SMART CONTRACT AUDIT REPORT

for

Trustless Finance

Prepared By: Yiqun Chen

PeckShield

June 26, 2021

Document Properties

Client	Trustless Finance
Title	Smart Contract Audit Report
Target	TFDao and Trustless Currency Protocol
Version	1.0
Author	Xuxian Jiang
Auditors	Jing Wang, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	June 26, 2021	Xuxian Jiang	Final Release
1.0-rc2	June 6, 2021	Xuxian Jiang	Release Candidate #2
1.0-rc1	May 29, 2021	Xuxian Jiang	Release Candidate #1
0.5	May 15, 2021	Xuxian Jiang	Add More Findings #4
0.4	May 10, 2021	Xuxian Jiang	Add More Findings #3
0.3	May 3, 2021	Xuxian Jiang	Add More Findings #2
0.2	April 27, 2021	Xuxian Jiang	Add More Findings #1
0.1	April 19, 2021	Xuxian Jiang	Initial Draft

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 **TFDao and Trustless Currency Protocol (TCP)** design document and related smart contract source code, 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 TFDao and Trustless Currency Protocol

The `Trustless Currency Protocol` aims to build a more trustless, broadly distributed stablecoin, which recognizes and addresses various weaknesses in current stablecoin solutions. In particular, existing collateralized stablecoin lending protocols rely on a high degree of trust for mechanisms from prices and collateral to governance and interest rate updates. They are built on price feeds that simply trust in all powerful parties, are backed by centralized collateral that can be frozen by centralized entities or governments, rely on weekly governance votes to adjust interest rates, and do not distribute the protocol token broadly to those that provide value for the protocol: the community.

The basic information of `TFDao And TCP` is as follows:

Table 1.1: Basic Information of TFDao and TCP

Item	Description
Issuer	Trustless Finance
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	June 26, 2021

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

- https://github.com/TrustlessFinance/tcp.git (1bf415c)

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

- https://github.com/TrustlessFinance/tcp.git (c864dde)

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

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.

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

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 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

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.

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.

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the TFDao and TCP 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	5	■■■■■
Low	7	■■■■■■■
Informational	2	■■
Total	14	

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 5 medium-severity vulnerabilities, 7 low-severity vulnerabilities, and 2 informational recommendations.

Table 2.1: Key TFDao and Trustless Currency Protocol Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Fixed
PVE-002	Medium	Possible Costly Market lendZhu From Improper Initialization	Time And State	Fixed
PVE-003	Low	Improved Precision By Multiplication And Division Reordering	Numeric Errors	Fixed
PVE-004	Low	Improved Negative Interest Calculation	Numeric Errors	Fixed
PVE-005	Low	Precise Liquidity Range Validation in _-isLiquidityInRange()	Coding Practices	Fixed
PVE-006	Informational	Simplified approve() In ProtocolToken	Coding Practice	Fixed
PVE-007	Informational	Proper _initHook() Initialization In Accounting	Numeric Errors	Fixed
PVE-008	Low	Resized nftIDs Array For getPoolPositionNftIdsByOwner()	Coding Practice	Fixed
PVE-009	Medium	Possible Flashloan Manipulation Of No Price Confidence	Business Logic	Fixed
PVE-010	Low	The Lack of LiquidationAccount Rewards Handling	Business Logic	Fixed
PVE-011	Medium	Proper Reward/Collateral Attribution in TFDao/Settlement	Business Logic	Fixed
PVE-012	Low	Improved cumulativeDebt Calculation in Accrued Interest	Business Logic	Fixed
PVE-013	Medium	Improved Debt Change Accounting in _-adjustPosition()	Business Logic	Fixed
PVE-014	Medium	Improved Reward Accounting in Governor	Numeric Errors	Fixed

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

3 | Detailed Results

3.1 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Multiple Contracts`
- 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 3.1: `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 `transferFrom()` as well, i.e., `safeTransferFrom()`.

In the following, we show the `lend()` routine in the `Lend` contract. If the `USDT` token is supported as `zhu`, the unsafe version of `require(zhu.transferFrom(msgSender, address(accounting), zhuCount))` (line 87) may revert as there is no return value in the `USDT` token contract's `transfer()` implementation (but the `IERC20` interface expects a return value)!

```
74      function lend(uint zhuCount) external lockProtocol runnable {
75          _requireEffect(zhuCount > 0);
76          address msgSender = msg.sender;
77
78          // accrue interest first so that a user can't take advantage of immediately lending then
79          // unlending when there is a delay in accruing positive interest on borrows.
80          governor.market().accrueInterest();
81
82          // determine how many LendZhu tokens the given amount of zhu is worth.
83          uint lendTokenCount = zhuCount._mul(_lendTokenExchangeRate());
84
85          // Take in the lenders Zhu and give them LendZhu in return.
86          // Deposit Zhu on to the Accounting contract for safe keeping even if this contract is upgraded.
87          require(zhu.transferFrom(msgSender, address(accounting), zhuCount));
88          lendZhu.mintTo(msgSender, lendTokenCount);
89
90          emit Lend(msgSender, zhuCount, lendTokenCount);
91      }
```

Listing 3.2: `Lend::lend()`

The same issue is also present in other contracts, including `TFDao`, `Accounting`, and `Rewards`.

