SMART CONTRACT AUDIT REPORT

for

CBI/JUKU

Prepared By: Xiaomi Huang

PeckShield

February 18, 2023

Document Properties

Client	JUKU
Title	Smart Contract Audit Report
Target	JUKU
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Patrick Lou, Xuxian Jiang
Reviewed by	Patrick Lou
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 18, 2023	Xuxian Jiang	Final Release
1.0-rc1	February 8, 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 smart contract source code of the JUKU platform, 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 JUKU

The JUKU platform is the Dapp allowing users to use a custodial wallet for the investments/withdraw of the CBI tokens into/out the various bundles/liquidity pools. It has its native token, i.e., CBI. Users can invest their CBI tokens into staking functionality that allows them to generate and receive rewards in JUKU tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of JUKU

Item	Description
Name	JUKU
Type	Solidity Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 18, 2023

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

- https://github.com/cbiglobal/cbi-smart-contracts.git (0a6cb36)

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

- https://github.com/blaize-tech/cbi-smart-contracts.git (311f391)

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 the 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 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

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

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. 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 JUKU platform. 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	1	
Medium	3	
Low	3	
Informational	0	
Total	7	

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 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 3 low-severity vulnerabilities.

Table 2.1: Key JUKU Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect Spend Allowance Management in CBI/JUKU	Business Logic	Resolved
PVE-002	Medium	Potential Sandwich/MEV Attacks For Reduced Returns	Time And State	Resolved
PVE-003	Medium	Improper Limit Enforcement in CBI_-Treasury	Business Logic	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Low	Improved Logic in updateLowerAdmin()	Coding Practices	Resolved
PVE-006	Low	Revisited Quorum Enforcement in MultiSigTreasury	Business Logic	Resolved
PVE-007	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Resolved

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 Incorrect Spend Allowance Management in CBI/JUKU

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

Description

The `JUKU` platform has two standard ERC20-compliant tokens – `CBI` and `JUKU`. Both tokens may charge a commission for each transfer. While examining the current commission-related logic, we notice the related spend allowance need to be adjusted.

To elaborate, we show below the implementation of a standard ERC20 function, i.e., `transferFrom()`. As the name indicates, this function transfers the tokens from an owner's account to the receiver account, but only if the transaction initiator has sufficient allowance that has been previously approved by the owner to the transaction initiator. However, it comes to our attention that the spend allowance is adjusted based on the actual amount received by the receiver account (line 136)! The adjustment in fact needs to deduct the sent amount, which includes the `feesAmount` as well.

```
126     function transferFrom(
127         address from,
128         address to,
129         uint256 amount
130     ) public override returns (bool) {
131         if (taxFee == 0) {
132             _spendAllowance(from, msg.sender, amount);
133             _transfer(from, to, amount);
134         } else {
135             (uint256 sendAmount, uint256 feesAmount) = _checkFees(from, amount);
136             _spendAllowance(from, msg.sender, sendAmount);
137             _transfer(from, to, sendAmount);
138             _transfer(from, feeCollector, feesAmount);
```

```
        }
        return true;
    }
```

Listing 3.1: `CBI::transferFrom()`

Recommendation Revise the above `transferFrom()` function to properly adjust the spending allowance after the transfer. The same issue is applicable to both `CBI` and `JUKU` token contracts.

Status The issue has been fixed by this commit: `311f391`.

3.2 Potential Front-Running/MEV With Reduced Return

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CBI_Treasury`, `MultiSigTreasury`
- Category: Time and State [9]
- CWE subcategory: CWE-682 [4]

Description

The `JUKU` platform support two treasuries, i.e., `CBI_Treasury` and `CMultiSigTreasury`. Both treasuries have the need of swapping an input token to another. However, our analysis shows the current conversion does not enforce meaningful slippage control.

```
    function _swapTokens(
        address inputToken,
        address outputToken,
        uint256 amount,
        address user,
        string memory userId
    ) internal {
        require(amount > 0, "CBI_Treasury: Zero amount");
        Token storage inputTokenInfo = allowedTokensInfo[inputToken];
        Token storage outputTokenInfo = allowedTokensInfo[outputToken];
        require(
            inputTokenInfo.allowed && outputTokenInfo.allowed,
            "CBI_Treasury: Not allowed token"
        );
        uint balanceInputToken = IERC20(inputToken).balanceOf(address(this));
        require(
            balanceInputToken >= amount,
            "CBI_Treasury: Not enough token balance"
        );
        require(
            balanceInputToken - amount >= inputTokenInfo.swapLimit,
            "CBI_Treasury: Token swap limit exceeded"
```

```
336         );
337 
338         address[] memory path = new address[](2);
339         path[0] = inputToken;
340         path[1] = outputToken;
341 
342         uint256[] memory swapAmounts = swapRouter.swapExactTokensForTokens(
343             amount,
344             0,
345             path,
346             address(this),
347             block.timestamp
348         );
349         emit SwapTokens(
350             inputToken,
351             outputToken,
352             amount,
353             swapAmounts[1],
354             user,
355             userId
356         );
357     }
```

