SMART CONTRACT AUDIT REPORT

for

FIREBIRD SWAP

Prepared By: Yiqun Chen

Hangzhou, China

June 28, 2021

Document Properties

Client	Firebird Finance
Title	Smart Contract Audit Report
Target	Firebird Swap
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 28, 2021	Shulin Bie	Final Release
1.0-rc	June 25, 2021	Shulin Bie	Release Candidate

Contact

For more information about this document and its contents, please contact PeckShield Inc.

Name	Yiqun Chen
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction 4

1.1 About Firebird Swap . . . 4

1.2 About PeckShield . . . 5

1.3 Methodology . . . 5

1.4 Disclaimer . . . 7

2 Findings 9

2.1 Summary . . . 9

2.2 Key Findings . . . 10

3 Detailed Results 11

3.1 Potential Sandwich/MEV Attack For zapOutToPair() . . . 11

3.2 Accommodation Of Possible Non-Compliant ERC20 Tokens . . . 12

3.3 Suggested Event Generation For setGovernance() . . . 14

3.4 Potential DoS With Permission-Based Operations . . . 15

3.5 Improved Validation Of Function Arguments . . . 17

4 Conclusion 19

References 20

1 Introduction

Given the opportunity to review the `Firebird Swap` design document and related smart contract source code, we in the report outline our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given branch of `Firebird Swap` can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About Firebird Swap

`Firebird` is a one-stop DeFi platform that aims to revolutionize DeFi services by meeting all DeFi needs. The `Swap` feature is an important part of `Firebird`, which is an automated liquidity marketplace that serves as a decentralized exchange and yield-farming platform with the lowest swap fees, the best exchange rates, and price impact on the market.

The basic information of `Firebird Swap` is as follows:

Table 1.1: Basic Information of Firebird Swap

Item	Description
Name	Firebird Finance
Website	https://app.firebird.finance/swap
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 28, 2021

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note this audit will only cover the following three contract files: `FireBirdFactory.sol`, `FireBirdZap.sol` and `FireBirdRouter.sol`.

- https://github.com/firebird-finance/firebird-core.git (645e22f)

And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/firebird-finance/firebird-core.git (0f83e5e)

1.2 About PeckShield

PeckShield Inc. [9] is a leading blockchain security company with the goal of elevating the security, privacy, and usability of current blockchain ecosystems by offering top-notch, industry-leading services and products (including the service of smart contract auditing). We are reachable at Telegram (https://t.me/peckshield), Twitter (http://twitter.com/peckshield), or Email (contact@peckshield.com).

Table 1.2: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would additionally build a PoC to demonstrate the possibility of exploitation. The concrete list of check items is shown in Table 1.3.

In particular, we perform the audit according to the following procedure:

- Basic Coding Bugs: We first statically analyze given smart contracts with our proprietary static code analyzer for known coding bugs, and then manually verify (reject or confirm) all the issues found by our tool.
- Semantic Consistency Checks: We then manually check the logic of implemented smart contracts and compare with the description in the white paper.
- Advanced DeFi Scrutiny: We further review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.
- Additional Recommendations: We also provide additional suggestions regarding the coding and development of smart contracts from the perspective of proven programming practices.

To better describe each issue we identified, we categorize the findings with Common Weakness Enumeration (CWE-699) [7], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release, and does not give any warranties on finding all possible security issues of the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit-based assessment cannot be considered comprehensive, we always recommend proceeding with several independent audits and a public bug bounty program to ensure the security of smart contract(s). Last but not least, this security audit should not be used as investment advice.