Recommendation Accommodate the above-mentioned idiosyncrasy with safe-version implementation of ERC20-related `transfer()`, `transferFrom()`, and `approve()`.

Status The issue has been addressed by the following commit: `3541620b`.

3.2 Possible Costly Market lendZhu From Improper Initialization

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Lends`
- Category: Time and State [6]
- CWE subcategory: CWE-362 [3]

Description

The `TCP` protocol allows users to deposit supported `ZHU` and get in return `lendZhu` tokens to represent the pool share. While examining the share calculation with the given deposits, we notice an issue that may unnecessarily make the pool token extremely expensive and bring hurdles (or even causes loss) for later depositors.

To elaborate, we show below the `lend()` routine. This routine is used for participating users to deposit the supported asset (e.g., `ZHU`) and get respective `lendZhu` pool tokens in return. The issue occurs when the pool is being initialized under the assumption that the current pool is empty.

```
    function lend(uint zhuCount) external lockProtocol runnable {
        _requireEffect(zhuCount > 0);
        address msgSender = msg.sender;

        // accrue interest first so that a user can't take advantage of immediately lending then
        // unlending when there is a delay in accruing positive interest on borrows.
        governor.market().accrueInterest();

        // determine how many LendZhu tokens the given amount of zhu is worth.
        uint lendTokenCount = zhuCount._mul(_lendTokenExchangeRate());

        // Take in the lenders Zhu and give them LendZhu in return.
        // Deposit Zhu on to the Accounting contract for safe keeping even if this contract is upgraded.
        require(zhu.transferFrom(msgSender, address(accounting), zhuCount));
        lendZhu.mintTo(msgSender, lendTokenCount);

        emit Lend(msgSender, zhuCount, lendTokenCount);
    }
```

Listing 3.3: `Lends::lend()`

```
    function _lendTokenExchangeRate() internal view returns (uint) {
        uint lends = zhu.balanceOf(address(accounting));
        uint totalLendZhu = lendZhu.totalSupply();
```

```
199         return (totalLendZhu == 0 || lends == 0) ? TCPSafeMath.ONE : totalLendZhu._div(
                lends);
200     }
```

Listing 3.4: `Lends::_lendTokenExchangeRate()`

Specifically, when the pool is being initialized (line 199), the token exchange rate directly takes the value of `TCPSafeMath.ONE` (line 199) and the malicious actor can simply deposit $1WEI$. As this is the first deposit, the current total supply equals the calculated `lendTokenCount = zhuCount._mul(_lendTokenExchangeRate())= 1 WEI`. With that, the actor can further deposit a huge amount of `ZHU` with the goal of making the `lendZhu` pool token extremely expensive.

An extremely expensive `lendZhu` pool token can be very inconvenient to use as a small number of $10WEI$ may denote a large value. Furthermore, it can lead to a precision issue in truncating the computed pool tokens for deposited assets. If truncated to be zero, the deposited assets are essentially considered dust and kept by the pool without returning any pool tokens.

This is a known issue that has been mitigated in popular `Uniswap`. When providing the initial liquidity to the contract (i.e. when totalSupply is 0), the liquidity provider must sacrifice 1000 LP tokens (by sending them to $address(0)$). By doing so, we can ensure the granularity of the LP tokens is always at least 1000 and the malicious actor is not the sole holder. This approach may bring an additional cost for the initial liquidity provider, but this cost is expected to be low and acceptable.

Recommendation Revise current execution logic of `_lendTokenExchangeRate()` to defensively calculate the share amount when the pool is being initialized. An alternative solution is to ensure guarded launch that safeguards the first deposit to avoid being manipulated.

Status The issue has been addressed by the following commit: `b7a948e7`.

3.3 Improved Precision By Multiplication And Division Reordering

- ID: PVE-003
- Severity: Low
- Likelihood: Medium
- Impact: Low

- Target: `Multiple Contracts`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

`SafeMath` is a widely-used Solidity `math` library that is designed to support safe `math` operations by preventing common overflow or underflow issues when working with `uint256` operands. While it indeed blocks common overflow or underflow issues, the lack of `float` support in `Solidity` may introduce another subtle, but troublesome issue: precision loss. In this section, we examine one possible precision loss source that stems from the different orders when both multiplication (`mul`) and division (`div`) are involved.

