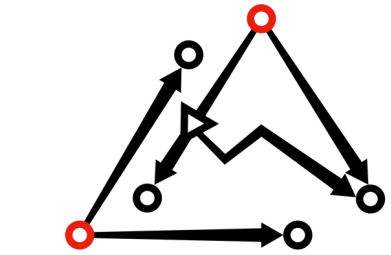

Least Authority

PRIVACY MATTERS

TRON Protocol

Security Audit Report

# TRON

Updated Final Report Version: 6 March 2020

# Table of Contents

# Overview

## Background

TRON has requested that Least Authority perform a security audit of their TRON Protocol, including the following components:

- TRON Protocol
    - Blockchain-based operating systems that aims to offer scalable, high-availability and high-throughput support that underlies all the decentralized applications in the TRON ecosystem.
- TRON Virtual Machine (TVM)
    - Allows users to develop decentralized applications (DAPPs) for themselves or their communities with smart contracts.

## Project Dates

- **January 28 - February 12**: Code review completed *(Completed)*
- **February 14**: Delivery of Initial Audit Report *(Completed)*
- **February 19 - 20:** Verification completed *(Completed)*
- **February 29:** Delivery of Final Audit Report *(Completed)*
- **March 3 - 5**: Second round of verification completed *(Completed)*
- **March 6**: Delivery of Updated Final Audit Report *(Completed)*

## Review Team

- Emery Hall, Security Researcher and Engineer, Least Authority
- Jehad Baeth, Security Researcher and Engineer, Least Authority
- Nathan Fain, Security Researcher and Engineer, Deflect
- Tobais Heldt, Security Researcher and Engineer, Deflect

# Coverage

## Target Code and Revision

For this audit, we performed research, investigation, and review of the TRON Protocol followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:

- Protocol: https://github.com/tronprotocol/protocol
- Java implementation of the TRON Protocol (java-tron): https://github.com/tronprotocol/java-tron

Specifically, we examined the Git revisions for our initial review:

```
c5a3032bb1aa423a6f7434fe50980c83f3a6114a
```

For the verification, we examined the Git revision:

```
507f63d62c7d4fde9c13c44a71f2e40538f43b66
```

For the second round of verification, we examined Git revision:

```
12bf8563dda7bc9c882f315dea75c3b2c489ecf2
```

All file references in this document use Unix-style paths relative to the project's root directory.

## Supporting Documentation

The following documentation was available to the review team:

- Whitepaper Version: 2.0 (TRON Protocol Version: 3.2): https://tron.network/static/doc/white_paper_v_2_0.pdf
- TRON Developer Hub: https://developers.tron.network/
- TRON Virtual Machine: https://developers.tron.network/docs/virtual-machine-introduction

## Areas of Concern

Our investigation focused on the following areas:

- Correctness of the protocol implementation;
- Attack vectors related to building, running & maintaining TRON nodes (eg version upgrades, downloading new clients, fork notifications);
- User funds are secure on the blockchain and cannot be transferred without user permission;
- Vulnerabilities within each component as well as secure interaction between the network components;
- Correctly passing requests to the network core;
- Data privacy, data leaking, and information integrity;
- Key management implementation: secure private key storage and proper management of encryption and signing keys;
- Handling large volumes of network traffic;
- Resistance to DDoS and similar attacks;
- Aligning incentives with the rest of the network;
- Vulnerabilities, potential misuse, and gaming of smart contracts;
- Any attack that impacts funds, such as draining or manipulating of funds;
- Mismanagement of funds via transactions;
- Inappropriate permissions and excess authority;
- Secure communication between the nodes;
- Special token issuance model; and
- Anything else as identified during the initial analysis phase.

# Findings

## General Comments

During our review, we found the TRON code to be well-structured and clear to follow. TRON demonstrates organized modular architecture, an effort to adhere to standardized development processes, and has adequate unit test coverage along with an open access test network environment - all of which are practices that are helpful for reducing the risk of security issues. TRON also maintains a considerable amount of documentation and appears to proactively engage with the TRON community by maintaining an open source code base, which supports independent security reviews, in addition to public communications channels, including Gitter and Telegram, which encourage community participation. Both of these efforts help to encourage the discoverability of security issues and complement independent reviews like this one.

