SMART CONTRACT AUDIT REPORT

for

Cadabra

Prepared By: Xiaomi Huang

PeckShield

November 22, 2023

Document Properties

Client	Cadabra
Title	Smart Contract Audit Report
Target	Cadabra
Version	1.0
Author	Xuxian Jiang
Auditors	Colin Zhong, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	November 22, 2023	Xuxian Jiang	Final Release
1.0-rc2	November 12, 2023	Xuxian Jiang	Release Candidate #2
1.0-rc1	October 29, 2023	Xuxian Jiang	Release Candidate #1

Contact

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

Name	Xiaomi Huang
Phone	+86 183 5897 7782
Email	contact@peckshield.com

Contents

1 Introduction

Given the opportunity to review the design document and related source code of the `Cadabra` 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 Cadabra

`Cadabra` is the gateway to the optimal passive yield in `DeFi` by aggregating together potential yield sources, assessing their risk score, and combining them into simple, yet effective automated strategies. The strategies aim to guarantee continuity of passive yield: for a given asset (or a group of assets) a strategy will find the most profitable protocols now and in the future (so that users don't have to worry about relocating the liquidity anymore). The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The Cadabra

Item	Description
Name	Cadabra
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	November 22, 2023

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

- https://github.com/cadabra-finance/cadabra-contracts.git (fd3e8fb)

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

- https://github.com/cadabra-finance/cadabra-contracts.git (64b02f7)

1.2 About PeckShield

PeckShield Inc. [12] 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

Likelihood

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.

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

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

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

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

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

2 | Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `Cadabra` 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 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	1	■
Informational	0	
Total	5	

We have so far identified a list of potential issues. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

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

Table 2.1: Key Cadabra Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Medium	Incorrect Redeem Slippage Control Enforcement in Router	Business Logic	Resolved
PVE-002	High	Incorrect Value Calculation in VelodromPoolAdapter_qStablePair	Business Logic	Resolved
PVE-003	High	Forced Investment Risk in BalancerUpgradeable and VelodromPoolAdapter	Time and State	Resolved
PVE-004	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Mitigated
PVE-005	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that 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 need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Incorrect Redeem Slippage Control Enforcement in Router

- ID: PVE-001
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Router`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

To facilitate the user interaction, the `Cadabra` protocol provides a `Router` contract that defines easy-to-use functions, i.e., `invest()` and `redeem()`. While reviewing the key redeem logic, we notice its current implementation is flawed.

In the following, we show the implementation of the affected `redeem()` routine. It has an argument `minAmounts` to impose necessary minimum received amount on the receiver side after the redemption. However, it comes to our attention that the balance is measured on `msg.sender` (lines 92 and 100), not the given `receiver`.

```
77     function redeem(
78         address balancer,
79         uint shares,
80         IAdapter targetAdapter,
81         address receiver,
82         TokenAmount[] memory minAmounts,
83         uint32 deadline
84     ) external override returns (address[] memory tokens, uint[] memory amounts)
85     {
86         if (deadline < block.timestamp) {
87             revert Expired(deadline);
88         }
89         uint256[] memory balancesBefore = new uint256[](minAmounts.length);
90         for (uint i = 0; i < minAmounts.length; i++) {
91             TokenAmount memory ta = minAmounts[i];
92             balancesBefore[i] = IERC20(ta.token).balanceOf(msg.sender);
```

```
        }

        SafeERC20.safeTransferFrom(IERC20(balancer), msg.sender, address(this), shares);
        (tokens, amounts) = IBalancer(balancer).redeem(shares, targetAdapter, receiver);

        for (uint i = 0; i < minAmounts.length; i++) {
            TokenAmount memory ta = minAmounts[i];
            uint balanceAfter = IERC20(ta.token).balanceOf(msg.sender);
            uint diff = balanceAfter - balancesBefore[i];
            if (diff < ta.amount) {
                revert InsufficientTokenRedeemed(ta.token, diff, ta.amount);
            }
        }
    }
```

Listing 3.1: `Router::redeem()`

Recommendation Revise the above `redeem()` routine to properly measure the balance difference so that we can enforce the minimum received amount.

Status This issue has been fixed in the following commit: `b664930`.

