Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project:	TRON (TRX)
Website:	tron.network
Platform:	Binance Network
Language:	Solidity
Date:	April 15th, 2025

Table of Contents

Introduction

As part of EtherAuthority's community smart contract audit initiatives, the smart contract of the TRON Token from tron.network was audited. The audit was performed using manual analysis and automated software tools. This report presents all the findings regarding the audit performed on April 15th, 2025.

The purpose of this audit was to address the following:

- Ensure that all claimed functions exist and function correctly.
- Identify any security vulnerabilities that may be present in the smart contract.

Project Background

The TRX contract is an **ERC20 token** with **meta-transaction support** and **role-based minting** designed for compatibility with platforms like **Polygon (PoS Bridge)**. It enables gasless transactions for users and controlled minting through a trusted predicate contract.

Key Features:

- **ERC20 Token:** Inherits standard ERC20 functionality with name and symbol configuration during deployment.
- **Meta-Transaction Support (via NativeMetaTransaction):** Enables users to sign transactions off-chain and have relayers submit them, allowing **gasless interactions**.
  - Implements **EIP-712 typed data signatures**
  - Prevents replay with a **nonce** per user
  - Relayers call executeMetaTransaction(...), executing the signed function
- **Sender Context Resolution (ContextMixin):** Overrides _msgSender() to accurately determine the actual sender in meta-transaction calls by extracting the sender address from msg.data.
- **Access Control (AccessControlMixin):** Extends OpenZeppelin's AccessControl to include:
  - Custom error messages with contract ID
  - A PREDICATE_ROLE used to restrict minting
- **Minting via IMintableERC20:** Implements a mint function restricted to addresses

with PREDICATE_ROLE, typically assigned to Polygon’s bridge contracts.

Audit scope

Name	**Code Review and Security Analysis Report for TRON (TRX) Token Smart Contract**
Platform	**Binance Network**
File	TRX.sol
File Smart Contract Code	0xce7de646e7208a4ef112cb6ed5038fa6cc6b12e3
Audit Date	April 15th, 2025

Claimed Smart Contract Features

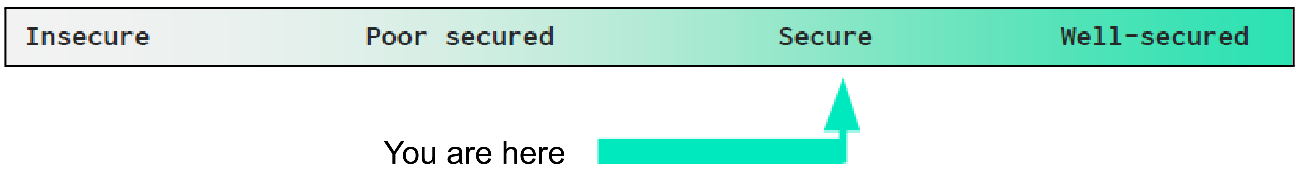
Claimed Feature Details	Our Observation
Tokenomics: • Name: TRON • Symbol: TRX	YES, This is valid.
Key Features: **ERC20 Token Standard** • Fully implements the ERC20 token standard. • Supports basic token functions such as transferring, balancing, and minting. **Meta-Transaction Support** • Gasless Transactions: Allows users to interact with the contract without paying gas fees directly by leveraging relayers. • EIP-712 Signing: Users sign meta-transactions off-chain, and relayers submit them with signatures for verification. • Nonces: Prevents replay attacks by maintaining a nonce for each user, ensuring each meta-transaction is unique. • Meta-Transaction Execution: Relayers can execute functions on behalf of users through the `executeMetaTransaction()` function. **Access Control with Role Management**.	YES, This is valid.

• The `PREDICATE_ROLE` role is required for minting tokens, ensuring that only trusted addresses (e.g., a bridge contract) can mint new tokens. • Custom error messages for role-based access control to improve clarity. **Minting Functionality** • Implements the `IMintableERC20` interface to support minting. • Only addresses with the `PREDICATE_ROLE` can mint new tokens. **Sender Context Resolution (via `ContextMixin`)** • Overwrites the `_msgSender()` function to properly resolve the actual sender's address in meta-transactions, even when relayers are involved. • Handles relayer-based interactions where `msg.sender` is not the actual user but the relayer. **SafeMath** • Uses `SafeMath` to prevent overflows and underflows, ensuring safe arithmetic operations (important for Solidity versions prior to 0.8.x). **EIP-712 Compatibility** • Integrates with EIP-712 for structured and secure signature generation, enabling gasless transactions.	

