TRC25

Smart Contract Audit Report
Prepared for Tomochain

Date Issued: Sep 4, 2023
Project ID: AUDIT2023016
Version: v2.0
Confidentiality Level: Public

inspeX
CYBERSECURITY PROFESSIONAL SERVICE

Report Information

Project ID	AUDIT2023016
Version	v2.0
Client	Tomochain
Project	TRC25
Auditor(s)	Wachirawit Kanpanluk Sorawish Laovakul
Author(s)	Wachirawit Kanpanluk Sorawish Laovakul
Reviewer	Natsasit Jirathammanuwat
Confidentiality Level	Public

Version History

Version	Date	Description	Author(s)
2.0	Sep 4, 2023	Update scope	Sorawish Laovakul
1.0	Sep 1, 2023	Full report	Wachirawit Kanpanluk Sorawish Laovakul

Contact Information

Company	Inspex
Phone	(+66) 90 888 7186
Telegram	t.me/inspexco
Email	audit@inspex.co

Table of Contents

1. Executive Summary

As requested by Tomochain, Inspex team conducted an audit to verify the security posture of the TRC25 smart contracts on Aug 29, 2023 and 4 Sep, 2023. During the audit, Inspex team examined all smart contracts and the overall operation within the scope to understand the overview of TRC25 smart contracts. Static code analysis, dynamic analysis, and manual review were done in conjunction to identify smart contract vulnerabilities together with technical & business logic flaws that may be exposed to the potential risk of the platform and the ecosystem. Practical recommendations are provided according to each vulnerability found and should be followed to remediate the issue.

1.1. Audit Result

In the initial audit, Inspex found 2 high, 1 medium, and 2 info-severity issues. With the project team's prompt response in resolving the issues found by Inspex, all issues were resolved or mitigated in the reassessment. Therefore, Inspex trusts that TRC25 smart contracts have high-level protections in place to be safe from most attacks.

1.2. Disclaimer

This security audit is not produced to supplant any other type of assessment and does not guarantee the discovery of all security vulnerabilities within the scope of the assessment. However, we warrant that this audit is conducted with goodwill, professional approach, and competence. Since an assessment from one single party cannot be confirmed to cover all possible issues within the smart contract(s), Inspex suggests conducting multiple independent assessments to minimize the risks. Lastly, nothing contained in this audit report should be considered as investment advice.

2. Project Overview

2.1. Project Introduction

The provided standard enables the implementation of a consistent API for TRC25 within smart contracts. This standard offers fundamental features such as token transfers, approvals for third-party spending, and fee management to prevent misuse.

This token standard permits token holders to conduct transfers while covering gas fees with the token itself instead of the native network token. It can also facilitate gas-sponsored smart contracts as an alternative use case.

Scope Information:

Project Name	TRC25
Website	https://tomochain.com/
Smart Contract Type	Ethereum Smart Contract
Chain	Tomochain
Programming Language	Solidity
Category	Token

Audit Information:

Audit Method	Whitebox
Audit Date	Aug 29, 2023 and 4 Sep, 2023
Reassessment Date	Aug 31, 2023

The audit method can be categorized into two types depending on the assessment targets provided:

1. **Whitebox**: The complete source code of the smart contracts are provided for the assessment.
2. **Blackbox**: Only the bytecodes of the smart contracts are provided for the assessment.

2.2. Scope

The following smart contracts were audited and reassessed by Inspex in detail:

Initial Audit: (Commit: f47f4859f815528e701cc9ba1535bcaae2326612, 66cc5ac2cf544e7fd8bed2bbfa2374f821fad848)

Contract	Location (URL)
TRC25	https://github.com/tomochain/trc25/blob/f47f4859f8/contracts/TRC25.sol
TRC25Upgradable	https://github.com/tomochain/trc25/blob/f47f4859f8/contracts/TRC25Upgradable.sol
Address	https://github.com/tomochain/trc25/blob/f47f4859f8/contracts/libraries/Address.sol
SafeMath	https://github.com/tomochain/trc25/blob/f47f4859f8/contracts/libraries/SafeMath.sol
TRC25Permit	https://github.com/tomochain/trc25/blob/66cc5ac2cf/contracts/TRC25Permit.sol
ECDSA	https://github.com/tomochain/trc25/blob/66cc5ac2cf/contracts/libraries/ECDSA.sol
EIP712	https://github.com/tomochain/trc25/blob/66cc5ac2cf/contracts/libraries/EIP712.sol