3.2 Incorrect Value Calculation in VelodromPoolAdapter_qStablePair

- ID: PVE-002
- Severity: High
- Likelihood: High
- Impact: High
- Target: `VelodromPoolAdapter\_qStablePair`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `Cadabra` protocol provides a specific adapter to interact with the `Velodrome` pools. In the process of analyzing the logic to calculate the managed assets value, we notice the current approach to calculate the asset value should be revisited.

In the following, we show the implementation of the related `_values()` routine. This routine is designed to measure the pool value under investment. The measurement relies on the use of `_convertibleToken`. For simplicity, we illustrate with one execution path, i.e., `IS_TOKEN1_QUOTE_IN_BASE_POOL` and `IS_TOKEN1_QUOTE_IN_QUOTE_POOL` are both `true`. With that, while the first step (line 79) is properly executed to calculate the initial `price0`, the next step examines the `QUOTE_POOL` to compute the final `price0 = VelodromeUtils.amount1(price0, BASE1, QUOTE_BASE, qr0, qr1, IS_QUOTE_POOL_STABLE)* QUOTE_PRICE_FACTOR`, instead of current `price0 = VelodromeUtils.amount1(price0, BASE0, QUOTE_BASE,` qr0, qr1, IS_QUOTE_POOL_STABLE)* QUOTE_PRICE_FACTOR (line 82). The reason is the common _convertibleToken in QUOTE_POOL becomes token0, hence using BASE1 as base0 and QUOTE_BASE as base1. The same issue also affects other execution paths.

```
70    function _values(uint256 _amount0, uint256 _amount1) internal view override returns
          (uint256 _value0, uint256 _value1) {
71        (uint r0, uint r1) = VelodromeUtils.reserves(address(POOL));
72        (uint qr0, uint qr1) = VelodromeUtils.reserves(address(QUOTE_POOL));
73
74        uint price0;
75        uint price1;
76
77        if (IS_TOKEN1_QUOTE_IN_BASE_POOL) {
78            // TOKEN0(POOL) -> TOKEN1(POOL)
79            price0 = VelodromeUtils.amount1(BASE0, BASE0, BASE1, r0, r1,
                  IS_BASE_POOL_STABLE);
80            if (IS_TOKEN1_QUOTE_IN_QUOTE_POOL) {
81                // TOKEN1(POOL) -> TOKEN1(QUOTE_POOL)
82                price0 = VelodromeUtils.amount1(price0, BASE0, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
83                price1 = VelodromeUtils.amount1(BASE1, BASE0, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
84            } else {
85                // TOKEN1(POOL) -> TOKEN0(QUOTE_POOL)
86                price0 = VelodromeUtils.amount0(price0, BASE1, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
87                price1 = VelodromeUtils.amount0(BASE1, BASE1, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
88            }
89        } else {
90            // TOKEN1(POOL) -> TOKEN0(POOL)
91            price1 = VelodromeUtils.amount0(BASE1, BASE0, BASE1, r0, r1,
                  IS_BASE_POOL_STABLE);
92            if (IS_TOKEN1_QUOTE_IN_QUOTE_POOL) {
93                // TOKEN0(POOL) -> TOKEN1(QUOTE_POOL)
94                price0 = VelodromeUtils.amount1(BASE0, BASE0, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
95                price1 = VelodromeUtils.amount1(price1, BASE0, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
96            } else {
97                // TOKEN0(POOL) -> TOKEN0(QUOTE_POOL)
98                price0 = VelodromeUtils.amount0(BASE1, BASE1, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
99                price1 = VelodromeUtils.amount0(price1, BASE1, QUOTE_BASE, qr0, qr1,
                      IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
100           }
101       }
102
103       _value0 = _amount0 * price0 / BASE0;
104       _value1 = _amount1 * price1 / BASE1;
```

```
    }
```

