SMART CONTRACT AUDIT REPORT

for

OpenSwap Protocol

Prepared By: Yiqun Chen

PeckShield

October 25, 2021

Document Properties

Client	OpenSwap
Title	Smart Contract Audit Report
Target	OpenSwap
Version	1.0
Author	Xuxian Jiang
Auditors	Shulin Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	October 25, 2021	Xuxian Jiang	Final Release
1.0-rc	October 1, 2021	Xuxian Jiang	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the OpenSwap design document and related smart contract source code, 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 branch of OpenSwap 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 OpenSwap

OpenSwap aims to address two primary issues of current AMM-based DEX solutions: slippage and impermanent loss. The key idea is to provide on-chain spot market-priced swaps in the form of liquidity queues with further innovations on new technologies and concepts such as hybrid smart routing, and inter-chain liquidity swaps. The audited implementation focuses on the current liquidity queues logic. OpenSwap enriches the current DEX landscape and also presents a unique contribution to current DeFi ecosystem.

The basic information of OpenSwap is as follows:

Table 1.1: Basic Information of OpenSwap

Item	Description
Name	OpenSwap
Website	https://openswap.xyz/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	October 25, 2021

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

- https://github.com/openswapdex/openswap-core.git (4d1826d)

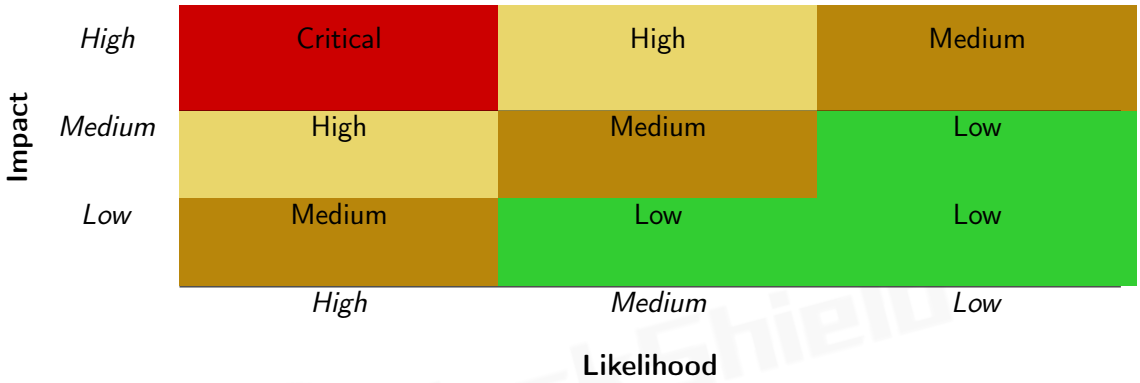
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/openswapdex/openswap-core.git (ac44f2f)

1.2 About PeckShield

PeckShield Inc. [12] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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) [10], 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 OpenSwap 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	2	■■
Low	3	■■■
Informational	1	■
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, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key OpenSwap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved _getOfferList() Logic	Business Logic	Fixed
PVE-002	Low	Reentrancy Protection of OSWAP_RangePair	Time and State	Fixed
PVE-003	Medium	Trust Issue Of Admin Keys	Security Features	Resolved
PVE-004	Medium	Improved Sanity Checks For Function Arguments	Business Logic	Fixed
PVE-005	Informational	Unused State/Code Removal	Coding Practices	Fixed
PVE-006	Low	Inconsistency Between Document And Implementation	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 Detailed Results

3.1 Improved _getOfferList() Logic

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-708 [5]

Description

As mentioned earlier, `OpenSwap` offers `liquidity queues` in a number of categories, each based on a different pricing mechanism. In this section, we examine the `Spot Price Queues` where trades will occur based on oracle-based market prices. In particular, these queues are designed for market makers (liquidity providers) seeking to exit positions without slippage at spot-market prices. In other words, market makers will be able to place their liquidity on a specific queue, i.e., `BNB -> USDT`, and be protected from any impermanent loss since they only provide a single asset. While examining its implementation, we notice an internal helper function `_getOfferList()` can be improved.

