SMART CONTRACT AUDIT REPORT

for

JPEG'd Protocol

Prepared By: Yiqun Chen

PeckShield

December 31, 2021

Document Properties

Client	JPEG'd Protocol
Title	Smart Contract Audit Report
Target	JPEG'd
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 18, 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 JPEG'd 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 JPEG'd

JPEG'd introduces a new DeFi primitive - non-fungible debt positions (NFDP). It creates a permissionless and trustless collateralized debt position with NFTs as the collateral so that NFT holders can obtain liquidity. In particular, NFT-based collateral can be deposited into a vault to mint PUSD, which joins a basket of other tokens to peg its value as close to $1 as possible at all times. Additionally, incentives will be offered to liquidity providers to add liquidity to the pool. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of JPEG'd

Item	Description
Name	JPEG'd Protocol
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/iceboxup/jpegd.git (679bb3c)

1.2 About PeckShield

PeckShield Inc. [13] 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 [12]:

- 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 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 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

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) [11], 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 JPEG'd 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	0	
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 2 medium-severity vulnerabilities and 3 low-severity vulnerabilities.

Table 2.1: Key JPEG'd Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible Costly JPEGD-yVault LP From Improper Initialization	Time and State	Confirmed
PVE-002	Low	Asset Consistency Check Between Vault And Strategy	Coding Practices	Confirmed
PVE-003	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Confirmed
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Duplicate Pool Detection and Prevention	Business Logic	Confirmed

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 Possible Costly JPEGD-yVault LP From Improper Initialization

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: YVault
- Category: Time and State [7]
- CWE subcategory: CWE-362 [4]

Description

The JPEG'd protocol allows users to deposit fungible assets into autocompounding strategy contracts (e.g. StrategyPUSDConvex). The users will get in return JPEGD-wrapped tokens to represent the vault pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the deposit() routine, which is used to deposit the supported assets and get respective pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    /// @notice Allows users to deposit `token`. Contracts can't call this function
    /// @param _amount The amount to deposit
    function deposit(uint256 _amount) public noContract(msg.sender) {
        require(_amount > 0, "INVALID_AMOUNT");
        uint256 balanceBefore = balance();
        token.safeTransferFrom(msg.sender, address(this), _amount);
        uint256 supply = totalSupply();
        uint256 shares;
        if (supply == 0) {
            shares = _amount;
        } else {
            //balanceBefore can't be 0 if totalSupply is > 0
            shares = (_amount * supply) / balanceBefore;
```

```
151         }
152         _mint(msg.sender, shares);
153 
154         emit Deposit(msg.sender, _amount);
155     }
```

Listing 3.1: YVault::deposit()

Specifically, when the pool is being initialized (line 146), the share value directly takes the value of `shares = _amount` (line 147), which is manipulatable by the malicious actor. As this is the first deposit, the current total supply equals the calculated `shares = 1 WEI`. With that, the actor can further donate a huge amount of assets with the goal of making the pool token extremely expensive.

An extremely expensive pool token can be very inconvenient to use as a small number of $1WEI$ may denote a large value. Furthermore, it can lead to precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status This issue has been confirmed. The team will exercise extra caution in properly initializing the pool.

3.2 Asset Consistency Check Between Vault And Strategy

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Controller`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1099 [1]

Description

In JPEG'd, there is a one-to-one mapping between a `vault` and its `strategy`. To properly link a `vault` with its `strategy`, it is natural for the two to operate on the same underlying asset. If these two have different underlying assets, the link should not be successful. If we examine the `setStrategy()` routine in the `Controller` contract, this routine allows for dynamic binding of the `vault` with a new `strategy` (line 97). A successful binding needs to satisfy a number of requirements. One specific one is shown as follows: `require(IVault(vaults[_token]).token()== IStrategy(_strategy).want())`. Apparently, this requirement guarantees the consistency of the underlying asset between the `vault` and its associated `strategy`.

