SMART CONTRACT AUDIT REPORT

for

RhinoFi Protocol

Prepared By: Xiaomi Huang

PeckShield

June 30, 2022

Document Properties

Client	RhinoFi
Title	Smart Contract Audit Report
Target	RhinoFi
Version	1.0
Author	Xuxian Jiang
Auditors	Patrick Liu, Jing Wang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 30, 2022	Xuxian Jiang	Final Release
1.0-rc	June 25, 2022	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 source code of the `RhinoFi` 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 RhinoFi

`RhinoFi` is a fast, liquid, and private non-custodial trading portal built on Ethereum, leveraging `StarkWare`'s Layer-2 scaling technology (`ZK-Rollup/Validium`). `RhinoFi` provides high-speed API and UI access to deep order-books, allowing for 9000+ transactions per second, privacy-by-default, competitive fees, and less counter-party risk transactions. This audit covers an extended `UniswapV2` to allow synchronization of the `AMM` pool state with a layer 2 state whilst maintaining the original rules of the curve. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of RhinoFi

Item	Description
Name	RhinoFi
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 30, 2022

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

- https://github.com/RhinoFi/contracts.git (ef4ab61)

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

- https://github.com/RhinoFi/contracts.git (fade1e6)

1.2 About PeckShield

PeckShield Inc. [13] 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

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [12]:

- 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) [11], 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 `RhinoFi` 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	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 Audit Findings of RhinoFi Protocol

ID	Severity	Title	Category	Status
PVE-001	Medium	Possible DoS against PairHolder::upgradeTo()	Business Logic	Resolved
PVE-002	Low	Implicit Assumption Enforcement In AddLiquidity()	Security Features	Resolved
PVE-003	Informational	Redundant Validation in verifyNonceAndLocked()	Coding Practices	Resolved
PVE-004	Low	Permissionless Privileged Functions in UniswapV2Router02	Security Feature	Resolved
PVE-005	Medium	Trust on Admin Keys	Security Features	Mitigated
PVE-006	Low	Reentrancy Risk in DVFDepositContract	Time And State	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 Possible DoS against PairHolder::upgradeTo()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `PairHolder`
- Category: Business Logic [9]
- CWE subcategory: CWE-841 [6]

Description

The `RhinoFi` protocol is designed to be upgradeable by adopting the popular `Beacon`-based proxy pattern. This approach allows multiple proxies to be upgraded to a different implementation in a single transaction, by holding not the implementation address in storage like `UpgradeableProxy`, but the address of a `UpgradeableBeacon` contract. While examining the current upgrade logic, we observe a possible denial-of-scenario issue.

To elaborate, we show below the implementation of two related routines: `upgradeTo()` and `requestUpgradeTo()`. The second one makes the request by indicating the `newImplementation` address while the first one actually makes the changes after the given `upgradeDelay` expires. However, it comes to our attention the second routine can be invoked by anyone, indicating any intended upgrade can be blocked! To fix this issue, there is a need to make the `requestUpgradeTo()` permissioned by validating the caller.

```
function upgradeTo(address _newImplementation) public override {
  require(
    upgradeDelay == 0  implementationRequestTs + upgradeDelay >= block.timestamp,
    'UPGRADE_DELAY_NOT_REACHED'
  );

  super.upgradeTo(_newImplementation);
}

function requestUpgradeTo(address _newImplementation) public {
  newImplementation = _newImplementation;
```

```
    implementationRequestTs = block.timestamp;
}
```

Listing 3.1: `PairHolderr::upgradeTo()/requestUpgradeTo()`

Recommendation Revise the above `requestUpgradeTo()` routine to properly validate the caller.

Status This issue has been resolved in the following commit hash: `917f27c`.