Table 1.4: Common Weakness Enumeration (CWE) Classifications Used in This Audit

Category	Summary
Configuration	Weaknesses in this category are typically introduced during the configuration of the software.
Data Processing Issues	Weaknesses in this category are typically found in functionality that processes data.
Numeric Errors	Weaknesses in this category are related to improper calculation or conversion of numbers.
Security Features	Weaknesses in this category are concerned with topics like authentication, access control, confidentiality, cryptography, and privilege management. (Software security is not security software.)
Time and State	Weaknesses in this category are related to the improper management of time and state in an environment that supports simultaneous or near-simultaneous computation by multiple systems, processes, or threads.
Error Conditions, Return Values, Status Codes	Weaknesses in this category include weaknesses that occur if a function does not generate the correct return/status code, or if the application does not handle all possible return/status codes that could be generated by a function.
Resource Management	Weaknesses in this category are related to improper management of system resources.
Behavioral Issues	Weaknesses in this category are related to unexpected behaviors from code that an application uses.
Business Logics	Weaknesses in this category identify some of the underlying problems that commonly allow attackers to manipulate the business logic of an application. Errors in business logic can be devastating to an entire application.
Initialization and Cleanup	Weaknesses in this category occur in behaviors that are used for initialization and breakdown.
Arguments and Parameters	Weaknesses in this category are related to improper use of arguments or parameters within function calls.
Expression Issues	Weaknesses in this category are related to incorrectly written expressions within code.
Coding Practices	Weaknesses in this category are related to coding practices that are deemed unsafe and increase the chances that an exploitable vulnerability will be present in the application. They may not directly introduce a vulnerability, but indicate the product has not been carefully developed or maintained.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `Firebird Swap` implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logic, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	1	■
Informational	1	■
Undetermined	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities, 1 low-severity vulnerability, 1 informational recommendation, and 1 undetermined issue.

Table 2.1: Key Firebird Swap Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Potential Sandwich/MEV Attack For zapOutToPair()	Time and State	Fixed
PVE-002	Low	Accommodation Of Possible Non-Compliant ERC20 Tokens	Coding Practices	Fixed
PVE-003	Informational	Suggested Event Generation For setGovernance()	Status Codes	Fixed
PVE-004	Undetermined	Potential DoS With Permission-Based Operations	Time and State	Confirmed
PVE-005	Medium	Improved Validation Of Function Arguments	Coding Practices	Fixed

Beside the identified issues, we emphasize that for any user-facing applications and services, it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms should kick in at the very moment when the contracts are being deployed on mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Potential Sandwich/MEV Attack For zapOutToPair()

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: FireBirdZap
- Category: Time and State [6]
- CWE subcategory: CWE-682 [4]

Description

In the FireBirdZap contract, we notice the zapOutToPair() function is used to exchange the LP Token to two different kinds of tokens in the pair on behalf of msg.sender. The pair is specified by the first input argument _from of the zapOutToPair() function.

We notice the zapOutToPair() transaction is routed to FireBirdRouter or UniswapV2Router. We will take FireBirdRouter as our example. The FireBirdRouter::removeLiquidity() is called in line 114 to exchange LP Token to two different kinds of tokens in the pair. We observe the fifth input argument amountAMin and the sixth input argument amountBMin of the FireBirdRouter::removeLiquidity() function are both 1, which means this transaction does not specify any restriction on possible slippage and is therefore vulnerable to possible front-running attacks.


```
97      // _from: must be a pair lp
98      function zapOutToPair(address _from, uint amount) public nonReentrant returns (
            uint256 amountA, uint256 amountB) {
99          IERC20(_from).safeTransferFrom(msg.sender, address(this), amount);
100         _approveTokenIfNeeded(_from);

102         IFireBirdPair pair = IFireBirdPair(_from);
103         address token0 = pair.token0();
104         address token1 = pair.token1();
105         bool isfireBirdPair = fireBirdFactory.isPair(_from);

107         if (token0 == WBNB  token1 == WBNB) {
108             if (isfireBirdPair) {
```

```
                (amountA, amountB) = fireBirdRouter.removeLiquidityETH(_from, token0 !=
                    WBNB ? token0 : token1, amount, 1, 1, msg.sender, block.timestamp);
            } else {
                (amountA, amountB) = uniRouter.removeLiquidityETH(token0 != WBNB ?
                    token0 : token1, amount, 1, 1, msg.sender, block.timestamp);
            }
        } else {
            if (isfireBirdPair) {
                (amountA, amountB) = fireBirdRouter.removeLiquidity(_from, token0,
                    token1, amount, 1, 1, msg.sender, block.timestamp);
            } else {
                (amountA, amountB) = uniRouter.removeLiquidity(token0, token1, amount,
                    1, 1, msg.sender, block.timestamp);
            }
        }
    }
```

Listing 3.1: `FireBirdZap::zapOutToPair()`

Recommendation Improve the above function by adding necessary slippage control with the user-specified `amountAMin` and `amountBMin`.

Status The issue has been addressed by the following commit: `5f18a16`.

