SMART CONTRACT AUDIT REPORT

for

Copycat Protocol

Prepared By: Yiqun Chen

PeckShield

December 25, 2021

Document Properties

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

Version Info

Version	Date	Author(s)	Description
1.0	December 25, 2021	Xuxian Jiang	Final Release
1.0-rc1	December 22, 2021	Xuxian Jiang	Release Candidate #1

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

Given the opportunity to review the design document and related smart contract source code of the Copycat Finance protocol, we outline in the report 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 version of smart contracts 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 Copycat Finance

Copycat Finance is a decentralized COPY TRADING and COPY FARMING platform on Binance Smart Chain. It is designed to connect a multi-strategies pool of master traders and master farmers worldwide. Social trading and farming revolution incentivize traders to join and boost up their passive income. After the success of launching the platform and joining into Binance Labs Incubation Program, Copycat Finance is attempting to roll out the outstanding deliverables; they are Copy Trading V2, Copy Farming, and Copy Gaming. These mentioned features will again be another part of the DeFi revolution. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Copycat

Item	Description
Name	Copycat Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	December 25, 2021

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

- https://github.com/copycatfinance/copycat_v2.git (88393f1)

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

- https://github.com/copycatfinance/copycat_v2.git (b68c971)

1.2 About PeckShield

PeckShield Inc. [11] 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 the OWASP Risk Rating Methodology [10]:

- 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 Audit Checklist

Category	Checklist Items
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 checklist of 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) [9], 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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

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 Logic	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 implementation of the `Copycat Finance` protocol. 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	3	■ ■ ■
Informational	1	■
Total	6	

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 medium-severity vulnerabilities, 3 low-severity vulnerabilities, and 1 informational recommendation.

Table 2.1: Key Copycat Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	CPC Fee Collection From Unknowing Users	Business Logic	Fixed
PVE-002	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Fixed
PVE-003	Low	Improved sync() Logic For Optimal Liquidity	Business Logic	Fixed
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-006	Informational	Redundant State/Code Removal	Coding Practice	Fixed

Besides 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 CPC Fee Collection From Unknowing Users

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: CopycatLeader
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

At the core of the Copycat protocol is the CopycatLeader contract that allows for non-contract users to add assets into the protocol and get minted with the corresponding pool share. While reviewing the share minting logic, we notice the required LeaderDepositCopycatFee collection is flawed.

To elaborate, we show below the related depositTo() function. It implements a rather straight-forward logic in transferring user assets into the contract and mint the corresponding pool share. It comes to our attention the associated copycat fee for the deposit is collected from the given argument to, instead of msg.sender. As a result, an unknowing user may be charged for the deposit fee.

```
function depositTo(address to, uint256 percentage, IERC20 refToken, uint256
    maxRefAmount) payable public virtual nonReentrant onlyEOA returns(uint256
    totalShare) {
  require(!disabled, "D");

  uint256 refAmount = 0;
  uint256 bnbBefore = address(this).balance;

  ICopycatAdapter[] memory adapters = S.getAdapters(address(this));

  // Collect CPC fee
  uint256 depositCopycatFee = S.getLeaderDepositCopycatFee(address(this));
  if (depositCopycatFee > 0 && msg.sender != address(factory) && to != owner()) {
    S.collectLeaderFee(to, depositCopycatFee);
  }

```

```
    // Transfer tokens
    for (uint i = 0; i < tokens.length; i++) {
      IERC20 token = tokens[i];

      uint256 amount = token.balanceOf(address(this)) * percentage / 1e18;

      if (amount > 0) {
        if (i > 0  msg.value == 0) {
          token.transferFrom(msg.sender, address(this), amount);
        } else {
          WETH.deposit{value: amount}();
        }

        if (token == refToken) {
          refAmount += amount;
        }
      }
    }
    ...
}
```

Listing 3.1: `CopycatLeader::depositTo()`

Recommendation Properly revise the above `depositTo()` routine to collect deposit fee from `msg.sender`, instead of the user-provided argument `to`.

Status The issue has been fixed by this commit: `bc3f771`.

3.2 Implicit Assumption Enforcement In AddLiquidity()

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CopycatLeader`
- Category: Coding Practices [7]
- CWE subcategory: CWE-628 [3]

Description

In the `CopycatLeader` contract, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity via the `UniswapRouterV2::addLiquidity()` routine. To elaborate, we show below the related code snippet.

```
  function addLiquidity(
    IUniswapV2Router02 router,
    address tokenA,
    address tokenB,
```

```
    uint256 amountADesired,
    uint256 amountBDesired,
    uint256 amountAMin,
    uint256 amountBMin
  )
    external
    nonReentrant
    onlyOwner
    returns (
      uint256 amountA,
      uint256 amountB,
      uint256 liquidity
    )
  {
    require(tokensType[tokenA] != 0 && tokensType[tokenB] != 0, "E");

    IERC20(tokenA).approve(address(router), amountADesired);
    IERC20(tokenB).approve(address(router), amountBDesired);

    (amountA, amountB, liquidity) = router.addLiquidity(
      tokenA,
      tokenB,
      amountADesired,
      amountBDesired,
      amountAMin,
      amountBMin,
      address(this),
      block.timestamp
    );

    address pair = IUniswapV2Factory(router.factory()).getPair(tokenA, tokenB);
    require(S.tokenAllowed(pair), "T");
    _addToken(IERC20(pair), 2);
  }
```

