SMART CONTRACT AUDIT REPORT

for

Coresky

Prepared By: Xiaomi Huang

PeckShield

May 5, 2025

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	May 5, 2025	Xuxian Jiang	Final Release
1.0-rc	May 2, 2025	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

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

Coresky aims to be the ultimate Meme incubation platform, offering one-click Meme token creation and community-driven voting to support high-quality projects. Backed by a $50 million market cap pool, the platform provides liquidity and market incentives for top-ranked Meme tokens. It is built to provide a user-friendly environment for launching tokens, initializing liquidity pools, conducting secure token trades, and migrating token pairs to prominent decentralized exchanges such as UniswapV2. The basic information of audited contracts is as follows:

Table 1.1: Basic Information of The Coresky

Item	Description
Name	Coresky
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 5, 2025

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers only two files under the contracts/ directory, i.e., Meme/MemeSubjectShares.sol and Meme/TokenFactory.sol.

- https://github.com/Coreskyofficial/core-contract.git (bfbd99c)

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

- https://github.com/Coreskyofficial/core-contract.git (8b80ebc)

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 \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	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.

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

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

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 `Coresky` 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	1	■
Medium	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	6	

We have so far identified a list of potential issues. 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, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Coresky Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Token Fund/Collateral Amount Update in TokenFactory	Business Logic	Resolved
PVE-002	Medium	Improper Fee Percent Update/Use in TokenFactory	Business Logic	Resolved
PVE-003	Low	Possibly Inaccurate Payment Token in Buy/Sell Events	Coding Practices	Resolved
PVE-004	High	Possible Denial-of-Service in Enabling Token Trading	Business Logic	Resolved
PVE-005	Low	Incorrect setBondingCurve() Logic in TokenFactory	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Incorrect Token Fund/Collateral Amount Update in TokenFactory

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

Description

The `Coresky` protocol has a core `TokenFactory` contract to orchestrate the token launch, buy, and sell operations. While examining the token-selling logic, we notice an issue that does not properly update the raised token fund amount.

To elaborate, we show below the related `sell()` routine. It has a rather straightforward logic in validating the given input and then selling the intended token amount. However, it comes to our attention that the raised fund amount should be updated as `tokenInfo.funds -= receivedToken + fee;`, not current `tokenInfo.funds -= receivedToken;` (line 555). The reason is that the fee amount should also be deducted. In addition, the respective token collateral amount should also be updated, i.e., `collateral[tokenAddress] = tokenInfo.funds;`.

```
531     function sell(uint256 serialNo, address tokenAddress, uint256 amount) external
            nonReentrant {
532         TokenInfo storage tokenInfo = _tokenInfos[tokenAddress];
533         require(_templates.length > 0, "Template is empty");
534         Template memory t = _templates[tokenInfo.template];
535         // Verify used
536         if (bitmap.get(serialNo)) revert AlreadyUsed(serialNo);
537         // Mark it used
538         bitmap.set(serialNo);
539         require(
540             tokens[tokenAddress] == TokenState.FUNDING,
541             "Funding has not start"
```

```
        );
        require(amount > 0, "Amount should be greater than zero");
        Token token = Token(tokenAddress);
        uint256 receivedToken = BondingCurve(t.bondingCurve).getFundsReceived(
            token.totalSupply(),
            amount
        );
        // calculate fee
        uint256 _fee = calculateFee(receivedToken, (feeRecipient == address(0) ? 0 :
            feePercent));
        receivedToken -= _fee;
        tradingTotalFeeByToken[tokenAddress] += _fee;
        tradingTotalFeeByQuote[tokenInfo.quote] += _fee;
        token.burn(msg.sender, amount);
        tokenInfo.funds -= receivedToken;

        if(feeRecipient != address(0) && _fee > 0){
            Transfer(feeRecipient, tokenInfo.quote, _fee);
        }
        Transfer(msg.sender, tokenInfo.quote, receivedToken);
        emit Sell(tokenAddress, msg.sender, serialNo, paymentToken, amount,
            receivedToken, block.timestamp);
    }
```

Listing 3.1: `TokenFactory::sell()`

Recommendation Improve the above routine by properly updating the raised token fund amount as well as the collateral amount.

Status The issue has been resolved in the following commit: `8b80ebc`.

3.2 Improper Fee Percent Update/Use in TokenFactory

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: High

- Target: `TokenFactory`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, `Coresky` has a core `TokenFactory` contract to orchestrate the token launch, buy, and sell operations. While examining the token-buying logic, we notice an issue that does not use the intended trading fee.

