SMART CONTRACT AUDIT REPORT

for

TempleDAO Protocol

Prepared By: Xiaomi Huang

PeckShield

April 28, 2022

Document Properties

Client	TempleDAO
Title	Smart Contract Audit Report
Target	TempleDAO
Version	1.0-rc
Author	Xuxian Jiang
Auditors	Jing Wang, Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Confidential

Version Info

Version	Date	Author(s)	Description
1.0-rc1	April 28, 2022	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 TempleDAO protocol, we outline in the report our systematic approach to evaluate potential security issues in the smart contract implementation, expose possible semantic inconsistencies between smart contract code and design document, and provide additional suggestions or recommendations for improvement. Our results show that the given version of smart contracts can be further improved due to the presence of several issues related to either security or performance. This document outlines our audit results.

1.1 About TempleDAO

TempleDAO aims to offer DeFi users steady-growing, low-volatility assets, and help DAOs re-imagine their products with gamified ‘metaverse‘ experiences. Its first offering is the TEMPLE token, which is a fractionally-backed, low-volatility, yield-bearing token. It offers DeFi users a comfortable middle-ground between inflationary stable coins and hyper-volatile tokens. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of The TempleDAO Protocol

Item	Description
Issuer	TempleDAO
Website	https://templedao.link/
Type	EVM Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	April 28, 2022

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

- https://github.com/TempleDAO/temple.git (23f0872)

1.2 About PeckShield

PeckShield Inc. [10] 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

		High	*Medium*	*Low*
Impact	*High*	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low

Likelihood

1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [9]:

- Likelihood represents how likely a particular vulnerability is to be uncovered and exploited in the wild;
- Impact measures the technical loss and business damage of a successful attack;
- Severity demonstrates the overall criticality of the risk.

Likelihood and impact are categorized into three ratings: *H*, *M* and *L*, i.e., *high*, *medium* and *low* respectively. Severity is determined by likelihood and impact and can be classified into four categories accordingly, i.e., *Critical*, *High*, *Medium*, *Low* shown in Table 1.2.

To evaluate the risk, we go through a list of check items and each would be labeled with a severity category. For one check item, if our tool or analysis does not identify any issue, the contract is considered safe regarding the check item. For any discovered issue, we might further deploy contracts on our private testnet and run tests to confirm the findings. If necessary, we would

Table 1.3: The Full List of Check Items

Category	Check Item
Basic Coding Bugs	Constructor Mismatch
	Ownership Takeover
	Redundant Fallback Function
	Overflows & Underflows
	Reentrancy
	Money-Giving Bug
	Blackhole
	Unauthorized Self-Destruct
	Revert DoS
	Unchecked External Call
	Gasless Send
	Send Instead Of Transfer
	Costly Loop
	(Unsafe) Use Of Untrusted Libraries
	(Unsafe) Use Of Predictable Variables
	Transaction Ordering Dependence
	Deprecated Uses
Semantic Consistency Checks	Semantic Consistency Checks
Advanced DeFi Scrutiny	Business Logics Review
	Functionality Checks
	Authentication Management
	Access Control & Authorization
	Oracle Security
	Digital Asset Escrow
	Kill-Switch Mechanism
	Operation Trails & Event Generation
	ERC20 Idiosyncrasies Handling
	Frontend-Contract Integration
	Deployment Consistency
	Holistic Risk Management
Additional Recommendations	Avoiding Use of Variadic Byte Array
	Using Fixed Compiler Version
	Making Visibility Level Explicit
	Making Type Inference Explicit
	Adhering To Function Declaration Strictly
	Following Other Best Practices

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) [8], which is a community-developed list of software weakness types to better delineate and organize weaknesses around concepts frequently encountered in software development. Though some categories used in CWE-699 may not be relevant in smart contracts, we use the CWE categories in Table 1.4 to classify our findings.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the `TempleDAO` protocol implementation. During the first phase of our audit, we study the smart contract source code and run our in-house static code analyzer through the codebase. The purpose here is to statically identify known coding bugs, and then manually verify (reject or confirm) issues reported by our tool. We further manually review business logics, examine system operations, and place DeFi-related aspects under scrutiny to uncover possible pitfalls and/or bugs.

Severity	# of Findings	
Critical	0	
High	0	
Medium	2	■ ■
Low	5	■ ■ ■ ■ ■
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 that need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 2 medium-severity vulnerabilities and 5 low-severity vulnerabilities.

