Security Audit Report for Impossible Finance Swap-core Contracts

Date: Dec 10, 2021

Version: 1.2

Contact: contact@blocksecteam.com

Contents

Report Manifest

Item	Description
Client	Impossible Finance
Target	Impossible Finance Swap-core Contracts

Version History

Version	Date	Description
1.0	Dec 06, 2021	First Release
1.1	Dec 07, 2021	Second Release
1.2	Dec 10, 2021	Update the new commit hash

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

The target contract is Impossible Finance Swap-core Contracts. The detailed description is in the following link: Impossible Finance.

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

The files that are audited in this report include the following ones.

Repo Name	Github URL
Impossible Swap Core	https://github.com/ImpossibleFinance/impossible-swap-core

The commit hash before the audit is 29aaef89f996acdbee92b67c4d95fb608dc8b876. The commit hash that fixes the issues found in this audit is 665c2d9a18b4d0475a527c25f41779b6a9cce89c.

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 do 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.

- **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
- 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 [1] and Common Weakness Enumeration [2]. Accordingly, the severity measured in this report are classified into four categories: **High**, **Medium**, **Low** and **Undetermined**.

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

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

Chapter 2 Findings

In total, we find eight potential issues in the smart contract, and we also have one recommendation, as follows:

- High Risk: 1
- Medium Risk: 3
- Low Risk: 4
- Recommendation: 1

ID	Severity	Description	Category
1	High	*Reentrancy Vulnerability in ImpossibleWrappedToken*	Software Security
2	Low	*Incorrect Emitted Event*	Software Security
3	Medium	*Deposit Front-running Vulnerability*	DeFi Security
4	Low	*Requirement Inconsistency in ImpossibleLibrary and ImpossiblePair*	DeFi Security
5	Low	*Hardstop Mechanism Inconsistency*	DeFi Security
6	Medium	*Does Not Consider the Impact of Deflation/Rebasing Tokens*	DeFi Security
7	Low	*Avoid to Empty the Pair*	DeFi Security
8	Medium	*Improper Design of* `kLast` *in Fees Collection*	DeFi Security
9	-	*Documentation Inconsistency*	Recommendation

The details are provided in the following sections.

2.1 Software Security

2.1.1 Reentrancy Vulnerability in ImpossibleWrappedToken

Status Confirmed and fixed.

Description There exists a potential reentrancy vulnerability in ImpossibleWrappedToken, including the `deposit(address)` function and the `deposit(address, uint256)` function. Specifically, an attacker could reenter `deposit(address)` after invoking `deposit(address, uint256)` if the underlying token supports callback mechanism. The exploitation is as follows:

1. The attacker invokes `deposit(address, uint256)`, which further invokes `underlying.transferFrom()`.
2. `underlying.transferFrom()` first modifies the balances of the sender and the receiver, and then invokes the callback specified by the attacker. Notice that at this time, those underlying tokens have been successfully transferred to ImpossibleWrappedToken but the corresponding `underlyingBalance` has not been updated.
3. The attacker's callback reenters `deposit(address)`, because `underlyingBalance` is out of date, the attacker could receive wrapped tokens.
4. When the execution returns back to `deposit(address, uint256)`, the attacker could still receive wrapped tokens.

```
53    function deposit(address dst, uint256 wad) public returns (uint256 transferAmt) {
54        transferAmt = wad.mul(ratioDenom).div(ratioNum);
55        bool success = IERC20(underlying).transferFrom(msg.sender, address(this), transferAmt);
```

```
56        require(success, 'ImpossibleWrapper: TRANSFERFROM_FAILED');
57        _deposit(dst, wad);
58        underlyingBalance = underlyingBalance.add(transferAmt);
59        emit Transfer(address(0), msg.sender, wad);
60    }
61
62    // wad = amount of wrapped tokens
63    function deposit(address dst) public override returns (uint256 transferAmt) {
64        uint256 balance = IERC20(underlying).balanceOf(address(this));
65        transferAmt = balance.sub(underlyingBalance);
66        uint256 wad = transferAmt.mul(ratioNum).div(ratioDenom);
67        _deposit(dst, wad);
68        underlyingBalance = balance;
69        emit Transfer(address(0), dst, wad);
70    }
```

Listing 2.1: ImpossibleWrappedToken.sol

Impact The attacker may receive wrapped tokens twice while only deposit once.