Listing 3.2: `CBI_Treasury::_swapTokens()`

To elaborate, we show above one example routine `_swapTokens()`. We notice the conversion is routed to an external `swapRouter` in order to swap one asset to another. And the swap operation does not specify any restriction on possible slippage (line 344) and is therefore vulnerable to possible front-running attacks, resulting in a smaller gain for this round of conversion. A similar issue also exists in `_swapTokensForExactToken()` from both treasury contracts as well as swap-related routines from `YieldOptimizer` and `YieldOptimizerStaking`.

Note that this is a common issue plaguing current AMM-based DEX solutions. Specifically, a large trade may be sandwiched by a preceding sell to reduce the market price, and a tailgating buy-back of the same amount plus the trade amount. Such sandwiching behavior unfortunately causes a loss and brings a smaller return as expected to the trading user because the swap rate is lowered by the preceding sell. As a mitigation, we may consider specifying the restriction on possible slippage caused by the trade or referencing the `TWAP` or `time-weighted average price` of `UniswapV2`. Nevertheless, we need to acknowledge that this is largely inherent to current blockchain infrastructure and there is still a need to continue the search efforts for an effective defense.

Recommendation Develop an effective mitigation (e.g., slippage control) to the above front-running attack to better protect the interests of farming users.

Status The issue has been fixed by this commit: `311f391`.

3.3 Improper Limit Enforcement in CBI_Treasury

- ID: PVE-003
- Severity: Medium
- Likelihood: Low
- Impact: Medium
- Target: `CBI_Treasury`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned earlier, the `JUKU` platform support two treasuries - `CBI_Treasury` and `CMultiSigTreasury` . While examining the allowed tokens in the first treasury, we notice the use of `swapLimit` and `withdrawLimit` to ensure certain limits for swap and withdraw operations. However, our analysis shows their enforcement needs to be revisited.

If we use the `swapLimit` as an example, we show below the implementation of the related `_swapTokens()` function. The `swapLimit` enforcement is applied to ensure the input token amount for each swap will not exceed the given limit. However, the current implementation `require(balanceInputToken - amount >= inputTokenInfo.swapLimit)` basically ensures the remaining amount after the swap will not be smaller than the swap limit! An intended enforcement should be revised as `require(amount <= inputTokenInfo.swapLimit);`.

```
    function _swapTokens(
        address inputToken,
        address outputToken,
        uint256 amount,
        address user,
        string memory userId
    ) internal {
        require(amount > 0, "CBI_Treasury: Zero amount");
        Token storage inputTokenInfo = allowedTokensInfo[inputToken];
        Token storage outputTokenInfo = allowedTokensInfo[outputToken];
        require(
            inputTokenInfo.allowed && outputTokenInfo.allowed,
            "CBI_Treasury: Not allowed token"
        );
        uint balanceInputToken = IERC20(inputToken).balanceOf(address(this));
        require(
            balanceInputToken >= amount,
            "CBI_Treasury: Not enough token balance"
        );
        require(
            balanceInputToken - amount >= inputTokenInfo.swapLimit,
            "CBI_Treasury: Token swap limit exceeded"
        );

        address[] memory path = new address[](2);
```

```
        path[0] = inputToken;
        path[1] = outputToken;
        ...
    }
```

Listing 3.3: `CBI_Treasury::_swapTokens()`

A similar issue is also applicable to the `swapLimit` enforcement in `_swapTokensForExactToken()` and the `withdrawLimit` enforcement in `_withdraw()`.

Recommendation Revisit the above-mentioned routines to properly apply the intended `swapLimit` and `withdrawLimit`.

