SMART CONTRACT AUDIT REPORT

for

PawnFi Protocol

Prepared By: Xiaomi Huang

PeckShield

June 5, 2023

Document Properties

Client	PawnFi
Title	Smart Contract Audit Report
Target	PawnFi
Version	1.1
Author	Xuxian Jiang
Auditors	Stephen Bie, Luck Hu, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.1	June 5, 2023	Xuxian Jiang	Post Final #1
1.0	April 1, 2023	Xuxian Jiang	Final Release
1.0-rc	March 12, 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 PawnFi 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 version of PawnFi 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 PawnFi

PawnFi is a leading provider of instant liquidity solutions for NFTs. By leveraging the P-Token mechanism and integrating multi-modal features, PawnFi is designed to unlock deep liquidity and tap into the potential of NFTs without requiring ownership or digital asset transfers. In contrast, existing platforms like Blur/OpenSea offer a semi-primary market, where tokens are traded among different whales and reach fewer buyers or users, resulting in a market ceiling. The P-Token mechanism provides a protective layer that safeguards NFTs against extreme circumstances, enabling them to circulate safely across the broader DeFi ecosystem. The basic information of PawnFi is as follows:

Table 1.1: Basic Information of PawnFi

Item	Description
Name	PawnFi
Website	https://pawnfi.com
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 5, 2023

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

- https://github.com/PawnFi/PAWNtoken.git (7baa4c7)
- https://github.com/PawnFi/Marketplace.git (c4c8789)
- https://github.com/PawnFi/Tools.git (c559d22)
- https://github.com/PawnFi/DAO.git (77ca8fa)
- https://github.com/PawnFi/NFTFactory.git (966b049)
- https://github.com/PawnFi/Lending.git (f4ee084)
- https://github.com/PawnFi/LiquidityBoosting.git (7496621)

And here are the commit IDs after all fixes for the issues found in the audit have been checked in:

- https://github.com/PawnFi/PAWNtoken.git (70d869b)
- https://github.com/PawnFi/Marketplace.git (ff23392)
- https://github.com/PawnFi/Tools.git (9ee405c)
- https://github.com/PawnFi/DAO.git (2fab3c6)
- https://github.com/PawnFi/NFTFactory.git (a5cbf8a)
- https://github.com/PawnFi/Lending.git (7fdebf9)
- https://github.com/PawnFi/LiquidityBoosting.git (70e2282)

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).

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;

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	High	Medium	Low
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

- 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.

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 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 `PawnFi` 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	3	■■■
Low	2	■■
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 3 medium-severity vulnerabilities, 2 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key PawnFi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Informational	Implicit Decimal Assumption of Staking Token in SAFE	Business Logic	Fixed
PVE-002	Medium	Potential getMarket() Manipulation via FeeManager::initialMarket()	Business Logic	Fixed
PVE-003	Low	Timely rewardsPerSecond Refresh Before Changing reductionRatio	Coding Practices	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Time and State	Fixed
PVE-005	Medium	Improved Order Matching in PawnfiApproveTrade	Business Logic	Fixed
PVE-006	Medium	Trust Issue Of Admin Keys	Security Features	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 Implicit Decimal Assumption of Staking Token in SAFE

- ID: PVE-001
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: SAFE
- Category: Coding Practices [5]
- CWE subcategory: CWE-1041 [1]

Description

The PawnFi protocol has a DAO in place, which offers a governance mechanism to incentivize placing token in SAFE, voting, and participating in governance processes to earn rewards. The DAO enables iToken staking for PAWN token mining. Additionally, the contract provides for the distribution of lending market revenue to voting power holders. While examining the supported staking token in DAO, we notice an implicit assumption on its decimal and this implicit assumption is better explicitly enforced.

To elaborate, we show below the implementation of the related availableVotes() function. This function is designed to return the amount of unused votes for the current week. It comes to our attention the computed userTotalVotes is derived from userTotalVotes = tokenSAFE.userWeight(user)/ 1e18 (line 207). The division of 1e18 implicitly assumes the decimals of the staking token is 18. With that, we suggest to enforce the assumption when the staking token is applied.

```
199     /**
200      * @notice Get the amount of unused votes for for the current week being voted on
201      * @param user Address to query
202      * @return uint Amount of unused votes Amount of unused votes
203      */
204     function availableVotes(address user) external view returns (uint256) {
205         uint256 week = getWeek();
206         uint256 usedVotes = userVotes[user][week];
207         uint256 userTotalVotes = tokenSAFE.userWeight(user) / 1e18;
208         return userTotalVotes - usedVotes;
209     }
```

Listing 3.1: IncentiveVoting::availableVotes()

Recommendation Make the implicit assumption of the staking token's decimals in `DAO` explicit.

Status The issue has been fixed by this commit: `d735196`.

