SMART CONTRACT AUDIT REPORT

for

EURXB

Prepared By: Shuxiao Wang

PeckShield

May 5, 2021

Document Properties

Client	EURxb
Title	Smart Contract Audit Report
Target	EURxb
Version	1.0
Author	Xuxian Jiang
Auditors	Yiqun Chen, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2021	Xuxian Jiang	Final Release
1.0-rc	April 25, 2021	Xuxian Jiang	Release Candidate
0.2	April 6, 2021	Xuxian Jiang	Additional Findings
0.1	April 1, 2021	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **EURxb** protocol, we outline in this 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 contract can be further improved due to the presence of several issues. This document outlines our audit results.

1.1 About EURxb

The `EURxb.finance` protocol is the means by which existing securities are used to create blockchain based tokens with greater utility. The core focus of the protocol is to ensure a safe, regulated, and auditable path for institutional funds to gain exposure to Decentralized Finance (DeFi) and blockchain based financial instruments through `ISIN` registered securities. The `EURxb` is an ERC20 Euro Stable coin that earns real time interest of 7% per annum for the duration of the protocol's bond reserves' term. The `EURxb` is collateralized by `ISIN` registered securities (or `green bonds`) as `ERC721 NFTs` which are further over-collateralized (at a rate of 133%) by tokenized `ERC721 NFT`-based security assets.

The basic information of EURxb is as follows:

Table 1.1: Basic Information of EURxb

Item	Description
Issuer	EURxb
Website	https://eurxb.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2021

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

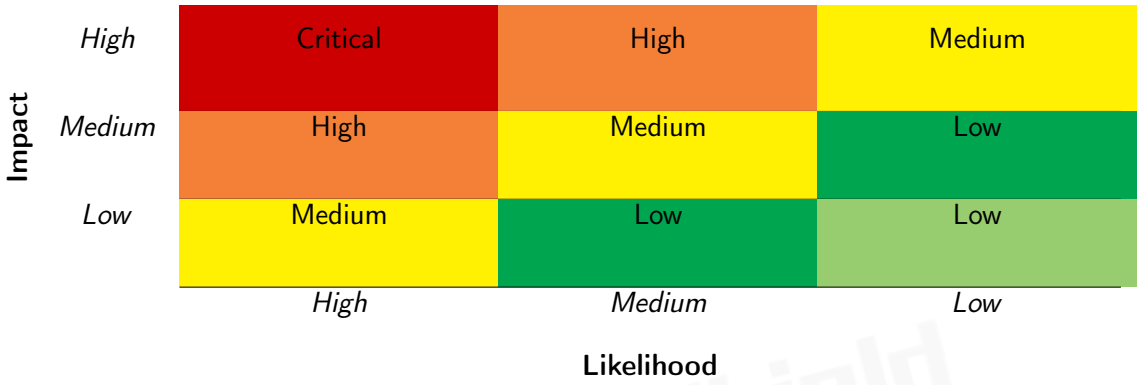
this audit.

- https://github.com/EURxbfinance/SmartBond.git (4eaaa4f)

1.2 About PeckShield

PeckShield Inc. [11] 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 [10]:

- 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

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

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 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) [9], 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 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 EURxb implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, 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	1	■
Low	2	■■
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 medium-severity vulnerability, 2 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key EURxb Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Unused allowList In BondToken	Coding Practices	Confirmed
PVE-002	Low	Improved Validation Of deposit()	Coding Practices	Confirmed
PVE-003	Informational	Removal Of Redundant Transfer in DDP::withdraw()	Business Logic	Resolved
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Resolved
PVE-005	Informational	Improved Logic Of balanceByTime()	Error Conditions, Return Values, Status Codes	Resolved

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

3 Detailed Results

3.1 Unused allowList In BondToken

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `BondToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the `EURxb` protocol is the means by which existing securities are used to create blockchain based tokens with greater utility. The implementation of this process follows four steps. The first three steps mirror the traditional procedure: the issue of an `ISIN` registered bond based on an extensive due diligence process on the underlying security, the grading of the bond, and the issue of an `ISIN` number by the `National Numbering Agency` in Norway. The bond's security is in the form of cash, cash equivalents, and other balance sheet assets. These assets are then tokenized, those tokens are collateralized to create bond tokens, and those bond tokens are made fungible to create a stablecoin.

In the following, we examine the `BondToken` implementation. Specifically, we focus on its initialization routine `configure()` to properly configure various entities of the protocol. This routine is properly protected with the `initializer` modifier that ensures it can only be initialized once. However, the first parameter `allowList` is not currently used. With that, it is suggested to revisit the interface definition in avoiding the pass-in of unused parameters.

```
53    function configure(address allowList, address sat, address ddp) external initializer
          {
54        require(allowList != address(0), "list address is invalid");
55        require(sat != address(0), "sat address is invalid");
56        require(ddp != address(0), "ddp address is invalid");

58        _setupRole(TokenAccessRoles.minter(), sat);
59        _setupRole(TokenAccessRoles.transferer(), sat);
```

```
61          _setupRole(TokenAccessRoles.burner(), ddp);
62          _setupRole(TokenAccessRoles.transferer(), ddp);

64          _ddp = ddp;
65          _sat = sat;
66      }
```

Listing 3.1: BondToken::configure()

Recommendation Remove the passed-in parameter of `allowList` if it is deemed not useful.