In particular, we use the `Market::_updatePositionImpl()` as an example. This routine is used to update the current market position.

```
    function _updatePositionImpl(
        IAccounting.DebtPosition memory _position,
        IAccounting.SystemDebtInfo memory sdi,
        uint64 timeNow,
        uint64 periods
    ) internal pure returns (IAccounting.DebtPosition memory position, uint rewards) {
        position = _position;

        // if this is the first initialization don't accrue interest or calculate rewards
        // if there is no debt then there is no rewards or interest to accrue
        if (position.lastTimeUpdated > 0 && position.debt > 0) {

            // update the position debt.
            position.debt = position.debt.mulDiv(sdi.debtExchangeRate, position.startDebtExchangeRate);

            // calculate the average system debt during the time since the last time this position
            // was updated.
            uint avgDebtPerPeriods = sdi.cumulativeDebt.sub(position.startCumulativeDebt) / periods;

            // given the rewards added since last update and this position's portion of total
            // debt, calculate the TCP this position is entitled to.
            // Scale it by the amount of time since the last update, to prevent against outsized rewards for
            // exceedingly small lengths of time between updates that cross a period border.
            rewards = position.debt
                .mulDiv(sdi.totalTCPRewards.sub(position.startTCPRewards), avgDebtPerPeriods)
                .mulDiv(timeNow.sub(position.lastTimeUpdated), PERIOD_LENGTH * periods);
        }

        // update the position metadata to be current
        position.startDebtExchangeRate = sdi.debtExchangeRate;
        position.startTCPRewards = sdi.totalTCPRewards;
        position.startCumulativeDebt = sdi.cumulativeDebt;
        position.lastTimeUpdated = timeNow;
    }
```

Listing 3.5: `Market::_updatePositionImpl()`

We notice the calculation of the resulting rewards (line 635) involves mixed multiplication and division. For improved precision, it is better to calculate the rewards by canceling out the periods (lines 629 and 637). Similarly, the calculation of `_requireWellCollateralized()` in the same contract (lines 403 – 404) can be accordingly adjusted. Note that the resulting precision loss may be just a small number, but it plays a critical role when certain boundary conditions are met. And it is always the preferred choice if we can avoid the precision loss as much as possible.

Note the `Liquidations::_discoverUndercollateralizedPositionsImpl()` and `Lend::mintZhu()` routines can be similarly improved.

Recommendation Revise the above calculations to better mitigate possible precision loss.

Status The issue has been addressed by the following commits: `629ad861`, `42e669bb`, `c5e952e1`, and `be1bad31`.

3.4 Improved Negative Interest Calculation

- ID: PVE-004
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Market`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The `Zhu` stablecoin is designed to support both positive and negative interest rates. The calculations are only completed at most every period, which is every one hour to save gas. To elaborate, we show below the `_calculateInterest()` routine that is designed to calculate interest given the current protocol debt, interest rate, elapsed time since last accrual, as well as the current reserves.

While analyzing the negative interest rate, we notice the new exchange rate calculation (line 538) can be improved. Specifically, it is currently calculated as `sdi.debtExchangeRate._div(periodInterestRate)` (line 540), which needs to be revised as the following:

`sdi.debtExchangeRate._mul(TCPSafeMath.ONE.sub(periodInterestRateAbsoluteValue))`.

```
517     function _calculateInterest(
518         IAccounting.SystemDebtInfo memory sdi,
519         uint64 periods,
520         uint annualInterestRate,
521         bool positiveInterestRate,
522         uint reserves,
523         uint _interestPortionToLenders
524     ) internal pure returns (CalculatedInterestInfo memory cii) {
```

```
        // if the interest rate for borrowing ZHU is currently 0%, our job is easy, don't change anything.
        if (annualInterestRate == 0) {
            cii.newDebt = sdi.debt;
            cii.newExchangeRate = sdi.debtExchangeRate;
            return cii;
        }

        // Given an annual interest rate and the number of periods we are behind, calculate the
        // interest rate since the last update, relative to 1.
        uint periodInterestRate = TCPSafeMath.ONE.add(annualInterestRate.mulDiv(periods, PERIODS_PER_YEAR));

        // the debt exchange rate should increase if there is a positive interest rate (debt increases)
        // or decrease if there is a negative interest rate.
        cii.newExchangeRate = positiveInterestRate
            ? sdi.debtExchangeRate._mul(periodInterestRate)
            : sdi.debtExchangeRate._div(periodInterestRate);

        // given this new exchange rate, it is trivial to calculate the total system debt.
        cii.newDebt = sdi.debt.mulDiv(cii.newExchangeRate, sdi.debtExchangeRate);

        // now we must allocate the increased system debt between reserves and interest to ZHU lenders.
        // if the system debt has decreased (negative interest) then positive interest only is taken from reserves,
        // but ONLY if there are sufficient reserves.
        if (cii.newDebt > sdi.debt) {
            // positive interest case
            uint additionalDebt = cii.newDebt - sdi.debt;
            // the interestPortionToLenders of the additional debt goes to those who are lending ZHU
            cii.additionalLends = additionalDebt._mul(_interestPortionToLenders);
            // the rest is added to reserves.
            cii.additionalReserves = additionalDebt - cii.additionalLends;
        } else if (cii.newDebt < sdi.debt) {
            // negative interest case
            uint debtReduction = sdi.debt - cii.newDebt;
            // we must first check that there are reserves to pay the proposed negative interest.
            if (debtReduction >= reserves) {
                // Don't accrue negative interest if there is insufficient reserves, otherwise
                // there would be unbacked zhu.
                cii.newDebt = sdi.debt;
                cii.newExchangeRate = sdi.debtExchangeRate;
            } else {
                // If there are sufficient reserves, they need to be reduced to pay negative interest.
```

```
                cii.reducedReserves = debtReduction;
            }
        }
    }
```

Listing 3.6: `Market::_calculateInterest()`

Recommendation Revisit the new exchange rate calculation by better taking into account the negative interest rate.

