



SMART CONTRACT AUDIT REPORT

for

Preon Finance



Prepared By: Xiaomi Huang

PeckShield
May 24, 2024

Document Properties

Client	Preon
Title	Smart Contract Audit Report
Target	Preon
Version	1.0
Author	Xuxian Jiang
Auditors	Jinzhuo Shen, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 24, 2024	Xuxian Jiang	Final Release
1.0-rc	May 6, 2024	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1	Introduction	4
1.1	About Preon	4
1.2	About PeckShield	5
1.3	Methodology	5
1.4	Disclaimer	7
2	Findings	10
2.1	Summary	10
2.2	Key Findings	11
3	Detailed Results	12
3.1	Revisited notifyRewardAmount() Logic in Bribe	12
3.2	Improper totalSupply Checkpoint in RewardsDistributorV2	13
3.3	Timely Reward Resume Upon Reviving Gauges	14
3.4	Revisited getClaimablePREON() Logic in LockedPREON	15
3.5	Improved Validation on Protocol Parameters	16
3.6	Trust Issue of Admin Keys	17
4	Conclusion	19
	References	20

1 | Introduction

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

`Preon` is a decentralized protocol that allows `Ether` or liquid staking derivatives (LSDs) holders to obtain maximum liquidity against their collateral without paying interest. It took inspiration from `Gravita` and is designed to be multi-collateral. Each position can have only one collateral type and it is linked to a specific stability pool. Also, the protocol allows cross-collateral positions, linked to a single stability pool. Note the protocol-wide debt token is called `STAR`, while the governance token is `PREON`. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of Preon

Item	Description
Name	Preon
Type	Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 24, 2024

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note the given repository has a number of contract file and directories and this audit only covers the following contracts: `Voter.sol`, `VeArtProxy.sol`, `RewardDistributorV2.sol`, `Minter.sol` (under `VotingEscrow`), `PreonOFT.sol`, `LockedPREON.sol`, `CommunityIssuance.sol`, `CrossChainLocker.sol`

, CrosschainMessages/CrossChainConfig.sol, CrosschainMessages/[library](#)/CrossChainMessages.sol (under PREON), TokenTreasury.sol, AeroOptionsTokenSwapper.sol, OptionsToken.sol, Oracles/PreonOracle.sol (under OptionsToken), UniV2/UniV2LPOracle.sol, Pendle/PendleLpUsdOracle.sol, and Pendle/PendlePtUsdOracle.sol (under Oracles).

- <https://github.com/PreonMoney/preon-contracts.git> (b6add76)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (<https://t.me/peckshield>), Twitter (<http://twitter.com/peckshield>), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

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 accordingly classified into four categories, 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) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings. 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

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.



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.



2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the design and implementation of the `Preon` protocol smart contracts. 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	2	
Low	4	
Informational	0	
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 and 4 low-severity vulnerabilities.

Table 2.1: Key Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Revisited <code>notifyRewardAmount()</code> Logic in Bribe	Business Logic	Confirmed
PVE-002	Low	Improper <code>totalSupply</code> Checkpoint in <code>RewardsDistributorV2</code>	Coding Practices	Resolved
PVE-003	Low	Timely Reward Resume Upon Reviving Gauges	Business Logic	Resolved
PVE-004	Low	Revisited <code>getClaimablePREON()</code> Logic in <code>LockedPREON</code>	Business Logic	Resolved
PVE-005	Low	Improved Validation on Protocol Parameters	Business Logic	Resolved
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	Mitigated

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 Revisited notifyRewardAmount() Logic in Bribe

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

Description

The Preon protocol creates a gauge for the supported pool and the created gauge can be bribed with additional rewards. While reviewing the bribe logic, we notice the reward addition for bribe is distributed in a wrong bribe period.

In the following, we show the implementation of the related `notifyRewardAmount()` routine, which basically adds new reward starting next bribe period. However, it comes to our attention that the internal variable `adjustedTstamp` (line 340) is computed as the start of current bribe period, which should add one `DURATION` to yield the start of next bribe period.

```
324     function notifyRewardAmount(  
325         address token,  
326         uint256 amount  
327     ) external nonReentrant {  
328         require(amount > 0);  
329         if (!isReward[token]) {  
330             require(  
331                 IVoter(voter).isWhitelisted(token),  
332                 "bribe tokens must be whitelisted"  
333             );  
334             require(  
335                 rewards.length < MAX_REWARD_TOKENS,  
336                 "too many rewards tokens"  
337             );  
338         }  
339         // bribes kick in at the start of next bribe period
```

```

340     uint256 adjustedTstamp = getEpochStart(block.timestamp);
341     uint256 epochRewards = tokenRewardsPerEpoch[token][adjustedTstamp];

343     _safeTransferFrom(token, msg.sender, address(this), amount);
344     tokenRewardsPerEpoch[token][adjustedTstamp] = epochRewards + amount;

346     periodFinish[token] = adjustedTstamp + DURATION;

348     if (!isReward[token]) {
349         isReward[token] = true;
350         rewards.push(token);
351     }

353     emit NotifyReward(msg.sender, token, adjustedTstamp, amount);
354 }

```

Listing 3.1: Bribe::notifyRewardAmount()

Recommendation Revise the above routine to kick in new bribes at the start of next bribe period.

