SMART CONTRACT AUDIT REPORT

for

TiTi Protocol

Prepared By: Yiqun Chen

PeckShield

February 18, 2022

Document Properties

Client	TiTi Protocol
Title	Smart Contract Audit Report
Target	TiTi
Version	1.0
Author	Xuxian Jiang
Auditors	Stephen Bie, Yiqun Chen, Xuxian Jiang
Reviewed by	Yiqun Chen
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	February 18, 2022	Xuxian Jiang	Final Release
1.0-rc1	February 12, 2022	Xuxian Jiang	Release Candidate #1

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 design document and related smart contract source code of the TiTi 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 TiTi

TiTi protocol is designed to bring a new type of elastic supply algorithm stablecoin solution to DeFi and Web3. By incorporating the multi-assets-reserve mechanism, TiTi is designed to be decentralized with high capital utilization, stable efficient from risk-proof reserves and multi-asset reserves, and resistant to volatility risks. The protocol has its own stablecoin TiUSD, which is designed to be a new trading medium to meet the investment needs of different investors. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of TiTi

Item	Description
Name	TiTi Protocol
Website	https://titi.finance/
Type	Ethereum Smart Contract
Platform	Solidity
Audit Method	Whitebox
Latest Audit Report	February 18, 2022

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

- https://github.com/TiTi-Finance/TiTi-Core-Protocol.git (69bbfdb)

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

- https://github.com/TiTi-Finance/TiTi-Core-Protocol.git (418189b)

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

Impact \ Likelihood	*High*	*Medium*	*Low*
High	Critical	High	Medium
Medium	High	Medium	Low
Low	Medium	Low	Low

Likelihood

1.3 Methodology

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

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

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) [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. Moreover, in case there is an issue that may affect an active protocol that has been deployed, the public version of this report may omit such issue, but will be amended with full details right after the affected protocol is upgraded with respective fixes.

1.4 Disclaimer

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

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

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

2 Findings

2.1 Summary

Here is a summary of our findings after analyzing the implementation of the `TiTi` 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	1	■
Medium	2	■ ■
Low	1	■
Informational	1	■
Total	5	

We have so far identified a list of potential issues: some of them involve subtle corner cases that might not be previously thought of, while others refer to unusual interactions among multiple contracts. For each uncovered issue, we have therefore developed test cases for reasoning, reproduction, and/or verification. After further analysis and internal discussion, we determined a few issues of varying severities need to be brought up and paid more attention to, which are categorized in the above table. More information can be found in the next subsection, and the detailed discussions of each of them are in Section 3.

2.2 Key Findings

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

Table 2.1: Key TiTi Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Incorrect Role Initialization in TiTiToken/TiUSDToken	Business Logic	Resolved
PVE-002	Medium	Proper Burn Logic in MarketMakerFund::withdrawAll()	Business Logic	Resolved
PVE-003	Low	Simplified Logic in getReward()	Coding Practices	Resolved
PVE-004	Medium	Trust Issue of Admin Keys	Security Features	Mitigated
PVE-005	Informational	Removal Of Unused State/Code	Business Logic	Resolved

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

3 | Detailed Results

3.1 Incorrect Role Initialization in TiTiToken/TiUSDToken

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: Medium
- Target: `TiTiToken`, `TiUSDToken`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

`TiTi` protocol provides its own stablecoin `TiUSDToken` and the related token contract defines three different roles: `SNAPSHOT_ROLE`, `MINTER_ROLE`, and `DEFAULT_ADMIN_ROLE`. As their names indicate, the first role allows for the holder to activate the snapshot mechanism so that the balances and total supply can be recorded for later access; the second role is capable of minting additional tokens into circulation; and the last one manages the above two roles.

While examining these three roles, we notice the role assignment logic needs to be corrected. In particular, we show below the related `constructor()` function of `TiUSDToken`. It mistakenly sets up the wrong admin role for `SNAPSHOT_ROLE` and `MINTER_ROLE` (lines 23-24). The intended admin role should be the second parameter, instead of the first one!

```
20    constructor() ERC20("TiUSD", "TiUSD") ERC20Permit("TiUSD") {
21        _setupRole(DEFAULT_ADMIN_ROLE, msg.sender);
22        _setupRole(SNAPSHOT_ROLE, msg.sender);
23        _setRoleAdmin(DEFAULT_ADMIN_ROLE, MINTER_ROLE);
24        _setRoleAdmin(DEFAULT_ADMIN_ROLE, SNAPSHOT_ROLE);
25    }
```

Listing 3.1: `TiUSDToken::constructor()`

The same issue is also applicable to the `constructor()` and `setNewMinters()` (shown below) in the `TiTiToken` contract.

```
    function setNewMinters(address[] memory _newMinters) external onlyRole(
        DEFAULT_ADMIN_ROLE)
```

```
    {
        for (uint i; i < _newMinters.length; i++) {
            _setupRole(DEFAULT_ADMIN_ROLE, _newMinters[i]);
        }
    }
```

Listing 3.2: `TiTiToken::setNewMinters()`

Recommendation Properly assign the admin roles for the different roles in `TiTiToken` and `TiUSDToken`.