To elaborate, we show below its implementation. This helper is only used in the public `swap(uint256 amount0Out, uint256 amount1Out, address to, bytes calldata data)` function with four arguments. The helper function is used to extract the offer list array from the given fourth argument `data`. We notice the implementation has an internal validation on the input `calldata` size (line 443): `lt(calldatasize(), add(offset, add(size, 0x20)))`. Our analysis shows it misses the inclusion of `bytes length` field of the fourth argument. In other words, the validation needs to be `lt(calldatasize(), add(offset, add(size, 0x40)))`!

```
437     function _getOfferList(uint256 offset) internal pure returns(uint256[] memory list,
            uint256 dataRead) {
438         require(msg.data.length >= offset.add(0x40), "Invalid offer list");
439         assembly {
440             let count := calldataload(add(offset, 0x20))
441             let size := mul(count, 0x20)
```

```
443             if lt(calldatasize(), add(offset, add(size, 0x20))) { // count * 0x20 + 0x20
                    + 0x44
444                 revert(0, 0)
445             }
446             let mark := mload(0x40)
447             mstore(0x40, add(mark, add(size, 0x20))) // malloc
448             mstore(mark, count) // array length
449             calldatacopy(add(mark, 0x20), add(offset, 0x40), size) // copy data to list
450             list := mark
451             dataRead := add(size, 0x20)
452         }
453     }
```

Listing 3.1: `OSWAP_RangePair::_getOfferList()`

Note the same issue is also applicable to another routine `_decodeData()` in other two contracts `OSWAP_RestrictedPair` and `OSWAP_RestrictedPair2`.

Recommendation Revise the above two functions `_getOfferList()` and `_decodeData()` to properly enforce the `calldata` size.

Status The issue has been fixed by this commit: `13e819a`.

3.2 Reentrancy Protection of OSWAP_RangePair

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `OSWAP_RangePair`
- Category: Time and State [9]
- CWE subcategory: CWE-663 [4]

Description

A common coding best practice in Solidity is the adherence of `checks-effects-interactions` principle. This principle is effective in mitigating a serious attack vector known as `re-entrancy`. Via this particular attack vector, a malicious contract can be reentering a vulnerable contract in a nested manner. Specifically, it first calls a function in the vulnerable contract, but before the first instance of the function call is finished, second call can be arranged to re-enter the vulnerable contract by invoking functions that should only be executed once. This attack was part of several most prominent hacks in Ethereum history, including the `DAO` [14] exploit, and the recent `Uniswap/Lendf.Me` hack [13].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `OAXDEX_Administrator` as an example, the `executeVetoVoting()` function (see the code snippet below) is provided to externally call a token contract to transfer assets. However, the invocation

of an external contract requires extra care in avoiding the above re-entrancy. For example, the interaction with the external contract (line 103) start before effecting the update on the internal state (line 104), hence violating the principle. In this particular case, if the external contract has certain hidden logic that may be capable of launching re-entrancy via the same entry function.

```
101     function executeVetoVoting(address votingContract) external override {
102         require(checkVote(vetoVotingVote[votingContract]), "Admin: executeVetoVoting:
                Quorum not met");
103         IOAXDEX_Governance(governance).veto(votingContract);
104         clearVote(vetoVotingVote[votingContract]);
105     }
```

Listing 3.2: OAXDEX_Administrator::executeVetoVoting()

The similar issue is also present in other functions, e.g., executeFactoryShutdown(), executeFactoryRestart(), executePairShutdown(), executePairRestart() from the OAXDEX_Administrator contract, and the adherence of the checks-effects-interactions best practice is strongly recommended. In addition, the updateProviderOffer() and removeAllLiquidity functions from the OSWAP_RangePair contract can be improved to have the lock modifier to thwart potential reentrancy.

