BlockSec

Security Audit Report for Impossible Finance

Aug 2, 2021

Report Manifest

Item	Description
Client	Impossible Finance
Target	Impossible Finance Launchpad
Version	1.3
Author	Lei Wu
Auditors	Siwei Wu, Hang Feng, Lei Wu, Yajin Zhou
Reviewed by	Lei Wu
Approved by	Yajin Zhou

Version History

Version	Date	Description
1.3	Aug 02, 2021	Add maximum total stake cap
1.2	July 27, 2021	Minor changes after discussion
1.1	July 24, 2021	Minor changes after discussion
1.0	July 22, 2021	First Release
1.0-rc2	July 20, 2021	Release Candidate 2
1.0-rc1	July 18, 2021	Release Candidate 1

Content

1. Introduction

1.1 About Impossible Finance Launchpad

The Impossible Launchpad is a fair platform for conducting IDOs for Impossible projects. It is proposed by Impossible Finance[1], 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.

Project Information	Description
Website	https://docs.impossible.finance/launchpad/overview
Type	Smart Contract
Programming Language	Solidity
Git Repository	https://github.com/ImpossibleFinance/launchpad-contracts
Audit Approach	Semi-automatic and manual verification

The commit hash value of the initial reviewed files is:
6334cf50449c4db363fecf2e70ed453e0864b695
The commit hash value after checking in the fixes reported by this audit is: *cfd6d48428d266795ef8ef73e8aa0ddd08a7f354.*

1.2 About BlockSec

BlockSec [1] is a research group founded by security researchers from

[1] For brevity, we will use IF in the following of this report.

Zhejiang University. The team is mainly focusing on the security of the blockchain ecosystem. They have published multiple research papers in prestigious security conferences, reported several zero-day attacks of DeFi applications, and released detailed analysis reports of high impact security incidents. They can be reached at email (contact@blocksecteam.com) and twitter (https://twitter.com/BlockSecTeam).

1.3 Disclaimer

This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the inexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts. Besides, this report does not constitute personal investment advice or a personal recommendation.

1.4 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **DeFi Semantic Analysis** We study the business logic and the economic model of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas

optimization, code style, and etc.

The concrete checkpoints are as following：

Checklist	
Software Security	Reentrancy
	DoS
	Access Control
	Data Handling/Data Flow
	Exception Handling
	Untrusted External Call/ Control Flow
	Initialization Consistency
	Events Operation
	Error-prone Randomness
	Improper Use of the Proxy System
DeFi Security	Semantic Consistency Functionality Consistency Business Logic
	Token Operation
	Emergency Mechanism
	Oracle Security
	Whitelist/ Blacklist
	Economic Impact
Additional Recommendation	Gas Optimization
	Code Quality & Style
	Deprecated Uses

1.5 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including *OWASP Risk Rating Methodology* [2] and *Common Weakness Enumeration* [3]. Accordingly, the severity measured in this report are classified into four categories: ***High***, ***Medium***, ***Low*** and ***Undetermined***.

2. Summary of Findings

In total, we have identified 7 potential issues, as follows:

- High Risk: 0
- Medium Risk: 1
- Low Risk: 2
- Undetermined: 4

ID	Severity	Description	Category
BWE-1	Medium	Incorrect Array Index	Software Security
BWE-2	Low	Unchecked Function Parameters	Software Security
BWE-3	Undetermined	Potential Unfair staking	DeFi Security
BWE-4	Undetermined	Neglection of Potential Token Deflation	DeFi Security
BWE-5	Undetermined	Potential Abuse of _trackId	DeFi Security
BWE-6	Undetermined	Inconsistency between Documentation and Code	DeFi Security
BWE-7	Low	Follow the Checks-Effects-Interactions Pattern	Additional Recommendation

The details are provided in the next section.

