SMART CONTRACT AUDIT REPORT

for

TempleDAO Core

Prepared By: Xiaomi Huang

PeckShield

Jun 10, 2022

Document Properties

Client	TempleDAO
Title	Smart Contract Audit Report
Target	TempleDAO Core
Version	1.0
Author	Shulin Bie
Auditors	Shulin Bie, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	Jun 10, 2022	Shulin Bie	Final Release
1.0-rc	Jun 4, 2022	Shulin Bie	Release Candidate

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 Core module, 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 Core

TempleDAO aims to offer DeFi users steady-growing, low-volatility assets, and help DAOs re-imagine their products with gamified 'metaverse' experiences. Its offering TEMPLE token 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 audited TempleDAO Core module allows the user to stake their TEMPLE token to earn rewards.

Table 1.1: Basic Information of TempleDAO Core

Item	Description
Target	TempleDAO Core
Website	https://templedao.link/
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	Jun 10, 2022

In the following, we show the Git repository of reviewed files and the commit hash value used in this audit. Please note this audit only covers the protocol/contracts/core sub-directory.

- https://github.com/TempleDAO/temple/tree/core (36b0b3b)

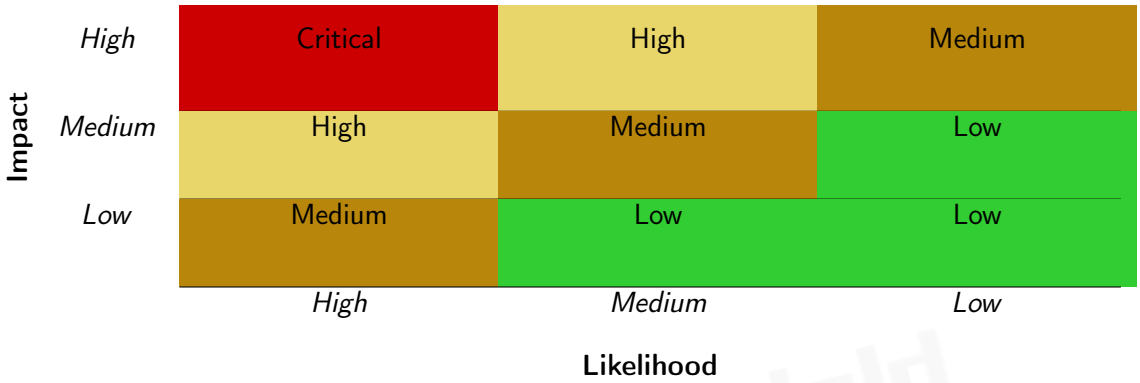
And this is the commit ID after all fixes for the issues found in the audit have been checked in:

- https://github.com/TempleDAO/temple/tree/core (0d3d0f6)

1.2 About PeckShield

PeckShield Inc. [9] 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



1.3 Methodology

To standardize the evaluation, we define the following terminology based on OWASP Risk Rating Methodology [8]:

- 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

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

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) [7], 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 Core` 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 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	1	
Low	1	
Informational	1	
Total	4	

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 1 high-severity vulnerability, 1 medium-severity vulnerability, 1 low-severity vulnerability, and 1 informational recommendation.

Table 2.1: Key TempleDAO Core Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Improper Signature Verification In Vault::withdrawFor()	Business Logic	Fixed
PVE-002	Low	Incompatibility With Deflationary/Rebasing Tokens	Business Logic	Confirmed
PVE-003	Informational	Immutable States If Only Set at Constructor()	Coding Practices	Fixed
PVE-004	Medium	Trust Issue Of Admin Keys	Security Features	Confirmed

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 Improper Signature Verification In Vault::withdrawFor()

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Vault`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

In the `TempleDAO Core` implementation, the `Vault` contract is one of the main entries for interaction with users. In particular, one entry routine, i.e., `withdrawFor()`, allows others to withdraw the deposited assets on behalf of the owner of the deposit via off-chain signature verification mechanism. While examining its logic, we observe the current signature verification should be improved.

