SMART CONTRACT AUDIT REPORT

for

DIVA protocol

Prepared By: Xiaomi Huang

PeckShield

May 21, 2022

Document Properties

Client	DIVA protocol
Title	Smart Contract Audit Report
Target	DIVA protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 21, 2022	Xuxian Jiang	Final Release
1.0-rc	May 19, 2022	Luck Hu	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 DIVA protocol, we outline in this report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contract can be further improved due to the presence of the identified issues. This document outlines our audit results.

1.1 About DIVA protocol

DIVA protocol adopts the Diamond Standard (EIP-2535) and aims to be a decentralized piece of infrastructure that allows its users to create and settle fully customizable event-linked products (also known as derivatives) in a permissionless way. After depositing collateral, a user is issued two directionally reversed positions, referred to as long and short positions, that combined represent a claim on the deposited collateral, but when held in isolation exposes the user to the upside (via the long position) or downside (via the short position) of the underlying metric. The payoffs of long and short positions are zero-sum meaning that for every unit of collateral that the long position may gain, the short position will lose and vice versa. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of DIVA protocol

Item	Description
Name	DIVA protocol
Website	https://www.divaprotocol.io/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 21, 2022

In the following, we show the Git repositories of reviewed files and the commit hash values used

in this audit.

- https://github.com/divaprotocol/diva-contracts.git (34099a5)
- https://github.com/divaprotocol/whitelist.git (fba517f)

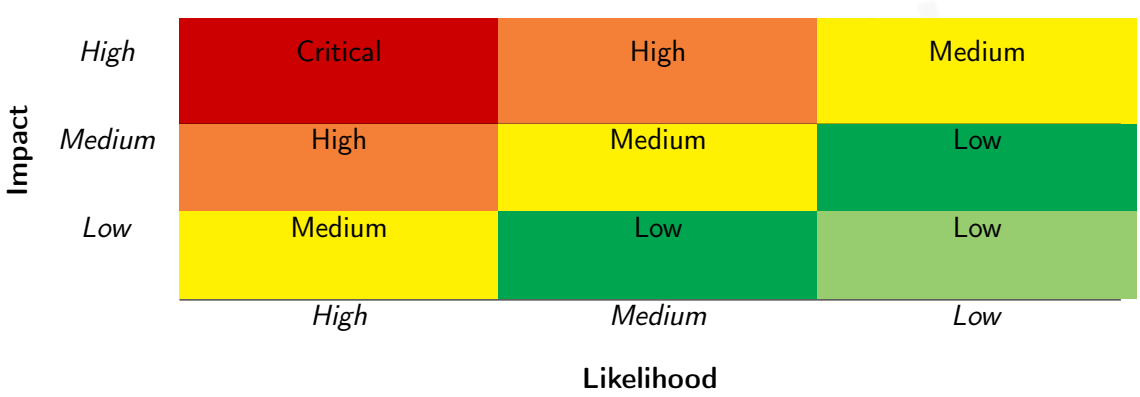
And these are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/divaprotocol/diva-contracts.git (443652d)
- https://github.com/divaprotocol/whitelist.git (fba517f)

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.

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

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

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.

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 implementation of the DIVA 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 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	1	■
Total	4	

We have so far identified a list of potential issues: some of them involve the coding practices, while others refer to the concerns of admin keys, etc. 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 (Medium/Low/Informational) need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 medium-severity vulnerability, 2 low-severity vulnerabilities, and 1 informational issue.

Table 2.1: Key DIVA protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of Function Arguments	Coding Practices	Fixed
PVE-002	Informational	Suggested Event Generation For redeemPositionToken()	Coding Practices	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-004	Low	Suggested immutable Usages For Gas Efficiency	Coding Practices	Fixed

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 Improved Validation on Function Arguments

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `PoolFacet`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [2]

Description

The `PoolFacet` contract provides a `createContingentPool()` routine to create a new contingent pool and mint short and long position tokens for users. At the beginning of the routine, it checks the input parameters for validity (lines 28-38). While examining the parameters validation in this routine, we notice the absence of the validation for a key pool parameter.