Table 2.1: Key TempleDAO Audit Findings

ID	Severity	Title	Category	Status
PVE-001	Low	Proper lastUpdatedEpoch Initialization In TempleStaking	Business Logic	Resolved
PVE-002	Medium	Improper Funding Source In LockedOGTemple::lockFor()	Business Logic	Resolved
PVE-003	Low	Implicit Assumption Enforcement In AddLiquidity()	Coding Practices	Resolved
PVE-004	Low	Proper Allowance Management in Zap::mintAndStakeZapsOC()	Coding Practices	Resolved
PVE-005	Low	Accommodation of Non-ERC20-Compliant Tokens	Coding Practices	Confirmed
PVE-006	Low	Improved Validation on Function Arguments	Coding Practices	Confirmed
PVE-007	Medium	Trust on Admin Keys	Security Features	Mitigated

Besides recommending specific countermeasures to mitigate these issues, we also emphasize that it is always important to develop necessary risk-control mechanisms and make contingency plans, which may need to be exercised before the mainnet deployment. The risk-control mechanisms need to kick in at the very moment when the contracts are being deployed in mainnet. Please refer to Section 3 for details.

3 | Detailed Results

3.1 Proper lastUpdatedEpoch Initialization In TempleStaking

- ID: PVE-001
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TempleStaking
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

TempleDAO protocol has the built-in TempleStaking contract to incentivize the protocol users who stake the protocol tokens Temple. While reviewing the staking-related rewards logic, we notice the current implementation needs to be improved.

To elaborate, we show below the lastUpdatedEpoch state that, as the name indicates, keeps track of the laste updated epoch when the accumulationFactor is accumulated. It comes to our attention that this state is not initialized in the constructor() when the contract is constructed. As a result, unless the startTimestamp is properly computed, the non-initialization of lastUpdatedEpoch will bring unnecessary unprediction for the very first update of accumulationFactor!

```
    constructor(
        TempleERC20Token _TEMPLE,
        ExitQueue _EXIT_QUEUE,
        uint256 _epochSizeSeconds,
        uint256 _startTimestamp) {

        require(_startTimestamp < block.timestamp, "Start timestamp must be in the past"
            );
        require(_startTimestamp > (block.timestamp - (24 * 2 * 60 * 60)), "Start
            timestamp can't be more than 2 days in the past");

        TEMPLE = _TEMPLE;
        EXIT_QUEUE = _EXIT_QUEUE;
        // Each version of the staking contract needs it's own instance of OGTemple users can use to
        // claim back rewards
        OG_TEMPLE = new OGTemple();
        epochSizeSeconds = _epochSizeSeconds;
        startTimestamp = _startTimestamp;
        epy = ABDKMath64x64.fromUInt(1);
        accumulationFactor = ABDKMath64x64.fromUInt(1);
    }
```

Listing 3.1: `TempleStaking::constructor()`

Recommendation Improve the above-mentioned function to properly initialize the `lastUpdatedEpoch` , which needs to be aligned with the `startTimestamp`.

Status This issue has been resolved as the team considers the current setup is aligned with the `startTimestamp` initialization.

3.2 Improper Funding Source In LockedOGTemple::lockFor()

- ID: PVE-002
- Severity: Medium
- Likelihood: Low
- Impact: Medium

- Target: `LockedOGTemple`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `TempleDAO` has a key `LockedOGTemple` contract that provides the functionality of bookkeeping `OGTemple` tokens that are locked. While reviewing the current locking logic, we notice the public routine `lockFor()` needs to be revised.

To elaborate, we show below the implementation of this `lockFor()` routine. By design, this function is used to perform deposit and lock tokens for a user. This routine has a number of arguments and the first one `_staker` is the address to receive the balance. It comes to our attention that the `_staker` address is also the one to actually provide the assets, `SafeERC20.safeTransferFrom(ogTempleToken, _staker, address(this), _amountOGTemple)` (line 45). In fact, the `msg.sender` should be the one to provide the assets for locking! Otherwise, this function may be abused to lock tokens from users who have approved the locking contract before without their notice.

```
36    function lockFor(address _staker, uint256 _amountOGTemple, uint256 _unlockDelaySeconds) public {
37        LockedEntry storage lockEntry = ogTempleLocked[_staker];

39        lockEntry.amount += _amountOGTemple;
```

```
        uint256 newLockedUntilTimestamp = block.timestamp + _unlockDelaySeconds;
        if (newLockedUntilTimestamp > lockEntry.lockedUntilTimestamp) {
            lockEntry.lockedUntilTimestamp = newLockedUntilTimestamp;
        }

        SafeERC20.safeTransferFrom(ogTempleToken, _staker, address(this),
            _amountOGTemple);
        emit Lock(_staker, _amountOGTemple, lockEntry.amount, lockEntry.
            lockedUntilTimestamp);
    }
```

Listing 3.2: `LockedOGTemple::lockFor()`

Recommendation Revise the above routine to use the right funding source to transfer the assets for locking.

