SMART CONTRACT AUDIT REPORT

for

Moremoney Protocol

Prepared By: Yiqun Chen

PeckShield

December 31, 2021

Document Properties

Client	Moremoney
Title	Smart Contract Audit Report
Target	Moremoney
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	December 31, 2021	Xuxian Jiang	Final Release
1.0-rc1	December 20, 2021	Xuxian Jiang	Release Candidate #1

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 design document and related smart contract source code of the Moremoney 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 Moremoney

Moremoney is a decentralized borrowing protocol that lets users take on interest free loans using both liquid and illiquid tokens as collateral, while still earning farm reward and/or interest on the collateral. Loans are issued out in the protocol's dollar pegged stablecoin. Base tokens like USDT, ETH, AVAX as well as LPT and other form of ibTKNs are supported as collateral. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Moremoney

Item	Description
Name	Moremoney
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 31, 2021

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

- https://github.com/MoreMoney-Finance/contracts.git (08984bd)

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

- https://github.com/MoreMoney-Finance/contracts.git (3f4d457)

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 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 `Moremoney` 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	0	
Medium	2	
Low	3	
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 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 2 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Moremoney Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper Contract Disabling in DependencyController	Coding Practices	Fixed
PVE-002	Low	Improved Validation in ControllerActions	Coding Practices	Fixed
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-004	Medium	Proper APF Updates in Strategy	Business Logic	Fixed
PVE-005	Informational	Generation of Meaningful Events For Important State Changes	Coding Practices	Fixed
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides 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 Proper Contract Disabling in DependencyController

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: DependencyController
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The Moremoney protocol makes extensive use of DependencyController to manage the roles, characters, and access control policies. While reviewing its logic, we notice one of its internal routine _disableContract() can be improved to remove stale states.

To elaborate, we show below the full implementation of the _disableContract() function. It comes to our attention that when a contract has been removed, its roles and characters, associated states are not fully removed, e.g., characterDependenciesByContr[contr] and roleDependenciesByContr[contr].

```
    /// Completely remove all roles, characters and un-manage a contract
    function _disableContract(address contr) internal {
        managedContracts.remove(contr);

        uint256[] memory charactersPlayed = DependentContract(contr)
            .charactersPlayed();
        uint256[] memory rolesPlayed = DependentContract(contr).rolesPlayed();

        uint256 len = rolesPlayed.length;
        for (uint256 i = 0; len > i; i++) {
            if (roles.roles(contr, rolesPlayed[i])) {
                _removeRole(rolesPlayed[i], contr);
            }
        }

        len = charactersPlayed.length;
        for (uint256 i = 0; len > i; i++) {
            if (roles.mainCharacters(charactersPlayed[i]) == contr) {
```

```
                _setMainCharacter(charactersPlayed[i], address(0));
            }
        }

        uint256[] storage dependsOnCharacters = characterDependenciesByContr[
            contr
        ];
        len = dependsOnCharacters.length;
        for (uint256 i; len > i; i++) {
            dependentsByCharacter[dependsOnCharacters[i]].remove(contr);
        }

        uint256[] storage dependsOnRoles = roleDependenciesByContr[contr];
        len = dependsOnRoles.length;
        for (uint256 i; len > i; i++) {
            dependentsByRole[dependsOnRoles[i]].remove(contr);
        }
    }
```

Listing 3.1: DependencyController::_disableContract()

Recommendation Remove the associated (stale) states once the contract is un-managed by `DependencyController`.

Status The issue has been fixed in the following commit: `3ce9e32`.

3.2 Improved Validation in Controller-Actions

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

To efficiently organize various controller actions, the `Moremoney` protocol prepares a number of controller actions, e.g., `DependencyCleaner`, `OracleActivation`, `StrategyTokenActivation`, and `TokenActivation`. While reviewing these controller actions, we observe that they can benefit from improved validation.

To illustrate, we show below the `DependencyCleaner` contract, which is designed to remove a variety of roles from the respective contracts. The current implementation assumes the given arrays of `_contracts` and `_roles2nix` share the same length. However, this assumption is not enforced yet! With that, we suggest to explicitly add the following requirement `require(_contracts.length == _roles2nix.length)` to validate this assumption.

```
    constructor(
        address[] memory _contracts,
        uint256[] memory _roles2nix,
        address _roles
    ) RoleAware(_roles) {
        contracts = _contracts;
        roles2nix = _roles2nix;
    }
```

Listing 3.2: `DependencyCleaner::constructor()`

The same issue is also applicable to other controller actions: `OracleActivation`, `StrategyTokenActivation`, and `TokenActivation`.

Recommendation Improve current controller actions by validating their input arguments.

Status The issue has been fixed in the following commit: `3ce9e32`.

3.3 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Business Logic [7]
- 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 the following, we examine the `transfer()` routine and related idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
```