Status The issue has been fixed by this commit: `311f391`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [6]
- CWE subcategory: CWE-287 [3]

Description

In the `JUKU` platform, there is a privileged `owner/admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and pool 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.

```
    function updateAllowedToken(
        address token,
        bool allowed,
        uint swapLimit,
        uint withdrawLimit
    ) external onlyOwner {
        require(Address.isContract(token), "CBI_Treasury: Not contract");
        Token storage tokenInfo = allowedTokensInfo[token];
        if (tokenInfo.tokenAddress == address(0)) {
            IERC20(token).safeApprove(address(swapRouter), type(uint256).max);
            tokenInfo.tokenAddress = token;
        }

        tokenInfo.allowed = allowed;
        tokenInfo.swapLimit = swapLimit;
```

```
        tokenInfo.withdrawLimit = withdrawLimit;

        emit UpdateAllowedToken(token, swapLimit, withdrawLimit, allowed);
    }

    function updateAdmin(address newAdmin) external onlyOwner {
        require(newAdmin != address(0), "CBI_Treasury: Null address");
        require(
            newAdmin != admin,
            "CBI_Treasury: new admin equal to the current admin"
        );
        admin = newAdmin;
        emit UpdateAdmin(newAdmin);
    }
```

Listing 3.4: Example `Setters` in the `CBI_Treasury` Contract

```
    function replenishStaking(address token, uint256 amount)
        external
        onlyStaking
        whenNotPaused
    {
        _withdraw(token, amount, staking);
        emit ReplenishStaking(token, amount, staking);
    }

    /**
    @dev The function updates the address of the staking smart contract
    Only the owner or admin can call.
    @param newStaking new staking address
    */
    function setStaking(address newStaking) external onlyAdmin {
        staking = newStaking;
        emit SetStaking(newStaking);
    }
    ...
    function pause() external onlyOwner {
        _pause();
    }

    /**
    @dev The function turns off the pause, the contract returns to the normal working
        state
    Only the owner can call.
    */
    function unPause() external onlyOwner {
        _unpause();
    }

    /**
    @dev External function for emergency withdrawing tokens from the YO.
    Only the owner or admin can call.
    @param token token address.
```

```
    @param amount withdraw token amount.
    @param recipient recipient wallet address.
    */
    function emergencyWithdraw(
        address token,
        uint256 amount,
        address recipient
    ) external onlyOwner {
        _withdraw(token, amount, recipient);
        emit EmergencyWithdraw(token, amount, recipient);
    }
```

Listing 3.5: Example Privileged Functions in the `YieldOptimizer` Contract

Apparently, if the privileged `owner/admin` account is a plain EOA account, this may be worrisome and pose counter-party risk to the exchange 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 if current contracts have the support of being deployed behind a proxy, 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 that these privileged functions should be called by a trusted multi-sig account, not a plain EOA account.

3.5 Improved Logic in updateLowerAdmin()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `CMultiSigTreasury`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1126 [2]

Description

As mentioned earlier, the `JUKU` platform support `CBI_Treasury` and `CMultiSigTreasury`. While examining the specific settings in the second treasury, we notice a related `setter`, i.e., `updateLowerAdmin()`, can be improved.

Specifically, this specific `setter` is intended to update the lower admin `lowerAdmin`. While the current implementation properly revokes the role from the current `lowerAdmin` and configures the role for the new `newLowerAdmin`, the storage variable `lowerAdmin` however is not updated with the given `newLowerAdmin`!

```
    /**
    @dev the function can update lower admin.
    Only owner can call.
    @param newLowerAdmin new lower admin address.
    */
    function updateLowerAdmin(address newLowerAdmin) external onlyOwner{
        require(newLowerAdmin != address(0), "MultiSigTreasury: Zero address");
        revokeRole(LOWER_ADMIN_ROLE, lowerAdmin);
        _setupRole(LOWER_ADMIN_ROLE, newLowerAdmin);

        emit UpdateLowerAdmin(newLowerAdmin);
    }
```

Listing 3.6: `MultiSigTreasury::updateLowerAdmin()`

Recommendation Revise the `updateLowerAdmin()` implementation to properly update the `lowerAdmin` state.

Status The issue has been fixed by this commit: `311f391`.

3.6 Revisited Quorum Enforcement in MultiSigTreasury

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MultiSigTreasury`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

The `MultiSigTreasury` contract allows for multi-sig management of various treasury operations. In particular, the intended operation needs to be agreed by at least `quorum` of current trusted verifiers. While examining the current logic, we notice its implementation has a corner case that is not addressed.

