SMART CONTRACT AUDIT REPORT

for

Pandora Protocol

Prepared By: Yiqun Chen

PeckShield

January 30, 2022

Document Properties

Client	Pandora Protocol
Title	Smart Contract Audit Report
Target	Pandora
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Patrick Liu, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	January 30, 2022	Xuxian Jiang	Final Release
1.0-rc1	January 25, 2022	Xuxian Jiang	Release Candidate #1

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 design document and related smart contract source code of the `Pandora` 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 Pandora

`Pandora` recognizes the shortcomings of existing `DEXs` (in mainly incentivizing farmers without rewards for traders) and proposes an inclusive reward system that offers traders and farmers sustainable income and multiple benefits in an attempt to maintain high user retention rates. By gamifying its protocol, `Pandora` attracts users and keeps them engaged as well as creates a user-centered decentralized ecosystem where all participants are properly incentivized and empowered to make decisions on governance. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Pandora

Item	Description
Name	Pandora Protocol
Website	https://pandora.digital/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	January 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/PandoraDigital/smart-contract.git (063def7)

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

- https://github.com/PandoraDigital/smart-contract.git (d0aa319)

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

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 the 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 Audit Checklist

Category	Checklist Items
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 checklist of 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 `Pandora` 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 logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	1	■
Medium	2	■ ■
Low	4	■ ■ ■ ■
Informational	0	
Total	7	

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 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 1 high-severity vulnerability, 2 medium-severity vulnerabilities, and 4 low-severity vulnerabilities.

Table 2.1: Key Pandora Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Possible Randomness Perturbance in Random::computeSeed()	Coding Practices	Resolved
PVE-002	Low	Improved Sanity Checks For System Parameters	Coding Practices	Resolved
PVE-003	Medium	Possible Sandwich/MEV For Reduced Returns	Time And State	Resolved
PVE-004	Low	Timely Minting of Rewards Before Allocation/Rate Update	Business Logic	Resolved
PVE-005	Low	System Fee Bypass With Direct safeTransferFrom()	Business Logic	Mitigated
PVE-006	High	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-007	Medium	Proper Withdrawal Logic in Farming	Business Logic	Resolved

Besides 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 Possible Randomness Perturbance in Random::computeSeed()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Random
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [3]

Description

The Pandora protocol is innovative in instilling gaming elements in the DEX/DeFi design. In this process, there is a natural need of computing the randomness in different settings. While reviewing the randomness logic, we notice potential perturbance in current implementation.

To elaborate, we show below the related computeSeed() routine from the Random contract. This routine computes the random seed based on a number of factors, including block.timestamp, block.gaslimit, block.number, block.coinbase, and tx.origin. Note the miner (or the current block producer) is in the position of being capable of adjusting the block.timestamp, block.timestamp, and block.gaslimit, which could greatly affect the generated seed.

```
    function computerSeed(uint256 salt) internal view returns (uint256) {
        uint256 seed =
        uint256(
            keccak256(
                abi.encodePacked(
                    (block.timestamp)
                    + block.gaslimit
                    + uint256(keccak256(abi.encodePacked(blockhash(block.number)))) / (
                        block.timestamp)
                    + uint256(keccak256(abi.encodePacked(block.coinbase))) / (block.
                        timestamp)
                    + (uint256(keccak256(abi.encodePacked(tx.origin)))) / (block.
                        timestamp)
```

```
                    + block.number * block.timestamp
                )
            )
        );
//          seed = (seed % PRECISION) * getLatestPrice(BNB);
//          seed = (seed % PRECISION) * getLatestPrice(ETH);
//          seed = (seed % PRECISION) * getLatestPrice(BTC);
        if (salt > 0) {
            seed = seed % PRECISION * salt;
        }
        return seed;
    }
```

Listing 3.1: `Random::computeSeed()`

From another perspective, we have to admit that the randomness in the blockchain is a hard issue. And there is a need to be aware of the inherent weakness of current randomness schemes and proactively develop mitigation solutions.

Recommendation Explore possible improvements to mitigate the above randomness issue.

Status The issue has been fixed by including the off-chain oracles in the following commit: `b3b7db8`.

3.2 Improved Sanity Checks For System Parameters

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `Pandora` protocol is no exception. Specifically, if we examine the `NFTRouter` contract, it has defined a number of system-wide risk parameters, e.g., `createPandoBoxFee`, `upgradeBaseFee`, and `pandoBoxPerDay`. In the following, we show the representative `setter` routines that allow for their update.

```
    function setPandoBoxPerDay(uint256 _value) external onlyOwner {
        pandoBoxPerDay = _value;
    }

    function setCreatePandoBoxFee(uint256 _newFee) external onlyOwner {
        createPandoBoxFee = _newFee;
```

```
    }

    function setUpgradeBaseFee(uint256 _newFee) external onlyOwner {
        upgradeBaseFee = _newFee;
    }

    function setJackpotAddress(address _addr) external onlyOwner {
        pandoPot = IPandoPot(_addr);
    }
```

Listing 3.2: Example Setters in NFTRouter

This parameter defines an important aspect of the protocol operation and needs to exercise extra care when configuring or updating it. Our analysis shows the configuration logic on it can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to undesirable consequences. For example, an unlikely mis-configuration of `createPandoBoxFee` may bring high cost for protocol users and hurt the protocol adoption.