Status This issue has been fixed in the following commits: `418189b`.

3.2 Proper Burn Logic in MarketMakerFund::withdrawAll()

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MarketMakerFund`
- Category: Business Logic [7]
- CWE subcategory: CWE-841 [4]

Description

The `TiTi` protocol has a unique approach to enable decentralized single-asset yield farming, i.e., `Market Maker Fund (MMF)`. The protocol makes use of the so-called `protocol added value (PAV)` as the source of dividends to `MMF`, which allows for raising more market-making funds and hence increasing market-making depth to further reduce user transaction slippage and enhance the protocol robustness. Our analysis on the `MMF` feature shows the current withdrawal logic needs to be improved.

To illustrate, we show below the core `withdrawAll()` routine, which is designed to allow market makers to withdraw their investments and burn associated `TiTiToken`. It comes to our attention that the burnt share is assumed to be in the current contract, which only holds when the `isStaking` is configured to be `true`. In other words, when the flag `isStaking` is `false`, there is a need to call `_burn(msg.sender, allShare)`, instead of the current `_burn(address(this), allShare)` (line 136).

```
125     function withdrawAll() external onlyEOA nonReentrant whenNotPaused {
126         uint allShare = balanceOf(msg.sender);
127         if (isStaking) {
128             allShare += lpStakingPool.balanceOf(msg.sender);
129             lpStakingPool.withdraw(allShare, msg.sender);
130             lpStakingPool.getReward(msg.sender);
131         }
132 
133         reordersController.reorders();
134         uint amount = getShareValue(allShare);
```

```
        _burn(address(this), allShare);

        uint pegPrice = reordersController.pegPrice();

        uint tiusdAmount = amount * pegPrice / precisionConv;

        if (isToken0) {
            mammSwapPair.removeLiquidity(tiusdAmount, amount);
        } else {
            mammSwapPair.removeLiquidity(amount, tiusdAmount);
        }

        tiusdToken.burn(tiusdToken.balanceOf(address(this)));
        baseToken.safeTransfer(msg.sender, baseToken.balanceOf(address(this)));
        reordersController.sync();
    }
```

Listing 3.3: `MarketMakerFund::withdrawAll()`

Recommendation Revise the above withdrawal logic by burning the given share on the proper source account.

Status This issue has been fixed in the following commits: `418189b`.

3.3 Simplified Logic in getReward()

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `MMFLPStakingPool`
- Category: Coding Practices [6]
- CWE subcategory: CWE-1050 [1]

Description

The `TiTi` protocol has the familiar `Synthetix`-based `MMFLPStakingPool` contract to allow for staking users to be rewarded. Within this contract, there is a `getReward()` routine that is intended to obtain the calling user's staking rewards. The logic is rather straightforward in calculating possible reward, which, if not zero, is then allocated to the calling (staking) user.

Our examination shows that the current implementation logic can be further optimized. In particular, the `getReward()` routine has a modifier, i.e., `updateReward(msg.sender)`, which timely updates the calling user's (earned) rewards in `rewards[msg.sender]` (line 94).

```
    function getReward() external updateReward(msg.sender) checkStart {
        uint256 reward = earned(msg.sender);
```

```
185         if (reward > 0) {
186             rewards[msg.sender] = 0;
187             titi.safeTransfer(msg.sender, reward);
188             emit RewardPaid(msg.sender, reward);
189         }
190     }
```

Listing 3.4: MMFLPStakingPool::getReward()

Having the modifier `updateReward()`, there is no need to re-calculate the earned reward for the caller `msg.sender`. In other words, we can simply re-use the calculated `rewards[msg.sender]` and assign it to the `reward` variable (line 184).

```
    modifier updateReward(address account) {
        rewardPerTokenStored = rewardPerToken();
        lastUpdateTime = lastTimeRewardApplicable();
        if (account != address(0)) {
            rewards[account] = earned(account);
            userRewardPerTokenPaid[account] = rewardPerTokenStored;
        }
        _;
    }
```

Listing 3.5: MMFLPStakingPool::updateReward()

Recommendation Avoid the duplicated calculation of the caller's reward in `getReward()`, which also leads to (small) beneficial reduction of associated gas cost.

```
183     function getReward() external updateReward(msg.sender) checkStart {
184         uint256 reward = rewards[msg.sender];
185         if (reward > 0) {
186             rewards[msg.sender] = 0;
187             titi.safeTransfer(msg.sender, reward);
188             emit RewardPaid(msg.sender, reward);
189         }
190     }
```

Listing 3.6: Revised MMFLPStakingPool::getReward()

Status This issue has been fixed in the following commits: `418189b`.

