Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: TRON (TRX) Token
Website: tron.network
Platform: Ethereum
Language: Solidity
Date: May 13th, 2024

Table of contents

Email: audit@EtherAuthority.io

`

THIS IS SECURITY AUDIT REPORT DOCUMENT AND WHICH MAY CONTAIN INFORMATION WHICH IS CONFIDENTIAL. WHICH INCLUDES ANY POTENTIAL VULNERABILITIES AND MALICIOUS CODES WHICH CAN BE USED TO EXPLOIT THE SOFTWARE. THIS MUST BE REFERRED INTERNALLY AND ONLY SHOULD BE MADE AVAILABLE TO THE PUBLIC AFTER ISSUES ARE RESOLVED.

Email: audit@EtherAuthority.io

Introduction

As part of EtherAuthority's community smart contracts audit initiatives, the TRON token smart contract from tron.network was audited extensively. The audit has been performed using manual analysis as well as using automated software tools. This report presents all the findings regarding the audit performed on May 13th, 2024.

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 contract that also implements the `IMintableERC20` interface. Let's break down its key features and functionalities:
    - **Contract Inheritance:** It inherits from:
        - **ERC20:** This is the standard ERC20 token contract implementation providing basic token functionalities.
        - **`AccessControlMixin`:** This mixin provides role-based access control functionalities.
        - **`NativeMetaTransaction`:** This mixin enables meta-transactions using the EIP712 standard.
    - **Constructor:** It takes two parameters `name_` and `symbol_` and initializes the ERC20 token with the provided name and symbol.
    - **Role Definition:** It defines a role named `PREDICATE_ROLE` using the `bytes32` constant.
    - **Constructor Initialization:**
        - It sets up the contract ID using `_setupContractId`.
        - It sets up the `PREDICATE_ROLE` role for a specific address (likely a bridge or predicate contract).
    - **Minting Functionality:**
        - It implements the `mint` function from the `IMintableERC20` interface, allowing the designated predicate contract to mint new tokens.

        - The `mint` function is restricted to only be called by an address with the `PREDICATE_ROLE`.
    - **Message Sender Override:**
        - It overrides the `_msgSender` function from `ContextMixin` to return the actual sender of the message.

- Overall, this contract represents a token (`TRX`) with minting capabilities restricted to a designated predicate contract. This setup suggests that the token may be used in a sidechain or layer 2 scaling solution where token transfers are facilitated by a predicate contract acting as a bridge between Ethereum and the sidechain/layer 2 network.

Audit scope

Name	**Code Review and Security Analysis Report for TRON (TRX) Token Smart Contract**
Platform	**Ethereum**
File	TRX.sol
Smart Contract Code	0x50327c6c5a14dcade707abad2e27eb517df87ab5
Audit Date	May 13th, 2024

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation
Tokenomics: • Name: TRON • Symbol: TRX • Decimals: 6 • ERC712 VERSION: 1	**YES, This is valid.**
Ownership Control: • A Mint Token by the Predicate Role Owner.	**YES, This is valid.**

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contracts are **"Secured"**. Also, these contracts contain owner control, which does not make them fully 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 4 very low-level issues.

Investor 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	The solidity version is not specified	Passed
	Solidity version is too old	Moderated
	Integer overflow/underflow	Passed
	Function input parameters lack check	Passed
	Function input parameters check bypass	Passed
	Function access control lacks management	Passed
	Critical operation lacks event log	Passed
	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	Moderated
	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**

Email: audit@EtherAuthority.io

Business Risk Analysis

Category	Result
Buy Tax	0%
Sell Tax	0%
Cannot Buy	No
Cannot Sell	No
Max Tax	0%
Modify Tax	Not Detected
Fee Check	No
Is Honeypot	Not Detected
Trading Cooldown	Not Detected
Can Pause Trade?	No
Pause Transfer?	Not Detected
Max Tax?	No
Is it Anti-whale?	Not Detected
Is Anti-bot?	Not Detected
Is it a Blacklist?	Not Detected
Blacklist Check	No
Can Mint?	Yes
Is it Proxy?	Not Detected
Can Take Ownership?	Not Detected
Hidden Owner?	Not Detected
Self Destruction?	Not Detected
Auditor Confidence	High

Overall Audit Result: **PASSED**

Email: audit@EtherAuthority.io

Code Quality

This audit scope has 1 smart contract. Smart contract contains Libraries, Smart contracts, inherits and Interfaces. This is a compact and well written smart contract.