3.2 Potential getMarket() Manipulation via FeeManager::initialMarket()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FeeManager`
- Category: Business Logic [6]
- CWE subcategory: CWE-708 [3]

Description

The `PawnFi` protocol's `DAO` has a `FeeManager` contract that is designed to collect and distribute protocol fees from various lending pools. To facilitate the management of lending pools, `FeeManager` provides an `initialMarket()` routine to initialize the given lending market. Our analysis shows that this routine can be exploited to manipulate the market accounting.

Specifically, we show below the implementation of this routine. It comes to our attention this specific routine is not guarded and can be invoked by anyone. As a result, a malicious actor may call this routine to provide a crafted `market`, which bypasses the `pendingAdmin` check and re-initializes the accounting of the underlying asset. Note the underlying asset is returned from `underlyingAsset(market)` (line 151), which can be fully controlled by the malicious actor.

```
148     function initialMarket(address market) public {
149         address pendingAdmin = ICToken(market).pendingAdmin();
150         if(pendingAdmin == address(this)) {
151             address asset = underlyingAsset(market);
152             getMarket[asset] = market;
153             ICToken(market)._acceptAdmin();
154             uint256 totalReserves = ICToken(market).totalReserves();
155             marketInfo[market] = MarketInfo({
156                 lastReserves: totalReserves,
157                 lastTime: startTime,
158                 claimed: 0
159             });
160         }
161     }
```

Listing 3.2: `FeeManager::initialMarket()`

Recommendation Make the above function privileged so that only the owner is allowed to add a new market.

Status The issue has been fixed by this commit: `ff7601d`.

3.3 Timely rewardsPerSecond Refresh Before Changing reductionRatio

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `IncentiveVoting`
- Category: Business Logic [6]
- CWE subcategory: CWE-708 [3]

Description

The `PawnFi` protocol supports flexible support of rewards, which can be controlled in a number of protocol parameters, e.g., `rewardsPerSecond` and `reductionRatio`. While examining the dynamic update of `reductionRatio`, we notice the lack of timely refresh of certain protocol state before applying the new reduction ratio.

In particular, we show below the full implementation of `setReductionRatio()`, which is designed to apply a new reduction rate. It comes to our attention that the `rewardsPerSecond` array is not timely updated on previous epochs for reward distribution. With that, we need to timely update the array before applying the new reduction rate.

```
    /**
     * @notice Set reduction rate
     * @param newReductionRatio New reduction rate
     */
    function setReductionRatio(uint256 newReductionRatio) external onlyOwner {
        require(newReductionRatio < DENOMINATOR);
        emit ReductionRatioUpdate(reductionRatio, newReductionRatio);
        reductionRatio = newReductionRatio;
    }
```

Listing 3.3: `IncentiveVoting::setReductionRatio()`

Recommendation Revise the above routine to properly update the `rewardsPerSecond` array. An example revision is shown below:

```
    /**
     * @notice Set reduction rate
     * @param newReductionRatio New reduction rate
     */
    function setReductionRatio(uint256 newReductionRatio) external onlyOwner {
        require(newReductionRatio < DENOMINATOR);
        uint256 week = getWeek();
        _refreshRewardPerSecond(week, rewardsPerSecond.length);
```

```
        emit ReductionRatioUpdate(reductionRatio, newReductionRatio);
        reductionRatio = newReductionRatio;
    }

    function _refreshRewardPerSecond(uint256 week, uint256 length) private {
        if (length <= week / 4) {
            uint256 perSecond = rewardsPerSecond[length-1];
            while (length <= week / 4) {
                perSecond = perSecond * reductionRatio / DENOMINATOR;
                length += 1;
                rewardsPerSecond.push(perSecond);
            }
        }
    }
```

Listing 3.4: Revised `IncentiveVoting::setReductionRatio()`

Status The issue has been fixed by this commit: `ec97432`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [6]
- CWE subcategory: CWE-708 [3]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. Specifically, the `transfer()` routine does not have a return value defined and implemented. However, the `IERC20` interface has defined the `transfer()` interface with a `bool` return value. As a result, the call to `transfer()` may expect a return value. With the lack of return value of `USDT`'s `transfer()`, the call will be unfortunately reverted.

```
    function transfer(address _to, uint _value) public onlyPayloadSize(2 * 32) {
        uint fee = (_value.mul(basisPointsRate)).div(10000);
        if (fee > maximumFee) {
            fee = maximumFee;
        }
        uint sendAmount = _value.sub(fee);
        balances[msg.sender] = balances[msg.sender].sub(_value);
```

```
        balances[_to] = balances[_to].add(sendAmount);
        if (fee > 0) {
            balances[owner] = balances[owner].add(fee);
            Transfer(msg.sender, owner, fee);
        }
        Transfer(msg.sender, _to, sendAmount);
    }
```

Listing 3.5: USDT::transfer()