Reassessment: (Commit: d4ee09e322b41a4e316e45a4f372f70a10846246)

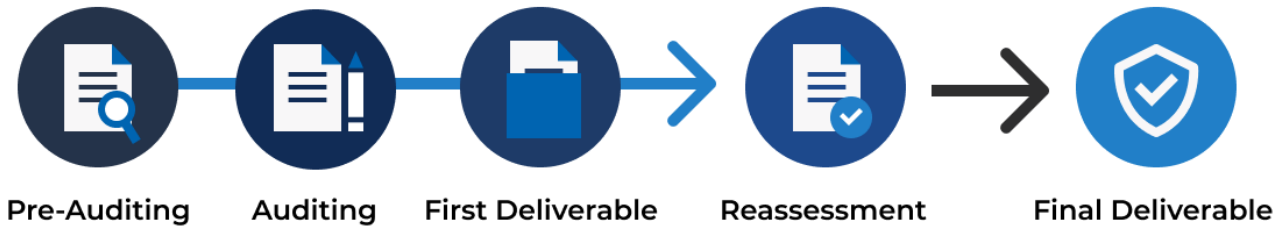
Contract	Location (URL)
TRC25	https://github.com/tomochain/trc25/blob/d4ee09e322/contracts/TRC25.sol
TRC25Upgradable	https://github.com/tomochain/trc25/blob/d4ee09e322/contracts/TRC25Upgradable.sol
Address	https://github.com/tomochain/trc25/blob/d4ee09e322/contracts/libraries/Address.sol
SafeMath	https://github.com/tomochain/trc25/blob/d4ee09e322/contracts/libraries/SafeMath.sol

The assessment scope covers only the in-scope smart contracts and the smart contracts that they inherit from.

3. Methodology

Inspex conducts the following procedure to enhance the security level of our clients' smart contracts:

1. **Pre-Auditing**: Getting to understand the overall operations of the related smart contracts, checking for readiness, and preparing for the auditing
2. **Auditing**: Inspecting the smart contracts using automated analysis tools and manual analysis by a team of professionals
3. **First Deliverable and Consulting**: Delivering a preliminary report on the findings with suggestions on how to remediate those issues and providing consultation
4. **Reassessment**: Verifying the status of the issues and whether there are any other complications in the fixes applied
5. **Final Deliverable**: Providing a full report with the detailed status of each issue



3.1. Test Categories

Inspex smart contract auditing methodology consists of both automated testing with scanning tools and manual testing by experienced testers. We have categorized the tests into 3 categories as follows:

1. **General Smart Contract Vulnerability (General)** - Smart contracts are analyzed automatically using static code analysis tools for general smart contract coding bugs, which are then verified manually to remove all false positives generated.
2. **Advanced Smart Contract Vulnerability (Advanced)** - The workflow, logic, and the actual behavior of the smart contracts are manually analyzed in-depth to determine any flaws that can cause technical or business damage to the smart contracts or the users of the smart contracts.
3. **Smart Contract Best Practice (Best Practice)** - The code of smart contracts is then analyzed from the development perspective, providing suggestions to improve the overall code quality using standardized best practices.

3.2. Audit Items

The testing items checked are based on our Smart Contract Security Testing Guide (SCSTG) v1.0 (https://github.com/InspexCo/SCSTG/releases/download/v1.0/SCSTG_v1.0.pdf) which covers most prevalent risks in smart contracts. The latest version of the document can also be found at https://inspex.gitbook.io/testing-guide/.

The following audit items were checked during the auditing activity:

Testing Category	Testing Items
1. Architecture and Design	1.1. Proper measures should be used to control the modifications of smart contract logic 1.2. The latest stable compiler version should be used 1.3. The circuit breaker mechanism should not prevent users from withdrawing their funds 1.4. The smart contract source code should be publicly available 1.5. State variables should not be unfairly controlled by privileged accounts 1.6. Least privilege principle should be used for the rights of each role
2. Access Control	2.1. Contract self-destruct should not be done by unauthorized actors 2.2. Contract ownership should not be modifiable by unauthorized actors 2.3. Access control should be defined and enforced for each actor roles 2.4. Authentication measures must be able to correctly identify the user 2.5. Smart contract initialization should be done only once by an authorized party 2.6. tx.origin should not be used for authorization
3. Error Handling and Logging	3.1. Function return values should be checked to handle different results 3.2. Privileged functions or modifications of critical states should be logged 3.3. Modifier should not skip function execution without reverting
4. Business Logic	4.1. The business logic implementation should correspond to the business design 4.2. Measures should be implemented to prevent undesired effects from the ordering of transactions 4.3. msg.value should not be used in loop iteration
5. Blockchain Data	5.1. Result from random value generation should not be predictable 5.2. Spot price should not be used as a data source for price oracles 5.3. Timestamp should not be used to execute critical functions 5.4. Plain sensitive data should not be stored on-chain 5.5. Modification of array state should not be done by value 5.6. State variable should not be used without being initialized

