SMART CONTRACT AUDIT REPORT

for

MasterStaker

Prepared By: Yiqun Chen

Hangzhou, China

November 22, 2021

Document Properties

Client	Nearth
Title	Smart Contract Audit Report
Target	MasterStaker
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 22, 2021	Shulin Bie	Final Release
1.0-rc	November 1, 2021	Shulin Bie	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 `MasterStaker` contract, 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 MasterStaker

`MasterStaker` improves on the widely-used `MasterChef` contract by introducing a unified support of using both `ERC20` and `ERC721` tokens to stake for rewards. With that, the user can make the same contract to stake supported assets to the pool to earn rewards. It enriches the DeFi market and presents a unique contribution to current DeFi ecosystem.

The basic information of `MasterStaker` is as follows:

Table 1.1: Basic Information of MasterStaker

Item	Description
Target	MasterStaker
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 22, 2021

In the following, we show the `MD5` hash value of the compressed file used in this audit:

- MD5 (MasterStaker-20211030.sol) = a06a4c1257681cd2a6fd8d2010c3bb13

And here is the final `MD5` hash value of the compressed file after all fixes for the issues found in the audit have been checked in:

- MD5 (MasterStaker-20211122.sol) = 12993fc7fde554e2f4e1e56119d3153a

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

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

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

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) [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 `MasterStaker` 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 logic, 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	1	■
Undetermined	0	
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, the smart contract `MasterStaker` is 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key MasterStaker Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools During Pool Weight Changes	Business Logic	Fixed
PVE-002	Low	Duplicate Pool Detection and Prevention	Business Logic	Fixed
PVE-003	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-005	Informational	Recommended Explicit Pool Validity Checks	Security Features	Fixed

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 Timely massUpdatePools During Pool Weight Changes

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

Description

The MasterStaker contract implements an incentive mechanism that rewards the staking of supported assets with the rewardToken token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of tokens in the reward pool.

The reward pools can be dynamically added via add() and the weights of supported pools can be adjusted via set(). When analyzing the pool weight update routine set(), we notice the need of timely invoking massUpdatePools() to update the reward distribution before the new pool weight becomes effective.

```
    function set(
        uint256 _pid,
        uint256 _allocPoint,
        bool _withUpdate
    ) external onlyOwner {
        if (_withUpdate) {
            massUpdatePools();
        }
        totalAllocPoint = totalAllocPoint - poolInfo[_pid].allocPoint + _allocPoint;
        poolInfo[_pid].allocPoint = _allocPoint;
        emit Set(_pid, _allocPoint, _withUpdate);
    }
```

Listing 3.1: MasterStaker::set()

If the call to massUpdatePools() is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the onlyOwner modifier), which greatly alleviates the concern.

Recommendation Timely invoke massUpdatePools() when any pool's weight has been updated. In fact, the third parameter (_withUpdate) to the set() routine can be simply ignored or removed.

Status This issue has been addressed by the team in the file: 12993fc7fde554e2f4e1e56119d3153a.

3.2 Duplicate Pool Detection and Prevention

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: MasterStaker
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

The MasterStaker contract provides an incentive mechanism that rewards the staking of supported assets with the rewardToken token. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*multiplier/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure or moderated by a privileged account). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in add(), whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool with a duplicate token from being added. Though it is a privileged interface (protected with the modifier onlyOwner), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(
        PoolType _poolType,
```

```
        uint256 _allocPoint,
        address _token,
        bool _withUpdate
    ) external onlyOwner {
        require(_poolType == PoolType.IERC20  _poolType == PoolType.IERC721,
                "add: Unsupported PoolType");
        if (_withUpdate) {
            massUpdatePools();
        }
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                poolType: _poolType,
                token: _token,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accRewardPerShare: 0
            })
        );
        emit Add(_poolType, _allocPoint, _token, _withUpdate);
    }
```

Listing 3.2: `MasterStaker::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(address _token) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[pid].token != _token, "add: existing pool?");
        }
    }

    function add(
        PoolType _poolType,
        uint256 _allocPoint,
        address _token,
        bool _withUpdate
    ) external onlyOwner {
        require(_poolType == PoolType.IERC20  _poolType == PoolType.IERC721,
                "add: Unsupported PoolType");
        if (_withUpdate) {
            massUpdatePools();
        }
        checkPoolDuplicate(_token);
        uint256 lastRewardBlock =
            block.number > startBlock ? block.number : startBlock;
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
```

```
            PoolInfo({
                poolType: _poolType,
                token: _token,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accRewardPerShare: 0
            })
        );
        emit Add(_poolType, _allocPoint, _token, _withUpdate);
    }
```

Listing 3.3: Revised `MasterChef::add()`

We point out that if a new pool with a duplicate staking token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status This issue has been addressed by the team in the file: 12993fc7fde554e2f4e1e56119d3153a.

3.3 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MasterStaker`
- Category: Business Logic [4]
- CWE subcategory: CWE-841 [2]

Description

By design, the `MasterStaker` contract is the main entry for interaction with users. In particular, one entry routine, i.e., `depositERC20()`, accepts user deposits of the supported `token` assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `MasterStaker` contract. These asset-transferring routines will work well under the assumption that the vault's internal asset balances (specified by the `user.amount`) are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
    function depositERC20(
        uint256 _pid,
        uint256 _amount
    ) external nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        PoolType poolType = pool.poolType;
        require (poolType == PoolType.IERC20, "depositERC20: Wrong pool");
        IERC20 token = IERC20(pool.token);
        UserInfo storage user = userInfo[_pid][msg.sender];
```

```
        ...
        token.safeTransferFrom(
            address(msg.sender),
            address(this),
            _amount
        );
        user.amount = user.amount + _amount;
        user.rewardDebt = user.amount * pool.accRewardPerShare / 1e12;
        emit DepositERC20(msg.sender, _pid, _amount);
    }
```

Listing 3.4: MasterStaker::depositERC20()

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 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 depositERC20(), may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts. Apparently, these balance inconsistencies are damaging to accurate and precise portfolio management of MasterStaker and affects protocol-wide operation and maintenance.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the MasterStaker before and after the transfer() or transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into MasterStaker. In MasterStaker, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better 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 The issue has been confirmed by the team.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: MasterStaker
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

In the MasterStaker contract, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the owner account.