```
82    function setStrategy(IERC20 _token, IStrategy _strategy)
83        external
84        onlyRole(STRATEGIST_ROLE)
85    {
86        require(
87            approvedStrategies[_token][_strategy] == true,
88            "STRATEGY_NOT_APPROVED"
89        );
90
91        IStrategy _current = strategies[_token];
92        if (address(_current) != address(0)) {
93            //withdraw all funds from the current strategy
94            _current.withdrawAll();
95            _current.withdraw(address(jpeg));
96        }
97        strategies[_token] = _strategy;
98    }
```

Listing 3.2: Controller :: setStrategy()

However, if we examine the `constructor()` of current `strategy` contracts (e.g., `StrategyPUSDConvex`), the requirement of having the same underlying asset is not enforced. A new `strategy` deployment with an ill-provided list of arguments with an unmatched underlying asset may cause unintended consequences, including possible asset loss. With that, we suggest to maintain an invariant by ensuring the consistency of the underlying asset when a new `strategy` is being deployed or linked.

Recommendation Ensure the consistency of the underlying asset between the `vault` and its associated `strategy`. An example revision is shown below.

```
    function setStrategy(IERC20 _token, IStrategy _strategy)
        external
        onlyRole(STRATEGIST_ROLE)
    {
        require(
            approvedStrategies[_token][_strategy] == true,
            "STRATEGY_NOT_APPROVED"
        );
        require(vaults[_token]).token() == IStrategy(_strategy).want(), "!asset")

        IStrategy _current = strategies[_token];
        if (address(_current) != address(0)) {
            //withdraw all funds from the current strategy
```

```
            _current.withdrawAll();
            _current.withdraw(address(jpeg));
        }
        strategies[_token] = _strategy;
    }
```

Listing 3.3: Revised Controller::setStrategy()

Status The issue has been acknowledged.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: Multiple Contracts
- Category: Numeric Errors [10]
- CWE subcategory: CWE-190 [2]

Description

SafeMath is a widely-used Solidity math library that is designed to support safe math operations by preventing common overflow or underflow issues when working with uint256 operands. While it indeed blocks common overflow or underflow issues, the lack of float support in Solidity may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (mul) and division (div) are involved.

In particular, we use the NFTVault::_calculateAdditionalInterest() as an example. This routine is used to calculate the additional global interest since last time the contract's state was updated.

```
    function _calculateAdditionalInterest() internal view returns (uint256) {
        // Number of seconds since {accrue} was called
        uint256 elapsedTime = block.timestamp - totalDebtAccruedAt;
        if (elapsedTime == 0) {
            return 0;
        }

        if (totalDebtAmount == 0) {
            return 0;
        }

        // Accrue interest
        uint256 interestPerYear = (totalDebtAmount *
            settings.debtInterestApr.numerator) /
```

```
567                 settings.debtInterestApr.denominator;
568         uint256 interestPerSec = interestPerYear / 365 days;

