Ether Authority

www.EtherAuthority.io
audit@etherauthority.io

SMART CONTRACT

Security Audit Report

Project: Tron Staker Token
Website: tronstaker.io
Platform: Binance Smart Chain
Language: Solidity
Date: March 23rd, 2022

Table of contents

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.

Introduction

EtherAuthority was contracted by the Binance Smart Chain team to perform the Security audit of the Tron Staker Token smart contract code. 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 March 23rd, 2022.

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 TronStaker Contract is a staking smart contract having functions like invest, withdraw.

Audit scope

Name	Code Review and Security Analysis Report for Tron Staker Token Smart Contract
Platform	**BSC / Solidity**
File	TronStaker.sol
File MD5 Hash	34B92705724B433C63899EDC669D4BBD
Updated MD5 Hash	D81850F75122162B67101E9F5F71BE71
Online Code Link	https://github.com/TronStaker/TronStaker-Contract/blob/main/TronStaker.sol
Last Commit ID	2390797ecfdb5af391cbacf270e8d656435dff04
Audit Date	March 23rd, 2022
Revise Audit Date	March 24th, 2022

This is a private and confidential document. No part of this document should be disclosed to third party without prior written permission of EtherAuthority.

Email: audit@EtherAuthority.io

Claimed Smart Contract Features

Claimed Feature Detail	Our Observation

Tokenomics:
 	
● Invest Minimum Amount: 10 TRX
 	
● Project Fee: 12%
   ○ Marketing Fee: 10%
   ○ Liquidity Fee: 2%
 	
● Daily Interest Rate: 0.25%
 	
● Withdraw Fee: 10%
 	
● Percents Divider: 10000
 	
● Time Step: 1 day
 	
● Referral Percentage:
   ○ level 1: 6%
   ○ level 2: 3%
   ○ level 3: 1%
	Yes. This is valid.

Audit Summary

According to the standard audit assessment, Customer`s solidity smart contracts are **"Well Secured"**. This token contract does not contain owner control, which does make it fully decentralized.

| Insecure | Poor secured | Secure | Well-secured |

You are here

We used various tools like Slither, Solhint and Remix IDE. At the same time this finding is based on 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. General overview is presented in AS-IS section and all identified issues can be found in the Audit overview section.

We found 1 critical, 0 high, 0 medium and 1 low and some very low level issues. Critical issues have been fixed.

Investors Advice: 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	Passed
	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	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	Passed
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 not set	Passed
	"Short Address" Attack	Passed
	"Double Spend" Attack	Passed

Overall Audit Result: PASSED

Code Quality

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

The libraries in Tron Staker 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 Tron Staker Token.

The Tron Staker Token team has **not** provided scenario and unit test scripts, which would have helped to determine the integrity of the code in an automated way.

Code parts are **not well** commented on smart contracts.

Documentation

We were given a Tron Staker Token smart contract code in the form of a Github Web Link.The hash of that code is mentioned above in the table.

As mentioned above, code parts are **not well** commented. So it is not 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.

Another source of information was its official website https://tronstaker.io/ which provided rich information about the project architecture and tokenomics.

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 used in external smart contract calls.

AS-IS overview

Functions

Sl.	Functions	Type	Observation	Conclusion
1	constructor	write	Visibility for constructor is ignored, Missing required error message	Refer Audit Findings
2	invest	write	Passed	No Issue
3	withdraw	write	Passed	No Issue
4	getContractBalance	read	Passed	No Issue
5	getPlanInfo	read	Passed	No Issue
6	getPercent	read	Passed	No Issue
7	getResult	read	Passed	No Issue
8	getUserDividends	read	Passed	No Issue
9	getUserCheckpoint	read	Passed	No Issue
10	getUserReferrer	read	Passed	No Issue
11	getUserDownlineCount	read	Passed	No Issue
12	getUserReferralBonus	read	Passed	No Issue
13	getUserReferralTotalBonus	read	Passed	No Issue
14	getUserReferralWithdrawn	read	Passed	No Issue
15	getUserAvailable	read	Passed	No Issue
16	getUserAmountOfDeposits	read	Passed	No Issue
17	getUserTotalDeposits	read	Passed	No Issue
18	getUserDepositInfo	read	Passed	No Issue
19	isContract	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 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.

Audit Findings

Critical Severity

(1) Any user can invest tokens without having mentioned token:

Any user can execute the invest function as tokens have not been deducted from the user's wallet.

Resolution: We suggest correcting the code.
Status: Fixed

High Severity

No High severity vulnerabilities were found.

Medium

No Medium severity vulnerabilities were found.