To elaborate, we show below the code snippet of the `PoolFacet::createContingentPool()` routine. As the name indicates, it is designed to accept user input and create a contingent pool for users. The pool has a key parameter `expiryTime` which represents the expiration time of the position tokens expressed as a unix timestamp in seconds. And the value of the reference asset observed at that point in time determines the payoffs for long and short position tokens. Per design, the user needs to give a value in the future for the `expiryTime`. However, this routine doesn't validate the `expiryTime`. As a result, the user may give an expiration date in the past which makes the created pool invalid. Based on this, we suggest to validate the `expiryTime` to be a reasonable value in the future.

```
14  function createContingentPool(PoolParams calldata _poolParams)
15        external
16        override
17        nonReentrant
18        returns (uint256)
19    {
20        // Get references to relevant storage slots
21        LibDiamond.PoolStorage storage ps = LibDiamond.poolStorage();
22        LibDiamond.GovernanceStorage storage gs = LibDiamond.governanceStorage();
```

```
23
24          // Create reference to collateral token corresponding to the provided pool Id
25          IERC20Metadata collateralToken = IERC20Metadata(_poolParams.collateralToken);
26
27          // Check validity of input parameters
28          require(bytes(_poolParams.referenceAsset).length > 0, "DIVA: no reference asset"
                );
29          require(_poolParams.floor <= _poolParams.inflection, "DIVA: floor > inflection")
                ;
30          require(_poolParams.cap >= _poolParams.inflection, "DIVA: cap < inflection");
31          require(_poolParams.dataProvider != address(0), "DIVA: data provider 0x0");
32          require(_poolParams.gradient <= 10**18, "DIVA: gradient > 1e18");
33          require(_poolParams.collateralAmount >= 10**6, "DIVA: collateral amount < 1e6");
                 // to reduce rounding errors
34          require(_poolParams.collateralAmount <= _poolParams.capacity, "DIVA: pool
                capacity exceeded");
35          require(
36              (collateralToken.decimals() <= 18) && (collateralToken.decimals() >= 6),
37              "DIVA: collateral token decimals > 18 or < 6"
38          );
39
40          // Increment `poolId` every time a new pool is created. Index
41          // starts at 1. No overflow risk when using compiler version >= 0.8.0.
42          ps.poolId++;
43          ...
44          // Store `Pool` struct in `pools` mapping for the newly generated `poolId`
45          ps.pools[ps.poolId] = LibDiamond.Pool(
46              _poolParams.floor,
47              _poolParams.inflection,
48              _poolParams.cap,
49              _poolParams.gradient,
50              _poolParams.collateralAmount,
51              0,                                      // finalReferenceValue
52              _poolParams.capacity,
53              block.timestamp,
54              address(_shortToken),
55              0,                                      // payoutShort
56              address(_longToken),
57              0,                                      // payoutLong
58              _poolParams.collateralToken,
59              _poolParams.expiryTime,
60              address(_poolParams.dataProvider),
61              gs.protocolFee,
62              gs.settlementFee,
63              LibDiamond.Status.Open,
64              _poolParams.referenceAsset
65          );
66
67          // Number of position tokens is set equal to the total collateral to
68          // standardize the max payout at 1.0.
69          _shortToken.mint(msg.sender, _poolParams.collateralAmount);
70          _longToken.mint(msg.sender, _poolParams.collateralAmount);
```

```
71
72          // Log pool creation
73          emit PoolIssued(ps.poolId, msg.sender, _poolParams.collateralAmount);
74
75          return ps.poolId;
76      }
```

Listing 3.1: `PoolFacet::createContingentPool()`

Recommendation Enforce the parameters validation in the `createContingentPool()` routine to ensure the input `expiryTime` is a reasonable value in the future.

Status This issue has been fixed by this commit: `8ccacfa`.

3.2 Suggested Event Generation For redeemPositionToken()

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `SettlementFacet`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [4]

Description

In `Ethereum`, the `event` is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an `event` is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. `Events` can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

While examining the events that reflect the position token dynamics in the `SettlementFacet::redeemPositionToken()` routine, we notice there is a lack of emitting an event to reflect the redeeming of the given position token. To elaborate, we show below the code snippet of the `SettlementFacet::redeemPositionToken()` routine.

```
197     function redeemPositionToken(
198         address _positionToken,
199         uint256 _amount
200     )
201         external
202         override
203         nonReentrant
204     {
205         // Get references to relevant storage slots
206         LibDiamond.PoolStorage storage ps = LibDiamond.poolStorage();
```

```
207         LibDiamond.GovernanceStorage storage gs = LibDiamond.governanceStorage();
208         ...

210         uint8 _decimals = (IERC20Metadata(_pool.collateralToken)).decimals();

212         // If status is "Confirmed", burn position tokens and return collateral to user
213         if (_pool.statusFinalReferenceValue == LibDiamond.Status.Confirmed) {
214             // Burn position tokens
215             _positionTokenInstance.burn(msg.sender, _amount);

217             uint256 _tokenPayoutAmount;

219             if (_positionToken == _pool.longToken) {
220                 _tokenPayoutAmount = _pool.payoutLong;   // net of fees
221             } else { // Can only be shortToken due to require statement at the beginning

223                 _tokenPayoutAmount = _pool.payoutShort;  // net of fees
224             }

226             // Calculate collateral amount to return
227             uint256 _amountToReturn = (_tokenPayoutAmount * _amount) / (10**uint256(
                    _decimals)); // decimal math with integers

229             // Return collateral to caller and reduce `collateralBalance` accordingly
230             LibDiamond._returnCollateral(
231                 _poolId,
232                 _pool.collateralToken,
233                 msg.sender,
234                 _amountToReturn
235             );
236         }
237     }
```

Listing 3.2: `SettlementFacet::redeemPositionToken()`

With that, we suggest to add a new event `RedeemPositionToken(uint256 indexed poolId, uint256 indexed positionToken, address indexed from, uint256 amount, uint256 returnAmount)` whenever user redeems the position token. Also, the `poolId/positionToken/from` parameters are better `indexed`. Note each emitted event is represented as a topic that usually consists of the signature (from a `keccak256` hash) of the event name and the types (`uint256`, `string`, etc.) of its parameters. Each indexed type will be treated like an additional topic. If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the `poolId/positionToken/from` parameters are typically queried, it is better treated as topics, hence the need of being `indexed`.