The libraries in TRX Token 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 has no scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

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

Documentation

We were given a TRX Token smart contract code in the form of an Etherscan 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 are used in this smart contract infrastructure that are 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

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Passed	No Issue
2	mint	external	Centralized risk, Access Control Vulnerability	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	Gas Optimization	Refer Audit Findings
14	decreaseAllowance	write	Gas Optimization	Refer Audit Findings
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	_setupContractId	internal	Passed	No Issue
22	only	modifier	Passed	No Issue
23	executeMetaTransaction	write	Gas Optimization	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
28	_msgSender	internal	Passed	No Issue
29	_msgData	internal	Passed	No Issue
30	initializer	modifier	Passed	No Issue
31	_initializeEIP712	internal	initializer	No Issue
32	_setDomainSeperator	internal	Passed	No Issue
33	getDomainSeperator	read	Passed	No Issue
34	getChainId	write	Passed	No Issue
35	toTypedMessageHash	internal	Passed	No Issue
36	hasRole	read	Passed	No Issue
37	getRoleMemberCount	read	Passed	No Issue
38	getRoleMember	read	Passed	No Issue

39	getRoleAdmin	read	Passed	No Issue
40	grantRole	write	Gas Optimization	Refer Audit Findings
41	revokeRole	write	Gas Optimization	Refer Audit Findings
42	renounceRole	write	Gas Optimization	Refer Audit Findings
43	_setupRole	internal	Passed	No Issue
44	_setRoleAdmin	internal	Passed	No Issue
45	_grantRole	write	Passed	No Issue
46	_revokeRole	write	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 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 lose
Low	Low-level vulnerabilities are mostly related to outdated, unused etc. code snippets, that can't have 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.

Email: audit@EtherAuthority.io

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) Access Control Vulnerability:

```
  /**
     * @dev See {IMintableERC20-mint}.
     */
    function mint(address user, uint256 amount) external override
only(PREDICATE_ROLE) {
        _mint(user, amount);
    }
```

The contract implements role-based access control (RBAC) using the PREDICATE_ROLE. If unauthorized addresses gain access to this role, they could exploit it to perform unauthorized minting, compromising the integrity of the token system.

Resolution: Grant the PREDICATE_ROLE only to highly trusted addresses, ideally using multi-signature schemes or other secure methods of assigning roles. Regularly audit and monitor the assignment of roles to prevent unauthorized access. Implement additional checks and safeguards to mitigate the impact of potential role compromise.

Very Low / Informational / Best practices:

(1) Use latest solidity version:

```
pragma solidity >=0.6.0 <0.8.0;
```

Use the latest solidity version while contract deployment to prevent any compiler version level bugs.

Resolution: Please use versions greater than 0.8.7.

(2) Language Specific:

```
pragma solidity >=0.6.0 <0.8.0;
```

The contract has an unlocked compiler version. An unlocked compiler version in the source code of the contract permits the user to compile it at or above a particular version. This, in turn, leads to differences in the generated bytecode between compilations due to differing compiler version numbers. This can lead to an ambiguity when debugging as compiler specific bugs may occur in the codebase that would be hard to identify over a span of multiple compiler versions rather than a specific one.

Resolution: We suggest that the compiler version is instead locked at the lowest version possible that the contract can be compiled at. For example, for version v0.6.6.

(3) Gas Optimization:

The public functions that are never called by the contract could be declared external.

- decreaseAllowance
- grantRole
- increaseAllowance
- renounceRole
- revokeRole
- executeMetaTransaction

Resolution: The “external“functions are more efficient than “public” functions.

(4) Centralized risk:

```
  /**
     * @dev See {IMintableERC20-mint}.
     */
    function mint(address user, uint256 amount) external override
only(PREDICATE_ROLE) {
        _mint(user, amount);
    }
```

Only PREDICATE_ROLE can mint a token.

Resolution: To make the smart contract 100% decentralized. We suggest renouncing ownership of the smart contract once its function is completed.

Centralization

This smart contract has some functions which can be executed by the Admin (Owner) only. If the admin wallet private key would be compromised, then it would create trouble. Following are Admin functions:

AccessControl.sol

- grantRole: Grants `role` to `account` can be set by the owner.
- revokeRole: Revokes `role` from `account` by the owner.
- renounceRole: Renounce Role from `account` by the owner.

TRX.sol

- mint: A Mint Token by the Predicate Role Owner

To make the smart contract 100% decentralized, we suggest renouncing ownership of the smart contract once its function is completed.

