SMART CONTRACT AUDIT REPORT

for

Magpie Protocol

Prepared By: Xiaomi Huang

PeckShield

August 16, 2022

Document Properties

Client	Magpie
Title	Smart Contract Audit Report
Target	Magpie Protocol
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 16, 2022	Luck Hu	Final Release
1.0-rc1	August 8, 2022	Luck Hu	Release Candidate #1

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the Magpie 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 Magpie Protocol

Magpie is an innovative yield-boosting protocol that provides users with boosted stablecoin yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. Conversely, WOM holders can also benefit from Magpie by converting their WOM token into mWOM (Magpie WOM) to earn a share of Magpie's profit. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Magpie Protocol

Item	Description
Issuer	Magpie
Website	https://www.magpies.xyz/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 16, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note that the audit scope only covers the following contracts: MasterMagpie.sol, WombatStaking.sol, WombatPoolHelper.sol, BaseRewarder.sol, mWom.sol, locker.sol.

- https://github.com/magpiexyz/magpie_contracts.git (4ada7e9)

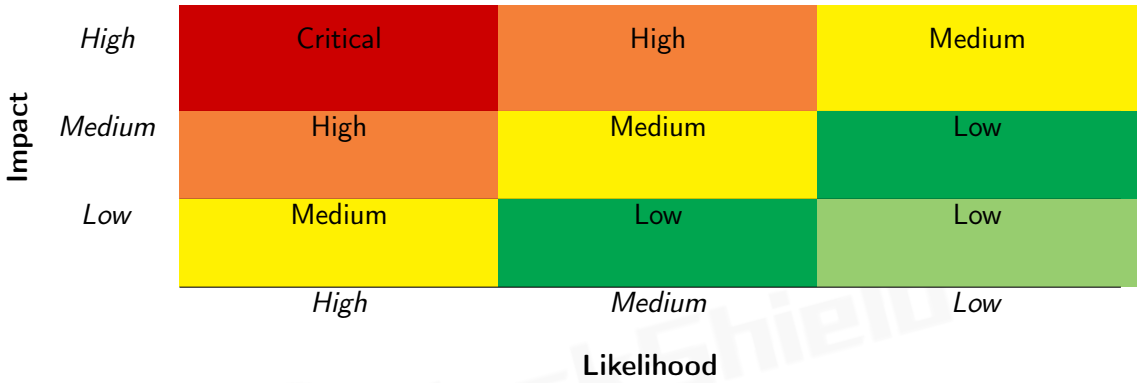
And here is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/magpiexyz/magpie_contracts.git (23ba731)

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 `Magpie` protocol 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	1	■
Medium	2	■■
Low	1	■
Informational	0	
Total	4	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 2 medium-severity vulnerabilities and 1 low-severity vulnerability.

Table 2.1: Key Magpie Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Logic To Distribute Caller Fee	Business Logic	Fixed
PVE-002	High	Incorrect Token Flow in withdraw()	Business Logic	Fixed
PVE-003	Low	Improved Sanity Checks For Function Parameters	Coding Practices	Fixed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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

3 Detailed Results

3.1 Revisited Logic To Calculate The Rewards

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `WombatStaking`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `WombatStaking` contract interacts with the `Wombat Exchange` to provide services, such as adding new liquidity, staking `LP` token on `MasterWombat`, and staking `WOM` to get `veWom`. It provides an external `harvest()` interface for users or `WombatPoolHelper` to claim rewards from `MasterWombat`.