Listing 3.2: `VelodromePoolAdapter_qStablePair::_values()`

Recommendation Revise the above routine as follows.

```
    function _values(uint256 _amount0, uint256 _amount1) internal view override returns (uint256 _value0, uint256 _value1) {
        (uint r0, uint r1) = VelodromeUtils.reserves(address(POOL));
        (uint qr0, uint qr1) = VelodromeUtils.reserves(address(QUOTE_POOL));

        uint price0;
        uint price1;

        if (IS_TOKEN1_QUOTE_IN_BASE_POOL) {
            // TOKEN0(POOL) -> TOKEN1(POOL)
            price0 = VelodromeUtils.amount1(BASE0, BASE0, BASE1, r0, r1, IS_BASE_POOL_STABLE);
            if (IS_TOKEN1_QUOTE_IN_QUOTE_POOL) {
                // TOKEN1(POOL) -> TOKEN1(QUOTE_POOL)
                price0 = VelodromeUtils.amount1(price0, BASE1, QUOTE_BASE, qr0, qr1, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
                price1 = VelodromeUtils.amount1(BASE1, BASE1, QUOTE_BASE, qr0, qr1, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
            } else {
                // TOKEN1(POOL) -> TOKEN0(QUOTE_POOL)
                price0 = VelodromeUtils.amount0(price0, BASE1, QUOTE_BASE, qr1, qr0, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
                price1 = VelodromeUtils.amount0(BASE1, BASE1, QUOTE_BASE, qr1, qr0, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
            }
        } else {
            // TOKEN1(POOL) -> TOKEN0(POOL)
            price1 = VelodromeUtils.amount0(BASE1, BASE0, BASE1, r0, r1, IS_BASE_POOL_STABLE);
            if (IS_TOKEN1_QUOTE_IN_QUOTE_POOL) {
                // TOKEN0(POOL) -> TOKEN1(QUOTE_POOL)
                price0 = VelodromeUtils.amount1(BASE0, BASE0, QUOTE_BASE, qr0, qr1, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
                price1 = VelodromeUtils.amount1(price1, BASE0, QUOTE_BASE, qr0, qr1, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
            } else {
                // TOKEN0(POOL) -> TOKEN0(QUOTE_POOL)
                price0 = VelodromeUtils.amount0(BASE1, BASE0, QUOTE_BASE, qr1, qr0, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
                price1 = VelodromeUtils.amount0(price1, BASE0, QUOTE_BASE, qr1, qr0, IS_QUOTE_POOL_STABLE) * QUOTE_PRICE_FACTOR;
            }
        }

        _value0 = _amount0 * price0 / BASE0;
        _value1 = _amount1 * price1 / BASE1;
```

```
    }
```

Listing 3.3: Revised `VelodromePoolAdapter_qStablePair::_values()`

Status This issue has been fixed in the following commit: `447e3ef`.

3.3 Forced Investment Risk in BalancerUpgradeable and VelodromPoolAdapter

- ID: PVE-003
- Severity: High
- Likelihood: Medium
- Impact: High
- Target: `Multiple Contracts`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `Cadabra` protocol is a yield-farming protocol with a number of strategies that aim to guarantee continuity of passive yield. While examining current investment logic, we notice a potential force investment risk that has been exploited in earlier hacks, e.g., `yDAI` [13] and `BT.Finance` [1]. To elaborate, we show below the related `BaseAdapter::_investInternal()` routine.

Specifically, the `BalancerUpgradeable` contract is designed and implemented to invest user funds, harvest growing yields, and return any gains, if any, to the users. In addition, the investment logic interacts with underlying strategies via associated `adapters`. To elaborate, we show below the implementation of the affected `VelodromePoolAdapter::_invest()` routine.

```
    function _investInternal(address dustReceiver) internal returns (uint256 valueBefore
        , uint256 valueAfter) {
        (valueBefore,) = value();
        _invest();
        (valueAfter,) = value();
        _returnDust(dustReceiver);
    }
```

Listing 3.4: `BaseAdapter::_investInternal()`

```
    function _invest() internal override {
        uint deposit0;
        uint deposit1;
        {
            // Because pair requires a certain proportion of the tokens for the deposit
                we can't just deposit all
            // the tokens we have on our balance. We need to deposit them in accordance
                to the pair's ratio
```

```
            uint balance0 = TOKEN0.balanceOf(address(this));
            uint balance1 = TOKEN1.balanceOf(address(this));
            (uint reserve0, uint reserve1,) = POOL.getReserves();
            if(reserve0 == 0  reserve1 == 0){
                revert ZeroReserveBalance(reserve0,reserve1);
            }

            deposit0 = balance0;
            deposit1 = deposit0 * reserve1 / reserve0;

            if (deposit1 > balance1) {
                deposit1 = balance1;
                deposit0 = deposit1 * reserve0 / reserve1;
            }
        }

        TOKEN0.safeTransfer(address(POOL), deposit0);
        TOKEN1.safeTransfer(address(POOL), deposit1);
        POOL.mint(address(this));
        GAUGE.deposit(POOL.balanceOf(address(this)));
    }
```