Low

(1) Minimum investment amount is not as per the requirement:
Required minimum investment amount is 10 TRX but in code it is set to 1 TRX. So that is incorrect as per the feature required.

Resolution: We suggest correcting the minimum investment amount to 10 TRX.
Status: Fixed

Very Low / Informational / Best practices:

(1) Visibility for constructor is ignored:

Warning: Visibility for constructor is ignored. If you want the contract to be non-deployable, making it "abstract" is sufficient.

Resolution: We suggest removing the "public" keyword from the constructor of the TronStaker.

Status: Acknowledged

(2) SafeMath Library:

SafeMath Library is used in this contract code, but the compiler version is greater than or equal to 0.8.0, Then it will be not required to use, solidity automatically handles overflow/underflow.

Resolution: We suggest removing the SafeMath library and use normal math operators, It will improve code size, and less gas consumption.

Status: Acknowledged

(3) Multiple pragma:

There are multiple pragmas with different compiler versions.

Resolution: We suggest using only one pragma and removing the other.

Status: Acknowledged

(4) Missing error message:

```
constructor(address wallet, uint256 startDate) {
    require(!isContract(wallet));
    require(startDate > 0);
    commissionWallet = wallet;
    startUNIX = startDate;
```

There is no error message in the require statement.

Resolution: We suggest setting relevant error messages to identify the failure of the transaction.

Status: Acknowledged

(5) Variables should be immutable:

```
uint256 public startUNIX;
address public commissionWallet;
```

Variables that are defined within the constructor but further remain unchanged should be marked as immutable to save gas and to ease the reviewing process of third-parties

Resolution: Consider marking these variables as immutable.

Status: Acknowledged

(6) Unused Modifier / variables / contract:

- Contract ReentrancyGuard, Ownable, Pausable
- Modifier nonReentrant
- Variables totalRefBonus, decimalPoint.

Resolution: Remove unused modifier / variables / contracts from the code.

Status: Acknowledged

Conclusion

We were given a contract code. And we have used all possible tests based on given objects as files. We have observed some issues and some of them are critical. Critical issues have been fixed. So, **it's good to go to 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 contract, based on standard audit procedure scope, is **"Well 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 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 - TronStaker Token
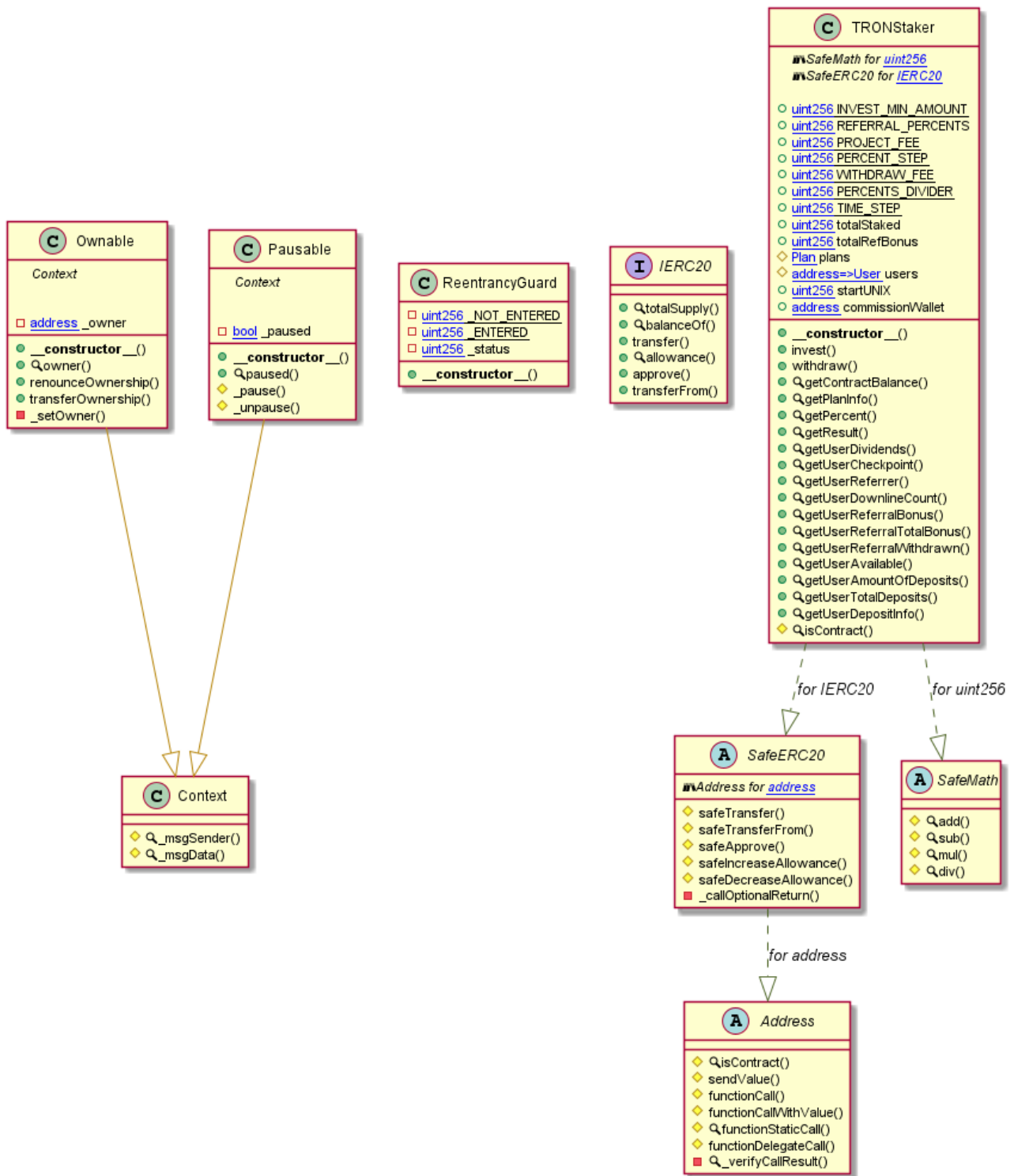
Slither Results Log