Testing Category	Testing Items
6. External Components	6.1. Unknown external components should not be invoked 6.2. Funds should not be approved or transferred to unknown accounts 6.3. Reentrant calling should not negatively affect the contract states 6.4. Vulnerable or outdated components should not be used in the smart contract 6.5. Deprecated components that have no longer been supported should not be used in the smart contract 6.6. Delegatecall should not be used on untrusted contracts
7. Arithmetic	7.1. Values should be checked before performing arithmetic operations to prevent overflows and underflows 7.2. Explicit conversion of types should be checked to prevent unexpected results 7.3. Integer division should not be done before multiplication to prevent loss of precision
8. Denial of Services	8.1. State changing functions that loop over unbounded data structures should not be used 8.2. Unexpected revert should not make the whole smart contract unusable 8.3. Strict equalities should not cause the function to be unusable
9. Best Practices	9.1. State and function visibility should be explicitly labeled 9.2. Token implementation should comply with the standard specification 9.3. Floating pragma version should not be used 9.4. Builtin symbols should not be shadowed 9.5. Functions that are never called internally should not have public visibility 9.6. Assert statement should not be used for validating common conditions

3.3. Risk Rating

OWASP Risk Rating Methodology (https://owasp.org/www-community/OWASP_Risk_Rating_Methodology) is used to determine the severity of each issue with the following criteria:

- **Likelihood**: a measure of how likely this vulnerability is to be uncovered and exploited by an attacker
- **Impact**: a measure of the damage caused by a successful attack

Both likelihood and impact can be categorized into three levels: **Low**, **Medium**, and **High**.

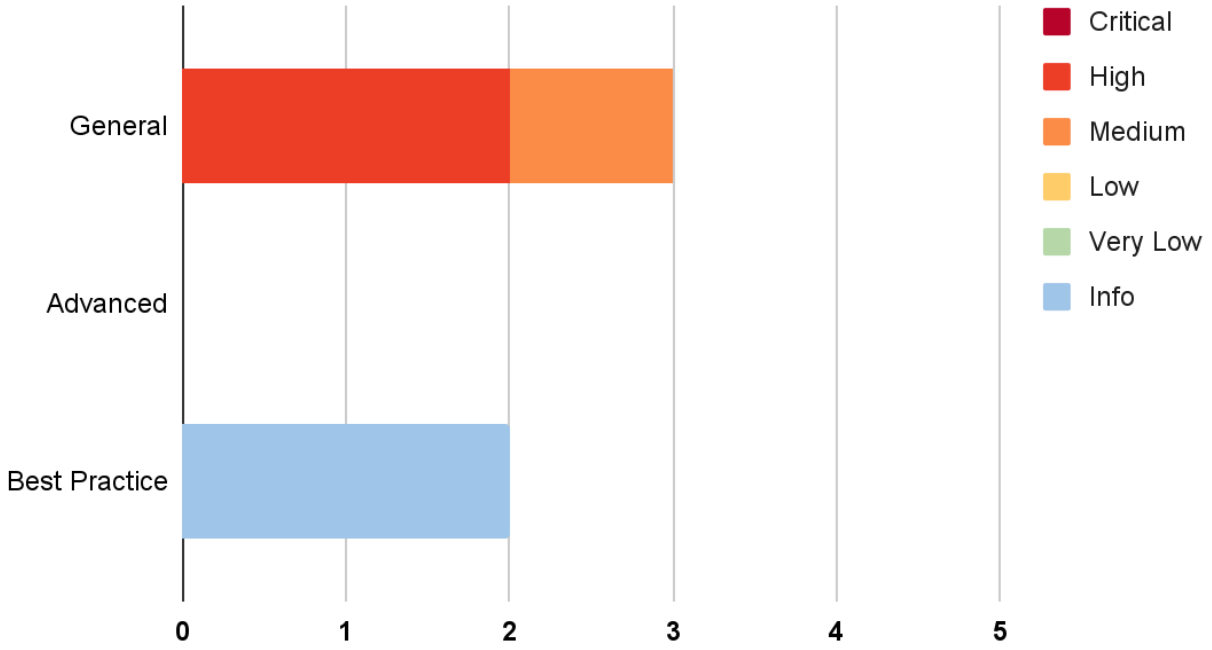
Severity is the overall risk of the issue. It can be categorized into five levels: **Very Low**, **Low**, **Medium**, **High**, and **Critical**. It is calculated from the combination of likelihood and impact factors using the matrix below. The severity of findings with no likelihood or impact would be categorized as **Info**.

