SMART CONTRACT AUDIT REPORT

for

Zenlink Hybrid AMM

Prepared By: Xiaomi Huang

PeckShield

May 19, 2022

Document Properties

Client	Zenlink
Title	Smart Contract Audit Report
Target	Zenlink Hybrid AMM
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 19, 2022	Xuxian Jiang	Final Release
1.0-rc1	May 12, 2022	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 smart contract source code of the `Zenlink Hybrid AMM` 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 audited protocol 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 Zenlink

`Zenlink` is an underlying cross-chain `DEX` protocol based on `Polkadot`, and is committed to become the `DEX` composable hub of `Polkadot`. By accessing the ultimate, open and universal cross-chain `DEX` protocol based on `Substrate`, `Zenlink DEX` enables all parachains to build `DEX` and achieve liquidity sharing in one click. The `Zenlink DEX` protocol includes `Module`, `WASM`, and `EVM` implementations, which are flexible and adaptable, allowing for customizable compositions and interoperability with different `DeFi` modules. The audited `Zenlink Hybrid AMM` contract is a major update of the `Zenlink DEX` protocol, which combines `Standard AMM` and `Stable AMM` and is connected by `Zenlink Smart Order Routing` to provide a better trading experience for `DeFi` users. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Zenlink Hybrid AMM

Item	Description
Name	Zenlink
Website	https://zenlink.pro/en/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	May 19, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Note this audit covers the following contracts: `StableSwapRouter.sol`, `SwapRouterV1.sol`, `StableSwapStorage.sol`, and `StableSwap.sol`.

- https://github.com/zenlinkpro/zenlink-evm-contracts.git (7d26987)

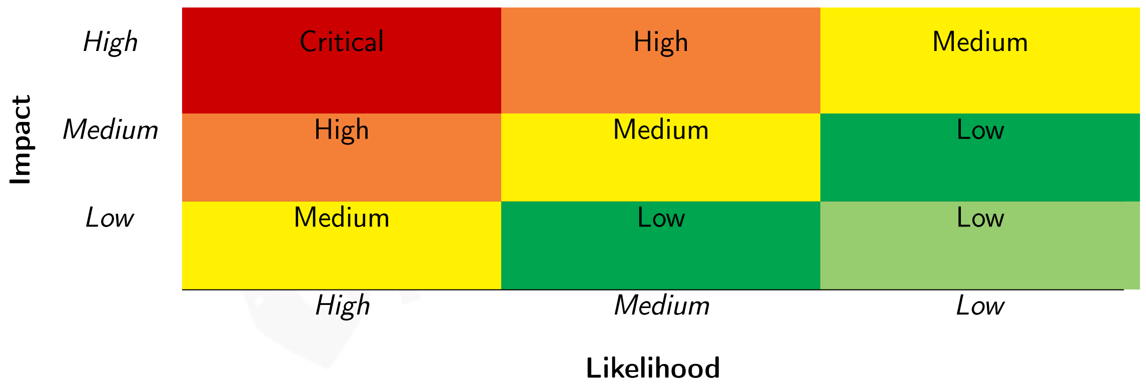
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/zenlinkpro/zenlink-evm-contracts.git (ee3c510)

1.2 About PeckShield

PeckShield Inc. [10] 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



High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low
	High	*Medium*	*Low*

Impact

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- 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) [8], 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 1.3: 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 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 `Zenlink Hybrid AMM` 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 logic, 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 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, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Zenlink Hybrid AMM Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Consistent Use of whenNotPaused Modifier in StableSwap	Coding Practices	Resolved
PVE-002	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Resolved
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-004	Low	Revisited removeLiquidityImbalance Logic in StableSwapStorage	Numeric Errors	Resolved

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 Consistent Use of whenNotPaused Modifier in StableSwap

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `StableSwap`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

As a hybrid `AMM` protocol, `Zenlink` provides core swap-related functionalities, e.g., adding/removing liquidity and swapping one token to another. The protocol also has the `pausable` feature that may be exercised when there is a need to temporarily pause the protocol. While examining the current usage of the `whenNotPaused` modifier, we notice the (minor) inconsistency in its usages.

