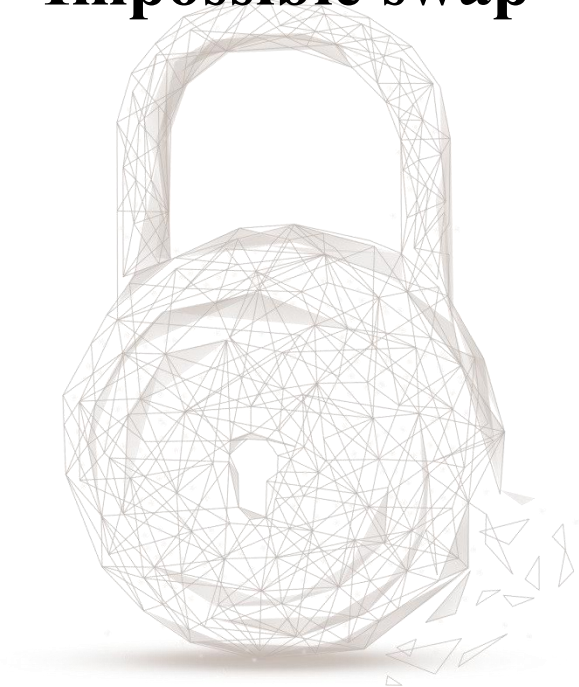
BEOSIN

Blockchain Security

Smart Contract Audit Report

for

Impossible swap



Audit Number: 202112031726

Contract Name: Impossible swap

Deployment Platform: Binance Smart Chain

Audit Project Link:

https://github.com/ImpossibleFinance/impossible-swap-core

Commit Hash:

e0203e5154096ebda242d0d4155c671324816088 (Initial)

665c2d9a18b4d0475a527c25f41779b6a9cce89c (Final)

Audit Start Date: 2021.06.18

Audit Completion Date: 2021.12.03

Audit Report Update date: 2021.12.09

Update Content of the Audit Report: Updated commit hash

Audit Result: Pass

Audit Team: Beosin Technology Co. Ltd.

Audit Results Overview

Beosin Technology has used several methods including Formal Verification, Static Analysis, Typical Case Testing and Manual Review to audit three major aspects of Impossible swap smart contract, including Coding Conventions, General Vulnerability and Business Security. **After auditing, the Impossible swap project was found to have 1 Critical-risk, 1 High-risk, 6 Low-risk, 2 info items. As of the completion of the audit, all risk items have been fixed or properly handled. The overall result of the Impossible swap smart contract is Pass.** The following is the detailed audit information for this project.

Index	Risk items	Risk level	Fix results status
ImpossiblePair-1	*cheapSwap* function design flaws	Critical	Fixed
ImpossibleRouterExtension-1	Design flaws in the *removeLiquidity* function	High	Fixed
ImpossiblePair-2	*Burn* function design flaws	Low	Fixed
ImpossibleWrapperFactory-1	*createPair* function design flaws	Low	Fixed
ImpossibleWrappedToken-1	*deposit* function event triggering error	Low	Fixed
ImpossibleWrappedToken-2	*withdraw* function logic implementation error	Low	Fixed
ImpossibleWrappedToken-3	The *transfer* and *transferFrom* functions do not determine the zero address	Info	Fixed
ImpossibleWrappedToken-4	Redundant code for the *_withdraw* function	Info	Fixed
ImpossibleWrappedToken-5	Design flaws in the *deposit* function	Info	Fixed
ImpossibleFactory-1	Design flaws in the *createPair* function	Low	Fixed