Impact \ Likelihood	Low	Medium	High
Low	Very Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	Critical

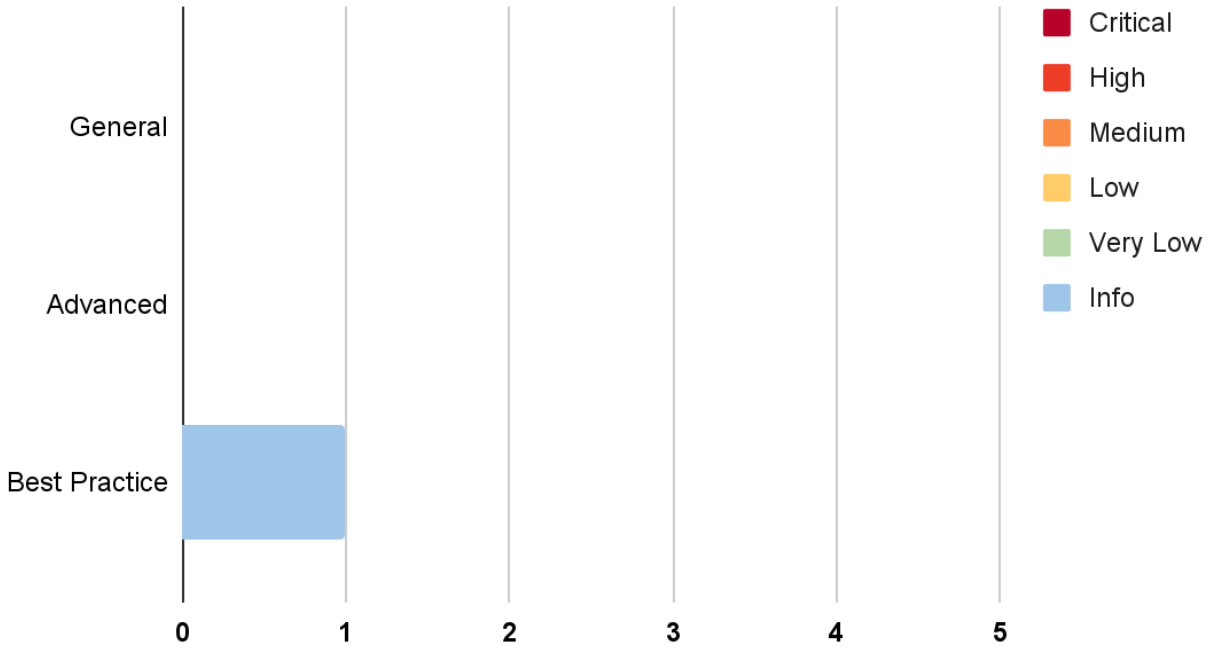
4. Summary of Findings

The following charts show the number of the issues found during the assessment and the issues acknowledged in the reassessment, categorized into three categories: **General**, **Advanced**, and **Best Practice**.

Assessment:



Reassessment:



The statuses of the issues are defined as follows:

Status	Description
Resolved	The issue has been resolved and has no further complications.
Resolved *	The issue has been resolved with mitigations and clarifications. For the clarification or mitigation detail, please refer to Chapter 5.
Acknowledged	The issue's risk has been acknowledged and accepted.
No Security Impact	The best practice recommendation has been acknowledged.

The information and status of each issue can be found in the following table:

ID	Title	Category	Severity	Status
IDX-001	Centralized Control of State Variable	General	High	Resolved *
IDX-002	Token Manual Minting by Contract Owner	General	High	Resolved
IDX-003	Integer Overflow	General	Medium	Resolved
IDX-004	Inexplicit Solidity Compiler Version	Best Practice	Info	No Security Impact
IDX-005	Improper Function Visibility	Best Practice	Info	Resolved

* The mitigations or clarifications by Tomochain can be found in Chapter 5.