3.2 Accommodation Of Possible Non-Compliant ERC20 Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `FireBirdZap`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1109 [1]

Description

Though there is a standardized ERC-20 specification, many token contracts may not strictly follow the specification or have additional functionalities beyond the specification. In this section, we examine the `transfer()` routine and possible idiosyncrasies from current widely-used token contracts.

In particular, we use the popular token, i.e., `ZRX`, as our example. We show the related code snippet below. On its entry of `transfer()`, there is a check, i.e., `if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to])`. If the check fails, it returns `false`. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event.*

The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.2: `ZRX.sol`

Because of that, a normal call to `transfer()` is suggested to use the safe version, i.e., `safeTransfer()`, In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful.

In the following, we show the `withdrawTokenAmount()` and `_withdraw()` routines in the `FireBirdZap` contract. If the `USDT` token is supported as `tokenAddress`, the unsafe version of `IERC20(token).transfer(to, amount)` (line 457) and `IERC20(_token).transfer(_to, _balance)` (line 469) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value).

```
455     function withdrawTokenAmount(address token, address to, uint256 amount) external
            onlyGovernance {
456         require(to != address(0), "Zap: Invalid Receiver Address");
457         IERC20(token).transfer(to, amount);
458         emit Withdraw(token, amount, to);
459     }
```

Listing 3.3: `FireBirdZap::withdrawTokenAmount()`

```
461     function _withdraw(address _token, address _to) internal {
462         if (_token == address(0)) {
463             TransferHelper.safeTransferETH(_to, address(this).balance);
464             emit Withdraw(_token, address(this).balance, _to);
465             return;
```

```
        }

        uint256 _balance = IERC20(_token).balanceOf(address(this));
        IERC20(_token).transfer(_to, _balance);
        emit Withdraw(_token, _balance, _to);
    }
```

Listing 3.4: FireBirdZap::_withdraw()

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related transfer().

Status The issue has been addressed by the following commit: 5f18a16.

3.3 Suggested Event Generation For setGovernance()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: FireBirdZap
- Category: Status Codes [6]
- CWE subcategory: CWE-391 [2]

Description

In Ethereum, the event is an indispensable part of a contract and is mainly used to record a variety of runtime dynamics. In particular, when an event is emitted, it stores the arguments passed in transaction logs and these logs are made accessible to external analytics and reporting tools. Events can be emitted in a number of scenarios. One particular case is when system-wide parameters or settings are being changed. Another case is when tokens are being minted, transferred, or burned.

In the following, we use the FireBirdZap contract as an example. While examining the events that reflect the FireBirdZap dynamics, we notice there is a lack of emitting an event to reflect governance changes.

```
    function setGovernance(address _governance) external onlyGovernance {
        governance = _governance;
    }
```

Listing 3.5: FireBirdZap::setGovernance()

With that, we suggest to add a new event NewGovernance whenever the new governance is changed. Also, the new governance information is better indexed. Note each emitted event is represented as a topic that usually consists of the signature (from a keccak256 hash) of the event name and the types (uint256, string, etc.) of its parameters. Each indexed type will be treated like an additional topic.

If an argument is not indexed, it will be attached as data (instead of a separate topic). Considering that the `governance` information is typically queried, it is better treated as a topic, hence the need of being `indexed`.

Recommendation Properly emit the `NewGovernance` event with accurate information to timely reflect state changes. This is very helpful for external analytics and reporting tools.

Status The issue has been addressed by the following commit: `5f18a16`.

3.4 Potential DoS With Permission-Based Operations

- ID: PVE-004
- Severity: Undetermined
- Likelihood: Low
- Impact: Low
- Target: `FireBirdRouter/FireBirdZap`
- Category: Time and State [6]
- CWE subcategory: CWE-682 [4]

Description

The `FireBirdERC20` contract implements the EIP2612 specification by providing the `permit()` support. This specification is proposed to address a limiting factor in the earlier ERC20 design where the ERC20 `approve()` function itself is defined in terms of `msg.sender`. This EIP-2612 specification basically extends the ERC20 standard with a new function `permit()`, which allows users to modify the allowance mapping using a signed message, instead of through `msg.sender`. To elaborate, we show the `permit()` implementation.

```
    function permit(address owner, address spender, uint value, uint deadline, uint8 v, bytes32 r, bytes32 s) external {
        require(deadline >= block.timestamp, 'FLP: EXPIRED');
        bytes32 digest = keccak256(
            abi.encodePacked(
                '\x19\x01',
                DOMAIN_SEPARATOR,
                keccak256(abi.encode(PERMIT_TYPEHASH, owner, spender, value, nonces[owner]++, deadline))
            )
        );
        address recoveredAddress = ecrecover(digest, v, r, s);
        require(recoveredAddress != address(0) && recoveredAddress == owner, 'FLP: INVALID_SIGNATURE');
        _approve(owner, spender, value);
    }
```