Status The issue has been addressed by the following commit: `03a6527a`.

3.5 Precise Liquidity Range Validation in _isLiquidityInRange()

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Rewards`
- Category: Coding Practices [7]
- CWE subcategory: CWE-1041 [1]

Description

The `TCP` protocol has built-in incentive mechanisms to encourage participating users to provide required liquidity. Specifically, users can provide liquidity around the current price for the given pool. The current price of the pool, and therefore whether the provided liquidity is "in-range", is confirmed by taking an instant `TWAP` at the time of locking liquidity. The `Rewards` contract takes in tokens from the user, and creates a tokenized position through the `Uniswap V3 NonfungiblePositionManager`, and stores the resulting token within the protocol. Users are rewarded with a portion of the constant stream of protocol tokens proportional to the share of total virtual liquidity they have provided.

To validate whether the provided liquidity is "in-range," the protocol has a handy helper `_isLiquidityInRange()`. This helper essentially checks the given `tick` falls within the range, i.e., `position.tickLower < tick && tick < position.tickUpper` (line 648). However, the range validation needs to be performed as `position.tickLower <= tick && tick < position.tickUpper`.

```
643     /// @dev Checks whether or not the position provides liquidity that is in range
644     function _isLiquidityInRange(
645         IAccounting.PoolPosition memory position,
646         int24 tick
647     ) internal pure returns (bool) {
648         return position.tickLower < tick && tick < position.tickUpper;
649     }
```

Listing 3.7: `Rewards::_isLiquidityInRange()`

Recommendation Revise the above `_isLiquidityInRange()` routine to properly validate whether the given liquidity falls "in-range."

Status The issue has been addressed by the following commit: `7f8069de`.

3.6 Simplified approve() In ProtocolToken

- ID: PVE-006
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `ProtocolToken`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The `TCP` protocol has a governance token that is heavily forked from the `COMP` token contract. While examining the changes, we notice the resulting `approve()` can be simplified.

To elaborate, we show below the `approve()` function. It comes to our attention the internal `if`-statement can be simplified as its `then`-branch and `else`-branch can be consolidated into `amount = rawAmount`.

```
    function approve(address spender, uint rawAmount) external returns (bool) {
        uint256 amount;
        if (rawAmount == uint256(-1)) {
            amount = uint256(-1);
        } else {
            // NOTE: removed amount = safe256(rawAmount, "approve: amount exceeds 256
                bits");
            amount = rawAmount;
        }

        allowances[msg.sender][spender] = amount;

        emit Approval(msg.sender, spender, amount);
        return true;
    }
```

Listing 3.8: `ProtocolToken::approve`

Recommendation Simplify the `approve()` logic by removing the redundant `if`-condition.

Status The issue has been addressed by the following commit: `8c3ff2db`.

3.7 Proper _initHook() Initialization In Accounting

- ID: PVE-007
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Accounting`
- Category: Numeric Errors [9]
- CWE subcategory: CWE-190 [2]

Description

The `TCP` protocol is conscious in its gas cost by minimizing the consumed storage sizes. For example, the system debt information is saved in the following data structure `SystemDebtInfoStorage` while its computation is based on the memory-version `SystemDebtInfo`.

```
    struct SystemDebtInfo {
        uint debt;
        uint totalTCPRewards;
        uint cumulativeDebt;
        uint debtExchangeRate;
    }

    struct SystemDebtInfoStorage {
        uint cumulativeDebt;
        uint128 debt;
        uint128 debtExchangeRate;
        uint128 totalTCPRewards;
    }
```

Listing 3.9: `Accounting::_initHook()`

The packed storage is helpful to reduce the number of storage reads and writes. During our analysis with the above the system debt information, we notice its initialization can be improved. To elaborate, we show below the `_initHook()` routine that initializes various accounting-related storages, including the system debt information. It comes to our attention the initialization of system debt is cast into `uint96`, which should be `uint128`.

```
    function _initHook() internal override {
        // initialize system debt information.
        sdi.debtExchangeRate = TCPSafeMath.ONE.toUint96();

        // initialize the liquidation account, which is managed separately from user
            positions.
        liquidationAccount.startDebtExchangeRate = TCPSafeMath.ONE.toUint128();

        _setNFTApproval(address(governor.rewards()));

        validUpdate[this.stopIndexingDebtPositions.selector] = true;
        validUpdate[this.stopIndexingPoolPositions.selector] = true;
```

```
    }
```

Listing 3.10: `Accounting::_initHook()`

Recommendation Properly initialize the system debt information with the right type cast information.