However, TRON's use of a Proof of Stake (PoS) consensus algorithm exposes it to a host of potential attacks that have not yet been identified, largely due to PoS being a fairly new approach to achieving blockchain consensus. This exposure is inherent given that design and development of resilient PoS

systems are still in the early stages and have yet to be extensively researched. Systems that do run PoS in production face criticism for being too centralized, not addressing the "nothing-at-stake problem", and various complexities with regard to block reorganization. Comparatively, the security issues with Proof of Work (PoW) are better researched due to the number, scale and maturity of production systems available. TRON has taken reasonable steps to minimize the inherent risk with PoS, including soliciting an audit of their codebase and adding mechanisms like stacking and slashing.

Precautions like stacking and slashing are incorporated to incentivize good behavior and discourage selfishness by network participants, however, it is not yet clear if these will be sufficient or stable throughout PoS network changes. The use of PoS by TRON and other comparable PoS algorithms in other blockchains will likely require close observation and potentially require adjustments to the mechanisms in place over time. Further efforts such as routine third-party reviews of the code will help to protect against potential, unknown vulnerabilities.

Finally, simplification of core components of the TRON codebase by removing unused code and reviewing SonarQube code linting results should be a continuous effort. TRON has taken the initiative to remove bug level sonar issues for legacy sonar problems, and the development team has stated their intent to gradually address SonarQube issues. We acknowledge that such efforts are undertaken incrementally and iteratively, in order to increase code readability and allow for better review of code quality.

## Table of Findings

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.



## Issues

### Issue A: Unsafe Random Usage

**Location**

java-tron/framework/src/main/java/org/tron/core/zen/address/SpendingKey.java

#### Synopsis

Random classes are not cryptographically strong and the numbers chosen are not completely random because a definite mathematical algorithm is used to select them.

#### Impact

Compromise of the secure generation of `SpendingKey` .

#### Feasibility

Moderate. An attacker would have to control the system elements (system time) used to seed the Random number generator. This would require a side channel where the attacker has installed malware on the target.

#### Technical Details

It is not safe to use Random class for tasks that require a high level of security. Furthermore, this code creates a Random object and uses it to generate but one random number at a time. Random numbers are guessable, especially since the only usage spotted of this method is using a magic number of (0) as a seed value.

#### Mitigation

Replace the Random class usage with a `java.security.SecureRandom` instead and avoid allocating a new `SecureRandom` for each random number needed.

#### Remediation

Use `SecureRandom` for all tasks that require a high level of security. In case of using `Random`class, initialize the `Random`object once, and then call `nextBytes()/nextInt()...` etc each time a new random number is needed.

#### Status

The unsafe `Random`object has been replaced with a correct `SecureRandom` object usage. As a result, our suggested remediation has been fully implemented.

#### Verification

Resolved.

## Issue B: Eclipse Attacks On TRON Nodes

#### Location

[java-tron/framework/src/main/java/org/tron/common/overlay/discover/node/statistics/NodeStatistics.java]

#### Synopsis

`NodeStatistics.getReputation()` does not verify if connected nodes forward traffic correctly and transactions are received by the network.

#### Impact

The value of `maxActiveNodes` can be filled up with adversarial nodes that can censor incoming and outgoing traffic, leading to a DOS on the node and Super Representative. In particular, this vulnerability can be chained or used as a preparation for other types of attacks for Super Representative or exchanges.

#### Preconditions

Due to the implementation of the maxActiveNodesWithSameIp = 2 cap, an attacker needs 15 different IP addresses to control a node's view of the TRON VM. After a node is restarted, the attacker must fill up the node's connection pool before any non-malicious connections are established. If a node is running, an attacker can use the reputation metrics in getScore() to establish connections to a node with a high score (e.g. high p2pHandShake.count, low discoverMessageLatency ). As a result, CheckConnectNumberTask would remove non-adversarial peers with lower reputation over time.