Status This issue has been confirmed.

3.2 Improved Validation Of deposit()

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

Description

In Section 3.1, we have briefly examined the four key steps in tokenizing existing securities to create greater utility. The final step in the process, which goes beyond the tokenization of the bond, is to use the `EURxb bond NFT` (`EBND`) as collateral in the `Delegated Debenture Position` (`DDP`) module. This allows for the creation of fungible debt instruments on a public blockchain, minted in the form of a fractionalized interest bearing Euro stablecoin, the `EURxb`. In the following, we examine the deposit logic in the `DDP` module.

To elaborate, we show below the `deposit()` routine of the `DDP` module. This routine accepts as arguments `EURxb bond NFT`, its value, and maturity and is programmed to be invoked only from the `Bond` contract.

```
48      function deposit(
49          uint256 tokenId,
50          uint256 value,
51          uint256 maturity,
52          address to) external override
53      {
54          // only bond is allowed to deposit
55          require(_msgSender() == _bond,
56              "caller is not allowed to deposit");

58          // mint EURxb tokens: amount of EURxb FT tokens = value of Bond NFT token.
```

```
59          IEURxb(_eurxb).mint(to, value);
60          IEURxb(_eurxb).addNewMaturity(value, maturity);
61      }
```

Listing 3.2: DDP::deposit()

Our analysis with the deposit logic shows the first parameter `tokenId` is not validated. However, we do not find any possible ways yet for exploitation. As a precaution, it is always helpful to apply necessary validation to check the presence of the given `tokenID` for strengthened security.

Recommendation Apply the same validate check in `deposit()` on the given `tokenID` as the `withdraw()` counterpart. An example revision is shown below.

```
48      function deposit(
49          uint256 tokenId,
50          uint256 value,
51          uint256 maturity,
52          address to) external override
53      {
54          // only bond is allowed to deposit
55          require(_msgSender() == _bond,
56              "caller is not allowed to deposit");
57
58          // check if token exists
59          require(
60              IBondToken(_bond).hasToken(tokenId),
61              "bond token id does not exist");
62
63          // mint EURxb tokens: amount of EURxb FT tokens = value of Bond NFT token.
64          IEURxb(_eurxb).mint(to, value);
65          IEURxb(_eurxb).addNewMaturity(value, maturity);
66      }
```

Listing 3.3: DDP::deposit()

Status This issue has been confirmed.

3.3 Removal Of Redundant Transfer in DDP::withdraw()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: DDP
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The EURxb protocol makes good use of a number of reference contracts, such as ERC20, ERC721, Context, and AccessControl, to facilitate its code implementation and organization. For example, the DDP smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the DDP::withdraw() routine, this routine can be invoked by any user to repay bond token. To elaborate, we show below its implementation. Our analysis shows that right before the burn of the specified bond NFT, there is a ownership transfer operation to the user if the is not initiated by the owner.

```
66      function withdraw(uint256 tokenId) external {
67          // check if token exists
68          require(
69              IBondToken(_bond).hasToken(tokenId),
70              "bond token id does not exist");
71
72          address user = _msgSender();
73
74          // get token properties
75          ( uint256 value, /* uint256 interest */, uint256 maturity ) = IBondToken(_bond)
76              .getTokenInfo(tokenId);
77
78          address owner = IERC721(_bond).ownerOf(tokenId);
79          bool isOwner = owner == user;
80
81          if (!isOwner) {
82              require (
83                  IAllowList(_allowList).isAllowedAccount(user),
84                  "user is not allowed");
85              require(
86                  block.timestamp > maturity + _claimPeriod,
87                  "claim period is not finished yet");
88          }
89
90          // check if enough money to repay
91          require(IERC20(_eurxb).balanceOf(user) >= value,
92              "not enough EURxb to withdraw");
93
```

```
94          // burn EURxb
95          IEURxb(_eurxb).burn(user, value);
96          if (maturity > block.timestamp) {
97              // only if maturity has not arrived yet
98              IEURxb(_eurxb).removeMaturity(value, maturity);
99          }
100
101         if (!isOwner) {
102             // if not owner, need to transfer ownership first
103             IBondToken(_bond).safeTransferFrom(owner, user, tokenId);
104         }
105
106         // burn token
107         IBondToken(_bond).burn(tokenId);
108     }
```

Listing 3.4: DDP::withdraw()

Recommendation Improve the `withdraw()` logic by removing the unnecessary `transfer()` (line 103).

Status This issue has been resolved. The team has informed us the there is a need to firstly move the bond to the one who repays it, and then burn it. With that, the bank could later verify that it was this person who burned the token. However, it is agreed that from a code point of view, this logic is unnecessary.