● Allows users to sign transactions off-chain with a domain separator, ensuring safe and verifiable signing of meta-transactions. **Event Emission** ● MetaTransactionExecuted: Emits an event every time a meta-transaction is successfully executed, capturing the user address, relayer address, and the function signature.	

Audit Summary

According to the standard audit assessment, the Customer's solidity-based smart contract is **“Secured.”** This token contract does not have any ownership control, hence it is **100% decentralized**.



We used various tools like Slither, Solhint, and Remix IDE. At the same time, this finding is based on a critical analysis of the manual audit.

All issues found during automated analysis were manually reviewed, and applicable vulnerabilities are presented in the Audit overview section. The general overview is presented in the AS-IS section, and all identified issues can be found in the Audit overview section.

We found 0 critical, 0 high, 0 medium, 1 low, and 2 very low-level issues.

Investors' Advice: A Technical audit of the smart contract does not guarantee the ethical nature of the project. Any owner-controlled functions should be executed by the owner with responsibility. All investors/users are advised to do their due diligence before investing in the project.

Technical Quick Stats

Main Category	Subcategory	Result
Contract Programming	Solidity version not specified	Passed
	Solidity version too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack of check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Moderated
	Human/contract checks bypass	Passed
	Random number generation/use vulnerability	N/A
	Fallback function misuse	Passed
	Race condition	Passed
	Logical vulnerability	Passed
	Features claimed	Passed
	Other programming issues	Moderated
Code Specification	Function visibility not explicitly declared	Passed
	Var. storage location not explicitly declared	Passed
	Use keywords/functions to be deprecated	Passed
	Unused code	Passed
Gas Optimization	“Out of Gas” Issue	Passed
	High consumption ‘for/while’ loop	Passed
	High consumption ‘storage’ storage	Passed
	Assert() misuse	Passed
Business Risk	The maximum limit for mintage is not set	Passed
	“Short Address” Attack	Passed
	“Double Spend” Attack	Passed

Overall Audit Result: PASSED

Code Quality

This audit scope has 1 smart contract. Smart contracts contain Libraries, Smart contracts, inheritance, and Interfaces. This is a compact and well-written smart contract.

The libraries in TRON (TRX) are part of its logical algorithm. A library is a different type of smart contract that contains reusable code. Once deployed on the blockchain (only once), it is assigned a specific address, and its properties/methods can be reused many times by other contracts in the TRX Token.

The EtherAuthority team lacks scenario and unit test scripts, which would have helped to determine the integrity of the code automatically.

Code parts are well commented on in the smart contract. Ethereum's NatSpec commenting style is recommended.

Documentation

We were given a TRX Token smart contract code in the form of a BSCscan web link.

As mentioned above, code parts are well commented on. And the logic is straightforward. So it is easy to quickly understand the programming flow as well as complex code logic. Comments are very helpful in understanding the overall architecture of the protocol.

Use of Dependencies

As per our observation, the libraries used in this smart contract infrastructure that is based on well-known industry standard open-source projects.

Apart from libraries, its functions are not used in external smart contract calls.

AS-IS overview

TRX.sol

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Hardcoded Role Address	Refer Audit Findings
2	mint	external	Missing Event for Mint Function	Refer Audit Findings
3	_msgSender	internal	Passed	No Issue
4	name	read	Passed	No Issue
5	symbol	read	Passed	No Issue
6	decimals	read	Passed	No Issue
7	totalSupply	read	Passed	No Issue
8	balanceOf	read	Passed	No Issue
9	transfer	write	Passed	No Issue
10	allowance	read	Passed	No Issue
11	approve	write	Passed	No Issue
12	transferFrom	write	Passed	No Issue
13	increaseAllowance	write	Passed	No Issue
14	decreaseAllowance	write	Passed	No Issue
15	_transfer	internal	Passed	No Issue
16	_mint	internal	Passed	No Issue
17	_burn	internal	Passed	No Issue
18	_approve	internal	Passed	No Issue
19	_setupDecimals	internal	Passed	No Issue
20	_beforeTokenTransfer	internal	Passed	No Issue
21	only	modifier	Passed	No Issue
22	_setupContractId	internal	Passed	No Issue
23	executeMetaTransaction	write	Reentrancy Risk in Meta-Transaction Execution	Refer Audit Findings
24	hashMetaTransaction	internal	Passed	No Issue
25	getNonce	read	Passed	No Issue
26	verify	internal	Passed	No Issue
27	msgSender	internal	Passed	No Issue

