis Report

Customer: Moonscore
Prepared on: 16th October 2021
Platform: BSC
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.

Document

Name	Smart Contract Code Review and Security Analysis Report of Moonscore
Platform	BSC / Solidity
File 1	Basic.sol
MD5 hash	3E5D57B7D105BBD333F2B8A7F8E85C76
SHA256 hash	837D94826D67F68E8837E2B721D18EEE97C06E81CDDB04DA9D0AC12B93A88C36
File 2	Pixel.sol
MD5 hash	E74E4EE05731DAD93E065C4A9D33CC97
SHA256 hash	18C77CA652CA20992AA71CFA759F20F97E2EA66B50583C484E55A1FE62EA25DD
File 3	PixelFarm.sol
MD5 hash	75434F824DCF9ADE0F05B24926202CF0
SHA256 hash	1D988F21F6BF945D4C501FE2DDEBF064A3861DC4F479FC3411C6E1AFBCA15FF9
File 4	ZapVault.sol

MD5 hash	B8F2F06B87CA18A308294751714FCE32
SHA256 hash	F930D32A7C87058D84031A3CD0DB5A4F01C7DC2A9D3BB0EAB15 B846ADF1CDE4C
File 5	Migrations.sol
MD5 hash	28EC9047C38780DB52FF4F0F9E553831
SHA256 hash	105AE27F87C4014E470A6EE9C65A37866FF7407188EBCED308C3F0 10F83361E4
Date	16/10/2021

Introduction

RD Auditors (Consultant) were contracted by Moonscore (Customer) to conduct a Smart Contracts Code Review and Security Analysis. This report represents the findings of the security assessment of the customer`s smart contracts and its code review conducted between 07 - 16 Oct 2021.

This contract consists of five files.

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 **well-secured.**

You are Here

Insecure   Poorly Secured   Secure   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	0
Critical	0
High	0
Medium	0
Low	0
Very Low	0

Code Quality

Please note that within this report EnumerableSet, ReentrancyGuard, ERC20 Address, Ownable, SafeMath, Pausable, ERC721, IERC165, IERC721Receiver are taken from the popular OpenZeppelin library.

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 Moonscore team has not provided scenario and unit test scripts, which would help to determine the integrity of the code in an automated way.

Overall, the code is almost commented. Commenting can provide rich documentation for functions, return variables and more. Use of Ethereum Natural Language Specification Format (NatSpec) for commenting is recommended.

Documentation

We were given the Moonscore code as a github link:

https://github.com/halaprix/moonscore_contracts/tree/main/contracts

The hash of that file is mentioned 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

Moonscore

File And Function Level Report

File: Basic.sol

Contract: Basic

Import: EnumerableSet,ReentrancyGuard,ERC20 Address,SafeERC20,Ownable,SafeMath,Pausable

Inherit: Ownable,ReentrancyGuard,Pausable

Observation: Passed

Test Report: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	deposit	write	Passed	All Passed	No Issue	Passed
2	farm	write	Passed	All Passed	No Issue	Passed
3	_farm	write	Passed	All Passed	No Issue	Passed
4	withdraw	write	Passed	All Passed	No Issue	Passed
5	earn	write	Passed	All Passed	No Issue	Passed
6	distributeFees	write	Passed	All Passed	No Issue	Passed
7	convertDustToEarned	write	Passed	All Passed	No Issue	Passed
8	pause	write	Passed	All Passed	No Issue	Passed
9	unpause	write	Passed	All Passed	No Issue	Passed
10	setEntranceFeeFactor	write	Passed	All Passed	No Issue	Passed
11	setControllerFee	write	Passed	All Passed	No Issue	Passed

12	setGov
13	setOnlyGov
14	inCaseTokensG etStuck
15	_safeSwap
16	buyBack

File: Pixel.sol

Contract: PIXEL

Import: ERC20,Address,SafeERC20,Ownable

Inherit: Ownable,ERC20

Observation: Passed

Test Report: Passed

Score: Passed

Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	mint	write	Passed	All Passed	No Issue	Passed

File:	Migrations.sol
Contract:	Migrations
Observation:	Passed
Test Report:	Passed
Score:	Passed
Conclusion:	Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	setCompleted	write	Passed	All Passed	No Issue	Passed

File:	PixelFarm.sol
Contract:	PixelFarm
Import:	EnumerableSet,ReentrancyGuard,ERC20,Address SafeERC20,Ownable,ERC721,IERC165,Address, IERC721Receiver
Inherit:	Ownable,ReentrancyGuard,ERC165,IERC721Receiver
Observation:	Passed
Test Report:	Passed
Score:	Passed
Conclusion:	Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	poolLength	read	Passed	All Passed	No Issue	Passed
2	add	write	Passed	All Passed	No Issue	Passed
3	set	write	Passed	All Passed	No Issue	Passed
4	getMultiplier	read	Passed	All Passed	No Issue	Passed
5	pendingPIXEL	read	Passed	All Passed	No Issue	Passed
6	currentPenalty	read	Passed	All Passed	No Issue	Passed
7	stakedWantTokens	read	Passed	All Passed	No Issue	Passed
8	massUpdatePools	write	Passed	All Passed	No Issue	Passed
9	updatePool	write	Passed	All Passed	No Issue	Passed
10	deposit	write	Passed	All Passed	No Issue	Passed
11	withdraw	write	Passed	All Passed	No Issue	Passed
12	depositNFTv1	write	Passed	All Passed	No Issue	Passed
13	depositNFTv2	write	Passed	All Passed	No Issue	Passed
14	withdrawNFTv1	write	Passed	All Passed	No Issue	Passed
15	withdrawNFTv2	write	Passed	All Passed	No Issue	Passed
16	emergencyWithdraw	write	Passed	All Passed	No Issue	Passed
17	_updateBoostedShares	write	Passed	All Passed	No Issue	Passed
18	safePIXELTransfer	write	Passed	All Passed	No Issue	Passed
19	inCaseTokensGetStuck	write	Passed	All Passed	No Issue	Passed
20	setPenaltyBase	write	Passed	All Passed	No Issue	Passed