Table 1. Key Audit Findings

Risk Descriptions and Fix Results

[ImpossiblePair-1 Critical] *cheapSwap* function design flaws

- Description: The *cheapSwap* function does not validate the k value and will result in the liquidity pool sub-funds being impaired.

```
334  function cheapSwap(
335      uint256 amount0Out,
336      uint256 amount1Out,
337      address to,
338      bytes calldata data
339  ) external override onlyIFRouter nonReentrant {
340      if (amount0Out > 0) _safeTransfer(token0, to, amount0Out); // optimistically transfer tokens
341      if (amount1Out > 0) _safeTransfer(token1, to, amount1Out); // optimistically transfer tokens
342      if (data.length > 0) IImpossibleCallee(to).ImpossibleCall(msg.sender, amount0Out, amount1Out, data);
343      uint256 balance0 = IERC20(token0).balanceOf(address(this));
344      uint256 balance1 = IERC20(token1).balanceOf(address(this));
345      if (isXybk) {
346          bool side = balance0 >= balance1;
347          uint256 ratio = side ? ratioStart : ratioEnd;
348          if (side && ratio > 0) {
349              require(balance1.mul(ratio) < balance0.mul(100 - ratio), 'IF: EXCEED_UPPER_STOP');
350          } else if (!side && ratio < 100) {
351              require(balance0.mul(ratio) > balance1.mul(100 - ratio), 'IF: EXCEED_LOWER_STOP');
352          }
353      }
354      (uint256 _reserve0, uint256 _reserve1) = getReserves(); // gas savings
355      uint256 amount0In = balance0 > _reserve0 - amount0Out ? balance0 - (_reserve0 - amount0Out) : 0;
356      uint256 amount1In = balance1 > _reserve1 - amount1Out ? balance1 - (_reserve1 - amount1Out) : 0;
357      _update(balance0, balance1);
358      emit Swap(msg.sender, amount0In, amount1In, amount0Out, amount1Out, to);
359  }
```

Figure 1 source code of *cheapSwap* function(unfixed)

- Fix recommendations: Perform k-value check.
- Fix results: Fixed, the project side turns off this feature.

[ImpossibleRouterExtension-1 High] Design flaws in the *removeLiquidity* function

- Description: When the user calls the Router contract to remove the liquidity, the Router will first call the ImpossibleRouterExtension contract to transfer LP tokens to the pair contract, where the msg.sender is the Router contract and not the user, and the Router contract does not have LP token, so the transfer will fail.

```
function removeLiquidity(
    address tokenA,
    address tokenB,
    uint256 liquidity,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
) public virtual override ensure(deadline) nonReentrant returns (uint256 amountA, uint256 amountB) {
    (amountA, amountB) = IImpossibleRouterExtension(routerExtension).removeLiquidity(
        tokenA,
        tokenB,
        liquidity,
        amountAMin,
        amountBMin
    );
    unwrapSafeTransfer(tokenA, to, amountA);
    unwrapSafeTransfer(tokenB, to, amountB);
}
```

Figure 2 source code of *removeLiquidity* function(unfixed)(Router.sol)

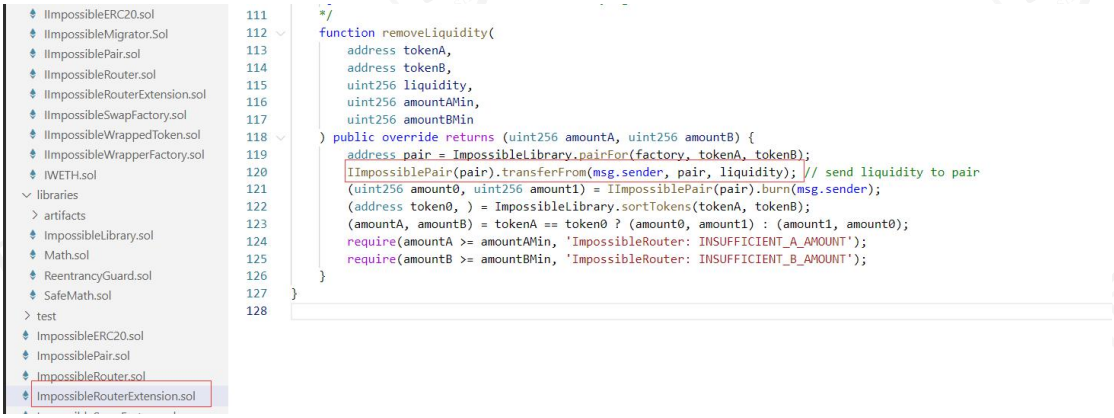


Figure 3 source code of *removeLiquidity* function (unfixed)(Extension,sol)

