SMART CONTRACT AUDIT REPORT

for

Mushrooms

Prepared By: Yiqun Chen

PeckShield

July 5, 2021

Document Properties

Client	Mushrooms Finance
Title	Smart Contract Audit Report
Target	Mushrooms
Version	1.0
Author	Yiqun Chen
Auditors	Xuxian Jiang, Yiqun Chen
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 5, 2021	Yiqun Chen	Final Release
1.0-rc	July 4, 2021	Yiqun Chen	Release Candidate

Contact

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

Name	Yiqun Chen
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 `Mushrooms` 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 Mushrooms

`Mushrooms` is a yield-farming boosted portfolio tool, which innovatively combines the traits of both DeFi yield-aggregator and effective trading algorithms. Users provide either stablecoin or exposure tokens, and the algorithm of the protocol will keep these two assets remaining equal value. Once the price of the exposure token increases, it will sell to secure some profit in stablecoin, and if the price of the exposure token decreases, it will buy it with the stablecoin. At the same time, all the balanced assets are used to yield-farming for maximum the profits.

The basic information of the `Mushrooms` protocol is as follows:

Table 1.1: Basic Information of The `Mushrooms` Protocol

Item	Description
Name	Mushrooms Finance
Website	https://mushrooms.finance/
Type	BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 5, 2021

In the following, we show the audited contract code deployed at the `BSC` chain with the following addresses:

- https://bscscan.com/address/0xB86eace0Ce0d3f463B415e8B3463e331F1d95b6e#code
- https://bscscan.com/address/0xb06661A221Ab2Ec615531f9632D6Dc5D2984179A#code

And this is the final revised contract code after all fixes have been checked in :

- https://bscscan.com/address/0xaec70f921de4870894ae95c91a4525160402881c#code

1.2 About PeckShield

PeckShield Inc. [9] 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 [8]:

- 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) [7], 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 `Mushrooms` implementation. 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	2	■■
Low	2	■■
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 2 medium-severity vulnerabilities and 2 low-severity vulnerabilities.

Table 2.1: Key Mushrooms Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Input Validation of deposit()	Coding Practices	Fixed
PVE-002	Low	Mushmon Token Idiosyncrasies	Coding Practices	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Medium	Proper Minimum Balance Enforcement	Coding Practices	Fixed

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 Improved Input Validation of deposit()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MushMons
- Category: Coding Practices [6]
- CWE subcategory: CWE-391 [4]

Description

In the Mushrooms protocol, the user is allowed to deposit either stablecoin (e.g., USDC) or exposure token (e.g., BTC). For example, the user deposits 1000 USDC, then the protocol will swap half value of the deposit, which is 500 USDC for BTC, to keep the value of USDC and BTC owned by the user equal. In the following, we list below the related deposit() function.

```
function deposit(uint256 _amount, address _exposure, uint256 _exposureAmount) public nonReentrant {

    require(_validExposureTokens[_exposure], '!invalidExposureToken');

    require(yieldVaults[msg.sender][address(token)] != address(0), '!invalidYieldVault');
    require(yieldVaults[msg.sender][_exposure] != address(0), '!invalidYieldVaultExposure');

    uint256 shares = _amount;
    TriggerResult memory tr;

    uint256 markPrice = getMarkPrice(_exposure);
    uint256 _amt;
    if (_amount > 0){
        _amt = _calculateDeposit(address(token), _amount);
        // record the principal for later rebalancing
        tokenBalances[msg.sender][_exposure] = tokenBalances[msg.sender][_exposure].add(_amt);
        tr = TriggerResult(true, true, _amt.div(2), markPrice);
```

```
        } else{
            _amt = _calculateDeposit(address(_exposure), _exposureAmount);
            exposureBalances[msg.sender][_exposure] = exposureBalances[msg.sender][_exposure].add(_amt);
            shares = _convertExposure2Stablecoin(_exposure, _amt, markPrice);
            tr = TriggerResult(true, false, _amt.div(2), markPrice);
        }

        // mint share according to stalecoin value
        _mint(msg.sender, shares);

        // rebalancing with 50-50 style in stablecoin and _exposure token
        uint256 delta = _parseTriggerResultAndSwap(msg.sender, _exposure, tr);

        // ensure the minted share (denominated stablecoin value) in acceptable range
        uint256 maxBal = exposureMaxBalances[_exposure];
        if (maxBal > 0){
            require(balanceOf(msg.sender) <= maxBal, '!maxAllowedBalance');
        }

        // deposit in yield farming
        if (tr.increase){
            _yieldFarmingDeposit(msg.sender, address(token), yieldVaults[msg.sender][address(token)], tr.amount);
            _yieldFarmingDeposit(msg.sender, _exposure, yieldVaults[msg.sender][_exposure], delta);
        } else{
            _yieldFarmingDeposit(msg.sender, address(token), yieldVaults[msg.sender][address(token)], delta);
            _yieldFarmingDeposit(msg.sender, _exposure, yieldVaults[msg.sender][_exposure], tr.amount);
        }

        emit Deposit(msg.sender, _exposure, _amount, _exposureAmount, markPrice, shares);
    }
```