5. Detailed Findings Information

5.1. Centralized Control of State Variable

ID	IDX-001
Target	TRC25 TRC25Upgradable
Category	General Smart Contract Vulnerability
CWE	CWE-284: Improper Access Control
Risk	**Severity:** High **Impact:** High The controlling authorities can change the critical state variables to gain additional profit. Thus, it is unfair to the other users. **Likelihood:** Medium There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved * Due to the TRC25 contract being an abstract base requiring inheritance by other contracts, the team has added the `virtual` modifier into the privilege functions. This modification enables contracts that inherit from it to override the default implementation and incorporate their distinct mechanisms into these functions in commit `d4ee09e322b41a4e316e45a4f372f70a10846246`. Nevertheless, the effect of the privileged functions is contingent upon the specific implementation choices made by the inheriting contracts that override these functions.

5.1.1. Description

Critical state variables can be updated at any time by the controlling authorities. Changes in these variables can cause impacts to the users, so the users should accept or be notified before these changes are effective.

However, there is currently no constraint to prevent the authorities from modifying these variables without notifying the users.

The controllable privileged state update functions are as follows:

Target	Function	Modifier
TRC25.sol(L:227)	transferOwnership()	onlyOwner
TRC25.sol(L:237)	setFee()	onlyOwner
TRC25Upgradable.sol(L:230)	transferOwnership()	onlyOwner
TRC25Upgradable.sol(L:240)	setFee()	onlyOwner

5.1.2. Remediation

In the ideal case, the critical state variables should not be modifiable to keep the integrity of the smart contract. However, if modifications are needed, Inspex suggests implementing a community-run smart contract governance to control the use of these functions.

If implementing the smart contract governance is not possible, Inspex suggests mitigating the risk of this issue by using a timelock mechanism to delay the changes for a reasonable amount of time at least 24 hours.

5.2. Token Manual Minting by Contract Owner

ID	IDX-002
Target	TRC25 TRC25Upgradable
Category	General Smart Contract Vulnerability
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity: High** **Impact: High** The contract owner can arbitrarily mint the affected tokens. **Likelihood: Medium** There is nothing to restrict the changes from being done; however, this action can only be done by the contract owner.
Status	Resolved The team has resolved this issue by removing the `mint()` function in commit `d4ee09e322b41a4e316e45a4f372f70a10846246`.

5.2.1. Description

In the TRC25 contract, the `mint()` function can be called to mint the token to any address as desired.

TRC25.sol

```
911 function mint(address recipient, uint256 amount) public onlyOwner returns
    (bool) {
912       _mint(recipient, amount);
913       return true;
914 }
```

The absence of restrictions on changes, solely controlled by the contract owner, can result in the owner's ability to freely mint tokens. This situation can lead to a lack of trust and raise worries about token inflation.

The owner's freely mint function is as follows:

Target	Function	Modifier
TRC25.sol(L:193)	mint()	onlyOwner
TRC25Upgradable.sol(L:196)	mint()	onlyOwner

5.2.2. Remediation

Inspex suggests removing the `mint()` function in both contracts or implementing the max supply in the `constructor()` function and modifying the `_mint()` function, for example:

TRC25.sol

```
32  uint256 private _maxSupply;
33
34  constructor(string memory name, string memory symbol, uint8 decimals_, uint256
    maxSupply) {
35        _name = name;
36        _symbol = symbol;
37        _decimals = decimals_;
38        _owner = msg.sender;
39        _maxSupply = maxSupply;
40  }
```

```
296 function _mint(address to, uint256 amount) internal {
297       require(to != address(0), "TRC25: mint to the zero address");
298       require(_totalSupply.add(amount) <= _maxSupply, "TRC25: mint amount
    exceeds max supply");
299       _totalSupply = _totalSupply + amount;
300       _balances[to] = _balances[to] + amount;
301       emit Transfer(address(0), to, amount);
302 }
```

5.3. Integer Overflow

ID	IDX-003
Target	TRC25 TRC25Upgradable
Category	General Smart Contract Vulnerability
CWE	CWE-190: Integer Overflow or Wraparound
Risk	**Severity:** Medium **Impact:** High The token balance can overflow in the minting process, resulting in an unexpected loss of funds. **Likelihood:** Low The `mint()` function can only be used by the contract owner.
Status	Resolved The team has resolved this issue by using the safe operation functions from the SafeMath library in commit `d4ee09e322b41a4e316e45a4f372f70a10846246`.