- Fix recommendations: Modify related logic.
- Fix results: Fixed.

```
function removeLiquidity(
    address tokenA,
    address tokenB,
    uint256 liquidity,
    uint256 amountAMin,
    uint256 amountBMin,
    address to,
    uint256 deadline
) public virtual override ensure(deadline) nonReentrant returns (uint256 amountA, uint256 amountB) {
    address pair = ImpossibleLibrary.pairFor(factory, tokenA, tokenB);
    IImpossiblePair(pair).transferFrom(msg.sender, pair, liquidity); // send liquidity to pair
    (amountA, amountB) = IImpossibleRouterExtension(routerExtension).removeLiquidity(
        tokenA,
        tokenB,
        pair,
        amountAMin,
        amountBMin
    );
    unwrapSafeTransfer(tokenA, to, amountA);
    unwrapSafeTransfer(tokenB, to, amountB);
}
```

Figure 4 source code of *removeLiquidity* function (fixed)(Router.sol)

```
function removeLiquidity(
    address tokenA,
    address tokenB,
    address pair,
    uint256 amountAMin,
    uint256 amountBMin
) public override returns (uint256 amountA, uint256 amountB) {
    (uint256 amount0, uint256 amount1) = IImpossiblePair(pair).burn(msg.sender);
    (address token0, ) = ImpossibleLibrary.sortTokens(tokenA, tokenB);
    (amountA, amountB) = tokenA == token0 ? (amount0, amount1) : (amount1, amount0);
    require(amountA >= amountAMin, 'ImpossibleRouter: INSUFFICIENT_A_AMOUNT');
    require(amountB >= amountBMin, 'ImpossibleRouter: INSUFFICIENT_B_AMOUNT');
}
```

Figure 5 source code of *removeLiquidity* function (fixed)(Extension,sol)

[ImpossiblePair-2 Low] *burn* function design flaws

- Description: The feeto address is receiving a fee when exiting liquidity (the fee is sent to the feeto address). If a feeto address remove from liquidity, it will result in never remove from liquidity.

```
*/
function burn(address to) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
    (uint256 _reserve0, uint256 _reserve1) = getReserves(); // gas savings
    bool feeOn = _mintFee(_reserve0, _reserve1);
    address _token0 = token0; // gas savings
    address _token1 = token1; // gas savings
    uint256 balance0 = IERC20(_token0).balanceOf(address(this));
    uint256 balance1 = IERC20(_token1).balanceOf(address(this));
    uint256 liquidity = balanceOf[address(this)];

    {
        uint256 _totalSupply = totalSupply;
        amount0 = liquidity.mul(balance0) / _totalSupply;
        amount1 = liquidity.mul(balance1) / _totalSupply;
        require(amount0 > 0 && amount1 > 0, 'IF: INSUFFICIENT_LIQUIDITY_BURNED');

        if (feeOn) {
            uint256 _feeRatio = withdrawalFeeRatio; // 1/201 ~= 0.4975%
            amount0 -= amount0.div(_feeRatio);
            amount1 -= amount1.div(_feeRatio);
            // Takes the 0.4975% Fee of LP tokens and adds allowance to claim for the IImpossibleSwapFactory feeTo Address
            uint256 transferAmount = liquidity.div(_feeRatio);
            _safeTransfer(address(this), IImpossibleSwapFactory(factory).feeTo(), transferAmount);
            _burn(address(this), liquidity.sub(transferAmount));
        } else {
            _burn(address(this), liquidity);
        }
        _safeTransfer(_token0, to, amount0);
        _safeTransfer(_token1, to, amount1);
    }

    {
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
        _update(balance0, balance1);
        if (feeOn) kLast = isXybk ? xybkComputeK(balance0, balance1) : balance0.mul(balance1);
    }
    emit Burn(msg.sender, amount0, amount1, to);
}
```

Figure 6 source code of *burn* function(unfixed)

