SMART CONTRACT AUDIT REPORT

for

Project Chosen

Prepared By: Xiaomi Huang

PeckShield

May 5, 2022

Document Properties

Client	Project Chosen
Title	Smart Contract Audit Report
Target	Chosen
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2022	Xuxian Jiang	Final Release
1.0-rc	April 30, 2022	Xuxian Jiang	Release Candidate

Contact

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

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

Contents

1 Introduction

Given the opportunity to review the `Project Chosen` design document and related smart contract source code, 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 audited protocol 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 Project Chosen

`Project Chosen` allows users to choose the projects they are optimistic about, and then predict whether the price of this project will rise or fall compared with the IDO price on the seventh day after `TGE`. Then, the protocol admin can enter it according to the objective price on the seventh day after `TGE`, and then settle the reward for the successful prediction and generate the winner medal `NFT`. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Chosen

Item	Description
Name	Project Chosen
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2022

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

- https://github.com/Project-Chosen/chosen-smart-contract.git (3dcdeac)

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

- https://github.com/Project-Chosen/chosen-smart-contract.git (4e7e29c)

1.2 About PeckShield

PeckShield Inc. [10] 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 [9]:

- 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) [8], 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 `Project Chosen` 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	3	■ ■ ■
Low	2	■ ■
Informational	1	■
Total	6	

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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation,

Table 2.1: Key Chosen Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Logic in VoucherNft::queryUserAllNft()	Business Logic	Resolved
PVE-002	Medium	Revisited Logic in IgoOrePool::adminSafeTransfer()	Business Logic	Resolved
PVE-003	Medium	Potential Lock of User Stakes	Numeric Errors	Resolved
PVE-004	Informational	Simplified Logic in receiveRewards()	Business Logic	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Resolved

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 Improved Logic in VoucherNft::queryUserAllNft()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: VoucherNft
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The Project Chosen protocol will generate a winner medal NFT to the user who successfully predicts the token price movement after the token's TGE. While examining this medal NFT logic, we notice the current implementation can be improved.

To elaborate, we show below the related queryUserAllNft() function. It is a getter routine that is designed to return the information about the given list of NFTs. However, the current implementation does not properly initialize the return array and fails to return the queried information.

```
    function queryUserAllNft(uint256[] memory _ids)
        public
        view
        returns (NftInfo[] memory nftArray)
    {
        for (uint256 i = 0; i < _ids.length; i++) {
            nftArray[nftArray.length] = nftInfo[_ids[i]];
        }
        return nftArray;
    }
```

Listing 3.1: VoucherNft::queryUserAllNft()

Recommendation Improve the above queryUserAllNft() routine to properly return the queried NFT information.

Status This issue has been resolved as the team clarifies that `nftArray` does not need to be specially initialized: If `nftArray` is not filled with matching data, it will just return empty.

3.2 Revisited Logic in IgoOrePool::adminSafeTransfer()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `IgoOrePool`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

The `Project Chosen` protocol has a built-in incentive mechanism that encourages participating users to stake the configured `_mortgageLp` for rewards. The incentive mechanism is mainly implemented in the `IgoOrePool` contract and our analysis shows the contract also contains a privileged function that needs to be revisited.

To elaborate, we show below this privileged function `adminSafeTransfer()`. As the name indicates, this function facilitates the administrative transfer of the funds in the current incentive pool. However, it allows the current operator to withdraw the staked funds from protocol users. This design needs to be revisited so that the staked funds will not be jeopardized. In other words, there is a need to ensure the given `_token` can not be the staked token `_mortgageLp` with the following requirement: `require(_token != _mortgageLp)`.

```
    function adminSafeTransfer(
        address _token,
        address _to,
        uint256 _amount
    ) public onlyOperator {
        _upgradeSafeTransfer(_token, _to, _amount);
    }

    function _upgradeSafeTransfer(
        address _token,
        address _to,
        uint256 _amount
    ) internal {
        require(_token != address(0), "Token can not be 0x0!");
        require(_amount > 0, "Transfer limit cannot be 0!");

        uint256 balance = IERC20(_token).balanceOf(address(this));

        if (_amount > balance) {
            require(
                _amount.sub(balance) < uint256(1).mul(1e18),
```

```
                "Insufficient contract balance!"
            );
            IERC20(_token).safeTransfer(_to, balance);
        } else {
            IERC20(_token).safeTransfer(_to, _amount);
        }
    }
```

Listing 3.2: `IgoOrePool::adminSafeTransfer()`

Recommendation Revise the above `adminSafeTransfer()` function so that the user stakes cannot be administratively transferred without their permission.

