SMART CONTRACT AUDIT REPORT

for

Pionex SwapX

Prepared By: Xiaomi Huang

PeckShield

September 19, 2023

Document Properties

Client	Pionex
Title	Smart Contract Audit Report
Target	Pionex SwapX
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	September 19, 2023	Xuxian Jiang	Final Release
1.0-rc	September 15, 2023	Xuxian Jiang	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 Pionex SwapX 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 Pionex SwapX

The Pionex SwapX protocol is a decentralized aggregator protocol and contains contracts to integrate a variety of AMMs/DEXs together in order to enable swapping from any asset on supported AMMs to any asset in either direction. It currently supports a number of combinations, including UniswapV2, UniswapV2, and UniswapV3. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Pionex SwapX Protocol

Item	Description
Name	Pionex
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	September 19, 2023

In the following, we show the compressed file with smart contracts and its checksum hash values used in this audit.

- swapx.zip: a3e8671ca465f9b8dd084c97c77bd38059d60c38004f782fa631e00ddd724638

And this is the checksum hash value after all fixes for the issues found in the audit have been checked in:

- swapx.zip: a3e8671ca465f9b8dd084c97c77bd38059d60c38004f782fa631e00ddd724638

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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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 `Pionex SwapX` 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	0	
Medium	2	■■
Low	4	■■■■
Informational	0	
Total	6	

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 2 medium-severity vulnerabilities and 4 low-severity vulnerabilities.

Table 2.1: Key Pionex SwapX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Parameter Validation in SwapX	Coding Practices	Resolved
PVE-002	Low	Potential Inconsistency Between Actual Swaps And Return Amounts	Business Logic	Resolved
PVE-003	Low	Improved Logic in swapMixedMultiHopExactOut()	Business Logic	Resolved
PVE-004	Medium	Extra Funds Return in swapV2ExactIn()/swapV3MultiHopExactIn()	Business Logic	Resolved
PVE-005	Low	Corner Case Handling in swapMixedMultiHopExactIn()	Business Logic	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 Improved Parameter Validation in SwapX

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SwapX
- 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 SwapX protocol is no exception. Specifically, if we examine the SwapX contract, it has defined a number of protocol-wide risk parameters, such as feeRate and feeExcludeList. In the following, we show the corresponding routines that allow for their changes.

```
    function setFeeRate(uint256 rate) external onlyOwner {
        feeRate = rate;
    }

    function setFeeCollector(address addr) external onlyOwner {
        feeCollector = addr;
    }

    function setWETH(address addr) external onlyOwner {
        WETH = addr;
    }

    function setFeeExclude(address addr, bool isExcluded) external onlyOwner {
        feeExcludeList[addr] = isExcluded;
    }
```

Listing 3.1: Example setters in SwapX

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, an unlikely mis-configuration of `feeRate` may charge unreasonably high fee in the `fee` payment, hence incurring cost to borrowers or hurting the adoption of the protocol.

Recommendation Validate any changes regarding these system-wide parameters to ensure they fall in an appropriate range.

Status The issue has been resolved by validating the given `feeRate` to be smaller than `feeDenominator`.

3.2 Potential Inconsistency Between Actual Swaps And Return Amounts

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

Description

As a swap aggregator, the `SwapX` protocol aims to seamlessly interact with integrated `DEXs` via a number of well-defined functions. While examining the exported functions, we notice certain inconsistency between actual swap amounts with the expected return values.

In the following, we use the `swapV2MultiHopExactIn()` routine as an example. As the name indicates, this routine is used to perform a multi-hop swap with the input amount specified by the trading user. We notice the actual swap function supports fee-on-transfer tokens, while the return amount is computed via the library function `getAmountsOut()`, which does not support fee-on-transfer tokens. As a result, the actual output amount from swap is different from current return value. Note other two functions, i.e., `swapV2MultiHopExactOut()` and `swapMixedMultiHopExactIn()`, share the same issue.

```
    function swapV2MultiHopExactIn(
        address tokenIn,
        uint256 amountIn,
        uint256 amountOutMin,
        address[] calldata path,
        address recipient,
        uint deadline,
        address factory
```

```
    ) payable public nonReentrant whenNotPaused checkDeadline(deadline) returns (uint[]
        memory amounts){

        require(amountIn > 0, "SwapX: amount in is zero");

        uint256 fee = takeFee(tokenIn, amountIn);
        amountIn = amountIn - fee;

        address firstPool = UniswapV2Library.pairFor(factory, path[0], path[1]);
        if (tokenIn == address(0)) {
            tokenIn = WETH;
            pay(tokenIn, address(this), firstPool, amountIn);
            require(msg.value >= amountIn + fee, "SwapX: amount in and value mismatch");
        } else
            pay(tokenIn, msg.sender, firstPool, amountIn);
        require(tokenIn == path[0], "invalid path");

        amounts = UniswapV2Library.getAmountsOut(factory, amountIn, path);

        uint balanceBefore = IERC20Upgradeable(path[path.length - 1]).balanceOf(
            recipient);
        _swapSupportingFeeOnTransferTokens(path, recipient, factory);
        require(
            IERC20Upgradeable(path[path.length - 1]).balanceOf(recipient).sub(
                balanceBefore) >= amountOutMin,
            'SwapX: insufficient output amount'
        );
    }
```

Listing 3.2: `SwapX::swapV2MultiHopExactIn()`

Recommendation Be consistent between actual swap amount and expected return values in the above-mentioned functions.