5.3.1. Description

In the `TRC25` contract, the `mint()` function can be called by the `onlyOwner` to mint the token to any address as desired.

TRC25.sol

```
193 function mint(address recipient, uint256 amount) public onlyOwner returns
    (bool) {
194     _mint(recipient, amount);
195     return true;
196 }
```

```
296 function _mint(address to, uint256 amount) internal {
297     require(to != address(0), "TRC25: mint to the zero address");
298     _totalSupply = _totalSupply + amount;
299     _balances[to] = _balances[to] + amount;
300     emit Transfer(address(0), to, amount);
301 }
```

The caller determines the `amount` parameter, which creates a potential risk of surpassing the `type(uint256).max` value. If the contract is compiled with Solidity version below 0.8.0, this will result in an overflow, which will lead to a loss of authority and a potential balance overflow for users.

5.3.2. Remediation

Inspex suggests using Solidity compiler version greater than 0.8.0 or using the functions from the SafeMath library instead of mathematical operators.

5.4. Inexplicit Solidity Compiler Version

ID	IDX-004
Target	TRC25 TRC25Upgradable Address SafeMath TRC25Permit ECDSA EIP712
Category	Smart Contract Best Practice
CWE	CWE-1104: Use of Unmaintained Third Party Components
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	No Security Impact The team has acknowledged this issue. However, the inexplicit compiler version has no direct impact.

5.4.1. Description

The Solidity compiler versions declared in the smart contracts were not explicit. Each compilation may be done using different compiler versions, which may potentially result in compatibility issues.

TRC25.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity >=0.7.0;
```

The affected contracts are listed in the table below:

Contract	Version
Address	>=0.7.0
SafeMath	>=0.7.0 <0.8.0
TRC25	>=0.7.0
TRC25Upgradable	>=0.7.0
TRC25Permit	>=0.6.2

ECDSA	>=0.6.2
EIP712	>=0.6.2

5.4.2. Remediation

Inspex suggests fixing the Solidity compiler to the latest stable version. At the time of the audit, the latest stable version of Solidity compiler in major 0.7 is v0.7.6.

TRC25.sol

```
1 // SPDX-License-Identifier: MIT
2 pragma solidity 0.7.6;
```

5.5. Improper Function Visibility

ID	IDX-005
Target	TRC25 TRC25Upgradable
Category	Smart Contract Best Practice
CWE	CWE-710: Improper Adherence to Coding Standards
Risk	**Severity:** Info **Impact: None** **Likelihood: None**
Status	Resolved The team has resolved this issue by changing the functions' visibility to external in commit `d4ee09e322b41a4e316e45a4f372f70a10846246`.

5.5.1. Description

Public functions that are never called internally by the contract itself should have external visibility. This improves the readability of the contract, allowing clear distinction between functions that are externally used and functions that are also called internally.

The following source code shows that the `setFee()` function of the TRC25 contract is set to public and it is never called from any internal function.

TRC25.sol

```
237 function setFee(uint256 fee) public onlyOwner {
238     _minFee = fee;
239     emit FeeUpdated(fee);
240 }
```

The following table contains all functions that have public visibility and are never called from any internal function.

Target	Function
TRC25.sol(L:193)	mint()
TRC25.sol(L:201)	burn()
TRC25.sol(L:214)	acceptOwnership()
TRC25.sol(L:227)	transferOwnership()
TRC25.sol(L:237)	setFee()
TRC25Upgradable.sol(L:196)	mint()
TRC25Upgradable.sol(L:204)	burn()
TRC25Upgradable.sol(L:217)	acceptOwnership()
TRC25Upgradable.sol(L:230)	transferOwnership()
TRC25Upgradable.sol(L:240)	setFee()

5.5.2. Remediation

Inspex suggests changing all functions' visibility to external if they are not called from any internal function as shown in the following example:

TRC25.sol

```
237 function setFee(uint256 fee) external onlyOwner {
238     _minFee = fee;
239     emit FeeUpdated(fee);
240 }
```

6. Appendix

6.1. About Inspex

Inspex is formed by a team of cybersecurity experts highly experienced in various fields of cybersecurity. We provide blockchain and smart contract professional services at the highest quality to enhance the security of our clients and the overall blockchain ecosystem.

Follow Us On:

Website	https://inspex.co
Twitter	@InspexCo
Facebook	https://www.facebook.com/InspexCo
Telegram	@inspex_announcement

inspeX

CYBERSECURITY PROFESSIONAL SERVICE