- Fix recommendations: It is suggested to add a judgment that if the to address is a feeto address then no fee will be charged.
- Fix results: Fixed.

```
function burn(address to) external override nonReentrant returns (uint256 amount0, uint256 amount1) {
    (uint256 _reserve0, uint256 _reserve1) = getReserves(); // gas savings
    bool feeOn = _mintFee(_reserve0, _reserve1);
    address _token0 = token0; // gas savings
    address _token1 = token1; // gas savings
    uint256 balance0 = IERC20(_token0).balanceOf(address(this));
    uint256 balance1 = IERC20(_token1).balanceOf(address(this));
    uint256 liquidity = balanceOf[address(this)];

    {
        uint256 _totalSupply = totalSupply;
        amount0 = liquidity.mul(balance0) / _totalSupply;
        amount1 = liquidity.mul(balance1) / _totalSupply;
        require(amount0 > 0 || amount1 > 0, 'IF: INSUFFICIENT_LIQUIDITY_BURNED');

        address _feeTo = IImpossibleSwapFactory(factory).feeTo();
        // Burning fees are paid if burn tx doesnt originate from not IF fee collector
        if (feeOn && tx.origin != _feeTo) {
            uint256 _feeRatio = withdrawalFeeRatio; // default is 1/201 ~= 0.4975%
            amount0 -= amount0.div(_feeRatio);
            amount1 -= amount1.div(_feeRatio);
            // Transfers withdrawalFee of LP tokens to IF feeTo
            uint256 transferAmount = liquidity.div(_feeRatio);
            _safeTransfer(address(this), IImpossibleSwapFactory(factory).feeTo(), transferAmount);
            _burn(address(this), liquidity.sub(transferAmount));
        } else {
            _burn(address(this), liquidity);
        }

        _safeTransfer(_token0, to, amount0);
        _safeTransfer(_token1, to, amount1);
    }

    {
        balance0 = IERC20(_token0).balanceOf(address(this));
        balance1 = IERC20(_token1).balanceOf(address(this));
        _update(balance0, balance1);
        if (feeOn) kLast = isXybk ? xybkComputeK(balance0, balance1) : balance0.mul(balance1);
    }
    emit Burn(msg.sender, amount0, amount1, to);
}
```

Figure 7 source code of *burn* function (fixed)

[ImpossibleWrapperFactory-1 Low] *createPair* function design flaws

- Description: When creating warp, it only verifies that the ratioDenominator is not equal to zero, but does not determine that the ratioNumerator is not equal to zero, and in the calculation, the ratioNumerator is used as the divisor.

```
function createPairing(
    address underlying,
    uint256 ratioNumerator,
    uint256 ratioDenominator
) external onlyGovernance returns (address) {
    require(
        tokensToWrappedTokens[underlying] == address(0x0) && wrappedTokensToTokens[underlying] == address(0x0),
        'IF: PAIR_EXISTS'
    );
    require(ratioDenominator != 0, 'IF: INVALID_DENOMINATOR');
    ImpossibleWrappedToken wrapper = new ImpossibleWrappedToken(underlying, ratioNumerator, ratioDenominator);
    tokensToWrappedTokens[underlying] = address(wrapper);
    wrappedTokensToTokens[address(wrapper)] = underlying;
    emit WrapCreated(underlying, address(wrapper), ratioNumerator, ratioDenominator);
    return address(wrapper);
}
```

Figure 8 source code of *createPairing* function (unfixed)

- Fix recommendations: Suggest adding ratioNumerator != 0.
- Fix results: Fixed.

```
function createPairing(
    address underlying,
    uint256 ratioNumerator,
    uint256 ratioDenominator
) external onlyGovernance returns (address) {
    require(
        tokensToWrappedTokens[underlying] == address(0x0) && wrappedTokensToTokens[underlying] == address(0x0),
        'IF: PAIR_EXISTS'
    );
    require(ratioNumerator != 0 && ratioDenominator != 0, 'IF: INVALID_RATIO');
    ImpossibleWrappedToken wrapper = new ImpossibleWrappedToken(underlying, ratioNumerator, ratioDenominator);
    tokensToWrappedTokens[underlying] = address(wrapper);
    wrappedTokensToTokens[address(wrapper)] = underlying;
    emit WrapCreated(underlying, address(wrapper), ratioNumerator, ratioDenominator);
    return address(wrapper);
}
```

Figure 9 source code of *createPairing* function (fixed)

[ImpossibleWrappedToken-1 Low] *deposit* function event triggering error

- Description: Event trigger error in the deposit function of the ImpossibleWrappedToken contract.

