SMART CONTRACT AUDIT REPORT

for

Alpaca's Delta Neutral Vault

Prepared By: Patrick Liu

PeckShield

March 1, 2022

Document Properties

Client	Alpaca Finance Protocol
Title	Smart Contract Audit Report
Target	Delta Neutral Vault
Version	1.0
Author	Xuxian Jiang
Auditors	Xuxian Jiang, Jing Wang
Reviewed by	Patrick Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	March 1, 2022	Xuxian Jiang	Final Release
1.0-rc	February 24, 2022	Xuxian Jiang	Release Candidate

Contact

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

Name	Patrick Liu
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the the Alpaca Finance Protocol regarding the support of a new Delta Neutral Vault and its associated workers, 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 Alpaca

The Alpaca Finance Protocol is a leveraged yield farming and leveraged liquidity providing protocol running on Binance Smart Chain (BSC). The audited implementation extends the previous version by adding the support of a new Delta Neutral Vault and its associated workers, including DeltaNeutralPancakeWorker02 and DeltaNeutralMdexWorker02. Note the Delta Neutral Vault is designed to take a long and short position in an asset at the same time to cancel out the effect on the outstanding portfolio when the asset's price moves. The basic information of the audited contracts is as follows:

Table 1.1: Basic Information of the audited protocol

Item	Description
Name	Alpaca Finance Protocol
Website	https://www.alpacafinance.org/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	March 1, 2022

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

this audit:

- https://github.com/alpaca-finance/bsc-alpaca-contract.git (aa26c36)

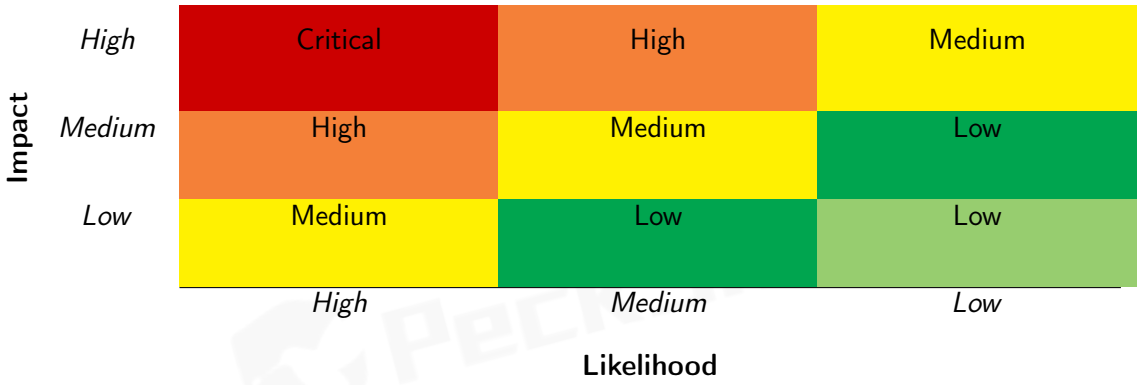
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/alpaca-finance/bsc-alpaca-contract.git (f5f93a2)

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



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 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) [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.

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 implementation of the `Alpaca Finance Protocol` regarding the support of a new `Delta Neutral Vault` and its associated `workers`. 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	0	
Total	3	

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 and 2 low-severity vulnerabilities.

Table 2.1: Key Audit Findings of Delta Neutral Vault Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-002	Low	Accommodation of approve() Idiosyncrasies	Coding Practices	Resolved
PVE-003	Low	Potential Sandwich Attacks For Reduced Returns	Time and State	Resolved

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Trust Issue of Admin Keys

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

The new `DeltaNeutralVault` is developed based on the previous `Alpaca Vaults`. And there is a privileged `owner` account that plays a critical role in governing and regulating the new vault operations (e.g., parameter setting and worker adjustment). 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 their related privileged accesses in current contracts.