Conclusion

We were given a contract code in the form of Etherscan web links. And we have used all possible tests based on given objects as files. We had observed 1 low and 4 informational issues in the smart contracts. And those issues are not critical. So, **it's good to go for the production**.

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

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

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

Email: audit@EtherAuthority.io

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 web site 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 to not 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 is 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 in accordance with the best industry practices at the date of this report, in relation to: 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).

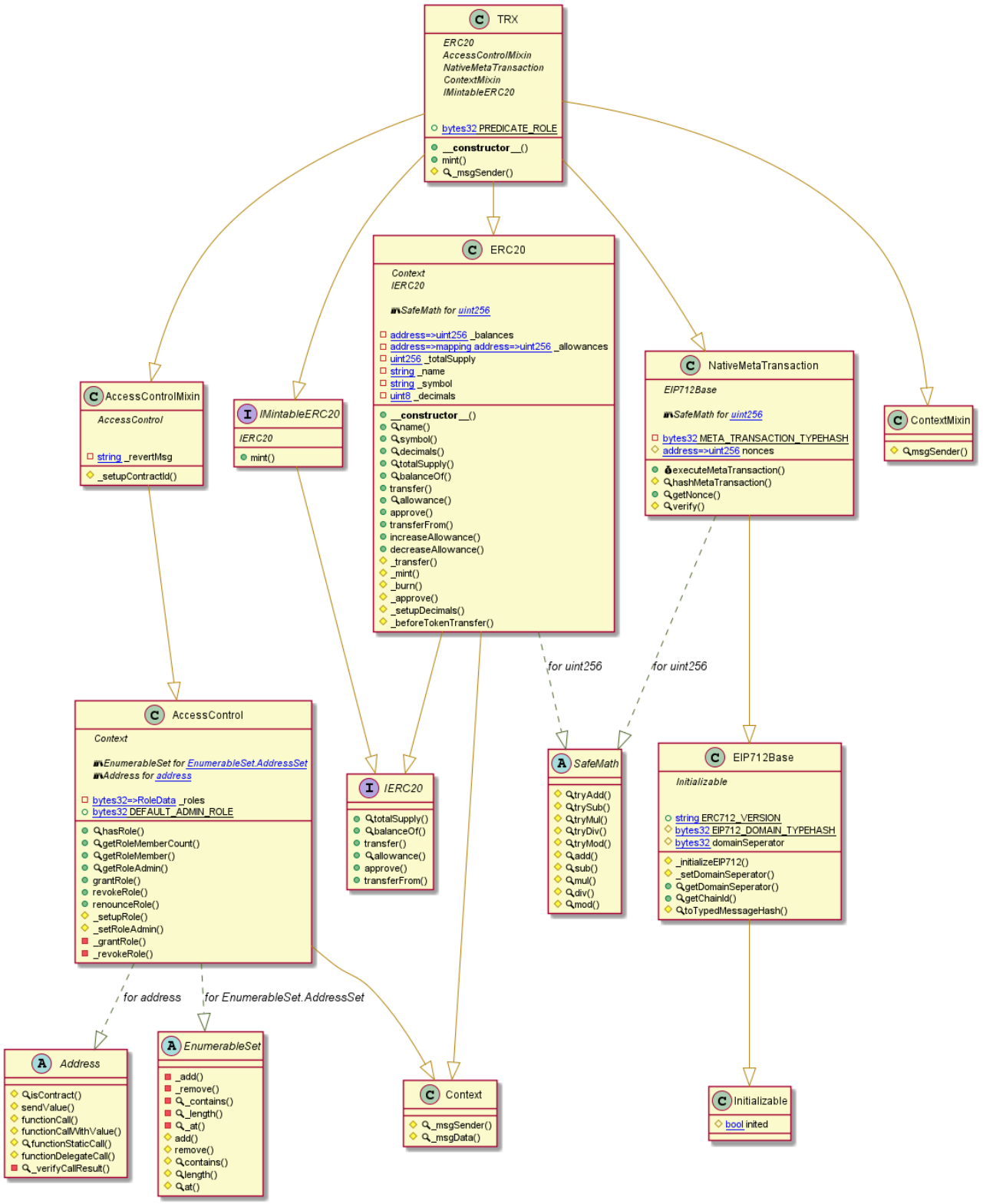
Due to the fact that the total number of test cases are unlimited, the audit makes no statements or warranties on security of the code. It also cannot be considered as a sufficient assessment regarding the utility and safety of the code, bugfree status or any other statements of the contract. While we 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 did the analysis of the project altogether. Below are the results.