Suggestion Add reentrancy guards as function modifiers.

2.1.2 Incorrect Emitted Event

Status Confirmed and fixed.

Description The line 59 in `ImpossibleWrappedToken.deposit(address, uint256)` emits an incorrect event. The `Transfer` event should log the received address (`dst`) instead of the caller `msg.sender`.

```
53    function deposit(address dst, uint256 wad) public returns (uint256 transferAmt) {
54        transferAmt = wad.mul(ratioDenom).div(ratioNum);
55        bool success = IERC20(underlying).transferFrom(msg.sender, address(this), transferAmt);
56        require(success, 'ImpossibleWrapper: TRANSFERFROM_FAILED');
57        _deposit(dst, wad);
58        underlyingBalance = underlyingBalance.add(transferAmt);
59        emit Transfer(address(0), msg.sender, wad);
60    }
```

Listing 2.2: deposit:ImpossibleWrappedToken.sol

Impact The emitted event can be misleading and break other implementations that rely on these logs.

Suggestion Replace `msg.sender` with `dst` in line 59.

2.2 DeFi Security

2.2.1 Deposit Front-running Vulnerability

Status Confirmed and fixed.

Description The design of `ImpossibleWrappedToken.deposit(address)` takes deflation/inflation or re-basing tokens into account. However, the transfer and deposit operations are separated. As a result, the attacker could launch the front-running attack by inserting a deposit transaction between user's transfer and deposit transactions.

```
63    function deposit(address dst) public override returns (uint256 transferAmt) {
64        uint256 balance = IERC20(underlying).balanceOf(address(this));
65        transferAmt = balance.sub(underlyingBalance);
66        uint256 wad = transferAmt.mul(ratioNum).div(ratioDenom);
67        _deposit(dst, wad);
68        underlyingBalance = balance;
69        emit Transfer(address(0), dst, wad);
70    }
```

Listing 2.3: deposit:ImpossibleWrappedToken.sol

Impact User's deposited tokens could be stolen by the attacker.

Suggestion A better practice is to ensure that both token transfer operation and deposit operation are executed in one transaction.

2.2.2 Requirement Inconsistency in ImpossibleLibrary and ImpossiblePair

Status Confirmed and fixed.

Description In the execution of token swap action, the Router needs to know the token (or ether) output amount which is calculated through `ImpossibleLibrary.getAmountsOut()`, and the return value of `getAmountsOut()` could be equal to pair's `reserveOut`. However, when the two values are equal, the execution of swap will be reverted by `ImpossiblePair.swap()` because of the requirement specified in line 514 in ImpossiblePair.

```
119   function getAmountOut(
120       uint256 amountIn,
121       address tokenIn,
122       address tokenOut,
123       address factory
124   ) internal view returns (uint256 amountOut) {
125       require(amountIn > 0, 'ImpossibleLibrary: INSUFFICIENT_INPUT_AMOUNT');
126       uint256 reserveIn;
127       uint256 reserveOut;
128       uint256 amountInPostFee;
129       address pair;
130       bool isMatch;
131       {
132           // Avoid stack too deep
133           (address token0, ) = sortTokens(tokenIn, tokenOut);
134           isMatch = tokenIn == token0;
135           (reserveIn, reserveOut, pair) = getReserves(factory, tokenIn, tokenOut);
136           require(reserveIn > 0 && reserveOut > 0, 'ImpossibleLibrary: INSUFFICIENT_LIQUIDITY');
137       }
138       uint256 artiLiqTerm;
139       bool isXybk;
140       {
141           // Avoid stack too deep
142           uint256 fee;
143           (fee, isXybk) = IImpossiblePair(pair).getFeeAndXybk();
144           amountInPostFee = amountIn.mul(10000 - fee);
145       }
```

```
146
147         /// If xybk invariant, set reserveIn/reserveOut to artificial liquidity instead of actual
                liquidity
148         if (isXybk) {
149             (uint256 boost0, uint256 boost1) = IImpossiblePair(pair).calcBoost();
150             uint256 sqrtK = xybkComputeSqrtK(isMatch, reserveIn, reserveOut, boost0, boost1);
151             /// since balance0=balance1 only at sqrtK, if final balanceIn >= sqrtK means balanceIn
                    >= balanceOut
152             /// Use post-fee balances to maintain consistency with pair contract K invariant check
153             if (amountInPostFee.add(reserveIn.mul(10000)) >= sqrtK.mul(10000)) {
154                 /// If tokenIn = token0, balanceIn > sqrtK => balance0>sqrtK, use boost0
155                 artiLiqTerm = calcArtiLiquidityTerm(isMatch ? boost0 : boost1, sqrtK);
156                 /// If balance started from <sqrtK and ended at >sqrtK and boosts are different,
                        there'll be different amountIn/Out
157                 /// Don't need to check in other case for reserveIn < reserveIn.add(x) <= sqrtK
                        since that case doesnt cross midpt
158                 if (reserveIn < sqrtK && boost0 != boost1) {
159                     /// Break into 2 trades => start point -> midpoint (sqrtK, sqrtK), then midpoint
                            -> final point
160                     amountOut = reserveOut.sub(sqrtK);
161                     amountInPostFee = amountInPostFee.sub((sqrtK.sub(reserveIn)).mul(10000));
162                     reserveIn = sqrtK;
163                     reserveOut = sqrtK;
164                 }
165             } else {
166                 /// If tokenIn = token0, balanceIn < sqrtK => balance0<sqrtK, use boost1
167                 artiLiqTerm = calcArtiLiquidityTerm(isMatch ? boost1 : boost0, sqrtK);
168             }
169         }
170         uint256 numerator = amountInPostFee.mul(reserveOut.add(artiLiqTerm));
171         uint256 denominator = (reserveIn.add(artiLiqTerm)).mul(10000).add(amountInPostFee);
172         uint256 lastSwapAmountOut = numerator / denominator;
173         amountOut = (lastSwapAmountOut > reserveOut) ? reserveOut.add(amountOut) :
                lastSwapAmountOut.add(amountOut);
174     }
```

