# SMART CONTRACT AUDIT REPORT

for

# JanisDex

Prepared By: Xiaomi Huang

PeckShield

May 5, 2023

## Document Properties

| | |
|---|---|
| Client | JanisDex |
| Title | Smart Contract Audit Report |
| Target | JanisDex |
| Version | 1.0 |
| Author | Xuxian Jiang |
| Auditors | Luck Hu, Xuxian Jiang |
| Reviewed by | Patrick Lou |
| Approved by | Xuxian Jiang |
| Classification | Public |

## Version Info

| Version | Date | Author(s) | Description |
|---|---|---|---|
| 1.0 | May 5, 2023 | Xuxian Jiang | Final Release |
| 1.0-rc | April 22, 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 JanisDex 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 JanisDex

Arbitrum Janis DEX is a UniswapV2-based decentralized exchange on Arbitrum One. Features include a community fairlaunch release method, multiple token rewards, a transaction fee pool for J tokens, platform dividends received by ownership token holders, extinction pools that sacrifice user deposits in exchange for yield, NFT deposit pools, and NFT yield boosting for farms and pools that reads NFT stats. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The JanisDex Protocol

| Item | Description |
|---|---|
| Name | JanisDex |
| Type | Ethereum Smart Contract |
| Platform | Solidity |
| Audit Method | Whitebox |
| Latest Audit Report | May 5, 2023 |

In the following, we show the hash values of the given JanisDex files for audit.

- JanisDex-11th-April.zip (md5: 5f95eed34930531ce5cdeec10ac11eca)
- JanisDex-20th-April.zip (md5: 52f56f4ce7c13f3109c0b6ad5465a6ad)

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

- https://github.com/LithiumSwapTech/janis-contracts.git (f72f025)

## 1.2 About PeckShield

PeckShield Inc. [10] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

| 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 [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

## 1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: 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 JanisDex 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 | 2 | ■■ |
| Medium | 1 | ■ |
| Low | 2 | ■■ |
| Informational | 0 | |
| Total | 5 | |

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 high-severity vulnerabilities, 1 medium-severity vulnerabilitity, and 2 low-severity vulnerabilities.

Table 2.1: Key JanisDex Audit Findings

| ID | Severity | Title | Category | Status |
|---|---|---|---|---|
| PVE-001 | High | Incorrect K-Invariant Enforcement in UniswapV2Pair | Business Logic | Resolved |
| PVE-002 | Low | Implicit Assumption Enforcement In AddLiquidity() | Coding Practices | Resolved |
| PVE-003 | Medium | Trust Issue of Admin Keys | Security Features | Mitigated |
| PVE-004 | High | Double Minting of Janis Reward in JanisMinter | Business Logic | Resolved |
| PVE-005 | Low | Inconsistent Protocol Parameter Enforcement in JanisMasterChef | Coding Practices | Resolved |

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 Incorrect K-Invariant Enforcement in UniswapV2Pair

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `UniswapV2Pair`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

The `JanisDex` protocol is a fork of the popular `UniswapV2` protocol with fee adjustment as well as other features. While examining the impact on the enforced `K-Invariant` due to the fee adjustment, we notice the invariant is not properly enforced.

