SMART CONTRACT AUDIT REPORT

for

Takara Protocol

Prepared By: Xiaomi Huang

PeckShield

May 6, 2025

Document Properties

Client	Takara
Title	Smart Contract Audit Report
Target	Takara
Version	1.0
Author	Xuxian Jiang
Auditors	Matthew Jiang, Xuxian Jiang
Reviewed by	Xiaomi Huang
Approved by	Xuxian Jiang
Classification	Public

Version Info

Version	Date	Author(s)	Description
1.0	May 6, 2025	Xuxian Jiang	Final Release
1.0-rc1	May 4, 2025	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 **4**

1.1 About Takara 4

1.2 About PeckShield 5

1.3 Methodology 5

1.4 Disclaimer 7

2 Findings **9**

2.1 Summary 9

2.2 Key Findings 10

3 Detailed Results **11**

3.1 Lack of Caller Validation in MultiRewardDistributor Upgrade 11

3.2 Incorrect getAggregatorData() Logic in CompositeOracle 12

3.3 Non ERC20-Compliance of TToken 14

3.4 Possibly Inaccurate Reward Disbursement in MultiRewardDistributor 16

3.5 Lack of Consistent Decimals Validation in TErc20 18

3.6 Trust Issue of Admin Keys 19

4 Conclusion **22**

References **23**

1 Introduction

Given the opportunity to review the design document and related smart contract source code of the `Takara` 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 Takara

`Takara` is a non-custodial liquidity market protocol where users can lend any supported assets, leveraging their capital to borrow other supported assets. The protocol allows users to have complete control over their funds and offers competitive interest rates without any intermediaries involved. The basic information of the audited protocol is as follows:

Table 1.1: Basic Information of Takara

Item	Description
Target	Takara
Type	EVM Smart Contract
Language	Solidity
Audit Method	Whitebox
Latest Audit Report	May 6, 2025

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

- https://github.com/Takara-Lend/Takara_Contract.git (d182a1b)

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

- https://github.com/Takara-Lend/Takara_Contract.git (f24f1e2)

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

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 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 contract is considered safe regarding the check item. For any discovered issue, we might further

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

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 implementation of the `Takara` 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	3	■ ■ ■
Low	2	■ ■
Informational	0	
Total	6	

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

2.2 Key Findings

Overall, these smart contracts are well-designed and engineered, though the implementation can be improved by resolving the identified issues (shown in Table 2.1), including 1 high-severity vulnerability, 3 medium-severity vulnerabilities, and 2 low-severity vulnerabilities.

Table 2.1: Key Takara Audit Findings

ID	Severity	Title	Category	Status
PVE-001	High	Lack of Caller Validation in MultiRewardDistributor Upgrade	Security Features	Resolved
PVE-002	Medium	Incorrect getAggregatorData() Logic in CompositeOracle	Business Logic	Resolved
PVE-003	Low	Accommodation of Non-ERC20-Compliant Tokens	Business Logic	Confirmed
PVE-004	Medium	Possibly Inaccurate Reward Disbursement in MultiRewardDistributor	Business Logic	Resolved
PVE-005	Low	Lack of Consistent Decimals Validation in TErc20	Coding Practices	Resolved
PVE-006	Medium	Trust Issue of Admin Keys	Security Features	Mitigated

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 Lack of Caller Validation in MultiRewardDistributor Upgrade

- ID: PVE-001
- Severity: High
- Likelihood: High
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

To facilitate possible future upgrade, the `Takara` protocol has a number of core contracts and each is instantiated as a proxy with its actual logic contract in the backend. While examining the related proxy upgrade logic, we notice an issue that needs to properly validate the caller for the proxy contract upgrade.

In the following, we show an example upgrade routine from the `MultiRewardDistributorV2` contract. We notice its `_authorizeUpgrade()` routine does not validate the caller. To fix, there is a need to validate the caller is authorized for the update. Specifically, the authorized entity can be the owner of the implementation contract (or the bearer of a designated role), which is authorized to call `_upgradeToAndCall()` on the proxy side and then update the implementation.

```
    function _authorizeUpgrade(address newImplementation) internal virtual override {}
```

Listing 3.1: `MultiRewardDistributorV2::_authorizeUpgrade()`

Moreover, the `Takara` protocol has a built-in oracle `PythAggregatorV3`, which exposes a public the sensitive routine `updateFeeds()` to update the prices of supported token in the protocol without validating the caller.