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

Figure 10 source code of *deposit* function (unfixed)

- Fix recommendations: Modify msg.sender to dst.
- Fix results: Fixed.

```
// amt = amount of wrapped tokens
function deposit(address dst, uint256 sendAmt) public override nonReentrant returns (uint256 wad) {
    TransferHelper.safeTransferFrom(address(underlying), msg.sender, address(this), sendAmt);
    uint256 receiveAmt = IERC20(underlying).balanceOf(address(this)).sub(underlyingBalance);
    wad = receiveAmt.mul(ratioNum).div(ratioDenom);
    balanceOf[dst] = balanceOf[dst].add(wad);
    totalSupply = totalSupply.add(wad);
    underlyingBalance = underlyingBalance.add(receiveAmt);
    emit Transfer(address(0), dst, wad);
}
```

Figure 11 source code of *deposit* function (fixed)

[ImpossibleWrappedToken-2 Low] *withdraw* function logic implementation error

- Description: In the ImpossibleWrappedToken contract, updating the value of underlyingBalance is improperly positioned because the value of transferAmt is not currently being calculated, which will result in an inaccurate calculation of the value of underlyingBalance.

```
function withdraw(address dst, uint256 wad) public override nonReentrant returns (uint256 transferAmt) {
    balanceOf[msg.sender] = balanceOf[msg.sender].sub(wad);
    totalSupply = totalSupply.sub(wad);
    underlyingBalance = underlyingBalance.sub(transferAmt);
    transferAmt = wad.mul(ratioDenom).div(ratioNum);
    bool success = underlying.transfer(dst, transferAmt);
    require(success, 'IF Wrapper: UNDERLYING_TRANSFER_FAIL');
    emit Transfer(msg.sender, address(0), wad);
    return transferAmt;
}
```

Figure 12 source code of *withdraw* function (unfixed)

- Fix recommendations: The underlyingBalance value update should be placed after the transferAmt value calculation.
- Fix results: Fixed.

```
// wad = amount of wrapped tokens
function withdraw(address dst, uint256 wad) public override nonReentrant returns (uint256 transferAmt) {
    balanceOf[msg.sender] = balanceOf[msg.sender].sub(wad);
    totalSupply = totalSupply.sub(wad);
    transferAmt = wad.mul(ratioDenom).div(ratioNum);
    TransferHelper.safeTransfer(address(underlying), dst, transferAmt);
    underlyingBalance = underlyingBalance.sub(transferAmt);
    emit Transfer(msg.sender, address(0), wad);
}
```

Figure 13 source code of *withdraw* function (fixed)

[ImpossibleWrappedToken-3 Info] The *transfer* and *transferFrom* functions do not determine the zero address

- Description: In the ImpossibleWrappedToken contract, there is no judgment that the target address of the *transfer* and *transferFrom* functions is not zero.

```
74
75      function transfer(address dst, uint256 wad) public override returns (bool) {
76          return transferFrom(msg.sender, dst, wad);
77      }
78
79      function transferFrom(
80          address src,
81          address dst,
82          uint256 wad
83      ) public override returns (bool) {
84          require(balanceOf[src] >= wad, '');
85
86          if (src != msg.sender && allowance[src][msg.sender] != uint256(-1)) {
87              require(allowance[src][msg.sender] >= wad, 'ImpossibleWrapper: INSUFF_ALLOWANCE');
88              allowance[src][msg.sender] -= wad;
89          }
90
91          balanceOf[src] -= wad;
92          balanceOf[dst] += wad;
93
94          emit Transfer(src, dst, wad);
95
96          return true;
97      }
98  }
99
```

Figure 14 source code of *transfer&transferFrom* function (unfixed)

- Fix recommendations: It is recommended to determine that the target address of the *transfer* and *transferFrom* functions is not zero.
- Fix results: Fixed.

```
    }

    function transfer(address dst, uint256 wad) public override returns (bool) {
        require(dst != address(0x0), 'IF Wrapper: INVALID_DST');
        return transferFrom(msg.sender, dst, wad);
    }

    function transferFrom(
        address src,
        address dst,
        uint256 wad
    ) public override returns (bool) {
        require(balanceOf[src] >= wad, '');
        require(dst != address(0x0), 'IF Wrapper: INVALID_DST');

        if (src != msg.sender && allowance[src][msg.sender] != uint256(-1)) {
            require(allowance[src][msg.sender] >= wad, 'ImpossibleWrapper: INSUFF_ALLOWANCE');
            allowance[src][msg.sender] -= wad;
        }

        balanceOf[src] -= wad;
        balanceOf[dst] += wad;

        emit Transfer(src, dst, wad);

        return true;
    }
}
```

Figure 15 source code of *transfer&transferFrom* function (fixed)

[ImpossibleWrappedToken-4 Info] Redundant code for the *_withdraw* function

- Description: In the ImpossibleWrappedToken contract, the *_withdraw* function does not implement functionality.

```
function _withdraw(address dst, uint256 wad) internal returns (uint256 transferAmt) {}

function amtToUnderlyingAmt(uint256 amt) public view override returns (uint256) {
    return amt.mul(ratioDenom).div(ratioNum);
}
```

Figure 16 source code of *_withdraw* function (unfixed)

- Fix recommendations: Suggested deletion.
- Fix results: Fixed, deleted the code.