To elaborate, we show below the related code snippet of the `Vault` contract. Within the `withdrawFor()` routine, the following statement is executed (line 103) to calculate the hash of the signature information: `bytes32 structHash = keccak256(abi.encode(WITHDRAW_FOR_TYPEHASH, owner, amount, deadline, _useNonce(owner)))`. However, it comes to our attention that the user who represents the owner of the deposit is not specified in the signature information. With that, a malicious actor has the capability to withdraw others' assets via a so-called front-running attack.

```
100     function withdrawFor(address owner, uint256 amount, uint256 deadline, uint8 v,
            bytes32 r, bytes32 s) public {
101         require(block.timestamp <= deadline, "Vault: expired deadline");
102
103         bytes32 structHash = keccak256(abi.encode(WITHDRAW_FOR_TYPEHASH, owner, amount,
                deadline, _useNonce(owner)));
104         bytes32 digest = _hashTypedDataV4(structHash);
105         address signer = ECDSA.recover(digest, v, r, s);
106
107         require(signer == owner, "Vault: invalid signature");
108
109         withdrawFor(owner, msg.sender, amount);
```

```
110     }
```

Listing 3.1: `Vault::withdrawFor()`

Recommendation Improve the signature verification mechanism in the `withdrawFor()` routine.

Status The issue has been addressed in this commit: `29beb15`.

3.2 Incompatibility With Deflationary/Rebasing Tokens

- ID: PVE-002
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `Vault/VaultProxy`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

As mentioned earlier in Section 3.1, the `Vault` contract is one of the main entries in `TempleDAO Core` for interaction with users. In particular, one entry routine, i.e., `deposit()`, accepts the deposits of the supported assets. Naturally, the contract implements a number of low-level helper routines to transfer assets in or out of the `Vault` contract. These asset-transferring routines work as expected with standard ERC20 tokens: namely the vault's internal asset balances are always consistent with actual token balances maintained in individual ERC20 token contracts.

```
178     function deposit(uint256 amount) public {
179         depositFor(msg.sender, amount);
180     }
181
182     /**
183      * @dev shared implementation of depositFor. Allows callers to deposit and lock on
             behalf of _account.
184             Care needs to be taken when calling this to ensure that the caller is
                passing the correct args in,
185             otherwise they may mistakenly lock _sender funds attributed to a wallet they
                 have no control over.
186      */
187     function depositFor(address _account, uint256 _amount) public {
188         require(inEnterExitWindow(), "Vault: Cannot join vault when outside of enter/
                exit window");
189
190         uint256 feePerTempleScaledPerHour = joiningFee.calc(firstPeriodStartTimestamp,
                periodDuration, address(this));
191         uint256 fee = _amount * feePerTempleScaledPerHour / 1e18;
192
193         require(_amount > fee, "Vault: Cannot join when fee is higher than amount");
194         uint256 amountStaked = _amount - fee;
```

```
195
196         if (_amount > 0) {
197             _mint(_account, amountStaked);
198             SafeERC20.safeTransferFrom(templeToken, msg.sender, vaultedTempleAccount,
                    _amount);
199             templeExposureToken.mint(address(this), _amount);
200         }
201
202         emit Deposit(_account, _amount, amountStaked);
203     }
```

Listing 3.2: `Vault::deposit()`

However, there exist other ERC20 tokens that may make certain customizations to their ERC20 contracts. One type of these tokens is deflationary tokens that charge certain fee for every `transfer()` or `transferFrom()`. (Another type is rebasing tokens such as `YAM`.) As a result, this may not meet the assumption behind these low-level asset-transferring routines. In other words, the above operations, such as `deposit()`, may introduce unexpected balance inconsistencies when comparing internal asset records with external ERC20 token contracts.

One possible mitigation is to measure the asset change right before and after the asset-transferring routines. In other words, instead of bluntly assuming the amount parameter in `transfer()` or `transferFrom()` will always result in full transfer, we need to ensure the increased or decreased amount in the `Vault` contract before and after the `transfer()` or `transferFrom()` is expected and aligned well with our operation. Though these additional checks cost additional gas usage, we consider they are necessary to deal with deflationary tokens or other customized ones if their support is deemed necessary.

Recommendation If current codebase needs to support possible deflationary tokens, it is better to check the balance before and after the `transfer()/transferFrom()` call to ensure the book-keeping amount is accurate. This support may bring additional gas cost.

Status The issue has been confirmed by the team. The team decides to leave it as is considering there is no need to support deflationary/rebasing token.

