SMART CONTRACT AUDIT REPORT

for

MintStakeShare

Prepared By: Xiaomi Huang

PeckShield

October 22, 2024

Document Properties

Client	MSS
Title	Smart Contract Audit Report
Target	MSS
Version	1.0
Author	Xuxian Jiang
Auditors	Daisy Cao, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 22, 2024	Xuxian Jiang	Final Release
1.0-rc	October 1, 2024	Xuxian Jiang	Release Candidate

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 MSS 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 MSS

MSS is an innovative token distribution and staking protocol that is designed for fairness and deep liquidity. The staking feature pays 2% daily returns on compounding, and 1% daily return when collecting rewards. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Audited Contracts

Item	Description
Target	MSS
Website	https://docs.mintstakeshare.com/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	October 22, 2024

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

- https://github.com/MintStakeShare/mss-contracts.git (e46b72b)

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

- https://github.com/MintStakeShare/mss-contracts.git (651e98f)

1.2 About PeckShield

PeckShield Inc. [7] 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

Impact

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [6]:

- 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.

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

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

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) [5], 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 implementation of the `MSS` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	5	

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 high-severity vulnerability, 2 medium-severity vulnerabilities, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key MSS Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Revisited Staking Reward Accounting Logic in Staking	Business Logic	Resolved
PVE-002	Medium	Improper Liquidity-Adding Logic in MintStakeShareExpansion	Business Logic	Resolved
PVE-003	Low	Revisited Token Rescue Logic in MintStakeShareExpansion	Security Features	Resolved
PVE-004	Informational	Incorrect Slippage Control in Swapper::_swap()	Business Logic	Resolved
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

Beside the identified issues, we emphasize that for any user-facing applications and services, 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 should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Revisited Staking Reward Accounting Logic in Staking

- ID: PVE-001
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: Staking
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The MSS protocol has a core Staking contract that features the payment of 2% daily returns on compounding, and 1% daily return in reward collection. In the process of examining its logic to calculate user rewards, we notice the calculation should be improved.

To elaborate, we show below the related _collect() routine that is designed to collect user rewards. It comes to our attention that the reward collection also reduces the user staked amount, which should not be the case.

```
    function _collect(address _user) internal {
        UserInfo storage user = userInfo[_user];
        uint256 _pending = _calculatePending(user.amount, user.lastReward);
        if (_pending > user.amount) {
            _pending = user.amount;
        }
        if (_pending > 0) {
            uint collectAmount = (_pending * collectClaimPercent) / 1000;
            _sendRewards(_user, collectAmount);
            user.totalCollected += (collectAmount);
            gameInfo.totalCollected += (collectAmount);
            gameInfo.totalRewards += (collectAmount);

            // CHANGE HERE
            user.amount -= collectAmount;

            emit Collect(_user, collectAmount);
        }
```

```
        user.lastReward = block.timestamp;
    }
```

Listing 3.1: `Staking::_collect()`

Moreover, the reward calculation in other routines `_addToStake()` and `_compound()` also need to be adjusted to timely update the user's `lastReward`. By doing so, we can avoid double crediting the user rewards.

Recommendation Strengthen the above-mentioned routines to properly account for user rewards.

Status This issue has been fixed in the following commit: `651e98f`.

3.2 Improper Liquidity-Adding Logic in MintStakeShareExpansion

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MintStakeShareExpansion`
- Category: Business Logic [4]
- CWE subcategory: N/A

Description

The `MSS` protocol has a core `MintStakeShareExpansion` contract for the token distribution. It also has the built-in logic to add liquidity to the `AMM`-based liquidity pair. While examining the logic to add liquidity, we notice current implementation should be improved.

In the following, we show the implementation of the related routine, i.e., `_mintAndAddLiquidityWithETH()`. It has a rather straightforward logic in adding the intended `Ether` as well as the paired token amount to the liquidity. However, it has an implicit assumption that the pair's `token0` is always `WETH`. This implicit assumption does not hold and should be explicitly validated and adjusted, if necessary.

```
    function _mintAndAddLiquidityWithETH(
        uint256 ethAmount
    ) private returns (uint256 liquidity) {
        require(address(this).balance >= ethAmount, "Insufficient ETH balance");
        IWETH(WETH).deposit{value: ethAmount}();

        (uint res0, uint res1, ) = IUniswapV2Pair(lpPair).getReserves();
        uint tokens;
        if (res0 > 0 && res1 > 0) {
            tokens = _getLiquidityAmount(ethAmount, res0, res1);
        } else {
```

```
            tokens = getTokenMintAmount(ethAmount);
        }

        assert(IWETH(WETH).transfer(lpPair, ethAmount));

        _mint(lpPair, tokens);
        liquidity = IUniswapV2Pair(lpPair).mint(address(0xdead));
        emit AddLiquidity(tokens, ethAmount, liquidity);
    }
```

Listing 3.2: MintStakeShareExpansion::_mintAndAddLiquidityWithETH()

Recommendation Revise the above routine to ensure the liquidity is properly added.

Status This issue has been fixed in the following commit: 651e98f.