3.4 Trust Issue of Admin Keys

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

Description

In the `EURxb` protocol, there is a special administrative account, i.e., `operator`. This `operator` account plays a critical role in governing and regulating the system-wide operations (e.g., account whitelisting, SAT mints/burns, and parameter setting). It also has the privilege to regulate or govern the flow of assets among the involved components, i.e., `SecurityAssetToken`, `BondToken`, `DDP`, and `EURxb`.

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

```
36      function allowAccount (address account) external onlyOperator {
```

```
37          IAllowListChange(_allowList).allowAccount(account);
38      }

40      function disallowAccount(address account) external onlyOperator {
41          IAllowListChange(_allowList).disallowAccount(account);
42      }

44      function mintSecurityAssetToken(
45          address to,
46          uint256 value,
47          uint256 maturity) external onlyOperator
48      {
49          ISecurityAssetToken(_sat).mint(to, value, maturity);
50      }

52      function burnSecurityAssetToken(uint256 tokenId) external onlyOperator {
53          ISecurityAssetToken(_sat).burn(tokenId);
54      }

56      function transferSecurityAssetToken(
57          address from,
58          address to,
59          uint256 tokenId) external onlyOperator
60      {
61          ISecurityAssetToken(_sat).transferFrom(from, to, tokenId);
62      }

64      function setClaimPeriod(uint256 claimPeriod) external onlyOperator {
65          IDDP(_ddp).setClaimPeriod(claimPeriod);
66      }
```

Listing 3.5: Various Setters in MultiSignature

We emphasize that the privilege assignment with various core contracts is necessary and required for proper protocol operations. In current implementation, the `operator` is governed by a `multi-sig` account.

We point out that a compromised `operator` account would allow the attacker to maliciously change other settings to compromise funds, which directly undermines the assumption of the `EURxb` protocol.

Recommendation Transition the `operator` account itself to be a multi-sig account. It is also suggested to mediate new changes of privileged operations with 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 a multi-sig account to regulate the governance privileges.

3.5 Improved Logic Of balanceByTime()

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: EURxb
- Category: Status Codes [8]
- CWE subcategory: CWE-391 [3]

Description

As discussed earlier, the EURxb is an ERC20-based Euro stablecoin that earns real time interest of 7% per annum for the duration of the protocol's bond reserves' term. In the following, we examine a specific helper routine, i.e., balanceByTime(), in the EURxb token contract.

To elaborate, we show below the routine implementation. As the name indicates, this routine is designed to properly return the account balance at the queried timestamp. Our analysis shows that this routine has properly taken into possible interest from expiring maturity, it unnecessarily reverts the query when the queried timestamp is earlier than the current interest accrual timestamp accrualTimestamp, i.e., timestamp < _accrualTimestamp.

```
249     function balanceByTime(address account, uint256 timestamp)
250         public
251         view
252         returns (uint256)
253     {
254         if (super.balanceOf(account) > 0 && _holderIndex[account] > 0) {
255             uint256 currentTotalActiveValue = _totalActiveValue;
256             uint256 currentExpIndex = _expIndex;
257             uint256 head = _list.getHead();
258             uint256 currentAccrualTimestamp = _accrualTimestamp;
259             (uint256 amount, uint256 maturityEnd, , uint256 next) = _list.getNodeValue(
                    head);
260             while (
261                 _list.listExists() && maturityEnd <= timestamp &&
                        currentAccrualTimestamp < maturityEnd
262             ) {
263                 currentExpIndex = _calculateInterest(
264                     maturityEnd,
265                     _annualInterest.mul(currentTotalActiveValue),
266                     currentExpIndex,
267                     currentAccrualTimestamp
268                 );
269                 currentAccrualTimestamp = maturityEnd;

271                 uint256 deleteAmount = _deletedMaturity[maturityEnd];
272                 currentTotalActiveValue = currentTotalActiveValue.sub(amount.sub(
                        deleteAmount));
```

```
274                 if (next == 0) {
275                     break;
276                 }
278                 (amount, maturityEnd, , next) = _list.getNodeValue(next);
279             }
281             currentExpIndex = _calculateInterest(
282                 timestamp,
283                 _annualInterest.mul(currentTotalActiveValue),
284                 currentExpIndex,
285                 currentAccrualTimestamp
286             );
287             return super.balanceOf(account).mul(currentExpIndex).div(_holderIndex[
                    account]);
288         }
289         return super.balanceOf(account);
290     }
```

Listing 3.6: EURxb::balanceByTime()

Recommendation This is a minor issue and it does not affect the normal functionality in any negative way.

Status This issue has been resolved as this `balanceByTime()` function is implemented for convenience and Tthe timestamp as a parameter is needed to look ahead.

4 Conclusion

In this audit, we have analyzed the EURxb design and implementation. The system presents a unique offering as an ERC20-based Euro stablecoin that earns real time interest of 7% per annum for the duration of the protocol's bond reserves' term. The current code base is well organized and 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-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.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 CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

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

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

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