Security Audit Report for Impossible Finance Launchpad contract

Date: Sep 14, 2022

Version: 4.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Impossible Finance
Target	Impossible Finance Launchpad contract

Version History

Version	Date	Description
2.0	May 23, 2022	Second version
3.0	June 15, 2022	Third version
4.0	Sep 14, 2022	fourth version

About BlockSec The BlockSec Team focuses on the security of the blockchain ecosystem, and collaborates with leading DeFi projects to secure their products. The team is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious 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, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The repository that has been audited includes launchpad-contracts (IFLaunchpad) [1].

The auditing process is iterative. Specifically, we will audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following. Our audit report is responsible for the only initial version (`Version 1`), as well as new codes (in the following versions) to fix issues in the audit report.

Project	Version	Commit SHA
IF Launchpad	Version 1	48644aa00587e46f32725c6b5094746cdc32d5c3
	Version 2	233a159d769564992677ea03dd7e1555152a4d94
	Version 3	1ae64239d94362e21d1a4a92ecff724cd5504580
	Version 4	76eec79c5b99f8e119480520e2b9a86c68d17d2a
	Version 5	1f668fad63d466ae34cd71668701762046d59d38
	Version 6	58ac788ec196b5a6ac81f9a6552cf8412b1cc57e

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale, or any other product, service, or other assets. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other assets.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any specific project. 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 nonexistence 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.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain, and the computing infrastructure are out of the scope.

[1] https://github.com/ImpossibleFinance/launchpad-contracts

1.3 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.
- **Semantic Analysis** We study the business logic 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, etc.

We show the main concrete checkpoints in the following.

1.3.1 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Access control
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 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]. The overall *severity* of the risk is determined by *likelihood* and *impact*. Specifically, likelihood estimates how likely a particular vulnerability can be uncovered and exploited by an attacker, while the impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact			
Impact	*High*	High	Medium
	Low	Medium	Low
		High	*Low*
		Likelihood	

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered issue will fall into one of the following four categories:

- **Undetermined** No response yet.
- **Acknowledged** The issue has been received by the client, but not confirmed yet.
- **Confirmed** The issue has been recognized by the client, but not fixed yet.
- **Fixed** The issue has been confirmed and fixed by the client.

[2] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[3] https://cwe.mitre.org/

Chapter 2 Findings

In total, we find **seven** potential issues. We have **three** recommendation.

- High Risk: 1
- Medium Risk: 1
- Low Risk: 5
- Recommendations: 3

ID	Severity	Description	Category	Status
1	High	Prevent Users Claim Rewards Immediately after the First Staking	DeFi Security	Fixed
2	Medium	Update Checkpoint after Emergency Withdraw	DeFi Security	Fixed
3	Low	Potential Unfair Staking	DeFi Security	Confirmed
4	Low	Potential Unfair Claiming	DeFi Security	Confirmed
5	Low	Do Not Use Elastic Supply Tokens I	DeFi Security	Confirmed
6	Low	Do Not Use Elastic Supply Tokens II	DeFi Security	Confirmed
7	Low	Refund `IDIA` Tokens after Batch Mint Vouchers	DeFi Security	Confirmed
8	-	Unchecked Function Parameters I	Recommendation	Fixed
9	-	Unchecked Function Parameters II	Recommendation	Fixed
10	-	Unchecked Sender Provided by the Forwarder	Recommendation	Confirmed
11	-	Pay Attention to Issues in the Cross-chain Process	Notes	

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Prevent Users Claim Rewards Immediately after the First Staking

Severity High

Status Fixed in `Version 4`

Introduced by `Version 3`

Description The contract `vIDIA.sol` uses the following formula to calculate the number of rewards that a user can claim:

$$userReward = \frac{userInfo.stakedAmt \times (rewardPerShare - userInfo.lastRewardPerShare)}{FACTOR} \tag{2.1}$$

```
function calculateUserReward(address user) public view returns (uint256) {
    return
        (userInfo[user].stakedAmt *
            (rewardPerShare - userInfo[user].lastRewardPerShare)) / FACTOR;
}
```

Listing 2.1: vIDIA.sol

The function `stake` in the contract `vIDIA.sol` updates the `stakedAmt`. But it does not update the `lastRewardPerShare` when someone does not have `stakedAmt` before the staking.

