SMART CONTRACT AUDIT REPORT

for

SynFutures V2

Prepared By: Xiaomi Huang

PeckShield

July 1, 2022

Document Properties

Client	SynFutures
Title	Smart Contract Audit Report
Target	SynFutures V2
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 1, 2022	Shulin Bie	Final Release
1.0-rc	June 16, 2022	Shulin Bie	Release Candidate

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 smart contract source code of the `SynFutures V2` 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 SynFutures V2

`SynFutures V2` is a decentralized derivatives platform. In comparison to `SynFutures V1`, `SynFutures V2` provides a more streamlined, easier-to-navigate user experience for both traders and liquidity providers, as well as more trading products and features designed with increased capital efficiency. For traders, `SynFutures V2` introduces `Perpetual Futures`, which is a never-ending futures contract with native permissionless listing, guaranteed price convergence to spot index, and a forward-looking funding mechanism. For LPs, `SynFutures V2` will be the first AMM-based derivatives protocol to natively incorporate ranged liquidity provision and limit orders, in addition to the vanilla AMM liquidity provision.

Table 1.1: Basic Information of SynFutures V2

Item	Description
Target	SynFutures V2
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	July 1, 2022

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

- https://github.com/SynFutures/v2-contracts.git (d6df565)

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

- https://github.com/SynFutures/v2-contracts.git (ff28a81)

1.2 About PeckShield

PeckShield Inc. [12] 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 OWASP Risk Rating Methodology [11]:

- 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.

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

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 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) [10], 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 `SynFutures V2` 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 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	1	
Low	3	
Undetermined	1	
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 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 medium-severity vulnerability, 3 low-severity vulnerabilities, and 1 undetermined issue.

Table 2.1: Key SynFutures V2 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Fixed
PVE-002	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Fixed
PVE-003	Low	Suggested Addition Of rescueToken() To SynFuturesV2Underlying	Coding Practices	Confirmed
PVE-004	Undetermined	Suggested Reentrancy Protection In Current Implementation	Time and State	Fixed
PVE-005	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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

3 | Detailed Results

3.1 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SynFuturesV2Router
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

In the SynFutures V2 implementation, the SynFuturesV2Router contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., deposit(), accepts the deposits of the supported assets. While examining its logic, we observe the incoming token (i.e., quoteInfo.quote) is transferred to the SynFuturesV2Router contract and then transferred to the SynFuturesV2Underlying contract. This is reasonable under the assumption that these transfers will always result in full transfer. Otherwise, the transaction will be reverted.

```
    function deposit(address underlying, address trader, uint amount) public payable
        nonReentrant {
        Types.QuoteInfo memory quoteInfo = IUnderlying(underlying).quoteInfo();
        _deposit(underlying, quoteInfo, trader, amount);
        return;
    }

    function _deposit(
        address underlying, Types.QuoteInfo memory quoteInfo, address trader, uint
            amount
    ) internal returns (uint) {
        require(msg.value == 0, "invalid msg.value");
        uint tokenAmount = amount / quoteInfo.scaler;//10**(18 - token decimals)
        IERC20(quoteInfo.quote).safeTransferFrom(msg.sender, address(this), tokenAmount)
            ;

        amount = tokenAmount * quoteInfo.scaler;
        IUnderlying(underlying).deposit(trader, amount);
```

```
        return amount;
    }
```

Listing 3.1: SynFuturesV2Router::deposit()

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every transfer() or transferFrom(). (Another type is rebasing tokens such as YAM.) As a result, this may not meet the assumption behind these routines related to token transfer.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in transfer() or transferFrom() will always result in full transfer, we need to ensure the increased or decreased amount in the SynFuturesV2Router contract before and after the transfer() or transferFrom() is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Another mitigation is to regulate the set of ERC20 tokens that are permitted into SynFutures V2. In SynFutures V2 protocol, it is indeed possible to effectively regulate the set of tokens that can be supported. Keep in mind that there exist certain assets (e.g., USDT) that may have control switches that can be dynamically exercised to suddenly become one.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the transfer()/transferFrom() call to ensure the book-keeping amount is accurate. This support may bring additional gas cost. Also, keep in mind that certain tokens may not be deflationary for the time being. However, they could have a control switch that can be exercised to turn them into deflationary tokens. One example is the widely-adopted USDT.

Status The issue has been addressed by the following commit: 2fe40d9.