Severity Definitions

Risk Level	Description
Critical	Critical vulnerabilities are usually straightforward to exploit and can lead to token loss, etc.
High	High-level vulnerabilities are difficult to exploit; however, they also have a significant impact on smart contract execution, e.g., public access to crucial
Medium	Medium-level vulnerabilities are important to fix; however, they can't lead to tokens being lost
Low	Low-level vulnerabilities are mostly related to outdated, unused, etc. code snippets, which can't have a significant impact on execution
Lowest / Code Style / Best Practice	Lowest-level vulnerabilities, code style violations, and info statements can't affect smart contract execution and can be ignored.

Audit Findings

Critical Severity

No Critical severity vulnerabilities were found.

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Missing Event for Mint Function:

The `mint()` function does not emit a custom event (though the ERC20 `_mint()` emits a `Transfer` event).

Resolution: Consider emitting a `Mint(address indexed to, uint256 amount)` event for explicitness and better tracking of mint events.

Very Low / Informational / Best practices:

(1) Reentrancy Risk in Meta-Transaction Execution:

The `executeMetaTransaction()` function uses low-level `.call()` to invoke functions on the same contract:

```
(bool success, bytes memory returnData) = address(this).call(
    abi.encodePacked(functionSignature, userAddress)
);
```

This can lead to reentrancy vulnerabilities if the called function is not reentrancy-safe.

Resolution:

- Use the `nonReentrant` modifier (from OpenZeppelin's `ReentrancyGuard`) on `executeMetaTransaction()`.
- Audit all internally callable functions for reentrancy safety.
- Prefer using `delegatecall` for internal meta-transactions where possible and safe.

(2) Hardcoded Role Address:

The `PREDICATE_ROLE` is assigned to a hardcoded address:

```
constructor(string memory name_, string memory symbol_)
        public
        ERC20(name_, symbol_)
    {
        _setupContractId("TRX");
        _setupRole(PREDICATE_ROLE, 0xCa266910d92a313E5F9eb1AfFC462bcbb7d9c4A9);
        _initializeEIP712(name_);

}
```

This lacks flexibility and could pose a risk if the address becomes compromised or needs replacement.

Resolution:

- Allow role assignment via a constructor argument or external admin-only function.
- Assign role management to `DEFAULT_ADMIN_ROLE` or use `grantRole()` at deployment time.

Centralization Risk

The TRX Token smart contract does not have any ownership control, hence it is 100% decentralized.

Therefore, there is **no** centralization risk.

Conclusion

We were given a contract code in the form of bscscan web links. We have used all possible tests based on the given objects as files. We observed 1 low and 2 informational issues in the smart contract, and those issues are not critical. So, **it's good to go for production**.

Since possible test cases can be unlimited for such smart contract protocols, we provide no such guarantee of future outcomes. We have used all the latest static tools and manual observations to cover the maximum possible test cases to scan everything.

Smart contracts within the scope were manually reviewed and analyzed with static analysis tools. Smart Contract's high-level description of functionality was presented in the As-is overview section of the report.

The audit report contains all found security vulnerabilities and other issues in the reviewed code.

The security state of the reviewed smart contract, based on the standard audit procedure scope, is **"Secured".**

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review:

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis:

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's website to get a high-level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation.

Documenting Results:

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early, even if they are later shown not to represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this, we analyze the feasibility of an attack in a live system.

Suggested Solutions:

We search for immediate mitigations that live deployments can take, and finally, we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation are an ongoing collaborative process after we deliver our report, and before the details are made public.

Disclaimers

EtherAuthority.io Disclaimer

EtherAuthority team has analyzed this smart contract under the best industry practices at the date of this report, concerning: cybersecurity vulnerabilities and issues in smart contract source code, the details of which are disclosed in this report, (Source Code); the Source Code compilation, deployment and functionality (performing the intended functions).