Status The issue has been fixed in the following deployment: 0x879B843.

3.3 Implicit Assumption Enforcement In AddLiquidity()

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

- Target: `TempleFraxAMMRouter`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

Description

The `TempleDAO` protocol has the built-in `DEX`, which provides the `TempleFraxAMMRouter` contract with normal liquidity-providing routines. For example, the `addLiquidity()` routine (see the code snippet below) is provided to add `amountADesired` amount of `tokenA` and `amountBDesired` amount of `tokenB` into the pool as liquidity. To elaborate, we show below the related code snippet.

```
    function _addLiquidity(
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin
    ) internal virtual returns (uint amountA, uint amountB) {
        (uint reserveA, uint reserveB,) = pair.getReserves();
        if (reserveA == 0 && reserveB == 0) {
            (amountA, amountB) = (amountADesired, amountBDesired);
        } else {
            uint amountBOptimal = quote(amountADesired, reserveA, reserveB);
            if (amountBOptimal <= amountBDesired) {
                require(amountBOptimal >= amountBMin, 'TempleFraxAMMRouter:
                    INSUFFICIENT_FRAX');
                (amountA, amountB) = (amountADesired, amountBOptimal);
            } else {
                uint amountAOptimal = quote(amountBDesired, reserveB, reserveA);
                assert(amountAOptimal <= amountADesired);
                require(amountAOptimal >= amountAMin, 'TempleFraxAMMRouter:
                    INSUFFICIENT_TEMPLE');
                (amountA, amountB) = (amountAOptimal, amountBDesired);
            }
        }
    }
    function addLiquidity(
        uint amountADesired,
        uint amountBDesired,
        uint amountAMin,
        uint amountBMin,
        address to,
        uint deadline
    ) external virtual ensure(deadline) returns (uint amountA, uint amountB, uint
        liquidity) {
        (amountA, amountB) = _addLiquidity(amountADesired, amountBDesired, amountAMin,
            amountBMin);
        SafeERC20.safeTransferFrom(templeToken, msg.sender, address(pair), amountA);
        SafeERC20.safeTransferFrom(fraxToken, msg.sender, address(pair), amountB);
        liquidity = pair.mint(to);
    }
```

Listing 3.3: TempleFraxAMMRouter::addLiquidity()

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

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

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

3.4 Proper Allowance Management in Zap::mintAndStakeZapsOC()

- ID: PVE-004
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: Zap
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

Description

To facilitate user interactions, the TempleDAO protocol provides the Zap contract to allow for combined operations of swap and mint. While analyzing the associated approval management, we notice one specific routine mintAndStakeZapsOC() can be improved.

In the following, we show the implementation of this routine. This routine allows to conveniently deposit the supported assets, swap to the intended stableToken, and then combine the mint and stake operations in one single transaction. The sequence of actions involves the allowance setting to facilitate the token swaps and transfers. It comes to our attention it will be helpful to properly reset the allowances to avoid unnecessary exposure of allowance to external contracts after the final transfer.

```
function mintAndStakeZapsOC(uint256 _amountTokenInMaximum, address _tokenAddress,
  uint256 _fraxAmountOut, bytes memory _path) external payable {
 require((_tokenAddress != address(0) && msg.value == 0)   (_tokenAddress ==
   address(0) && msg.value > 0), "ETH only accepted if zapping ETH.");

 uint256 _amountIn;

 if (_tokenAddress != address(0)) {
  _amountIn = _amountTokenInMaximum;
  SafeERC20.safeTransferFrom(IERC20(_tokenAddress), msg.sender, address(this),
    _amountIn);
  SafeERC20.safeIncreaseAllowance(IERC20(_tokenAddress), address(uniswapRouter),
    _amountIn);
 } else {
  _amountIn = msg.value;
 }

 ISwapRouter.ExactOutputParams memory params =
   ISwapRouter.ExactOutputParams({
     path: _path,
     recipient: address(this),
     deadline: block.timestamp,
     amountOut: _fraxAmountOut,
     amountInMaximum: _amountIn
   });
```

```
    uint256 _amountInPaid = uniswapRouter.exactOutput{ value: msg.value }(params); // Get amount of FRAX returned

    //The following will revert if _fraxAmountOut (Frax) is more than allocated amount. If we want to handle
    //the case where we proceed and refund excess Frax, the mintAndStakeFor has to be updated.
    SafeERC20.safeIncreaseAllowance(stablecToken, address(openingCeremonyContract), _fraxAmountOut);
    openingCeremonyContract.mintAndStakeFor(msg.sender, _fraxAmountOut);

    if (_amountInPaid < _amountIn) {
        uint256 excess = _amountIn - _amountInPaid;
        if (_tokenAddress != address(0)) {
          SafeERC20.safeTransfer(IERC20(_tokenAddress), msg.sender, excess);
        }
        else {
          uniswapRouter.refundETH();
          (bool success,) = msg.sender.call{ value: excess }("");
          require(success, "Refund of excess ETH failed");
        }
    }

    emit ZapComplete(msg.sender, _tokenAddress, _amountInPaid, _fraxAmountOut);
  }
```