To elaborate, we show below the example privileged operations in `DeltaNeutralVaultConfig`. This routine contract defines various configuration parameters used in the new vault and most of them are security-sensitive to the entire protocol.

```
191   function setwhitelistedReinvestors(address[] calldata _callers, bool _ok) external
          onlyOwner {
192     for (uint256 _idx = 0; _idx < _callers.length; _idx++) {
193       whitelistedReinvestors[_callers[_idx]] = _ok;
194       emit LogSetWhitelistedReinvestors(msg.sender, _callers[_idx], _ok);
195     }
196   }
197
198   /// @notice Set leverage level.
199   /// @dev Must only be called by owner.
200   /// @param _newLeverageLevel The new leverage level to be set. Must be >= 3
201   function setLeverageLevel(uint8 _newLeverageLevel) external onlyOwner {
202     if (_newLeverageLevel < MIN_LEVERAGE_LEVEL) {
```

```
203         revert LeverageLevelTooLow();
204       }
205       leverageLevel = _newLeverageLevel;
206       emit LogSetLeverageLevel(msg.sender, _newLeverageLevel);
207     }
208
209     /// @notice Set exempted fee callers.
210     /// @dev Must only be called by owner.
211     /// @param _callers addresses to be exempted.
212     /// @param _ok The new ok flag for callers.
213     function setFeeExemptedCallers(address[] calldata _callers, bool _ok) external
            onlyOwner {
214       for (uint256 _idx = 0; _idx < _callers.length; _idx++) {
215         feeExemptedCallers[_callers[_idx]] = _ok;
216         emit LogSetFeeExemptedCallers(msg.sender, _callers[_idx], _ok);
217       }
218     }
```

Listing 3.1: Example Privileged Operations in DeltaNeutralVaultConfig

We need to mention that it will be worrisome if the privileged `owner` account is a plain EOA account. The discussion with the team confirms that the `owner` account is currently managed by a timelock. A plan needs to be in place to migrate it under community governance. Note that 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed with the team. For the time being, it will be mitigated by a 24-hour timelock to balance efficiency and timely adjustment. After the protocol becomes stable, it is expected to migrate to a multi-sig account, and eventually be managed by community proposals for decentralized governance.

3.2 Accommodation of approve() Idiosyncrasies

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

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. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
194     /**
195     * @dev Approve the passed address to spend the specified amount of tokens on behalf
            of msg.sender.
196     * @param _spender The address which will spend the funds.
197     * @param _value The amount of tokens to be spent.
198     */
199     function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

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

207         allowed[msg.sender][_spender] = _value;
208         Approval(msg.sender, _spender, _value);
209     }
```

Listing 3.2: USDT Token Contract

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. Similarly, there is a safe version of `transfer()` as well, i.e., `safeTransfer()`.

```
38      /**
39       * @dev Deprecated. This function has issues similar to the ones found in
40       * {IERC20-approve}, and its usage is discouraged.
41       *
42       * Whenever possible, use {safeIncreaseAllowance} and
43       * {safeDecreaseAllowance} instead.
44       */
45      function safeApprove(
46          IERC20 token,
47          address spender,
48          uint256 value
49      ) internal {
50          // safeApprove should only be called when setting an initial allowance,
51          // or when resetting it to zero. To increase and decrease it, use
52          // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
53          require(
54              (value == 0)  (token.allowance(address(this), spender) == 0),
55              "SafeERC20: approve from non-zero to non-zero allowance"
56          );
57          _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
                spender, value));
58      }
```

Listing 3.3: `SafeERC20::safeApprove()`

In current implementation, if we examine the `DeltaNeutralVaultGateway::_swap()` routine that is designed to swap one token to another for position adjustment. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (lines 192 and 197). And it is suggested to perform twice: the first one resets the allowance to 0 and the second time sets the intended allowance amount.

```
174   function _swap(address _token, uint256 _swapAmount) internal {
175     IDeltaNeutralVaultConfig config = IDeltaNeutralVaultConfig(deltaNeutralVault.config
            ());
176     ISwapRouter _router = ISwapRouter(config.getSwapRouter());
177
178     address _stableToken = deltaNeutralVault.stableToken();
179     address _assetToken = deltaNeutralVault.assetToken();
180     address _nativeToken = config.getWrappedNativeAddr();
181
182     address[] memory _path = new address[](2);
183     address _token0 = _token;
184     address _token1 = _token == _stableToken ? _assetToken : _stableToken;
185     _path[0] = _token0;
186     _path[1] = _token1;
187
188     if (_token0 == _nativeToken  _token1 == _nativeToken) {
189       if (_token0 == _nativeToken) {
190         _router.swapExactETHForTokens{ value: _swapAmount }(0, _path, address(this),
                block.timestamp);
191       } else {
192         IERC20Upgradeable(_token).approve(address(_router), _swapAmount);
```

