SMART CONTRACT AUDIT REPORT

for

Lemma Stablecoin V3

Prepared By: Xiaomi Huang

PeckShield

August 26, 2022

Document Properties

Client	Lemma Finance
Title	Smart Contract Audit Report
Target	Lemma Stablecoin V3
Version	1.0
Author	Xiaotao Wu
Auditors	Xiaotao Wu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	August 26, 2022	Xiaotao Wu	Final Release
1.0-rc	August 9, 2022	Xiaotao Wu	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

Given the opportunity to review the design document and related smart contract source code of the Lemma Stablecoin V3 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 Lemma Stablecoin V3

On the basis of Lemma Stablecoin V2, Lemma Stablecoin V3 introduces the Lemma Synthetic token. The Lemma Synthetic token is a yield bearing ERC20 token backed by spot assets and/or long perpetual positions with no leverage. For example, a synthetic ETH could be backed by a long ETH/USD perpetual futures position and/or spot ETH. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Lemma Stablecoin V3

Item	Description
Name	Lemma Finance
Website	https://lemma.finance/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 26, 2022

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

- https://github.com/lemma-finance/basis-trading-stablecoin/tree/merging-Sunny-20220723-almost-final-20220725 (021e554)

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

- https://github.com/lemma-finance/basis-trading-stablecoin/tree/merging-Sunny-20220723-almost-final-20220725 (c32273f)

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.

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.

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

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 `Lemma Stablecoin V3` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	🟦
Medium	1	🟦
Low	2	🟦 🟦
Informational	0	
Total	4	

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 high-severity vulnerability, 1 medium-severity vulnerability and 2 low-severity vulnerabilities.

Table 2.1: Key Lemma Stablecoin V3 Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Suggested Whitelisting of Collateral In LemmaSynth	Security Features	Resolved
PVE-002	Low	Incorrect Deposit/Withdraw Amount In LemmaSynth/USDLemma	Business Logic	Resolved
PVE-003	Low	Accommodation Of Non-ERC20-Compliant Tokens	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Confirmed

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 Suggested Whitelisting of Collateral In LemmaSynth

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: LemmaSynth
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

Lemma Stablecoin V3 introduces the Lemma Synthetic token, which is a yield bearing ERC20 token backed by spot assets and/or long perpetual positions with no leverage. Users can get synthetic tokens by depositing collateral into the perpDEXWrapper. When examining the implementation of the LemmaSynth contract, we notice there is a lack of restriction on the collateral which can be used to mint Lemma Synthetic tokens.

To elaborate, we use the depositTo() routine as an example and show the related code snippet below. In the depositTo() function, we notice the collateral is specified by the input argument without necessary verification. If a malicious actor uses evil ERC20 tokens as collateral to mint Lemma Synthetic tokens, it may cause unpredictable losses to the platform users.

```
/// @notice Deposit collateral like USDC. to mint Synth specifying the exact amount of Synth
/// @param to Receipent of minted Synth
/// @param amount Amount of Synth to mint
/// @param maxCollateralAmountRequired Maximum amount of collateral to be used to mint given Synth
/// @param collateral Collateral to be used to mint Synth
function depositTo(
    address to,
    uint256 amount,
    uint256 maxCollateralAmountRequired,
    IERC20Upgradeable collateral
) public nonReentrant onlyOneFunInSameTx {
    // first trade and then deposit
    IPerpetualMixDEXWrapper perpDEXWrapper = IPerpetualMixDEXWrapper(perpLemma);
    require(address(perpDEXWrapper) != address(0), "invalid DEX/collateral");
    (, uint256 _collateralRequired_1e18) = perpDEXWrapper.openLongWithExactBase(
        amount,
        IPerpetualMixDEXWrapper.Basis.IsSynth
    );

    uint256 _collateralRequired = (address(collateral) == tailCollateral) ? amount :
        _collateralRequired_1e18;
    _collateralRequired = perpDEXWrapper.getAmountInCollateralDecimalsForPerp(
        _collateralRequired,
        address(collateral),
        false
    );
    if (address(collateral) != tailCollateral) {
        require(_collateralRequired_1e18 <= maxCollateralAmountRequired, "collateral
            required execeeds maximum");
    }
    _perpDeposit(perpDEXWrapper, address(collateral), _collateralRequired);
    _mint(to, amount);
    emit DepositTo(address(perpDEXWrapper), address(collateral), to, amount,
        _collateralRequired);
}
```

Listing 3.1: LemmaSynth::depositTo()

Note similar issue also exists in the depositToWExactCollateral() routine of the same contract.