Status This issue has been fixed in the commit: e4ba664.

3.3 Potential Lock of User Stakes

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `IgoOrePool`
- Category: Numeric Errors [7]
- CWE subcategory: CWE-190 [1]

Description

As mentioned earlier, the `Project Chosen` protocol shares an incentivizer mechanism that is inspired from `Synthetix`. In this section, we focus on a routine, i.e., `rewardPerToken()`, which is responsible for calculating the reward rate for each staked token. And it is part of the `updateReward()` modifier that would be invoked up-front for almost every public function in `IgoOrePool` to update and use the latest reward rate.

The reason is due to the known potential underflow pitfall when the `endTime` parameter is inappropriately configured. In particular, as the `rewardPerToken()` routine involves the multiplication of three `uint256` integers and the first integer depends on the `lastTimeRewardApplicable().sub(lastUpdateTime)` (lines $114-115$). While the `endTime` parameter is configured to be smaller than `lastUpdateTime`, it may result in an undesirable underflow, which effectively reverts the `rewardPerToken()` execution!

```
104     function lastTimeRewardApplicable() public view returns (uint256) {
105         return Math.min(block.timestamp, endTime);
106     }
107
108     function rewardPerToken() public view returns (uint256) {
109         if (totalPower == 0) {
110             return intervalReward;
111         }
```

```
        return
            intervalReward.add(
                lastTimeRewardApplicable()
                    .sub(lastUpdateTime)
                    .mul(miningOutput)
                    .mul(1e18)
                    .div(totalPower)
            );
    }
```

Listing 3.3: `IgoOrePool::rewardPerToken()`

```
    function updateStartTime(uint256 _time) public onlyOperator {
        startTime = _time;
    }

    function updateEndTime(uint256 _time) public onlyOperator {
        endTime = _time;
    }

    function updateMiningOutput(uint256 _yield) public onlyOperator {
        intervalReward = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        miningOutput = _yield;
    }
```

Listing 3.4: `IgoOrePool::updateStartTime()/updateEndTime()/updateMiningOutput()`

The underflow may in essence lock all deposited funds! Note that an authentication check on the caller of `onlyOperator()` greatly alleviates such concern. Currently, only the `operator` address is able to call `updateEndTime()` and this address can be set when the contract is deployed. Apparently, if the `operator` is a normal address, it may put users' funds at risk. To mitigate this issue, it is necessary to have the ownership under the governance control and ensure the `endTime` parameter will not be configured to underflow and lock users' funds.

Recommendation Mitigate the potential underflow risk in the `IgoOrePool` pool.

Status This issue has been fixed in the commit: e4ba664.

3.4 Simplified Logic in receiveRewards()

- ID: PVE-004
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `IgoOrePool`
- Category: Business Logic [6]
- CWE subcategory: CWE-770 [3]

Description

As mentioned earlier, the `Project Chosen` protocol shares an incentivizer mechanism inspired from `Synthetix`, which has the `receiveRewards()` routine to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `receiveRewards()` routine has a modifier, i.e., `updateReward(msg.sender))`, which timely updates the given user's (earned) rewards in `rewards[_account]` (line 219).

```
218     function receiveRewards() public updateReward(msg.sender) {
219         uint256 reward = earned(msg.sender);
220         if (reward > 0) {
221             rewards[msg.sender] = 0;
222             IERC20(_profitLp).transfer(msg.sender, reward);
223         } else {
224             reward = 0;
225         }
226         emit ReceiveRewards(msg.sender, reward);
227     }
```

Listing 3.5: `IgoOrePool::receiveRewards()`

```
    modifier updateReward(address _account) {
        intervalReward = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (_account != address(0)) {
            rewards[_account] = earned(_account);
            userLastIntervalReward[_account] = intervalReward;
        }
        _;
    }
```

Listing 3.6: `IgoOrePool::updateReward()`

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the given user. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 219).