#### Feasibility

Low to moderate probability that enough of the peer nodes can be controlled by the attacker to manipulate the blockchain state of the target VM. In terms of cost, this attack could be conducted for very little using cloud-based VPS services like AWS, where an attacker can easily spin up many hosts for this purpose.

#### Mitigation

Implement whitelisting for node connections and establish and drop connections randomly to make it harder for an attacker to consistently control a victim nodes view.

#### Remediation

A regression test could consist of a node initialized with a full connection pool, where all connections do not forward any outgoing traffic. Mitigation strategies can be measured by the time the node requires to establish and hold a connection to a non-malicious fraction of the network (e.g. containing a validator majority).

#### Status

java-tron implemented the following three strategies to mitigate the aforementioned issue:

1. Provide a whitelisted list of trusted nodes;
2. Randomly drop and establish connections to inactive nodes with exclusion to trusted nodes; and
3. Restricting the number of incoming connections from the same IP address to one.

#### Verification

Resolved.

## Suggestions

### Suggestion 1: Unused Code Should Be Removed

#### Location

Examples (not exhaustive):

- ECKey.java verify() and decodeFromDER() methods
- SM2.java verify() and decodeFromDER() methods

#### Synopsis

Various portions of unused code are left in the codebase. For example, the deserialization CVE non-finding (Non-Finding A) in this report covers code in java-tron which is no longer in use (several methods in ECKey.java and SM2.java ). Leftover unused code will add overhead to future security reviews and code maintenance and, where possible, these portions of code should be removed.

Additionally, because the EthereumJ project is no longer maintained, there is little reason for TRON to maintain code compatibility with EthereumJ and can assume ownership of all left over code in java-tron. Typically, keeping unused code from the original fork maintains compatibility between the forks which makes importing updates, patches and bug-fixes easier.

#### Mitigation

Remove unused/dead code paths.

#### Status

The TRON team has removed unused (vulnerable) code segments from java-tron code base, thus remediating the issue as suggested.

#### Verification

Resolved.

## Suggestion 2: Secure Upgrade Instructions

#### Location

Odyssey 3.6.6 Upgrade Instructions

#### Synopsis

Upgrade instructions for a TRON node are distributed via Odyssey 3.6.6 Upgrade Instructions, which could be used as a phishing template by a malicious actor. Additionally, the information to verify the upgrade is posted in the same channel as the software itself (GitHub Release Page). If this source is compromised, an attacker could change both the hash and the package, yielding valid verification on the for the user.

#### Mitigation

Consider publishing the information required to verify the upgrades for the TRON nodes in a different and redundant location, especially the hash and signature. Also, consider not using the MD5 hash function to verify the upgrades integrity, as MD5 is not considered secure. Ideally, integrity and authenticity of the upgrade could be verified with an EC signature, where the public key of the TRON team is published via TRON's website, Keybase, a gpg keyserver, or another respected source for keys.

#### Status

The TRON team has acknowledged this suggestion and stated their intention to use EC signature instead of the MD5 hash function for their upcoming version 3.7 release. However, we advise that the TRON team still also consider publishing the information required to verify TRON node upgrades in an alternative location for optimal security.

#### Verification

Unresolved.

## Suggestion 3: Review SonarQube Code Linter Results

#### Location

Automated reports provided by the SonarQube code linter.

#### Synopsis

The java-tron repository is already configured to provide automated code review reports using the SonarQube code linter. The linter provides suggestions of areas of code that should be reviewed for

security and stability. Due to limited time available for review, it was not possible for analysts to fully examine the results but a brief review indicates that various portions of code may warrant review. For example, several uses of regular expressions were flagged for review by SonarQube. Several of these issues relate to 3rd party code included in java-tron while others may be specific to code written by TRON.

As a SonarQube configuration is already provided in the java-tron repository, it is possible that TRON is already aware of the issues raised by the linter.

#### Mitigation

Review issues raised and mark as Fixed or False Positive with a description of reason for status change.

#### Status

TRON acknowledged the suggestion and stated that bug level sonar issues have been addressed for legacy sonar problems and that they intend to gradually fix SonarQube issues.