A large amount (i.e. $\frac{stakedAmt \times (rewardPerShare - 0)}{FACTOR}$) of rewards can be claimed immediately after the first staking.

```
function stake(uint256 amount) external notHalted {
    address sender = _msgSender();
    claimReward(sender);
    ERC20(underlying).safeTransferFrom(sender, address(this), amount);
    totalStakedAmt += amount;
    userInfo[sender].stakedAmt += amount;
    _mint(sender, amount);
    emit Stake(sender, amount);
}
```

Listing 2.2: vIDIA.sol

```
function claimReward(address sender) public {
    uint256 reward = calculateUserReward(sender);
    if (reward > 0) {
        // reset user's rewards sum
        userInfo[sender].lastRewardPerShare = rewardPerShare;
        // transfer reward to user
        ERC20 claimedTokens = ERC20(underlying);
        claimedTokens.safeTransfer(sender, reward);
        emit ClaimReward(sender, reward);
    }
}
```

Listing 2.3: vIDIA.sol

The possible attack process is as following:

1. The attacker invokes the function `stake`.
2. Then the attacker invokes the function `claimReward` directly. Since the `stakedAmt` has been updated in the previous step, the attacker can claim a large number of rewards.
3. To withdraw the startup principal, the attacker can leverage many strategies, such as:

- Invoking the function `claimStaked` directly with some fees.
- Invoking the function `unstake`, then claiming unstakes after two weeks.
- Invoking the function `emergencyWithdrawStaked`, when the contract is halted.

Impact Underlying tokens staked in the contract can be stolen by the malicious user.

Suggestion I Modify the visibility of the function `claimReward`.

Suggestion II Handle the special case: `userInfo.lastRewardPerShare = 0`.

2.1.2 Update Checkpoint after Emergency Withdraw

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `emergencyWithdraw` can be invoked when a track is disabled. Normally a user can withdraw staked tokens according to the latest checkpoint. However, the checkpoint of this user is not updated in the function `emergencyWithdraw`.

As a result, an attacker can withdraw twice by calling the function `emergencyWithdraw` and then the function `unstake` when a track is disabled.

```
function emergencyWithdraw(uint24 trackId) external nonReentrant {
    // require track is disabled
    require(trackDisabled[trackId], 'track !disabled');

    // require can only emergency withdraw once
    require(
        !hasEmergencyWithdrawn[trackId][_msgSender()],
        'already called'
    );

    // set emergency withdrawn status to true
    hasEmergencyWithdrawn[trackId][_msgSender()] = true;

    // get track info
    TrackInfo storage track = tracks[trackId];

    //// get user latest checkpoint

    // get number of user's checkpoints within this track
    uint32 userCheckpointCount = userCheckpointCounts[trackId][
        _msgSender()
    ];

    // get user's latest checkpoint
    UserCheckpoint storage checkpoint = userCheckpoints[trackId][
        _msgSender()
    ][userCheckpointCount - 1];

    // transfer the specified amount of stake token from this contract to user
    track.stakeToken.safeTransfer(_msgSender(), checkpoint.staked);

    // emit
    emit EmergencyWithdraw(trackId, _msgSender(), checkpoint.staked);
}
```

Listing 2.4: IFAllocationMaster.sol

Impact Users' staking tokens may be stolen.

Suggestion Update checkpoints after the emergency withdraw.

2.1.3 Potential Unfair Staking

Severity Low

Status Confirmed

Introduced by Version 1

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.

```
function addUserCheckpoint(
    uint24 trackId,
    uint104 amount,
    bool addElseSub
) internal {
    // get track info
    TrackInfo storage track = tracks[trackId];

    // get user checkpoint count
    uint32 nCheckpointsUser = userCheckpointCounts[trackId][_msgSender()];

    // get track checkpoint count
    uint32 nCheckpointsTrack = trackCheckpointCounts[trackId];

    // get latest track checkpoint
    TrackCheckpoint memory trackCp = trackCheckpoints[trackId][
        nCheckpointsTrack - 1
    ];

    // if this is first checkpoint
    if (nCheckpointsUser == 0) {
        // check if amount exceeds maximum
        require(amount <= track.maxTotalStake, 'exceeds staking cap');
```

Listing 2.5: IFAllocationMaster.sol

Although there is a limit on the maximum stake cap `TrackInfo.maxTotalStake` of a user, an attacker can bypass it easily by creating lots accounts and switching his/her accounts.

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

Suggestion Add timelock mechanism for unstaking.

Feedback from the Project One user can only KYC one wallet address, which is guaranteed by the manual review.

2.1.4 Potential Unfair Claiming

Severity Low

Status Confirmed

Introduced by `Version 1`

Description Some functions in the contract `IFAllocationSale` can be attacked by **Sybil Attack**.

When allocation of `saleToken` is overided, every user has a fixed limit to claim or purchase. However, an attacker can bypass it by creating many accounts.