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In current implementation, if we examine the `RewardDistributor::claim()` routine that is designed to claim the funds according to the given Merkle proof. To accommodate the specific idiosyncrasy, there is a need to user `safeTransfer()`, instead of `transfer()` (line 38).

```
28    function claim(
29        address account,
30        uint256 amount,
31        bytes32[] memory proof
32    ) external {
33        bytes32 leaf = keccak256(abi.encodePacked(account, amount));
34        require(!claimed[leaf], "Airdrop already claimed");
35        MerkleVerifier._verifyProof(leaf, merkleRoot, proof);
36        claimed[leaf] = true;

38        require(IERC20(token).transfer(account, amount), "Transfer failed");

40        emit Claimed(account, amount);
41    }

43    function reclaim(uint256 amount) external onlyOwner {
44        require(block.timestamp > reclaimPeriod, "Tokens cannot be reclaimed");
45        require(IERC20(token).transfer(msg.sender, amount), "Transfer failed");
46    }
```

Listing 3.6: RewardDistributor::claim()/reclaim()

In the meantime, we also suggest to use the safe-version of `transfer()` in other related routines, including `RewardDistributor::reclaim()` and `FeeManager::claimRefreshReward()`.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`.

Status The issue has been fixed by the following commits: `8a679cd`, `f66fb37`, and `c855196`.

3.5 Improved Order Matching in PawnfiApproveTrade

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: PawnfiApproveTrade
- Category: Business Logic [6]
- CWE subcategory: CWE-708 [3]

Description

The PawnFi protocol has a built-in Marketplace component, which offers an NFT listing. It facilitates the purchase of a signed NFT by a buyer from a seller, with the seller being contractually obligated to sell the NFT to the buyer at the price agreed upon and signed by both parties. While reviewing the current logic, we notice the current order validation can be improved.

In the following, we show the implementation of the current order validation routine _validate (). The signature validation in essence computes the hash of the given order and ensure it is not expired, cancelled, or finished. It comes to our attention that the order expiry is validated by ensuring `require(order.deadline >= block.timestamp)` (line 324). This validation can be improved as follows: `require(order.deadline ==0 || order.deadline >= block.timestamp)`.

```
323     function _validate(Order memory order) internal view returns (bytes32 digest) {
324         require(order.deadline >= block.timestamp, "Transaction expired!");
325 
326         digest = hashOrder(order);
327         require(!cancelledOrFinalized[digest], "Already cancel or finalized.");
328 
329         if (AddressUpgradeable.isContract(order.maker)) {
330             // 0x1626ba7e is the interfaceId for signature contracts (see IERC1271)
331             require(IERC1271Upgradeable(order.maker).isValidSignature(digest, order.sig)
                    == 0x1626ba7e, "Invalid signature");
332         } else {
333             address signer = digest.recover(order.sig);
334             require(signer != address(0) && signer == order.maker, "Invalid signature");
335         }
336     }
```

Listing 3.7: PawnfiApproveTrade::_validate()

Recommendation Improve the above validation of the given order to better accommodate the specific case when order.deadline==0.

Status The issue has been fixed by this commit: f43ac09.

3.6 Trust Issue of Admin Keys

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

Description

In the PawnFi protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., configuring various parameters and adding new allowed tokens). 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 the related privileged accesses in current contracts.

```
    function setITokenStaking(IITokenStaking newITokenStaking, address[] memory
        initialApprovedTokens) external {
        require(address(iTokenStaking) == address(0));
        iTokenStaking = newITokenStaking;
        for (uint i = 0; i < initialApprovedTokens.length; i++) {
            address token = initialApprovedTokens[i];
            isApproved[token] = true;
            approvedTokens.push(token);
            newITokenStaking.addPool(token);
        }
    }

    /**
     * @notice Set feeManager contract address
     * @param newFeeManager feeManager contract address
     */
    function setFeeManager(IFeeManager newFeeManager) external {
        require(address(feeManager) == address(0));
        feeManager = newFeeManager;
    }

    /**
     * @notice Set reduction rate
     * @param newReductionRatio New reduction rate
     */
    function setReductionRatio(uint256 newReductionRatio) external onlyOwner {
        require(newReductionRatio < DENOMINATOR);
        emit ReductionRatioUpdate(reductionRatio, newReductionRatio);
        reductionRatio = newReductionRatio;
    }
```

Listing 3.8: Example Privileged Functions in IncentiveVoting

Note that if the privileged `owner` account 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 as the team plans to use the `DAO` to act as the privileged owner.

4 Conclusion

In this audit, we have analyzed the PawnFi design and implementation. The protocol is a leading provider of instant liquidity solutions for NFTs. By leveraging the P-Token mechanism and integrating multi-modal features, PawnFi is designed to unlock deep liquidity and tap into the potential of NFTs without requiring ownership or digital asset transfers. In contrast, existing platforms like Blur/OpenSea offer a semi-primary market, where tokens are traded among different whales and reach fewer buyers or users, resulting in a market ceiling. The P-Token mechanism provides a protective layer that safeguards NFTs against extreme circumstances, enabling them to circulate safely across the broader DeFi ecosystem. 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-708: Incorrect Ownership Assignment. https://cwe.mitre.org/data/definitions/708.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.