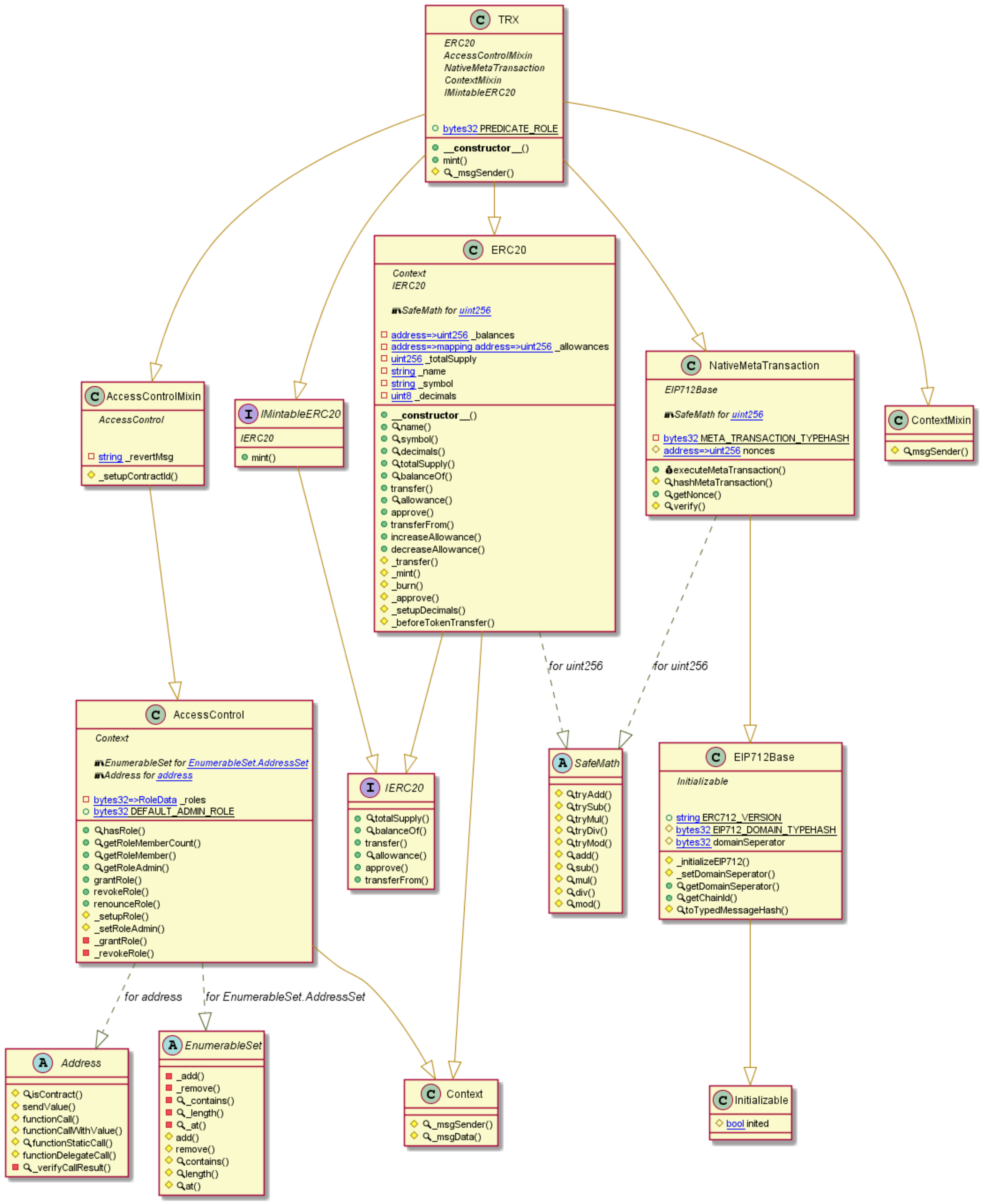
Since the total number of test cases is unlimited, the audit makes no statements or warranties on the security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bug-free status, or any other statements of the contract. While we have done our best in conducting the analysis and producing this report, it is important to note that you should not rely on this report only. We also suggest conducting a bug bounty program to confirm the high level of security of this smart contract.

