SMART CONTRACT AUDIT REPORT

for

COGI Protocol

Prepared By: Yiqun Chen

PeckShield

November 12, 2021

Document Properties

Client	COGI
Title	Smart Contract Audit Report
Target	COGI
Version	1.0
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Final

Version Info

Version	Date	Author(s)	Description
1.0	November 12, 2021	Yiqun Chen	Final Release
1.0-rc	November 10, 2021	Yiqun Chen	Release Candidate

Contact

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

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the **COGI** 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 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 COGI

`COGI` is an innovative `GameFi` project that combines `COGI` token and `NFT Market`. `NFT` plays a lot of roles in the `COGI` protocol (e.g., `characters` and `equipments`). And users are able to trade `NFT` via the `NFT Market`. The `NFT` holder creates the sell order in the market, and the `NFT` listed for sale will be transferred from the owner to the market. The price of the `NFT` for sale should not be less than the `listing price` (0.001 `Ether`). Once sold, it will be transferred from the market to the buyer.

The basic information of COGI is as follows:

Table 1.1: Basic Information of COGI

Item	Description
Issuer	COGI
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 12, 2021

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

- https://github.com/COGI-Technology/cogi_contracts (c5dc614)

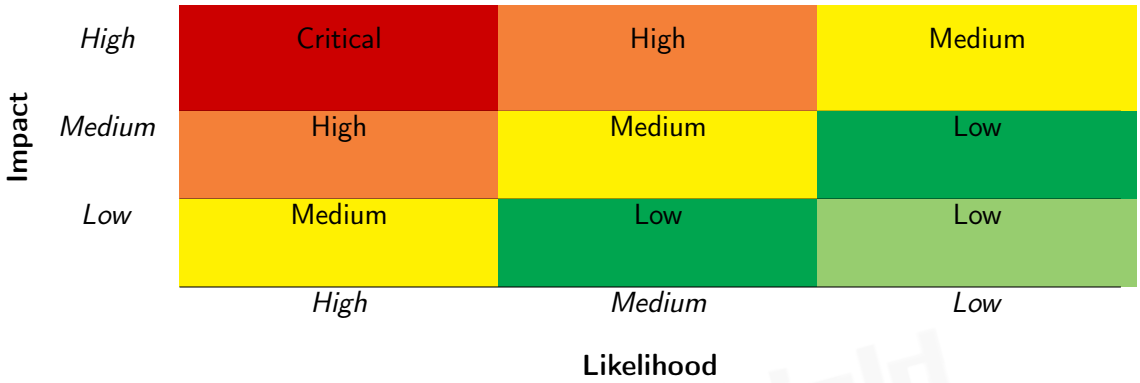
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/COGI-Technology/cogi_contracts (d87a739)

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 the 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 checklist of 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

Table 1.3: The Full Audit Checklist

Category	Checklist Items
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

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) [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 Logic	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 implementation of the COGI 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 logic, 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	1	■
Informational	2	■■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities 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, 1 low-severity vulnerability, and 2 informational recommendations.

Table 2.1: Key COGI Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-002	Informational	Redundant Code Removal	Coding Practices	Fixed
PVE-003	Low	Market Bypass With Direct ERC721 safeTransferFrom()	Business logic	Resolved
PVE-004	High	Improper Amount Of Ether Transferred	Business logic	Fixed
PVE-005	Informational	Improved Ether Transfers	Business Logics	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Trust Issue Of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `COGI` protocol, there is a special `admin` account with the `MINTER_ROLE` role. This `admin` account plays a critical role in governing and regulating the system-wide operations (e.g., minting additional tokens into circulation). 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 `admin` account and one of its related privileged accesses in current contract.

```
44      function _mint(address account, uint256 amount) internal virtual override {
45          require(_supply.add(amount) <= _maxSupply, "Over maxSupply");
46          _supply = _supply.add(amount);
47          super._mint(account, amount);
48      }


51      function mint(uint256 amount) public virtual {
52          require(hasRole(MINTER_ROLE, _msgSender()), "Must have minter role to mint");
53          _mint(_msgSender(), amount);
54      }
```

Listing 3.1: `CogiERC20::mint()`

We emphasize that the privilege assignment is necessary and consistent with the token design. However, it is worrisome if the `admin` account is a plain EOA account. A revised 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.

The discussion with the team has confirmed that they will mint the maximum amount of tokens once the contract is deployed. As a result, no more tokens will be minted for any addresses.

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.

