SMART CONTRACT AUDIT REPORT

for

NewBitcoinCity

Prepared By: Xiaomi Huang

PeckShield
October 20, 2023

Document Properties

Client	NewBitcoinCity
Title	Smart Contract Audit Report
Target	NewBitcoinCity
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 20, 2023	Xuxian Jiang	Final Release
1.0-rc	October 19, 2023	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

Given the opportunity to review the design document and related source code of the `NewBitcoinCity` 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 NewBitcoinCity

`NewBitcoinCity (NBC)` is an exclusive social app that offers an array of exceptional features. It does not require email accounts or wallets, no initial deposits, and still provides seamless integrations with other platforms. The unique `8-2-0` fee structure empowers creators, rewards referrers, and prioritizes the community. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The NewBitcoinCity

Item	Description
Name	NewBitcoinCity
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 20, 2023

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

- https://github.com/TrustlessMarket/alpha-keys-contract.git (0247f33)

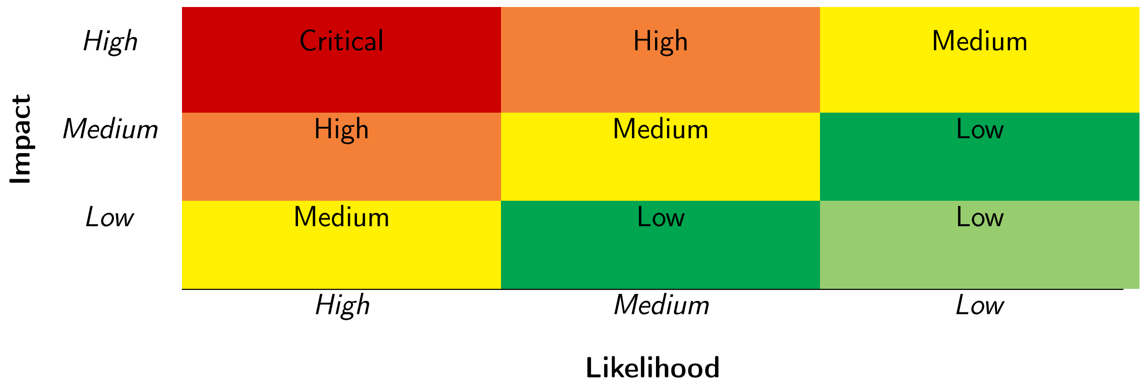
And here is the commit ID after fixes for the issues found in the audit have been checked in:

- https://github.com/TrustlessMarket/alpha-keys-contract.git (a9950fc)

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



	High	Critical	High	Medium
Impact	*Medium*	High	Medium	Low
	Low	Medium	Low	Low
		High	*Medium*	*Low*
			Likelihood	

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.

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

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

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 implementation of the `NewBitcoinCity (NBC)` 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	2	🟦🟦
Low	2	🟦🟦
Informational	0	
Total	5	

We have so far identified a list of potential issues. 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 2 low-severity vulnerabilities.

Table 2.1: Key NewBitcoinCity Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited Price Calculation in AlphaKeysToken	Business Logic	Resolved
PVE-002	Low	Incorrect Order Locking Validation in AlphaKeysFactory	Business Logic	Resolved
PVE-003	High	Revisited TokenA Buy Price in threeThreeTradeBTC()	Business Logic	Resolved
PVE-004	Low	Improved Parameter Validations in AlphaKeysFactory	Coding Practices	Resolved
PVE-005	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 Price Calculation in AlphaKeysToken

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

Description

The `NBC` protocol issues keys as `ERC20`-compliant tokens, ensuring that each user has their own `ERC20` contracts for keys. To issue a new key on `NBC`, the approval from the `NBC` admin is required to verify that the `Twitter` data and user information match. In the process of examining the issued keys, we notice the trade price calculation should be improved.

To elaborate, we show below the related `getPriceV2()` routine. It has a rather straightforward logic in pricing the share purchase. However, it should be revisited to compute `sum1` as `0` when `supply<= NUMBER_UNIT_PER_ONE_ETHER` (line 172). Note the same adjustment should be made for `sum2` as well (line 179).

```
168     function getPriceV2(
169         uint256 supply,
170         uint256 amount
171     ) internal pure returns (uint256) {
172         uint256 sum1 = supply == 0
173             ? 0
174             : ((supply - NUMBER_UNIT_PER_ONE_ETHER) *
175                 supply *
176                 (2 *
177                     (supply - NUMBER_UNIT_PER_ONE_ETHER) +
178                     NUMBER_UNIT_PER_ONE_ETHER)) / 6;
179         uint256 sum2 = supply == 0 && amount == 1
180             ? 0
181             : ((supply - NUMBER_UNIT_PER_ONE_ETHER + amount) *
182                 (supply + amount) *
```

```
                (2 *
                    (supply - NUMBER_UNIT_PER_ONE_ETHER + amount) +
                    NUMBER_UNIT_PER_ONE_ETHER)) / 6;
        uint256 summation = sum2 - sum1;
        return
            (summation * ONE_ETHER) /
            PRICE_KEYS_DENOMINATOR /
            (NUMBER_UNIT_PER_ONE_ETHER *
                NUMBER_UNIT_PER_ONE_ETHER *
                NUMBER_UNIT_PER_ONE_ETHER);
    }
```

Listing 3.1: `AlphaKeysToken::getPriceV2()`

Recommendation Improve the above routine by funding the extra payment back to the buyer.