Recommendation Apply necessary reentrancy prevention by following the checks-effects-interactions principle and utilizing the necessary nonReentrant modifier to block possible re-entrancy.

Status The issue has been fixed by this commit: f260bfd.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the OpenSwap protocol, there is a privileged admin account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and adding new oracle updates). 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.

```
296     function veto(address voting) external override {
297         require(msg.sender == admin, "OAXDEX: Not from shutdown admin");
298         IOAXDEX_VotingContract(voting).veto();
```

```
299         _closeVote(voting);
300         emit Veto(voting);
301     }
```

Listing 3.3: OAXDEX_Governance::veto()

```
15     function mint(address account, uint256 amount) external {
16         require(msg.sender == minter, "Not from minter");
17         _mint(account, amount);
18     }
```

Listing 3.4: OpenSwap::mint()

Note that if the privileged admin account or the related minter is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange users. 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 that current contracts may have 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 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 This issue has been resolved. The team clarifies that the veto() caller is configured to be OAXDEX_Administrator contract, which requires majority of admins to veto.

3.4 Improved Sanity Checks For Function Arguments

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: OSWAP_HybridRouter2
- Category: Business Logic [8]
- CWE subcategory: CWE-708 [5]

Description

To facilitate the user trading, the OpenSwap protocol has a built-in router contract that provides a number of swap-related functions, i.e., swapExactTokensForTokens() and swapTokensForExactTokens().

These functions also make use of an internal helper routine `getPathOut()` to build up the traverse path for the swap. Our analysis shows that this `getPathOut()` is currently flawed.

To illustrate, we show below its implementation. This routine has a rather straightforward logic in identifying each member in the path. However, it comes to our attention the internal `for`-loop starts with the wrong condition. Specifically, it needs to start with `j = pair.length - 1`, instead of `j = pair.length` (line 71).

```
64    function getPathOut(address[] calldata pair, address tokenOut) public override view
          returns (address[] memory path) {
65        uint256 length = pair.length;
66        require(length > 0, 'INVALID_PATH');
67        path = new address[](length + 1);
68        path[path.length - 1] = tokenOut;
69        (address[] memory token0, address[] memory token1) = IOSWAP_HybridRouterRegistry
              (registry).getPairTokens(pair);
70        uint256 i;
71        for (uint j = pair.length ; j > 0; j--) {
72            i = j - 1;
73            path[i] = _findToken(token0[i], token1[i], tokenOut);
74            tokenOut = path[i];
75        }
76    }
```

Listing 3.5: `OSWAP_HybridRouter2::getPathOut()`

Moreover, two other functions `getAmountIn()` and `getAmountOut()` in `OSWAP_HybridRouter` can be improved by requiring the given `fee` parameter no larger than the fee denominator, i.e., `require(fee <10**6`.

Recommendation Correct the flaw in the above `getPathOut()` helper routine.

Status The issue has been fixed by this commit: `f260bfd`.

3.5 Unused State/Code Removal

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The `OpenSwap` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the

OSWAP_RestrictedPair smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the OSWAP_RestrictedPair contract, it has defined a number of states and constants, such as FEE_BASE, FEE_BASE_SQ, and WEI. We notice the FEE_BASE_SQ constant is never used in current contract and thus can be safely removed.

```
15 contract OSWAP_RestrictedPair is IOSWAP_RestrictedPair, OSWAP_PausablePair {
16     using SafeMath for uint256;
17
18     uint256 constant FEE_BASE = 10 ** 5;
19     uint256 constant FEE_BASE_SQ = (10 ** 5) ** 2;
20     uint256 constant WEI = 10**18;
21
22     bytes32 constant FEE_PER_ORDER = "RestrictedPair.feePerOrder";
23     bytes32 constant FEE_PER_TRADER = "RestrictedPair.feePerTrader";
24     bytes32 constant MAX_DUR = "RestrictedPair.maxDur";
25     bytes4 private constant SELECTOR = bytes4(keccak256(bytes('transfer(address,uint256)
           ')));
26
27     ...
28     }
```

Listing 3.6: The OSWAP_RestrictedPair Contract

Note the same constant is also defined, but not used in another contract OSWAP_RestrictedPair2. Therefore, it is also suggested for removal.

