Security Audit Report for Impossible Finance Farm Contract

Date: Sep 14, 2022

Version: 1.0

Contact: contact@blocksec.com

Contents

Report Manifest

Item	Description
Client	Impossible Finance
Target	Impossible Finance Farm Contract

Version History

Version	Date	Description
1.0	Sep 14, 2022	First Release

About BlockSec The BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec 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 successfully protected digital assets that are worth more than 5 million dollars by blocking multiple attacks. 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 impossible-farm contracts [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		Commit SHA
impossible-farm	`Version 1`	`e3db9399939fa4aa9d6477db086f04ad95854216`
	`Version 2`	`9092b65e1418d8d4a31a669e95bcb16cefaed99a`

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 asset. 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 asset.

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

1.3 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.

[1] https://github.com/ImpossibleFinance/impossible-farm

- **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, and 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
- Permission management
- 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 is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while 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	*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 item will fall into one of the following four categories:

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

1.5 Fuzzing testing

Besides the static analysis and manual code review, we also use an in-house fuzzing tool during this audit. We generate test cases and feed them to the contract. Our tool leverages the five oracles to locate the vulnerabilities in the contract.

Specifically, the fuzzing process of Impossible Finance Farm Contract is as follows:

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

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

- First, we compiled contracts of Impossible Finance Farm Contract (including `SmartChefFactory` and `SmartChefInitializable`) and generated go bindings of them using the tool `abigen` [4].
- Then, we deployed a runtime environment of the entire project (using the tool we developed).
- Based on the business logic of `SmartChefInitializable`, we wrote five oracles, which cover main functionality of Impossible Finance Farm Contract.
- Finally, we run the go native fuzzer.

Oracle	Test cases	Result
The amount of the staked token that can be `withdrawn` should be equal to the amount that has been `deposited` in the same block.	10,400,057	ALL PASSED
Rewards that can be claimed should be equal to the result of the function `pendingReward`.	6,377,673	ALL PASSED
Same deposit strategies should have same debt (i.e. `UserInfo`).	113,830	ALL PASSED
Same deposit strategies should `withdraw` same staked tokens and reward tokens.	113,830	ALL PASSED
After the reward, the result of the function `pendingReward` for new depositors should always return 0.	162,008	ALL PASSED
After stopping the reward, a new depositor can only `withdraw` tokens that he has deposited.	162,008	ALL PASSED

[4] https://geth.ethereum.org/docs/dapp/native-bindings

Chapter 2 Findings

In total, we find **one** potential issues.

- High Risk: 0
- Medium Risk: 0
- Low Risk: 1
- Recommendations: 0
- Note: 2

ID	Severity	Description	Category	Status
1	Low	Can not support deflation token	DeFi Security	Fixed
2	-	Inconsistent naming between parameter variables	Notes	Fixed
3	-	Inconsistent between comments and code	Notes	Fixed

The details are provided in the following sections.

2.1 DeFi Security

2.1.1 Can not support deflation token

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The UserInfo in SmartChefInitializable records the token amount that user has claimed, not the real transferred token amount. Some deflation tokens, which adjust the amount during token transfer, may cause the inconsistencies of the recorded debt.

```
function deposit(uint256 _amount) external nonReentrant {
  UserInfo storage user = userInfo[msg.sender];

  if (hasUserLimit) {
      require(_amount + user.amount <= poolLimitPerUser, "User amount above limit");
  }

  _updatePool();

  if (user.amount > 0) {
      uint256 pending = (user.amount * accTokenPerShare / PRECISION_FACTOR) - user.rewardDebt;
      if (pending > 0) {
          rewardToken.safeTransfer(msg.sender, pending);
      }
  }

  if (_amount > 0) {
      user.amount = user.amount + _amount;
      stakedToken.safeTransferFrom(msg.sender, address(this), _amount);
  }
```

```

    user.rewardDebt = user.amount * accTokenPerShare / PRECISION_FACTOR;

    emit Deposit(msg.sender, _amount);
}
```

Listing 2.1: SmartChefInitializable.sol

2.2 Notes

2.2.1 Inconsistent name between parameter variables

Status Fixed in Version 2

Introduced by Version 1

Description Some parameter variables in the contract SmartChefFactory are inconsistent with the contract SmartChefInitializable.