3.3 Revisited Token Rescue Logic in MintStakeShareExpansion

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MintStakeShareExpansion
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the MSS protocol, there is a privileged function transferForeignToken() with the purpose of rescuing funds that are accidently sent to the protocol contracts. In the process of examining the token-rescue logic, we notice current implementation can be improved.

```
    function transferForeignToken(
        address _token,
        address _to
    ) external onlyOwner returns (bool _sent) {
        require(_token != address(0), "_token address cannot be 0");
        uint256 _contractBalance = IERC20(_token).balanceOf(address(this));
        _sent = IERC20(_token).transfer(_to, _contractBalance);
    }
```

Listing 3.3: MintStakeShareExpansion::transferForeignToken()

To elaborate, we show above the implementation of the related routine. It comes to our attention that current implementation does not thoroughly validate the given input token, which should be considered as a foreign token. In other words, it also needs to validate against address(this) so that the user funds will not be transferred out.

Recommendation Revise the above routine to ensure the given input token is indeed a foreign token being rescued.

Status This issue has been fixed in the following commit: `651e98f`.

3.4 Incorrect Slippage Control in Swapper::_swap()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Swapper`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The `MSS` protocol has a core `Swapper` contract that is designed to facilitate user buy/sell operations. While reviewing the logic to perform the actual swap operation, we notice current implementation is flawed.

In the following, we show the implementation of the related `_swap()` routine. It has a rather straightforward logic in swapping the input token to the output token. However, when the input token is not equal to `WETH`, the input token will be converted to `WETH` first. This conversion has the slippage control in place to avoid unintended `MEV` sandwiching. However, the slippage control parameter mistakenly uses the one for the output token (line 1466).

```
1440     function _swap(
1441         address tokenOut,
1442         uint256 tokenAmountOutMin,
1443         address tokenIn,
1444         uint256 tokenInAmount,
1445         address _to
1446     ) internal {
1447         uint256 wethAmount;
1448
1449         if (tokenIn == WETH) {
1450             wethAmount = tokenInAmount;
1451         } else {
1452             IUniswapV2Pair pair = IUniswapV2Pair(uniswapV2Pair);
1453             bool isInputA = pair.token0() == tokenIn;
1454             require(
1455                 isInputA   pair.token1() == tokenIn,
1456                 "Input token not present in input pair"
1457             );
1458             address[] memory path;
1459
1460             path = new address[](2);
1461             path[0] = tokenIn;
```

```
            path[1] = WETH;
            uniswapV2Router
                .swapExactTokensForTokensSupportingFeeOnTransferTokens(
                    tokenInAmount,
                    tokenAmountOutMin,
                    path,
                    _to,
                    block.timestamp
                );
            wethAmount = IERC20(WETH).balanceOf(address(this));
        }

        if (tokenOut != WETH) {
            address[] memory basePath;

            basePath = new address[](2);
            basePath[0] = WETH;
            basePath[1] = tokenOut;

            uniswapV2Router
                .swapExactTokensForTokensSupportingFeeOnTransferTokens(
                    wethAmount,
                    tokenAmountOutMin,
                    basePath,
                    _to,
                    block.timestamp
                );
        }
    }
```

Listing 3.4: `Swapper::_swap()`

Recommendation Correct the above routine to properly make use of the slippage control parameter.

Status This issue has been resolved as the logic ensures that `tokenIn` and `tokenOut` will not be `WETH` at the same time.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the `MSS` protocol, there is a privileged account (`owner`) that plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters, manage AMM pairs, and recover stuck funds). In the following, we show the representative functions potentially affected by the privilege of this account.

```
    function setAutomatedMarketMakerPair(
        address pair,
        bool value
    ) external onlyOwner {
        require(
            pair != lpPair,
            "The pair cannot be removed from automatedMarketMakerPairs"
        );

        _setAutomatedMarketMakerPair(pair, value);
    }
    ...
    function setInitialPrice(uint256 _initialPrice) external onlyOwner {
        require(
            currentPriceTier() == 0,
            "Initial price can only be set in first price tier"
        );
        initialPrice = ((1e18 * 1e18) / (_initialPrice));
    }
    ...
    function enableTrading() external onlyOwner {
        require(initialPrice > 0, "Initial price must be set");
        require(!tradingActive, "Cannot reenable trading");
        tradingActive = true;
        swapEnabled = true;
        tradingActiveBlock = block.number;

        uint ethBalance = address(this).balance;
        _mintAndAddLiquidityWithETH(ethBalance);

        emit EnabledTrading();
    }
```

Listing 3.5: Example Privileged Operations in `MintStakeShareExpansion`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the privileged account is not governed by a DAO-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive vault parameters, which directly undermines the assumption of the vault design.

In the meantime, the staking contract makes use of the proxy contract to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged account to the intended DAO-like governance contract. All changed 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 The issue has been confirmed and will be mitigated with the use of a multi-sig to manage the privileged account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `MSS` protocol, which is an innovative token distribution and staking protocol that is designed for fairness and deep liquidity. The staking feature pays `2%` daily returns on compounding, and `1%` daily return when collecting rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

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

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

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

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

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