Status This issue has been confirmed.

3.2 Improper totalSupply Checkpoint in RewardsDistributorV2

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: RewardsDistributorV2
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

To incentivize the long-time stakers with inflation, the Preon protocol has a built-in RewardsDistributorV2 contract to calculate inflation and adjust emission balances accordingly. While reviewing current implementation, we notice the logic to checkpoint total supply needs to be improved.

To elaborate, we show below the related `_checkpoint_total_supply()` routine. This routine is proposed to checkpoint the total supply at the current timestamp. It comes to our attention that the calculation of `ve_supply[t]` may yield a negative `int256`(`pt.bias - pt.slope * dt`) (lines 93–96). And when it is negative, the type cast to `uint` makes it positive, which leads to an incorrect calculation of total supply (in this case, the resulting total supply should be 0.).

```

75     function _checkpoint_total_supply() internal {
76         address ve = voting_escrow;
77         uint t = time_cursor;

```

```

78         uint rounded_timestamp = (block.timestamp / WEEK) * WEEK;
79         IVotingEscrow(ve).checkpoint();
80
81         for (uint i = 0; i < 20; i++) {
82             if (t > rounded_timestamp) {
83                 break;
84             } else {
85                 uint epoch = _find_timestamp_epoch(ve, t);
86                 IVotingEscrow.Point memory pt = IVotingEscrow(ve).point_history(
87                     epoch
88                 );
89                 int128 dt = 0;
90                 if (t > pt.ts) {
91                     dt = int128(int256(t - pt.ts));
92                 }
93                 ve_supply[t] = Math.max(
94                     uint(int256(pt.bias - pt.slope * dt)),
95                     0
96                 );
97             }
98             t += WEEK;
99         }
100
101         time_cursor = t;
102     }

```

Listing 3.2: RewardsDistributorV2::_checkpoint_total_supply()

Recommendation Revise the above routine to properly checkpoint the total supply.

Status This issue has been fixed in the following commit: 6ce6cba.

3.3 Timely Reward Resume Upon Reviving Gauges

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Voter
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The Preon protocol creates a gauge for the supported pool and the created gauge can be killed or revived based on the community needs. While reviewing the current gauge-reviving logic, we notice a revived gauge needs to be properly re-initialized!

To elaborate, we show below the related `reviveGauge()` routine. While it properly marks the gauge alive (line 281), it does not properly set the associated `supplyIndex`, i.e., `supplyIndex[_gauge] = index`, making it still eligible for rewards even before its revive.

```

278     function reviveGauge(address _gauge) external {
279         require(msg.sender == emergencyCouncil, "not emergency council");
280         require(!isAlive[_gauge], "gauge already alive");
281         isAlive[_gauge] = true;
282         emit GaugeRevived(_gauge);
283     }

```

Listing 3.3: `Voter::reviveGauge()`

Recommendation Revise the above logic to properly revive a current gauge.

Status This issue has been fixed in the following commit: `0d317634`.

3.4 Revisited `getClaimablePREON()` Logic in `LockedPREON`

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `LockedPREON`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Preon` protocol has a vesting schedule contract `LockedPREON`. This contract is reserved for linear vesting to the team members and the advisors team. In the process of examining the vesting logic, we notice current approach to calculate claimable `PREON` has a possible arithmetic underflow issue.

In particular, we show below the implementation of the related routine. It has a rather straightforward logic in calculating vested tokens. We notice that in the branch of `entityRule.startVestingDate <= block.timestamp` and `block.timestamp < entityRule.endVestingDate` (line 134), the claimable amount is computed as `((entityRule.totalSupply / TWO_YEARS)* (block.timestamp - entityRule.createdDate)) - entityRule.claimed`, which may be reverted when the first element of `((entityRule.totalSupply / TWO_YEARS)* (block.timestamp - entityRule.createdDate))` is smaller than the second element of `entityRule.claimed`. Note it is possible when the specific vesting schedule has been adjusted via `lowerEntityVesting()` with a smaller `entityRule.totalSupply`.

```

124     function getClaimablePREON(
125         address _entity
126     ) public view returns (uint256 claimable) {
127         Rule memory entityRule = entitiesVesting[_entity];
128         claimable = 0;

```

```

129
130     if (entityRule.startVestingDate > block.timestamp) return claimable;
131
132     if (block.timestamp >= entityRule.endVestingDate) {
133         claimable = entityRule.totalSupply - entityRule.claimed;
134     } else {
135         claimable =
136             ((entityRule.totalSupply / TWO_YEARS) *
137              (block.timestamp - entityRule.createdDate)) -
138             entityRule.claimed;
139     }
140
141     return claimable;
142 }

```

Listing 3.4: LockedPREON::getClaimablePREON()

Recommendation Revise the above logic to properly compute the claimable token amount.