Status This issue has been resolved by setting the last element in the return `amounts` so that it is consistent with actual swap amount.

3.3 Improved Logic in swapMixedMultiHopExactOut()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: SwapX
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier, the SwapX protocol seamlessly interacts with integrated DEXs via a number of well-defined functions. While examining one specific function swapMixedMultiHopExactOut(), we notice its logic needs to be improved.

To elaborate, we show below the implementation of this swapMixedMultiHopExactOut() routine. This routine is used to perform a mixed multi-hop swap with the output amount specified by the trading user. Specifically, in the mixed UniswapV2-UniswapV3 case, we notice the first-hop swap relies on the prior knowledge of intermediate swap amount, which may not be realistic. Instead, we can compute the expected intermediate amount from the final amount in the second-hop swap and then walk back to the first-hop swap.

```
        ...
        } else if (isStrEqual(params.routes[0], "v2") && isStrEqual(params.routes[1], "v3")) {
            // NOTE: v3 not support fee-on-transfer token, so the mid-token amountIn is exactly same as params.amountIn2
            // v3 path bytes is reversed
            (tokenOut, ,) = params.path2.decodeFirstPool();

            address poolAddress1 = UniswapV2Library.pairFor(params.factory1, tokenIn, tokenOut1);
            address[] memory path1 = new address[](2);
            path1[0] = tokenIn;
            path1[1] = tokenOut1;
            uint[] memory amounts1 = UniswapV2Library.getAmountsIn(params.factory1, params.amountIn2, path1);
            amountIn = amounts1[0];
            if (tokenIn == WETH) {
                pay(tokenIn, address(this), poolAddress1, amountIn);
            } else
                pay(tokenIn, msg.sender, poolAddress1, amountIn);

            _swap(amounts1, path1, address(this), params.factory1);

            uint amountIn2 = exactOutputInternal(
                params.amountOut,
                params.recipient,
```

```
                0,
                SwapCallbackData({path: params.path2, payer: address(this)})
            );
            require(amountIn2 == params.amountIn2, "SwapX: not support fee-on-transfer
                token for V3");

        } else if (isStrEqual(params.routes[0], "v3") && isStrEqual(params.routes[1], "
            v2")) {
            ...
        }
```

Listing 3.3: `SwapX::swapMixedMultiHopExactOut()`

Recommendation Revise the above logic to compute the intermediary amount instead of relying on the user input.

Status This issue has been resolved as the the intermediary amount will computed from the team's backend service.

3.4 Extra Funds Return in swapV2ExactIn()/swapV3MultiHopExactIn()

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

Description

To facilitate the token swaps, the `SwapX` protocol supports a set of well-defined interfaces that allow the user to specify the exact input amount or expected output amount. While examining two specific functions, e.g., `swapV2ExactIn()` and `swapV3MultiHopExactIn()`, we notice the possibility of receiving extra native coins from the trading user and these extra coins can be better returned back to the user.

To elaborate, we show below the `swapV2ExactIn()` routine implementation. This routine allows the user to directly swap native coins in `ETH` to another token. And it comes to our attention that current implementation only validates the input amount is sufficient (line 174), but does not return the extra funds back.

```
155     function swapV2ExactIn(
156         address tokenIn,
157         address tokenOut,
```

```
        uint256 amountIn,
        uint256 amountOutMin,
        address poolAddress
    ) payable public nonReentrant whenNotPaused returns (uint amountOut){

        require(poolAddress != address(0), "SwapX: invalid pool address");
        require(amountIn > 0, "SwapX: amout in is zero");

        uint256 fee = takeFee(tokenIn, amountIn);
        amountIn = amountIn - fee;

        bool nativeOut = false;
        if (tokenOut == address(0))
            nativeOut = true;

        if (tokenIn == address(0)) {
            require(msg.value >= amountIn + fee, "SwapX: amount in and value mismatch");
            tokenIn = WETH;
            pay(tokenIn, address(this), poolAddress, amountIn);
        } else
            pay(tokenIn, msg.sender, poolAddress, amountIn);

        uint balanceBefore = nativeOut ?
            IERC20Upgradeable(WETH).balanceOf(address(this)) :  IERC20Upgradeable(
                tokenOut).balanceOf(msg.sender);

        IUniswapV2Pair pair = IUniswapV2Pair(poolAddress);
        address token0 = pair.token0();
        uint amountInput;
        uint amountOutput;
        { // scope to avoid stack too deep errors
            (uint reserve0, uint reserve1,) = pair.getReserves();
            (uint reserveInput, uint reserveOutput) = tokenIn == token0 ? (reserve0,
                reserve1) : (reserve1, reserve0);
            amountInput = IERC20Upgradeable(tokenIn).balanceOf(address(pair)).sub(
                reserveInput);
            amountOutput = UniswapV2Library.getAmountOut(amountInput, reserveInput,
                reserveOutput);
        }
        (uint amount0Out, uint amount1Out) = tokenIn == token0 ? (uint(0), amountOutput)
             : (amountOutput, uint(0));
        address to = nativeOut ? address(this) : msg.sender;
        pair.swap(amount0Out, amount1Out, to, new bytes(0));

        if (nativeOut) {
            amountOut = IERC20Upgradeable(WETH).balanceOf(address(this)).sub(
                balanceBefore);
            IWETH(WETH).withdraw(amountOut);
            (bool success, ) = address(msg.sender).call{value: amountOut}("");
            require(success, "SwapX: send ETH out error");
        } else {
            amountOut = IERC20Upgradeable(tokenOut).balanceOf(msg.sender).sub(
```

```
                balanceBefore);
        }
        require(
            amountOut >= amountOutMin,
            'SwapX: insufficient output amount'
        );
    }
```