```
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.3: ZRX.sol

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 the following, we show the `depositMigrationTokens()` routine in the `StrategyRegistry` contract. If the `USDT` token is supported as `token`, the unsafe version of `IERC20(token).approve(destination, amount)` (line 85) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
77      /// Endpoint for strategies to deposit tokens for migration destinations
78      /// to later withdraw
79      function depositMigrationTokens(address destination, address token)
80          external
81          nonReentrant
82      {
83          uint256 amount = IERC20(token).balanceOf(msg.sender);
84          IERC20(token).safeTransferFrom(msg.sender, address(this), amount);
85          IERC20(token).approve(destination, amount);
86      }
```

Listing 3.4: `StrategyRegistry::depositMigrationTokens()`

We should highlight that for the `safeApprove()` support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount.

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

Status The issue has been fixed in the following commit: `3ce9e32`.

3.4 Proper APF Updates in Strategy

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Strategy`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

`Moremoney` is a decentralized borrowing protocol that lets users take on `interest free loans` using both liquid and illiquid tokens as collateral, while still earning farm reward and/or interest on the collateral. While reviewing the current strategies to earn farm rewards, we notice one internal routine to update the `annual percentage factor (APF)` needs to be updated.

To elaborate, we show below the `_updateAPF()` function. The `APF` metric is defined as `APF = APR + 100%`. This function implements a rather straightforward logic in computing the elapsed time, calculating the resulting rate, and deriving the current `APF`. It comes to our attention that the `if`-condition of `iaddedBalance > 0 && tokenMeta.apfLastUpdated > block.timestamp` (line 610) should be revised as `iaddedBalance > 0 && tokenMeta.apfLastUpdated < block.timestamp`. Otherwise, the elapsed time is wrongfully computed.

```
604     function _updateAPF(
605         address token,
606         uint256 addedBalance,
607         uint256 basisValue
608     ) internal {
609         TokenMetadata storage tokenMeta = tokenMetadata[token];
610         if (addedBalance > 0 && tokenMeta.apfLastUpdated > block.timestamp) {
611             uint256 lastUpdated = tokenMeta.apfLastUpdated;
612             uint256 timeDelta = lastUpdated > 0
613                 ? block.timestamp - lastUpdated
614                 : 1 weeks;
615 
616             uint256 newRate = ((addedBalance + basisValue) *
617                 10_000 *
618                 (365 days)) /
619                 basisValue /
620                 timeDelta;
621 
622             uint256 smoothing = lastUpdated > 0 ? apfSmoothingPer10k : 0;
623             tokenMeta.apf =
624                 (tokenMeta.apf * smoothing) /
625                 10_000 +
626                 (newRate * (10_000 - smoothing)) /
627                 10_000;
628             tokenMeta.apfLastUpdated = block.timestamp;
```

```
        }
    }
```

Listing 3.5: `Strategy::_updateAPF()`

Recommendation Revise the above `_updateAPF()` routine to apply the right elapsed time to compute the `APF`.

Status The issue has been fixed in the following commit: `3ce9e32`.

3.5 Generation of Meaningful Events For Important State Changes

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `IsolatedLending`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the `IsolatedLending` contract as an example. This contract has public functions that are used to configure various system parameters. While examining the events that reflect their changes, we notice there is a lack of emitting important events that reflect important state changes. For example, when the minting fee for an asset is updated, there is no respective event being emitted (line 52).

```
42      /// Set the debt ceiling for an asset
43      function setAssetDebtCeiling(address token, uint256 ceiling)
44          external
45          onlyOwnerExecDisabler
46      {
47          assetConfigs[token].debtCeiling = ceiling;
48      }

50      /// Set minting fee per an asset
51      function setFeesPer10k(address token, uint256 fee) external onlyOwnerExec {
52          assetConfigs[token].feePer10k = fee;
```

```
    }

    /// Set central parameters per an asset
    function configureAsset(
        address token,
        uint256 ceiling,
        uint256 fee
    ) external onlyOwnerExecActivator {
        AssetConfig storage config = assetConfigs[token];
        config.debtCeiling = ceiling;
        config.feePer10k = fee;
    }
```

Listing 3.6: `IsolatedLending::setAssetDebtCeiling()/setFeesPer10k()`

Recommendation Properly emit respective events when important protocol-wide parameters are changed.

Status The issue has been fixed in the following commit: `3ce9e32`.

3.6 Trust Issue of Admin Keys

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

Description

In the `Moremoney` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., authorizing other roles, setting various parameters, and adjusting external oracles). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Moremoney` protocol.

```
    /// Run an executor contract in the executor role (which has ownership privileges
        throughout)
    function executeAsOwner(address executor) external onlyOwner nonReentrant {
        uint256[] memory requiredRoles = Executor(executor).rolesPlayed();
        uint256[] memory requiredCharacters = Executor(executor)
            .charactersPlayed();
        address[] memory extantCharacters = new address[](
```

```
            requiredCharacters.length
        );

        for (uint256 i = 0; requiredRoles.length > i; i++) {
            _giveRole(requiredRoles[i], executor);
        }

        for (uint256 i = 0; requiredCharacters.length > i; i++) {
            extantCharacters[i] = roles.mainCharacters(requiredCharacters[i]);
            _setMainCharacter(requiredCharacters[i], executor);
        }

        uint256[] memory dependsOnCharacters = DependentContract(executor)
            .dependsOnCharacters();
        uint256[] memory dependsOnRoles = DependentContract(executor)
            .dependsOnRoles();
        characterDependenciesByContr[executor] = dependsOnCharacters;
        roleDependenciesByContr[executor] = dependsOnRoles;

        updateCaches(executor);
        currentExecutor = executor;
        Executor(executor).execute();
        currentExecutor = address(0);

        uint256 len = requiredRoles.length;
        for (uint256 i = 0; len > i; i++) {
            _removeRole(requiredRoles[i], executor);
        }

        for (uint256 i = 0; requiredCharacters.length > i; i++) {
            _setMainCharacter(requiredCharacters[i], extantCharacters[i]);
        }
    }
```

Listing 3.7: DependencyController::executeAsOwner()

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that it is currently managed by a multi-sig account. We point out that a compromised `owner` account would allow the attacker to undermine necessary assumptions behind the protocol and subvert various protocol operations.

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 and partially mitigated with future plans for decentralized governance.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Moremoney` protocol, which is a decentralized borrowing protocol that lets users take on `interest free loans` using both liquid and illiquid tokens as collateral, while still earning farm reward and/or interest on the collateral. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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] 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.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.