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

Status The team has confirmed that there is no need to validate these risk parameters. After the deployment, the team will exercise extra caution when configuring them.

3.3 Possible Sandwich/MEV For Reduced Returns

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `Pandora` protocol has a unique incentivize mechanism that aims to engage trading and farming users. This is proposed to address the shortcomings of current `DEX`s and improve the user retention rates. Within this process, there is a constant need of swapping one token to another. While reviewing the related token-swapping logic, we notice the current implementation may be improved.

To elaborate, we show below the related `Treasury::_swap()` routine. As the name indicates, it has a rather straightforward logic in swapping the given amount of `fromToken` to `toToken`.

```
    function _swap(
        address fromToken,
        address toToken,
```

```
        uint256 amountIn
    ) internal returns (uint256 amountOut) {
        // Checks
        // X1 - X5: OK
        IUniswapV2Pair pair =
            IUniswapV2Pair(factory.getPair(fromToken, toToken));
        require(address(pair) != address(0), "Treasury: Cannot convert");

        // Interactions
        // X1 - X5: OK
        (uint256 reserve0, uint256 reserve1, ) = pair.getReserves();
        uint256 amountInWithFee = amountIn.mul(997);
        if (fromToken == pair.token0()) {
            amountOut =
                amountInWithFee.mul(reserve1) /
                reserve0.mul(1000).add(amountInWithFee);
            IERC20(fromToken).safeTransfer(address(pair), amountIn);
            pair.swap(0, amountOut, address(this), new bytes(0));
            // TODO: Add maximum slippage?
        } else {
            amountOut =
                amountInWithFee.mul(reserve0) /
                reserve1.mul(1000).add(amountInWithFee);
            IERC20(fromToken).safeTransfer(address(pair), amountIn);
            pair.swap(amountOut, 0, address(this), new bytes(0));
            // TODO: Add maximum slippage?
        }
    }
```

Listing 3.3: `InitialLiquidityPool::_swap()`

We notice the current logic seems to validate the amount of returned amount. However, it is computed based on the instant DEX liquidity, which may be manipulated in frontrunning or MEV attacks. In other words, the current approach does not effectively specify the required restriction on possible slippage. As a result, it may result in a smaller amount of swapped amount.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation to the above front-running situations to better protect the interests of trading/farming users.

Status The issue has been fixed by this commit: `570dca6`.

3.4 Timely Minting of Rewards Before Allocation/Rate Update

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Medium
- Target: `Farming`, `PSRStaking`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `Pandora` protocol provides incentive mechanisms that reward the staking of supported assets. The rewards are carried out by designating a number of staking pools into which supported assets can be staked. And staking users are rewarded in proportional to their share of LP tokens in the reward pool.

The reward pools can be dynamically added via `add()` and the weights of supported pools can be adjusted via `set()`. When analyzing the pool weight update routine `set()`, we notice the need of timely invoking `massUpdatePools()` to update the reward distribution before the new pool weight becomes effective.

```
    function set(uint256 _pid, uint256 _allocPoint, IRewarder _rewarder, bool overwrite)
         public onlyOwner {
        totalAllocPoint = totalAllocPoint.sub(poolInfo[_pid].allocPoint).add(_allocPoint
            );
        poolInfo[_pid].allocPoint = _allocPoint.to64();
        if (overwrite) { rewarder[_pid] = _rewarder; }
        emit LogSetPool(_pid, _allocPoint, overwrite ? _rewarder : rewarder[_pid],
            overwrite);
    }

    function setRewardPerBlock(uint256 _rewardPerBlock) public onlyOwner {
        rewardPerBlock = _rewardPerBlock;
        emit LogRewardPerBlock(_rewardPerBlock);
    }
```

Listing 3.4: `Farming::set()`

If the call to `massUpdatePools()` is not immediately invoked before updating the pool weights, certain situations may be crafted to create an unfair reward distribution. Moreover, a hidden pool without any weight can suddenly surface to claim unreasonable share of rewarded tokens. Fortunately, this interface is restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern. Note similar routines from `TradingPool`, `PSRStaking`, and `Referral` contracts share the same issue.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight has been updated.

Status The issue has been fixed by this commit: `ec5ccb8`.

3.5 System Fee Bypass With Direct safeTransferFrom()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `NftMarket`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Pandora` protocol has a `NftMarket` contract that allows to trade assets as `ERC721`-based NFT tokens, which naturally follow the standard implementation, e.g., `transferFrom()/safeTransferFrom()` . By design, each tradable asset can be set by its owner with the selling `price`. Any interested user can buy it by fulfilling its price. When a price is fulfilled, the NFT token is transferred to the buyer. Some percentage (represented by `systemFeePercent / ONE_HUNDRED_PERCENT`) of the funds is transferred from that buyer to the `adminWallet` and the rest is transferred to the current seller.