Listing 3.5: `VelodromePoolAdapter::_invest()`

It comes to our attention that the above investment logic does not perform any health check: it does not have the stability check on the liquidity pool into which the user funds will be added. In other words, if the configured strategy blindly invests the deposited funds into an imbalanced `Velodrome` pool, the strategy will not result in a profitable investment. In fact, earlier incidents (`yDAI` and `BT.Finance` hacks [1, 13]) have prompted the need of a guarded call before kicking off the actual investment. For the very same reason, we argue for the guarded stability check associated with every single `_invest()` call.

In the meantime, it is important to highlight that the current approach to evaluate the total value managed by the protocol is not reliable. Specifically, it suffers from a sandwich-based attack in arbitrarily inflate or deflate the value at will.

Recommendation Ensure the target liquidity pool is stable before the user funds can be added into as liquidity.

Status This issue has been fixed by the following commit: `e434f48`.

3.4 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

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

In particular, we use the popular stablecoin, i.e., `USDT`, as our example. We show the related code snippet below. On its entry of `approve()`, there is a requirement, i.e., `require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)))`. This specific requirement essentially indicates the need of reducing the allowance to 0 first (by calling `approve(_spender, 0)`) if it is not, and then calling a second one to set the proper allowance. This requirement is in place to mitigate the known `approve()/transferFrom()` race condition (https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729).

```
    /**
    * @dev Approve the passed address to spend the specified amount of tokens on behalf
        of msg.sender.
    * @param _spender The address which will spend the funds.
    * @param _value The amount of tokens to be spent.
    */
    function approve(address _spender, uint _value) public onlyPayloadSize(2 * 32) {

        // To change the approve amount you first have to reduce the addresses`
        //  allowance to zero by calling `approve(_spender, 0)` if it is not
        //  already 0 to mitigate the race condition described here:
        //  https://github.com/ethereum/EIPs/issues/20#issuecomment-263524729
        require(!((_value != 0) && (allowed[msg.sender][_spender] != 0)));

        allowed[msg.sender][_spender] = _value;
        Approval(msg.sender, _spender, _value);
    }
```

Listing 3.6: USDT Token **Contract**

Because of that, a normal call to `approve()` is suggested to use the safe version, i.e., `safeApprove()`, 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 `transfer()` as well, i.e., `safeTransfer()`.

```
/**
 * @dev Deprecated. This function has issues similar to the ones found in
 * {IERC20-approve}, and its usage is discouraged.
 *
 * Whenever possible, use {safeIncreaseAllowance} and
 * {safeDecreaseAllowance} instead.
 */
function safeApprove(
    IERC20 token,
    address spender,
    uint256 value
) internal {
    // safeApprove should only be called when setting an initial allowance,
    // or when resetting it to zero. To increase and decrease it, use
    // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'
    require(
        (value == 0)  (token.allowance(address(this), spender) == 0),
        "SafeERC20: approve from non-zero to non-zero allowance"
    );
    _callOptionalReturn(token, abi.encodeWithSelector(token.approve.selector,
        spender, value));
}
```

Listing 3.7: `SafeERC20::safeApprove()`

In current implementation, if we examine the `SwapExecutor::executeSwaps()` routine that is designed to execute an intended swap. To accommodate the specific idiosyncrasy, there is a need to use `safeApprove()`, instead of `approve()` (line 22).

```
19  function executeSwaps(IBalancer.SwapInfo[] calldata swaps) public {
20      for (uint i = 0; i < swaps.length; i++) {
21          IBalancer.SwapInfo calldata swap = swaps[i];
22          IERC20(swap.token).approve(swap.callee, swap.amount);
23          Address.functionCall(swap.callee, swap.data);
24      }
25  }
```

Listing 3.8: `SwapExecutor::executeSwaps()`

Note the `defaultSwap()` routine in the same contract can be similarly improved.