Technical Disclaimer

Smart contracts are deployed and executed on the blockchain platform. The platform, its programming language, and other software related to the smart contract can have their own vulnerabilities that can lead to hacks. Thus, the audit can't guarantee explicit security of the audited smart contracts.

Appendix

Code Flow Diagram - TRON (TRX) Token



Slither Results Log

Slither is a Solidity static analysis framework that uses vulnerability detectors, displays contract details, and provides an API for writing custom analyses. It helps developers identify vulnerabilities, improve code comprehension, and prototype custom analyses quickly. The analysis includes a report with warnings and errors, allowing developers to quickly prototype and fix issues.

We analyzed the project altogether. Below are the results.

Slither Log >> TRX.sol

```
INFO:Detectors:
Contract locking ether found:
	Contract TRX (TRX.sol#1619-1653) has payable functions:
	 - NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8)
(TRX.sol#774-808)
	But does not have a function to withdraw the ether
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#contracts-that-lock-ether
INFO:Detectors:
3 different versions of Solidity are used:
	- Version constraint >=0.6.0<0.8.0 is used by:
		->=0.6.0<0.8.0 (TRX.sol#9)
		->=0.6.0<0.8.0 (TRX.sol#36)
		->=0.6.0<0.8.0 (TRX.sol#116)
		->=0.6.0<0.8.0 (TRX.sol#333)
		->=0.6.0<0.8.0 (TRX.sol#880)
		->=0.6.0<0.8.0 (TRX.sol#1372)
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-us
ed
INFO:Detectors:
Redundant expression "this (TRX.sol#27)" inContext (TRX.sol#21-30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
INFO:Detectors:
executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) should be declared external:
	- NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8)
(TRX.sol#774-808)
Moreover, the following function parameters should change its data location:
functionSignature location should be calldata
Reference:
https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-decl
ared-external
INFO:Slither:TRX.sol analyzed (14 contracts with 93 detectors), 22 result(s) found
```

Solidity Static Analysis

Static code analysis is used to identify many common coding problems before a program is released. It involves examining the code manually or using tools to automate the process. Static code analysis tools can automatically scan the code without executing it.