To elaborate, we show below the `buy()` routine. This routine is provided to support trading on whitelisted `NFTs`. It comes to our attention that instead of transferring a `systemFeePayment` amount to `adminWallet` for each trade, it is possible for the current seller and the buyer to directly negotiate a price, without paying the `systemFeePayment`. The `NFT` can then be arranged and delivered by the current owner to directly call `transferFrom()/safeTransferFrom()` with the buyer as the recipient.

```
    function buy(address erc721, uint256 tokenId)
        public
        whenNotPaused
        nonReentrant
    {
        address msgSender = _msgSender();

        uint256 askId = currentAsks[erc721][tokenId];

        Ask memory info = asks[erc721][tokenId][askId];

        require(info.price > 0, "NftMarket: token price at 0 are not for sale");

        _payout(erc721, info.erc20, tokenId, info.price, msgSender, info.seller);

        IERC721(erc721).transferFrom(address(this), msgSender, tokenId);

        emit TokenSold(erc721, info.erc20, msgSender, info.seller, info.price, tokenId,
            askId);
```

```

        delete asks[erc721][tokenId][askId];
        delete currentAsks[erc721][tokenId];
    }
```

Listing 3.5: `NftMarket::buy()`

Recommendation Implement a locking mechanism so that any `NFT` tokens need to be locked in order to be only tradable in `Pandora`.

Status The issue has been mitigated by this commit: `ec5ccb8`.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [2]

Description

In the `Pandora` protocol, there is a special administrative account, i.e., `owner`. This `owner` account plays a critical role in governing and regulating the system-wide operations (e.g., configuring various incentives, setting protocol parameters, and adjusting external oracles). It also has the privilege to regulate or govern the flow of assets among the involved components.

With great privilege comes great responsibility. Our analysis shows that the `owner` account is indeed privileged. In the following, we show representative privileged operations in the `Pandora` protocol.

```
    function setMigrator(IMigratorChef _migrator) public onlyOwner {
        migrator = _migrator;
    }

    function migrate(uint256 _pid) public {
        require(address(migrator) != address(0), "MasterChefV2: no migrator set");
        IERC20 _lpToken = lpToken[_pid];
        uint256 bal = _lpToken.balanceOf(address(this));
        _lpToken.approve(address(migrator), bal);
        IERC20 newLpToken = migrator.migrate(_lpToken);
        require(bal == newLpToken.balanceOf(address(this)), "MasterChefV2: migrated
            balance must match");
        require(addedTokens[address(newLpToken)] == false, "Token already added");
        addedTokens[address(newLpToken)] = true;
        addedTokens[address(_lpToken)] = false;
```

```
        lpToken[_pid] = newLpToken;
    }
```

Listing 3.6: An Example Privileged Migration Operation in Farming

Note that the privilege assignment with various core contracts may be necessary and required for proper protocol operations. However, it is worrisome if the owner is not governed by a DAO-like structure. We point out that a compromised owner account would allow the attacker to drain the funds in the current farming contract and undermine necessary assumptions behind the protocol and subvert various protocol operations.

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 confirmed and partially mitigated with the planned timelock contract.

3.7 Proper Withdrawal Logic in Farming

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: Multiple Contracts
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, the Pandora protocol has a unique incentivize mechanism that aims to engage trading and farming users. While reviewing the current farming logic, we notice an important user-facing function needs to be improved.

In the following, we use the related withdrawAll() function. This function allows the user to withdraw all previously deposited funds from the farming contract. However, it comes to our attention that the given amount to the actual withdraw function withdraw() is 0 with the purpose of withdrawing all current deposits. A further examination on the withdraw() function shows the given 0 amount does not be interpreted as the full withdrawal! The inconsistency may bring unnecessary confusion to farming users and therefore needs to be resolved.

```
    function withdrawAll(address to) public {
        for (uint256 i = 0; i < poolInfo.length; i++) {
            withdraw(i, 0, to);
        }
```

```
    }
```

Listing 3.7: `Farming::withdrawAll()`

```
    function withdraw(uint256 pid, uint256 amount, address to) public {
        PoolInfo memory pool = updatePool(pid);
        UserInfo storage user = userInfo[pid][msg.sender];

        // Effects
        user.rewardDebt = user.rewardDebt.sub(int256(amount.mul(pool.accRewardPerShare)
            / ACC_PAN_PRECISION));
        user.amount = user.amount.sub(amount);

        // Interactions
        IRewarder _rewarder = rewarder[pid];
        if (address(_rewarder) != address(0)) {
            _rewarder.onReward(pid, msg.sender, to, 0, user.amount);
        }

        lpToken[pid].safeTransfer(to, amount);

        emit Withdraw(msg.sender, pid, amount, to);
    }
```

Listing 3.8: `Farming::withdraw()`

Recommendation Revise the inconsistency between `withdraw()` and `withdrawAll()` functions in performing a full withdrawal.

Status The issue has been fixed by this commit: `570dca6`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Pandora` protocol, which proposes an inclusive reward system that offers traders and farmers sustainable income and multiple benefits in an attempt to maintain high user retention rates. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` 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-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.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.