SMART CONTRACT AUDIT REPORT

for

COFIX COMMUNITY

PeckShield

Prepared By: Shuxiao Wang

Hangzhou, China
November 17, 2020

Document Properties

Client	CoFiX Community
Title	Smart Contract Audit Report
Target	CoFiX
Version	1.0
Author	Xuxian Jiang
Auditors	Huaguo Shi, Jeff Liu, Xuxian Jiang
Reviewed by	Jeff Liu
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 17, 2020	Xuxian Jiang	Final Release
1.0-rc1	November 15, 2020	Xuxian Jiang	Release Candidate #1
0.4	November 10, 2020	Xuxian Jiang	Additional Findings #3
0.3	November 6, 2020	Xuxian Jiang	Additional Findings #2
0.2	November 2, 2020	Xuxian Jiang	Additional Findings #1
0.1	October 31, 2020	Xuxian Jiang	Initial Draft

Contact

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

Name	Shuxiao Wang
Phone	+86 173 6454 5338
Email	contact@peckshield.com

Contents

PeckShield

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the **CoFiX** protocol, 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 CoFiX 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 CoFiX Protocol

CoFiX is a fully decentralized exchange protocol that allows for calculating and managing inherent risks in current DeFi protocols. In particular, by integrating a reliable price feed with a computable risk factor from NEST, CoFiX can reliably bring accurate market prices and computable risks for traders and market makers. With the novel capability of pricing in the computable risks with each respective trade, CoFiX ensures that market makers are motivated to make the market, and the traders trade with minimal price spread. CoFiX pushes forward the current DEX frontline and presents a valuable contribution to current DeFi ecosystem.

The basic information of the CoFiX protocol is as follows:

Table 1.1: Basic Information of the CoFiX Protocol

Item	Description
Issuer	CoFiX Community
Website	https://cofix.io/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 17, 2020

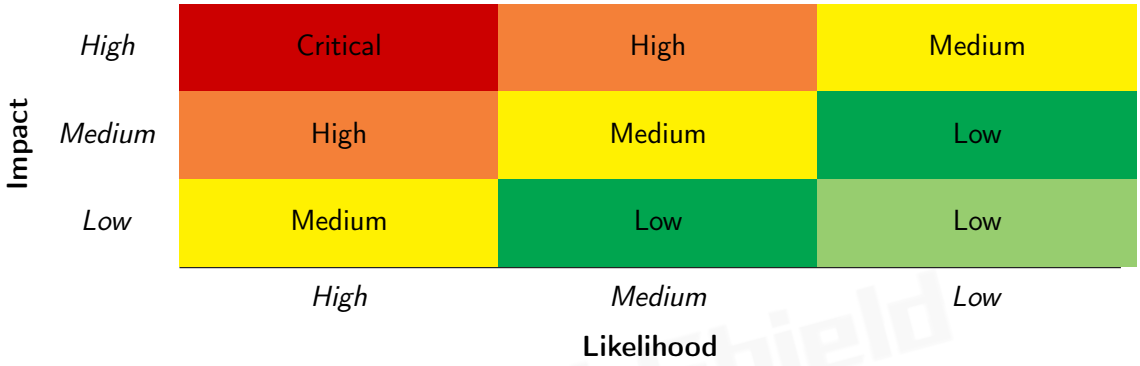
In the following, we show the Git repository of reviewed files and the commit hash value used in this audit:

- https://github.com/Computable-Finance/CoFiX.git (9306e6e)

1.2 About PeckShield

PeckShield Inc. [16] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [11]:

- 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.

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

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 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) [10], 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 audit does not give any warranties on finding all possible security issues of the given smart contract(s), 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 CoFiX 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 logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	2	■■
Medium	2	■■
Low	5	■■■■■
Informational	3	■■■
Total	12	

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 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 high-severity vulnerability, 2 medium-severity vulnerabilities, 5 low-severity vulnerabilities, and 3 informational recommendations.

Table 2.1: Key CoFiX Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Gas-Efficient New CoFiXPair Deployment	Coding Practices	Confirmed
PVE-002	High	Double Price Compensations in burn() For Non-WETH Withdrawals	Business Logic	Confirmed
PVE-003	Low	Improved Event Generation	Error Conditions, Return Values, Status Codes	Confirmed
PVE-004	High	Leftover Return in swapTokensForExactETH()	Business Logic	Confirmed
PVE-005	Low	Improved transferFrom() in CoFiXERC20	Coding Practices	Confirmed
PVE-006	Informational	Improved Pair Validation in CoFiXRouter	Coding Practices	Confirmed
PVE-007	Informational	Inconsistency Between Documentation and Implementation	Coding Practices	Fixed
PVE-008	Low	Fallback Conversion of ETH to WETH in CoFixPair	Business Logic	Confirmed
PVE-009	Medium	Potential Lockup of User Funds in Staking	Business Logic	Confirmed
PVE-010	Low	Requirement of Oracle For Pair Creation	Business Logic	Confirmed
PVE-011	Informational	Removal of Unused Code	Coding Practices	Confirmed
PVE-012	Medium	Trust Issue of Admin Keys Behind CoFiToken	Security Features	Confirmed

Beside the identified issues, we also note that the current implementation does not support deflationary tokens (e.g., in `CoFiXRouter`). Also, the known front-running issue is inherent in current DEXs and `CoFiX` is no exception. Last, 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 Gas-Efficient New Pair Deployment

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CoFiXFactory`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In CoFiX, `CoFiXPair` acts as a trustless intermediary between liquidity providers and trading users. The liquidity providers `deposit` certain amounts of `assets` into the `CoFiXPair` pool and in return get the tokenized pool share of current reserves. Later on, the liquidity providers can `withdraw` their own share by returning the pool tokens back to the pool. With assets in the pool, users can submit `swap` requests with managed risks and the trading price is determined according to the reliable price feed from `NEST`, instead of the AMM price curve in `Uniswap`.