In the following, we show the code snippet from the affected `buy()` routine. Note that each token sale is guided with the associated template configuration. And within the template configuration, there is a parameter named minTradingFee, which is used as the trading fee. With that, if the feeRecipient parameter is not configured (line 473), we should use a local fee percent variable _feePercent, instead of blindly updating the global feePercent to zero. In other words, we shall use the following statement, i.e., int _feePercent = (feeRecipient == address(0)? 0 : t.minTradingFee));. And the use of feePercent (lines 477 and 483) should be replaced with _feePercent. Note the sell() routine shares the same issue.

```
473         if(feeRecipient == address(0)){
474             feePercent = 0;
475         }
476         //
477         uint256 contributionWithoutFee = (valueToBuy * FEE_DENOMINATOR) / (
                FEE_DENOMINATOR + feePercent);
478         //
479         if (contributionWithoutFee > remainingEthNeeded) {
480             contributionWithoutFee = remainingEthNeeded;
481         }
482         // calculate fee
483         uint256 _fee = calculateFee(contributionWithoutFee, feePercent);
484         uint256 totalCharged = contributionWithoutFee + _fee;
485         uint256 valueToReturn = valueToBuy > totalCharged
486             ? valueToBuy - totalCharged
487             : 0;
488
489         tradingTotalFeeByToken[tokenAddress] += _fee;
490         tradingTotalFeeByQuote[tokenInfo.quote] += _fee;
```

Listing 3.2: TokenFactory::buy()

Recommendation Revise the above-mentioned routines to use the right trading fee.

Status The issue has been resolved in the following commit: 8b80ebc.

3.3 Possibly Inaccurate Payment Token in Buy/Sell Events

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

- Target: TokenFactory
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

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.

In the following, we use the `TokenFactory` contract as an example. This contract has public functions that are used to buy and sell tokens. While examining the actual `Buy/Sell` events, we notice the emitted payment token should be `t.quote`, not current `paymentToken` (lines 528 and 561).

```
531     function sell(uint256 serialNo, address tokenAddress, uint256 amount) external
            nonReentrant {
532         TokenInfo storage tokenInfo = _tokenInfos[tokenAddress];
533         require(_templates.length > 0, "Template is empty");
534         Template memory t = _templates[tokenInfo.template];
535         // Verify used
536         if (bitmap.get(serialNo)) revert AlreadyUsed(serialNo);
537         // Mark it used
538         bitmap.set(serialNo);
539         require(
540             tokens[tokenAddress] == TokenState.FUNDING,
541             "Funding has not start"
542         );
543         require(amount > 0, "Amount should be greater than zero");
544         Token token = Token(tokenAddress);
545         uint256 receivedToken = BondingCurve(t.bondingCurve).getFundsReceived(
546             token.totalSupply(),
547             amount
548         );
549         // calculate fee
550         uint256 _fee = calculateFee(receivedToken, (feeRecipient == address(0) ? 0 :
                feePercent));
551         receivedToken -= _fee;
552         tradingTotalFeeByToken[tokenAddress] += _fee;
553         tradingTotalFeeByQuote[tokenInfo.quote] += _fee;
554         token.burn(msg.sender, amount);
555         tokenInfo.funds -= receivedToken;
556 
557         if(feeRecipient != address(0) && _fee > 0){
558             Transfer(feeRecipient, tokenInfo.quote, _fee);
559         }
560         Transfer(msg.sender, tokenInfo.quote, receivedToken);
561         emit Sell(tokenAddress, msg.sender, serialNo, paymentToken, amount,
                receivedToken, block.timestamp);
562     }
```

Listing 3.3: `TokenFactory::sell()`

Recommendation Properly emit the respective events `Buy/Sell` with the correct payment token information.

Status The issue has been resolved in the following commit: `8b80ebc`.

3.4 Possible Denial-of-Service in Enabling Token Trading

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

Description

As mentioned earlier, `Coresky` builds an innovative `DeFi` platform that caters to the launch, liquidity management, and automated market operations of new tokens. When a new token is launched, it will go through the so-called graduation process. Our analysis shows the token graduation may suffer from a denial-of-service issue.