3.2 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SynFuturesV2Router
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [2]

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 this section, we examine the `approve()` routine and analyze possible idiosyncrasies from current widely-used token contracts. In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below.

```
/**
* @dev Approve the passed address to spend the specified amount of tokens on behalf
    of msg.sender.
* @param _spender The address which will spend the funds.
* @param _value The amount of tokens to be spent.
*/
function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

    // To change the approve amount you first have to reduce the addresses`
    //  allowance to zero by calling `approve(_spender, 0)` if it is not
    //  already 0 to mitigate the race condition described here:
    //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
    require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

    allowed[msg.sender][_spender] = _value;
    Approval(msg.sender, _spender, _value);
}
```

Listing 3.2: USDT Token **Contract**

It is important to note that the `approve()` function does not have a return value. However, the `IERC20` interface has defined the following `approve()` interface with a `bool` return value: `function approve(address spender, uint256 amount)external returns (bool)`. As a result, the call to `approve()` may expect a return value. With the lack of return value of `USDT`'s `approve()`, the call will be unfortunately reverted.

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`. 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.

In the following, we show the internal `_createUnderlying()` routine in the `SynFuturesV2Router` contract. If the `USDT` token is supported as `quoteInfo.quote`, the unsafe version of `IERC20(quoteInfo.quote).approve(underlying, LibMathUnsigned.POSITIVE_INT256_MAX())` (line 142) may revert as there is no return value in the `USDT` token contract's `approve()` implementation (but the `IERC20` interface expects a return value)!

```
111     function _createUnderlying(
112         string calldata marketType, bytes calldata deployData, bytes calldata
                initializeData, bool withNativeToken
113     ) internal returns (address) {
114         address market = markets[marketType];
115         require(market != address(0), "unknown market");
116         address underlying;
117         {
```

```
            bytes32 index;
            address expectedUnderlying;
            // deployData varies as market, but should always contains the expected
                deployed pair address(which is
            // used as the feeders' mapping key), to check the deployed one use the
                correct feeder.
            (parameters, index, expectedUnderlying) = IMarket(market).newUnderlying(
                deployData);
            // check whether the same pair exists
            require(underlyings[index] == address(0), "underlying exists");

            address beacon = beacons[marketType];
            underlying = address(new SynFuturesV2UnderlyingProxy{salt : index}(beacon));
            // check whether the pair deployed is the same as specified in deployData
            require(underlying == expectedUnderlying, "underlying addr mismatch");

            delete parameters;
            underlyings[index] = underlying;
            underlyingsIndex.push(index);

            IObserver(observer).addUnderlying(underlying);
        }

        Types.QuoteInfo memory quoteInfo = IUnderlying(underlying).quoteInfo();
        {
            (uint amount) = abi.decode(initializeData, (uint));
            // approve underlying as underlying is created
            IERC20(quoteInfo.quote).approve(underlying, LibMathUnsigned.
                POSITIVE_INT256_MAX());
            ...
        }

        ...
    }
```

Listing 3.3: `SynFuturesV2Router::_createUnderlying()`

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

Status The issue has been addressed by the following commit: `2fe40d9`.

3.3 Suggested Addition Of rescueToken() To SynFuturesV2Underlying

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SynFuturesV2Underlying
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

By design, the SynFutures V2 protocol supports multiple SynFuturesV2Underlying contracts and holds various types of underlying tokens. From past experience with current popular DeFi protocols, e.g., YFI/Curve, we notice that there is always non-trivial possibilities that non-related tokens may be accidentally sent to the contract(s). To avoid unnecessary loss of protocol users, we suggest to add the support of rescuing remaining tokens. This is a design choice for the benefit of protocol users.

Recommendation Add the support of rescuing remaining tokens in SynFuturesV2Underlying. An example addition is shown below:

```
function rescueToken(address _token, address _to, uint256 _amount) external
    onlyOwner {
    require(_token != quoteInfo.quote, "Should not withdraw staking Token");
    IERC20(_token).safeTransfer(_to, _amount);
    emit Recovered(_token, _to, _amount);
}
```

Listing 3.4: SynFuturesV2Underlying::rescueToken()

Status The issue has been confirmed by the team.