Status The issue has been fixed by this commit: `375e78f` and `67bc9b1`.

3.2 Incorrect Order Locking Validation in AlphaKeysFactory

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

Description

The `NBC` protocol has a core `AlphaKeysFactory` contract for key instantiation and various types of trades. In the process of analyzing the unique type of `(3,3) friend trade`, we notice the related order validation is flawed. To elaborate, we show below the related `threeThreeTrade()` routine.

This type of trade works as follows: a user A initiates a friend request `(3,3)` to another user B by calling the function `threeThreeRequest()` with inputs specifying the token amount and the maximum price after fees. The user A can cancel the request at any time using `threeThreeCancel()` while the user B has the option to (1) reject the request using `threeThreeReject()`, in which case the tokens will be transferred back to the user A; or (2) accept the request via `threeThreeTrade()`, resulting in the issued keys being locked for 30 days. Both options require to check the given friend request order is `locked` or valid. However, it comes to our attention that the below `threeThreeTrade()` routine validates with the following requirement, i.e., `require(!order.locked)` (line 699), which should be revised as `require(order.locked)`.

```
687     function threeThreeTrade(
688         bytes32 orderId,
689         uint256 buyPriceAAfterFeeMax
```

```
    ) external notContract nonReentrant {
        require(buyPriceAAfterFeeMax > 0, "AKF_BPNZ");
        //
        ThreeThreeTypes.Order storage order = _threeThreeOrders[orderId];
        //
        require(
            order.status == ThreeThreeTypes.OrderStatus.Unfilled,
            "AKF_BOS"
        );
        require(!order.locked, "AKF_BOT");
        //
        address tokenB = order.tokenB;
        address ownerB = IAlphaKeysToken(tokenB).getPlayer();
        //
        require(_msgSender() == ownerB, "AKF_NOB");
        // save ownerB
        order.ownerB = ownerB;
        order.status = ThreeThreeTypes.OrderStatus.Filled;
        ...
    }
```

Listing 3.2: `AlphaKeysFactory::threeThreeTrade()`

Recommendation Improve the above routine by properly validating the friend request order.

Status The issue has been fixed by this commit: `f60f8f5`.

3.3 Revisited TokenA Buy Price in threeThreeTradeBTC()

- ID: PVE-003
- Severity: High
- Likelihood: High
- Impact: Medium

- Target: `AlphaKeysFactory`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, `NBC` supports the unique type of `(3,3) friend trade`. While examining the associated trading functions, we notice a key routine makes use of a wrong price, which charges more then intended for the buying user.

