# SMART CONTRACT AUDIT REPORT

for

# Bool Network

Prepared By: Xiaomi Huang

PeckShield

March 10, 2024

## Document Properties

| | |
|---|---|
| Client | Bool |
| Title | Smart Contract Audit Report |
| Target | Bool |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Jason Shen, Xuxian Jiang |
| Reviewed by | Xiaomi Huang |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | March 10, 2024 | Xuxian Jiang | Final Release |
| 1.0-rc | February 29, 2024 | Xuxian Jiang | 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** **4**
- 1.1 About Bool Network . . . 4
- 1.2 About PeckShield . . . 5
- 1.3 Methodology . . . 5
- 1.4 Disclaimer . . . 7

**2 Findings** **9**
- 2.1 Summary . . . 9
- 2.2 Key Findings . . . 10

**3 Detailed Results** **11**
- 3.1 Possible Signature Verification Bypass in Messenger . . . 11
- 3.2 Improved txUniqueIdentification Logic in MessageChannelController . . . 12
- 3.3 Improved Validation Upon Parameter Changes . . . 14
- 3.4 Trust Issue of Admin Keys . . . 15

**4 Conclusion** **18**

**References** **19**

# 1 Introduction

Given the opportunity to review the design document and related source code of the Bool Network 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 Bool Network

Bool Network proposes an external verification model to facilitate arbitrary message transmission (AMT) across heterogeneous networks. The protocol adopts an decentralized off-chain signature scheme to ensure the security, consistency, and on-chain verifiability of any cross-chain message from/to any chain. This audit focuses on the on-chain contracts in Bool Network. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Bool

| Item | Description |
|---|---|
| Name | Bool |
| Type | Smart Contract |
| Language | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | March 10, 2024 |

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

- https://github.com/boolnetwork/bool-contracts-v1.git (2f1c260)

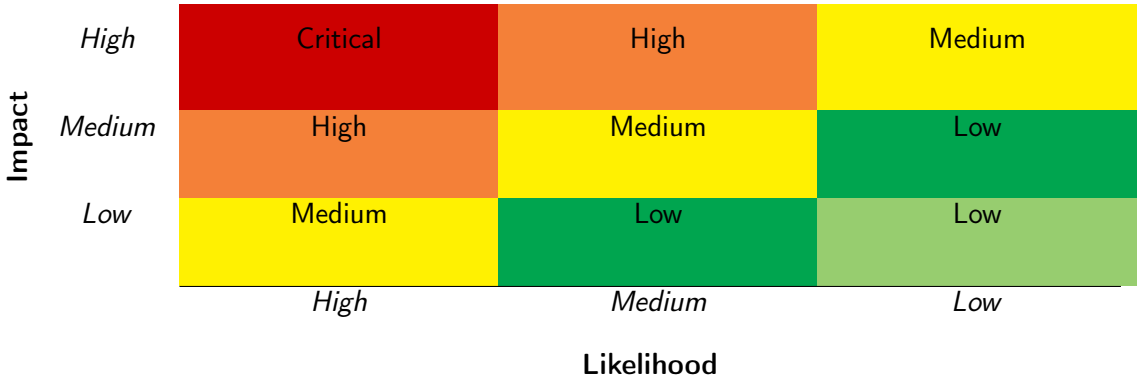
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/boolnetwork/bool-contracts-v1.git (9eca87d)

## 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




## 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 accordingly classified into four categories, 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

## 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 design and implementation of the `Bool Network` protocol. 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 | 1 | ■ |
| 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 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, 1 medium-severity vulnerability, and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | High | Possible Signature Verification Bypass in Messenger | Business Logic | Resolved |
| PVE-002 | Low | Improved txUniqueIdentification Logic in MessageChannelController | Coding Practices | Resolved |
| PVE-003 | Low | Improved Validation Upon Parameter Changes | Coding Practices | Confirmed |
| PVE-004 | Medium | Trust Issue Of Admin Keys | Security Features | Confirmed |

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 Possible Signature Verification Bypass in Messenger

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Messenger`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

### Description

The `Bool Network` protocol has a core `Messenger` constract that is responsible for facilitating cross-chain communication across networks. In the process of verifying received messages from `Bool Network`, we notice the verification logic may be bypassed.