In particular, for a given withdraw operation, the treasury contract has a public `withdrawVote()` function to conduct voting for the withdrawal of tokens. If the total number of `for`-votes reaches the specific `quorum`, the withdraw operation will be performed (lines 303-319). However, in the corner case of `quorum=1`, the number of `for`-votes will be initialized to `trxInfo.confirmations=1` when the withdraw operation is created (in `createWithdrawTrx()`). In other words, even if all verifiers vote `for` on the withdraw operation, the condition on triggering the execution cannot not achieved. The reason is that the only condition will be `if (trxInfo.confirmations == quorum)` (line 303)!

```
282         function withdrawVote(uint256 trxId, bool confirm)
283         external
284         onlyRole(VERIFIYER_ROLE)
285     {
286         require(
287             trxId < trxCounter,
288             "MultiSigTreasury: Transaction not created"
289         );
290         TrxData storage trxInfo = trxData[trxId];
291         require(
292             trxInfo.status == TrxStatus.Pending,
293             "MultiSigTreasury: Transaction completed"
294         );
295         require(!isVoted(trxId), "MultiSigTreasury: You have already voted");
296 
297         if (confirm) {
298             _addConfirmation(trxId);
299         } else {
300             _rejectTrx(trxId);
301         }
302 
303         if (trxInfo.confirmations == quorum) {
304             trxInfo.status = TrxStatus.Confirmed;
305             if (!trxInfo.withdrawArgs.isFtm) {
```

```
                _withdraw(
                    trxId,
                    trxInfo.withdrawArgs.token,
                    trxInfo.withdrawArgs.amount,
                    trxInfo.withdrawArgs.recipient
                );
            } else {
                _withdrawFTM(
                    trxId,
                    payable(trxInfo.withdrawArgs.recipient),
                    trxInfo.withdrawArgs.amount
                );
            }
        }

        if ((admins.length - trxInfo.rejects < quorum)) {
            trxInfo.status = TrxStatus.Rejected;
        }
    }
```

Listing 3.7: `MultiSigTreasury::withdrawVote()`

Recommendation Revisit the voting routine to remove the above-mentioned corner case.

Status The issue has been fixed by this commit: `311f391`.

3.7 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-007
- Severity: Low
- Likelihood: Low
- Impact: High
- Target: `YieldOptimizer`
- 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 3.8: 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 `addPool()` routine in the `YieldOptimizer` contract. If the `USDT` token is supported as one of `poolTokens`, the unsafe version of `IERC20Upgradeable(poolTokens[i]).approve(vault, type(uint256).max)` (line 442) 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)!

```
397     function addPool(
398         bytes32 _poolId,
399         address _poolAddress,
400         address _depositToken,
401         address _exitToken,
402         bytes32 _swapRouteForDepositToken,
403         bytes32 _swapRouteForExitToken,
404         bytes32[] memory _swapRoutes,
405         uint256 _exitTokenIndex,
406         bool _isDepositInOneToken,
407         bool _isExitInOneToken
408     ) external onlyAdmin whenNotPaused {
```

```
        Pool storage pool = poolInfo[_poolAddress];
        _require(pool.bptToken == address(0), Errors.POOL_IS_ADDED);

        (address[] memory poolTokens, , ) = IVault(vault).getPoolTokens(
            _poolId
        );
        _require(
            _swapRoutes.length == poolTokens.length,
            Errors.INVALID_ARRAY_LENGHTS
        );

        pool.tokens = poolTokens;
        pool.poolId = _poolId;
        pool.tokensWeights = IWeightedPool(_poolAddress).getNormalizedWeights();
        pool.depositToken = _depositToken;
        pool.exitToken = _exitToken;
        pool.swapRouteForDepositToken = _swapRouteForDepositToken;
        pool.swapRouteForExitToken = _swapRouteForExitToken;
        pool.bptToken = _poolAddress;
        pool.swapRoutes = _swapRoutes;
        pool.isActive = true;
        pool.exitTokenIndex = _exitTokenIndex;
        pool.isDepositInOneToken = _isDepositInOneToken;
        pool.isExitInOneToken = _isExitInOneToken;
        pool.isDefaultAllocations = true;

        Epoch storage epoch = poolRewards[_poolAddress][
            rewardsEpochCounter[_poolAddress]
        ];
        epoch.start = block.timestamp;
        pool.currentEpoch = rewardsEpochCounter[_poolAddress];
        IERC20Upgradeable(_poolAddress).approve(vault, type(uint256).max);
        for (uint256 i = 0; i < poolTokens.length; i++) {
            IERC20Upgradeable(poolTokens[i]).approve(vault, type(uint256).max);
        }
        emit AddPool(
            pool.bptToken,
            pool.poolId,
            pool.tokens,
            pool.tokensWeights
        );
    }
```

Listing 3.9: `YieldOptimizer::addPool()`

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

Status The issue has been fixed by this commit: `311f391`.

4 Conclusion

In this audit, we have analyzed the JUKU design and implementation. The JUKU platform is the Dapp allowing users to use a custodial wallet for the investments/withdraw of the CBI tokens into/out the various bundles/liquidity pools. It has its native token, i.e., CBI. Users can invest their CBI tokens into staking functionality that allows them to generate and receive rewards in JUKU tokens. 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-1041: Use of Redundant Code. https://cwe.mitre.org/data/definitions/1041.html.

[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.