Recommendation Whitelist the given collateral so that only the intended tokens can be supported as collateral to mint Lemma Synthetic tokens.

Status This issue has been fixed in the following commit: e95e98c.

3.2 Incorrect Deposit/Withdraw Amount In LemmaSynth/USDLemma

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: LemmaSynth/USDLemma
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned in Section 3.1, the LemmaSynth contract allows users to mint Lemma Synthetic tokens by depositing collateral into the perpDEXWrapper. Specifically, the depositToWExactCollateral() function allows users to deposit collateral to mint `Lemma Synthetic` tokens by specifying the exact amount of collateral. While examining the routine, we notice the collateral amount deposited to the `perpDEXWrapper` is not correct.

To elaborate, we show below its code snippet. It comes to our attention that the collateral amount should not be scaled (line 182). In other words, the collateral amount deposited to the `perpDEXWrapper` should be `collateralAmount`, instead of current `_collateralRequired` (line 200).

```
180     /// @notice Deposit collateral like USDC to mint Synth specifying the exact amount of collateral
181     /// @param to Receipent of minted Synth
182     /// @param collateralAmount Amount of collateral to deposit in the collateral decimal format
183     /// @param minSynthToMint Minimum Synth to mint
184     /// @param collateral Collateral to be used to mint Synth
185     /// @dev The minted amount depends on the Real Perp Mark Price
186     /// @dev In the specific case of PerpV2, since it is implemented as an UniV3 Pool and opening a position means running a swap on it, slippage has also to be taken into account
187     function depositToWExactCollateral(
188         address to,
189         uint256 collateralAmount,
190         uint256 minSynthToMint,
191         IERC20Upgradeable collateral
192     ) external nonReentrant onlyOneFunInSameTx {
193         IPerpetualMixDEXWrapper perpDEXWrapper = IPerpetualMixDEXWrapper(perpLemma);
194         require(address(perpDEXWrapper) != address(0), "invalid DEX/collateral");
195         uint256 _collateralRequired = perpDEXWrapper.getAmountInCollateralDecimalsForPerp(
196             collateralAmount,
197             address(collateral),
198             false
199         );
200         _perpDeposit(perpDEXWrapper, address(collateral), _collateralRequired);
201         (uint256 _lemmaSynthToMint, ) = perpDEXWrapper.openLongWithExactQuote(
202             collateralAmount,
203             IPerpetualMixDEXWrapper.Basis.IsSynth
204         );
205         require(_lemmaSynthToMint >= minSynthToMint, "Synth minted too low");
206         _mint(to, _lemmaSynthToMint);
207         emit DepositTo(address(perpDEXWrapper), address(collateral), to, _lemmaSynthToMint, _collateralRequired);
208     }
```

Listing 3.2: LemmaSynth::depositToWExactCollateral()

Note the `LemmaSynth::withdrawToWExactCollateral()` and `USDLemma::depositToWExactCollateral()/withdrawToWExactCollateral()` routines share a similar issue.

Recommendation Use the correct collateral amount to deposit/withdraw for the above mentioned functions.

Status This issue has been resolved as the Lemma team confirms that the input argument collateralAmount for above mentioned functions is always 18 decimal scaled, instead of in the collateral decimal format.

3.3 Accommodation Of Non-ERC20-Compliant Tokens

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: PerpLemmaCommon
- 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).

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

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

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

Listing 3.3: USDT Token Contract

Because of that, a normal call to approve() with a currently non-zero allowance may fail. To accommodate the specific idiosyncrasy, there is a need to approve() twice: the first one reduces the allowance to 0; and the second one sets the new allowance.

Moreover, it is important to note that for certain non-compliant ERC20 tokens (e.g., USDT), the approve() function does not have a return value. However, the IERC20 interface has defined the following approve() interface with a bool return value: function approve(address spender, uint256 amount)external returns (bool). As a result, the call to approve() may expect a return value. With the lack of return value of USDT's approve(), the call will be unfortunately reverted.