To elaborate, we show below the related functions, i.e., `removeLiquidity()`, `removeLiquidityOneToken()`, and `removeLiquidityImbalance()`, in the `StableSwap` contract. It comes to our attention that the first funtion does not have the `whenNotPaused` modifier while the last two do have the modifier. For consistency, we suggest to apply the `whenNotPaused` modifier in these functions.

```
    function removeLiquidity(
        uint256 lpAmount,
        uint256[] memory minAmounts,
        uint256 deadline
    ) external override nonReentrant deadlineCheck(deadline) returns (uint256[] memory)
        {
        return swapStorage.removeLiquidity(lpAmount, minAmounts);
    }

    function removeLiquidityOneToken(
        uint256 lpAmount,
        uint8 index,
        uint256 minAmount,
        uint256 deadline
```

```
    ) external override nonReentrant whenNotPaused deadlineCheck(deadline) returns (
        uint256) {
        return swapStorage.removeLiquidityOneToken(lpAmount, index, minAmount);
    }

    function removeLiquidityImbalance(
        uint256[] memory amounts,
        uint256 maxBurnAmount,
        uint256 deadline
    ) external override nonReentrant whenNotPaused deadlineCheck(deadline) returns (
        uint256) {
        return swapStorage.removeLiquidityImbalance(amounts, maxBurnAmount);
    }
```

Listing 3.1: `StableSwap::removeLiquidity()/removeLiquidityOneToken()/removeLiquidityImbalance()`

Recommendation Apply the `whenNotPaused` modifier consistently in `StableSwap`

Status This issue has been resolved as the team ensures the liquidity providers always have the choice to remove the liquidity even when the protocol is paused. The other two functions are designed to have the `whenNotPaused` modifier with the purpose of mitigating possible arbitrage from imbalanced liquidity removal.

3.2 Implicit Assumption Enforcement In AddLiquidity()

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

- Target: `Router`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [4]

Description

In the `Zenlink Hybrid AMM` protocol, the `addLiquidity()` routine (see the code snippet below) is provided to add `amount0Desired` amount of `token0` and `amount1Desired` amount of `token1` into the pool as liquidity via the internal `_addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address token0,
        address token1,
        uint256 amount0Desired,
        uint256 amount1Desired,
        uint256 amount0Min,
        uint256 amount1Min,
        address to,
```

```
        uint256 deadline
    )
        public
        override
        ensure(deadline)
        returns (
            uint256 amount0,
            uint256 amount1,
            uint256 liquidity
        )
    {
        (amount0, amount1) = _addLiquidity(
            token0,
            token1,
            amount0Desired,
            amount1Desired,
            amount0Min,
            amount1Min
        );
        address pair = Helper.pairFor(factory, token0, token1);
        Helper.safeTransferFrom(token0, msg.sender, pair, amount0);
        Helper.safeTransferFrom(token1, msg.sender, pair, amount1);
        liquidity = IPair(pair).mint(to);
    }
```

Listing 3.2: `Router::addLiquidity()`

```
    function _addLiquidity(
        address token0,
        address token1,
        uint256 amount0Desired,
        uint256 amount1Desired,
        uint256 amount0Min,
        uint256 amount1Min
    ) private returns (uint256 amount0, uint256 amount1) {
        if (IFactory(factory).getPair(token0, token1) == address(0)) {
            IFactory(factory).createPair(token0, token1);
        }
        (uint256 reserve0, uint256 reserve1) = Helper.getReserves(
            factory,
            token0,
            token1
        );
        if (reserve0 == 0 && reserve1 == 0) {
            (amount0, amount1) = (amount0Desired, amount1Desired);
        } else {
            uint256 amount1Optimal = Helper.quote(
                amount0Desired,
                reserve0,
                reserve1
            );
            if (amount1Optimal <= amount1Desired) {
                require(
```

```
                    amount1Optimal >= amount1Min,
                    "Router: INSUFFICIENT_1_AMOUNT"
                );
                (amount0, amount1) = (amount0Desired, amount1Optimal);
            } else {
                uint256 amount0Optimal = Helper.quote(
                    amount1Desired,
                    reserve1,
                    reserve0
                );
                require(amount0Optimal <= amount0Desired);
                require(
                    amount0Optimal >= amount0Min,
                    "Router: INSUFFICIENT_0_AMOUNT"
                );
                (amount0, amount1) = (amount0Optimal, amount1Desired);
            }
        }
    }
```

Listing 3.3: `Router::_addLiquidity()`

It comes to our attention that the `Router` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amount0Desired >= amount0Min` and `amount1Desired >= amount1Min`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `Router` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amount1Desired >= amount1Min` and `amount0Desired >= amount0Min` explicitly in the `addLiquidity()` function.

Status The issue has been confirmed and the team plans to address it in the next release.

3.3 Trust Issue of Admin Keys

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

Description

In the `Zenlink Hybrid AMM` protocol, there is a special administrative account (`admin`). This `admin` account plays a critical role in governing and regulating the protocol-wide operations (e.g., configure parameters and execute privileged operations). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that this privileged account needs to be scrutinized. In the following, we examine their related privileged accesses in current protocol.