Recommendation Improve the above-mentioned privileged routines by properly validate the caller and manage the authorized entities.

Status This issue has been fixed in the following commit: `ba819a3`.

3.2 Incorrect getAggregatorData() Logic in CompositeOracle

- ID: PVE-002
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `CompositeOracle`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

The `Takara` protocol has a core `CompositeOracle` contract to manage critical price feeds. In the process of examining the oracle-related logic, we notice current implementation should be improved.

In the following, we show the implementation of a related routine, i.e., `getAggregatorData()`. As the name indicates, this routine is designed to retrieve the latest price feed from a configured aggregator of multiple price feeds for the same token. However, it comes to our attention that it only makes use of the first price feed (line 162).

```
157     function getAggregatorData(TToken tToken) internal view returns (uint256, uint256,
            uint256) {
158         address[] memory aggregatorAddresses = aggregators[address(tToken)];
159         require(aggregatorAddresses.length > 0, "No aggregators configured");

161         for (uint256 i = 0; i < aggregatorAddresses.length; i++) {
162             address aggregatorAddr = aggregatorAddresses[0];

164             if (l2Aggregators[aggregatorAddr]) {
165                 chainlinkL2SequencerCheck();
166             }

168             AggregatorV3Interface aggregator = AggregatorV3Interface(aggregatorAddr);
169             (bool success, bytes memory data) =
170                 aggregatorAddr.staticcall(abi.encodeWithSelector(aggregator.
                        latestRoundData.selector));

172             if (!success) {
173                 continue;
174             }

176             (, int256 answer,, uint256 updatedAt,) = abi.decode(data, (uint80, int256,
                    uint256, uint256, uint80));
177             if (answer <= 0  (block.timestamp - updatedAt) >= freshCheck) {
178                 continue;
179             }

181             uint256 rawPrice = uint256(answer);
```

```
            uint256 decimals = uint256(aggregator.decimals());
            uint256 scaledPrice = scalePrice(rawPrice, decimals, 18);
            return (rawPrice, scaledPrice, decimals);
        }

        revert NoPriceFeedAvailable();
    }
```

Listing 3.2: `CompositeOracle::getAggregatorData()`

In addition, the token price price does not differentiate the `TToken` from the underlying token. In other words, we need to take into account the associated `TToken-to-underlying` exchange rate. Moreover, the underlying token price has been normalized by multiplying with the scaling factor of `10 ** (36 - 2 * decimals)` (line 146), which needs to be revised as `10 ** (18 - decimals)`.

```
135     function getUnderlyingScaledPrice(TToken tToken) internal view returns (uint256
            price) {
136         ERC20 underlying = getUnderlying(tToken);
137         uint256 decimals = address(underlying) == address(0) ? 18 : underlying.decimals
                ();

139         uint256 feedDecimals;

141         (uint256 rawPrice,, uint256 decimals_) = getAggregatorData(tToken);
142         feedDecimals = decimals_;
143         price = scalePrice(rawPrice, feedDecimals, decimals);

145         // Multiply by 10^36 and then divide by the square of the underlying token's
                decimals
146         price = price * 10 ** (36 - 2 * decimals);
147     }
```

Listing 3.3: `CompositeOracle::getUnderlyingScaledPrice()`

Recommendation Revise the above-mentioned routines to properly provide the token price feeds.

Status This issue has been fixed in the following commit: `f24f1e2`.

3.3 Non ERC20-Compliance of TToken

- ID: PVE-003
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: TToken
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

Each asset supported by the Takara protocol is integrated through a so-called TToken contract, which is an ERC20 compliant representation of balances supplied to the protocol. By minting TTokens, users can earn interest through the TToken's exchange rate, which increases in value relative to the underlying asset, and further gain the ability to use TTokens as collateral. In the following, we examine the ERC20 compliance of these TTokens.