3.2 Redundant Code Removal

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [2]

Description

The COGI protocol makes good use of a number of reference contracts, such as Initializable, ERC20Upgradeable, ERC20PausableUpgradeable, ERC20SnapshotUpgradeable, to facilitate its code implementation and organization. The CogiERC20 contract has so far imported at least six reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

Specifically, the ERC20PausableUpgradeable and the ERC20SnapshotUpgradeable import is not necessary as it is not used at all in CogiERC20.

```
17  contract CogiERC20 is
18      Initializable,
19      ERC20Upgradeable,
20      OwnableUpgradeable,
21      ERC20PausableUpgradeable,
22      ERC20SnapshotUpgradeable,
23      AccessControlEnumerableUpgradeable
```

Listing 3.2: CogiERC20

Besides, there is also a repeated initialization in the CogiERC721 contract that can be removed. To facilitate our discussion, we show the code snippet of CogiERC721 and ERC721Upgradeable below. The CogiERC721 contract uses the initialize() function to configure system parameters. However, as an internal function exists in the __ERC721_init() function, the __ERC721_init_unchained() function

would be called twice in the `initialize()` function, which is redundant. Also, the `onTokenTransfer()` event has never been triggered in the implementation, which can be removed as well.

```
59      function initialize(string memory _name, string memory _symbol, address
            contract_address) public virtual initializer {
60          __ERC721_init(_name, _symbol);
61          __Ownable_init();
62          __Context_init_unchained();
63          __AccessControlEnumerable_init_unchained();
64          __ERC721_init_unchained(_name, _symbol);
65          __ERC721Burnable_init_unchained();
66          proxyRegistryAddress = contract_address;
67          _setupRole(DEFAULT_ADMIN_ROLE, _msgSender());
68          _setupRole(MINTER_ROLE, _msgSender());
69          _setupRole(PAUSER_ROLE, _msgSender());
70      }
```

Listing 3.3: CogiERC721:: initialize ()

```
44      function __ERC721_init(string memory name_, string memory symbol_) internal
            initializer {
45          __Context_init_unchained();
46          __ERC165_init_unchained();
47          __ERC721_init_unchained(name_, symbol_);
48      }
49
50      function __ERC721_init_unchained(string memory name_, string memory symbol_)
            internal initializer {
51          _name = name_;
52          _symbol = symbol_;
53      }
```

Listing 3.4: ERC721Upgradeable::__ERC721_init()/__ERC721_init_unchained()

Moreover, while reviewing the `CogiNFTMarket` contract, we identified that there are two functions declared as `payable` functions. Typically, we expect `msg.value` is checked inside those functions since they could have the use cases of receiving ether. However, we do not see those use cases in the `publishItemToMarket()` function, which means the `payable` modifier is not necessary here.

```
50      function publishItemToMarket(
51      address nftContract,
52      uint256 tokenId,
53      uint256 price
54    ) public payable nonReentrant {
55      require(price > 0, "Price must be at least 1 wei");
56      require(price >= listingPrice, "Price must be equal or greater to listing price");
57
58      _itemIds.increment();
59      uint256 itemId = _itemIds.current();
60
61      idToMarketItem[itemId] =  MarketItem(
62        itemId,
```

```
63        nftContract,
64        tokenId,
65        payable(msg.sender),
66        payable(address(0)),
67        price,
68        false
69      );
70
71      IERC721(nftContract).transferFrom(msg.sender, address(this), tokenId);
72
73      emit MarketItemCreated(
74        itemId,
75        nftContract,
76        tokenId,
77        msg.sender,
78        address(0),
79        price,
80        false
81      );
82    }
```

Listing 3.5: CogiNFTMarket::publishItemToMarket()

Recommendation Remove unused reference contracts in the `CogiERC20` contract (i.e., `ERC20PausableUpgradeable` and `ERC20SnapshotUpgradeable`). Remove the internal `__ERC721_init_unchained()` function in the `initialize()` in the `CogiERC721` contract. Remove the `onTokenTransfer()` event in the `CogiERC721` contract. Remove the `payable` modifier in the `publishItemToMarket()` function in the `CogiNFTMarket` contract.

Status The issue has been fixed by this commit: `d87a739`.

3.3 Market Bypass With Direct ERC721 safeTransferFrom()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CogiNFTMarket`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

Each tradable asset supported in the COGI marketplace is represented as an `ERC721`-based NFT token, which naturally has the standard implementation, e.g., `transferFrom()/safeTransferFrom()`. By design, each tradable asset is listed for sale in the `CogiNFTMarket` will be transferred from the owner to the market. Once sold, it will be transferred from the market to the buyer after the

listingPrice is collected. Note the current implementation does not support the royalty that may be credited to the original NFT creator.

```
86      /* Transfers ownership of the item, as well as funds between parties */
87    function purchaseExecution(
88      address nftContract,
89      uint256 itemId
90      ) public payable nonReentrant {
91      uint price = idToMarketItem[itemId].price;
92      uint tokenId = idToMarketItem[itemId].tokenId;
93      require(msg.value == price, "Please submit the asking price in order to complete the
            purchase");

95      bool sent = idToMarketItem[itemId].seller.send(msg.value);
96      require(sent, "Failed to send Ether");
97      IERC721(nftContract).transferFrom(address(this), msg.sender, tokenId);
98      idToMarketItem[itemId].owner = payable(msg.sender);
99      idToMarketItem[itemId].sold = true;
100     _itemsSold.increment();
101     payable(owner).transfer(listingPrice);
102   }
```

Listing 3.6: CogiNFTMarket::purchaseExecution()

To elaborate, we show above the purchaseExecution() routine. This routine is used for buying a listed NFT with proper tax payment. It comes to our attention that instead of paying the tax amount, it is possible for the current owner and the buyer to directly negotiate a price, without paying the listingPrice (on behalf of CogiNFTMarket). The NFT can then be arranged and delivered by the current owner to directly call transferFrom()/safeTransferFrom() with the buyer as the recipient.