Slither log >> TronStaker.sol

```
INFO:Detectors:
Reentrancy in TRONStaker.invest(address,uint8,IERC20,uint256) (TronStaker.sol#740-812):
        External calls:
        - token.safeTransfer(commissionWallet,fee) (TronStaker.sol#751)
        State variables written after the call(s):
        - totalStaked = totalStaked.add(_betAmount) (TronStaker.sol#802)
        - user.referrer = referrer (TronStaker.sol#759)
        - users[upline].levels[i] = users[upline].levels[i].add(1) (TronStaker.sol#765)
        - users[upline_scope_0].bonus = users[upline_scope_0].bonus.add(amount) (TronStaker.sol#779)
        - users[upline_scope_0].totalBonus = users[upline_scope_0].totalBonus.add(amount) (TronStaker.sol#780-782)
        - user.checkpoint = block.timestamp (TronStaker.sol#790)
        - user.deposits.push(Deposit(plan,percent,_betAmount,profit,block.timestamp,finish)) (TronStaker.sol#798-800)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-2
INFO:Detectors:
Reentrancy in TRONStaker.invest(address,uint8,IERC20,uint256) (TronStaker.sol#740-812):
        External calls:
        - token.safeTransfer(commissionWallet,fee) (TronStaker.sol#751)
        Event emitted after the call(s):
        - FeePayed(msg.sender,fee) (TronStaker.sol#753)
        - NewDeposit(msg.sender,plan,percent,_betAmount,profit,block.timestamp,finish) (TronStaker.sol#803-811)
        - Newbie(msg.sender) (TronStaker.sol#791)
        - RefBonus(upline_scope_0,msg.sender,i_scope_1,amount) (TronStaker.sol#783)
Reentrancy in TRONStaker.withdraw(IERC20) (TronStaker.sol#814-841):
        External calls:
        - token.safeTransfer(msg.sender,totalAmount) (TronStaker.sol#838)
        Event emitted after the call(s):
        - Withdrawn(msg.sender,totalAmount) (TronStaker.sol#840)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-3
INFO:Detectors:
TRONStaker.invest(address,uint8,IERC20,uint256) (TronStaker.sol#740-812) uses timestamp for comparisons
        Dangerous comparisons:
        - users[referrer].deposits.length > 0 && referrer != msg.sender (TronStaker.sol#758)
        - upline != address(0) (TronStaker.sol#764)
        - upline_scope_0 != address(0) (TronStaker.sol#774)
TRONStaker.withdraw(IERC20) (TronStaker.sol#814-841) uses timestamp for comparisons
        Dangerous comparisons:
        - referralBonus > 0 (TronStaker.sol#822)
```

```
        - require(bool,string)(totalAmount > 0,User has no dividends) (TronStaker.sol#827)
        - contractBalance < totalAmount (TronStaker.sol#831)
TRONStaker.getPercent(uint8) (TronStaker.sol#856-867) uses timestamp for comparisons
        Dangerous comparisons:
        - block.timestamp > startUNIX (TronStaker.sol#857)
TRONStaker.getUserDividends(address) (TronStaker.sol#895-930) uses timestamp for comparisons
        Dangerous comparisons:
        - from < to (TronStaker.sol#918)
        - block.timestamp > user.deposits[i].finish (TronStaker.sol#923)
        - user.deposits[i].finish < block.timestamp (TronStaker.sol#915-917)
TRONStaker.getUserTotalDeposits(address) (TronStaker.sol#1007-1015) uses timestamp for comparisons
        Dangerous comparisons:
        - i < users[userAddress].deposits.length (TronStaker.sol#1012)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#block-timestamp
INFO:Detectors:
Address.isContract(address) (TronStaker.sol#291-301) uses assembly
        - INLINE ASM (TronStaker.sol#297-299)
Address._verifyCallResult(bool,bytes,string) (TronStaker.sol#491-511) uses assembly
        - INLINE ASM (TronStaker.sol#503-506)
TRONStaker.isContract(address) (TronStaker.sol#1039-1045) uses assembly
        - INLINE ASM (TronStaker.sol#1041-1043)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#assembly-usage
INFO:Detectors:
Address.functionCall(address,bytes) (TronStaker.sol#350-355) is never used and should be removed
Address.functionCallWithValue(address,bytes,uint256) (TronStaker.sol#382-394) is never used and should be removed
Address.functionDelegateCall(address,bytes) (TronStaker.sol#462-472) is never used and should be removed
Address.functionDelegateCall(address,bytes,string) (TronStaker.sol#480-489) is never used and should be removed
Address.functionStaticCall(address,bytes) (TronStaker.sol#426-437) is never used and should be removed
Address.functionStaticCall(address,bytes,string) (TronStaker.sol#445-454) is never used and should be removed
Address.sendValue(address,uint256) (TronStaker.sol#319-330) is never used and should be removed