Table 3.1: Basic View-Only Functions Defined in The ERC20 Specification

Item	Description	Status
name()	Is declared as a public view function	✓
	Returns a string, for example "Tether USD"	✓
symbol()	Is declared as a public view function	✓
	Returns the symbol by which the token contract should be known, for example "USDT". It is usually 3 or 4 characters in length	✓
decimals()	Is declared as a public view function	✓
	Returns decimals, which refers to how divisible a token can be, from 0 (not at all divisible) to 18 (pretty much continuous) and even higher if required	✓
totalSupply()	Is declared as a public view function	✓
	Returns the number of total supplied tokens, including the total minted tokens (minus the total burned tokens) ever since the deployment	✓
balanceOf()	Is declared as a public view function	✓
	Anyone can query any address' balance, as all data on the blockchain is public	✓
allowance()	Is declared as a public view function	✓
	Returns the amount which the spender is still allowed to withdraw from the owner	✓

The ERC20 specification defines a list of API functions (and relevant events) that each token contract is expected to implement (and emit). The failure to meet these requirements means the token contract cannot be considered to be ERC20-compliant. Naturally, as part of our audit, we examine the list of API functions defined by the ERC20 specification and validate whether there exist any inconsistency or incompatibility in the implementation or the inherent business logic of the audited contract(s).

Table 3.2: Key `State-Changing` Functions Defined in The ERC20 Specification

Item	Description	Status
transfer()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the caller does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring to zero address	✓
transferFrom()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token transfer status	✓
	Reverts if the spender does not have enough token allowances to spend	×
	Updates the spender's token allowances when tokens are transferred successfully	✓
	Reverts if the from address does not have enough tokens to spend	×
	Allows zero amount transfers	✓
	Emits Transfer() event when tokens are transferred successfully (include 0 amount transfers)	✓
	Reverts while transferring from zero address	✓
	Reverts while transferring to zero address	✓
approve()	Is declared as a public function	✓
	Returns a boolean value which accurately reflects the token approval status	✓
	Emits Approval() event when tokens are approved successfully	✓
	Reverts while approving to zero address	✓
Transfer() event	Is emitted when tokens are transferred, including zero value transfers	✓
	Is emitted with the from address set to $address(0x0)$ when new tokens are generated	✓
Approval() event	Is emitted on any successful call to approve()	✓

Our analysis shows that there are several ERC20 inconsistency or incompatibility issues found in the `TToken` contract. Specifically, the current `mint()` function might emit the `Transfer` event with the contract itself as the source address. Note the ERC20 specification states that "*A token contract which creates new tokens SHOULD trigger a Transfer event with the _from address set to 0x0 when tokens are created.*" A similar issue is also present in the `transferFrom()` function.

In the surrounding two tables, we outline the respective list of basic `view-only` functions (Table 3.1) and key `state-changing` functions (Table 3.2) according to the widely-adopted ERC20 specification. In addition, we perform a further examination on certain features that are permitted by the ERC20 specification or even further extended in follow-up refinements and enhancements (e.g., ERC777/ERC2222), but not required for implementation. These features are generally helpful, but may also impact or bring certain incompatibility with current DeFi protocols. Therefore, we consider it is important to highlight them as well. This list is shown in Table 3.3.

Table 3.3: Additional `Opt-in` Features Examined in Our Audit

Feature	Description	Opt-in
Deflationary	Part of the tokens are burned or transferred as fee while on transfer()/transferFrom() calls	—
Rebasing	The balanceOf() function returns a re-based balance instead of the actual stored amount of tokens owned by the specific address	—
Pausable	The token contract allows the owner or privileged users to pause the token transfers and other operations	✓
Blacklistable	The token contract allows the owner or privileged users to blacklist a specific address such that token transfers and other operations related to that address are prohibited	✓
Mintable	The token contract allows the owner or privileged users to mint tokens to a specific address	✓
Burnable	The token contract allows the owner or privileged users to burn tokens of a specific address	✓

Recommendation Revise the `TToken` implementation to ensure its ERC20-compliance.