```
193
194         _router.swapExactTokensForETH(_swapAmount, 0, _path, address(this), block.
                timestamp);
195       }
196     } else {
197       IERC20Upgradeable(_token).approve(address(_router), _swapAmount);
198
199       _router.swapExactTokensForTokens(_swapAmount, 0, _path, address(this), block.
              timestamp);
200     }
201   }
```

Listing 3.4: `DeltaNeutralVaultGateway::_swap()`

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

Status This issue has been confirmed. If the protocol is deployed to only support ERC20-compliant tokens, this issue can be left unaddressed.

3.3 Potential Sandwich Attacks For Reduced Returns

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [3]

Description

As a yield farming and leveraged liquidity providing protocol, `Alpaca` has a constant need of performing token swaps between base and farming tokens. In the following, we examine the re-investment logic from the new `DeltaNeutralVault` contract.

To elaborate, we show below the `reinvest()` implementation. As the name indicates, it is designed to re-invest whatever this vault has earned to the position. In the meantime, the caller is whitelisted so that only authorized entities are allowed to invoke this function.

```
137   function reinvest(
138     uint8[] memory _actions,
139     uint256[] memory _values,
140     bytes[] memory _datas
141   ) external onlyReinvestors {
142     uint256 _alpacaBountyBps = config.alpacaBountyBps();
143
144     // 1.  claim reward from fairlaunch
```

```
145     uint256 _equityBefore = totalEquityValue();
146     uint256 _alpacaBefore = IERC20Upgradeable(alpacaToken).balanceOf(address(this));
147
148     // ddc63262 is a signature of harvest(uint256)
149     // low-level call prevent revert when harvest fail
150     config.fairLaunchAddr().call(abi.encodeWithSelector(0xddc63262, IVault(stableVault).
            fairLaunchPoolId()));
151     config.fairLaunchAddr().call(abi.encodeWithSelector(0xddc63262, IVault(assetVault).
            fairLaunchPoolId()));
152
153     uint256 _alpacaAfter = IERC20Upgradeable(alpacaToken).balanceOf(address(this));
154
155     // 2. collect alpaca bounty
156     uint256 _bounty = ((_alpacaBountyBps) * (_alpacaAfter - _alpacaBefore)) / MAX_BPS;
157     IERC20Upgradeable(alpacaToken).safeTransfer(config.getTreasuryAddr(), _bounty);
158
159     // 3. swap alpaca
160     address[] memory reinvestPath = config.getReinvestPath();
161     if (reinvestPath.length == 0) {
162       revert BadReinvestPath();
163     }
164
165     uint256 _rewardAmount = _alpacaAfter - _bounty;
166     IERC20Upgradeable(alpacaToken).approve(config.getSwapRouter(), _rewardAmount);
167     ISwapRouter(config.getSwapRouter()).swapExactTokensForTokens(
168       _rewardAmount,
169       0,
170       reinvestPath,
171       address(this),
172       block.timestamp
173     );
174
175     // 4. execute reinvest
176     {
177       _execute(_actions, _values, _datas);
178     }
179
180     // 5. sanity check
181     uint256 _equityAfter = totalEquityValue();
182     if (_equityAfter <= _equityBefore) {
183       revert UnsafePositionEquity();
184     }
185
186     emit LogReinvest(_equityBefore, _equityAfter);
187   }
```

Listing 3.5: `DeltaNeutralVault::reinvest()`

We notice the reward portion to `rewardAmount` is routed to `PancakeSwap` and the actual swap operation `swapExactTokensForTokens()` does not specify any restriction (with `amountOutMin=0`) on possible slippage and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for

this round of yielding.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running attack to better protect the interests of farming users.

Status This issue has been fixed in the following commit: `616cd92`.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Alpaca Finance Protocol`, which is a leveraged-yield farming protocol built on the `Binance Smart Chain`. With the new support of additional vaults and workers, the system makes it distinctive and valuable when compared with current yield farming offerings. The current code base is well organized and those identified issues are promptly confirmed and fixed.

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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.