21	setZapAddress	write	Passed	All Passed	No Issue	Passed
22	onERC721Received	write	Passed	All Passed	No Issue	Passed

File: ZapVault.sol

Contract: ZapVault

Import: SafeERC20,IERC20, OwnableUpgradeable, IUniswapV2Pair, IUniswapV2Router02

Inherit: OwnableUpgradeable

Observation: Passed

Test Report: Passed

Score: Passed

Conclusion: Passed

Sl.	Function	Type	Observation	Test Report	Conclusion	Score
1	initialize	write	Passed	All Passed	No Issue	Passed
2	receive	write	Passed	All Passed	No Issue	Passed
3	isLp	read	Passed	All Passed	No Issue	Passed
4	routePair	read	Passed	All Passed	No Issue	Passed
5	zapInToken	write	Passed	All Passed	No Issue	Passed
6	zapIn	write	Passed	All Passed	No Issue	Passed

7	zapOut	write	Passed	All Passed	No Issue	Passed
8	getBalanceOfToken	read	Passed	All Passed	No Issue	Passed
9	_approveTokenIfNeeded	write	Passed	All Passed	No Issue	Passed
10	_approveTokenIfNeededVault	write	Passed	All Passed	No Issue	Passed
11	_swapBNBToFlip	write	Passed	All Passed	No Issue	Passed
12	_swapBNBForToken	write	Passed	All Passed	No Issue	Passed
13	_swapTokenForBNB	write	Passed	All Passed	No Issue	Passed
14	_swap	write	Passed	All Passed	No Issue	Passed
15	setRoutePairAddress	write	Passed	All Passed	No Issue	Passed
16	setNotLp	write	Passed	All Passed	No Issue	Passed

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:

No very low severity vulnerabilities were found.

1. Please make sure this hardcoded address is valid or correct.

```
address private constant CAKE = 0x0E09FaBB73Bd3Ade0a17ECC321fD13a19e81cE82;
address private constant DSL = 0x72FEAC4C0887c12db21CEB161533Fd8467469e6b;
address private constant SOUL = 0x67d012F731c23F0313CEA1186d0121779c77fcFE;
// 0x094616f0bdfb0b526bd735bf66eca0ad254ca81f main:0xbb4CdB9CBd36B01bD1cBaEBF2De08
address private constant WBNB = 0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c;
address private constant BUSD = 0xe9e7CEA3DedcA5984780Bafc599bD69ADd087D56;
address private constant USDT = 0x55d398326f99059fF775485246999027B3197955;
address private constant DAI = 0x1AF3F329e8BE154074D8769D1FFa4eE058B1DBc3;
address private constant USDC = 0x8AC76a51cc950d9822D68b83fE1Ad97B32Cd580d;
address private constant VAI = 0x4BD17003473389A42DAF6a0a729f6Fdb328BbBd7;
address private constant BTCB = 0x7130d2A12B9BCbFAe4f2634d864A1Ee1Ce3Ead9c;
address private constant ETH = 0x2170Ed0880ac9A755fd29B2688956BD959F933F8;
```

2. Please check the amount before transfer. This function is executed even for 0 token transfer.

```
// Withdraw without caring about rewards and nfts. EMERGENCY ONLY.
function emergencyWithdraw(uint16 _pid) public nonReentrant {
    PoolInfo storage pool = poolInfo[_pid];
    UserInfo storage user = userInfo[_pid][msg.sender];

    uint256 wantLockedTotal = IStrategy(poolInfo[_pid].strat)
        .wantLockedTotal();
    uint256 sharesTotal = IStrategy(poolInfo[_pid].strat).sharesTotal();
    uint256 amount = (user.shares * wantLockedTotal) / sharesTotal;

    IStrategy(poolInfo[_pid].strat).withdraw(msg.sender, amount);

    pool.want.safeTransfer(address(msg.sender), amount);
    emit EmergencyWithdraw(msg.sender, _pid, amount);
    user.shares = 0;
    user.boostedShares = 0;
    user.rewardDebt = 0;
    user.nftId1 = 0;
```

3. Infinite loop possibility. The block´s gas limit could be reached, if the pool length is larger.

```

    // Update reward variables for all pools. Be careful of gas spending!
    function massUpdatePools() public {
        uint256 length = poolInfo.length;
        for (uint16 pid = 0; pid < length; ++pid) {
            updatePool(pid);
        }
    }

```

Conclusion

We were given a contract file and have used all possible tests based on the given object. The contract is written systematically, so it is ready to go for production.

Since possible test cases can be unlimited and developer level documentation (code flow diagram with function level description) not provided, for such an extensive smart contract protocol, we provide no such guarantee of future outcomes. 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 now “well secured”

Note For Contract Users

Owner has full control over the smart contract. Thus, technical auditing does not guarantee the project's ethical side.

Please do your due diligence before investing. Our audit report is never an investment advice.

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.

RD
AUDITORS

Email: info@rdauditors.com
Website: www.rdauditors.com