Listing 3.6: `FireBirdERC20::permit()`

We notice the `FireBirdERC20::permit()` is called by `FireBirdRouter::removeLiquidityWithPermit()`, `FireBirdRouter::removeLiquidityETHWithPermit()`, `FireBirdRouter::removeLiquidityETHWithPermit-SupportingFeeOnTransferTokens()`, `FireBirdZap::zapOutToPairWithPermit()` and `FireBirdZap::zapOutWith-Permit()`. We will take the `FireBirdRouter::removeLiquidityWithPermit()` function as our example to describe this vulnerability.

To elaborate, we show below the related code snippet of the `FireBirdRouter` contract. In the `removeLiquidityWithPermit()` function, the `FireBirdERC20::permit()` function is called (line 584) to assign the allowance of `msg.sender` to the `FireBirdRouter` contract before removing the liquidity of `msg.sender`. If the attacker front-runs the `FireBirdERC20::permit()` function with the same arguments, the normal trade from the user will be blocked as the signed message has become out-of-work. So far, we also do not know how an attacker can exploit this vulnerability to earn profit. After internal discussion, we determined this vulnerability still need to be brought up and paid more attention to.

```
571     function removeLiquidityWithPermit(
572         address pair,
573         address tokenA,
574         address tokenB,
575         uint liquidity,
576         uint amountAMin,
577         uint amountBMin,
578         address to,
579         uint deadline,
580         bool approveMax, uint8 v, bytes32 r, bytes32 s
581     ) external virtual override ensure(deadline) returns (uint amountA, uint amountB) {
582         {
583             uint value = approveMax ? uint(-1) : liquidity;
584             IFireBirdPair(pair).permit(msg.sender, address(this), value, deadline, v, r,
                    s);
585         }
586         (amountA, amountB) = _removeLiquidity(pair, tokenA, tokenB, liquidity,
                amountAMin, amountBMin, to);
587     }
```

Listing 3.7: `FireBirdRouter::removeLiquidityWithPermit()`

Recommendation The severity of this issue is still undetermined as it largely depends on the context of being used. Fortunately, there is an alternative version without making use of the `Permit` feature.

Status This issue has been confirmed. Considering that this is part of the original `Uniswap` code base, the team decides to leave it as is to minimize the difference from the original `Uniswap` and reduce the risk of introducing bugs as a result of changing the behavior.

3.5 Improved Validation Of Function Arguments

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `FireBirdRouter`
- Category: Coding Practices [5]
- CWE subcategory: CWE-628 [3]

Description

In the `FireBirdRouter` contract, we observe the `_swapSingleSupportFeeOnTransferTokens()` function is used to swap the exact amount of `tokenIn` to `tokenOut`. To elaborate, we show below the related code snippet. In the function, the `FireBirdFormula::getFactoryReserveAndWeights()` function is called (line 460) to acquire the current state of the pool which is a `FireBirdPair` instance that stores two different tokens address and the related info, including `reserve0`, `reserve1`, `tokenWeight0`, `swapFee`, and so on. These parameters of the pool will be used by `FireBirdFormula::getAmountOut()` (line 462) to calculate the mount of `tokenOut` swapped with the exact amount of `tokenIn`.