Status

Status This issue has been confirmed. Considering that this is part of the original `Compound` code base, the team decides to leave it as is to minimize the difference from the original `Compound` and reduce the risk of introducing bugs as a result of changing the behavior.

3.4 Possibly Inaccurate Reward Disbursement in MultiRewardDistributor

- ID: PVE-004
- Severity: Medium
- Likelihood: Medium
- Impact: Medium
- Target: `MultiRewardDistributor/V2`
- Category: Business Logic [6]
- CWE subcategory: CWE-841 [3]

Description

To incentivize the protocol users, `Takara` has a core `MultiRewardDistributor` contract to allow for multi-asset reward distribution. While examining the actual logic for reward disbursement, we notice an issue that may not accurately calculate the reward amount for a borrower.

For elaboration, we show below the implementation of the related routine `disburseBorrowerRewardsInternal()` in `MultiRewardDistributor`. While this routine has a basic logic in computing the user reward balance and then sending the reward to the user, there is also a need to ensure the user borrow balance (line 913) should be consistent with the latest `marketBorrowIndex`. In other words, we need to replace the current call of `borrowBalanceStored()` with `borrowBalanceCurrent()`. Note the same issue also affects the `getOutstandingRewardsForUser()` routine as well as the same routines in `PartnerMultiRewardDistributor`.

```
907     function disburseBorrowerRewardsInternal(TToken _tToken, address _borrower, bool
            _sendTokens) internal {
908         MarketEmissionConfig[] storage configs = marketConfigs[address(_tToken)];
909 
910         Exp memory marketBorrowIndex = Exp({mantissa: _tToken.borrowIndex()});
911         TTokenData memory tTokenData = TTokenData({
912             tTokenBalance: _tToken.balanceOf(_borrower),
913             borrowBalanceStored: _tToken.borrowBalanceStored(_borrower)
914         });
915 
916         // Iterate over all market configs and update their indexes + timestamps
917         for (uint256 index = 0; index < configs.length; index++) {
918             MarketEmissionConfig storage emissionConfig = configs[index];
919 
920             // Go calculate the total outstanding rewards for this user
921             uint256 owedRewards = calculateBorrowRewardsForUser(
922                 emissionConfig, emissionConfig.config.borrowGlobalIndex,
                        marketBorrowIndex, tTokenData, _borrower
923             );
924 
925             // Update user's index to global index
926             emissionConfig.borrowerIndices[_borrower] = emissionConfig.config.
                    borrowGlobalIndex;
927 
928             // Update the accrued borrow side rewards for this user
929             emissionConfig.borrowerRewardsAccrued[_borrower] = owedRewards;
930 
931             emit DisbursedBorrowerRewards(
932                 _tToken,
933                 _borrower,
934                 emissionConfig.config.emissionToken,
935                 emissionConfig.borrowerRewardsAccrued[_borrower]
936             );
937 
938             // If we are instructed to send out rewards, do so and update the
                    borrowerRewardsAccrued to
939             // 0 if it was successful, or to `pendingRewards` if there was insufficient
                    balance to send
940             if (_sendTokens) {
941                 // Emit rewards for this token/pair
942                 uint256 pendingRewards = sendReward(
943                     payable(_borrower),
```

```
                    emissionConfig.borrowerRewardsAccrued[_borrower],
                    emissionConfig.config.emissionToken
                );

                emissionConfig.borrowerRewardsAccrued[_borrower] = pendingRewards;
            }
        }
    }
```

Listing 3.4: `MultiRewardDistributor::disburseBorrowerRewardsInternal()`

Recommendation Accurately compute the reward amount for the borrowing users.

Status This issue has been resolved. The team confirms that rewards are designed to be passively updated and current setup meets the design needs.