```
    function setMigrator(
        IMigratorChef _migrator
    ) external onlyOwner {
        migrator = _migrator;
    }

    // Migrate lp token to another lp contract. Can be called by anyone. We trust that
        migrator contract is good.
    function migrate(
        uint256 _pid
    ) external {
        require(address(migrator) != address(0), "migrate: No migrator");
        PoolInfo storage pool = poolInfo[_pid];
        PoolType poolType = pool.poolType;
        if (poolType == PoolType.IERC20) {
            IERC20 token = IERC20(pool.token);
            uint256 bal = token.balanceOf(address(this));
            token.safeApprove(address(migrator), bal);
            IERC20 newtoken = IERC20(migrator.migrate(token));
            require(bal == newtoken.balanceOf(address(this)), "migrate: Bad");
            pool.token = address(newtoken);
        } else if (poolType == PoolType.IERC721) {
            revert("Not implemented yet for IERC721");
        }
    }
```

Listing 3.5: MasterStaker::setMigrator()&&migrate()

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

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 by the team.

3.5 Recommended Explicit Pool Validity Checks

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: MasterStaker
- Category: Security Features [3]
- CWE subcategory: CWE-287 [1]

Description

The MasterStaker contract implements a reward mechanism, which has been tasked with the reward distribution to various pools and stakers. In the following, we show the key PoolInfo data structure. Note all added pools are maintained in an array poolInfo.

```
    enum PoolType {
        IERC20,
        IERC721
    }
    // Info of each pool.
    struct PoolInfo {
        PoolType poolType; // The type of the token as PoolType
        address token; // Address of LP token contract.
        uint256 allocPoint; // How many allocation points assigned to this pool. rewards to distribute per block.
        uint256 lastRewardBlock; // Last block number that rewards distribution occurs.
        uint256 accRewardPerShare; // Accumulated rewards per share, times 1e12. See below.
    }
    // The Reward Token!
    IERC20 public rewardToken;
    // Block number when bonus reward period ends.
    uint256 public bonusEndBlock;
    // Reward tokens created per block.
    uint256 public rewardPerBlock;
    // The migrator contract. It has a lot of power. Can only be set through governance (owner).
    IMigratorChef public migrator;
    // Info of each pool.
```

```
    PoolInfo[] public poolInfo;
```

Listing 3.6: `MasterStaker`

When there is a need to add a new pool, set a new `allocPoint` for an existing pool, stake (by depositing the supported assets), unstake (by redeeming previously deposited assets), query pending rewards, there is a constant need to perform sanity checks on the pool validity. The current implementation simply relies on the implicit, compiler-generated bound-checks of arrays to ensure the pool index stays within the array range `[0, poolInfo.length-1]`. However, considering the importance of validating given pools and their numerous occasions, a better alternative is to make explicit the sanity checks by introducing a new modifier, say `validatePool`. This new modifier essentially ensures the given `_pool_id` or `_pid` indeed points to a valid, live pool, and additionally give semantically meaningful information when it is not.

```
    function depositERC20(
        uint256 _pid,
        uint256 _amount
    ) external nonReentrant {
        PoolInfo storage pool = poolInfo[_pid];
        PoolType poolType = pool.poolType;
        require (poolType == PoolType.IERC20, "depositERC20: Wrong pool");
        IERC20 token = IERC20(pool.token);
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);
        ...
    }
```

Listing 3.7: `MasterStaker::depositERC20()`

We highlight that there are a number of functions that can be benefited from the new pool-validating modifier, including `set()`, `migrate()`, `pendingRewards()`, `updatePool()`, `depositERC20()`, `depositERC721()`, `harvest()`, `withdrawERC20()`, `withdrawERC721()`, and `emergencyWithdraw()`.

Recommendation Apply necessary sanity checks to ensure the given `_pid` is legitimate. Accordingly, a new modifier `validatePool` can be developed and appended to each function in the above list.

```
    modifier validatePool(uint256 _pid) {
        require(_pid < poolInfo.length, "chef: pool exists?");
        _;
    }

    function depositERC20(
        uint256 _pid,
        uint256 _amount
    ) external nonReentrant validatePool(_pid) {
        PoolInfo storage pool = poolInfo[_pid];
        PoolType poolType = pool.poolType;
```

```
        require (poolType == PoolType.IERC20 , "depositERC20: Wrong pool");
        IERC20 token = IERC20(pool.token);
        UserInfo storage user = userInfo[_pid][msg.sender];

        updatePool(_pid);
        ...
    }
```

Listing 3.8: `MasterStaker::depositERC20()`

Status This issue has been addressed by the team in the file: 12993fc7fde554e2f4e1e56119d3153a.

4 Conclusion

In this audit, we have analyzed the `MasterStaker` design and implementation. `MasterStaker` improves on the widely-used `MasterChef` contract by introducing `ERC721` token staking, which allows the holders of the supported staking `ERC20/ERC721` token to deposit their assets to the pool to earn rewards. It enriches the DeFi market and presents a unique contribution to current DeFi ecosystem. 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-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.