SMART CONTRACT AUDIT REPORT

for

FusionX

Prepared By: Xiaomi Huang

PeckShield

July 20, 2023

Document Properties

Client	FusionX Finance
Title	Smart Contract Audit Report
Target	FusionX
Version	1.0
Author	Luck Hu
Auditors	Luck Hu, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	July 20, 2023	Luck Hu	Final Release
1.0-rc1	July 18, 2023	Luck Hu	Release Candidate #1

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 FusionX 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 audited protocol 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 FusionX

FusionX builds the native DeFi ecosystem on the Mantle network. The audited FusionX V3 is designed and implemented based on PancakeSwap V3, which uses Uniswap V3's core design but extends with liquidity provider incentives. The MasterChefV3 allows liquidity providers to farm their position NFTs to earn FUSIONX. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of FusionX

Item	Description
Name	FusionX Finance
Website	https://fusionx.finance
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	July 20, 2023

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

- https://github.com/FusionX-Finance/v3-contracts-for-audit/tree/fusionx-v3 (715e606)

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

- https://github.com/FusionX-Finance/v3-contracts-for-audit/tree/fusionx-v3 (e868576)

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).

Table 1.2: Vulnerability Severity Classification

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

Likelihood

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;
- 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

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

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) [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 `FusionX` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	0	
Informational	1	■
Total	3	

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 and 1 informational recommendation.

Table 2.1: Key FusionX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Timely massUpdatePools() in MasterChefV3	Business Logic	Acknowledged
PVE-002	Informational	Improved Naming for Position NFT	Coding Practices	Fixed
PVE-003	Medium	Trust Issue of Admin Keys	Security Features	Acknowledged

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 Timely massUpdatePools() in MasterChefV3

- ID: PVE-001
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: MasterChefV3
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The MasterChefV3 contract provides an incentive mechanism that rewards the staking of FusionXV3 position NFTs with the FUSIONX tokens. 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 active liquidity behind their positions that are deposited in the 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, bool _withUpdate) external onlyOwner
        onlyValidPid(_pid) {
        uint32 currentTime = uint32(block.timestamp);
        PoolInfo storage pool = poolInfo[_pid];
        ILMPool LMPool = ILMPool(pool.v3Pool.lmPool());
        if (address(LMPool) != address(0)) {
            LMPool.accumulateReward(currentTime);
        }

        if (_withUpdate) massUpdatePools();
        totalAllocPoint = totalAllocPoint - pool.allocPoint + _allocPoint;
        pool.allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.1: MasterChefV3::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, these interfaces are restricted to the owner (via the `onlyOwner` modifier), which greatly alleviates the concern. Note the same issue is also applicable to the `add()/set()/upkeep()` routines, etc.

Recommendation Timely invoke `massUpdatePools()` when any pool's weight or the reward rate has been updated. In fact, the `_withUpdate` parameter to the `set()`, `add()` and `upkeep()` routines can be simply ignored or removed.

```
    function set(uint256 _pid, uint256 _allocPoint, bool _withUpdate) external onlyOwner
         onlyValidPid(_pid) {
        uint32 currentTime = uint32(block.timestamp);
        PoolInfo storage pool = poolInfo[_pid];
        ILMPool LMPool = ILMPool(pool.v3Pool.lmPool());
        if (address(LMPool) != address(0)) {
            LMPool.accumulateReward(currentTime);
        }

        massUpdatePools();
        totalAllocPoint = totalAllocPoint - pool.allocPoint + _allocPoint;
        pool.allocPoint = _allocPoint;
        emit SetPool(_pid, _allocPoint);
    }
```

Listing 3.2: Revised MasterChefV3::set()

Status The issue has been acknowledged by the team.

3.2 Improved Naming for Position NFT

- ID: PVE-002
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: Multiple Contracts
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

The `FusionX` protocol is designed and implemented based on `PancakeSwap V3`. While reviewing the naming of the submodules in `FusionX`, we notice it still uses the name of `Pancake`.