In the following, we shows the implementation of the related `receiveFromBool()` routine. It has a rather straightforward logic in validating the received message and invoking the `receiveFromMessenger()` function on the intended `anchor`. And the validation is performed in an internal helper `_verifySignature()`. Unfortunately, the helper relies on the given user input and the baseline comparison also depends on external input, which is not trustworthy either. As a result, the current validation can be readily bypassed. A bypassed cross-chain message can be crafted to completely mess up the consumer's state, potentially draining funds in the pool.

```
129     function receiveFromBool(
130         Message memory message,
131         bytes calldata signature
132     ) external override nonReentrant {
133         // First and foremost, check the replay attack
134         (uint32 srcChainId, , uint192 nonce) = _unpackAndCheckTxIdentification(
135             message.txUniqueIdentification
136         );

138         // Check if the signature is valid
139         if (!_verifySignature(message, signature)) revert INVALID_SIGNATURE();
```

```
141         address dstAnchor = message.dstAnchor.fromBytes32ToAddress();

143         IAnchor.MessageStatus status = IAnchor(dstAnchor).receiveFromMessenger(
144             message.txUniqueIdentification,
145             message.crossType,
146             message.bnExtraFeed,
147             message.payload
148         );
149         ...
150     }
```

Listing 3.1: `Messenger::receiveFromBool()`

```
293     function _verifySignature(
294         Message memory message,
295         bytes calldata signature
296     ) internal view returns (bool) {
297         address committee = IAnchor(message.dstAnchor.fromBytes32ToAddress()).committee
                ();
298         bytes32 structDigest = keccak256(abi.encode(message));
299         bytes32 signedHash = ECDSA.toEthSignedMessageHash(structDigest);
300         return ECDSA.recover(signedHash, signature) == committee;
301     }
```

Listing 3.2: `Messenger::_verifySignature()`

**Recommendation** Improve the above routine to ensure the signature validation is securely performed without being bypassed.

**Status** This issue has been fixed by the following commit: `cb6c4ea`.

## 3.2 Improved txUniqueIdentification Logic in MessageChannelController

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MessageChannelController`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

### Description

As mentioned in Section 3.1, `Bool Network` has a core `Messenger` constract that is responsible for facilitating cross-chain communication across networks. This constract inherits from a parent contract `MessageChannelController` that defines the unidirectional channel nonce and generates the unique

message ID `txUniqueIdentification`. While examining the unique ID generation and related encoding/decoding, we notice current encoding/decoding logic can be improved.

To elaborate, we show below the implementation of the related routines, i.e., `_cptIdentification()` and `_unpackAndCheckTxIdentification()`. The former routine generates the unique ID of the next cross-chain message while the latter decodes the ID back to its member fields. It come to our attention that `txUniqueIdentification` is defined as `bytes32`, which in essence is an `uint256` integer. With that, we suggest to make use of bit operations by avoiding current `abi.encodePacked` approach. The bit operations can greatly simplify the code readability and reliability.

```
41      function _cptIdentification(
42          uint32 dstChainId
43      ) internal returns (bytes32 txUniqueIdentification) {
44          txUniqueIdentification = bytes32(
45              abi.encodePacked(
46                  uint32(_localChainId),
47                  uint32(dstChainId),
48                  ++_nextExportNonce[dstChainId]
49              )
50          );
51      }
52
53      /**
54       * @dev Unpacks the unique identification of the cross-chain message at the
             destination chain
55       * and checks if it is valid.
56       */
57      function _unpackAndCheckTxIdentification(
58          bytes32 txUniqueIdentification
59      ) internal view returns (uint32 srcChainId_, uint32 dstChainId_, uint192 nonce_) {
60          bytes memory _bytesTxId = abi.encode(txUniqueIdentification);
61          // solhint-disable-next-line no-inline-assembly
62          assembly {
63              srcChainId_ := mload(add(_bytesTxId, 4))
64              dstChainId_ := mload(add(_bytesTxId, 8))
65              nonce_ := mload(add(_bytesTxId, 32))
66          }
67          if (dstChainId_ != _localChainId) revert INVALID_CHAIN_ID(dstChainId_);
68          if (_importCheckPoints[srcChainId_][nonce_])
69              revert DUPLICATED_TX_IDENTIFICATION(txUniqueIdentification);
70      }
```

Listing 3.3: `MessageChannelController::_cptIdentification()/_unpackAndCheckTxIdentification()`

**Recommendation** Revise above-mentioned routines with suggested bit operations. An example revision is shown in the following:

```
41      function _cptIdentification(
42          uint32 dstChainId
43      ) internal returns (bytes32 txUniqueIdentification) {
44          txUniqueIdentification = bytes32(uint256(
```

```
            (uint256(_localChainId) << 224)
             (uint256(dstChainId) << 192)
             uint256(++_nextExportNonce[dstChainId])
        ));
    }
    function _unpackAndCheckTxIdentification(
        bytes32 txUniqueIdentification
    ) internal view returns (uint32 srcChainId_, uint32 dstChainId_, uint192 nonce_) {
        uint256 id = uint256(txUniqueIdentification);
        srcChainId_ = uint32(id >> 224);
        dstChainId_ = uint32(id >> 192);
        nonce_ = uint192(id);

        if (dstChainId_ != _localChainId) revert INVALID_CHAIN_ID(dstChainId_);
        if (_importCheckPoints[srcChainId_][nonce_])
            revert DUPLICATED_TX_IDENTIFICATION(txUniqueIdentification);
    }
```