Listing 3.4: `SwapX::swapV2ExactIn()`

Recommendation Return any extra fund, if any, back to the trading user. Note both `swapV2ExactIn()` and `swapV3MultiHopExactIn()` routines share this issue.

Status The issue has been resolved by returning extra funds back to the user.

3.5 Corner Case Handling in swapMixedMultiHopExactIn()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `SwapX`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

Among the set of exported swap routines, `SwapX` has a specific routine `swapMixedMultiHopExactIn()` to allow for a mixed multi-hop swap with the input amount specified by the user. While analyzing this specific routine, we notice there is a corner case that can be revisited.

To elaborate, we show below the code snippet of this swap routine. The specific corner case occurs when the user intends to swap by using the `WETH` as the input token instead of native coin `ETH`. Note that current implementation interprets the use of `WETH` as native coin `ETH` and thus does not support the direct trading of `WETH` as the input token. This limitation is unnecessary and can be seamlessly resolved.

```
    function swapMixedMultiHopExactIn (
        ExactInputMixedParams memory params
    ) payable public nonReentrant whenNotPaused checkDeadline(params.deadline) returns (
        uint256 amountOut) {

        require(params.routes.length == 2, "SwapX: only 2 routes supported");

        require(params.amountIn > 0, "SwapX: amount in is zero");
        if (msg.value > 0)
            require(msg.value >= params.amountIn, "SwapX: amount in and value mismatch")
                ;
```

```
        (address tokenIn, address tokenOut1,) = params.path1.decodeFirstPool();
        uint256 fee = takeFee(tokenIn, params.amountIn);
        params.amountIn = params.amountIn - fee;

        if (isStrEqual(params.routes[0], "v2") && isStrEqual(params.routes[1], "v2")) {
            // uni - sushi, or verse
            address poolAddress1 = UniswapV2Library.pairFor(params.factory1, tokenIn,
                tokenOut1);
            if (tokenIn == WETH) {
                pay(tokenIn, address(this), poolAddress1, params.amountIn);
            } else
                pay(tokenIn, msg.sender, poolAddress1, params.amountIn);
                ...
    }
```

Listing 3.5: `SwapX::swapMixedMultiHopExactIn()`

Recommendation Resolve the above-mentioned limit by supporting both `WETH` and `ETH` as the input token.

Status The issue has been resolved by supporting both `WETH` and `ETH` as the input token.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `SwapX`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `SwapX` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., privileged account setting and fee adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
    function pause() external onlyOwner {
        _pause();
    }

    function unpause() external onlyOwner {
        _unpause();
    }
```

```

    function setFeeRate(uint256 rate) external onlyOwner {
        feeRate = rate;
    }

    function setFeeCollector(address addr) external onlyOwner {
        feeCollector = addr;
    }

    function setWETH(address addr) external onlyOwner {
        WETH = addr;
    }

    function setFeeExclude(address addr, bool isExcluded) external onlyOwner {
        feeExcludeList[addr] = isExcluded;
    }
```

Listing 3.6: Example Privileged Operations in SwapX

If the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, 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 changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been mitigated by having a multi-sig wallet to manage the admin key.

4 Conclusion

In this audit, we have analyzed the `Pionex SwapX` protocol design and implementation. `Pionex SwapX` is a decentralized aggregator protocol and contains contracts to integrate a variety of `AMMs/DEXs` together in order to enable swapping from any asset on supported `AMMs` to any asset in either direction. It currently supports a number of combinations, including `UniswapV2`, `UniswapV2`, and `UniswapV3`. During the audit, we notice that the current code base is well organized and those identified issues are promptly confirmed and fixed.

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-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.