When the pool does not exist, the first liquidity provider's `addLiquidity()` operation will trigger the creation of the pool (via the `createPair()` function). As the name indicates, `createPair()` performs necessary sanity checks and then instantiates the pool contract creation (line 55): `pair := create2(0, add(bytecode, 32), mload(bytecode), salt)`.

```
49      function createPair(address token) external override returns (address pair) {
50          require(token != address(0), 'CFactory: ZERO_ADDRESS');
51          require(getPair[token] == address(0), 'CFactory: PAIR_EXISTS');
52          bytes memory bytecode = type(CoFiXPair).creationCode;
53          bytes32 salt = keccak256(abi.encodePacked(token));
54          assembly {
55              pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
56          }
57          require(pair != address(0), "CFactory: Failed on deploy");
58
59          getPair[token] = pair;
60          allPairs.push(pair);
```

```
61
62          uint256 pairLen = allPairs.length;
63          string memory _idx = uint2str(pairLen);
64          string memory _name = append(pairNamePrefix, _idx);
65          string memory _symbol = append(pairSymbolPrefix, _idx);
66          ICoFiXPair(pair).initialize(WETH, token, _name, _symbol);
67
68          ICoFiXController(controller).addCaller(pair);
69          emit PairCreated(token, pair, pairLen);
70      }
```

Listing 3.1: CoFiXFactory.sol

The pair contract is a complicated one and its instantiation inevitably consumes significant amount of gas. Such gas-consuming pool contract deployment would discourage liquidity providers' engagement. An alternative would be to explore a proxy-based approach by implementing the pool contract as a logic one. By doing so, we only need to deploy a minimal proxy for each pair, hence lowering the entry barrier for liquidity providers, especially for the creation of trading pools. Recall that in order to prevent the first liquidity provider from monopolizing the liquidity pool, the provider has been penalized by forcibly burning the very first `MINIMUM_LIQUIDITY = 10 ** 9` pool shares. It is just not justifiable to further penalize early liquidity providers who introduce the trading pools into the CoFiX ecosystem!

Recommendation Explore the proxy-based approach of deploying pool contracts to lower the barrier for early participation.

Status The issue has been confirmed. Considering that there is only a limited number of pairs being deployed and the gas cost for the pair deployment is not a concern yet, the team decides to leave it as is and plans to consider the suggested deployment in CoFiX V2.

3.2 Double Price Compensations in burn() For Non-WETH Withdrawals

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `CoFiXPair`
- Category: Business Logic [8]
- CWE subcategory: CWE-754 [5]

Description

CoFiX defines a standard interface for both market makers and traders. In particular, market makers can `mint()`/`burn()` pool tokens by depositing/withdrawing supported assets into/out of created

pairs (instantiated from `CoFiXPair`). The traders can swap one supported asset for another (via `swapWithExact()`/`swapForExact()`). CoFiX also develops unique incentive mechanisms that allow not only market makers to stake the minted pool tokens for additional `CoFi` rewards, but also traders to participate in trade mining with `CoFi` rewards. Note that `CoFi` is the protocol-wide governance token in CoFiX.

CoFiX is innovative in taking advantage of a decentralized oracle like `NEST` to effectively calculate and manage associated risks. In particular, to accommodate the deviation of oracle prices and the delay between current block and the block with the latest effective `NEST` price, CoFiX develops a price compensation risk factor, i.e., K, when quoting prices from `NEST`. This compensation factor K is a coefficient related to the asset volatility rate δ and delay T. With K, when a trader makes a transaction, he does not directly use price P, but rather $P' = P * (1 + K)$ (when buying) or $P' = P * (1 - K)$ (when selling). The same K is also applicable for price compensations when liquidity providers `mint()`/`burn()` pool tokens.

In the following, we show the implementation of `burn()` and focus on the application of the compensation factor K when pool tokens are being burnt. Our analysis shows there is an undesirable issue in its execution logic that inappropriately applies the price compensation twice when non-`WETH` tokens are requested for withdrawals.

```
125     function burn(address outToken, address to) external payable override lock returns (
            uint amountOut, uint oracleFeeChange) {
126         address _token0 = token0;                                // gas savings
127         address _token1 = token1;                                // gas savings
128         uint balance0 = IERC20(_token0).balanceOf(address(this));
129         uint balance1 = IERC20(_token1).balanceOf(address(this));
130         uint liquidity = balanceOf[address(this)];

132         uint256 _ethBalanceBefore = address(this).balance;
133         uint256 fee;
134         {
135             bytes memory data = abi.encode(msg.sender, outToken, to, liquidity);
136             // query price
137             OraclePrice memory _op;
138             (_op.K, _op.ethAmount, _op.erc20Amount, _op.blockNum, _op.theta) =
                    _queryOracle(_token1, CoFiX_OP.BURN, data);
139             if (outToken == _token0) {
140                 (amountOut, fee) = calcOutToken0ForBurn(liquidity, _op); // navps
                        calculated
141             } else if (outToken == _token1) {
142                 (amountOut, fee) = calcOutToken1ForBurn(liquidity, _op); // navps
                        calculated
143             } else {
144                 revert("CPair: wrong outToken");
145             }
146         }
147         oracleFeeChange = msg.value.sub(_ethBalanceBefore.sub(address(this).balance));
```

```
149         require(amountOut > 0, "CPair: SHORT_LIQUIDITY_BURNED");
150         _burn(address(this), liquidity);
151         _safeTransfer(outToken, to, amountOut);
152         ...
153     }
```