Slither Log >> TRX.sol

```
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

EIP712Base.getChainId() (TRX.sol#716-722) uses assembly
        - INLINE ASM (TRX.sol#718-720)
ContextMixin.msgSender() (TRX.sol#854-873) uses assembly
        - INLINE ASM (TRX.sol#862-868)
Address.isContract(address) (TRX.sol#1203-1212) uses assembly
        - INLINE ASM (TRX.sol#1210)
Address._verifyCallResult(bool,bytes,string) (TRX.sol#1348-1365) uses assembly
        - INLINE ASM (TRX.sol#1357-1360)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage

Different versions of Solidity are used:
        - Version used: ['0.6.6', '>=0.6.0<0.8.0', '>=0.6.2<0.8.0']
        - 0.6.6 (TRX.sol#639)
        - 0.6.6 (TRX.sol#654)
        - 0.6.6 (TRX.sol#668)
        - 0.6.6 (TRX.sol#745)
        - 0.6.6 (TRX.sol#851)
        - 0.6.6 (TRX.sol#1589)
        - 0.6.6 (TRX.sol#1612)
        - >=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)
        - >=0.6.2<0.8.0 (TRX.sol#1180)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#different-pragma-directives-are-used
```

```
AccessControl._setRoleAdmin(bytes32,bytes32) (TRX.sol#1569-1572) is never used and should be removed
Address._verifyCallResult(bool,bytes,string) (TRX.sol#1348-1365) is never used and should be removed
Address.functionCall(address,bytes) (TRX.sol#1256-1258) is never used and should be removed
Address.functionCall(address,bytes,string) (TRX.sol#1266-1268) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (TRX.sol#1281-1283) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256,string) (TRX.sol#1291-1298) is never used and should be removed
Address.functionDelegateCall(address,bytes) (TRX.sol#1330-1332) is never used and should be removed
Address.functionDelegateCall(address,bytes,string) (TRX.sol#1340-1346) is never used and should be removed
Address.functionStaticCall(address,bytes) (TRX.sol#1306-1308) is never used and should be removed
Address.functionStaticCall(address,bytes,string) (TRX.sol#1316-1322) is never used and should be removed
Address.isContract(address) (TRX.sol#1203-1212) is never used and should be removed
Address.sendValue(address,uint256) (TRX.sol#1230-1236) is never used and should be removed
Context._msgData() (TRX.sol#26-29) is never used and should be removed
Context._msgSender() (TRX.sol#22-24) is never used and should be removed
ERC20._burn(address,uint256) (TRX.sol#578-586) is never used and should be removed
ERC20._setupDecimals(uint8) (TRX.sol#616-618) is never used and should be removed
EnumerableSet.add(EnumerableSet.Bytes32Set,bytes32) (TRX.sol#1024-1026) is never used and should be removed
EnumerableSet.add(EnumerableSet.UintSet,uint256) (TRX.sol#1133-1135) is never used and should be removed
EnumerableSet.at(EnumerableSet.Bytes32Set,uint256) (TRX.sol#1062-1064) is never used and should be removed
EnumerableSet.at(EnumerableSet.UintSet,uint256) (TRX.sol#1171-1173) is never used and should be removed
EnumerableSet.contains(EnumerableSet.Bytes32Set,bytes32) (TRX.sol#1041-1043) is never used and should be removed
EnumerableSet.contains(EnumerableSet.UintSet,uint256) (TRX.sol#1150-1152) is never used and should be removed
EnumerableSet.length(EnumerableSet.Bytes32Set) (TRX.sol#1048-1050) is never used and should be removed
EnumerableSet.length(EnumerableSet.UintSet) (TRX.sol#1157-1159) is never used and should be removed
EnumerableSet.remove(EnumerableSet.Bytes32Set,bytes32) (TRX.sol#1034-1036) is never used and should be removed
EnumerableSet.remove(EnumerableSet.UintSet,uint256) (TRX.sol#1143-1145) is never used and should be removed
SafeMath.div(uint256,uint256) (TRX.sol#248-251) is never used and should be removed
SafeMath.div(uint256,uint256,string) (TRX.sol#303-306) is never used and should be removed
SafeMath.mod(uint256,uint256) (TRX.sol#265-268) is never used and should be removed
SafeMath.mod(uint256,uint256,string) (TRX.sol#323-326) is never used and should be removed
SafeMath.mul(uint256,uint256) (TRX.sol#229-234) is never used and should be removed
SafeMath.sub(uint256,uint256) (TRX.sol#214-217) is never used and should be removed
SafeMath.tryAdd(uint256,uint256) (TRX.sol#137-141) is never used and should be removed
```

```