In the following, we show below the code snippet of the `NonfungiblePositionManager::constructor()` routine. The `NonfungiblePositionManager` contract wraps `FusionX V3` positions in the `ERC721` non-fungible token interface. However, in the initialization of `ERC721` specific properties, e.g., name/symbol, it still uses the name of `Pancake/PCS` (line 76), which may introduce confusion to users. Based on this, we suggest to improve the initialization of the position `NFT` with `FusionX` specific naming.

```
71   constructor(
72     address _deployer,
73     address _factory,
74     address _WETH9,
75     address _tokenDescriptor_
76   ) ERC721Permit('Pancake V3 Positions NFT-V1', 'PCS-V3-POS', '1')
       PeripheryImmutableState(_deployer, _factory, _WETH9) {
77     _tokenDescriptor = _tokenDescriptor_;
78   }
```

Listing 3.3: `NonfungiblePositionManager::constructor()`

What's more, while reviewing the token `URI` generation for a position `NFT` in the `NFTDescriptorEx::generateName()` routine, we notice it also uses the name of `Pancake` (line 187). As a result, the generated token `URI` string contains the name of `Pancake`. Based on this, we suggest to correct this with `FusionX` naming as well. Note the same issue is also applicable to the `NFTDescriptor::generateName()` routine.

```
179     function generateName(ConstructTokenURIParams memory params, string memory feeTier)
180     private
181     pure
182     returns (string memory)
183     {
184     return
185         string(
186             abi.encodePacked(
187                 'Pancake - ',
188                 feeTier,
189                 ' - ',
190                 escapeQuotes(params.quoteTokenSymbol),
191                 '/',
192                 escapeQuotes(params.baseTokenSymbol),
193                 ' - ',
194                 tickToDecimalString(
195                     !params.flipRatio ? params.tickLower : params.tickUpper,
196                     params.tickSpacing,
197                     params.baseTokenDecimals,
198                     params.quoteTokenDecimals,
199                     params.flipRatio
200                 ),
201                 '<>',
202                 tickToDecimalString(
203                     !params.flipRatio ? params.tickUpper : params.tickLower,
204                     params.tickSpacing,
205                     params.baseTokenDecimals,
206                     params.quoteTokenDecimals,
207                     params.flipRatio
208                 )
209             )
```

```
        );
    }
```

Listing 3.4: NFTDescriptorEx::generateName()

Recommendation Correct the position NFT name/symbol and the token URI with FusionX specific naming.

Status This issue has been fixed in the following commit: e868576.

3.3 Trust Issue of Admin Keys

- ID: PVE-003
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the FusionX protocol, there is a privileged owner account that plays a critical role in governing and regulating the protocol-wide operations (e.g., set pool weight in MasterChefV3). Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the MasterChefV3 contract as an example and show the representative functions potentially affected by the privileges of the owner account.

Specifically, the owner account is privileged to add new reward pool, set the pool weight, set emergency mode, and set the farm booster, etc.

```
    function setWhitelist(address _addr, bool isWhiteUser) external onlyOwner {
        whitelist[_addr] = isWhiteUser;
    }

    function setWhitelists(address[] calldata _addrs, bool isWhiteUser) external
        onlyOwner {
        for (uint256 i = 0; i < _addrs.length; i++) {
            whitelist[_addrs[i]] = isWhiteUser;
        }
    }

    function withdrawRaisingToken(uint256 _amount) external onlyOwner {
        require(block.timestamp > endTime, "not withdraw time");
        require(_amount <= address(this).balance, "not enough token");
        _safeTransferETH(msg.sender, _amount);
    }
```

```
    function withdrawOfferingToken(uint256 _amount) external onlyOwner {
        require(block.timestamp > endTime, "not withdraw time");
        require(_amount <= offeringToken.balanceOf(address(this)), "not enough token");
        address(offeringToken).safeTransfer(msg.sender, _amount);
    }
```

Listing 3.5: Example Privileged Operations in `MasterChefV3`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to the privileged account may also be a counter-party risk to the contract 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 Promptly transfer the administrative privileges to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status The issue has been acknowledged by the team.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `FusionX` protocol, which builds the native `DeFi` ecosystem on the `Mantle` network. The audited `FusionX V3` is designed and implemented based on `PancakeSwap V3`, which uses `Uniswap V3`'s core design but extends with liquidity provider incentives. The `MasterChefV3` allows liquidity providers to farm their position `NFTs` to earn `FUSIONX`. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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.