Listing 2.4: getAmountOut:ImpossibleLibrary.sol

```
506     function swap(
507         uint256 amount0Out,
508         uint256 amount1Out,
509         address to,
510         bytes calldata data
511     ) external override onlyIFRouter nonReentrant {
512         require(amount0Out > 0 || amount1Out > 0, 'IF: INSUFFICIENT_OUTPUT_AMOUNT');
513         (uint256 _reserve0, uint256 _reserve1) = getReserves(); // gas savings
514         require(amount0Out < _reserve0 && amount1Out < _reserve1, 'IF: INSUFFICIENT_LIQUIDITY');
```

Listing 2.5: swap:ImpossiblePair.sol

Impact The swap action which passes router will be reverted by ImpossiblePair.

Suggestion Replace $<$ with $<=$ in line 514.

2.2.3 Hardstop Mechanism Inconsistency

Status Confirmed and fixed.

Description The hardstop mechanism (`ratioStart` & `ratioEnd`) is not considered in `ImpossibleLibrary.getAmountOut()` by the Router. As a result, the amount returned by `getAmountOut()` might be rejected by `ImpossiblePair.swap()`.

```
537    if (_isXybk) {
538        bool side = balance0 >= balance1;
539        uint256 ratio = side ? ratioStart : ratioEnd;
540        if (side && ratio > 0) {
541            require(balance1.mul(ratio) < balance0.mul(100 - ratio), 'IF: EXCEED_UPPER_STOP');
542        } else if (!side && ratio < 100) {
543            require(balance0.mul(ratio) > balance1.mul(100 - ratio), 'IF: EXCEED_LOWER_STOP');
544        }
545    }
```

Listing 2.6: swap:ImpossiblePair.sol

Impact The swap action which passes the Router's check may be reverted by `ImpossiblePair.swap()`.

Suggestion Apply a global hardstop mechanism.

2.2.4 Does Not Consider the Impact of Deflation/Rebasing Tokens

Status Confirmed and fixed.