Take the function `withdrawGiveaway` for example, every user can claim `saleTokenAllocationOverride` amounts of `saleToken`. An attacker can claim all `saleToken` in one transaction by invoking the function `withdrawGiveaway` repeatedly using different accounts.

```
function withdrawGiveaway(bytes32[] calldata merkleProof)
    external
    nonReentrant
{
    // must be past end timestamp plus withdraw delay
    require(
        endTime + withdrawDelay < block.timestamp,
        ’cannot withdraw yet’
    );
    // prevent repeat withdraw
    require(hasWithdrawn[_msgSender()] == false, ’already withdrawn’);
    // must be a zero price sale
    require(salePrice == 0, ’not a giveaway’);
    // if there is whitelist, require that user is whitelisted by checking proof
    require(
        whitelistRootHash == 0 || checkWhitelist(_msgSender(), merkleProof),
        ’proof invalid’
    );
    uint256 saleTokenOwed;
    // each participant in the zero cost "giveaway" gets a flat amount of sale token
    if (saleTokenAllocationOverride == 0) {
        // if there is no override, fetch the total payment allocation
        saleTokenOwed = getUserStakeValue(_msgSender());
    } else {
        // if override, set the override amount
        saleTokenOwed = saleTokenAllocationOverride;
    }
    // sale token owed must be greater than 0
    require(saleTokenOwed != 0, ’withdraw giveaway amount 0’);

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

    // increment withdrawer count
    withdrawerCount += 1;

    // transfer giveaway sale token to participant
    saleToken.safeTransfer(_msgSender(), saleTokenOwed);

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

Listing 2.6: IFAllocationSale.sol

There exists a same issue in the function `_purchase`.

Impact All tokens can be claimed or purchased by a single malicious user.

Feedback from the Project In every sale the whitelist root hash will be set, which is guaranteed by the manual review.

2.1.5 Do Not Use Elastic Supply Tokens I

Severity Low

Status Confirmed

Introduced by Version 1

Description Some functions do not consider the elastic supply tokens when performing the token transfer, such as:

The function stake in the contract IFAllocationMaster:

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

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

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

Listing 2.7: IFAllocationMaster.sol

The function fund in the contract IFAllocationSale:

```
function fund(uint256 amount) external onlyFunder {
    // sale must not have started
    require(block.timestamp < startTime, 'sale already started');

    // transfer specified amount from funder to this contract
    saleToken.safeTransferFrom(_msgSender(), address(this), amount);

    // increase tracked sale amount
    saleAmount += amount;

    // emit
    emit Fund(_msgSender(), amount);
}
```

Listing 2.8: IFAllocationSale.sol

The function _purchase in the contract IFAllocationSale:

```
// 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;
```

Listing 2.9: IFAllocationSale.sol

If the token is a deflation token, there will be a difference between the recorded amount of transferred token 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.

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.

Feedback from the Project The deflation token will not be supported. We can run the manual review for the support tokens.

2.1.6 Do Not Use Elastic Supply Tokens II

Severity Low

Status Confirmed

Introduced by `Version 3`

Description Some functions in the contract `vIDIA.sol` do not consider the elastic supply tokens when performing the token transfer.

Impact It could cause the inconsistency between the recorded number and the actual tokens transferred.

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

Feedback from the Project The deflation token will not be supported. We can run the manual review for the support tokens.

2.1.7 Refund `IDIA` Tokens after Batch Mint Vouchers

Severity Low

Status Confirmed

Introduced by `Version 5`

Description The function `batchMint` in the contract `BatchMintVoucher` takes `totalValue` IDIA tokens from the caller and then creates vouchers according to the parameters passed.

However, there exists no restrictions for the amount of `IDIA` that the user transferred (i.e. `totalValue`). If there exist some `IDIA` tokens that are already owned by the contract `BatchMintVoucher`, the attacker can use these `IDIA` tokens to create free voucher and claim them.

```
function batchMint(
    // total value of the underlying token needed to mint the vouchers
    uint256 totalValue,
    VoucherParams calldata params
) external {
    // send the required tokens to the contract
    idia.transferFrom(msg.sender, address(this), totalValue);
    for (uint256 i = 0; i < params.users.length; i++) {
```