3.5 Lack of Consistent Decimals Validation in TErc20

- ID: PVE-005
- Severity: Low
- Likelihood: Low
- Impact: Low
- Target: `TErc20`
- Category: Coding Practices [5]
- CWE subcategory: CWE-1126 [1]

Description

As mentioned earlier, each asset supported in `Takara` is integrated through a so-called `TToken` contract, which is an `ERC20`-compliant representation of balances supplied to the protocol. As an `ERC20` -compliant representation, `TToken` has its own `name`, `symbol`, and `decimals` fields. Our analysis shows that its `decimals` field is better consistent with the underlying token.

In particular, we show below the initialization logic of the `TToken` contract. It comes to our attention that the `name`, `symbol`, and `decimals` are directly provided by the user. With that, it is better to apply necessary sanity checks to ensure the given `decimals` is equal to the underlying token's `decimals`, i.e., `require( decimals_ == underlying.decimals());`.

```
    function initialize(
        address underlying_,
        ComptrollerInterface comptroller_,
        InterestRateModel interestRateModel_,
        uint256 initialExchangeRateMantissa_,
        string memory name_,
        string memory symbol_,
        uint8 decimals_
    ) public {
        // TToken initialize does the bulk of the work
```

```
        super.initialize(comptroller_, interestRateModel_, initialExchangeRateMantissa_,
            name_, symbol_, decimals_);

        // Set underlying and sanity check it
        underlying = underlying_;
    }
```

Listing 3.5: `TErc20::initialize()`

Recommendation Improve the above initialization routine to ensure the `TToken` contract has the same `decimals` field as the underlying token.

Status This issue has been resolved as the team confirms it is part of the design.

3.6 Trust Issue of Admin Keys

- ID: PVE-006
- Severity: Medium
- Likelihood: Low
- Impact: High
- Target: `Multiple Contracts`
- Category: Security Features [4]
- CWE subcategory: CWE-287 [2]

Description

In the `Takara` protocol, there is a privileged `admin` account that plays a critical role in governing and regulating the system-wide operations (e.g., parameter setting and marketing 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 related privileged accesses in current contracts.

```
    function _setPriceOracle(PriceOracle newOracle) public returns (uint256) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PRICE_ORACLE_OWNER_CHECK);
        }
        ...
    }

    /**
     * @notice Sets the closeFactor used when liquidating borrows
     * @dev Admin function to set closeFactor
     * @param newCloseFactorMantissa New close factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure
     */
    function _setCloseFactor(uint256 newCloseFactorMantissa) external returns (uint256)
        {
```

```
        // Check caller is admin
        require(msg.sender == admin, "only admin can set close factor");
        ...
    }

    /**
     * @notice Sets the collateralFactor for a market
     * @dev Admin function to set per-market collateralFactor
     * @param tToken The market to set the factor on
     * @param newCollateralFactorMantissa The new collateral factor, scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setCollateralFactor(TToken tToken, uint256 newCollateralFactorMantissa)
        external returns (uint256) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.
                SET_COLLATERAL_FACTOR_OWNER_CHECK);
        }
        ...
    }

    /**
     * @notice Sets liquidationIncentive
     * @dev Admin function to set liquidationIncentive
     * @param newLiquidationIncentiveMantissa New liquidationIncentive scaled by 1e18
     * @return uint 0=success, otherwise a failure. (See ErrorReporter for details)
     */
    function _setLiquidationIncentive(uint256 newLiquidationIncentiveMantissa) external
        returns (uint256) {
        // Check caller is admin
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.
                SET_LIQUIDATION_INCENTIVE_OWNER_CHECK);
        }
        ...
    }
```

Listing 3.6: Example `Setters` in the `Supervisor/Whitelist`

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.

In the meantime, the protocol makes extensive use of proxy contracts to allow for future upgrades. The upgrade is a privileged operation, which also falls in this trust issue on the admin key.

Recommendation Make the privileges explicit to the protocol users.

Status This issue has been resolved and the team plans to adopt a more decentralized approach with `Timelock` and `Governor` contracts once things stabilize.

4 | Conclusion

In this audit, we have analyzed the design and implementation of the `Takara` protocol, which is a non-custodial liquidity market protocol where users can lend any supported assets, leveraging their capital to borrow other supported assets. The protocol allows users to have complete control over th 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-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-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.