[ImpossibleWrappedToken-5 Info] Design flaws in the *deposit* function

- Description: In the ImpossibleWrappedToken contract, only the return value of the *transferFrom* function in the *deposit* function is judged, so some tokens have no return value even if the transaction is successful.

```
// amt = amount of wrapped tokens
function deposit(address dst, uint256 amt) public override nonReentrant returns (uint256 wad) {
    bool success = underlying.transferFrom(msg.sender, address(this), amt);
    require(success, 'ImpossibleWrapper: TRANSFERFROM_FAILED');
    wad = amt.mul(ratioNum).div(ratioDenom);
    balanceOf[dst] = balanceOf[dst].add(wad);
    totalSupply = totalSupply.add(wad);
    underlyingBalance = underlyingBalance.add(amt);
    emit Transfer(address(0), dst, wad);
}
```

Figure 17 source code of *deposit* function (unfixed)

- Fix recommendations: It is recommended to use safeTransferFrom.
- Fix results: Fixed.

```
// amt = amount of wrapped tokens
function deposit(address dst, uint256 sendAmt) public override nonReentrant returns (uint256 wad) {
    TransferHelper.safeTransferFrom(address(underlying), msg.sender, address(this), sendAmt);
    uint256 receiveAmt = IERC20(underlying).balanceOf(address(this)).sub(underlyingBalance);
    wad = receiveAmt.mul(ratioNum).div(ratioDenom);
    balanceOf[dst] = balanceOf[dst].add(wad);
    totalSupply = totalSupply.add(wad);
    underlyingBalance = underlyingBalance.add(receiveAmt);
    emit Transfer(address(0), dst, wad);
}
```

Figure 18 source code of *deposit* function (fixed)

[ImpossibleFactory-1 Low] Design flaws in the *createPair* function

- Description: In the ImpossibleFactory contract, there is no modification of the function interface related to the variable whitelist, which defaults to false and will result in the inability to start the whitelist function.

```
4   import './interfaces/IImpossibleFactory.sol';
5   import './ImpossiblePair.sol';
6
7   contract ImpossibleFactory is IImpossibleFactory {
8       bytes32 public constant INIT_CODE_PAIR_HASH = keccak256(abi.encodePacked(type(ImpossiblePair).creationCode));
9
10      address public override feeTo;
11      address public override governance;
12      address public router;
13      bool whitelist;
14      mapping(address => bool) approvedTokens;
15
16      mapping(address => mapping(address => address)) public override getPair;
17      address[] public override allPairs;
18
19      constructor(address _governance) {
20          governance = _governance;
21      }
22
23      function allPairsLength() external view override returns (uint256) {
24          return allPairs.length;
25      }
26
27      function setRouter(address _router) external {
28          //require(msg.sender == address(governance), "IF: FORBIDDEN");
29          require(router == address(0x0), 'IF: ROUTER_SET');
30          router = _router;
31      }
32
33      function changeTokenAccess(address token, bool allowed) external {
34          require(msg.sender == address(governance), 'IF: FORBIDDEN');
35          approvedTokens[token] = allowed;
36      }
37
38      function createPair(address tokenA, address tokenB) external override returns (address pair) {
39          // tokens must not be identical (i.e. have same address)
40          if (whitelist) {
41              require(approvedTokens[tokenA] && approvedTokens[tokenB], 'IF: Unapproved tokens');
42          }
43          require(tokenA != tokenB, 'IF: IDENTICAL_ADDRESSES');
44          // order token addresses from low to high
```

Figure 19 source code of related code

- Fix recommendations: Suggest adding a function to modify the whitelist.
- Fix results: Fixed.

```
*/
function setWhitelist(bool b) external onlyGovernance {
    whitelist = b;
}
```

Figure 20 source code of *setWhitelist* function

Other Audit Items Descriptions

- Make sure the init code hash is the same as the current pair before deploying the Router address.