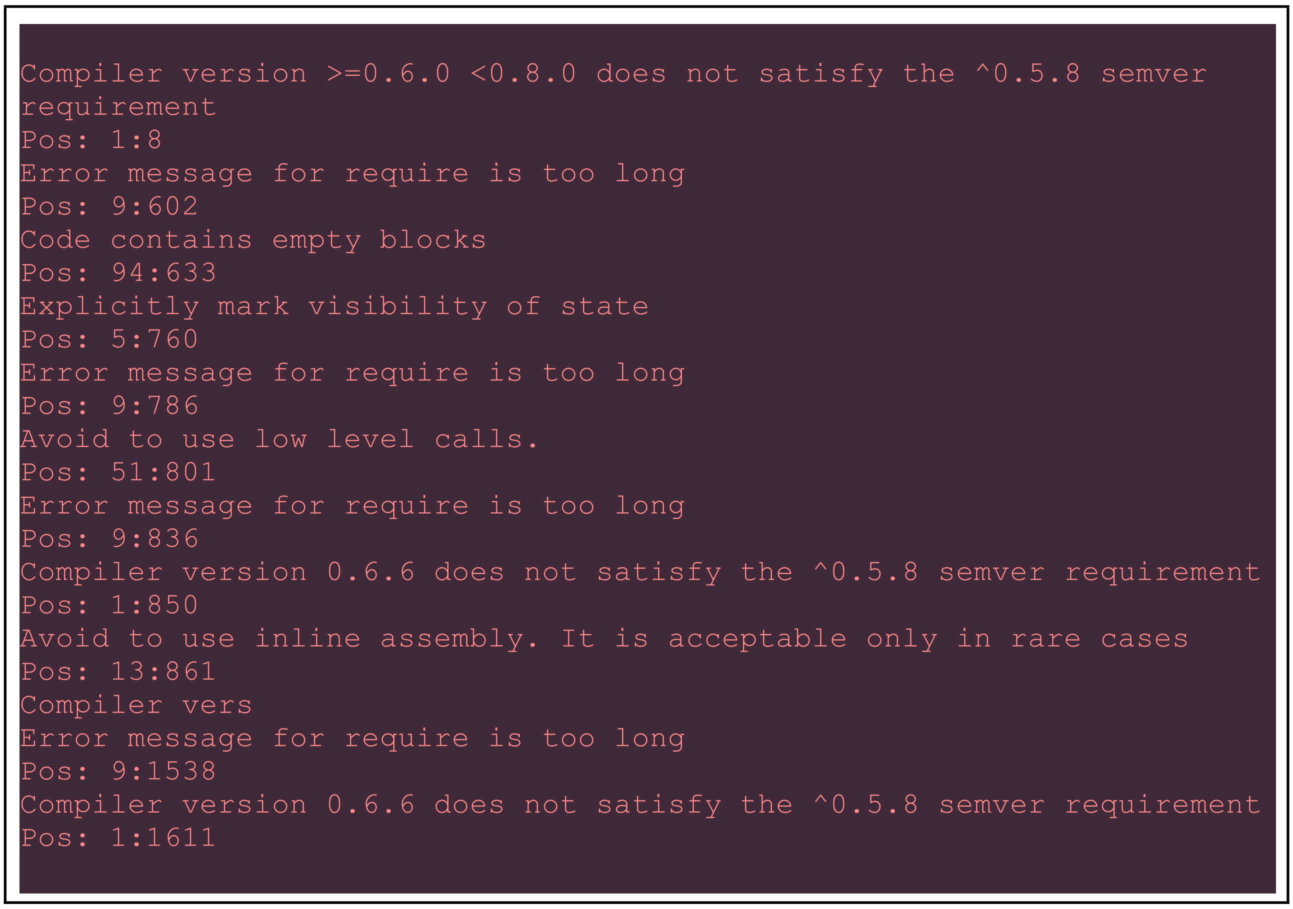
TRX.sol

```
Check-effects-interaction:
Potential violation of Checks-Effects-Interaction pattern in
Address.functionCallWithValue(address,bytes,uint256,string): Could potentially lead to
re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
Pos: 1291:4:

Inline assembly:
The Contract uses inline assembly, this is only advised in rare cases. Additionally static analysis
modules do not parse inline Assembly, this can lead to wrong analysis results.
Pos: 1357:16:

Low level calls:
Use of "delegatecall": should be avoided whenever possible. External code, that is called can
change the state of the calling contract and send ether from the caller's balance. If this is wanted
behaviour, use the Solidity library feature if possible.
Pos: 1344:50:

Gas costs:
Gas requirement of function TRX.mint is infinite: If the gas requirement of a function is higher
than the block gas limit, it cannot be executed. Please avoid loops in your functions or actions
that modify large areas of storage (this includes clearing or copying arrays in storage)
Pos: 1640:4:

Guard conditions:
Use "assert(x)" if you never ever want x to be false, not in any circumstance (apart from a bug in
your code). Use "require(x)" if x can be false, due to e.g. invalid input or a failing external
component.
Pos: 1599:8:

Delete from dynamic array:
Using "delete" on an array leaves a gap. The length of the array remains the same. If you want to
remove the empty position you need to shift items manually and update the "length" property.
Pos: 975:12:
```

Solhint Linter

Solhint Linters are the utility tools that analyze the given source code and report programming errors, bugs, and stylistic errors. For the Solidity language, there are some linter tools available that a developer can use to improve the quality of their Solidity contracts.

TRX.sol



```
Compiler version >=0.6.0 <0.8.0 does not satisfy the ^0.5.8 semver
requirement
Pos: 1:8
Error message for require is too long
Pos: 9:602
Code contains empty blocks
Pos: 94:633
Explicitly mark visibility of state
Pos: 5:760
Error message for require is too long
Pos: 9:786
Avoid to use low level calls.
Pos: 51:801
Error message for require is too long
Pos: 9:836
Compiler version 0.6.6 does not satisfy the ^0.5.8 semver requirement
Pos: 1:850
Avoid to use inline assembly. It is acceptable only in rare cases
Pos: 13:861
Compiler vers
Error message for require is too long
Pos: 9:1538
Compiler version 0.6.6 does not satisfy the ^0.5.8 semver requirement
Pos: 1:1611
```

Software analysis result:

This software reported many false positive results and some are informational issues. So, those issues can be safely ignored.

EtherAuthority

Email: audit@EtherAuthority.io