Listing 3.2: CoFiXPair.sol

Specifically, if a liquidity provider calls to `burn(USDT, to)` pool tokens of current `WETH-USDT` pool, this routine executes by calculating and transferring the `amountOut` of `USDT`. This amount is computed in the helper routine `calcOutToken1ForBurn()` (line 142). To elaborate, we further show its code snippet below.

```
461     // calc amountOut for token1 (ERC20 token) when send liquidity token to pool for
            burning
462     function calcOutToken1ForBurn(uint256 liquidity, OraclePrice memory _op) public view
            returns (uint256 amountOut, uint256 fee) {
463         /*
464         u &= c * (N_{p}^{'} / NAVPS_{BASE}) * P_{s}^{'} * (THETA_{BASE} - \theta)/THETA_
                {BASE} \\\\
465           &= c * \frac{N_{p}^{'}}{NAVPS_{BASE}} * \frac{erc20Amount}{ethAmount} * \frac
                {(k_{BASE} - k)}{(k_{BASE})} * \frac{THETA_{BASE} - \theta}{THETA_{BASE}}
                \\\\
466           &= \frac{c * N_{p}^{'} * erc20Amount * (k_{BASE} - k) * (THETA_{BASE} - \theta
                )}{NAVPS_{BASE}*ethAmount*k_{BASE}*THETA_{BASE}}
467         // amountOut = liquidity * navps * _op.erc20Amount * (K_BASE - _op.K) * (
                THETA_BASE - _op.theta) / NAVPS_BASE / _op.ethAmount / K_BASE / THETA_BASE;
468         */
469         uint256 navps = calcNAVPerShareForBurn(reserve0, reserve1, _op);
470         uint256 liqMulMany = liquidity.mul(navps).mul(_op.erc20Amount).mul(K_BASE.sub(
                _op.K)).mul(THETA_BASE.sub(_op.theta));
471         amountOut = liqMulMany.div(NAVPS_BASE).div(_op.ethAmount).div(K_BASE).div(
                THETA_BASE);
472         if (_op.theta != 0) {
473             // fee = liquidity * navps * (_op.theta) / NAVPS_BASE / THETA_BASE;
474             fee = liquidity.mul(navps).mul(_op.theta).div(NAVPS_BASE).div(THETA_BASE);
475         }
476         return (amountOut, fee);
477     }
```

Listing 3.3: CoFiXPair.sol

In essence, this routine computes the amount according to the equation: `amountOut = liquidity * navps *(1-theta)*(P(1-K))` (line 471) where `navps` denotes the net asset value per share. This number is returned from another helper routine `calcNAVPerShareForBurn()` as `navps = ( balance0 + balance1/(P*(1+K)))/ _totalSupply` (line 373).

```
354     // calc Net Asset Value Per Share for burn
355     // use it in this contract, for optimized gas usage
356     function calcNAVPerShareForBurn(uint256 balance0, uint256 balance1, OraclePrice
            memory _op) public view returns (uint256 navps) {
```

```
357         uint _totalSupply = totalSupply;
358         if (_totalSupply == 0) {
359             navps = NAVPS_BASE;
360         } else {
361             /*
362             N_{p}^{'} &= (A_{u}/P_{b}^{'} + A_{e})/S \\\\
363                       &= (A_{u}/(P * (1 + K)) + A_{e})/S \\\\
364                       &= (\frac{A_{u}}{\frac{erc20Amount}{ethAmount} * \frac{(k_{BASE} +
                              k)}{(k_{BASE})}} + A_{e})/S \\\\
365                       &= (\frac{A_{u}*ethAmount*k_{BASE}}{erc20Amount*(k_{BASE} + k)}+
                              A_{e}) / S \\\\
366                       &= (A_{u}*ethAmount*k_{BASE}+ A_{e}*erc20Amount*(k_{BASE} + k)) /
                              S / (erc20Amount*(k_{BASE} + k)) \\\\
367             N_{p}^{'} &= NAVPS_{BASE}*(A_{u}*ethAmount*k_{BASE}+ A_{e}*erc20Amount*(k_{
                    BASE} + k)) / S / (erc20Amount*(k_{BASE} + k)) \\\\
368             // navps = NAVPS_BASE * ( (balance1*_op.ethAmount*K_BASE) + (balance0*_op.
                    erc20Amount*(K_BASE+_op.K)) ) / _totalSupply / _op.erc20Amount / (K_BASE
                    +_op.K);
369             */
370             uint256 kbaseAddK = K_BASE.add(_op.K);
371             uint256 balance1MulEthKbase = balance1.mul(_op.ethAmount).mul(K_BASE);
372             uint256 balance0MulErcKbsk = balance0.mul(_op.erc20Amount).mul(kbaseAddK);
373             navps = NAVPS_BASE.mul( (balance1MulEthKbase).add(balance0MulErcKbsk) ).div(
                    _totalSupply).div(_op.erc20Amount).div(kbaseAddK);
374         }
375     }
```

Listing 3.4: CoFiXPair.sol

As a result, the above equation can be further expanded as follows: `amountOut = share * ( balance0*(P*(1-K))+ balance1*((1-K)/(1+K)))*(1-theta)` where `share = liquidity / _totalSupply`. Apparently, the calculation applies the compensation factor k to `WETH`'s `balance0` in the form of `balance0*(P*(1-K))`. This is reasonable as the corresponding portion of `WETH` in the burnt `liquidity` is swapped to `USDT`. However, the double application of k to `USDT`'s `balance1` in the form of `balance1 *((1-K)/(1+K))` is unjustified. The correct calculation is `balance1*(P*(1-K))`, similar to the `WETH` case.

It is important to note that the above application (of double compensation factors) reduces the amount at the cost of liquidity providers.

Recommendation Avoid applying double price compensations in `burn()` when the non-`WETH` asset is requested for withdrawal.

Status This issue has been confirmed. If the `outToken` is not `ETH`, liquidity providers could spend more fees when redeeming. However, the team considers the related logic is by design and follows the design document. In addition, `CoFiX` does not need the amounts of assets in the trading pool to be well balanced. And liquidity providers are free to choose which assets to redeem rather than redeeming at a specific percentage. Meanwhile, the team may explore a more accurate way as suggested in the next major update.

