SMART CONTRACT AUDIT REPORT

for

IFlaunchpad

Prepared By: Yiqun Chen

PeckShield

August 2, 2021

Document Properties

Client	Impossible Finance
Title	Smart Contract Audit Report
Target	IFlaunchpad
Version	1.0-rc
Author	Yiqun Chen
Auditors	Yiqun Chen, Xuxian Jian
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc	August 2, 2021	Yiqun Chen	Release Candidate

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 `IFlaunchpad` 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 IFlaunchpad

`Impossible Finance` is a multi-chain incubator, launchpad, and swap platform, offering a robust product-first ecosystem supporting top-tier blockchain projects with launching to targeted user audiences. As a part of the platform, `IFlaunchpad` is a fair platform for conducting IDOs for its projects. It features a new staking mechanism. Different from other launchpads, `IFlaunchpad` would not mint tokens for the user straightly. Instead, it increases the allowance of the user for purchasing sale tokens with specific tokens based on the allocation obtained through staking over time.

The basic information of the `IFlaunchpad` protocol is as follows:

Table 1.1: Basic Information of The `IFlaunchpad` Protocol

Item	Description
Issuer	Impossible Finance
Website	https://impossible.finance/
Type	BSC Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	August 2, 2021

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

- https://github.com/ImpossibleFinance/launchpad-contracts (12228fa)

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

- https://github.com/ImpossibleFinance/launchpad-contracts (cfd6d48)

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.

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 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.

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 `IFlaunchpad` 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	0	
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	4	

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, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key IFlaunchpad Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Improved Validation Of purchase()/whitelistedPurchase()	Coding Practices	Fixed
PVE-002	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed
PVE-003	Informational	Incorrect Event Data In emergencyTokenRetrieve()	Time and State	Fixed
PVE-004	Medium	Incorrect Argument Used In stake()	Coding Practices	Fixed

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 Improved Validation Of purchase()/whitelistedPurchase()

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: IFAllocationSale
- Category: Coding Practices [6]
- CWE subcategory: CWE-391 [3]

Description

The IFlaunchpad protocol provides incentive mechanisms that reward the staking of supported assets by increasing the allowance of the user for purchasing sale tokens with specific tokens based on the allocation obtained through staking over time. The price of the sale token is set during the deployment of the contract, and if the sale price is zero, then participants could get specific amount of sale tokens for free.

In the following, we list below the _purchase() function.

```
    // Internal function for making purchase in allocation sale
    // Used by external functions `purchase` and `whitelistedPurchase`
    function _purchase(uint256 paymentAmount) internal nonReentrant {
        // sale must be active
        require(startBlock <= block.number, 'sale has not begun');
        require(block.number <= endBlock, 'sale over');

        // amount must be greater than minTotalPayment
        // by default, minTotalPayment is 0 unless otherwise set
        require(paymentAmount > minTotalPayment, 'amount below min');

        // get user allocation as ratio (multiply by 10**18, aka E18, for precision)
        uint256 userWeight = allocationMaster.getUserStakeWeight(
            trackId,
            _msgSender(),
            allocSnapshotBlock
        );
        uint256 totalWeight = allocationMaster.getTotalStakeWeight(
```

```
            trackId,
            allocSnapshotBlock
        );

        // total weight must be greater than 0
        require(totalWeight > 0, 'total weight is 0');

        // determine allocation
        uint256 paymentTokenAllocation;

        // different calculation for whether override is set
        if (saleTokenAllocationOverride == 0) {
            // calculate allocation (times 10**18)
            uint256 allocationE18 = (userWeight * 10**18) / totalWeight;

            // calculate max amount of obtainable sale token
            uint256 saleTokenAllocationE18 = (saleAmount * allocationE18);

            // calculate equivalent value in payment token
            paymentTokenAllocation =
                (saleTokenAllocationE18 * salePrice) /
                SALE_PRICE_DECIMALS /
                10**18;
        } else {
            // override payment token allocation
            paymentTokenAllocation =
                (salePrice * saleTokenAllocationOverride) /
                SALE_PRICE_DECIMALS;
        }

        // console.log('sale token allocation', saleTokenAllocationE18 / 10**18);
        // console.log('payment token allocation', paymentTokenAllocation);

        // total payment received must not exceed max payment amount
        require(
            paymentReceived[_msgSender()] + paymentAmount <= maxTotalPayment,
            'exceeds max payment'
        );
        // total payment received must not exceed paymentTokenAllocation
        require(
            paymentReceived[_msgSender()] + paymentAmount <=
                paymentTokenAllocation,
            'exceeds allocation'
        );

        // transfer specified amount from user to this contract
        paymentToken.safeTransferFrom(
            address(_msgSender()),
            address(this),
            paymentAmount
        );

```

```
        // if user is paying for the first time to this contract, increase counter
        if (paymentReceived[_msgSender()] == 0) purchaserCount += 1;

        // increase payment received amount
        paymentReceived[_msgSender()] += paymentAmount;

        // emit
        emit Purchase(_msgSender(), paymentAmount);
    }
```

Listing 3.1: `IFAllocationSale::_purchase()`

As shown in the above implementation, the user does not have to purchase for sale tokens if the sale price is zero, and if the user tries to purchase, then it will revert with the result `exceeds allocation`. Although there is no loss here, it brings unnecessary confusion to the user. With that, we suggest to check whether the user tries to purchase with the zero sale price or not.

Recommendation Properly add explicit validation for free sale tokens in the internal `_purchase()` function.