Context._msgData() (TronStaker.sol#9-11) is never used and should be removed
Pausable._pause() (TronStaker.sol#130-133) is never used and should be removed
Pausable._unpause() (TronStaker.sol#142-145) is never used and should be removed
SafeERC20.safeApprove(IERC20,address,uint256) (TronStaker.sol#547-563) is never used and should be removed
SafeERC20.safeDecreaseAllowance(IERC20,address,uint256) (TronStaker.sol#581-602) is never used and should be removed
SafeERC20.safeIncreaseAllowance(IERC20,address,uint256) (TronStaker.sol#565-579) is never used and should be removed
SafeERC20.safeTransferFrom(IERC20,address,address,uint256) (TronStaker.sol#528-538) is never used and should be removed
```

```
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#dead-code
INFO:Detectors:
Pragma version^0.8.0 (TronStaker.sol#3) necessitates a version too recent to be trusted. Consider deploying with 0.6.12/0.7.6
solc-0.8.0 is not recommended for deployment
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity
INFO:Detectors:
Low level call in Address.sendValue(address,uint256) (TronStaker.sol#319-330):
        - (success) = recipient.call{value: amount}() (TronStaker.sol#325)
Low level call in Address.functionCallWithValue(address,bytes,uint256,string) (TronStaker.sol#402-418):
        - (success,returndata) = target.call{value: value}(data) (TronStaker.sol#414-416)
Low level call in Address.functionStaticCall(address,bytes,string) (TronStaker.sol#445-454):
        - (success,returndata) = target.staticcall(data) (TronStaker.sol#452)
Low level call in Address.functionDelegateCall(address,bytes,string) (TronStaker.sol#480-489):
        - (success,returndata) = target.delegatecall(data) (TronStaker.sol#487)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#low-level-calls
INFO:Detectors:
Parameter TRONStaker.invest(address,uint8,IERC20,uint256)._betAmount (TronStaker.sol#744) is not in mixedCase
Variable TRONStaker.REFERRAL_PERCENTS (TronStaker.sol#667) is not in mixedCase
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-to-solidity-naming-conventions
INFO:Detectors:
TRONStaker.totalRefBonus (TronStaker.sol#675) should be constant
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#state-variables-that-could-be-declared-constant
```

```
INFO:Detectors:
renounceOwnership() should be declared external:
        - Ownable.renounceOwnership() (TronStaker.sol#50-52)
transferOwnership(address) should be declared external:
        - Ownable.transferOwnership(address) (TronStaker.sol#58-64)
invest(address,uint8,IERC20,uint256) should be declared external:
        - TRONStaker.invest(address,uint8,IERC20,uint256) (TronStaker.sol#740-812)
withdraw(IERC20) should be declared external:
        - TRONStaker.withdraw(IERC20) (TronStaker.sol#814-841)
getContractBalance() should be declared external:
        - TRONStaker.getContractBalance() (TronStaker.sol#843-845)
getPlanInfo(uint8) should be declared external:
        - TRONStaker.getPlanInfo(uint8) (TronStaker.sol#847-854)
getUserCheckpoint(address) should be declared external:
        - TRONStaker.getUserCheckpoint(address) (TronStaker.sol#932-938)
getUserReferrer(address) should be declared external:
        - TRONStaker.getUserReferrer(address) (TronStaker.sol#940-946)
getUserDownlineCount(address) should be declared external:
        - TRONStaker.getUserDownlineCount(address) (TronStaker.sol#948-962)
getUserReferralTotalBonus(address) should be declared external:
        - TRONStaker.getUserReferralTotalBonus(address) (TronStaker.sol#972-978)
getUserReferralWithdrawn(address) should be declared external:
        - TRONStaker.getUserReferralWithdrawn(address) (TronStaker.sol#980-986)
getUserAvailable(address) should be declared external:
        - TRONStaker.getUserAvailable(address) (TronStaker.sol#988-997)
getUserAmountOfDeposits(address) should be declared external:
        - TRONStaker.getUserAmountOfDeposits(address) (TronStaker.sol#999-1005)
getUserTotalDeposits(address) should be declared external:
        - TRONStaker.getUserTotalDeposits(address) (TronStaker.sol#1007-1015)
getUserDepositInfo(address,uint256) should be declared external:
        - TRONStaker.getUserDepositInfo(address,uint256) (TronStaker.sol#1017-1037)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#public-function-that-could-be-declared-external
INFO:Slither:TronStaker.sol analyzed (9 contracts with 75 detectors), 52 result(s) found
INFO:Slither:Use https://crytic.io/ to get access to additional detectors and Github integration
```