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

Status This issue has been partially fixed in the following commit: `ce60ede`.

3.5 Trust Issue of Admin Keys

- ID: PVE-005
- Severity: Medium
- Likelihood: Medium
- Impact: High
- Target: Multiple Contracts
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In Cadabra, there is a privileged administrative account (with the DEFAULT_ADMIN_ROLE role). The administrative account plays a critical role in governing and regulating the protocol-wide operations. Our analysis shows that this privileged account needs to be scrutinized. In the following, we use the BalancerUpgradeable contract as an example and show the representative functions potentially affected by the privileges of the administrative account.

```
    function setFeeReceiver(address feeReceiver_) external override onlyRole(
        DEFAULT_ADMIN_ROLE) {
        emit FeeReceiverChanged($feeReceiver, feeReceiver_);
        $feeReceiver = feeReceiver_;
    }

    function addAdapter(address adapterAddress) external override onlyRole(
        ADD_ADAPTER_ROLE) returns (bool isAdded) {
        IAdapter adapter = IAdapter(adapterAddress);
        (uint v, uint a) = adapter.value();
        if (v != 0  a != 0) {
            revert AdapterNotEmpty(adapterAddress);
        }
        isAdded = $adapters.add(adapterAddress);
        if (isAdded) {
            emit AdapterAdded(adapterAddress);
        }
    }

    function removeAdapter(address adapterAddress) external override onlyRole(
        REMOVE_ADAPTER_ROLE) returns (bool) {
        if ($adapters.contains(adapterAddress)) {
            IAdapter adapter = IAdapter(adapterAddress);
            (uint v, uint a) = adapter.value();
            if (v != 0  a != 0) {
                revert AdapterNotEmpty(adapterAddress);
            }
            _deactivateAdapter(adapterAddress);
            emit AdapterRemoved(adapterAddress);
            return $adapters.remove(adapterAddress);
        }
        return false;
    }
```

```

    function activateAdapter(address adapterAddress) external override onlyRole(
        ACTIVATE_ADAPTER_ROLE) returns (bool) {
        uint l = $adapters.length();

        for (uint i=0; i < l; i++) {
            if ($adapters.at(i) == adapterAddress) {
                $isActiveAdapter[adapterAddress] = true;
                emit AdapterActivityChanged(adapterAddress, true);
                return true;
            }
        }

        return false;
    }

    function deactivateAdapter(address adapterAddress) external override onlyRole(
        DEACTIVATE_ADAPTER_ROLE) {
        _deactivateAdapter(adapterAddress);
    }
```

Listing 3.9: Example Privileged Operations in `BalancerUpgradeable`

We understand the need of the privileged functions for contract maintenance, but at the same time the extra power to the administrative account may also be a counter-party risk to the protocol users. It would be worrisome if the privileged administrative account is a plain EOA account. 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, the protocol makes use of the `UUPSUpgradeable` proxy contract to allow for future upgrades. The upgrade is privileged operation, which also falls in this trust issue on the admin key.

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changes 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 mitigated with the plan to transfer the privileged account to a `multi-sig` account.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `Cadabra` protocol, which is the gateway to the optimal passive yield in `DeFi` by aggregating together potential yield sources, assessing their risk score, and combining them into simple, yet effective automated strategies. The strategies aim to guarantee continuity of passive yield: for a given asset (or a group of assets) a strategy will find the most profitable protocols now and in the future (so that users don't have to worry about relocating the liquidity anymore). The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

Meanwhile, 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] BT Finance. BT.Finance Exploit Analysis Report. https://btfinance.medium.com/bt-finance-exploit-analysis-report-a0843cb03b28.

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

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

[4] MITRE. CWE-682: Incorrect Calculation. https://cwe.mitre.org/data/definitions/682.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 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. PeckShield Inc. https://www.peckshield.com.

[13] PeckShield. The yDAI Incident Analysis: Forced Investment. https://peckshield.medium.com/the-ydai-incident-analysis-forced-investment-2b8ac6058eb5.