```
    function setFee(uint256 newSwapFee, uint256 newAdminFee) external onlyAdmin {
        require(newSwapFee <= MAX_SWAP_FEE, "> maxSwapFee");
        require(newAdminFee <= MAX_ADMIN_FEE, "> maxAdminFee");
        swapStorage.adminFee = newAdminFee;
        swapStorage.fee = newSwapFee;
        emit NewFee(newSwapFee, newAdminFee);
    }

    /**
     * @notice Start ramping up or down A parameter towards given futureA_ and futureTime_
     * Checks if the change is too rapid, and commits the new A value only when it falls under
     * the limit range.
     * @param futureA the new A to ramp towards
     * @param futureATime timestamp when the new A should be reached
     */
    function rampA(uint256 futureA, uint256 futureATime) external onlyAdmin {
        require(block.timestamp >= swapStorage.initialATime + (1 days), "< rampDelay"); // please wait 1 days before start a new ramping
        require(futureATime >= block.timestamp + (MIN_RAMP_TIME), "< minRampTime");
        require(0 < futureA && futureA < MAX_A, "outOfRange");

        uint256 initialAPrecise = swapStorage.getAPrecise();
        uint256 futureAPrecise = futureA * StableSwapStorage.A_PRECISION;

        if (futureAPrecise < initialAPrecise) {
            require(futureAPrecise * (MAX_A_CHANGE) >= initialAPrecise, "> maxChange");
        } else {
            require(futureAPrecise <= initialAPrecise * (MAX_A_CHANGE), "> maxChange");
        }
```

```
        swapStorage.initialA = initialAPrecise;
        swapStorage.futureA = futureAPrecise;
        swapStorage.initialATime = block.timestamp;
        swapStorage.futureATime = futureATime;

        emit RampA(initialAPrecise, futureAPrecise, block.timestamp, futureATime);
    }
```

Listing 3.4: Example Privileged Operations in `StableSwap`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract 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 Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated by the team by having a multi-sig account as the `admin`.

3.4 Revisited removeLiquidityImbalance Logic in StableSwapStorage

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

- Target: `StableSwapStorage`
- Category: Numeric Errors [7]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the mixed uses when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `StableSwapStorage::removeLiquidityImbalance()` as an example. This routine is used to remove the liquidity in a way that will introduce the imbalance to the current pool. Because of possible rounding issues that may be caused from internal computation, there is a need to round up the `burnAmount` calculation by adding 1 to it, i.e., `burnAmount += 1`. By doing so, we can better protect the interest of the pool's remaining liquidity providers.

```
    function removeLiquidityImbalance(
        SwapStorage storage self,
        uint256[] memory amounts,
        uint256 maxBurnAmount
    ) external returns (uint256 burnAmount) {
        uint256 nCoins = self.pooledTokens.length;
        require(amounts.length == nCoins, "invalidAmountsLength");
        uint256 totalSupply = self.lpToken.totalSupply();
        require(totalSupply != 0, "totalSupply = 0");
        uint256 _fee = _feePerToken(self);
        uint256 amp = _getAPrecise(self);

        uint256[] memory newBalances = self.balances;
        uint256 D0 = _getD(_xp(self), amp);

        for (uint256 i = 0; i < nCoins; i++) {
            newBalances[i] -= amounts[i];
        }

        uint256 D1 = _getD(_xp(newBalances, self.tokenMultipliers), amp);
        uint256[] memory fees = new uint256[](nCoins);

        for (uint256 i = 0; i < nCoins; i++) {
            uint256 idealBalance = (D1 * self.balances[i]) / D0;
            uint256 diff = _distance(newBalances[i], idealBalance);
            fees[i] = (_fee * diff) / FEE_DENOMINATOR;
            self.balances[i] = newBalances[i] - ((fees[i] * self.adminFee) /
                FEE_DENOMINATOR);
            newBalances[i] -= fees[i];
        }

        // recalculate invariant with fee charged balances
        D1 = _getD(_xp(newBalances, self.tokenMultipliers), amp);
        burnAmount = ((D0 - D1) * totalSupply) / D0;
        assert(burnAmount > 0);
        require(burnAmount <= maxBurnAmount, "> slippage");

        self.lpToken.burnFrom(msg.sender, burnAmount);

        for (uint256 i = 0; i < nCoins; i++) {
            if (amounts[i] != 0) {
                self.pooledTokens[i].safeTransfer(msg.sender, amounts[i]);
            }
        }

        emit RemoveLiquidityImbalance(msg.sender, amounts, fees, D1, totalSupply -
            burnAmount);
    }
```

Listing 3.5: `StableSwapStorage::removeLiquidityImbalance()`

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status This issue has been fixed in the following commit: ee3c510.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Zenlink Hybrid AMM` protocol, which is the underlying unified and universal cross-chain `DEX` protocol and enables `parachains` to quickly have `DEX` functionality and share liquidity with other `parachains`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, we need to emphasize that 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[4] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.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 CATEGORY: Numeric Errors. https://cwe.mitre.org/data/definitions/189.html.

[8] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[9] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[10] PeckShield. PeckShield Inc. https://www.peckshield.com.