To elaborate, we show below the related `UniswapV2Pair::swap()` routine. As the name indicates, this routine performs the actual swap logic with the enforcement of `K-Invariant`. However, it comes to our attention that the current enforcement (line 223) mis-calculates the precision and may be exploited to drain funds from the pair. The correct enforcement should be the following: `require(balance0Adjusted.mul(balance1Adjusted)>= uint(_reserve0).mul(_reserve1).mul(10000**2)`!

```
199    function swap(uint amount0Out, uint amount1Out, address to, bytes calldata data)
           external lock {
200        require(amount0Out > 0  amount1Out > 0, 'UniswapV2: INSUFFICIENT_OUTPUT_AMOUNT')
               ;
201        (uint112 _reserve0, uint112 _reserve1,) = getReserves(); // gas savings
202        require(amount0Out < _reserve0 && amount1Out < _reserve1, 'UniswapV2:
               INSUFFICIENT_LIQUIDITY');
203
204        uint balance0;
205        uint balance1;
206        { // scope for _token{0,1}, avoids stack too deep errors
207        address _token0 = token0;
208        address _token1 = token1;
209        require(to != _token0 && to != _token1, 'UniswapV2: INVALID_TO');
```

```
        if (amount0Out > 0) _safeTransfer(_token0, to, amount0Out); // optimistically transfer tokens
        if (amount1Out > 0) _safeTransfer(_token1, to, amount1Out); // optimistically transfer tokens
        if (data.length > 0) IUniswapV2Callee(to).uniswapV2Call(msg.sender, amount0Out, amount1Out, data);
        balance0 = IERC20Uniswap(_token0).balanceOf(address(this));
        balance1 = IERC20Uniswap(_token1).balanceOf(address(this));
        }
        uint amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
        uint amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
        require(amount0In > 0  amount1In > 0, 'UniswapV2: INSUFFICIENT_INPUT_AMOUNT');
        { // scope for reserve{0,1}Adjusted, avoids stack too deep errors
        uint _swapFee = swapFee;
        uint balance0Adjusted = (balance0.mul(10000).sub(amount0In.mul(_swapFee)));
        uint balance1Adjusted = (balance1.mul(10000).sub(amount1In.mul(_swapFee)));
        require(balance0Adjusted.mul(balance1Adjusted) >= uint(_reserve0).mul(_reserve1).mul(1000**2), 'UniswapV2: K');
        }

        _update(balance0, balance1, _reserve0, _reserve1);
        emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
    }
```

Listing 3.1: `UniswapV2Pair::swap()`

**Recommendation** Revise the above logic to properly enforce the `K-Invariant`. Note that the same issue is also present in the library `UniswapV2Library`.

**Status** The issue has been fixed by following our suggestion.

## 3.2 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `UniswapV2Router02`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

### Description

In the `UniswapV2Router02` contract, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `UniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
    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, 'UniswapV2Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
    function addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external 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()`

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 fixed by adding the suggested requirement.

## 3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

### Description

In the `JanisDex` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., configure protocol parameters and add/adjust new reward pools). In the following, we use the `JanisMasterChef` contract as an example and show the representative functions potentially affected by the privileges of the owner.

```
    function updateJanisEmissionRate(uint _JanisPerSecond) public onlyOwner {
        require(_JanisPerSecond < 1e22, "emissions too high!");
        massUpdatePools();
        JanisPerSecond = _JanisPerSecond;
        emit UpdateJanisEmissionRate(msg.sender, _JanisPerSecond);
    }

    function updateYieldTokenEmissionRate(uint _yieldTokenPerSecond) public onlyOwner {
        require(_yieldTokenPerSecond < 1e22, "emissions too high!");
        massUpdatePools();
        yieldTokenPerSecond = _yieldTokenPerSecond;
        emit UpdateYieldTokenEmissionRate(msg.sender, _yieldTokenPerSecond);
```

```
    }
```

Listing 3.3: Example Privileged Operations in the `JanisMasterChef` Contract

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the privileged account may also be a counter-party risk to the protocol users. It is worrisome if the privileged account is a plain EOA account. 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.

**Recommendation** Promptly transfer the privileged accounts 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 confirmed and the team plans to use a multi-sig to manage the admin account.

## 3.4 Double Minting of Janis Reward in JanisMinter

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: High
- Target: `JanisMinter`
- Category: Business Logic [7]
- CWE subcategory: CWE-837 [4]

### Description

To facilitate the token issuance and management, the `JanisDex` protocol has a `JanisMinter` contract, which is the sole entity to have the privilege to mint new `JanisToken`. While reviewing various scenarios for token issuance, we notice a specific routine has a flawed implementation.

