SMART CONTRACT AUDIT REPORT

for

Save the Moon

Prepared By: Xiaomi Huang

PeckShield

June 24, 2022

Document Properties

Client	Save the Moon
Title	Smart Contract Audit Report
Target	Save the Moon
Version	1.0-rc
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	June 24, 2022	Xiaotao Wu	Release Candidate #1

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Save The Moon protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Save the Moon

Save the Moon is an initiative by LUNAtics to revive the current Luna Classic economy, giving new hope to millions of LUNC holders. Save the Moon aims to invigorate the LUNC ecosystem by incessantly buying and burning the LUNC supply through the MOON tokens. Whenever MOON is bought, a percentage of LUNC will be bought and sent to burn, diminishing LUNC's supply. When MOON is sold, likewise a percentage of MOON tokens will be sent to burn as well, thereby decreasing the supply of MOON tokens and removing it out of circulation. Constant burning cycles of LUNC will steadily reduce LUNC supply and MOON holders will periodically be rewarded with every LUNC burn through recurring airdrops of MOON tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Save the Moon

Item	Description
Name	Save the Moon
Website	https://savethemoon.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 24, 2022

In the following, we show the Git repository of reviewed files and the commit hash values used in this audit.

- https://github.com/MoonTokenSource/Contracts.git (e49410a)

And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/MoonTokenSource/Contracts.git (46ff39d)

1.2 About PeckShield

PeckShield Inc. [11] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [10]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [9], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Save The Moon` protocol. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	1	
Low	3	
Informational	0	
Total	4	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability and 3 low-severity vulnerabilities.

Table 2.1: Key Save the Moon Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Lack of Slippage Control In LPHelper	Time and State	Confirmed
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Low	Incompatibility with Deflationary/Rebasing Tokens	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Lack of Slippage Control In LPHelper

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: AddLiquidityHelper
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The AddLiquidityHelper contract of the Save The Moon protocol provides two helper routines for users to add liquidity pair to the Uniswap pools. By design, these two helper routines restrict one of the assets in the liquidity pair should be MOON. In particular, the addMOONETHLiquidity() routine is used to add liquidity by providing the MOON and ETH and the addTokenMOONLiquidity() routine is used to add liquidity by providing the MOONand another ERC20 token.

```
    function addMOONETHLiquidity(uint256 nativeAmount) external payable nonReentrant {
        require(msg.value > 0, "!sufficient funds");

        ERC20(address(_moon)).safeTransferFrom(msg.sender, address(this), nativeAmount);

        // approve token transfer to cover all possible scenarios
        ERC20(address(_moon)).approve(address(mooonSwapRouter), nativeAmount);

        // add the liquidity
        mooonSwapRouter.addLiquidityETH{value: msg.value}(
            address(_moon),
            nativeAmount,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            msg.sender,
            block.timestamp
        );


```

```
        if (address(this).balance > 0) {
            // not going to require/check return value of this transfer as reverting
                behaviour is undesirable.
            payable(address(msg.sender)).call{value: address(this).balance}("");
        }

        uint256 mooonBalance = ERC20(address(_moon)).balanceOf(address(this));

        if (mooonBalance > 0)
            _moon.transfer(msg.sender, mooonBalance);
    }

    function addTokenMOONLiquidity(address baseTokenAddress, uint256 baseAmount, uint256
         nativeAmount) external nonReentrant {
        ERC20(baseTokenAddress).safeTransferFrom(msg.sender, address(this), baseAmount);
        ERC20(address(_moon)).safeTransferFrom(msg.sender, address(this), nativeAmount);

        // approve token transfer to cover all possible scenarios
        ERC20(baseTokenAddress).approve(address(mooonSwapRouter), baseAmount);
        _moon.approve(address(mooonSwapRouter), nativeAmount);

        // add the liquidity
        mooonSwapRouter.addLiquidity(
            baseTokenAddress,
            address(_moon),
            baseAmount,
            nativeAmount ,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            msg.sender,
            block.timestamp
        );


        if (ERC20(baseTokenAddress).balanceOf(address(this)) > 0)
            ERC20(baseTokenAddress).safeTransfer(msg.sender, ERC20(baseTokenAddress).
                balanceOf(address(this)));

        if (ERC20(address(_moon)).balanceOf(address(this)) > 0)
            ERC20(address(_moon)).transfer(msg.sender, ERC20(address(_moon)).balanceOf(
                address(this)));
    }
```

Listing 3.1: AddLiquidityHelper::addMOONETHLiquidity()/addTokenMOONLiquidity()

While examining the implementation logic of these two routines, We observe that there is no slippage control in place, which opens up the possibility for front-running and potentially results in a smaller LP amount. Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the liquidity provider. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above sandwich arbitrage to better protect the interests of users.

Status This issue has been confirmed.