3.2 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV2Router02`
- Category: Coding Practices [8]
- CWE subcategory: CWE-628 [4]

Description

The `RhinoFi` protocol makes use of a revised `UniswapV2` implementation to serve as its AMM logic. Within the revised `UniswapV2` implementation, there is a routine `addLiquidity()` that is used to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity. Our analysis shows that the current implementation has an implicit assumption that needs to be explicitly enforced. To elaborate, we show below the related code snippet.

```
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) public virtual override ensure(deadline) returns (uint amountA, uint amountB, uint
         liquidity) {
        (amountA, amountB) = _addLiquidity(tokenA, tokenB, amountADesired,
            amountBDesired, amountAMin, amountBMin);
        address pair = UniswapV2Library.pairFor(factory, tokenA, tokenB);
        TransferHelper.safeTransferFrom(tokenA, msg.sender, pair, amountA);
        TransferHelper.safeTransferFrom(tokenB, msg.sender, pair, amountB);
        liquidity = IUniswapV2Pair(pair).mint(to);
    }
```

Listing 3.2: `UniswapV2Router02::addLiquidity()`

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'DVF_AMM_Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'DVF_AMM_Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.3: `UniswapV2Router02::_addLiquidity()`

It comes to our attention that the `Uniswap V2 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., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. 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 `Uniswap V2 Router` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

Status The issue has been confirmed.

3.3 Redundant Validation in verifyNonceAndLocked()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `PairWithL2Overlay`
- Category: Coding Practices [8]
- CWE subcategory: CWE-563 [3]

Description

The `RhinoFi` 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 `PairWithL2Overlay` 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 `verifyNonceAndLocked()` function from the `PairWithL2Overlay` contract, the function validates the given `nonceToUse` and the current lock status. So far, there are three requirement statements and it comes to our attention that the first requirement (line 161) is a subset of the second requirement (line 162). In other words, the first requirement can be safely removed.

```
157  function verifyNonceAndLocked(uint nonceToUse) private view {
158    bool isLockedLocal = isLocked();
159    bool isNonceUsed = nonce > nonceToUse; // TODO revert, temporary change
160
161    require(!(isLockedLocal && nonce == nonceToUse), 'DVF: DUPLICATE_REQUEST');
162    require(!isLockedLocal, 'DVF: LOCK_IN_PROGRESS');
163    require(!isNonceUsed, 'DVF: NONCE_ALREADY_USED');
164  }
```

Listing 3.4: `PairWithL2Overlay::verifyNonceAndLocked()`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been resolved as the team indicates the need of the most descriptive error .

3.4 Permissionless Privileged Functions in UniswapV2Router02

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

- Target: `UniswapV2Router02`
- Category: Coding Practices [8]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned in Section 3.2, the `RhinoFi` protocol makes use of a revised `UniswapV2` implementation to serve as its AMM logic. In the meantime, a number of functions added to the router to allow invocation of newly added pair functions, including `syncStarkware()`, `abortStarkware()`, `setupStarkware()`, and `toggleLayer2()`.

In the following, we examine two added functions. The first function `abortStarkware()` is designed to abort the L2 `Starkware` engine and restart a new one. And the second function `toggleLayer2()` toggles the layer2 status of the underlying pair. It comes to our attention that both functions are defined as permissionless, meaning that they can be invoked by anyone. However, the abort and toggle operations are sensitive ones, which may affect all current users. With that, there is a need to guard these functions to ensure that only the intended operator is able to invoke them.

```
function abortStarkware(address tokenA, address tokenB, string memory id) external
    returns(uint256 newVaultId) {
  address pairAddress = UniswapV2Library.pairFor(factory, tokenA, tokenB);
  IUniswapV2Pair pair = IUniswapV2Pair(pairAddress);
  newVaultId = pair.abortStarkware();
  emit Sync(pairAddress, id, 'abortStarkware', newVaultId);
}
```

Listing 3.5: `UniswapV2Router02::abortStarkware()`

```
function toggleLayer2(address tokenA, address tokenB, bool state) public {
  address pairAddress = UniswapV2Library.pairFor(factory, tokenA, tokenB);
  IUniswapV2Pair pair = IUniswapV2Pair(pairAddress);
  pair.toggleLayer2(state);
  (uint balance0, uint balance1, uint totalSupply) = getPairState(tokenA, tokenB);

  emit Layer2StateChange(pairAddress, state, balance0, balance1, totalSupply);
}
```

Listing 3.6: `UniswapV2Router02::toggleLayer2()`

Recommendation Improve the above routines by properly validating the calling user.