Status The issue has been addressed by the following commit: `438991dc`.

3.8 Resized nftIDs Array For getPoolPositionNftIdsByOwner()

- ID: PVE-008
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Accounting`
- Category: Coding Practices [7]
- CWE subcategory: CWE-563 [4]

Description

The `TCP` protocol maintains the accounting of all pool positions. A user can check whether a specific position is currently owned by an owner. To elaborate, we show below the `getPoolPositionNftIdsByOwner()` function that is designed to return all pool positions owned by a specific user. It comes to our attention that the returned list of NFT positions may have the wrong array size. Specifically, the returned array may not always have the size of `localIDs.length`. There is a need to readjust the size if the length does not match (line 652).

```
643     /// @dev Get the currently owned NFT positions by owner
644     function getPoolPositionNftIdsByOwner(address owner) external view override returns
            (uint128[] memory) {
645         uint64[] memory localIDs = poolPositionsByOwner[owner];
646         uint128[] memory nftIDs = new uint128[](localIDs.length);

648         uint j;
649         uint128 nftID;
650         for(uint i = 0; i < localIDs.length; i++) {
651             nftID = localNftID[localIDs[i]];
652             if (owner == poolPosition[nftID].owner) nftIDs[j++] = nftID;
653         }

655         return nftIDs;
656     }
```

Listing 3.11: `Accounting::getPoolPositionNftIdsByOwner()`

Recommendation Revise the above `getPoolPositionNftIdsByOwner()` routine to return the requested array with the right size.

Status The issue has been addressed by the following commit: `a19ce013`.

3.9 Possible Flashloan Manipulation Of No Price Confidence

- ID: PVE-009
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Settlement`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `TCP` protocol has developed a settlement procedure if the protocol is shutdown by governance. The settlement procedure works by firstly determining the `priceDiscoveryStartTime`, which is the last possible completion time for an auction or two days after shutdown, whichever is later. Next the so-called `price no confidence` phase starts and lasts until the `priceDiscoveryStartTime` determined in the previous step. During this time period, if more than `SETTLEMENT_PRICE_NO_CONFIDENCE_THRESHOLD` TCP tokens are locked in, then the new price provider contract for `ETH` needs to be installed via governance. After that, the new price provider can register the settlement price and anyone can then claim collateral from their positions or claim collateral for their `Zhu`.

To elaborate, we show below the `confirmNoPriceConfidence()` routine that is designed to determine whether a new price provider contract for `ETH` needs to be installed. Our analysis shows that it suffers from possible flashloan attacks by calling `stakeTokensForNoPriceConfidence()`, `confirmNoPriceConfidence()`, and `unstakeTokensForNoPriceConfidence()` sequentially within the same flashloan transaction.

```
    function confirmNoPriceConfidence() external lockProtocol {
        _requireCorrectStage(SettlementStage.WaitingForPriceTime);

        uint threshold = governor.distributedTCP()._mul(
            SETTLEMENT_PRICE_NO_CONFIDENCE_THRESHOLD);

        require(tcp.balanceOf(address(this)) > threshold, 'Insufficient no confidence
            votes');

        priceConfidence = false;

        emit NoConfidenceConfirmed(msg.sender);
    }
```

Listing 3.12: `Settlement::confirmNoPriceConfidence()`

Note a similar issue is also present in the governor emergency shutdown, the no-interest borrowing (and paying back within a period), and flash providing liquidity in `Rewards`.

Recommendation Revise the above settlement procedure by ensuring the staking tokens cannot be unstaked within the same block.

Status The issue has been addressed by the following commits: `196f2eab`, `33e84a81`, `54adb374`, and `2a9b79ab`.

3.10 The Lack of LiquidationAccount Rewards Handling

- ID: PVE-010
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Rewards`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Within the `TCP` protocol, there is a special account, i.e., `LiquidationAccount`, that holds the debt and collateral for liquidated accounts. By design, the protocol distributes the liquidity rewards among all participating entities, including `liquidity providers` of the protocol/collateral/reference pools and most importantly, `borrowers`. The `borrowers` also include the special `LiquidationAccount`.

We emphasize that the `LiquidationAccount` contributes to the system debt, which accrues the interest. Note that no one is able to claim the rewards, which effectively reduces the total inflation. However, the accrued interest from `LiquidationAccount` is counted into the total minted `TCP`, i.e., `distributedTCP()`

```
    /// @notice Returns the total supply of TCP that could possibly vote.
    function distributedTCP() public view override returns (uint) {
        return totalLiquidityRewards().add(tcpMinted);
    }
```

Listing 3.13: `Governor::distributedTCP()`

From another perspective, the distributed governance tokens may be used in various scenarios, including the settlement procedure for no price confidence, the proposal voting (that affects the quorum and proposal threshold in `TCPGovernorAlpha`), as well as emergence shutdown in `Governor`. With that, it is important to ensure the rewarded `TCP` to `LiquidationAccount` are properly removed for the total `distributedTCP()`.