Status This issue has been fixed in the following commit: 756c603.

3.5 Improved Validation on Protocol Parameters

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Minter
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The Preon protocol is no exception. Specifically, if we examine the Minter contract, it has defined a number of protocol-wide risk parameters, such as `teamRate` and `communityRate`. In the following, we show the corresponding routines that configure their values.

```

122     function setTeamRate(uint256 _teamRate) external {
123         if (msg.sender != team) revert Minter_notTeam();
124         if (_teamRate >= MAX_TEAM_RATE) revert Minter_maxTeamRate();
125         teamRate = _teamRate;
126     }
127
128     function setCommunityRate(uint256 _rate) external {
129         if (msg.sender != team) revert Minter_notTeam();
130         if (_rate >= PRECISION) revert Minter_setCommunityRate();
131         communityRate = _rate;
132     }

```



```

133
134     function setRebase(uint _rebase) external {
135         if (msg.sender != team) revert Minter_notTeam();
136         if (_rebase >= PRECISION) revert Minter_setRebaseRate();
137         REBASEMAX = _rebase;
138     }

```

Listing 3.5: Minter::setTeamRate()/setCommunityRate()/setRebase()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, an unlikely mis-configuration of `communityRate` may compromise the token distribution process, hence hurting the adoption of the protocol. In particular, the above setters can be improved by enforcing the following requirement: `require(teamRate + _rate <= PRECISION)`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status This issue has been fixed in the following commit: `1cda112`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the Preon protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure parameters, manage contracts, and upgrade proxies). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```

136     function setOracle(IPreonOracle oracle_) external onlyOwner {
137         oracle = oracle_;
138         emit SetOracle(oracle_);}
139
140     function setTreasury(ITokenTreasury treasury_) external onlyOwner {}
141

```

```

142     function setVoter(address _voter) external onlyOwner {
143         voter = _voter;
144
145         emit SetVoter(_voter);}
146
147     function setMinter(address _acc, bool _isMinter) external onlyOwner {
148         isMinter[_acc] = _isMinter;
149
150         emit SetMinter(_acc, _isMinter);}
151
152     function setDiscount(uint256 _discount) external onlyOwner {
153         if (_discount >= MAX_DISCOUNT _discount == MIN_DISCOUNT)
154             revert OptionsToken_InvalidDiscount();
155         discount = _discount;
156         emit SetDiscount(_discount);}
157
158     function setVotingEscrow(IVePreon _votingEscrow) external onlyOwner {
159         votingEscrow = _votingEscrow;
160
161         emit SetVeToken(address(_votingEscrow));}
162
163     function pause() external onlyOwner {
164         _pause();}
165
166     function unpause() external onlyOwner {
167         _unpause();

```

Listing 3.6: Example Privileged Functions in OptionsToken

Note that if the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation.

Moreover, it should be noted that current contracts may have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

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 mitigated as the team makes use of a multisig to act as the privileged owner.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Preon` protocol, which is a decentralized protocol that allows `Ether` or liquid staking derivatives (LSDs) holders to obtain maximum liquidity against their collateral without paying interest. It took inspiration from `Gravita` and is designed to be multi-collateral. Each position can have only one collateral type and it is linked to a specific stability pool. Also, the protocol allows cross-collateral positions, linked to a single stability pool. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.



References

- [1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. <https://cwe.mitre.org/data/definitions/1126.html>.
- [2] MITRE. CWE-287: Improper Authentication. <https://cwe.mitre.org/data/definitions/287.html>.
- [3] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. <https://cwe.mitre.org/data/definitions/841.html>.
- [4] MITRE. CWE CATEGORY: 7PK - Security Features. <https://cwe.mitre.org/data/definitions/254.html>.
- [5] MITRE. CWE CATEGORY: Bad Coding Practices. <https://cwe.mitre.org/data/definitions/1006.html>.
- [6] MITRE. CWE CATEGORY: Business Logic Errors. <https://cwe.mitre.org/data/definitions/840.html>.
- [7] MITRE. CWE VIEW: Development Concepts. <https://cwe.mitre.org/data/definitions/699.html>.
- [8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.
- [9] PeckShield. PeckShield Inc. <https://www.peckshield.com>.