3.4 Trust Issue of Admin Keys

- ID: PVE-004
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [5]
- CWE subcategory: CWE-287 [2]

Description

In the `TiTi` protocol, there is a privileged `owner` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and reward adjustment). It also has the privilege to control or govern the flow of assets managed by this protocol. Our analysis shows that the privileged account needs to be scrutinized. In the following, we examine the privileged account and the related privileged accesses in current contracts.

To elaborate, we show below the `migrate()` routine in the `MAMMSwapPair` contract. This routine allows the `owner` account to migrate all liquidity to another contract, which contains the funds from protocol users!

```
    function migrate(address _to) external nonReentrant onlyOwner {
        IERC20 _token0 = token0;
        IERC20 _token1 = token1;

        (uint112 _fund0, uint112 _fund1,) = getDepth();

        uint balance0 = _token0.balanceOf(address(this));
        uint balance1 = _token1.balanceOf(address(this));

        _token0.safeTransfer(_to, balance0);
        _token1.safeTransfer(_to, balance1);

        // Clear all status
        _update(0, 0, _fund0, _fund1);
        mmfFund0 = uint(0);
        mmfFund1 = uint(0);

        emit Migrate(_to, fund0, fund1, mmfFund0, mmfFund1);
    }
```

Listing 3.7: `MAMMSwapPair::migrate()`

Moreover, the `MAMMSwapPair` contract allows the privileged `owner` to configure various protocol parameters. It would be worrisome if the privileged `owner` account is a plain EOA account. Note that a multi-sig account could greatly alleviate this concern, though it is still far from perfect. Specifically, a better approach is to eliminate the administration key concern by transferring the role to a community-governed DAO. In the meantime, a timelock-based mechanism can also be considered as mitigation. Moreover, it should be noted if current contracts are to be deployed behind a proxy, there is a need to properly manage the proxy-admin privileges as they fall in this trust issue as well.

```
    function setNewReordersController(address _reordersController) external onlyOwner {
        address oldReorders = reordersController;
        reordersController = _reordersController;
        emit NewReordersController(oldReorders, _reordersController);
    }

    function setFeeTo(address _feeTo) external onlyOwner {
        address oldFeeTo = feeTo;
        feeTo = _feeTo;
        emit NewFeeTo(oldFeeTo, _feeTo);
    }

    function setNewMMF(address _mmf) external onlyOwner {
        require(_mmf != address(0), "MAMMSwapPair: Cannot be address(0)");
        address oldMMF = mmf;
        mmf = _mmf;
        emit NewMMF(oldMMF, _mmf);
    }

    /// @notice set a new period for the TWAP window
    function setPeriod(uint256 _period) external onlyOwner {
        require(_period != 0, "TiTiOracles: zero period");

        period = _period;
        emit TWAPPeriodUpdate(_period);
    }
```

Listing 3.8: Example Privileged Functions in `MAMMSwapPair`

Recommendation Promptly transfer the privileged account to the intended `DAO`-like governance contract. All changed to privileged operations may need to be mediated with necessary timelocks. Eventually, activate the normal on-chain community-based governance life-cycle and ensure the intended trustless nature and high-quality distributed governance.

Status This issue has been mitigated as the team clarifies the use of a timelock for admin-related operations.

3.5 Redundant State/Code Removal

- ID: PVE-005
- Severity: Informational
- Likelihood: N/A
- Impact: N/A
- Target: `Multiple Contracts`
- Category: Coding Practices [6]
- CWE subcategory: CWE-563 [3]

Description

The `TiTi` protocol makes good use of a number of reference contracts, such as `ERC20`, `SafeERC20`, `Pausable`, and `ReentrancyGuard`, to facilitate its code implementation and organization. For example, the `MarketMakerFund` smart contract has so far imported at least five reference contracts. However, we observe the inclusion of certain unused code or the presence of unnecessary redundancies that can be safely removed.

For example, if we examine closely the `MarketMakerFund` contract, it has defined a number of events to reflect runtime states. It comes to our attention that a specific event is never used, i.e., `NewOracles` (lines 46-49). Note that an unused event can be safely removed.

```
36      event NewReordersController(
37          address oldAddr,
38          address newAddr
39      );
40
41      event NewLPStakingPool(
42          address oldAddr,
43          address newAddr
44      );
45
46      event NewOracles(
47          address oldAddr,
48          address newAddr
49      );
50
51      event IsStaking(
52          bool isStaking
53      );
```

Listing 3.9: Various Events Defined in the `MarketMakerFund` Contract

Recommendation Consider the removal of the redundant state (or code) with a simplified, consistent implementation.

Status This issue has been fixed in the following commits: `418189b`.

4 Conclusion

In this audit, we have analyzed the design and implementation of the TiTi protocol, which is designed to bring a new type of elastic supply algorithm stablecoin solution to DeFi and Web3 that incorporates the Multi-Assets-Reserve mechanism. 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-1050: Excessive Platform Resource Consumption within a Loop. https://cwe.mitre.org/data/definitions/1050.html.

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

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