570         return elapsedTime * interestPerSec;
571     }
```

Listing 3.4: NFTVault:: _calculateAdditionalInterest ()

We notice the calculation of the final result (line 570) involves mixed multiplication and devision. For improved precision, it is better to calculate the multiplication before the division, i.e., `interestPerYear*interestPerSec/365 days`. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible. Note the `ERC20Vault::_getDebtInterest()` routine can be similarly improved.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been confirmed.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `JPEG'd` protocol, the privileged owner account plays a critical role in governing and regulating the system-wide operations (e.g., `vault/strategy` addition, reward adjustment, and parameter setting). It also has the privilege to control or govern the flow of assets for investment or full withdrawal among the three components, i.e., `vault`, `controller`, and `strategy`.

With great privilege comes great responsibility. Our analysis shows that the governance account is indeed privileged. In the following, we examine the current privilege management graph in the `JPEG'd` protocol (Figure 3.1).

We emphasize that the privilege assignment among `vault`, `controller`, and `strategy` is properly administrated. However, it is worrisome if the `governance` is not governed by a `DAO`-like structure. The discussion with the team has confirmed that the governance will be managed by a multi-sig account.

We point out that a compromised `governance` account would allow the attacker to add a malicious `controller` to steal all funds whenever the `farm()` call is made. It could also allow for the dynamic addition of a new malicious `strategy`, which directly undermines the assumption of the `JPEG'd` protocol.

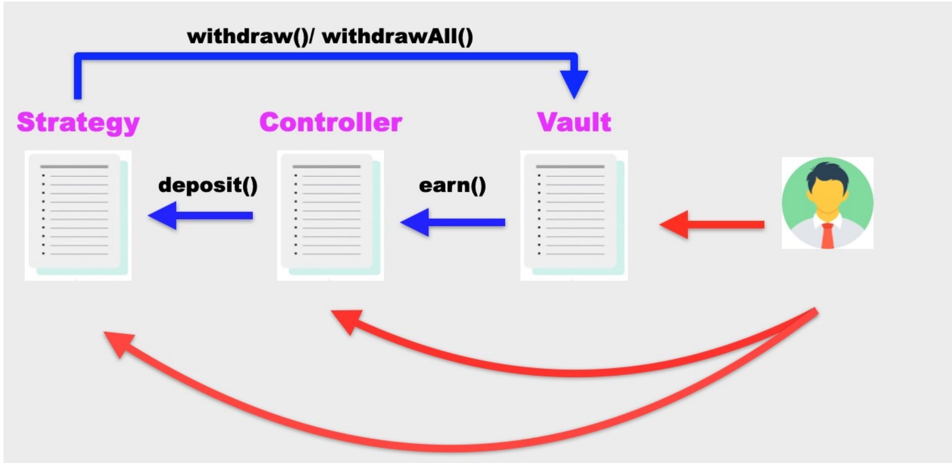

Figure 3.1: The Privilege Management Chain in JPEG'd

Recommendation Promptly transfer the governance privilege to the intended DAO-like governance contract. And 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/controller privileges.

3.5 Duplicate Pool Detection and Prevention

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: LPFarming
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [5]

Description

The JPEG'd protocol provides incentive mechanisms that reward the staking of supported assets with certain reward tokens. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. Each pool has its allocPoint*100%/totalAllocPoint share of scheduled rewards and the rewards for stakers are proportional to their share of LP tokens in the pool.

In current implementation, there are a number of concurrent pools that share the rewarded tokens and more can be scheduled for addition (via a proper governance procedure). To accommodate these new pools, the design has the necessary mechanism in place that allows for dynamic additions of new staking pools that can participate in being incentivized as well.

The addition of a new pool is implemented in `add()`, whose code logic is shown below. It turns out it did not perform necessary sanity checks in preventing a new pool but with a duplicate token from being added. Though it is a privileged interface (protected with the modifier `onlyOwner`), it is still desirable to enforce it at the smart contract code level, eliminating the concern of wrong pool introduction from human omissions.

```
    function add(uint256 _allocPoint, IERC20 _lpToken) external onlyOwner {
        _massUpdatePools();

        uint256 lastRewardBlock = _blockNumber();
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accRewardPerShare: 0
            })
        );
    }
```

Listing 3.5: `LPFarming::add()`

Recommendation Detect whether the given pool for addition is a duplicate of an existing pool. The pool addition is only successful when there is no duplicate.

```
    function checkPoolDuplicate(IERC20 _lpToken) public {
        uint256 length = poolInfo.length;
        for (uint256 pid = 0; pid < length; ++pid) {
            require(poolInfo[_pid].lpToken != _lpToken, "add: existing pool?");
        }
    }

    function add(uint256 _allocPoint, IERC20 _lpToken) external onlyOwner {
        _massUpdatePools();

        checkPoolDuplicate(_lpToken);
        uint256 lastRewardBlock = _blockNumber();
        totalAllocPoint = totalAllocPoint + _allocPoint;
        poolInfo.push(
            PoolInfo({
                lpToken: _lpToken,
                allocPoint: _allocPoint,
                lastRewardBlock: lastRewardBlock,
                accRewardPerShare: 0
            })
        );
```

```
    }
```

Listing 3.6: Revised `LPFarming::add()`

We point out that if a new pool with a duplicate LP token can be added, it will likely cause a havoc in the distribution of rewards to the pools and the stakers.

Status The issue has been acknowledged.

4 Conclusion

In this audit, we have analyzed the design and implementation of the JPEG'd protocol, which introduces a new DeFi primitive - non-fungible debt positions (NFDP). This new primitive allows NFT holders to use NFTs collateral to mint the stablecoin PUSD. 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-1099: Inconsistent Naming Conventions for Identifiers. https://cwe.mitre.org/data/definitions/1099.html.

[2] MITRE. CWE-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

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

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

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