Status The issue has been addressed in this commit: `879e356`.

3.2 Trust Issue Of Admin Keys

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `IFlaunchpad` protocol, there are two special administrative accounts, i.e., two `owner` accounts in the `IFAllocationMaster` and the `IFAllocationSale` contracts. These `owner` accounts play a critical role in governing and regulating the protocol-wide operations (e.g., setting various parameters, authorizing other roles). They also have the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that these privileged accounts need to be scrutinized. In the following, we examine the privileged `owner` account in the `IFAllocationMaster` contract and one of its related privileged accesses in current contract.

To elaborate, we show below the `disableTrack()` routine. This routine disables a track which used for marking stake information, and once the track is disabled, users are not allowed to deposit for staking. In the same time, they are not allowed to withdraw as well.

```
    // disables a track
    function disableTrack(uint24 trackId) external onlyOwner {
        // add a new checkpoint with 'disabled' set to true
        addTrackCheckpoint(trackId, 0, false, true, false);

        // 'DisableTrack' event emitted in function call above
    }
```

Listing 3.2: `IFAllocationMaster::disableTrack()`

It is worrisome if both these two privileged `owner` accounts are plain EOA accounts. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. The discussion with the team has confirmed that this privileged account will be managed by a governance contract. 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 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.

3.3 Incorrect Event Data In emergencyTokenRetrieve()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `IFAllocationSale`
- Category: Time and State [5]
- CWE subcategory: CWE-362 [2]

Description

Meaningful events are an important part in smart contract design as they can not only greatly expose the runtime dynamics of smart contracts, but also allow for better understanding about their behavior and facilitate off-chain analytics. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed.

In the following, we list below the `emergencyTokenRetrieve()` function with the emitted `EmergencyTokenRetrieve()` event.

```
    function emergencyTokenRetrieve(address token) external onlyOwner {
        // cannot be payment or sale tokens
        require(token != address(paymentToken));
        require(token != address(saleToken));
```

```
        // transfer all
        ERC20(token).safeTransfer(
            _msgSender(),
            ERC20(token).balanceOf(address(this))
        );

        // emit
        emit EmergencyTokenRetrieve(
            _msgSender(),
            ERC20(token).balanceOf(address(this))
        );
    }
```

Listing 3.3: `IFAllocationSale::emergencyTokenRetrieve()`

As shown in the above implementation, the `EmergencyTokenRetrieve()` event records the address of the receiver (`owner`) and the amount of tokens transferred to this address. However, the amount will always be zero as it records the remaining tokens in the `IFAllocationSale` contract.

Recommendation Revise the emitted `EmergencyTokenRetrieve()` event to record correct amount.

Status The issue has been addressed in this commit: `8b8747b`.

3.4 Incorrect Argument Used In stake()

- ID: PVE-004
- Severity: Medium
- Likelihood: High
- Impact: Medium
- Target: `IFAllocationMaster`
- Category: Coding Practices [6]
- CWE subcategory: CWE-559 [7]

Description

There are several data structures (e.g., `TrackCheckpoint`, `UserCheckpoint`, `TrackInfo`) defined in the `IFAllocationMaster` contract. These data structures are used for persisting the whole staking states of the protocol. When the user tries to deposit for staking, there will be two checkpoints added (e.g., `TrackCheckpoint` and `UserCheckpoint`). However, the user is not allowed to stake into a disabled track, so before that, it will check the status of the previous `TrackCheckpoint`.

In the following, we list below the `stake()` function.

```
    function stake(uint24 trackId, uint104 amount) external nonReentrant {
        // stake amount must be greater than 0
        require(amount > 0, 'amount is 0');

        // get track info
```

```
125         TrackInfo storage track = tracks[trackId];

127         // get latest track checkpoint
128         TrackCheckpoint storage checkpoint =
129             trackCheckpoints[trackId][trackCheckpointCounts[trackId]];

131         // cannot stake into disabled track
132         require(!checkpoint.disabled, 'track is disabled');

134         // transfer the specified amount of stake token from user to this contract
135         track.stakeToken.safeTransferFrom(_msgSender(), address(this), amount);

137         // add user checkpoint
138         addUserCheckpoint(trackId, amount, true);

140         // add track checkpoint
141         addTrackCheckpoint(trackId, amount, true, false, false);

143         // emit
144         emit Stake(trackId, _msgSender(), amount);
145     }
```

Listing 3.4: `IFAllocationMaster::stake()`

We notice a wrong argument used in the above implementation(line 128). To elaborate, we show the code snippet of the `stake()` function. The statement is `TrackCheckpoint storage checkpoint = trackCheckpoints[trackId][trackCheckpointCounts[trackId]]`. As the comment noted, it tries to get the latest track checkpoint. However, the result of `trackCheckpointCounts[trackId]` is the amount of check points in this track. And the argument input should be the index of the check point instead of the amount number.

Recommendation Change the statement shown above to `TrackCheckpoint storage checkpoint = trackCheckpoints[trackId][trackCheckpointCounts[trackId]-1]`.

Status The issue has been addressed in this commit: `8b8747b`.

4 Conclusion

In this audit, we have analyzed the `IFlaunchpad` protocol design and implementation. The `IFlaunchpad` protocol provides a decentralized liquidity platform for conducting fair, one-off sales where users have guaranteed allocations managed by the `IFAllocationMaster` contract.

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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

[3] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

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

[5] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.html.

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

[7] MITRE. CWE CATEGORY: Often Misused: Arguments and Parameters. https://cwe.mitre.org/data/definitions/559.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.