3.4 Suggested Reentrancy Protection In Current Implementation

- ID: PVE-004
- Severity: Undetermined
- Likelihood: Undetermined
- Impact: Undetermined
- Target: `SynFuturesV2Router/SynFuturesV2Underlying`
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

In the `SynFuturesV2Router` contract, we notice the `deposit()` routine is used to deposit the supported assets into the `SynFutures V2` protocol, and the `trade()` routine is used to buy `Long` or `Short` positions. While examining their logic, we notice the `deposit()` routine is under the reentrancy protection, however, the `trade()` routine is not.

To elaborate, we show below the related code snippet of the `SynFuturesV2Router` contract. Within the internal `_deposit()` routine (which is called inside the `deposit()` routine), we notice `IERC20(quoteInfo.quote).safeTransferFrom(msg.sender, address(this), tokenAmount)` (line 208) is called to transfer the token into the `SynFuturesV2Router` contract. If the `quoteInfo.quote` faithfully implements the ERC777-like standard, then the `trade()` routine is exposed to potential reentrancy vulnerability and this risk needs to be properly mitigated. Although we also do not know how a malicious actor can exploit this vulnerability to earn profit. After internal discussion, we consider it is necessary to bring this vulnerability up to the team. We suggest to use the `ReentrancyGuard::nonReentrant` modifier to protect all the public routines at the whole protocol level.

```
187     function trade(address underlying, uint expiry, int size, uint limitPrice, uint
            deadline) public payable {
188         IUnderlying(underlying).trade(msg.sender, expiry, size, limitPrice, deadline);
189     }
190
191     ...
192
193     function deposit(address underlying, address trader, uint amount) public payable
            nonReentrant {
194         Types.QuoteInfo memory quoteInfo = IUnderlying(underlying).quoteInfo();
195         _deposit(underlying, quoteInfo, trader, amount);
196         return;
197     }
198
199     ...
200
201
```

```
    function _deposit(
        address underlying, Types.QuoteInfo memory quoteInfo, address trader, uint amount
    ) internal returns (uint) {
        require(msg.value == 0, "invalid msg.value");
        uint tokenAmount = amount / quoteInfo.scaler;//10**(18 - token decimals)
        IERC20(quoteInfo.quote).safeTransferFrom(msg.sender, address(this), tokenAmount);

        amount = tokenAmount * quoteInfo.scaler;
        IUnderlying(underlying).deposit(trader, amount);
        return amount;
    }
```

Listing 3.5: `SynFuturesV2Router::trade()&&deposit()`

Recommendation Apply the non-reentrancy protection in all public routines.

Status The issue has been addressed by the following commit: `72e9f18`.

3.5 Trust Issue Of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `SynFutures V2` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring the price oracle related parameters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
    function setQuoteParam(address _quote, Types.QuoteParam calldata _param) public onlyOwner {
        quotes[_quote] = _param;
        emit SetQuoteParam(_quote, _param);
    }

    function setUnderlyingInfo(address _underlying, UnderlyingInfo calldata _info) public onlyOwner {
       _setUnderlyingInfo(_underlying, _info);
    }

```

```
    function _setUnderlyingInfo(address _underlying, UnderlyingInfo memory _info)
        internal {
        require(_info.underlyingType <= 3, "unsupported underlyingType");
        require(address(_info.feeder.aggregator0) != address(0), "invalid feeder");
        underlyingsInfo[_underlying] = _info;
        emit SetUnderlyingInfo(_underlying, _info);
    }
```

Listing 3.6: `ChainlinkMarket`

We emphasize that the privilege assignment may be necessary and consistent with the protocol design. However, it is worrisome if the `owner` is not governed by a `DAO`-like structure. Note that a compromised account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

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 The issue has been confirmed by the team.

4 | Conclusion

In this audit, we have analyzed the `SynFutures V2` design and implementation. `SynFutures V2` is a decentralized derivatives platform. In comparison to `SynFutures V1`, `SynFutures V2` provides a more streamlined, easier-to-navigate user experience for both traders and liquidity providers, as well as more trading products and features designed with increased capital efficiency. For traders, `SynFutures V2` introduces `Perpetual Futures`, which is a never-ending futures contract with native permissionless listing, guaranteed price convergence to spot index, and a forward-looking funding mechanism. For LPs, `SynFutures V2` will be the first AMM-based derivatives protocol to natively incorporate ranged liquidity provision and limit orders, in addition to the vanilla AMM liquidity provision. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

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

[4] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

[7] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

[8] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

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

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

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

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