Listing 3.1: `MushMons::deposit()`

As shown in the above implementation, the user can deposit only one kind of tokens at a time, and if the user tries to deposit both of them, then only stablecoin will be transferred in. Although there is no loss here, it brings unnecessary confusion to the user. With that, we suggest to check whether the user tries to deposit both of them or not. Instead of transferring the stablecoin in by default in this case, it's way more better to notify the user depositing only one kind of tokens a time.

Recommendation Revise the above `deposit()` routine to better validate the user input to ensure only one kind of tokens can be accepted.

Status This issue has been fixed by adding the statement of `require(_amount == 0 || _exposureAmount == 0, '!onlyOneAsset')`.

3.2 Mushmon Token Idiosyncrasies

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MushMons
- Category: Coding Practices [6]
- CWE subcategory: CWE-1041 [2]

Description

Similar to many other yield-farming protocols, the Mushrooms protocol gathers all funds of investors together to perform yield-farming and bring additional gains to investors. However, the difference is it will mint the amount of share tokens for the user based on the stablecoin value this user provides. The difference on the calculation of the share for the user makes its share token different as well. In the following, we list below the related deposit() and amountOfTokenForShare() functions.

```
    function deposit(uint256 _amount, address _exposure, uint256 _exposureAmount) public nonReentrant {

        require(_validExposureTokens[_exposure], '!invalidExposureToken');

        require(yieldVaults[msg.sender][address(token)] != address(0), '!invalidYieldVault');
        require(yieldVaults[msg.sender][_exposure] != address(0), '!invalidYieldVaultExposure');

        uint256 shares = _amount;
        TriggerResult memory tr;

        uint256 markPrice = getMarkPrice(_exposure);
        uint256 _amt;
        if (_amount > 0){
            _amt = _calculateDeposit(address(token), _amount);
            // record the principal for later rebalancing
            tokenBalances[msg.sender][_exposure] = tokenBalances[msg.sender][_exposure].add(_amt);
            tr = TriggerResult(true, true, _amt.div(2), markPrice);
        } else{
            _amt = _calculateDeposit(address(_exposure), _exposureAmount);
            exposureBalances[msg.sender][_exposure] = exposureBalances[msg.sender][_exposure].add(_amt);
            shares = _convertExposure2Stablecoin(_exposure, _amt, markPrice);
            tr = TriggerResult(true, false, _amt.div(2), markPrice);
        }

        // mint share according to stalecoin value
        _mint(msg.sender, shares);
```

```
        // rebalancing with 50-50 style in stablecoin and _exposure token
        uint256 delta = _parseTriggerResultAndSwap(msg.sender, _exposure, tr);

        // ensure the minted share (denominated stablecoin value) in acceptable range
        uint256 maxBal = exposureMaxBalances[_exposure];
        if (maxBal > 0){
            require(balanceOf(msg.sender) <= maxBal, '!maxAllowedBalance');
        }

        // deposit in yield farming
        if (tr.increase){
            _yieldFarmingDeposit(msg.sender, address(token), yieldVaults[msg.sender][address(token)], tr.amount);
            _yieldFarmingDeposit(msg.sender, _exposure, yieldVaults[msg.sender][_exposure], delta);
        } else{
            _yieldFarmingDeposit(msg.sender, address(token), yieldVaults[msg.sender][address(token)], delta);
            _yieldFarmingDeposit(msg.sender, _exposure, yieldVaults[msg.sender][_exposure], tr.amount);
        }

        emit Deposit(msg.sender, _exposure, _amount, _exposureAmount, markPrice, shares);
    }
```