The parameters used in the function deployPool are named as xxxBlock. However, their counterparts are named as xxxTime in the function initialize.

```
    function deployPool(
        IERC20 _stakedToken,
        IERC20Extended _rewardToken,
        uint256 _rewardPerBlock,
        uint256 _startBlock,
        uint256 _bonusEndBlock,
        uint256 _poolLimitPerUser,
        address _admin
    ) external onlyOwner {
        require(_stakedToken.totalSupply() >= 0);
        require(_rewardToken.totalSupply() >= 0);
        require(_stakedToken != _rewardToken, "Tokens must be be different");

        bytes memory bytecode = type(SmartChefInitializable).creationCode;
        bytes32 salt = keccak256(abi.encodePacked(_stakedToken, _rewardToken, _startBlock));
        address smartChefAddress;

        assembly {
            smartChefAddress := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }

        SmartChefInitializable(smartChefAddress).initialize(
            _stakedToken,
            _rewardToken,
            _rewardPerBlock,
            _startBlock,
            _bonusEndBlock,
            _poolLimitPerUser,
            _admin
        );

        emit NewSmartChefContract(smartChefAddress);
    }
```

Listing 2.2: SmartChefFactory.sol

```
function initialize(
    IERC20 _stakedToken,
    IERC20Extended _rewardToken,
    uint256 _rewardPerSecond,
    uint256 _startTime,
    uint256 _bonusEndTime,
    uint256 _poolLimitPerUser,
    address _admin
) external {
    require(!isInitialized, "Already initialized");
    require(msg.sender == SMART_CHEF_FACTORY, "Not factory");

    // Make this contract initialized
    isInitialized = true;

    stakedToken = _stakedToken;
    rewardToken = _rewardToken;
    rewardPerSecond = _rewardPerSecond;
    startTime = _startTime;
    bonusEndTime = _bonusEndTime;

    if (_poolLimitPerUser > 0) {
        hasUserLimit = true;
        poolLimitPerUser = _poolLimitPerUser;
    }

    uint256 decimalsRewardToken = uint256(rewardToken.decimals());
    require(decimalsRewardToken < 30, "Must be inferior to 30");

    PRECISION_FACTOR = 10**(30 - decimalsRewardToken);

    // Set the lastRewardTime as the startTime
    lastRewardTime = startTime;

    // Transfer ownership to the admin address who becomes owner of the contract
    transferOwnership(_admin);
}
```

Listing 2.3: SmartChefInitializable.sol

2.2.2 Inconsistent between comments and code

Status Fixed in Version 2

Introduced by Version 1

Description There are some inconsistencies between comments and code.

1. The documentation comment of the parameter `_rewardPerSecond` should be "reward per second" for the function `initialize`

```
/*
* @notice Initialize the contract
* @param _stakedToken: staked token address
* @param _rewardToken: reward token address
* @param _rewardPerSecond: reward per minute (in rewardToken)
* @param _startTime: start time
* @param _bonusEndTime: end time
* @param _poolLimitPerUser: pool limit per user in stakedToken (if any, else 0)
* @param _admin: admin address with ownership
*/
```

Listing 2.4: SmartChefInitializable.sol

2. Documentation comments for parameters `_from` and `_to` should be "time to start" and "time to finish" for the function `_getMultiplier`.

```
/*
* @notice Return reward multiplier over the given _from to _to block.
* @param _from: block to start
* @param _to: block to finish
*/
```

Listing 2.5: SmartChefInitializable.sol

3. Some other places that should modify "block" to "time" are as follows:

```
/*
* @notice Update reward per block
* @dev Only callable by owner.
* @param _rewardPerSecond: the reward per block
*/
```

Listing 2.6: SmartChefInitializable.sol

```
/**
 * @notice It allows the admin to update start and end blocks
 * @dev This function is only callable by owner.
 * @param _startTime: the new start block
 * @param _bonusEndTime: the new end block
*/
```

Listing 2.7: SmartChefInitializable.sol