3.3 Improved Event Generation

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Multiple Contracts
- Category: Status Codes [9]
- CWE subcategory: CWE-391 [4]

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 CoFiXPair contract as an example. This contract is designed to act as a trustless intermediary between liquidity providers and trading users. While examining the events that reflect the pool dynamics, we notice the emitted Swap event (line 315) contains incorrect information. Specifically, the event is defined as event Swap(address indexed sender, uint amountIn , uint amountOut, address outToken, address indexed to)) with a number of parameters: the first parameter sender encodes the address that performs the swap operation; the following two parameters amountIn and amountOut show the input and output amounts, respectively; the next parameter outToken indicates the requested token; while the last parameter to indicates the recipient of the swapped asset. The emitted event contains an incorrect amountIn information, which should be _amountInNeeded , instead of current amountIn (line 315). Note _amountInNeeded is the actual amount spent in this specific swap.

```
294         {
295             require(to != _token0 && to != _token1, "CPair: INVALID_TO");
296
297             amountOut = amountOutExact;
298             _safeTransfer(outToken, to, amountOut); // optimistically transfer tokens
299             if (tradeInfo[0] > 0) {
300                 if (ICoFiXFactory(factory).getTradeMiningStatus(_token1)) {
301                     // only transfer fee to protocol feeReceiver when trade mining is
                            enabled for this trading pair
302                     _safeSendFeeForCoFiHolder(_token0, tradeInfo[0]);
303                 } else {
304                     _safeSendFeeForLP(_token0, _token1, tradeInfo[0]);
305                     tradeInfo[0] = 0; // so router won't go into the trade mining logic
                            (reduce one more call gas cost)
306                 }
307             }
```

```
308             uint256 balance0 = IERC20(_token0).balanceOf(address(this));
309             uint256 balance1 = IERC20(_token1).balanceOf(address(this));
310
311             _update(balance0, balance1);
312             if (oracleFeeChange > 0) TransferHelper.safeTransferETH(msg.sender,
                    oracleFeeChange);
313         }
314
315         emit Swap(msg.sender, amountIn, amountOut, outToken, to);
```

Listing 3.5: CoFiXPair.sol

Moreover, it comes to our attention that the event `RewardAdded` has not `indexed` the user information. Note that 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, which means it will be attached as data (instead of a separate topic). Considering that the user is typically queried, it is typically treated as a topic, hence the need of being `indexed`.

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

Status The issue has been confirmed. The `Swap` event in the `swapForExact()` function is not accurate. However, it does not affect the funds in the contract. The team would fix this in the next major update. The improvement of the `RewardAdded` event would be adopted in the next minor update.

3.4 Leftover Return in swapTokensForExactETH()

- ID: PVE-004
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `CoFiXRouter`
- Category: Business Logic [8]
- CWE subcategory: CWE-754 [5]

Description

The CoFiX protocol is heavily inspired from the popular `Uniswap` and architecturally shares similar components. In particular, CoFiX also contains the core and periphery contracts. The periphery contract is mainly implemented in `CoFiXRouter` and provides a number of wrapper routines. In the following, we examine a specific routine named `swapTokensForExactETH()`.

As the name indicates, this routine is designed to facilitate the swap of a specific asset for an exact amount of `ETH`. To elaborate, we show its code snippet below. The execution logic is rather

straightforward in firstly transferring the `amountInMax` of the swapped asset to the pool, then calling the intended `swapForExact()` function from the core contract, next distributing trading rewards, if any, to the trader, and finally refund oracle fee to `msg.sender`.

```
288     // msg.value = oracle fee
289     function swapTokensForExactETH(
290         address token,
291         uint amountInMax,
292         uint amountOutExact,
293         address to,
294         address rewardTo,
295         uint deadline
296     ) external override payable ensure(deadline) returns (uint _amountIn, uint
            _amountOut)
297     {
298         require(msg.value > 0, "CRouter: insufficient msg.value");
299         address pair = pairFor(factory, token);
300         TransferHelper.safeTransferFrom(token, msg.sender, pair, amountInMax);
301         uint oracleFeeChange;
302         uint256[4] memory tradeInfo;
303         (_amountIn, _amountOut, oracleFeeChange, tradeInfo) = ICoFiXPair(pair).
                swapForExact{
304             value: msg.value}(WETH, amountOutExact, address(this));
305         // assert amountOutExact equals with _amountOut
306         require(_amountIn <= amountInMax, "CRouter: got less than expected");
307         IWETH(WETH).withdraw(_amountOut);
308
309         // distribute trading rewards - CoFi!
310         address vaultForTrader = ICoFiXFactory(factory).getVaultForTrader();
311         if (tradeInfo[0] > 0 && rewardTo != address(0) && vaultForTrader != address(0))
                {
312             ICoFiXVaultForTrader(vaultForTrader).distributeReward(pair, tradeInfo[0],
                    tradeInfo[1], tradeInfo[2], tradeInfo[3], rewardTo);
313         }
314
315         TransferHelper.safeTransferETH(to, amountOutExact);
316         // refund oracle fee to msg.sender, if any
317         if (oracleFeeChange > 0) TransferHelper.safeTransferETH(msg.sender,
                oracleFeeChange);
318     }
```

Listing 3.6: CoFiXRouter.sol

We note that this routine initially transfers `amountInMax` (line 300) of the swapped asset to the pool. However, the pool may only consumes `amountIn` as returned from `swapForExact()` (line 303). Therefore, in case there is any leftover (`amountInMax - amountIn > 0`), the leftover needs to returned back to the sender, i.e., `msg.sender`. However, this is not the case as the leftover simply stays in the periphery contract.

Recommendation Properly return the leftover amount, if any, back to the `msg.sender`.

Status The issue has been confirmed. The `swapETHForExactTokens()` and `swapTokensForExactETH()` functions are an experimental feature and not in production use for official clients. The team plans to disable the feature temporarily and improve these functions in the next minor update of the `CoFiXRouter` contract.

3.5 Improved transferFrom() in CoFiXERC20

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CoFiXERC20`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

In CoFiX, each pool token is compliant with the ERC20 standard so that it can be readily used for staking and other purposes. While reviewing the pool token implementation, we notice its `transferFrom()` routine can be improved.

Specifically, the current implementation (see the code below) makes a check on whether the sender has the spending allowance: If not, the `safemath` operation (line 79) will simply revert this transaction; If yes, the underlying `_transfer()` handler will transfer the requested assets from the source to the recipient.

```
77    function transferFrom(address from, address to, uint value) external override
          returns (bool) {
78        if (allowance[from][msg.sender] != uint(-1)) {
79            allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
80        }
81        _transfer(from, to, value);
82        return true;
83    }
```

Listing 3.7: CoFiXERC20.sol

We notice that a corner case can be better handled when `from == msg.sender`. It is reasonable that the owner is allowed to call `transferFrom()` while specifying himself as the first parameter. However, without prior allowance on himself, the current implementation simply reverts the transaction.