Listing 3.2: `MushMons::deposit()`

Our analysis shows that regardless of the previous yields, the share is calculated based on the stablecoin value this user provides. When the user deposits the exposure token, i.e., `BTC`, the protocol will calculate how much the `BTC` worth to the stablecoin according to the updated price of the `BTC`, and mint tokens based on this value.

The problem comes when the user tries to transfer his/her share tokens to others, no actual value transfer from the sender's `yieldVault` to the recipient's. If the recipient tries to withdraw with the new share, he/she would get the same amount of tokens as before.

```
    function amountOfTokenForShare(address _user, address _token, uint256 _share) public view returns (uint256) {
        address _tokenVault = yieldVaults[_user][_token];

        require(_tokenVault != address(0), "!invalidVault");
        require(_share <= balanceOf(_user), "!invalidShare");

        uint256 _yieldShare = yieldVaultShares[_user][_tokenVault];
        _yieldShare = _yieldShare.mul(_share).div(balanceOf(_user));
        return _convertToToken(_tokenVault, _yieldShare);
    }
```

Listing 3.3: `MushMons::amountOfTokenForShare()`

Specifically, the above `amountOfTokenForShare()` function will be called during the withdraw, and it is used for calculating the amount of tokens owned by the user in the `yieldVault`. In fact, no matter how large the `balanceOf(_user)` is, the `_yieldShare` (line 1316) of the user won't change, which means the transfer of the share tokens between users does not actually carry any weight.

Recommendation Disable the `transfer()` and the `transferFrom()` functions.

Status This issue has been fixed by disabling the internal transfer functionality.

3.3 Trust Issue Of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `MushMons`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `Mushrooms` protocol, there is a special administrative account, i.e., `governance`. This `governance` account plays a critical role in governing and regulating the protocol-wide operations (e.g., setting various parameters). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged `governance` account and one of its related privileged accesses in current contract.

To elaborate, we show below the `setUniRoute()` routine. This routine sets the address of the pool used to swap between stablecoin and exposure tokens for the user. And the `governance` account has the privilege to change it. As users transfer their funds to the contract first for swapping, the untrusted pool address may be risky.

```
function setUniRoute(address _route) external {
    require(msg.sender == governance, "!governance");

    IERC20(token).safeApprove(swapDEX, 0);
    IERC20(token).safeApprove(_route, uint256(-1));

    for (uint i = 0;i < _exposureTokens.length;i++){
        IERC20(_exposureTokens[i]).safeApprove(swapDEX, 0);
        IERC20(_exposureTokens[i]).safeApprove(_route, uint256(-1));
    }

    swapDEX = _route;
```

```
    }
```

Listing 3.4: `MushMons::setUniRoute()`

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 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 This issue has been confirmed.

3.4 Proper Minimum Balance Enforcement

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MushMons`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1042 [1]

