SMART CONTRACT AUDIT REPORT

for

FarmerLand v2

Prepared By: Xiaomi Huang

PeckShield

February 28, 2023

Document Properties

Client	FarmerLand
Title	Smart Contract Audit Report
Target	FarmerLand v2
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 28, 2023	Xuxian Jiang	Final Release
1.0-rc	February 23, 2023	Luck Hu	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 FarmerLand v2 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 FarmerLand v2

FarmerLand is a farm-themed idle P2E NFT game and DeFi protocol. It supports an auction system like that of Avarice, which is combined with an NFT minting, training, and yield boosting system similar to that of the original DeFi Kingdoms. NFTs can be upgraded in the Toolshed and staked in the Stable, where they independently earn USDC and WHEAT, or the Field, where they boost auction and WHEAT staking income. FarmerLand v2 features the addition of a deflationary NFT burn mechanic whereby Minotaur NFTs cost farmer, tool, or land NFTs to mint. The basic information of the FarmerLand v2 protocol is as follows:

Table 1.1: Basic Information of The FarmerLand v2 Protocol

Item	Description
Issuer	FarmerLand
Website	https://FarmerLand.gg/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 28, 2023

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

- https://github.com/LithiumSwapTech/farmerland-contracts.git (95aa80f)

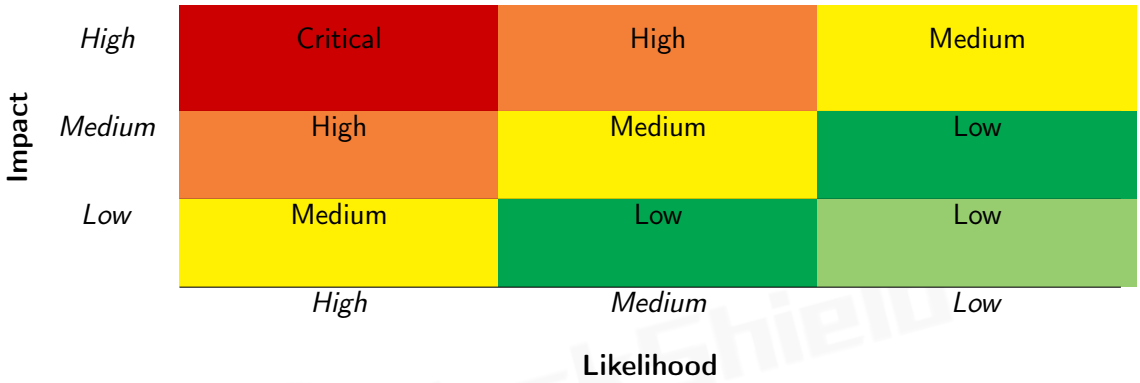
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/LithiumSwapTech/farmerland-contracts.git (0142b09)

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



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.

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.

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

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 `FarmerLand v2` 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 FarmerLand v2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Precision in calcStakeCollecting()	Coding Practices	Fixed
PVE-002	Low	Improved lastUSDCDistroTimestamp State Initialization	Business Logic	Fixed
PVE-003	Medium	Revisited Calculation of WHEAT Amount in clcTokenValue()	Business Logic	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Precision in calcStakeCollecting()

- ID: PVE-001
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `WHEAT`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `FarmerLand` protocol provides an incentive mechanism that rewards the staking of the `WHEAT` tokens with USDC dividends and `WHEAT` bonus. The USDC dividends are pooled from the lobby entry and distributed to stakers per their shares of the staked `WHEAT` tokens. While examining the calculation of the USDC dividends amount for one stake, we notice the precision can be improved to prevent possible profit loss for small stakes.

To elaborate, we show below the code snippet of the `calcStakeCollecting()` routine. As the name indicates, it is used to calculate the amount of USDC dividends for the give stake. It calculates the amount of USDC dividends for each day since the start day of the stake, and accumulates them to derive the total USDC dividends. The amount of the USDC dividends for each day is calculated via `(dayUSDCPool[_day] * _stakeValue)/ totalTokensInActiveStake[_day]` (line 755), which means all the pooled USDC dividends for each day are distributed per users stake shares. However, we notice there is no precision protection for the calculation. As a result, a small stake may get far less or no USDC dividends than expectation. Based on this, it is suggested to add proper precision protection for the calculation.

```
748     function calcStakeCollecting(address _address, uint _stakeId) public view returns (
            uint) {
749         uint userDivs;
750         uint _endDay = mapMemberStake[_address][_stakeId].endDay;
751         uint _startDay = mapMemberStake[_address][_stakeId].startDay;
752         uint _stakeValue = mapMemberStake[_address][_stakeId].tokenValue;
```

```
753
754         for (uint _day = _startDay; _day < _endDay && _day < currentDay; _day++) {
755             userDivs += (dayUSDCPool[_day] * _stakeValue) / totalTokensInActiveStake[
                    _day];
756         }
757
758         return (userDivs - mapMemberStake[_address][_stakeId].loansReturnAmount);
759     }
```

Listing 3.1: WHEAT::calcStakeCollecting()