To elaborate, we show below the code snippet of the `harvest()` routine. By design, it is implemented to claim rewards from the specified pool in `MasterWombat` and send the rewards to the rewarder. Specially, if the caller is not the `Magpie` contracts, the caller can get the caller fee in `mWom`. The original fee in `WOM` is locked to get `veWom` via the `convertWOM()` routine (line 347). After that, the routine transfers `mWom` to the caller (line 348). However, currently the contract may not hold `mWom` at all, which reverts the harvest operation. Our analysis shows that the `WOM` rewards will be converted to the `mWom` by invoking `Uint fee = IMWom(mWom).convert(feeAmount)` and the received `mWom` amount (`fee`) may be not equal to the `feeAmount` as the `veWom` may not be converted from `WOM` with the 1:1 conversion rate. As a result, it needs to transfer the `fee` amount of `mWom` to the caller: `IERC20(mWom).safeTransfer(msg.sender, fee)`.

```
329     /// @notice harvest a Pool from MGP
330     /// @param _depositToken the address of the deposit token Pool to harvest
331     /// @param _isUser true if this function is not called by the magpie Contracts. The
            caller gets the caller fee
332     function harvest(
333         address _depositToken,
334         bool _isUser
335     ) _onlyActivePool(_depositToken) external {
```

```
336         Pool storage poolInfo = pools[_depositToken];
337         uint256 beforeBalance = IERC20(wom).balanceOf(address(this));
338         uint256[] memory pids = new uint256[](1);
339         pids[0] = poolInfo.pid;
340         IMasterWombat(masterWombat).multiClaim(pids); //only claim from a specific
                deposit token Pool
341
342         uint256 rewards = IERC20(wom).balanceOf(address(this)) - beforeBalance;
343         uint256 afterFee = rewards;
344         if (_isUser) {
345             uint256 feeAmount = (rewards * CALLER_FEE) / FEE_DENOMINATOR;
346             IERC20(wom).approve(mWom, feeAmount);
347             this.convertWOM(feeAmount);
348             IERC20(mWom).safeTransfer(msg.sender, feeAmount);
349             afterFee = afterFee - feeAmount;
350         }
351         sendRewards(poolInfo.depositToken, poolInfo.rewarder, rewards, afterFee);
352
353         emit WomHarvested(rewards, rewards - afterFee);
354     }
355
356     /// @notice convert WOM to mWOM
357     /// @param _amount the number of WOM to convert
358     /// @dev the WOM must already be in the contract
359     function convertWOM(uint256 _amount) external returns(uint256) {
360         uint256 veWomMintedAmount = 0;
361         if (_amount > 0) {
362             IERC20(wom).approve(veWom, _amount);
363             veWomMintedAmount = IVeWom(veWom).mint(_amount, IVeWom(veWom).maxLockDays())
                    ;
364         }
365
366         emit WomConverted(_amount, IVeWom(veWom).maxLockDays());
367
368         return veWomMintedAmount;
369     }
```

Listing 3.1: `WombatStaking::harvest()`

Recommendation Revisit the above logic to properly convert the `WOM` to `mWom` and transfer the received amount of `mWom` to the caller.

Status The issue has been fixed by this commit: `4c0677d`.

3.2 Incorrect Token Flow in withdraw()

- ID: PVE-002
- Severity: High
- Likelihood: Medium
- Impact: Medium
- Target: `WombatPoolHelper, WombatStaking`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `WombatPoolHelper` is a helper smart contract to interact with `WombatStaking` (to provide liquidity on `Wombat Exchange`, etc.) and `MasterMagpie` (to stake `LP` token as a certificate and farm). While examining the token flow to withdraw assets from `WombatPoolHelper`, we notice the existence of incorrect token flow between `WombatPoolHelper` and `WombatStaking` which needs to be corrected.

To elaborate, we show below the code snippets of the `WombatPoolHelper:withdraw()` and `WombatStaking:withdraw()` routines. As implemented, the `WombatPoolHelper:withdraw()` takes the input `_amount` of `receiptToken` and unstakes it from `MasterMagpie` (line 137). The unstaked `receiptToken` `_amount` is then taken by invoking the `WombatStaking:withdraw()` (line 140). However, in the `WombatStaking:withdraw()` implementation, the input `_amount` is expected to be the amount of stables (`_depositToken`) which will be provided to `Wombat Exchange` as liquidity. Our analysis shows that, the `WombatPoolHelper:withdraw()` could be refactored to take the input `_amount` as the amount of the stables to withdraw which is the same as the `WombatPoolHelper:deposit()` where the input `_amount` represents the amount of the stables to deposit. With this, the `WombatPoolHelper:withdraw()` shall convert the input `_amount` of the stables to the amount of `receiptToken` which is needed to unstake from the `MasterMagpie`.

```
136     function withdraw(uint256 _amount, uint256 _minAmount) external override _harvest {
137         _unstake(_amount, msg.sender);
138         IWombatStaking(wombatStaking).withdraw(
139             depositToken,
140             _amount,
141             _minAmount,
142             msg.sender
143         );
144         emit NewWithdraw(msg.sender, _amount);
145     }
```

Listing 3.2: WombatPoolHelper:withdraw()

```
294     function withdraw(
295         address _depositToken,
296         uint256 _amount,
297         uint256 _minAmount,
298         address _sender
```

```
299     ) _onlyPoolHelper(_depositToken) external {
300         // _amount is the amount of stable
301         Pool storage poolInfo = pools[_depositToken];
302         uint256 sharesAmount = getSharesForDepositTokens(_amount, _depositToken);
303         uint256 lpAmount = getLPTokensForShares(sharesAmount, _depositToken);
304         IMintableERC20(poolInfo.receiptToken).burn(msg.sender, sharesAmount);
305 
306         IERC20(poolInfo.lpAddress).approve(poolInfo.depositTarget, lpAmount);
307         IMasterWombat(masterWombat).withdraw(poolInfo.pid, lpAmount);
308 
309         uint256 beforeWithdraw = IERC20(_depositToken).balanceOf(address(this));
310         IWombatPool(poolInfo.depositTarget).withdraw(
311             _depositToken,
312             lpAmount,
313             _minAmount,
314             address(this),
315             block.timestamp
316         );
317 
318         poolInfo.size -= _amount;
319         poolInfo.sizeLp -= lpAmount;
320 
321         IERC20(_depositToken).safeTransfer(
322             _sender,
323             IERC20(_depositToken).balanceOf(address(this)) - beforeWithdraw
324         );
325 
326         emit NewWithdraw(_sender, _depositToken, _amount);
327     }
```

Listing 3.3: WombatStaking:withdraw()

Recommendation Revised the `WombatPoolHelper:withdraw()` to correct the token flow during assets withdrawal.