SafeMath.tryDiv(uint256,uint256) (TRX.sol#173-176) is never used and should be removed
SafeMath.tryMod(uint256,uint256) (TRX.sol#183-186) is never used and should be removed
SafeMath.tryMul(uint256,uint256) (TRX.sol#158-166) is never used and should be removed
SafeMath.trySub(uint256,uint256) (TRX.sol#148-151) is never used and should be removed
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code

Pragma version>=0.6.0<0.8.0 (TRX.sol#9) is too complex
Pragma version>=0.6.0<0.8.0 (TRX.sol#36) is too complex
Pragma version>=0.6.0<0.8.0 (TRX.sol#116) is too complex
Pragma version>=0.6.0<0.8.0 (TRX.sol#333) is too complex
Pragma version0.6.6 (TRX.sol#639) allows old versions
Pragma version0.6.6 (TRX.sol#654) allows old versions
Pragma version0.6.6 (TRX.sol#668) allows old versions
Pragma version0.6.6 (TRX.sol#745) allows old versions
Pragma version0.6.6 (TRX.sol#851) allows old versions
Pragma version>=0.6.0<0.8.0 (TRX.sol#880) is too complex
Pragma version>=0.6.2<0.8.0 (TRX.sol#1180) is too complex
Pragma version>=0.6.0<0.8.0 (TRX.sol#1372) is too complex
Pragma version0.6.6 (TRX.sol#1589) allows old versions
Pragma version0.6.6 (TRX.sol#1612) allows old versions
solc-0.6.6 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity

Low level call in NativeMetaTransaction.executeMetaTransaction(address,bytes,bytes32,bytes32,uint8) (TRX.sol#774-808):
        - (success,returnData) = address(this).call(abi.encodePacked(functionSignature,userAddress)) (TRX.sol#802-804)
Low level call in Address.sendValue(address,uint256) (TRX.sol#1230-1236):
        - (success) = recipient.call{value: amount}() (TRX.sol#1234)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (TRX.sol#1291-1298):
        - (success,returndata) = target.call{value: value}(data) (TRX.sol#1296)
Low level call in Address.functionStaticCall(address,bytes,string) (TRX.sol#1316-1322):
        - (success,returndata) = target.staticcall(data) (TRX.sol#1320)
Low level call in Address.functionDelegateCall(address,bytes,string) (TRX.sol#1340-1346):
        - (success,returndata) = target.delegatecall(data) (TRX.sol#1344)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls

Redundant expression "this (TRX.sol#27)" inContext (TRX.sol#21-30)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#redundant-statements
```

```
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-declared-external
TRX.sol analyzed (14 contracts with 84 detectors), 65 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.

more

Pos: 1232:4:

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.

more

Pos: 1298:16:

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.

more

Pos: 1285:50:

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.

more

Pos: 1285:50:

Email: audit@EtherAuthority.io

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: 1563:4:

Constant/View/Pure functions:

NativeMetaTransaction.verify(address,struct NativeMetaTransaction.MetaTransaction,bytes32,bytes32,uint8) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 787:4:

Constant/View/Pure functions:

EnumerableSet.length(struct EnumerableSet.UintSet) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 1104:4:

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.
more
Pos: 1528:8:

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.
more
Pos: 922:12:

Solhint Linter

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: 1:3
Error message for require is too long
Pos: 9:549
Error message for require is too long
Pos: 9:572
Error message for require is too long
Pos: 9:573
Code contains empty blocks
Pos: 94:604
Explicitly mark visibility of state
Pos: 5:619
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 9:677
Explicitly mark visibility of state
Pos: 5:717
Error message for require is too long
Pos: 9:743
Avoid using low level calls.
Pos: 51:758
Error message for require is too long
Pos: 9:793
Avoid using inline assembly. It is acceptable only in rare cases
Pos: 13:814
Error message for require is too long
Pos: 9:1281
Error message for require is too long
Pos: 9:1435
Error message for require is too long
Pos: 9:1450
Error message for require is too long
Pos: 9:1470
```

Software analysis result:

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

Email: audit@EtherAuthority.io