Listing 3.2: `CopycatLeader::addLiquidity()`

```
    function _addLiquidity(
        address tokenA,
        address tokenB,
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        // create the pair if it doesn't exist yet
        if (IUniswapV2Factory(factory).getPair(tokenA, tokenB) == address(0)) {
            IUniswapV2Factory(factory).createPair(tokenA, tokenB);
        }
        (uint reserveA, uint reserveB) = UniswapV2Library.getReserves(factory, tokenA,
            tokenB);
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
```

```
        } else {
            uint amountBOptimal = UniswapV2Library.quote(amountADesired, reserveA,
                reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'UniswapV2Router:
                    INSUFFICIENT_B_AMOUNT');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = UniswapV2Library.quote(amountBDesired, reserveB,
                    reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'UniswapV2Router:
                    INSUFFICIENT_A_AMOUNT');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
```

Listing 3.3: `UniswapV2Router02::_addLiquidity()`

It comes to our attention that the `Uniswap V2 Router` has implicit assumptions on the `_addLiquidity()` routine. The above routine takes two amounts: `amountXDesired` and `amountXMin`. The first amount `amountXDesired` determines the desired amount for adding liquidity to the pool and the second amount `amountXMin` determines the minimum amount of used assets. There are two implicit conditions, i.e., `amountADesired >= amountAMin` and `amountBDesired >= amountBMin`. However, if these two conditions are not met, current logic will not trigger reverts because the code above performs asymmetric checks for these amounts. Hence, without stating these assumptions, slippage control for some trades on `Uniswap V2 Router` may not be checked and may not be taken into account at all in certain scenarios.

Recommendation Make the requirement of `amountADesired >= amountAMin` and `amountBDesired >= amountBMin` explicitly in the `addLiquidity()` function.

Status The issue has been fixed by adding the suggested requirement.

3.3 Improved sync() Logic For Optimal Liquidity

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CopycatAutoCompound`
- Category: Business Logic [8]
- CWE subcategory: CWE-837 [4]

Description

As mentioned earlier, the `Copycat` protocol is designed with a number of yield optimization strategies. Accordingly, there is a constant need of swapping one token to another. In the following, we examine related swap routines that are designed to assist the token swapping.

To elaborate, we show below a helper routine named `sync()` from the `CopycatAutoCompound` contract. The routine is used to convert half assets to `token0` and another half to `token1` so that the liquidity can be accordingly added. It comes to our attention that the current approach converts half assets to `token0` and then sends the another half for`token1` may result in slight token waste. In other words, the current conversion approach is not optimal.

```
function sync(uint256 minToken) public nonReentrant {
  require(msg.sender == address(masterchefAdapter));

  address token = address(masterchefAdapter.token());
  address reward = address(masterchefAdapter.reward());

  if (token != reward) {
    if (compoundType == 1) {
      sellToken(reward, IERC20(reward).balanceOf(address(this)));
      buyToken(token, WETH.balanceOf(address(this)));
    } else {
      sellToken(reward, IERC20(reward).balanceOf(address(this)));

      IUniswapV2Router02 router = IUniswapV2Router02(S.factory2router(IUniswapV2Pair(
          token).factory()));
      address token0 = IUniswapV2Pair(token).token0();
      address token1 = IUniswapV2Pair(token).token1();

      buyToken(token0, WETH.balanceOf(address(this)) / 2);
      buyToken(token1, WETH.balanceOf(address(this)));

      {
        uint256 amount0 = IERC20(token0).balanceOf(address(this));
        uint256 amount1 = IERC20(token1).balanceOf(address(this));

        IERC20(token0).approve(address(router), amount0);
        IERC20(token1).approve(address(router), amount1);

        router.addLiquidity(
          token0,
          token1,
          amount0,
          amount1,
          0,
          0,
          address(this),
          block.timestamp
        );
      }
```

```
        }
    }
```

Listing 3.4: `CopycatAutoCompound::sync()`

Moreover, the above conversion does not specify any slippage restriction, which may be abused in a possible sandwich or MEV attack for reduced return. Affected routines also include `swapTokensForExactTokens()` and `swapExactTokensForTokensSupportingFeeOnTransferTokens()`.

Recommendation Perform an optimal allocation of assets between two tokens for matched liquidity addition. Also add necessary slippage control to avoid unnecessary loss of swaps.

Status This issue has been fixed in the following commit: `cf21138`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `ReservePool`, `CakeMiner`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

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 the following, we examine the `transfer()` routine and related 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 3.5: 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. Similarly, there is a safe version of `approve()`/`transferFrom()` as well, i.e., `safeApprove()`/`safeTransferFrom()`.