Because of that, a normal call to approve() is suggested to use the safe version, i.e., safeApprove(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we use the PerpLemmaCommon::_swapOnUniV3() routine as an example. If the USDT token is supported as the tokenIn, the unsafe version of IERC20Decimals(tokenIn).approve() (lines 919 and 948) may revert as there is no return value in the USDT token contract's approve() implementation (but the IERC20Decimals interface expects a return value)!

```
908    /// @dev Helper function to swap on UniV3
909    function _swapOnUniV3(
910        address router,
911        bool isUSDLCollateralToUSDC,
912        bool isExactInput,
913        uint256 amount
914    ) internal returns (uint256) {
915        uint256 res;
916        address tokenIn = (isUSDLCollateralToUSDC) ? address(usdlCollateral) : address(
               usdc);
917        address tokenOut = (isUSDLCollateralToUSDC) ? address(usdc) : address(
               usdlCollateral);

919        IERC20Decimals(tokenIn).approve(router, type(uint256).max);
920        if (isExactInput) {
921            ISwapRouter.ExactInputSingleParams memory temp = ISwapRouter.
                   ExactInputSingleParams({
922                tokenIn: tokenIn,
923                tokenOut: tokenOut,
924                fee: 3000,
925                recipient: address(this),
926                deadline: type(uint256).max,
927                amountIn: amount,
928                amountOutMinimum: 0,
929                sqrtPriceLimitX96: 0
930            });
931            uint256 balanceBefore = IERC20Decimals(tokenOut).balanceOf(address(this));
932            res = ISwapRouter(router).exactInputSingle(temp);
933            uint256 balanceAfter = IERC20Decimals(tokenOut).balanceOf(address(this));
```

```
          res = uint256(int256(balanceAfter) - int256(balanceBefore));
      } else {
          ISwapRouter.ExactOutputSingleParams memory temp = ISwapRouter.
              ExactOutputSingleParams({
              tokenIn: tokenIn,
              tokenOut: tokenOut,
              fee: 3000,
              recipient: address(this),
              deadline: type(uint256).max,
              amountOut: amount,
              amountInMaximum: type(uint256).max,
              sqrtPriceLimitX96: 0
          });
          res = ISwapRouter(router).exactOutputSingle(temp);
      }
      IERC20Decimals(tokenIn).approve(router, 0);
      return res;
  }
```

Listing 3.4: `PerpLemmaCommon::_swapOnUniV3()`

Note the `SettlementTokenManager::settlementTokenRecieve()` routine shares a similar issue.

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

Status This issue has been fixed in the following commit: `e95e98c`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In `Lemma Stablecoin V3` protocol, there is a privileged account, i.e., `owner`. This account play a critical role in governing and regulating the system-wide operations (e.g., deposit/withdraw `settlementToken`, add/delete privileged `USDLEMMA_ROLE/REBALANCER_ROLE/ONLY_OWNER/PERPLEMMA_ROLE/USDC_TREASURY` roles, set key parameters for the `Lemma` protocol, etc.). Our analysis shows that this privileged account need to be scrutinized. In the following, we use the `LemmaSynth` contract as an example and show the representative functions potentially affected by the privileges of the `owner` account.

```
    function initialize(
        address _trustedForwarder,
        address _perpLemma,
        address _tailCollateral,
        string memory _name,
        string memory _symbol
    ) external initializer {
        __ReentrancyGuard_init();
        __ERC20_init(_name, _symbol);
        __ERC20Permit_init(_name);
        __ERC2771Context_init(_trustedForwarder);

        __AccessControl_init();
        _setRoleAdmin(LEMMA_SWAP, ADMIN_ROLE);
        _setRoleAdmin(ONLY_OWNER, ADMIN_ROLE);
        _setupRole(ADMIN_ROLE, msg.sender);
        grantRole(ONLY_OWNER, msg.sender);

        tailCollateral = _tailCollateral;
        updatePerpetualDEXWrapper(_perpLemma);
    }

    /// @notice Add address for perpetual dex wrapper for perpetual index and collateral - can only be called by owner
    /// @param _perpLemma The new PerpLemma Address
    function updatePerpetualDEXWrapper(address _perpLemma) public onlyRole(ONLY_OWNER) {
        require(_perpLemma != address(0), "Address can not be zero");
        perpLemma = _perpLemma;
        emit PerpetualDexWrapperUpdated(_perpLemma);
    }

    /// @notice setTailCollateral set tail collateral, By only owner Role
    /// @param _tailCollateral which collateral address is use to mint LemmaSynth
    function setTailCollateral(address _tailCollateral) external onlyRole(ONLY_OWNER) {
        tailCollateral = _tailCollateral;
        emit SetTailCollateral(_tailCollateral);
    }
```

Listing 3.5: Example Privileged Operations in `LemmaSynth`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the owner may also be a counter-party risk to the protocol users. It is worrisome if the privileged `owner` account is a plain EOA account. 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.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Lemma Stablecoin V3 protocol. On the basis of Lemma Stablecoin V2, Lemma Stablecoin V3 introduces the Lemma Synthetic token, which is a yield bearing ERC20 token backed by spot assets and/or long perpetual positions with no leverage. 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.