Recommendation There is no need to take the rewards into account for the protocol-wide liquidation account. An example revision is shown as follows:

```
    /// @notice Returns the total supply of TCP that could possibly vote.
    function distributedTCP() public view override returns (uint) {
        return totalLiquidityRewards().add(tcpMinted).sub(liquidationAccountRewards);
    }
```

Listing 3.14: `Governor::distributedTCP()`

Status The issue has been addressed by the following commit: `02e2cde1`.

3.11 Proper Reward/Collateral Attribution in TFDao/Settlement

- ID: PVE-011
- Severity: Medium
- Likelihood: Medium
- Impact: Low
- Target: `Multiple Contracts`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The `TCP` protocol makes a systematic use of `NFTs` to manage the ownership and reward dissemination. For example, the `TFDao` issues an `NFT` for each locking user and the liquidity `Rewards` contract mints an an `NFT` for each liquidity-providing user. In the analysis of the reward logic among these `NFTs`, we note the recipient of these rewards or collateral can be better attributed.

Using `TFDao` as an example, its internal handler `_distributeRewardsAndGetUpdatedPosition()` is designed to retrieve the given position from storage, update the global rewards, accrue rewards for the position, and distribute the rewards for the position. It comes to our attention that the rewards are distributed to the caller (line 463), not the `NFT` owner.

```
433     function _distributeRewardsAndGetUpdatedPosition(
434         uint64 positionNFTTokenID
435     ) internal returns (TokenPosition memory position) {
436         position = _getPosition(positionNFTTokenID);
437         // should never hit this
438         require(position.count > 0, 'Position DNE');

440         // update global inflation
441         _accrueInflation();

443         // if reward is up to date, then the position needs no updating
444         if (position.lastPeriodUpdated >= lastPeriodGlobalInflationUpdated) return
                position;

446         TokenRewardsStatus memory rs = _getRewardsStatus(position.tokenID);
```

```
        // calculate the average virtual count of tokens over the periods since the last
             time
        // this position was updated, so that we can calculate this positions portion of
             the overall
        // rewards that have been accrued since that last update.
        uint avgVirtualCountPerPeriods =
            rs.cumulativeVirtualCount.sub(position.startCumulativeVirtualCount)
                / (lastPeriodGlobalInflationUpdated - position.lastPeriodUpdated);

        // Multiply the total rewards since the last update by the ratio of this
            position's
        // contribution to the total virtual count during that time
        uint rewards = _virtualCount(position.count, position.durationMonths).mulDiv(
            rs.totalRewards.sub(position.startTotalRewards),
            avgVirtualCountPerPeriods);

        // accrue the rewards
        if (rewards > 0) {
            tfToken.mintTo(msg.sender, rewards);

            emit InflationDistributed(positionNFTTokenID, msg.sender, rewards);
        }

        // update the in memory position's global inflation variables
        position.startTotalRewards = rs.totalRewards;
        position.startCumulativeVirtualCount = rs.cumulativeVirtualCount;
        position.lastPeriodUpdated = _currentPeriod();

        // it is the calling function's responsibility to update the position storage if
             desired.
        // The caller may choose to delete the position instead, if the tokens are being
             unlocked
        return position;
    }
```

Listing 3.15: `TFDao::_distributeRewardsAndGetUpdatedPosition()`

The same issue is also present in the following two routines: `TFDao::unlockTokens()` and `Settlement::withdrawCollateral()`

Recommendation Revise the above mentioned three routines to properly attribute the rewards (or collateral) back to respective owners, not the current caller.

Status The issue has been addressed by the following commits: `0aa6431c`, and `bed7dfa3`, and `8585a1d0`.

3.12 Improved cumulativeDebt Calculation in Accrued Interest

- ID: PVE-012
- Severity: Low
- Likelihood: Medium
- Impact: Low
- Target: `Market`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

Within the `TCP` protocol, the `Market` contract is one core contract that users mainly interact with. It is responsible for allowing users to borrow `Zhu` and pay it back, charging positive interest on positions, paying negative interest to borrowers, and distributing protocol tokens `TCP` to those who borrow. In the following, we examine the interest accrual logic.