Recommendation To accommodate the corner case, the `transferFrom()` routine can be adjusted as follows:

```
77    function transferFrom(address from, address to, uint value) external override
          returns (bool) {
78        if (from != msg.sender && allowance[from][msg.sender] != uint(-1)) {
```

```
79              allowance[from][msg.sender] = allowance[from][msg.sender].sub(value);
80          }
81          _transfer(from, to, value);
82          return true;
83      }
```

Listing 3.8: CoFiXERC20.sol

Status The issue has been confirmed. The current code is a standard ERC20 token implementation with optimization on unlimited allowance features to reduce gas costs. The team would consider adopting the proposed improvement in the future.

3.6 Improved Pair Validation in CoFiXRouter

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CoFiXRouter`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.4, the CoFiX protocol is heavily inspired from the popular `Uniswap` and the implementation of CoFiX is similarly separated into the core and periphery contracts. The periphery contract is implemented in `CoFiXRouter` and provides a number of wrapper routines.

While analyzing these wrapper routines, we notice there is a constant need of validating the core pool address for interaction. For elaboration, we show below the `swapExactETHForTokens()` routine.

```
155     // msg.value = amountIn + oracle fee
156     function swapExactETHForTokens(
157         address token,
158         uint amountIn,
159         uint amountOutMin,
160         address to,
161         address rewardTo,
162         uint deadline
163     ) external override payable ensure(deadline) returns (uint _amountIn, uint
            _amountOut)
164     {
165         require(msg.value > amountIn, "CRouter: insufficient msg.value");
166         IWETH(WETH).deposit{value: amountIn}();
167         address pair = pairFor(factory, token);
168         assert(IWETH(WETH).transfer(pair, amountIn));
169         uint oracleFeeChange;
170         uint256[4] memory tradeInfo;
```

```
171         (_amountIn, _amountOut, oracleFeeChange, tradeInfo) = ICoFiXPair(pair).
                swapWithExact{
172             value: msg.value.sub(amountIn)}(token, to);
173         require(_amountOut >= amountOutMin, "CRouter: got less than expected");

175         // distribute trading rewards - CoFi!
176         address vaultForTrader = ICoFiXFactory(factory).getVaultForTrader();
177         if (tradeInfo[0] > 0 && rewardTo != address(0) && vaultForTrader != address(0))
                {
178             ICoFiXVaultForTrader(vaultForTrader).distributeReward(pair, tradeInfo[0],
                    tradeInfo[1], tradeInfo[2], tradeInfo[3], rewardTo);
179         }

181         // refund oracle fee to msg.sender, if any
182         if (oracleFeeChange > 0) TransferHelper.safeTransferETH(msg.sender,
                oracleFeeChange);
183     }
```

Listing 3.9: CoFiXRouter.sol

Starting from the line 167, i.e., `address pair = pairFor(factory, token)`, the entire routine has the assumption that the queried `pair` contains a valid address. Since the `token` parameter may be mistakenly given by the user, this assumption may not always hold. As a result, we suggest to add an explicit check on its validity. This issue is also applicable to a number of other related routines, e.g., `removeLiquidityGetToken()`, `removeLiquidityGetETH()`, `swapExactTokensForTokens()`, `swapExactTokensForETH()`, `swapETHForExactTokens()`, and `swapTokensForExactETH()`.

Recommendation Validate the pair address queried from `pairFor(factory, token)`.

Status The current implementation does not check the pair address for all of these functions. The condition the pair is zero means the trading pair for the target token is not created yet. So the whole transaction would be reverted finally. The team would consider adding the pair validation logic in the next minor update of `CoFiXRouter`.

3.7 Inconsistency Between Documentation and Implementation

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

There are a few misleading comments embedded among lines of solidity code, which bring unnecessary hurdles to understand and/or maintain the software.

A few example comments can be found in line 79 of `CoFiStakingRewards::_rewardPerTokenAndAccrued()` and line 31 of `CoFiXVaultForLP`. Using the `_rewardPerTokenAndAccrued()` routine as an example, the embedded comment indicates that "*50% of accrued to CoFi holders as dividend*." However, the enforcement (line 79) show it is 20% of accrued rewards to CoFi holders as dividend.

```
71      function _rewardPerTokenAndAccrued() internal view returns (uint256, uint256) {
72          if (_totalSupply == 0) {
73              // use the old rewardPerTokenStored, and accrued should be zero here
74              // if not the new accrued amount will never be distributed to anyone
75              return (rewardPerTokenStored, 0);
76          }
77          uint256 _accrued = accrued();
78          uint256 _rewardPerToken = rewardPerTokenStored.add(
79              _accrued.mul(1e18).mul(dividendShare).div(_totalSupply).div(100) // 50%
                    of accrued to CoFi holders as dividend
80              );
81          return (_rewardPerToken, _accrued);
82      }
```

Listing 3.10: CoFiStakingRewards.sol

Also, the white paper indicates "*Mining pool B, the liquidity mining pool: bt is the amount of CoFi tokens generated per block, bt >= 3.6864, start with b0=4 ...*" According to the implementation, the starting amount of CoFi tokens generated per block is `b0=9`.

Recommendation Ensure the consistency between documents (including embedded comments) and implementation.

Status This issue has been confirmed and accordingly fixed in the white paper. Also, it has been confirmed by the team that the `stdMiningRate` is actually $y_t * a_t$. Trader mining pools, CoFiX node pools, and LP pools are rewarded at a ratio of $8 : 1 : 1$. The rewards are distributed further in the `distributeReward()` function.

3.8 Fallback Conversion of ETH to WETH in CoFixPair

- ID: PVE-008
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CoFiXPair`
- Category: Business Logic [8]
- CWE subcategory: CWE-754 [5]

Description

By design, each pair in `CoFixPair` is naturally associated with two tokens, namely, `token0` and `token1`. Since `WETH` is always used as the reference token, current codebase has made an implicit assumption of `token0 = WETH`. Note that the `CoFixPair` contract has a built-in `receive()` routine that accepts `ETH` transfers.