In the following, we show the code snippet of the related `buy()` routine. When the new token launch process reaches the target `maxRaising`, the shown code snippet will be executed. As part of its logic, it will call the `uniswapV2Factory` contract to create the token pair and add the initial liquidity. However, the token pair creation may be blocked (line 504) if the external pair is already created, resulting in the token graduation failure.

```
501         // when reached FUNDING_GOAL
502         if (tokenCollateral >= tokenInfo.maxRaising) {
503             token.mint(address(this), t.initialLiquidity);
504             address pair = createLiquilityPool(tokenAddress, tokenInfo.quote);
505             uint256 liquidity = addLiquidity(
506                 tokenAddress,
507                 tokenInfo.quote,
508                 t.initialLiquidity,
509                 tokenCollateral
510             );
511             burnLiquidityToken(pair, liquidity);
512             tokenCollateral = 0;
513             tokens[tokenAddress] = TokenState.TRADING;
514             tokenInfo.status = uint256(TokenState.TRADING);
515             emit TokenLiqudityAdded(tokenAddress, block.timestamp);
516         }
517         collateral[tokenAddress] = tokenCollateral;
518         tokenInfo.funds = tokenCollateral;
```

Listing 3.4: `TokenFactory::buy()`

```
566     function createLiquilityPool(
567         address tokenAddress,
568         address quote
569     ) internal returns (address) {
570         IUniswapV2Factory factory = IUniswapV2Factory(uniswapV2Factory);
```

```
        IUniswapV2Router02 router = IUniswapV2Router02(uniswapV2Router);

        address tokenB = quote == address(0)
            ? router.WETH()
            : quote;
        address pair = factory.createPair(tokenAddress, tokenB);
        return pair;
    }
```

Listing 3.5: `TokenFactory::createLiquilityPool()`

Recommendation Revise the above routine to ensure the token graduation is not blocked.

Status The issue has been resolved in the following commit: `8b80ebc`.

3.5 Incorrect setBondingCurve() Logic in TokenFactory

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `TokenFactory`
- 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 `Coresky` protocol is no exception. Specifically, if we examine the `TokenFactory` contract, it has defined a number of protocol-wide risk parameters, such as `bondingCurve` and `paymentToken` . In the following, we show the corresponding routines that allow for their changes.

```
    function setPaymentToken(
        address _paymentToken
    ) external onlyRole(OPERATOR_ROLE) {
        paymentToken = _paymentToken;
        _templates[_defaultTemplate].quote = _paymentToken;
    }

    function setBondingCurve(
        address _bondingCurve
    ) external onlyRole(OPERATOR_ROLE) {
        bondingCurve = BondingCurve(_bondingCurve);
    }
```

Listing 3.6: `TokenFactory::setBondingCurve()`

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, the above `setBondingCurve()` setter can be improved by also updating the default template, i.e., `_templates[_defaultTemplate].bondingCurve = _bondingCurve;`.

Recommendation Validate any changes regarding these system-wide parameters to ensure they are properly applied.

Status The issue has been resolved in the following commit: `8b80ebc`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High

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

Description

In `Coresky`, there is a privileged administrative account (with the `DEFAULT_ADMIN_ROLE` role). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the `TokenFactory` contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function updateSwapAddress(
        address _swapV2Router,
        address _swapV2Factory
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        uniswapV2Router = _swapV2Router;
        uniswapV2Factory = _swapV2Factory;
    }

    function setTokenImpl(
        address _tokenImplementation
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        tokenImplementation = _tokenImplementation;
    }

    function setFundingGoal(
        uint256 _fundingGoal
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        FUNDING_GOAL = _fundingGoal;
        _templates[_defaultTemplate].maxRaising = _fundingGoal;
    }

    function setFeeRecipient(
        address _feeRecipient
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        feeRecipient = _feeRecipient;
    }

    function setFeePercent(
        uint256 _feePercent
    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
        feePercent = _feePercent;
        _templates[_defaultTemplate].minTradingFee = _feePercent;
    }
```

Listing 3.7: Example Privileged Operations in `TokenFactory`

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

Moreover, it should be noted that current contract has 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 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 mitigated as the team confirms that all the privileged roles will be transferred to a `multi-sig` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the Coresky protocol, which aims to be the ultimate Meme incubation platform, offering one-click Meme token creation and community-driven voting to support high-quality projects. Backed by a $50 million market cap pool, the platform provides liquidity and market incentives for top-ranked Meme tokens. It is built to provide a user-friendly environment for launching tokens, initializing liquidity pools, conducting secure token trades, and migrating token pairs to prominent decentralized exchanges such as UniswapV2. 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.