Description

In the `MushMons` contract, there is a `withdraw()` function that allows users to withdraw their funds from their `yieldVaults`. The protocol calculates the assets owned by the user and the amount of each asset the user requires to withdraw according to the input (`_shares`). Also, the protocol allows users to choose the type of the assets they want to withdraw. In the following, we list below the `withdraw()` function.

```
    function withdraw(uint256 _shares, address _exposure, uint256 _withdrawType) public
        nonReentrant {
        uint256 _balShare = balanceOf(msg.sender);

        require(_validExposureTokens[_exposure], '!invalidExposureToken');
        require(_balShare >= _shares, '!invalidWithdrawShare');

        uint256 _stablecoinAmt = amountOfTokenForShare(msg.sender, address(token),
            _shares);
        uint256 _exposureAmt = amountOfTokenForShare(msg.sender, _exposure, _shares);

        // burn share
        _burn(msg.sender, _shares);
```

```
        // ensure the remaining share (denominated stablecoin value) in acceptable range
        uint256 minBal = exposureMinBalances[_exposure];
        if (minBal > 0){
            require(_balShare >= minBal || _balShare == 0, '!minAllowedBalance');
        }

        // get required asset for withdrawal from yield farming if necessary
        if (_stablecoinAmt > 0) {
            uint256 _bal = token.balanceOf(address(this));
            if (_stablecoinAmt > _bal){
                require(yieldVaults[msg.sender][address(token)] != address(0), '!
                    invalidYieldVault');
                _yieldFarmingWithdraw(msg.sender, address(token), yieldVaults[msg.sender
                    ][address(token)], _stablecoinAmt.sub(_bal));
            }
        }

        if (_exposureAmt > 0) {
            uint256 _bal = IERC20(_exposure).balanceOf(address(this));
            if (_exposureAmt > _bal){
                require(yieldVaults[msg.sender][_exposure] != address(0), '!
                    invalidYieldVaultExposure');
                _yieldFarmingWithdraw(msg.sender, _exposure, yieldVaults[msg.sender][
                    _exposure], _exposureAmt.sub(_bal));
            }
        }

        uint256 markPrice = getMarkPrice(_exposure);
        if (_withdrawType == 0){
            uint256 delta = _swapInDex(_exposure, address(token), _exposureAmt,
                markPrice, true);
            _stablecoinAmt = _stablecoinAmt.add(delta);
            _exposureAmt = 0;
        } else if (_withdrawType == 1){
            uint256 delta = _swapInDex(address(token), _exposure, _stablecoinAmt,
                markPrice, false);
            _exposureAmt = _exposureAmt.add(delta);
            _stablecoinAmt = 0;
        }

        // update balances
        tokenBalances[msg.sender][_exposure] = tokenBalances[msg.sender][_exposure].mul(
            _balShare.sub(_shares)).div(_balShare);
        exposureBalances[msg.sender][_exposure] = exposureBalances[msg.sender][_exposure
            ].mul(_balShare.sub(_shares)).div(_balShare);
        // final withdrawal to user
        if (_stablecoinAmt > 0) {
            uint256 _tBal = token.balanceOf(address(this));
            token.safeTransfer(msg.sender, _stablecoinAmt > _tBal? _tBal :
                _stablecoinAmt);
        }
```

```
121         if (_exposureAmt > 0) {
122             uint256 _eBal = IERC20(_exposure).balanceOf(address(this));
123             IERC20(_exposure).safeTransfer(msg.sender, _exposureAmt > _eBal? _eBal :
                    _exposureAmt);
124         }

126         emit Withdraw(msg.sender, _exposure, _stablecoinAmt, _exposureAmt, markPrice,
                _shares);
127     }
```

Listing 3.5: `MushMons::withdraw()`

The withdraw logic requires calculating the assets owned by the user, burning the share of the user, updating the balances of the user, and transferring tokens back to the user.[1] It comes to our attention that the current implementation does not properly honor the minimum balance requirement (line 82). The actual balance needs to be retrieved for minimum balance enforcement, instead of using the current stale balance (`_balShare`).

Recommendation Properly enforce the minimal balance requirement in the above `withdraw()` function.

Status This issue has been fixed as suggested.

[1] An earlier version of this `withdraw()` function does not properly decrease the balance of the user when the user withdraws his/her funds. To the team's credit, the team discovers and independently fixes this issue.

4 Conclusion

In this audit, we have analyzed the `Mushrooms` protocol design and implementation. The `Mushrooms` protocol is a yield-farming boosted portfolio tool, which innovatively combines the traits of both DeFi yield-aggregator and effective trading algorithms. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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-1041: Updated Issues. https://cwe.mitre.org/data/definitions/1042.html.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

[4] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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

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

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

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

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