From another perspective, we notice that `CoFixPair` supports two bail-out functions, i.e., `sync()` and `skim()`. Specifically, as detailed in the `Uniswap` white paper, the `sync()` routine behaviors as a recovery mechanism in the case that a token asynchronously deflates the balance of a pair. In this case, trades will receive sub-optimal rates, and if no liquidity provider is willing to rectify the situation, the pair is stuck. Accordingly, `sync()` is designed to set the reserves of the contract to the current balances, providing a somewhat graceful recovery from this situation. The `skim()` routine instead functions as a recovery mechanism in case enough tokens are sent to an pair to overflow the internal `uint112` storage slots for reserves, which could otherwise cause trades to fail. Therefore, it is designed to allow a user to withdraw the difference between the current balance of the pair and $2^{112} - 1$ to the caller, if that difference is greater than 0.

```
318     // force balances to match reserves
319     function skim(address to) external override lock {
320         address _token0 = token0; // gas savings
321         address _token1 = token1; // gas savings
322         _safeTransfer(_token0, to, IERC20(_token0).balanceOf(address(this)).sub(reserve0
                ));
323         _safeTransfer(_token1, to, IERC20(_token1).balanceOf(address(this)).sub(reserve1
                ));
324     }
325
326     // force reserves to match balances
327     function sync() external override lock {
328         _update(IERC20(token0).balanceOf(address(this)), IERC20(token1).balanceOf(
                address(this)));
329     }
```

Listing 3.11: CoFiPair.sol

With the native support of WETH as token0, these two bail-out functions cannot remedy the situation if a token is sent as ETH, not WETH. Therefore, we suggest to add necessary conversion from ETH to WETH and make the transferred ETH available for sync() and skim().

Recommendation Extend sync() and skim() to natively support ETH payments.

Status The current prototype does not accept ETH as input for trading and the related contract can be kept as simple as possible. However, if this conversion is added, it should be even better when someone unintentionally sends ETH in this contract. The team would consider adding this conversion in the future.

3.9 Potential Lockup of User Funds in Staking

- ID: PVE-009
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: CoFiXStakingRewards
- Category: Business Logic [8]
- CWE subcategory: CWE-754 [5]

Description

The CoFiX protocol has developed unique incentive mechanisms to attract early users and encourage wide adoption. In this section, we focus on a modifier, i.e., updateReward(), which is responsible for calculating the reward rate for each staked token and is always invoked up-front for almost every public function in CoFiXStakingRewards to update and use the latest reward rate.

In the following, we show the updateReward() implementation. Note that this modifier is located in the critical execution path of every single staking and unstaking operation. If there is any specific operation within this modifier causes to revert the transaction execution, it will naturally fail the staking and unstaking operation. In other words, the staked funds may be locked in the pool.

```
196     modifier updateReward(address account) virtual {
197         // rewardPerTokenStored = rewardPerToken();
198         // uint256 newAccrued = accrued();
199         (uint256 newRewardPerToken, uint256 newAccrued) = _rewardPerTokenAndAccrued();
200         rewardPerTokenStored = newRewardPerToken;
201         if (newAccrued > 0) {
202             // distributeReward could fail if CoFiXVaultForLP is not minter of CoFi
                    anymore
203             // Should set reward rate to zero first, and then do a settlement of pool
                    reward by call getReward
204             ICoFiXVaultForLP(rewardsVault()).distributeReward(address(this), newAccrued)
                    ;
205         }
206         lastUpdateBlock = lastBlockRewardApplicable();
```

```
207         if (account != address(0)) {
208             rewards[account] = earned(account);
209             userRewardPerTokenPaid[account] = rewardPerTokenStored;
210         }
211         _;
212     }
```

Listing 3.12: CoFiXStakingRewards.sol

With that, we further notice that the internal call of `distributeReward()` has a few requirements (lines 150, 154, 159, and 165) that in unlikely situations may lead to revert. The unlikely situations include that the pool is in an `INVALID` state or the pool has been revoked from being part of `minters`.

```
148     function distributeReward(address to, uint256 amount) external override nonReentrant
            {
149         POOL_STATE poolState = poolInfo[msg.sender].state;
150         require(poolState != POOL_STATE.INVALID, "CVaultForLP: only pool valid");
151         if (poolState == POOL_STATE.DISABLED) {
152             return; // make sure tx would revert because user still want to withdraw and
                    getReward
153         }
154         require(to != address(0), "CVaultForTrader: invalid to");
155         // if poolState is enabled, then go on. caution: be careful when adding new pool
156         address vaultForTrader = ICoFiXFactory(factory).getVaultForTrader();
157         if (vaultForTrader != address(0)) { // if equal, means vaultForTrader is not set
                yet
158             address pair = ICoFiXStakingRewards(msg.sender).stakingToken();
159             require(pair != address(0), "CVaultForTrader: invalid pair");
160             uint256 pending = ICoFiXVaultForTrader(vaultForTrader).getPendingRewardOfLP(
                    pair);
161             if (pending > 0) {
162                 ICoFiXVaultForTrader(vaultForTrader).clearPendingRewardOfLP(pair);
163             }
164         }
165         ICoFiToken(cofiToken).mint(to, amount); // allows zero
166     }
```

Listing 3.13: CoFiXVaultForLP.sol

Recommendation Avoid locking up users' assets in the staking pool by ensuring every single operation in `updateReward()` will not revert. An example improvement is shown below.

```
196     modifier updateReward(address account) virtual {
197         // rewardPerTokenStored = rewardPerToken();
198         // uint256 newAccrued = accrued();
199         (uint256 newRewardPerToken, uint256 newAccrued) = _rewardPerTokenAndAccrued();
200         rewardPerTokenStored = newRewardPerToken;
201         if (newAccrued > 0 && ICoFiToken(rewardsToken).minters(address(this))) {
202             // distributeReward could fail if CoFiXVaultForLP is not minter of CoFi
                    anymore
203             // Should set reward rate to zero first, and then do a settlement of pool
                    reward by call getReward
```

```
204             ICoFiXVaultForLP(rewardsVault()).distributeReward(address(this), newAccrued)
                    ;
205         }
206         lastUpdateBlock = lastBlockRewardApplicable();
207         if (account != address(0)) {
208             rewards[account] = earned(account);
209             userRewardPerTokenPaid[account] = rewardPerTokenStored;
210         }
211         _;
212     }
```

Listing 3.14: CoFiXStakingRewards.sol

Status This issue has been confirmed. The team will consider implement the suggested improvement in the next major update. In the meantime, the `emergencyWithdraw()` function is present by design to ensure users can safely remove their assets.