3. Detailed Findings

3.1 Software Security

3.1.1 Incorrect Array Index

ID: BWE-1	**Type: Data Handling**
Risk level: Medium	Target: IFAllocationMaster.sol
Status: Fixed	

Description In the function stake() at line 676 of the contract, the index of the array is not reduced by 1 when obtaining the latest check point of the track.

```
667     function stake(uint24 trackId, uint104 amount) external nonReentrant {
668         // stake amount must be greater than 0
669         require(amount > 0, 'amount is 0');
670 
671         // get track info
672         TrackInfo storage track = tracks[trackId];
673 
674         // get latest track checkpoint
675         TrackCheckpoint storage checkpoint =
676             trackCheckpoints[trackId][trackCheckpointCounts[trackId]];
677 
678         // cannot stake into disabled track
679         require(!checkpoint.disabled, 'track is disabled');
680 
681         // transfer the specified amount of stake token from user to this contract
682         track.stakeToken.safeTransferFrom(_msgSender(), address(this), amount);
683 
684         // add user checkpoint
685         addUserCheckpoint(trackId, amount, true);
686 
687         // add track checkpoint
688         addTrackCheckpoint(trackId, amount, true, false, false);
689 
690         // emit
691         emit Stake(trackId, _msgSender(), amount);
692     }
```

Impact The storage variable checkpoint will get a null value, which will bypass the required check for *checkpoint.disabled* at line 679.

Suggestion Change the code to the following:

trackCheckpoints[trackId][trackCheckpointCounts[trackId]-1]

3.1.2 Unchecked Function Parameters

ID: BWE-2	**Type: Data Handling**
Risk level: Low	Target: IFAllocationSale.sol
Status: Fixed	

Description The parameters in the constructor of this contract are not checked, e.g., funder, to avoid potential misuse.

Impact The parameters cannot be modified after being set incorrectly.

Suggestion Verify these parameters in the constructor.

3.2 DeFi Security

3.2.1 Potential Unfair staking

ID: BWE-3	**Type: Economic Impact**
Risk level: Undetermined	Target: IFAllocationSale.sol
Status: Fixed. IF prevents this issue by adding the maximum total stake cap.	

Description The user's allocation for the staking of a platform token is determined by the stake amount held over time. However, a user can listen to the addTrack transaction and immediately stake a platform token. Since the user is the first one who participates into the IDO, he or she can have a better opportunity to get a large allocation than later participants. Then the user can unstake and withdraw his/her stake tokens without purchasing before the ending of the sale. This can prevent other fair participants who want to buy into an IDO, causing a denial of service.

Impact The can potentially cause a denial of service to an IDO.

Suggestion This issue can be prevented by applying some thresholds to avoid extremely large allocations.

3.2.2 Neglection of Potential Token Deflation

ID: BWE-4	**Type: Economic Impact**
Risk level: Undetermined	Target: IFAllocationMaster.sol IFAllocationSale.sol
Status: Confirmed. However, it should not be an issue for the IF launchpad because all the tokens will undergo a manual review process.	

Description The purchase and withdraw functions in the sale contract have a potential risk when performing the token transfer. If the token is a deflation token, there will be a difference between the recorded amount of transferred tokens in the sale smart contract and the actual number of transferred tokens. That's because a small amount of tokens will be burned by the token smart contract. This inconsistency could cause security impacts if some critical operations are based on the recorded amount of transferred tokens.

```
function _purchase(uint256 paymentAmount) internal nonReentrant {
    // sale must be active
    require(startBlock <= block.number, 'sale has not begun');
    require(block.number <= endBlock, 'sale over');

    // sale price must not be 0, which is a giveaway sale
    require(salePrice != 0, 'cannot purchase - giveaway sale');

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

    // get max payment of user
    uint256 remaining = getMaxPayment(_msgSender());

    // payment must not exceed remaining
    require(paymentAmount <= remaining, 'exceeds max payment');

    // 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;

    // increase total payment received amount
    totalPaymentReceived += paymentAmount;

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

```
function withdraw() external nonReentrant {
    // if there is a whitelist, an un-whitelisted user will
    // not have any sale tokens to withdraw
    // so we do not check whitelist here

    // must be past end block plus withdraw delay
    require(endBlock + withdrawDelay < block.number, 'cannot withdraw yet');
    // prevent repeat withdraw
    require(hasWithdrawn[_msgSender()] == false, 'already withdrawn');
    // must not be a zero price sale
    require(salePrice != 0, 'use withdrawGiveaway');

    // get payment received
    uint256 payment = paymentReceived[_msgSender()];

    // calculate amount of sale token owed to buyer
    uint256 saleTokenOwed = (payment * SALE_PRICE_DECIMALS) / salePrice;

    // set withdrawn to true
    hasWithdrawn[_msgSender()] = true;

    // increment withdrawer count
    withdrawerCount += 1;