Status The issue has been fixed by this commit: `0b024dd`.

3.3 Improved Sanity Checks For Function Parameters

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `MasterMagpie`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `MasterMagpie` contract is a `MasterChef` implementation in `Magpie`, which provides an incentive mechanism that rewards the staking of supported assets with `MGP`. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And stake holders are rewarded in proportional to their share of deposited token 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 update routine `set()`, we notice the need of properly validating the input pool before the new pool weight becomes effective.

To elaborate, we show below the code snippet of the `set()` routine. As the name indicates, it is used by the pool manager to update the weight, rewarder and locker for the pool given by the `_stakingToken`. At the beginning of the routine, it validates the input `_rewarder` and `_locker`. However, it does not validate the input pool. As a result, if the pool is an invalid one, this routine could successfully update the `totalAllocPoint` which will create an unfair reward distribution to stake holders.

```
615     function set(
616         address _stakingToken,
617         uint256 _allocPoint,
618         address _rewarder,
619         address _locker,
620         bool _overwrite
621     ) external _onlyPoolManager {
622         if (!Address.isContract(address(_rewarder)) && address(_rewarder) != address(0))
623             revert MustBeContractOrZero();
624 
625         if (!Address.isContract(address(_locker)) && address(_locker) != address(0))
626             revert MustBeContractOrZero();
627 
628         massUpdatePools();
629         totalAllocPoint =
630             totalAllocPoint -
631             tokenToPoolInfo[_stakingToken].allocPoint +
632             _allocPoint;
633 
634         tokenToPoolInfo[_stakingToken].allocPoint = _allocPoint;
635 
```

```
636         if (_overwrite) {
637             tokenToPoolInfo[_stakingToken].rewarder = _rewarder;
638             tokenToPoolInfo[_stakingToken].locker = _locker;
639         }
640
641         emit Set(
642             _stakingToken,
643             _allocPoint,
644             IBaseRewardPool(tokenToPoolInfo[_stakingToken].rewarder),
645             tokenToPoolInfo[_stakingToken].locker,
646             _overwrite
647         );
648     }
```

Listing 3.4: `MasterMagpie::set()`

Note it shares the same issue in the `updatePool()` routine which shall update only valid pools.

Recommendation Revisit the above mentioned routines to add proper sanity checks.

Status The issue has been fixed by this commit: `ce1715c`.

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 `Magpie` protocol, there is a privileged account, i.e., `owner`, that plays a critical role in governing and regulating the system-wide operations (e.g., set pool managers). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `MasterMagpie` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

Specifically, the privileged functions in `MasterMagpie` allow for the `owner` to set pool manager who can add/set the reward pools, toggle whether emergency withdraw is allowed, update the emission rate, etc.

```
95      function setPoolManagerStatus(address _account, bool _allowedManager)
96          external
97          onlyOwner
98      {
99          PoolManagers[_account] = _allowedManager;
```

```
100
101         emit PoolManagerStatus(_account, PoolManagers[_account]);
102     }
103
104     /// @notice update the status of emergencyWithdraw.
105     /// @notice WARNING : the contract should not be used after that action for anything
            other that withdrawing
106     function allowEmergency(bool _allow) external onlyOwner {
107         emergencyAllowed = _allow;
108
109         emit EmergencyUpdated(emergencyAllowed);
110     }
111
112     function updateEmissionRate(uint256 _mgpPerSec) public onlyOwner {
113         massUpdatePools();
114         uint256 oldEmissionRate = mgpPerSec;
115         mgpPerSec = _mgpPerSec;
116
117         emit UpdateEmissionRate(msg.sender, oldEmissionRate, mgpPerSec);
118     }
```

Listing 3.5: Example Privileged Operations in the `MasterMagpie` Contract

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

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All 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 mitigated as the team confirms they plan to use multi-sig for all admin roles.

4 Conclusion

In this audit, we have analyzed the Magpie design and implementation. Magpie is an innovative yield-boosting protocol that provides users with boosted stablecoin yields from the innovative stableswap platform – Wombat Exchange, without even having to hold the WOM token. Conversely, WOM holders can also benefit from Magpie by converting their WOM token into mWOM to earn a share of Magpie's profit. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

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

[3] MITRE. CWE-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.