In the `FireBirdFormula::getFactoryReserveAndWeights()` function, we notice if the third input argument `tokenA` is not equal to the `token0` address in the pool specified by the second input argument `pair`, `tokenA` will be treated as `token1` and the function will return the corresponding state of the pool. If `tokenA` is neither equal to `token0`, nor equal to `token1`, the function will return incorrect values, which may introduce unexpected behavior for those functions call it. We may need to validate `tokenA` is either equal to `token0` or equal to `token1` at the beginning of the function.

```
26    function _swapSingleSupportFeeOnTransferTokens(address tokenIn, address tokenOut,
          address pool, uint swapAmount, uint limitReturnAmount) internal returns(uint
          tokenAmountOut) {
27        TransferHelper.safeTransfer(tokenIn, pool, swapAmount);

29        uint amountOutput;
30        {
31            (, uint reserveInput, uint reserveOutput, uint32 tokenWeightInput, uint32
                  tokenWeightOutput, uint32 swapFee) = IFireBirdFormula(formula).
                  getFactoryReserveAndWeights(factory, pool, tokenIn);
32            uint amountInput = IERC20(tokenIn).balanceOf(pool).sub(reserveInput);
33            amountOutput = IFireBirdFormula(formula).getAmountOut(amountInput,
                  reserveInput, reserveOutput, tokenWeightInput, tokenWeightOutput,
                  swapFee);
34        }
35        uint balanceBefore = IERC20(tokenOut).balanceOf(address(this));
36        (uint amount0Out, uint amount1Out) = tokenIn == IFireBirdPair(pool).token0() ? (
              uint(0), amountOutput) : (amountOutput, uint(0));
37        IFireBirdPair(pool).swap(amount0Out, amount1Out, address(this), new bytes(0));
38        emit Exchange(pool, amountOutput, tokenOut);
```

```
        tokenAmountOut = IERC20(tokenOut).balanceOf(address(this)).sub(balanceBefore);
        require(tokenAmountOut >= limitReturnAmount,'Router: INSUFFICIENT_OUTPUT_AMOUNT'
            );
    }
```

Listing 3.8: `FireBirdRouter::_swapSingleSupportFeeOnTransferTokens()`

```
    function getFactoryReserveAndWeights(address factory, address pair, address tokenA)
        public override view returns (
        address tokenB,
        uint reserveA,
        uint reserveB,
        uint32 tokenWeightA,
        uint32 tokenWeightB,
        uint32 swapFee
    ) {
        address token0 = IFireBirdPair(pair).token0();
        (uint reserve0, uint reserve1,) = IFireBirdPair(pair).getReserves();
        uint32 tokenWeight0;
        uint32 tokenWeight1;
        (tokenWeight0, tokenWeight1, swapFee) = getFactoryWeightsAndSwapFee(factory,
            pair);

        if (tokenA == token0) {
            (tokenB, reserveA, reserveB, tokenWeightA, tokenWeightB) = (IFireBirdPair(
                pair).token1(), reserve0, reserve1, tokenWeight0, tokenWeight1);
        } else {
            (tokenB, reserveA, reserveB, tokenWeightA, tokenWeightB) = (token0, reserve1
                , reserve0, tokenWeight1, tokenWeight0);
        }
    }
```

Listing 3.9: `FireBirdFormula::getFactoryReserveAndWeights()`

Note a number of functions can be similarly improved, including `FireBirdFormula::getReserveAnd-Weights()`, `FireBirdFormula::getReserves()` and `FireBirdFormula::getOtherToken()`.

Recommendation Validate whether the token address is in the `pair` at the beginning of the functions.

Status The issue has been addressed by the following commit: `0f83e5e`.

4 | Conclusion

In this audit, we have analyzed the `Firebird Swap` design and implementation. As an important part of `Firebird` which is a one-stop DeFi platform that aims to revolutionize DeFi services by meeting all DeFi needs, `Firebird Swap` is an automated liquidity marketplace that serves as a decentralized exchange and yield-farming platform with the lowest swap fees, the best exchange rates, and price impact on the market. The current code base is well organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

[1] MITRE. CWE-1126: Declaration of Variable with Unnecessarily Wide Scope. https://cwe.mitre.org/data/definitions/1126.html.

[2] MITRE. CWE-391: Unchecked Error Condition. https://cwe.mitre.org/data/definitions/391.html.

[3] MITRE. CWE-628: Function Call with Incorrectly Specified Arguments. https://cwe.mitre.org/data/definitions/628.html.

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.html.

[5] MITRE. CWE CATEGORY: Bad Coding Practices. https://cwe.mitre.org/data/definitions/1006.html.

[6] MITRE. CWE CATEGORY: Error Conditions, Return Values, Status Codes. https://cwe.mitre.org/data/definitions/389.html.

[7] MITRE. CWE VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

[8] OWASP. Risk Rating Methodology. https://www.owasp.org/index.php/OWASP_Risk_Rating_Methodology.

[9] PeckShield. PeckShield Inc. https://www.peckshield.com.