Recommendation Implement a locking mechanism so that any NFTs, need to be locked in the CogiNFTMarket contract in order to be available for public auction.

Status The issue has been resolved by this commit: d87a739.

3.4 Improper Amount Of Ether Transferred

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: CogiNFTMarket
- Category: Business logic [7]
- CWE subcategory: CWE-841 [4]

Description

As mentioned in Section 3.3, each tradable asset is listed for sale in the CogiNFTMarket will be transferred from the owner to the market. Once sold, it will be transferred from the market to

the buyer after the `listingPrice` is collected. And the `listingPrice` should be paid by the buyer. However, the distribution of the `msg.value` (transferred in by the buyer) is incorrect.

To elaborate, we show below the `purchaseExecution()` routine from the `CogiNFTMarket` contract. This routine checks whether the `msg.value` equals to the `price` given by the seller or not. If equal, it will send all the received `Ether` to the seller (line 96). Then, the NFT will be transferred from the market to the buyer. However, since all the `Ether` are paid to the seller, there is nothing left for the payment of `listingPrice`. As a result, the execution will fail.

```
86    /* Transfers ownership of the item, as well as funds between parties */
87  function purchaseExecution(
88    address nftContract,
89    uint256 itemId
90    ) public payable nonReentrant {
91    uint price = idToMarketItem[itemId].price;
92    uint tokenId = idToMarketItem[itemId].tokenId;
93    require(msg.value == price, "Please submit the asking price in order to complete the
          purchase");

95    bool sent = idToMarketItem[itemId].seller.send(msg.value);
96    require(sent, "Failed to send Ether");
97    IERC721(nftContract).transferFrom(address(this), msg.sender, tokenId);
98    idToMarketItem[itemId].owner = payable(msg.sender);
99    idToMarketItem[itemId].sold = true;
100   _itemsSold.increment();
101   payable(owner).transfer(listingPrice);
102 }
```

Listing 3.7: CogiNFTMarket::purchaseExecution()

Recommendation Properly distribute the `msg.value` in the `purchaseExecution()` function (`msg.vaule` = `listingPrice` + price).

Status The issue has been fixed by this commit: `d87a739`.

3.5 Improved Ether Transfers

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CogiNFTMarket`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

In the `CogiNFTMarket` contract, the Solidity functions, `transfer()` and `send()`, are used for dealing with `Ether` transfers. For example, the `purchaseExecution()` function in the code snippet below allows

the buyer to send all the Ether to the owner address (line 101). However, as described in [1], when the owner address happens to be a contract which implements a callback function containing EVM instructions such as SLOAD, the 2300 gas supplied with transfer() might be insufficient, leading to an out-of-gas error. Note that the same issue also exists in Solidity's send() (line 96).

```
86      /* Transfers ownership of the item, as well as funds between parties */
87    function purchaseExecution(
88      address nftContract,
89      uint256 itemId
90      ) public payable nonReentrant {
91      uint price = idToMarketItem[itemId].price;
92      uint tokenId = idToMarketItem[itemId].tokenId;
93      require(msg.value == price, "Please submit the asking price in order to complete the
            purchase");

95      bool sent = idToMarketItem[itemId].seller.send(msg.value);
96      require(sent, "Failed to send Ether");
97      IERC721(nftContract).transferFrom(address(this), msg.sender, tokenId);
98      idToMarketItem[itemId].owner = payable(msg.sender);
99      idToMarketItem[itemId].sold = true;
100     _itemsSold.increment();
101     payable(owner).transfer(listingPrice);
102   }
```

Listing 3.8: CogiNFTMarket::purchaseExecution()

As recommended in [1], we suggest to stop using Solidity's transfer() and send() as well. Note that the use of call() leads to side effects such as reentrancy attacks and gas token vulnerabilities.

Recommendation Replace transfer() and send() with call().

Status The issue has been fixed by this commit: d87a739.

4 Conclusion

In this audit, we have analyzed the design and implementation of the COGI protocol. The system presents a unique offering to enable decentralized NFT operations. The audited COGI NFT Marketplace features fully trustless NFT trading and allows participants to safely discover, buy or sell any NFTs. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and fixed.

Moreover, 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] Steve Marx. Stop Using Solidity's transfer() Now. https://diligence.consensys.net/blog/2019/09/stop-using-soliditys-transfer-now/.

[2] MITRE. CWE-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

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