Recommendation Revisit the calcStakeCollecting() routine to add proper precision protection for the calculation of the USDC dividends. A revision example which uses the precision of $1e6$ can be found below:

```
748     function calcStakeCollecting(address _address, uint _stakeId) public view returns (
            uint) {
749         uint userDivs;
750         uint _endDay = mapMemberStake[_address][_stakeId].endDay;
751         uint _startDay = mapMemberStake[_address][_stakeId].startDay;
752         uint _stakeValue = mapMemberStake[_address][_stakeId].tokenValue;
753
754         for (uint _day = _startDay; _day < _endDay && _day < currentDay; _day++) {
755             userDivs += (dayUSDCPool[_day] * _stakeValue * 1e6) /
                    totalTokensInActiveStake[_day];
756         }
757         userDivs /= 1e6;
758
759         return (userDivs - mapMemberStake[_address][_stakeId].loansReturnAmount);
760     }
```

Listing 3.2: Revised WHEAT::calcStakeCollecting()

Status The issue has been fixed by this commit: 0142b09.

3.2 Improved lastUSDCDistroTimestamp State Initialization

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: MasterChef
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the FarmerLand protocol, all FarmerLand NFTs can be deposited in the Stable to earn rewards (i.e., WHEAT and USDC). The rewards are received from the WHEAT contract and distributed to users per

their deposit shares in the pool. While reviewing the drip of the rewards in the `MasterChef` contract, we notice the `lastUSDCDistroTimestamp` is not properly initialized at the first deposit.

To elaborate, we show below the code snippets of the `updatePool()/getUSDCDrip()` routines. As the name indicates, the `getUSDCDrip()` routine is used to drip USDC rewards to the pool. Specifically, the USDC rewards to be dripped are calculated per the time elapse since the last drip (line 411), which means if there is no time elapse, there is no reward available to drip. However, it comes to our attention that, the last drip time, i.e., `lastUSDCDistroTimestamp`, is initialized to `type(uint).max` (line 401). So there is no time elapse when the `getUSDCDrip()` routine is called for the first time, and the USDC reward can be available from the second time the `getUSDCDrip()` is triggered.

Note that the `getUSDCDrip()` routine can only be called from the `updatePool()` routine and only when there are NFTs locked in the pool (lines 146-147). As a result, the USDC reward can only be accumulated from the second time the `updatePool()` routine is triggered, and the first deposit has no reward accumulated until then. Based on this, it is suggested to initialize the `lastUSDCDistroTimestamp` to `block.timestamp` at the first deposit, so the USDC reward can be accumulated from the first deposit.

```
133
134   function updatePool() public {
135     PoolInfo storage pool = poolInfo;
136     if (block.timestamp <= pool.lastRewardTimestamp)
137         return;
138
139     uint lpSupply = pool.totalLocked;
140     if (lpSupply == 0) {
141         pool.lastRewardTimestamp = block.timestamp;
142         return;
143     }
144
145     // WHEAT pool is always pool 0.
146     if (poolInfo.totalLocked > 0) {
147         uint usdcRelease = getUSDCDrip();
148
149         if (usdcRelease > 0) {
150             accDepositUSDCRewardPerShare = accDepositUSDCRewardPerShare + ((usdcRelease
                    * 1e24) / poolInfo.totalLocked);
151             totalUSDCCollected = totalUSDCCollected + usdcRelease;
152         }
153         ...
154     }
155
156     pool.lastRewardTimestamp = block.timestamp;
157   }
```

Listing 3.3: `MasterChef::updatePool()`

```
401   uint public lastUSDCDistroTimestamp = type(uint).max;
402   uint public lastWHEATDistroTimestamp = type(uint).max;
403
```

```
404   function getUSDCDrip() internal returns (uint) {
405     uint usdcBalance = usdcCurrency.balanceOf(address(this));
406     if (promisedUSDC > usdcBalance)
407         return 0;
408
409     uint usdcAvailable = usdcBalance - promisedUSDC;
410     // only provide a drip if there has been some seconds passed since the last drip
411     uint blockSinceLastDistro = block.timestamp > lastUSDCDistroTimestamp ? block.
            timestamp - lastUSDCDistroTimestamp : 0;
412
413     // We distribute the usdc assuming the old usdc balance wanted to be distributed
            over usdcDistributionTimeFrameSeconds seconds.
414     uint usdcRelease = (blockSinceLastDistro * usdcAvailable) /
            usdcDistributionTimeFrameSeconds;
415
416     usdcRelease = usdcRelease > usdcAvailable ? usdcAvailable : usdcRelease;
417
418     lastUSDCDistroTimestamp = block.timestamp;
419     promisedUSDC += usdcRelease;
420
421     return usdcRelease;
422   }
```

Listing 3.4: `MasterChef::getUSDCDrip()`

Note the same issue is also applicable to the `lastWHEATDistroTimestamp` state variable, which shall also be properly initialized at the first deposit.

Recommendation Revisit the `updatePool()` routine to properly initialize the `lastUSDCDistroTimestamp` /`lastWHEATDistroTimestamp` to `block.timestamp` at the first deposit.