Recommendation Avoid the duplicated calculation of the caller's reward in `receiveRewards()`, which also leads to (small) beneficial reduction of associated gas cost.

```
218     function receiveRewards() public updateReward(msg.sender) {
219         uint256 reward = rewards[msg.sender];
220         if (reward > 0) {
221             rewards[msg.sender] = 0;
222             IERC20(_profitLp).transfer(msg.sender, reward);
223         } else {
224             reward = 0;
```

```
        }
        emit ReceiveRewards(msg.sender, reward);
    }
```

Listing 3.7: Revised `IgoOrePool::receiveRewards()`

Status This issue has been fixed in the commit: e4ba664.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TokenTransferProxy`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [4]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.8: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of `approve()/transferFrom()` as well, i.e., `safeApprove()/safeTransferFrom()`.

In current implementation, if we examine the `IgoOrePool::withdraw()` routine that is designed to transfer the funds back to the staking user. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (lines 210 and 213).

```
179     function withdraw(uint256 _amount) public updateReward(msg.sender) {
180         require(_amount > 0, "Redemption quantity cannot be zero!");
181         require(
182             _amount <= power[msg.sender],
183             "The redemption amount cannot exceed the mortgage amount!"
184         );

186         totalPower = totalPower.sub(_amount);
187         power[msg.sender] = power[msg.sender].sub(_amount);

189         for (uint256 i = 0; i < associationPool.length; i++) {
190             if (
191                 power[msg.sender] <
192                 IFundraising(associationPool[i]).getThreshold()
193             ) {
194                 uint8 rank = IFundraising(associationPool[i]).isWhiteList(
195                     msg.sender
196                 );
197                 if (rank == 1) {
198                     IFundraising(associationPool[i]).setWhiteList(
199                         msg.sender,
200                         3
201                     );
202                 }
203             }
204         }

206         uint256 fee = 0;
207         if (block.timestamp < (lastStakedTime[msg.sender] + punishTime)) {
208             fee = _amount.mul(feeRatio).div(100);
209             _amount = _amount.sub(fee);
210             IERC20(_mortgageLp).transfer(_feeAddress, fee);
211         }

213         IERC20(_mortgageLp).transfer(msg.sender, _amount);

215         emit Withdraw(msg.sender, _amount, fee);
216     }
```

Listing 3.9: `IgoOrePool::withdraw()`

In the meantime, we also suggest to use the safe-version of `transferFrom()` in another related routine, namely `IgoOrePool::stake()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status This issue has been fixed in the commit: e4ba664.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the Project Chosen protocol, there are special administrative accounts, i.e., owner and operator. These accounts play a critical role in governing and regulating the protocol-wide operations (e.g., configure investment/yield amounts and set various thresholds). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```
    function updateInvestLevel(InvestLevel[3] memory _investLevelList)
        external
        onlyOperator
    {
        for (uint8 i = 0; i < 3; i++) {
            investLevelList[i] = _investLevelList[i];
        }
    }

    //
    function updateSupAdmin(address _account, bool isAdd)
        external
        onlySuperAdmin
    {
        if (isAdd) {
            superAdminNum++;
            addAuth(_account);
            IHelper(helperAddr).addAuth(_account);
        } else {
            require(superAdminNum > 1, "Must keep one super admin");
            superAdminNum--;
            removeAuth(_account);
            IHelper(helperAddr).removeAuth(_account);
        }
        superAdmins[_account] = isAdd;
        emit UpdateSupAdmin(_account, isAdd, msg.sender);
    }
```

Listing 3.10: Example Privileged Operations in ProjectShare

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been confirmed by the team. The team clarifies that governance administrators of the on-chain community are not required to go through a DAO-style governance contract.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Project Chosen` protocol, which allows users to choose the projects they are optimistic about, and then predict whether the price of this project will rise or fall compared with the IDO price on the seventh day after `TGE`. Then, the protocol admin can enter it according to the objective price on the seventh day after `TGE`, and then settle the reward for the successful prediction and generate the winner medal `NFT`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

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

[3] MITRE. CWE-770: Allocation of Resources Without Limits or Throttling. https://cwe.mitre.org/data/definitions/770.html.

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

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

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

[7] MITRE. CWE CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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