3.2 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AddLiquidityHelper`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));
```

```
        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.2: `USDT Token Contract`

Because of that, a normal call to `approve()` with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to `approve()` twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we use the `AddLiquidityHelper::addTokenMOONLiquidity()` routine as an example. If the `USDT` token is supported as `baseTokenAddress`, the unsafe version of `ERC20(baseTokenAddress).approve(address(mooonSwapRouter), baseAmount)` may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
    function addTokenMOONLiquidity(address baseTokenAddress, uint256 baseAmount, uint256
         nativeAmount) external nonReentrant {
        ERC20(baseTokenAddress).safeTransferFrom(msg.sender, address(this), baseAmount);
        ERC20(address(_moon)).safeTransferFrom(msg.sender, address(this), nativeAmount);

        // approve token transfer to cover all possible scenarios
        ERC20(baseTokenAddress).approve(address(mooonSwapRouter), baseAmount);
        _moon.approve(address(mooonSwapRouter), nativeAmount);

        // add the liquidity
        mooonSwapRouter.addLiquidity(
            baseTokenAddress,
            address(_moon),
            baseAmount,
            nativeAmount ,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            msg.sender,
            block.timestamp
        );


        if (ERC20(baseTokenAddress).balanceOf(address(this)) > 0)
```

```
            ERC20(baseTokenAddress).safeTransfer(msg.sender, ERC20(baseTokenAddress).
                balanceOf(address(this)));

        if (ERC20(address(_moon)).balanceOf(address(this)) > 0)
            ERC20(address(_moon)).transfer(msg.sender, ERC20(address(_moon)).balanceOf(
                address(this)));
    }
```

Listing 3.3: `AddLiquidityHelper::addTokenMOONLiquidity()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`.

Status This issue has been fixed in the following commit: `7652a32`.

3.3 Incompatibility with Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AddLiquidityHelper`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.1, the `AddLiquidityHelper` contract provides two functions for users to add liquidity pair to the `Uniswap` pools, i.e., `addMOONETHLiquidity()` and `addTokenMOONLiquidity()`. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `AddLiquidityHelper` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the `LPHelper`'s internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contract. In the following, we show the `addTokenMOONLiquidity()` routine that is used to transfer `baseTokenAddress/_moon` to the `AddLiquidityHelper` contract.

```
    function addTokenMOONLiquidity(address baseTokenAddress, uint256 baseAmount, uint256
         nativeAmount) external nonReentrant {
        ERC20(baseTokenAddress).safeTransferFrom(msg.sender, address(this), baseAmount);
        ERC20(address(_moon)).safeTransferFrom(msg.sender, address(this), nativeAmount);

        // approve token transfer to cover all possible scenarios
        ERC20(baseTokenAddress).approve(address(mooonSwapRouter), baseAmount);
        _moon.approve(address(mooonSwapRouter), nativeAmount);

        // add the liquidity
        mooonSwapRouter.addLiquidity(
```

```
            baseTokenAddress,
            address(_moon),
            baseAmount,
            nativeAmount ,
            0, // slippage is unavoidable
            0, // slippage is unavoidable
            msg.sender,
            block.timestamp
        );


        if (ERC20(baseTokenAddress).balanceOf(address(this)) > 0)
            ERC20(baseTokenAddress).safeTransfer(msg.sender, ERC20(baseTokenAddress).
                balanceOf(address(this)));

        if (ERC20(address(_moon)).balanceOf(address(this)) > 0)
            ERC20(address(_moon)).transfer(msg.sender, ERC20(address(_moon)).balanceOf(
                address(this)));
    }
```

Listing 3.4: `AddLiquidityHelper::addTokenMOONLiquidity()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge a certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `addTokenMOONLiquidity()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of expecting the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation.

Another mitigation is to regulate the set of `baseTokenAddress` tokens that are permitted into the protocol for adding liquidity. Meanwhile, there exist certain assets that may exhibit control switches that can be dynamically exercised to convert into deflationary.

Recommendation If current codebase needs to support deflationary tokens, it is necessary to check the balance before and after the `transfer()`/`transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted `USDT`.

Status This issue has been fixed in the following commit: `46ff39d`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: XYZMoon
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In Save The Moon, there is a privileged account, i.e., owner. This account play a critical role in governing and regulating the system-wide operations (e.g., manage AMMpairs, exclude an account from reward or include an account to reward, exclude an account from fee charing or include an account to fee charing, enable or disable tax, set swapAndLiquifyEnabled, etc.). Our analysis shows that this privileged account need to be scrutinized. In the following, we use the XYZMoon contract as an example and show the representative functions potentially affected by the privileges of the owner account.

```
    function manageAmmPairs(address addr, bool isAdd) public onlyOwner{
        ammPairs[addr] = isAdd;
    }

    ...

    function excludeFromReward(address account) public onlyOwner {
        require(!_isExcluded[account], "Account is already excluded");
        if (_rOwned[account] > 0) {
            _tOwned[account] = tokenFromReflection(_rOwned[account]);
        }
        _isExcluded[account] = true;
        _excluded.push(account);
    }

    function includeInReward(address account) external onlyOwner {
        require(_isExcluded[account], "Account is already excluded");
        for (uint256 i = 0; i < _excluded.length; i++) {
            if (_excluded[i] == account) {
                _excluded[i] = _excluded[_excluded.length - 1];
                _tOwned[account] = 0;
                _isExcluded[account] = false;
                _excluded.pop();
                break;
            }
        }
    }

    ...
```

```
    function excludeFromFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = true;
    }

    function includeInFee(address account) public onlyOwner {
        _isExcludedFromFee[account] = false;
    }

    function enableOrDisableTax(bool isEnabled) external onlyOwner {
        if (isEnabled) {
            restoreAllFee();
        } else {
            removeAllFee();
        }
    }

    function setSwapAndLiquifyEnabled(bool _enabled) public onlyOwner {
        swapAndLiquifyEnabled = _enabled;
        emit SwapAndLiquifyEnabledUpdated(_enabled);
    }
```

Listing 3.5: Example Privileged Operations in `XYZMoon`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms they will consider using multi-sig for the privileged `owner` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Save The Moon` protocol. `Save the Moon` aims to invigorate the `LUNC` ecosystem by incessantly buying and burning the `LUNC` supply through the `MOON` tokens. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[4] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[5] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[7] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[10] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[11] PeckShield. PeckShield Inc. https://www.peckshield.com.