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status The issue has been fixed by this commit: 13e819a.

3.6 Inconsistency Between Document And Implementation

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: OSWAP_Pair
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

The OpenSwap protocol advances current DEX landscape by innovative liquidity queues. In the meantime, we notice the core DEX engine is influenced from UniswapV2 with extensions for various

liquidity queues. While reviewing the customized fee support, we notice the inconsistency between the document and the implementation.

In particular, we use the OSWAP_Pair contract as an example. The inconsistency comes from the protocol fee collection. If the protocol fee is collected at every trade, it may unnecessarily impose an additional gas cost on every trade. To avoid this, accumulated fees are collected only when liquidity is deposited or withdrawn. The contract computes the accumulated fees, and mints new liquidity tokens to the fee beneficiary, immediately before any tokens are minted or burned. To elaborate, we show below the related _mintFee() function.

```
108     // if fee is on, mint liquidity equivalent to 1/6th of the growth in sqrt(k)
109     function _mintFee(uint112 _reserve0, uint112 _reserve1) private returns (bool feeOn)
            {
110         (uint256 latestProtocolFee, address protocolFeeTo) = IOSWAP_Factory(factory).
                protocolFeeParams();
111         uint _protocolFee = protocolFee;
112         feeOn = protocolFeeTo != address(0);
113         uint _kLast = kLast; // gas savings
114         if (feeOn) {
115             if (_kLast != 0) {
116                 uint rootK = Math.sqrt(uint(_reserve0).mul(_reserve1));
117                 uint rootKLast = Math.sqrt(_kLast);
118                 if (rootK > rootKLast) {
119                     uint numerator = totalSupply.mul(rootK.sub(rootKLast)).mul(
                            _protocolFee);
120                     uint denominator = rootK.mul(FEE_BASE.sub(_protocolFee)).add(
                            rootKLast.mul(_protocolFee));
121                     uint liquidity = numerator / denominator;
122                     if (liquidity > 0) _mint(protocolFeeTo, liquidity);
123                 }
124             }
125         } else if (_kLast != 0) {
126             kLast = 0;
127         }
128         if (_protocolFee != latestProtocolFee)
129             protocolFee = latestProtocolFee;
130     }
```

Listing 3.7: OSWAP_Pair::_mintFee()

It comes to our attention that the current protocol fee occupies the _protocolFee/FEE_BASE, instead of the mentioned "1/6th of the growth in sqrt(k)" (line 108).

Recommendation Revise the above _mintFee() function to make it consistent on the percentage of protocol fee for collection.

Status The issue has been fixed by this commit: f260bfd.

4 | Conclusion

In this audit, we have analyzed the OpenSwap design and implementation. The protocol presents a unique offering as a decentralized DEX solution that addresses two primary issues of current AMM-based DEX solutions, i.e., slippage and impermanent loss. The current code base is clearly organized and those identified issues are promptly confirmed and resolved.

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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

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

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

[4] MITRE. CWE-663: Use of a Non-reentrant Function in a Concurrent Context. https://cwe.mitre.org/data/definitions/663.html.

[5] MITRE. CWE-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.html.

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

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

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

[9] MITRE. CWE CATEGORY: Concurrency. https://cwe.mitre.org/data/definitions/557.html.

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

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

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

[13] PeckShield. Uniswap/Lendf.Me Hacks: Root Cause and Loss Analysis. https://medium.com/@peckshield/uniswap-lendf-me-hacks-root-cause-and-loss-analysis-50f3263dcc09.

[14] David Siegel. Understanding The DAO Attack. https://www.coindesk.com/understanding-dao-hack-journalists.