Status The issue has been fixed by this commit: `0142b09`.

3.3 Revisited Calculation of Token Amount in clcTokenValue()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `WHEAT`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `FarmerLand` protocol, there is a daily auction lobby in which users can bid USDC for their share of the daily pool of the `WHEAT` tokens. The number of `WHEAT` tokens available in the auction lobby decreases by 1% each day. While reviewing the calculation of users share of the `WHEAT` tokens in the lobby pool, we notice it uses the lobby pool of current day, not the target day.

To elaborate, we show below the code snippet of the `clcTokenValue()` routine, which is used to calculate users share of the `WHEAT` tokens in the lobby pool of the target day. It comes to our attention that, the number of `WHEAT` tokens the user can share is calculated per `lastLobbyPool` (line 620), which is the available `WHEAT` tokens in the auction lobby of current day, i.e., `currentDay`, while not the target day, i.e., `_day`. As a result, user cannot receive the expected number of `WHEAT` tokens. Based on this, it is suggested to properly record the available `WHEAT` tokens in the auction lobby for each day, and use the recorded value of the target day to calculate users share of `WHEAT` tokens.

```
614  function clcTokenValue(address _address, uint _day) public view returns (uint) {
615      require(_day != 0, "lobby disabled on day 0!");
616      uint _tokenValue;
617      uint entryDay = mapMemberLobby[_address][_day].entryDay;
618
619      if (entryDay != 0 && entryDay < currentDay) {
620          _tokenValue = (lastLobbyPool * mapMemberLobby[_address][_day].entryAmount) /
                 lobbyEntry[entryDay];
621      } else {
622          _tokenValue = 0;
623      }
624
625      return _tokenValue;
626  }
```

Listing 3.5: WHEAT::clcTokenValue()

Recommendation Properly calculate users share of the `WHEAT` tokens based on the available `WHEAT` tokens in the auction lobby of the target day.

Status The issue has been fixed by this commit: `0142b09`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `FarmerLand` protocol, there are certain privileged accounts, i.e., `owner/admins`, that play critical roles in governing and regulating the system-wide operations (e.g., admins can mint NFTs freely, owner can flush the lottery pool). Our analysis shows that the privileged accounts need to be scrutinized. In the following, we use the `WHEAT` contract as an example and show the representative functions potentially affected by the privileges of the `owner/admins` accounts.

Specifically, the privileged functions in WHEAT allow for the owner to set the nftMasterChefAddress which is used to receive rewards for the emission of the NFT staking in the Stable, set the daoAddress which is the team wallets address to receive the DAO share of the lobby, set the lastLobbyPool which is the available WHEAT tokens in the auction lobby of current day, flush the lottery reward pool which is used to reward the lottery winner, etc.

```
85      function set_nftMasterChefAddress(address _nftMasterChefAddress) external onlyOwner
            () {
86          require(_nftMasterChefAddress != address(0), "!0");

88          address oldNftMasterChefAddress = nftMasterChefAddress;

90          nftMasterChefAddress = _nftMasterChefAddress;

92          emit  MCAddressSet(oldNftMasterChefAddress, _nftMasterChefAddress);
93      }

95      /* change team wallets address % */
96      function changeDaoAddress(address _daoAddress) external onlyOwner() {
97          require(_daoAddress != address(0), "!0");

99          address oldDaoAddress = daoAddress;

101         daoAddress = _daoAddress;

103         emit  DaoAddressSet(oldDaoAddress, _daoAddress);
104     }

106     function set_lastLobbyPool(uint _lastLobbyPool) external onlyOwner() {
107         uint oldLastLobbyPool = lastLobbyPool;

109         lastLobbyPool = _lastLobbyPool;

111         emit LastLobbySet(oldLastLobbyPool, _lastLobbyPool);
112     }

114     function flushLottyPool() external onlyOwner() nonReentrant {
115         if (lottery_Pool > 0) {
116             uint256 amount = lottery_Pool;
117             lottery_Pool = 0;
118             token_USDC.transfer(daoAddress, amount);

120             emit LottyPoolFlushed(daoAddress, amount);
121         }
122     }
```

Listing 3.6: SMARTDeFi::constructor()

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner/admins may also be a counter-party risk to the protocol users. It is worrisome if the privileged owner/admins accounts are plain EOA accounts. 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 accounts 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 project team.

4 | Conclusion

In this audit, we have analyzed the FarmerLand v2 protocol design and implementation. FarmerLand is a farm-themed idle P2E NFT game and DeFi protocol. It supports an auction system like that of Avarice, which is combined with an NFT minting, training, and yield boosting system similar to that of the original DeFi Kingdoms. NFTs can be upgraded in the Toolshed and staked in the Stable, where they independently earn USDC and WHEAT, or the Field, where they boost auction and WHEAT staking income. FarmerLand v2 features the addition of a deflationary NFT burn mechanic whereby Minotaur NFTs cost farmer, tool, or land NFTs to mint. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed or 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-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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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