Recommendation Properly emit the above-mentioned event when user redeems the position token. This is very helpful for external analytics and reporting tools.

Status The issue has been fixed by this commit: `75646e8`.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `GovernanceFacet`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [3]

Description

In the `DIVA` protocol, there is a privileged `contractOwner` account that plays a critical role in governing and regulating the system-wide operations (e.g., set the fee rates and the treasury address). In the following, we examine the privileged account and the related privileged accesses in current contract.

```
12    function setProtocolFee(uint96 _protocolFee)
13    external
14    override
15    onlyOwner
16  {
17    // Min fee introduced to have a minimum non-zero fee in `removeLiquidity`
18    if (_protocolFee > 0) {
19        require(
20            _protocolFee >= 100000000000000,
21            "DIVA: below min allowed"
22        ); // 0.01% = 0.0001
23        require(
24            _protocolFee <= 25000000000000000,
25            "DIVA: exceeds max allowed"
26        ); // 2.5% = 0.025
27    }
28
29    LibDiamond.GovernanceStorage storage gs = LibDiamond.governanceStorage();
30    gs.protocolFee = _protocolFee;
31
32    emit ProtocolFeeSet(msg.sender, _protocolFee);
33  }
34
35  function setSettlementFee(uint96 _settlementFee)
36    external
37    override
38    onlyOwner
39  {
40    // Min fee introduced to have a minimum non-zero fee in `removeLiquidity`
41    if (_settlementFee > 0) {
42        require(
43            _settlementFee >= 100000000000000,
44            "DIVA: below min allowed"
45        ); // 0.01% = 0.0001
46        require(
```

```
47            _settlementFee <= 25000000000000000,
48            "DIVA: exceeds max allowed"
49        ); // 2.5% = 0.025
50    }
51    LibDiamond.GovernanceStorage storage gs = LibDiamond.governanceStorage();
52    gs.settlementFee = _settlementFee;
53
54    emit SettlementFeeSet(msg.sender, _settlementFee);
55  }
```

Listing 3.3: Example Privileged Operations in `GovernanceFacet.sol`

```
125   function setTreasuryAddress(address _newTreasury)
126   external
127   override
128   onlyOwner
129 {
130   require(_newTreasury != address(0), "DIVA: 0x0");
131
132   LibDiamond.GovernanceStorage storage gs = LibDiamond.governanceStorage();
133   gs.treasury = _newTreasury;
134
135   emit TreasuryAddressSet(msg.sender, _newTreasury);
136 }
```

Listing 3.4: Example Privileged Operations in `GovernanceFacet.sol`

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Making the above privileges explicit among protocol users.

Status This issue has been confirmed by the team. And the team clarifies that: The contract owner will only have access to the privileged functionalities in the early phase of the protocol. Once the protocol is proven stable and bug-free, we will renounce the ownership by transferring it to the zero address. This will render the protocol immutable.

3.4 Suggested immutable Usages For Gas Efficiency

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PositionToken`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1099 [1]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

In the following, we show the key state variables defined in `PositionToken`. If there is no need to dynamically update these key state variables after the construction, they can be declared as either constants or `immutable` for gas efficiency. In particular, the state variables `_poolId/_owner/_decimals` can be defined as `immutable` as they will not be changed after their initialization in `constructor()`.

```
18  contract PositionToken is IPositionToken, ERC20 {
19    uint256 private _poolId;
20    address private _owner;
21    uint8 private _decimals;

23    constructor(
24        string memory name_,
25        string memory symbol_,
26        uint256 poolId_,
27        uint8 decimals_
28    ) ERC20(name_, symbol_) {
29        _owner = msg.sender;
30        _poolId = poolId_;
31        _decimals = decimals_;
32    }
33    ...
34  }
```

Listing 3.5: PositionToken.sol

Recommendation Revisit the state variables definition and make extensive use of `immutable` states for gas efficiency.

Status The issue has been fixed by this commit: `046226d`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the DIVA protocol, which aims to be a decentralized piece of infrastructure that allows its users to create and settle fully customizable event-linked products (also known as derivatives) in a permissionless way. The current code base is well organized and those identified issues are promptly confirmed and addressed.

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

References

[1] MITRE. CWE-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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

[7] MITRE. CWE 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.