In the following, we show the `adminRecoverToken()` routine in the `CopycatDepositer` contract. If the `USDT` token is supported as `token`, the unsafe version of `token.transfer(to, amount)` (line 242) may revert as there is no return value in the `USDT` token contract's `transfer()`/`transferFrom()` implementation (but the `IERC20` interface expects a return value)!

```
241 function adminRecoverToken(IERC20 token, address to, uint256 amount) external
        onlyOwner {
242   token.transfer(to, amount);
243   emit AdminRecoverToken(msg.sender, address(token), to, amount);
244 }
```

Listing 3.6: `CopycatDepositer::adminRecoverToken()`

Note this issue is also applicable to other routines, including `CopycatLeader::resetAllowance()`/`pluginRequestAllowance()`. For the `safeApprove()` support, there is a need to approve twice: the first time resets the allowance to zero and the second time approves the intended amount.

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()`/`transfer()`/`transferFrom()`.

Status This issue has been fixed in the following commit: `111500f`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [1]

Description

In the Copycat protocol, there is a privileged owner account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting, reward distribution, and contract adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and their related privileged accesses in current contracts.

```
event ToAdapter(address indexed caller, address indexed adapter, uint256 amount);
function toAdapter(ICopycatAdapter adapter, uint256 amount) public nonReentrant
    onlyOwner {
  require(S.pluginsEnMap(address(this), adapter), "F");
  S.adaptersToken(address(this), adapter).transfer(address(adapter), amount);
  adapter.sync();
  emit ToAdapter(msg.sender, address(adapter), amount);
}

event ToLeader(address indexed caller, address indexed adapter, uint256 amount);
function toLeader(ICopycatAdapter adapter, uint256 amount) public nonReentrant
    onlyOwner {
  require(S.pluginsEnMap(address(this), adapter), "F");
  adapter.withdrawTo(address(this), amount);
  emit ToLeader(msg.sender, address(adapter), amount);
}
```

Listing 3.7: Example Setters in the CopycatLeader Contract

In addition, we notice the owner account that is able to adjust various protocol-wide risk parameters. Apparently, if the privileged owner account is a plain EOA account, this may be worrisome and pose counter-party risk to the protocol users. Note that a multi-sig account could greatly alleviate this concern, though it is still 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 as mitigation.

Moreover, it should be noted that current contracts have the support of being deployed behind a proxy. And there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, 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 with the team. For the time being, the team has confirmed the plan to use a timelock to manage these privileged functions, not a plain EOA account.

3.6 Redundant State/Code Removal

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [2]

Description

The `Copycat` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeBEP20`, `SafeMath`, and `Address`, to facilitate its code implementation and organization. For example, the `CopycatLeader` smart contract has so far imported at least five reference 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 `swapExactTokensForTokensSupportingFeeOnTransferTokens` function from the `CopycatLeader` contract, the current slippage control on `require(IERC20(tokenOut).balanceOf(address(this))- beforeBalance >= mainOut * 97 / 100)` (line 510) is ineffective and can be simply removed. The same issue is also applicable to the `swapTokensForExactTokens()` in the same contract.

```
482  function swapExactTokensForTokensSupportingFeeOnTransferTokens(
483    IUniswapV2Router02 router,
484    uint256 amountIn,
485    uint256 amountOutMin,
486    address[] calldata path
487  ) external nonReentrant onlyOwner {
488    address tokenIn = path[0];
489    address tokenOut = path[path.length - 1];
490
491    require(tokensType[tokenIn] != 0 && tokensType[tokenOut] != 0, "E");
492
493    uint256 fee = S.FEE_LIST(5) * amountIn / 1e18;
494    IERC20(tokenIn).transfer(S.feeAddress(), fee);
495    amountIn -= fee;
```

```

    uint256 mainOut = mainPathSwapOut(router, amountIn, tokenIn, tokenOut);
    uint256 beforeBalance = IERC20(tokenOut).balanceOf(address(this));

    IERC20(tokenIn).approve(address(router), amountIn);

    router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
      amountIn,
      amountOutMin,
      path,
      address(this),
      block.timestamp
    );

    require(IERC20(tokenOut).balanceOf(address(this)) - beforeBalance >= mainOut * 97 /
        100, "I");
  }
```

Listing 3.8: `CopycatLeader::swapExactTokensForTokensSupportingFeeOnTransferTokens`

Recommendation Consider the removal of the redundant code with a simplified, consistent implementation.

Status This issue has been fixed.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Copycat Finance` protocol, which is a decentralized `COPY TRADING` and `COPY FARMING` platform on `Binance Smart Chain`. It is designed to connect a multi-strategies pool of master traders and master farmers worldwide. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Moreover, we need to emphasize that `Solidity-based` 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-287: Improper Authentication. https://cwe.mitre.org/data/definitions/287.html.

[2] MITRE. CWE-563: Assignment to Variable without Use. https://cwe.mitre.org/data/definitions/563.html.

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

[4] MITRE. CWE-837: Improper Enforcement of a Single, Unique Action. https://cwe.mitre.org/data/definitions/837.html.

[5] MITRE. CWE-841: Improper Enforcement of Behavioral Workflow. https://cwe.mitre.org/data/definitions/841.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 VIEW: Development Concepts. https://cwe.mitre.org/data/definitions/699.html.

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

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