Description The design and implementation of ImpossibleWrappedToken do not take deflation/rebasing tokens into consideration, which may result in the difference between the recorded amount (i.e., `transferAmt` in `ImpossibleWrappedToken.deposit(address, uint256)`) and the actual amount (i.e., `IERC20(underlying).balanceOf(address(this))`) of the underlying token, i.e., recorded amount > actual amount.

```
53    function deposit(address dst, uint256 wad) public returns (uint256 transferAmt) {
54        transferAmt = wad.mul(ratioDenom).div(ratioNum);
55        bool success = IERC20(underlying).transferFrom(msg.sender, address(this), transferAmt);
56        require(success, 'ImpossibleWrapper: TRANSFERFROM_FAILED');
57        _deposit(dst, wad);
58        underlyingBalance = underlyingBalance.add(transferAmt);
59        emit Transfer(address(0), msg.sender, wad);
60    }
```

Listing 2.7: deposit:ImpossibleWrappedToken.sol

Impact It may lead to the following results:

1. When user invokes `deposit(address, uint256)`, the contract will mint more wrapped tokens than it should be.
2. The `ImpossibleWrapperFactory.deletePairing()` will always fail because `totalSupply` can never be zero due to the lack of underlying tokens.

Suggestion Take deflation/rebasing tokens into account.

2.2.5 Avoid to Empty the Pair

Status Confirmed and fixed.

Description In the xybk scenario, a swap that empties a pool should not be allowed, as the original uniswap is not designed for the case that one of the reserves could be zero. Specifically, if any reserve is zero, the `addLiquidity()` action to this pair will be reverted.

Impact N/A

Suggestion Prevent any swap that will empty the pair.

2.2.6 Improper Design of `kLast` in Fees Collection

Status Confirmed and fixed.

Description For ImpossiblePair, the k calculated by `_xybkComputeK()` in xybk state will always be larger than that calculated by reserve0 * reserve1 in uni state, and `_xybkComputeK()` is a monotonically increasing function regarding the boost (b for short). In a very rare case:

1. Current pair state is xybk with a large b, and the `kLast` in the pair is updated upon a call to `mint()`/`burn()`.
2. The pair state is set to uni by `updateBoost(1, 1)` followed by `makeUni()`. In this period, `mint()`/`burn()` is never called so that the `kLast` is not updated.
3. The pair state is now uni. However, all calls to `_mintFee()` will fail because the `kLast` in xybk state will always be larger than the k in uni state (because `rootKLast` > `rootK`, and the condition in line 405 will fail).

```
396   function _mintFee(uint256 _reserve0, uint256 _reserve1) private returns (bool feeOn) {
397       address feeTo = IImpossibleSwapFactory(factory).feeTo();
398       feeOn = feeTo != address(0);
399       uint256 _kLast = kLast; // gas savings
400       if (feeOn) {
401           if (_kLast != 0) {
402               uint256 rootK =
403                   isXybk ? Math.sqrt(_xybkComputeK(_reserve0, _reserve1)) : Math.sqrt(_reserve0.
                        mul(_reserve1));
404               uint256 rootKLast = Math.sqrt(_kLast);
405               if (rootK > rootKLast) {
406                   uint256 numerator = totalSupply.mul(rootK.sub(rootKLast)).mul(4);
407                   uint256 denominator = rootK.add(rootKLast.mul(4));
408                   uint256 liquidity = numerator / denominator;
409                   if (liquidity > 0) _mint(feeTo, liquidity);
410               }
411           }
412       } else if (_kLast != 0) {
413           kLast = 0;
414       }
415   }
```

Listing 2.8: _mintFee:ImpossiblePair.sol

Impact Fees to the `feeTo()` function will fail to be collected.

Suggestion Update the `kLast` in `makeUni()`.

2.3 Additional Recommendation

2.3.1 Documentation Inconsistency

Status Confirmed.

Description The calculation formulas of `k` (which is used in swap) in code implementation and documentation are different:

- In code implementation:

$$k = \left[\sqrt{\left(\frac{(b-1)(x+y)}{2(2b-1)} \right)^2 + \frac{xy}{2b-1}} + \frac{(b-1)(x+y)}{2(2b-1)} \right]^2$$

- In documentation:

$$k = \left[\sqrt{\frac{(b-1)(x+y)^2}{2(2b-1)} + \frac{xy}{2b-1}} + \frac{(b-1)(x+y)}{2(2b-1)} \right]^2$$

Further deduction indicates that the former (as in code implementation) is correct.

Impact N/A

Suggestion Revise the documentation.

Chapter 3 Conclusion

In this audit, we have analyzed the business logic, the design, and the implementation of the Impossible Finance Swap-core Contracts. Overall, the current code base is well structured and implemented, 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.