Appendix 1 Vulnerability Severity Level

Vulnerability Level	Description	Example
Critical	Vulnerabilities that lead to the complete destruction of the project and cannot be recovered. It is strongly recommended to fix.	Malicious tampering of core contract privileges and theft of contract assets.
High	Vulnerabilities that lead to major abnormalities in the operation of the contract due to contract operation errors. It is strongly recommended to fix.	Unstandardized docking of the USDT interface, causing the user's assets to be unable to withdraw.
Medium	Vulnerabilities that cause the contract operation result to be inconsistent with the design but will not harm the core business. It is recommended to fix.	The rewards that users received do not match expectations.
Low	Vulnerabilities that have no impact on the operation of the contract, but there are potential security risks, which may affect other functions. The project party needs to confirm and determine whether the fix is needed according to the business scenario as appropriate.	Inaccurate annual interest rate data queries.
Info	There is no impact on the normal operation of the contract, but improvements are still recommended to comply with widely accepted common project specifications.	It is needed to trigger corresponding events after modifying the core configuration.

Appendix 2 Description of Audit Categories

No.	Categories	Subitems
1	Coding Conventions	Compiler Version Security
		Deprecated Items
		Redundant Code
		require/assert Usage
		Gas Consumption
2	General Vulnerability	Integer Overflow/Underflow
		Reentrancy
		Pseudo-random Number Generator (PRNG)
		Transaction-Ordering Dependence
		DoS (Denial of Service)
		Function Call Permissions
		call/delegatecall Security
		Returned Value Security
		tx.origin Usage
		Replay Attack
		Overriding Variables
3	Business Security	Business Logics
		Business Implementations

1. Coding Conventions

1.1. Compiler Version Security

The old version of the compiler may cause various known security issues. Developers are advised to specify the contract code to use the latest compiler version and eliminate the compiler alerts.

1.2. Deprecated Items

The Solidity smart contract development language is in rapid iteration. Some keywords have been deprecated by newer versions of the compiler, such as throw, years, etc. To eliminate the potential pitfalls they may cause, contract developers should not use the keywords that have been deprecated by the current compiler version.

1.3. Redundant Code

Redundant code in smart contracts can reduce code readability and may require more gas consumption for contract deployment. It is recommended to eliminate redundant code.

1.4. SafeMath Features

Check whether the functions within the SafeMath library are correctly used in the contract to perform mathematical operations, or perform other overflow prevention checks.

1.5. require/assert Usage

Solidity uses state recovery exceptions to handle errors. This mechanism will undo all changes made to the state in the current call (and all its subcalls) and flag the errors to the caller. The functions assert and require can be used to check conditions and throw exceptions when the conditions are not met. The assert function can only be used to test for internal errors and check non-variables. The require function is used to confirm the validity of conditions, such as whether the input variables or contract state variables meet the conditions, or to verify the return value of external contract calls.

1.6. Gas Consumption

The smart contract virtual machine needs gas to execute the contract code. When the gas is insufficient, the code execution will throw an out of gas exception and cancel all state changes. Contract developers are required to control the gas consumption of the code to avoid function execution failures due to insufficient gas.

1.7. Visibility Specifiers

Check whether the visibility conforms to design requirement.

1.8. Fallback Usage

Check whether the Fallback function has been used correctly in the current contract.

2. General Vulnerability

2.1. Integer overflow

Integer overflow is a security problem in many languages, and they are especially dangerous in smart contracts. Solidity can handle up to 256-bit numbers (2**256-1). If the maximum number is increased by 1, it will overflow to 0. Similarly, when the number is a uint type, 0 minus 1 will underflow to get the maximum number value. Overflow conditions can lead to incorrect results, especially if its possible results are not

expected, which may affect the reliability and safety of the program. For the compiler version after Solidity 0.8.0, smart contracts will perform overflow checking on mathematical operations by default. In the previous compiler versions, developers need to add their own overflow checking code, and SafeMath library is recommended to use.

2.2. Reentrancy

The reentrancy vulnerability is the most typical Ethereum smart contract vulnerability, which has caused the DAO to be attacked. The risk of reentry attack exists when there is an error in the logical order of calling the call.value() function to send assets.

2.3 Pseudo-random Number Generator (PRNG)

Random numbers may be used in smart contracts. In solidity, it is common to use block information as a random factor to generate, but such use is insecure. Block information can be controlled by miners or obtained by attackers during transactions, and such random numbers are to some extent predictable or collidable.