Listing 3.4: `Zap::mintAndStakeZapsOC()`

Recommendation Properly reset the allowance after the above operations are completed.

Status The issue has been resolved as the `Zap` contract is now replaced with `TempleZap` and the above pattern does not appear in the new `TempleZap` contract.

3.5 Accommodation of Non-ERC20-Compliant Tokens

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `TempleCashback`
- Category: Coding Practices [6]
- CWE subcategory: CWE-628 [3]

Description

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

In particular, we use the popular token, i.e., ZRX, as our example. We show the related code snippet below. On its entry of transfer(), there is a check, i.e., if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]). If the check fails, it returns false. However, the transaction still proceeds successfully without being reverted. This is not compliant with the ERC20 standard and may cause issues if not handled properly. Specifically, the ERC20 standard specifies the following: *"Transfers _value amount of tokens to address _to, and MUST fire the Transfer event. The function SHOULD throw if the message caller's account balance does not have enough tokens to spend."*

```
    function transfer(address _to, uint _value) returns (bool) {
        //Default assumes totalSupply can't be over max (2^256 - 1).
        if (balances[msg.sender] >= _value && balances[_to] + _value >= balances[_to]) {
            balances[msg.sender] -= _value;
            balances[_to] += _value;
            Transfer(msg.sender, _to, _value);
            return true;
        } else { return false; }
    }

    function transferFrom(address _from, address _to, uint _value) returns (bool) {
        if (balances[_from] >= _value && allowed[_from][msg.sender] >= _value &&
            balances[_to] + _value >= balances[_to]) {
            balances[_to] += _value;
            balances[_from] -= _value;
            allowed[_from][msg.sender] -= _value;
            Transfer(_from, _to, _value);
            return true;
        } else { return false; }
    }
```

Listing 3.5: ZRX.sol

Because of that, a normal call to transfer() is suggested to use the safe version, i.e., safeTransfer(), In essence, it is a wrapper around ERC20 operations that may either throw on failure or return false without reverts. Moreover, the safe version also supports tokens that return no value (and instead revert or throw on failure). Note that non-reverting calls are assumed to be successful. Similarly, there is a safe version of approve()/transferFrom() as well, i.e., safeApprove()/safeTransferFrom().

In the following, we show the claim() routine in the TempleCashback contract. If the USDT token is supported as tokenAddress, the unsafe version of IERC20(tokenAddress).transfer(msg.sender, tokenQuantity) (line 88) 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)!

```
72      function claim(
73          bytes32 hash,
74          bytes memory signature,
75          address tokenAddress,
76          uint256 tokenQuantity,
        uint256 nonce
    ) external payable {
        require(tokenQuantity > 0, "No funds allocated");
        require(!usedNonces[msg.sender][nonce], "Hash used");
        require(_matchVerifier(hash, signature), "Invalid signature");
        require(
            generateHash(tokenAddress, msg.sender, tokenQuantity, nonce) ==
                hash,
            "Hash fail"
        );
        usedNonces[msg.sender][nonce] = true;
        IERC20(tokenAddress).transfer(msg.sender, tokenQuantity);
        emit Withdrawal(tokenAddress, msg.sender, tokenQuantity);
    }
```

Listing 3.6: `TempleCashback::claim()`

Recommendation Accommodate the above-mentioned idiosyncrasy about ERC20-related `approve()/transfer()/transferFrom()`. Note the `safeApprove()` counterpart may need to invoke twice: the first time resets the allowance to 0 and the second time sets the intended spending allowance.

Status The issue has been confirmed.