Listing 3.4: Revised `MessageChannelController::_cptIdentification()/_unpackAndCheckTxIdentification()`

**Status** This issue has been fixed by the following commit: `4246f9a`.

## 3.3 Improved Validation Upon Parameter Changes

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Pool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Bool Network` protocol is no exception. Specifically, if we examine the `Pool` contract, it has defined a number of protocol-wide risk parameters, such as `swapLimit` and `feeRatio`. In the following, we show the corresponding routines that allow for their changes.

```
    function setSwapLimit(uint256 newLimit) external override onlyRouter {
        _setSwapLimit(newLimit);
    }

    /**
     * @dev Only callable by the Router.
     * @param newFeeRatio New fee ratio.
     */
    function setFeeRatio(uint16 newFeeRatio) external override onlyRouter {
        _setFeeRatio(newFeeRatio);
```

```
244     }
245     ...
246
247     function _setSwapLimit(uint256 newLimit) private {
248         uint256 previousLimit = swapLimit;
249         swapLimit = newLimit;
250         emit SwapLimitUpdated(previousLimit, newLimit);
251     }
252
253     function _setFeeRatio(uint16 newFeeRatio) private {
254         uint16 previousFeeRatio = feeRatio;
255         feeRatio = newFeeRatio;
256         emit FeeRatioUpdated(previousFeeRatio, newFeeRatio);
257     }
```

Listing 3.5: Pool::setSwapLimit() and Pool::setFeeRatio()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `feeRatio` may charge unreasonably high fee in the `pool` swaps, hence incurring cost to users or hurting the adoption of the protocol.

**Recommendation** Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

**Status** This issue has been confirmed.

## 3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

### Description

In `Bool Network`, there is a privileged `manager` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure various parameters, assign admins, collect fees, and execute privileged operations). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be

scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
171     function batchEnablePath(uint32[] calldata newPaths) external onlyManager {
172         for (uint32 i = 0; i < newPaths.length; i++) {
173             _modifyPath(newPaths[i], true);
174             _initNewPathNonce(newPaths[i]);
175         }
176     }
177
178     /**
179      * @dev Batch disables multiple paths.
180      * @param newPaths The IDs of the paths to disable.
181      */
182     function batchDisablePath(uint32[] calldata newPaths) external onlyManager {
183         for (uint32 i = 0; i < newPaths.length; i++) {
184             _modifyPath(newPaths[i], false);
185         }
186     }
187
188     /**
189      * @dev Sets the fee receiver.
190      * @param newFeeReceiver The address of the new fee receiver.
191      */
192     function setFeeReceiver(address newFeeReceiver) external {
193         _checkFeeReceiver(msg.sender);
194         _setFeeReceiver(newFeeReceiver);
195     }
196
197     /**
198      * @dev Sets the fee parser.
199      * @param newFeeParser The address of the new fee parser.
200      */
201     function setFeeParser(address newFeeParser) external onlyManager {
202         _setFeeParser(newFeeParser);
203     }
204
205     /**
206      * @dev Withdraws protocol fees.
207      * @param to The address to withdraw the protocol fees to.
208      * @param amount The amount of protocol fees to withdraw.
209      */
210     function withdrawProtocolFee(address payable to, uint256 amount) external {
211         _checkFeeReceiver(msg.sender);
212
213         address thisAddress_ = thisAddress;
214         uint256 collectableProtocolFee = _payableFees[thisAddress_];
215
216         if (collectableProtocolFee < amount) revert WITHDRAW_EXCEED_DEBIT();
217
218         _payableFees[thisAddress_] = collectableProtocolFee.sub(amount);
219
```

```
220         _transferNativeToken(to, amount);
221
222         emit ProtocolFeeWithdrawn(to, amount);
223     }
```

Listing 3.6: Example Privileged Functions in `Messenger`

Note that if the privileged `owner` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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. In the meantime, a timelock-based mechanism can also be considered as mitigation.

**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.

# 4 Conclusion

In this audit, we have analyzed the design and implementation of the Bool Network protocol, which proposes an external verification model to facilitate arbitrary message transmission (AMT) across heterogeneous networks. The protocol adopts an decentralized off-chain signature scheme to ensure the security, consistency, and on-chain verifiability of any cross-chain message from/to any chain. This audit focuses on the on-chain contracts in Bool Network. 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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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