To elaborate, we show below the `_accrueInterestImpl()` function that is designed to accrue global interest. To save gas, it only needs to be called once per one-hour period. While examining its logic, we notice the `cumulativeDebt` state is not properly updated (line 489). Specifically, it needs to use the current debt, instead of the next future debt after the update. This state is important as it is used to calculate a position's values given any accrued interest, hence influencing the `TCP` rewards.

```
458     function _accrueInterestImpl() internal {
459         // The current period, ending at shutdown time if the protocol is shutdown
460         uint64 period = _currentPeriodEndingAtShutdown();
461         if (period <= lastPeriodGlobalInterestAccrued) return;

463         // total number of periods we are calculating for. Safe due to the check above
464         uint64 periods = period - lastPeriodGlobalInterestAccrued;
465         // get system debt info
466         IAccounting.SystemDebtInfo memory sdi = accounting.getSystemDebtInfo();
467         // get rates contracdt
468         IRates rates = governor.rates();

470         // calculate new interest information including:
471         // global debt, debt exchange rate, and change in reserves and the lend fund.
472         CalculatedInterestInfo memory cii = _calculateInterest(
473             sdi,
474             periods,
475             rates.interestRateAbsoluteValue(),
476             rates.positiveInterestRate(),
477             zhu.reserves(),
478             interestPortionToLenders);

480         // Calculate how many TCP tokens should be allocated to borrowers.
481         uint rewardCount = governor.currentDailyRewardCount()
482             ._mul(governor.rewards().borrowRewardsPortion())
483             .mulDiv(periods, PERIODS_PER_DAY);
```

```
        // update system debt info according to new data.
        sdi = IAccounting.SystemDebtInfo({
            debt: cii.newDebt,
            totalTCPRewards: sdi.totalTCPRewards.add(rewardCount),
            cumulativeDebt: sdi.cumulativeDebt.add(cii.newDebt.mul(periods)),
            debtExchangeRate: cii.newExchangeRate
        });

        // save the new data.
        accounting.setSystemDebtInfo(sdi);

        // register that interest is up to date as of the current period.
        lastPeriodGlobalInterestAccrued = period;

        // expand or reduce reserves, and accrue interest to lent tokens.
        if (cii.additionalLends > 0) zhu.mintTo(address(accounting), cii.additionalLends);
        if (cii.additionalReserves > 0) zhu.mintTo(address(zhu), cii.additionalReserves);
        if (cii.reducedReserves > 0) zhu.burnReserves(cii.reducedReserves);

        emit InterestAccrued(
            period,
            periods,
            sdi.debt,
            sdi.totalTCPRewards,
            sdi.cumulativeDebt,
            sdi.debtExchangeRate);
    }
```

Listing 3.16: `Market::_accrueInterestImpl()`

Recommendation Properly calculate the `cumulativeDebt` for fair dissemination of accrued interest.

Status The issue has been addressed by the following commit: `9a30c3e5`.

3.13 Improved Debt Change Accounting in _adjustPosition()

- ID: PVE-013
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: Market
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

As mentioned in Section 3.12, the Market contract is responsible for allowing users to borrow Zhu and pay it back, charging positive interest on positions, paying negative interest to borrowers, and distributing protocol tokens TCP to those who borrow. In the following, we examine the borrow position management.

To elaborate, we show below the `_adjustPosition()` function that is designed to handle increasing debt and/or collateral of an account. It has a flawed logic in not properly handling all possible cases of debt changes. Specifically, when the given `debtChange` is negative and its absolute value is larger than the current position debt, extra Zhu tokens may be burn from the calling user: `zhu.burnFrom(msg.sender, uint(-debtChange))` (line 375). In this case, the proper amount to burn should be the position debt, which has been fully repaid.

```
310     function _adjustPosition(
311         IAccounting.DebtPosition memory position,
312         uint64 positionID,
313         int debtChange,
314         uint collateralDecrease
315     ) internal {
316         // CHECK INPUTS
317         uint collateralIncrease = msg.value;

319         require(!(collateralIncrease > 0 && collateralDecrease > 0), 'Cant increase &
                decrease collateral');
320         int collateralChange = collateralIncrease.toInt256();
321         if (collateralChange == 0) collateralChange = -(collateralDecrease.toInt256());

323         require(debtChange != 0  collateralChange != 0, 'Noop');

325         // HANDLE DEBT CHANGE
326         if (debtChange != 0) {
327             if (debtChange > 0)  {
328                 uint debtIncrease = uint(debtChange);

330                 _requireDebtSupported(accounting.debt().add(debtIncrease));
331                 // increase the debt count of the position.
332                 position.debt = position.debt.add(debtIncrease);
```

```
                // store the time of this borrow in order to prevent using borrow/payback for flash borrowing
                position.lastBorrowTime = _currentTime();
                // record that total system debt has increased
                accounting.increaseDebt(debtIncrease);
            } else {
                require(position.debt > 0, 'No debt to pay back');
                // Disallow flash borrowing.
                require(position.lastBorrowTime.add(minBorrowTime) < _currentTime(), 'No flash borrow');

                uint zhuToPayBack = uint(-debtChange);
                // allow user to pay back all debt easily after interest has been accrued.
                zhuToPayBack = zhuToPayBack.min(position.debt);
                // decrease the positions debt.
                position.debt -= zhuToPayBack;
                // register the decreased system debt
                accounting.decreaseDebt(zhuToPayBack);
            }
        }

        // HANDLE COLLATERAL CHANGE
        if (collateralChange != 0) {
            if (collateralChange > 0) position.collateral = position.collateral.add(uint(collateralChange));
            else position.collateral = position.collateral.sub(uint(-collateralChange));
        }

        // TEST FOR FAILURE CASES
        if (position.debt != 0) {
            require(position.debt >= minPositionSize, 'Position too small');

            // If the position is decreasing it's collateral to debt ratio, ensure that it is well collateralized
            if (collateralChange < 0  debtChange > 0) {
                _requireWellCollateralized(position.debt, position.collateral);
            }
        }

        // UPDATE STORAGE
        accounting.setPosition(positionID, position);

        // TRANSFER VALUE
        // mint zhu or burn zhu from the user
        if (debtChange != 0) {
            if (debtChange > 0) zhu.mintTo(msg.sender, uint(debtChange));
            else zhu.burnFrom(msg.sender, uint(-debtChange));
        }

        // take in collateral or send it back to the user
        if (collateralChange != 0) {
```

```
            if (collateralChange > 0) {
                (bool success, ) = address(accounting).call{value: uint(collateralChange
                    )}(new bytes(0));
                require(success, 'ETH_TRANSFER_FAILED');
            } else {
                accounting.sendCollateral(msg.sender, uint(-collateralChange));
            }
        }

        emit PositionAdjusted(positionID, debtChange, collateralChange);
    }
```