3.3 Immutable States If Only Set at Constructor()

- ID: PVE-003
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [5]
- CWE subcategory: CWE-561 [2]

Description

Since version 0.6.5, `Solidity` introduces the feature of declaring a state as `immutable`. An `immutable` state variable can only be assigned during contract creation, but will remain constant throughout the life-time of a deployed contract. The main benefit of declaring a state as `immutable` is that reading the state is significantly cheaper than reading from regular storage, since it is not stored in storage anymore. Instead, an `immutable` state will be directly inserted into the runtime code.

This feature is introduced based on the observation that the reading and writing of storage-based contract states are gas-expensive. Therefore, it is always preferred if we can reduce, if not eliminate, storage reading and writing as much as possible. Those state variables that are written only once are candidates of `immutable` states under the condition that each fits the pattern, i.e., "a constant, once assigned in the constructor, is read-only during the subsequent operation."

While examining all the state variables defined in the `TempleDAO Core` implementation, we observe there are several variables that need not to be updated dynamically. They can be declared as `immutable` for gas efficiency.

```
18    contract VaultProxy {
19        using ABDKMathQuad for bytes16;
20        /** @notice Tokens / Contracted required for the proxy contract  */
21        OGTemple public ogTemple;
22        TempleERC20Token public temple;
23        TempleStaking public templeStaking;
24        Faith public faith;
25
26        ...
27    }
```

Listing 3.3: `VaultProxy`

Recommendation Revisit the state variable definition and make good use of `immutable`/`constant` states.

Status The issue has been addressed in this commit: `29beb15`.

3.4 Trust Issue Of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [1]

Description

In the `TempleDAO Core` implementation, there is a privileged `owner` account that plays a critical role in governing and regulating the protocol-wide operations (e.g., configuring various system parameters). In the following, we show the representative functions potentially affected by the privilege of the `owner` account.

```
107     function addRevenue(IERC20[] memory exposureTokens, uint256[] memory amounts)
            external onlyOwner {
108         require(exposureTokens.length == amounts.length, "Exposures and amounts array
                must be the same length");
109
110         for (uint256 i = 0; i < exposureTokens.length; i++) {
111             pools[exposureTokens[i]].addRevenue(amounts[i]);
112         }
113     }
114
115     /**
116      * @notice Update mark to market of temple's various exposures
117      */
118     function updateExposureReval(IERC20[] memory exposureTokens, uint256[] memory revals
            ) external onlyOwner {
119         require(exposureTokens.length == revals.length, "Exposures and reval amounts
                array must be the same length");
120
121         for (uint256 i = 0; i < exposureTokens.length; i++) {
122             Exposure exposure = pools[exposureTokens[i]].exposure();
123             uint256 currentReval = exposure.reval();
124             if (currentReval > revals[i]) {
125                 exposure.decreaseReval(currentReval - revals[i]);
126             } else {
127                 exposure.increaseReval(revals[i] - currentReval);
128             }
129         }
130     }
```

Listing 3.4: `OpsManager::addRevenue()&&updateExposureReval()`

```
52     function withdraw(address token, address to, uint256 amount) external onlyOwner {
53         require(to != address(0), "to address zero");
54
```

```
55          if (token == address(0)) {
56              (bool sent,) = payable(to).call{value: amount}("");
57              require(sent, "send failed");
58          } else {
59              SafeERC20.safeTransfer(IERC20(token), to, amount);
60          }
61      }
```

Listing 3.5: `VaultedTemple::withdraw()`

We emphasize that the privilege assignment is indeed necessary and consistent with the protocol design. However, it is worrisome if the privileged account is a plain EOA account. A multi-sig account could greatly alleviate this concern, though it is still far from perfect. Note that a compromised privileged account would allow the attacker to modify a number of sensitive system parameters, which directly undermines the assumption of the protocol design.

Recommendation Suggest a multi-sig account to be the privileged `owner` account. Additionally, all changes to privileged operations may need to be mediated with a necessary timelock.

Status The issue has been confirmed by the team.

4 Conclusion

In this audit, we have analyzed the TempleDAO Core design and implementation. TempleDAO aims to offer DeFi users steady-growing, low-volatility assets, and help DAOs re-imagine their products with gamified 'metaverse' experiences. Its offering TEMPLE token 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 audited TempleDAO Core module allows the user to stake their TEMPLE token to earn rewards. The current code base is well structured and neatly organized. Those identified issues are promptly confirmed and addressed.

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

References

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

[2] MITRE. CWE-561: Dead Code. https://cwe.mitre.org/data/definitions/561.html.

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

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

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

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

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

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

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