2.4. Transaction-Ordering Dependence

In the process of transaction packing and execution, when faced with transactions of the same difficulty, miners tend to choose the one with higher gas cost to be packed first, so users can specify a higher gas cost to have their transactions packed and executed first.

2.5. DoS(Denial of Service)

DoS, or Denial of Service, can prevent the target from providing normal services. Due to the immutability of smart contracts, this type of attack can make it impossible to ever restore the contract to its normal working state. There are various reasons for the denial of service of a smart contract, including malicious revert when acting as the recipient of a transaction, gas exhaustion caused by code design flaws, etc.

2.6. Function Call Permissions

If smart contracts have high-privilege functions, such as coin minting, self-destruction, change owner, etc., permission restrictions on function calls are required to avoid security problems caused by permission leakage.

2.7. call/delegatecall Security

Solidity provides the call/delegatecall function for function calls, which can cause call injection vulnerability if not used properly. For example, the parameters of the call, if controllable, can control this contract to perform unauthorized operations or call dangerous functions of other contracts.

2.8. Returned Value Security

In Solidity, there are transfer(), send(), call.value() and other methods. The transaction will be rolled back if the transfer fails, while send and call.value will return false if the transfer fails. If the return is not correctly

judged, the unanticipated logic may be executed. In addition, in the implementation of the transfer/transferFrom function of the token contract, it is also necessary to avoid the transfer failure and return false, so as not to create fake recharge loopholes.

2.9. tx.origin Usage

The tx.origin represents the address of the initial creator of the transaction. If tx.origin is used for permission judgment, errors may occur; in addition, if the contract needs to determine whether the caller is the contract address, then tx.origin should be used instead of extcodesize.

2.10. Replay Attack

A replay attack means that if two contracts use the same code implementation, and the identity authentication is in the transmission of parameters, the transaction information can be replayed to the other contract to execute the transaction when the user executes a transaction to one contract.

2.11. Overriding Variables

There are complex variable types in Solidity, such as structures, dynamic arrays, etc. When using a lower version of the compiler, improperly assigning values to it may result in overwriting the values of existing state variables, causing logical exceptions during contract execution.

Appendix 3 Disclaimer

This report is made in response to the project code. No description, expression or wording in this report shall be construed as an endorsement, affirmation or confirmation of the project. This audit is only applied to the type of auditing specified in this report and the scope of given in the results table. Other unknown security vulnerabilities are beyond auditing responsibility. Beosin Technology only issues this report based on the attacks or vulnerabilities that already existed or occurred before the issuance of this report. For the emergence of new attacks or vulnerabilities that exist or occur in the future, Beosin Technology lacks the capability to judge its possible impact on the security status of smart contracts, thus taking no responsibility for them. The security audit analysis and other contents of this report are based solely on the documents and materials that the contract provider has provided to Beosin Technology before the issuance of this report, and the contract provider warrants that there are no missing, tampered, deleted; if the documents and materials provided by the contract provider are missing, tampered, deleted, concealed or reflected in a situation that is inconsistent with the actual situation, or if the documents and materials provided are changed after the issuance of this report, Beosin Technology assumes no responsibility for the resulting loss or adverse effects. The audit report issued by Beosin Technology is based on the documents and materials provided by the contract provider, and relies on the technology currently possessed by Beosin. Due to the technical limitations of any organization, this report conducted by Beosin still has the possibility that the entire risk cannot be completely detected. Beosin disclaims any liability for the resulting losses.

The final interpretation of this statement belongs to Beosin Technology.

Appendix 4 About Beosin

BEOSIN is a leading global blockchain security company dedicated to the construction of blockchain security ecology, with team members coming from professors, post-docs, PhDs from renowned universities and elites from head Internet enterprises who have been engaged in information security industry for many years. BEOSIN has established in-depth cooperation with more than 100 global blockchain head enterprises; and has provided security audit and defense deployment services for more than 1,000 smart contracts, more than 50 blockchain platforms and landing application systems, and nearly 100 digital financial enterprises worldwide. Relying on technical advantages, BEOSIN has applied for nearly 50 software invention patents and copyrights.

BEOSIN

Blockchain Security

Official Website

https://beosin.com

Twitter

https://twitter.com/Beosin_com