Listing 3.17: `Market::_adjustPosition()`

Recommendation Properly reflect the debt change for the borrow position in the above corner case.

Status The issue has been addressed by the following commit: `bc2f24f4`.

3.14 Improved Reward Accounting in Governor

- ID: PVE-014
- Severity: Medium
- Likelihood: High
- Impact: Low

- Target: `Governor`
- Category: Business Logic [8]
- CWE subcategory: CWE-841 [5]

Description

The governance tokens `TCP` has designed an inflation schedule. In particular, the `TCP` inflation will be elevated during genesis, and then will decrease smoothly from the end of genesis until one year after launch. After that `TCP` will inflate at a low, constant amount per day. This means that `TCP` will have a perpetual, low inflation rate that will decrease slowly forever. This is necessary to maintain liquidity incentives.

To elaborate, we show below the `_totalLiquidityRewards()` routine that is designed to compute the total liquidity rewards that have accrued up to the given period (included), starting at the first genesis period. The current calculation suffers from an off-by-one issue, which somehow returns smaller rewards amount. Specifically, if we consider an extreme case with 0 as the given `rewardsPeriod`, the current function returns 0 as the rewards. According to the schedule, it should be `GENESIS_PERIOD_REWARD_COUNT`.

```
    function _totalLiquidityRewards(uint64 rewardsPeriod) internal pure returns (uint) {
        // if we are in the extended period, add total genesis and bootstrap rewards to
        // the number of extended period rewards
```

```
708         if (rewardsPeriod > NON_EXTENDED_PERIODS) {
709             uint64 extendedPeriods = rewardsPeriod - NON_EXTENDED_PERIODS;
710             return
711                 TOTAL_GENESIS_REWARDS +
712                 TOTAL_BOOTSTRAP_REWARDS +
713                 extendedPeriods * EXTENDED_PERIOD_REWARDS;
714         }

716         // if we are in the bootstrap period, add total genesis to the number of
                bootstrap period rewards
717         if (rewardsPeriod > GENESIS_PERIODS) {
718             uint64 bootstrapPeriod = rewardsPeriod - GENESIS_PERIODS;
719             return
720                 TOTAL_GENESIS_REWARDS +
721                 ((FIRST_BOOTSTRAP_PERIOD_REWARDS + _rewardsForBootstrapPeriod(
                        bootstrapPeriod)) / 2) * bootstrapPeriod;
722         }

724         // if we are in the genesis period return the total number of genesis rewards
725         return rewardsPeriod * GENESIS_PERIOD_REWARD_COUNT;
726     }

728     /**
729      * 1m decreasing smoothly to 100k over 365 periods (days)
730      *
731      * pure for easy unit testing
732      */
733     function _rewardsForBootstrapPeriod(uint64 bootstrapPeriod) internal pure returns (
            uint) {
734         return 1e24 - (24657534e14 * bootstrapPeriod);
735     }
```

Listing 3.18: `Governor::_totalLiquidityRewards()`

Moreover, it is also worth mentioning the `24657534e14` constant (line 734) in the `_rewardsForBootstrapPeriod()` needs to be as precise as possible, i.e., `2465753424657534246575`.

Recommendation Properly compute the liquidity rewards in `Governor`.

Status The issue has been addressed by the following commit: `e35cf4f`.

4 Conclusion

In this audit, we have analyzed the `TFDao and TCP` design and implementation. The system presents a unique, robust offering by building a more trustless, broadly distributed stablecoin, which recognizes and addresses various weaknesses in current stablecoin solutions. 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-190: Integer Overflow or Wraparound. https://cwe.mitre.org/data/definitions/190.html.

[3] MITRE. CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition'). https://cwe.mitre.org/data/definitions/362.html.

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

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

[6] MITRE. CWE CATEGORY: 7PK - Time and State. https://cwe.mitre.org/data/definitions/361.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: Numeric Errors. https://cwe.mitre.org/data/definitions/189.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.