```
        // mint a voucher
        (, uint256 tokenId) = iicToken.mint(
            params.terms[i],
            params.values[i],
            params.maturities[i],
            params.percentages[i],
            params.originalInvestors[i]
        );
        address userAddr = params.users[i];
        // transfer the voucher to a user
        iicToken.transferFrom(address(this), userAddr, tokenId);
    }
}
```

Listing 2.10: BatchMintVoucher.sol

Suggestion Add some restrictions for the `totalValue` and refund extra `IDIA` tokens after the voucher creation.

Feedback from the Project The vouchers one can be ignored now. It is WIP.

2.2 Additional Recommendation

2.2.1 Unchecked Function Parameters I

Status Fixed in `Version 2`

Introduced by `Version 1`

Description Some parameters in the constructor of the contract `IFAllocationSale` are not checked to avoid a potential misuse, such as:

1) The `saleToken` should not be same as the `paymentToken`.
2) When the `salePrice` is not zero, the `paymentToken` and the `maxTotalPayment` should not be zero.
3) The variables `allocationMaster`, `trackId`, `allocSnapshotBlock` should not be zero, if the allocation will not be overrided.
4) The `allocSnapshotBlock` should be greater than the block number in the earliest checkpoint of the track, if the allocation will not be overrided.

```
constructor(
    uint256 _salePrice,
    address _funder,
    ERC20 _paymentToken,
    ERC20 _saleToken,
    IFAllocationMaster _allocationMaster,
    uint24 _trackId,
    uint80 _allocSnapshotBlock,
    uint256 _startTime,
    uint256 _endTime,
    uint256 _maxTotalPayment
) {
    // funder cannot be 0
    require(_funder != address(0), '0x0 funder');
```

```
        // sale token cannot be 0
        require(address(_saleToken) != address(0), '0x0 saleToken');
        // start timestamp must be in future
        require(block.timestamp < _startTime, 'start timestamp too early');
        // end timestamp must be after start timestamp
        require(_startTime < _endTime, 'end timestamp before start');

        salePrice = _salePrice; // can be 0 (for giveaway)
        funder = _funder;
        paymentToken = _paymentToken; // can be 0 (for giveaway)
        saleToken = _saleToken;
        allocationMaster = _allocationMaster; // can be 0 (with allocation override)
        trackId = _trackId; // can be 0 (with allocation override)
        allocSnapshotBlock = _allocSnapshotBlock; // can be 0 (with allocation override)
        startTime = _startTime;
        endTime = _endTime;
        maxTotalPayment = _maxTotalPayment; // can be 0 (for giveaway)
    }
```

Listing 2.11: IFAllocationSale.sol

Impact The wrong parameters cannot be fixed after being initialized.

Suggestion Verify these parameters in the constructor.

2.2.2 Unchecked Function Parameters II

Status Fixed in `Version 4`

Introduced by `Version 3`

Description The parameters `_admin` and `_underlying` should not be zero in the constructor of the contract `vIDIA.sol`. But there are no checks for them.

```
    constructor(
        string memory _name,
        string memory _symbol,
        address _admin,
        address _underlying
    ) AccessControlEnumerable() IFTokenStandard(_name, _symbol, _admin) {
        _setupRole(FEE_SETTER_ROLE, _admin);
        _setupRole(DELAY_SETTER_ROLE, _admin);
        _setupRole(WHITELIST_SETTER_ROLE, _admin);
        underlying = _underlying;

        // Add 0x0 to whitelist so _beforeTokenTransfer doesn't reject mint/burn txs
        // transfers to/from 0x0 fail by default
        EnumerableSet.add(whitelistAddresses, address(0x0));
    }
```

Listing 2.12: vIDIA.sol

Impact The wrong parameters cannot be fixed after being initialized.

Suggestion Verify these parameters in the constructor.

2.2.3 Unchecked Sender Provided by the Forwarder

Status Confirmed

Introduced by `Version 3`

Description In order to prevent the mistake of the forwarder, it is better to double check the real `sender` provided by the forwarder.

```
function _msgSender() internal view virtual override returns (address sender) {
    if (isTrustedForwarder(msg.sender)) {
        // The assembly code is more direct than the Solidity version using ‘abi.decode‘.
        // solhint-disable-next-line no-inline-assembly
        assembly {
            sender := shr(96, calldataload(sub(calldatasize(), 20)))
        }
    } else {
        return super._msgSender();
    }
}
```

Listing 2.13: ERC2771ContextUpdateable.sol

Impact The forwarder can provide a fake `sender` (such as: zero address, owner) to perform some privileged operations.

Suggestion Verify the `sender` provided by the forwarder.

Feedback from the Project The forwarder should be reliable.

2.3 Notes

2.3.1 Pay Attention to Issues in the Cross-chain Process

Introduced by `Version 5`

Description The new version of Impossible Finance Launchpad contract introduces some cross-chain features, which are supported by the contract `messageBus` of the `cBridge`.

There are some possible issues that the project should pay attention to.

1. The information from the target chain can not be used directly, because the cross-chain transactions have delays between the source chain and the target chain. Pay attention to the double spending problem.
2. The Information of the disabled track is not synchronized to the target chain. There is no way to check if a track is available in the target chain.