To elaborate, we show below the affected `threeThreeTradeBTC()` routine. It has a rather straightforward logic in completing the `(3,3) friend request`. Following the same user scenario, a user A initiates a friend request `(3,3)` to another user B by specifying the token amount and the maximum price after fees. And the user B has the option to (1) reject the request or (2) accept the request. If the request is accepted, both should share the equal buy price. However, our analysis shows that the user A paid `buyPriceAfterFee` while the user B paid `buyPriceBAfterFeeMax` and these two numbers are not equal (line 877).

```
821     function threeThreeTradeBTC(
822         bytes32 orderId
823     ) external notContract nonReentrant {
824         ThreeThreeTypes.Order storage order = _threeThreeOrders[orderId];
825         //
826         require(
827             order.status == ThreeThreeTypes.OrderStatus.Unfilled,
828             "AKF_BOS"
829         );
830         require(order.locked, "AKF_ONL");
831         require(order.amount == 0, "AKF_ONR");
832         //
833         address tokenB = order.tokenB;
834         address ownerB = IAlphaKeysToken(tokenB).getPlayer();
835         //
836         require(_msgSender() == ownerB, "AKF_NOB");
837         // save ownerB
838         order.ownerB = ownerB;
839         order.status = ThreeThreeTypes.OrderStatus.Filled;
840         //
841         address ownerA = order.ownerA;
842         address tokenA = order.tokenA;
843         uint256 buyPriceBAfterFeeMax = order.buyPriceBAfterFeeMax;
844         uint24 protocolFeeRatioA = IAlphaKeysToken(tokenA)
845             .getProtocolFeeRatio();
846         uint24 playerFeeRatioA = IAlphaKeysToken(tokenA).getPlayerFeeRatio();
847         uint256 amountA = NumberMath.getBuyAmountMaxWithCash(
848             protocolFeeRatioA,
849             playerFeeRatioA,
850             tokenA,
851             buyPriceBAfterFeeMax
852         );
853         uint24 protocolFeeRatioB = IAlphaKeysToken(tokenB)
854             .getProtocolFeeRatio();
855         uint24 playerFeeRatioB = IAlphaKeysToken(tokenB).getPlayerFeeRatio();
856         uint256 amountB = NumberMath.getBuyAmountMaxWithCash(
857             protocolFeeRatioB,
858             playerFeeRatioB,
859             tokenB,
860             buyPriceBAfterFeeMax
861         );
862         order.amountA = amountA;
863         order.amountB = amountB;
864         // AKF_BANM: buy amount not min
865         require(amountA > 0 && amountB > 0, "AKF_BANM");
866         //
867         address vault = _vault;
868         //
869         uint256 buyPriceAfterFee = _buyKeysForV2ByToken(
```

```
            tokenB,
            vault,
            amountB,
            buyPriceBAfterFeeMax,
            ownerA,
            TokenTypes.OrderType.ThreeThreeOrder
        );
        uint256 refundAmount = buyPriceBAfterFeeMax.sub(buyPriceAfterFee);
        if (refundAmount > 0) {
            TransferHelper.safeTransferFrom(_btc, vault, ownerA, refundAmount);
        }
        //
        _buyKeysForV2ByToken(
            tokenA,
            ownerB,
            amountA,
            buyPriceBAfterFeeMax,
            ownerB,
            TokenTypes.OrderType.ThreeThreeOrder
        );
        //
        emit ThreeThreeTradeBTC(
            orderId,
            tokenA,
            ownerA,
            tokenB,
            ownerB,
            amountA,
            amountB
        );
        //
        IAlphaKeysToken(tokenA).permitLock30D(ownerB, amountA);
        IAlphaKeysToken(tokenB).permitLock30D(ownerA, amountB);
    }
```

Listing 3.3: `AlphaKeysFactory::threeThreeTradeBTC()`

Recommendation Improve the above routine by making use of the correct buying price.

Status The issue has been fixed by this commit: `f60f8f5`.

3.4 Improved Parameter Validations in AlphaKeysFactory

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `AlphaKeysFactory`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

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

```
    function setProtocolFeeRatio(uint24 protocolFeeRatio) external onlyOwner {
        _protocolFeeRatio = protocolFeeRatio;
    }

    function getProtocolFeeRatio() external view returns (uint24) {
        return _protocolFeeRatio;
    }

    function setPlayerFeeRatio(uint24 playerFeeRatio) external onlyOwner {
        _playerFeeRatio = playerFeeRatio;
    }
```

Listing 3.4: AlphaKeysFactory::setProtocolFeeRatio() and AlphaKeysFactory::setPlayerFeeRatio()

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 `_protocolFeeRatio` may charge unreasonably high fee in the payment, 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. If necessary, also consider emitting relevant events for their changes.

Status The issue has been fixed by this commit: `5e0467d`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `AlphaKeysFactory`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `NBC`, there is a privileged administrative account, i.e., `owner`. The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `AlphaKeysFactory` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setAlphaKeysTokenImplementation(
        address playerShareTokenImplementationArg
    ) external onlyOwner {
        _playerShareTokenImplementation = playerShareTokenImplementationArg;
    }

    function setAdmin(address admin) external onlyOwner {
        _admin = admin;
    }

    function setBTC(address btc) external onlyOwner {
        require(btc.isContract(), "AKF_BINC");
        _btc = btc;
    }

    function setVault(address vault) external onlyOwner {
        require(vault.isContract(), "AKF_VINC");
        _vault = vault;
    }
```

Listing 3.5: Example Privileged Operations in `AlphaKeysFactory`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative 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 changes 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 The issue has been confirmed. The team is in the process of transferring the privileged account to the intended DAO-like governance contract. All code and parameter updates will undergo a thorough review and voting process within an on-chain, community-based governance life cycle. This ensures the intended trustless nature and high-quality distributed governance. However, establishing this DAO setup will require some time. The plan is to initiate later, ideally when `New Bitcoin City` has developed a strong and quality community. As of now, it's only been around a month since its inception.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `NewBitcoinCity` protocol, which is an exclusive social app on `Bitcoin` that offers an array of exceptional features. It does not require email accounts or wallets, no initial deposits, and provides seamless integrations with other platforms. The unique `8-2-0` fee structure empowers creators, rewards referrers, and prioritizes the community. 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.