3.10 Requirement of Oracle For Pair Creation

- ID: PVE-010
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CoFiXPair`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

As mentioned in Section 3.1, `CoFiXPair` acts as a trustless intermediary between liquidity providers and trading users. When the pool does not exist, the first liquidity provider's `addLiquidity()` operation will trigger the creation of the pool (via the `createPair()` function). Previously we have examined a possibility in exploiting a proxy-based approach for gas-efficient pair deployment. In this section, we further suggest the avoidance of creating a pair contract when related conditions are not met yet.

One specific precondition is the dependence of a price oracle for the paired token `token1`. CoFiX is unique in taking advantage of `NEST` oracle for price feeds and risk measurement (and mitigation). However, it also imposes an explicit dependence on the availability of price feeds for the supported tokens.

```
49     function createPair(address token) external override returns (address pair) {
50         require(token != address(0), 'CFactory: ZERO_ADDRESS');
51         require(getPair[token] == address(0), 'CFactory: PAIR_EXISTS');
52         bytes memory bytecode = type(CoFiXPair).creationCode;
53         bytes32 salt = keccak256(abi.encodePacked(token));
54         assembly {
55             pair := create2(0, add(bytecode, 32), mload(bytecode), salt)
```

```
56          }
57          require(pair != address(0), "CFactory: Failed on deploy");
58
59          getPair[token] = pair;
60          allPairs.push(pair);
61
62          uint256 pairLen = allPairs.length;
63          string memory _idx = uint2str(pairLen);
64          string memory _name = append(pairNamePrefix, _idx);
65          string memory _symbol = append(pairSymbolPrefix, _idx);
66          ICoFiXPair(pair).initialize(WETH, token, _name, _symbol);
67
68          ICoFiXController(controller).addCaller(pair);
69          emit PairCreated(token, pair, pairLen);
70      }
```

Listing 3.15: CoFiXFactory.sol

With that, the deployment of a new pair should be avoided if the respective oracle feed for the paired token is not ready yet.

Recommendation Avoid deploying a new pair if the paired `token1` is not currently available yet.

Status By design, the current prototype does not care whether the oracle exists or not. If not, the trading and addling liquidity features would not work. However, it has been agreed that it would indeed be better to add the oracle check at the very beginning.

3.11 Removal of Unused Code

- ID: PVE-011
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `CoFiXController`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [2]

Description

CoFiX makes use of a number of reference libraries and contracts, such as `SafeMath`, `ERC20`, and `IWETH`, to facilitate the protocol implementation and organization. For instance, the `CoFiXController` smart contract interacts with at least four different external contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `CoFiXController` contract, the current functionality of `calcVariance()` is not needed. Therefore, the full body of its implementation can be removed.

Similarly, a number of state variables or constant values in the same contract can be accordingly removed.

```
296     // calc Variance, a.k.a. sigma squared
297     function calcVariance(address token) internal returns (
298         int128 _variance,
299         uint256 _T,
300         uint256 _ethAmount,
301         uint256 _erc20Amount,
302         uint256 _blockNum
303     ) // keep these variables to make return values more clear
304     {
305         address oracle = voteFactory.checkAddress("nest.v3.offerPrice");
306         // query raw price list from nest oracle (newest to oldest)
307         uint256[] memory _rawPriceList = INest_3_OfferPrice(oracle).
                updateAndCheckPriceList{value: msg.value}(token, 50);
308         require(_rawPriceList.length == 150, "CoFiXCtrl: bad price len");
309         // calc P a.k.a. price from the raw price data (ethAmount, erc20Amount, blockNum
                )
310         uint256[] memory _prices = new uint256[](50);
311         for (uint256 i = 0; i < 50; i++) {
312             // 0..50 (newest to oldest), so _prices[0] is p49 (latest price), _prices
                    [49] is p0 (base price)
313             _prices[i] = calcPrice(_rawPriceList[i*3], _rawPriceList[i*3+1]);
314         }
315
316         ...
317
318         _T = block.number.sub(_rawPriceList[2]).mul(timespan);
319         return (_variance, _T, _rawPriceList[0], _rawPriceList[1], _rawPriceList[2]);
320     }
```

Listing 3.16: CoFiXController.sol

Recommendation Consider the removal of the unused code/functionality and state variables (including unused constant values).

Status The `calcVariance()` function is currently unused. The team would remove it in the next minor update.

3.12 Trust Issue of Admin Keys Behind CoFiToken

- ID: PVE-012
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `CoFiToken`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In CoFiX, there is a protocol-wide governance token, i.e., `CoFiToken`. This token contract has defined a state variable named `governance` that plays a critical role in governing and regulating its token issuance. Our analysis shows that the governance address is currently configured as `0xf51d8fdf98286e1ea846c79f1526ecc95b93abb8`. A further examination indicates it is a proxy to a multi-sig `GnosisSafe` account.

```
77      function addMinter(address _minter) external onlyGovernance {
78          minters[_minter] = true;
79          emit MinterAdded(_minter);
80      }
81
82      function removeMinter(address _minter) external onlyGovernance {
83          minters[_minter] = false;
84          emit MinterRemoved(_minter);
85      }
86
87      /// @notice mint is used to distribute CoFi token to users, minters are CoFi mining
            pools
88      function mint(address _to, uint256 _amount) external {
89          require(minters[msg.sender], "CoFi: !minter");
90          _mint(_to, _amount);
91          _moveDelegates(address(0), _delegates[_to], _amount);
92      }
```

Listing 3.17: CoFiToken.sol

To elaborate, we show above the sensitive operations that are related to `governance`. Specifically, it has the authority to add a new `minter` or remove an existing `minter`. Any added minter has the privilege to inflate `CoFi` supply.