Status This issue has been resolved and the team confirms that the router contract is almost stateless and does not have any permission checks.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [7]
- CWE subcategory: CWE-287 [2]

Description

In the `RhinoFi` protocol, there is a privileged `operator` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and role assignment). 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 setupStarkware(uint _assetId, uint _token0AssetId, uint _token1AssetId) external operatorOnly {
  require(_assetId != 0, 'ALREADY_SETUP');
  IStarkEx starkEx = getStarkEx();
  require(extractContractAddress(starkEx, _assetId) == address(this), 'INVALID_ASSET_ID');
  require(isValidAssetId(starkEx, _token0AssetId, token0), 'INVALID_TOKENA_ASSET_ID');
  require(isValidAssetId(starkEx, _token1AssetId, token1), 'INVALID_TOKENB_ASSET_ID');
  lpAssetId = _assetId;
  token0AssetId = _token0AssetId;
  token1AssetId = _token1AssetId;
  token0Quanatum = starkEx.getQuantum(_token0AssetId);
  token1Quanatum = starkEx.getQuantum(_token1AssetId);
}

function authorizeWithdrawals(uint blockNumberTo, uint lpAmount, bool validateId) external override operatorOnly {
  _authorizeWithdrawals(blockNumberTo, lpAmount, validateId);
}

function setWithdrawalDelay(uint newDelay) external operatorOnly {
  _setWithdrawalDelay(newDelay);
}
```

Listing 3.7: Example Privileged Operations in the `PairWithL2Overlay` Contract

Notice that the privilege assignment is necessary and consistent with the protocol design. In the meantime, the extra power to the owner may also be a counter-party risk to the protocol 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 Making the above privileges explicit among protocol users.

Status This issue has been confirmed with the team. For the time being, the team needs to have the owner (for upgrades) and operators to secure access to various contract functions. The owner keys will be multisig and eventually the governance contracts.

3.6 Reentrancy Risk in DVFDepositContract

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `DVFDepositContract`
- Category: Time and State [10]
- CWE subcategory: CWE-663 [5]

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` [15] exploit, and the recent `Uniswap/Lendf.Me` hack [14].

We notice there are occasions where the `checks-effects-interactions` principle is violated. Using the `DVFDepositContract` as an example, the `withdraw()` 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 90) start before effecting the update on the internal state (lines 105), 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.

```
86   function withdraw(address token, address to, uint256 amount, string calldata
         withdrawalId) external
87     _isAuthorized
88     _withUniqueWithdrawalId(withdrawalId)
89   {
90     IERC20Upgradeable(token).safeTransfer(to, amount);
91     emit BridgedWithdrawal(to, token, amount, withdrawalId);
92   }
93
94   modifier _withUniqueWithdrawalId(string calldata withdrawalId) {
95     require(
```

```
      bytes(withdrawalId).length > 0,
      "Withdrawal ID is required"
    );
    require(
      !processedWithdrawalIds[withdrawalId],
      "Withdrawal ID Already processed"
    );
    _;

    processedWithdrawalIds[withdrawalId] = true;
  }
```

Listing 3.8: `DVFDepositContract::withdraw()`

While the supported tokens in the protocol do implement rather standard ERC20 interfaces and their related token contracts are not vulnerable or exploitable for `re-entrancy`, it is important to take precautions to thwart possible `re-entrancy`.

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 confirmed and the related functions are only for authorized entities.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `RhinoFi` protocol, which is a fast, liquid, and private non-custodial trading portal built on Ethereum by leveraging `StarkWare`'s Layer-2 scaling technology (`ZK-Rollup`/`Validium`). This audit covers an extended `UniswapV2` to allow synchronization of the `AMM` pool state with a layer 2 state whilst maintaining the original rules of the curve. 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-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

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

[6] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.html.

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

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

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

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

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

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

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

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

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