To elaborate, we show below the related `mintReferralsOnly()` function. It implements a rather straightforward logic in minting the commission to the referrer as well as the reward to the referee, i.e., the user. It comes to our attention the user reward is incorrectly minted as it mints the `_minting` amount (line 135) one more time!

```
119     function mintReferralsOnly(address _user, uint _minting) public onlyOperator {
120         uint commission = _minting * referralBonusE4 / 1e4;
121         uint reward =  _minting * refereeBonusE4 / 1e4;
122 
123         address referrer = referrers[_user];
124 
125         if (referrer != address(0) && _user != address(0) && commission > 0) {
```

```
            totalReferralCommission[referrer] += commission;
            totalReferralCommissionPerUser[referrer][_user] += commission;

            JanisToken.mint(referrer, commission);

            emit JanisMinted(referrer, commission);
            emit ReferralCommissionRecorded(referrer, _user, commission);
        }
        if (_user != address(0) && referrer != address(0) && reward > 0) {
            JanisToken.mint(_user, _minting);
            totalRefereeReward[_user] += reward;
            totalRefereeRewardPerReferrer[_user][referrer] += reward;

            JanisToken.mint(_user, reward);

            emit JanisMinted(_user, reward);
            emit ReferralCommissionRecorded(_user, referrer, reward);
        }
    }
```

Listing 3.4: `JanisMinter::mintReferralsOnly()`

**Recommendation** Revise the above routine to ensure only commission and reward are minted.

**Status** The issue has been fixed with the suggestion implemented.

## 3.5 Inconsistent Protocol Parameter Enforcement in JanisMasterChef

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `JanisMasterChef`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

### Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `JanisDEX` protocol is no exception. Specifically, if we examine the `JanisMasterChef` contract, it has defined a number of protocol-wide risk parameters, such as `depositFeeBPOrNFTETHFee` and `JanisPerSecond`. In the following, we show the corresponding routines that allow for their changes.

```
    function setDepositFeeOnly(uint _pid, uint _depositFeeBPOrNFTETHFee, bool
        _withMassUpdate) public onlyOwner {
        if (_withMassUpdate) {
            massUpdatePools();
```

```
        } else {
            updatePool(_pid);
        }

        poolInfo[_pid].depositFeeBPOrNFTETHFee = _depositFeeBPOrNFTETHFee;
    }

    function setPoolScheduleKeepMultipliers(uint _pid, uint _startTime, uint _endTime,
        bool _withMassUpdate) external onlyOwner {
        require(_startTime == 0   _startTime > block.timestamp, "invalid startTime!");
        require(_endTime == 0   (_startTime == 0 && _endTime > block.timestamp + 20)   (
            _startTime > block.timestamp && _endTime > _startTime + 20), "invalid
            endTime!");

        if (_withMassUpdate) {
            massUpdatePools();
        } else {
            updatePool(_pid);
        }

        uint lastRewardTime = _startTime == 0 ? (block.timestamp > globalStartTime ?
            block.timestamp : globalStartTime) : _startTime;

        poolInfo[_pid].lastRewardTime = lastRewardTime;
        poolInfo[_pid].endTime = _endTime;
    }
```

Listing 3.5: JanisMasterChef::setDepositFeeOnly()/setPoolScheduleKeepMultipliers()

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, the `depositFeeBPOrNFTETHFee` parameter has the expection of `require(_depositFeeBPOrNFTETHFee <= 10000, "too high fee")`, which is not enforced in the above `setDepositFeeOnly()`.

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

**Status** The issue has been fixed by implementing the above suggestion.

# 4 Conclusion

In this audit, we have analyzed the `JanisDex` protocol design and implementation. It is a `UniswapV2`-based decentralized exchange on `Arbitrum One`. Features include a community fairlaunch release method, multiple token rewards, a transaction fee pool for `J` tokens, platform dividends received by ownership token holders, `extinction pools` that sacrifice user deposits in exchange for yield, `NFT` deposit pools, and `NFT` yield boosting for farms and pools that reads `NFT` stats. 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-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

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

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

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

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

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

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