It is worrisome if the `governance` account is a plain EOA account. The current multi-sig account greatly alleviates this concern. But it is far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered for mitigation.

Recommendation Promptly transfer the `governance` privilege of `CoFiToken` to the intended

governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been confirmed. For the time being, the DAO is still under design.

PeckShield

4 Conclusion

In this audit, we have analyzed the design and implementation of the CoFiX protocol. The system presents a unique offering in current DEX ecosystem with the support of computable risks for both traders and market makers. CoFiX pushes forward the current DEX frontline and presents a valuable contribution to current DeFi ecosystem. The current code base is well structured and neatly 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.

PeckShield

5 Appendix

5.1 Basic Coding Bugs

5.1.1 Constructor Mismatch

- Description: Whether the contract name and its constructor are not identical to each other.
- Result: Not found
- Severity: Critical

5.1.2 Ownership Takeover

- Description: Whether the set owner function is not protected.
- Result: Not found
- Severity: Critical

5.1.3 Redundant Fallback Function

- Description: Whether the contract has a redundant fallback function.
- Result: Not found
- Severity: Critical

5.1.4 Overflows & Underflows

- Description: Whether the contract has general overflow or underflow vulnerabilities [12, 13, 14, 15, 17].
- Result: Not found
- Severity: Critical

5.1.5 Reentrancy

- Description: Reentrancy [18] is an issue when code can call back into your contract and change state, such as withdrawing ETHs.
- Result: Not found
- Severity: Critical

5.1.6 Money-Giving Bug

- Description: Whether the contract returns funds to an arbitrary address.
- Result: Not found
- Severity: High

5.1.7 Blackhole

- Description: Whether the contract locks ETH indefinitely: merely in without out.
- Result: Not found
- Severity: High

5.1.8 Unauthorized Self-Destruct

- Description: Whether the contract can be killed by any arbitrary address.
- Result: Not found
- Severity: Medium

5.1.9 Revert DoS

- Description: Whether the contract is vulnerable to DoS attack because of unexpected `revert`.
- Result: Not found
- Severity: Medium

5.1.10 Unchecked External `Call`

- Description: Whether the contract has any external `call` without checking the return value.
- Result: Not found
- Severity: Medium

5.1.11 Gasless `Send`

- Description: Whether the contract is vulnerable to gasless `send`.
- Result: Not found
- Severity: Medium

5.1.12 `Send` Instead Of `Transfer`

- Description: Whether the contract uses `send` instead of `transfer`.
- Result: Not found
- Severity: Medium

5.1.13 Costly Loop

- Description: Whether the contract has any costly loop which may lead to `Out-Of-Gas` exception.
- Result: Not found
- Severity: Medium

5.1.14 (Unsafe) Use Of Untrusted Libraries

- Description: Whether the contract use any suspicious libraries.
- Result: Not found
- Severity: Medium

5.1.15 (Unsafe) Use Of Predictable Variables

- Description: Whether the contract contains any randomness variable, but its value can be predicated.
- Result: Not found
- Severity: Medium

5.1.16 Transaction Ordering Dependence

- Description: Whether the final state of the contract depends on the order of the transactions.
- Result: Not found
- Severity: Medium

5.1.17 Deprecated Uses

- Description: Whether the contract use the deprecated `tx.origin` to perform the authorization.
- Result: Not found
- Severity: Medium

5.2 Semantic Consistency Checks

- Description: Whether the semantic of the white paper is different from the implementation of the contract.
- Result: Not found
- Severity: Critical

5.3 Additional Recommendations

5.3.1 Avoid Use of Variadic Byte Array

- Description: Use fixed-size byte array is better than that of `byte[]`, as the latter is a waste of space.
- Result: Not found
- Severity: Low

5.3.2 Make Visibility Level Explicit

- Description: Assign explicit visibility specifiers for functions and state variables.
- Result: Not found
- Severity: Low

5.3.3 Make Type Inference Explicit

- Description: Do not use keyword `var` to specify the type, i.e., it asks the compiler to deduce the type, which is not safe especially in a loop.
- Result: Not found
- Severity: Low

5.3.4 Adhere To Function Declaration Strictly

- Description: Solidity compiler (version 0.4.23) enforces strict ABI length checks for return data from `calls()` [1], which may break the the execution if the function implementation does NOT follow its declaration (e.g., no return in implementing `transfer()` of ERC20 tokens).
- Result: Not found
- Severity: Low

PeckShield

References

[1] axic. Enforcing ABI length checks for return data from calls can be breaking. https://github.com/ethereum/solidity/issues/4116.

[2] MITRE. CWE-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[3] MITRE. CWE-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

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

[5] MITRE. CWE-754: Improper Check for Unusual or Exceptional Conditions. https://cwe.mitre.org/data/definitions/754.html.

[6] MITRE. CWE CATEGORY: 7PK - Security Features. https://cwe.mitre.org/data/definitions/254.html.

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

[8] MITRE. CWE CATEGORY: Business Logic Errors. https://cwe.mitre.org/data/definitions/840.html.

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

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

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

[12] PeckShield. ALERT: New batchOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10299). https://www.peckshield.com/2018/04/22/batchOverflow/.

[13] PeckShield. New burnOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-11239). https://www.peckshield.com/2018/05/18/burnOverflow/.

[14] PeckShield. New multiOverflow Bug Identified in Multiple ERC20 Smart Contracts (CVE-2018-10706). https://www.peckshield.com/2018/05/10/multiOverflow/.

[15] PeckShield. New proxyOverflow Bug in Multiple ERC20 Smart Contracts (CVE-2018-10376). https://www.peckshield.com/2018/04/25/proxyOverflow/.

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

[17] PeckShield. Your Tokens Are Mine: A Suspicious Scam Token in A Top Exchange. https://www.peckshield.com/2018/04/28/transferFlaw/.

[18] Solidity. Warnings of Expressions and Control Structures. http://solidity.readthedocs.io/en/develop/control-structures.html.