Solidity Static Analysis

TronStaker.sol

Security

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: 402:4:

Check-effects-interaction:

Potential violation of Checks-Effects-Interaction pattern in TRONStaker.withdraw(contract IERC20): Could potentially lead to re-entrancy vulnerability. Note: Modifiers are currently not considered by this static analysis.
more
Pos: 814:4:

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: 1041:8:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
more
Pos: 790:30:

Block timestamp:

Use of "block.timestamp": "block.timestamp" can be influenced by miners to a certain degree. That means that a miner can "choose" the block.timestamp, to a certain degree, to change the outcome of a transaction in the mined block.
more
Pos: 799:55:

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: 487:50:

Gas & Economy

Gas costs:

Gas requirement of function TRONStaker.getPlanInfo 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: 847:4:

Gas costs:

Gas requirement of function TRONStaker.getPercent 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: 856:4:

Gas costs:

Gas requirement of function TRONStaker.getUserTotalDeposits 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: 1007:4:

Gas costs:

Gas requirement of function TRONStaker.getUserDepositInfo 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: 1017:4:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
more
Pos: 904:8:

For loop over dynamic array:

Loops that do not have a fixed number of iterations, for example, loops that depend on storage values, have to be used carefully. Due to the block gas limit, transactions can only consume a certain amount of gas. The number of iterations in a loop can grow beyond the block gas limit which can cause the complete contract to be stalled at a certain point. Additionally, using unbounded loops incurs in a lot of avoidable gas costs. Carefully test how many items at maximum you can pass to such functions to make it successful.
more
Pos: 1012:8:

Miscellaneous

Constant/View/Pure functions:

IERC20.transfer(address,uint256) : Potentially should be constant/view/pure but is not. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 208:4:

Constant/View/Pure functions:

TRONStaker.isContract(address) : Is constant but potentially should not be. Note: Modifiers are currently not considered by this static analysis.

more

Pos: 1039:4:

Similar variable names:

TRONStaker.invest(address,uint8,contract IERC20,uint256) : Variables have very similar names "user" and "users". Note: Modifiers are currently not considered by this static analysis.

Pos: 755:8:

Similar variable names:

TRONStaker.invest(address,uint8,contract IERC20,uint256) : Variables have very similar names "user" and "users". Note: Modifiers are currently not considered by this static analysis.

Pos: 755:28:

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: 747:8:

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: 827:8:

Data truncated:

Division of integer values yields an integer value again. That means e.g. 10 / 100 = 0 instead of 0.1 since the result is an integer again. This does not hold for division of (only) literal values since those yield rational constants.

Pos: 657:20:

Solhint Linter

TronStaker.sol

```
TronStaker.sol:586:18: Error: Parse error: missing ';' at '{'
```

Software analysis result:

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