3.6 Improved Validation on Function Arguments

- ID: PVE-006
- Severity: Low
- Likelihood: Low
- Impact: Low

- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1126 [1]

Description

DeFi protocols typically have a number of system-wide parameters that can be dynamically configured on demand. The `TempleDAO` protocol is no exception. Specifically, if we examine the `ExitQueue` contract, it has defined a number of protocol-wide risk parameters, such as `maxPerEpoch` and `maxPerAddress` . In the following, we show the corresponding routines that allow for their changes.

```
    function setMaxPerEpoch(uint256 _maxPerEpoch) external onlyOwner {
        maxPerEpoch = _maxPerEpoch;
    }

    function setMaxPerAddress(uint256 _maxPerAddress) external onlyOwner {
        maxPerAddress = _maxPerAddress;
    }

    function setEpochSize(uint256 _epochSize) external onlyOwner {
        epochSize = _epochSize;
    }

    function setStartingBlock(uint256 _firstBlock) external onlyOwner {
        require(_firstBlock < firstBlock, "Can only move start block back, not forward");
        firstBlock = _firstBlock;
    }
```

Listing 3.7: ExitQueue::setMaxPerEpoch() and ExitQueue::setMaxPerAddress()

```
    function withdrawEpochs(uint256[] calldata epochs, uint256 length) external {
        uint256 totalAmount;
        for (uint i = 0; i < length; i++) {
            if (userData[msg.sender].Amount > 0) {
                uint256 amount = withdrawInternal(epochs[i], msg.sender, false);
                totalAmount += amount;
            }
        }
        SafeERC20.safeTransfer(TEMPLE, msg.sender, totalAmount);
        emit Withdrawal(msg.sender, totalAmount);
    }
```

Listing 3.8: ExitQueue::withdrawEpochs()

These parameters define various aspects of the protocol operation and maintenance and need to exercise extra care when configuring or updating them. Our analysis shows the update logic on these parameters can be improved by applying more rigorous sanity checks. Based on the current implementation, certain corner cases may lead to an undesirable consequence. For example, the above `withdrawEpochs()` can be improved by validating `require(epochs.length == length)`. Similar issues are also present in other functions, including `ExitQueue::migrate()` and `AcceleratedExitQueue::restake()/withdrawEpochs()/migrateTempleFromEpochs()`.

Recommendation Validate the given arguments to the above-mentioned functions.

Status The issue has been confirmed.

3.7 Trust Issue of Admin Keys

- ID: PVE-007
- Severity: Medium
- Likelihood: Medium
- Impact: Medium

- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

The `TempleDAO` protocol has a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., role assignment, token allocation, and parameter setting). It also has the privilege to control or govern the flow of assets among various protocol components. In the following, we examine the privileged account and the related privileged accesses in current contracts.

```
    function setAllocations(
        address[] memory _addresses,
        uint256[] memory _amounts
    ) external onlyOwner {
        require(
            _addresses.length == _amounts.length,
            "TempleTeamPayments: addresses and amounts must be the same length"
        );
        address addressZero = address(0);
        for (uint256 i = 0; i < _addresses.length; i++) {
            require(_addresses[i] != addressZero, "TempleTeamPayments: Address cannot be
                0x0");
            allocation[_addresses[i]] = _amounts[i];
        }
    }

    function setAllocation(address _address, uint256 _amount) external onlyOwner {
        require(_address != address(0), "TempleTeamPayments: Address cannot be 0x0");
        allocation[_address] = _amount;
    }

    function pauseMember(address _address)
        external
        onlyOwner
        addressExists(_address)
    {
        allocation[_address] = claimed[_address];
    }
```

Listing 3.9: Example Privileged Operations in `TempleTeamPayments`

We understand the need of the privileged functions for proper contract operations, but at the same time the extra power to these privileged accounts may also be a counter-party risk to the contract users. Therefore, we list this concern as an issue here from the audit perspective and highly recommend making these privileges explicit or raising necessary awareness among protocol users.

Recommendation Promptly transfer the `owner` privilege to the intended `DAO`-like governance contract. And activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team confirms that the owner privilege will be properly managed with the `TEMPLE` multisig.

4 Conclusion

In this audit, we have analyzed the design and implementation of the `TempleDAO` protocol, which aims to offer `DeFi` users steady-growing, low-volatility assets, and help `DAOs` re-imagine their products with gamified 'metaverse' experiences. Its first offering is the `TEMPLE` token, which is a fractionally-backed, low-volatility, yield-bearing token. It offers DeFi users a comfortable middle-ground between inflationary stable coins and hyper-volatile tokens. The current code base is clearly organized and those identified issues are promptly confirmed and fixed.

Meanwhile, we need to emphasize that `Solidity`-based smart contracts as a whole are still in an early, but exciting stage of development. To improve this report, we greatly appreciate any constructive feedbacks or suggestions, on our methodology, audit findings, or potential gaps in scope/coverage.

References

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

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

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

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

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

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

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

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

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

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