    // transfer owed sale token to buyer
    saleToken.safeTransfer(_msgSender(), saleTokenOwed);

    // emit
    emit Withdraw(_msgSender(), saleTokenOwed);
}
```

Impact This inconsistency could cause security impacts if some critical operations are based on the recorded number of transferred tokens. However, because all the tokens used in the IF launchpad are manually reviewed by IF to ensure that they are not deflation tokens, the launchpad will not be affected at the current stage.

Suggestion Check the balance of the token again after the token transfer operation.

3.2.3 Potential Abuse of _trackId

ID: BWE-5	**Type: Semantic Consistency**
Risk level: Undetermined	Target: IFAllocationSale.sol
Status: Not a problem since IF is a fully decentralized launchpad that could be used by any user to sell their tokens.	

Description The constructor of the contract allows the deployer to specify the trackId.

Impact A user can deploy a sale contract and specify the trackId as the one used by other deployed contracts.

Suggestion NA.

```
    constructor(
        uint256 _salePrice,
        address _funder,
        ERC20 _paymentToken,
        ERC20 _saleToken,
        IFAllocationMaster _allocationMaster,
        uint24 _trackId,
        uint80 _allocSnapshotBlock,
        uint256 _startBlock,
        uint256 _endBlock,
        uint256 _maxTotalPayment
    ) {
        salePrice = _salePrice;
        funder = _funder;
        paymentToken = _paymentToken;
        saleToken = _saleToken;
        allocationMaster = _allocationMaster;
        trackId = _trackId;
        allocSnapshotBlock = _allocSnapshotBlock;
        startBlock = _startBlock;
        endBlock = _endBlock;
        maxTotalPayment = _maxTotalPayment;
    }
```

3.3 Inconsistency between Documentation and Code

ID: BWE-6	**Type: Semantic Consistency**
Risk level: Undetermined	Target: IFAllocationSale.sol
Status: Fixed.	

Description There are some errors in the website's document [4].

1) When calculating the current received payment, the formula provided in the document is: paymentReceived = paymentToken.balanceOf(allocationSale) This calculation is not correct.

2) The calculation of saleTokensPurchased is described incorrectly in the document:

saleTokensPurchased = salePrice / (paymentReceived * SALE_PRICE_DECIMALS))

3) The formula to calculate the remaining sale tokens is wrong (remaining

= saleTokensPurchased/saleAmount)

Impact Users may have misunderstandings by reading the document.

Suggestion

- Use totalPaymentReceived in the contract to calculate the current received payment.
- saleTokensPurchased = paymentReceived*SALE_PRICE_DECIMALS / salePrice
- remaining = saleAmount – saleTokensPurchased

3.4 Additional Recommendation

3.4.1 Follow the Checks-Effects-Interactions Pattern

ID: BWE-7	**Type: Coding Quality & Style**
Risk level: Low	Target: IFAllocationSale.sol IFAllocationMaster.sol
Status: Not a problem since either a) the token is trusted (no existence of an untrusted external call); b) the "to" address is the sale smart contract itself.	

Description The security practice suggests that any external call is not trustworthy, namely, "calls to known contracts might in turn cause calls to unknown contracts" [5]. Indeed, sometimes it is counterintuitive, as stated in [6]: "The only negative consequence of using the Checks-Effects-Interactions pattern is, that it is counterintuitive to use, when coming from a different programming paradigm."

Suggestion NA

4. Conclusion

In this audit, we have analyzed the business logic, the design and the implementation of the IF Launchpad. Indeed, we are impressed by the efforts of IF trying to mitigate flashloan related threats. Overall, the current code base is well structured and implemented, i.e., most of the identified issues have been promptly discussed, confirmed or fixed. Meanwhile, as previously disclaimed, this report does not give any warranties on discovering all security issues of the smart contracts. We appreciate any constructive feedback or suggestions.

5. Reference

[1] https://www.blocksecteam.com/

[2] Risk Rating Methodology.

https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[3] Common Weakness Enumeration. https://cwe.mitre.org/

[4] https://docs.impossible.finance/launchpad/smart-contracts/ifallocationsale.sol#computing-remaining

[5] https://docs.soliditylang.org/en/develop/security-considerations.html#use-the-checks-effects-interactions-pattern

[6] https://fravoll.github.io/solidity-patterns/checks_effects_interactions.html