#### Verification

Partially Resolved.

## Suggestion 4: DataWord Mutability Might Lead to Unexpected Behavior

#### Location

java-tron/common/src/main/java/org/tron/common/runtime/vm/DataWord.java

#### Synopsis

DataWord (a data placeholder used extensively in TRON's code base) mutability might lead to undetectable wrong behaviors and hard to trace bugs. Byte[] data object value should not change after initialization, thus protecting the object from unintended and unintentional state change.

#### Mitigation

Re-implement DataWord class to be immutable:

- Declare the class and all mutable fields as final and all fields as private.
- Field initialization should be done using a deep-copy constructor.
- Getters should clone objects instead of returning an actual reference.
- Provide static methods that return new copies with different parameters as needed.

When possible, classes should be immutable. Providing thread safety, side-effect free access and improve correctness and changeability.

#### Status

Although the mutability of DataWord is not a security concern, we recommend measuring the impact on performance by implementing DataWord to be immutable and then running a performance profiling test to better estimate the impact. TRON states that, based on previous test results, implementing DataWord as immutable can cause performance degradation impact due to its extensive usage in TVM. As a result, they have decided against implementing DataWord class to be immutable.

#### Verification

Unresolved.

# Non-Findings

## Non-Finding A: Review of Known EthereumJ Deserialization CVE

**Location**

java-tron/crypto/src/main/java/org/tron/common/crypto/ECKey.java

java-tron/crypto/src/main/java/org/tron/common/crypto/sm2/SM2.java

**Synopsis**

A security related issue and CVE were reported in EthereumJ that concerns unsafe deserialization of data. The java-tron code base was forked from EthereumJ so analysts examined the applicability of this issue to java-tron. A review confirms that java-tron is not affected.

Technical details in this document will assist future researchers and analysts re-examining the deserialization issue.

**Technical Details**

The CVE and issue reported for EthereumJ lacks technical details and only provides the following technical information:

> *There is Unsafe Deserialization in ois.readObject in mine/Ethash.java and decoder.readObject in crypto/ECKey.java*

A review of the EthereumJ code confirms that the call to readObject() in Ethash.java uses standard Java deserialization methods. If these functions are passed, unsanitized input could lead to code execution. However, the Ethash.java code was not carried over from EthereumJ into java-tron as TRON is a staking protocol and does not include code for mining.

The second deserialization issue mentioned in the CVE is in ECKey.java. This code can be found in java-tron in SM2.java and ECKey.java. However, analysts believe that this use of readObject() is likely safe. The readObject() method is defined by the Bouncycastle cryptography library as part of the ASN1InputStream class. The ASN1InputStream#readObject() method is more restrictive than the standard java readObject() methods.

For further information concerning security concerns with deserialization of data see MITRE and OWASP discussions.

**Status**

Code segments related to the issue have been removed from the java-tron code base.

## Non-Finding B: Send Blocks to Corrupt Node's Internal DB

**Location**

java-tron/framework/src/main/java/org/tron/core/db/Manager.java

**Synopsis**

An attacker may send invalid or already published blocks to a node/validator. If the attacker is successful, it could corrupt the node's internal view of the chain or it's database, for example, by reaching a call of

chainBaseManager.getDynamicPropertiesStore().saveBlockEnergyUsage(0);

#### Impact

If an invalid block is stored into the node's DB, it might stop functioning, produce invalid blocks or transactions.

#### Technical Details

There are several potential code paths this or a related attack could be triggered. New blocks arrive to the node for instance in processBlock() :

/java-tron/framework/src/main/java/org/tron/core/net/messagehandler/BlockMsgHandler.java

The block verification mechanisms in place could be circumvented under some conditions:

validBlock() could be bypassed if a validator publishes a new block, just before receiving a block from a fork with a higher block number (but lower timestamp due to latency). The validator updates its internal view to the new fork.

If an attacker manages to set the flag on block.generatedByMyself() = true , validateSignature() and preValidateTransactionSign() is bypassed and the block is being processed by the node.

# Recommendations

We recommend that the unresolved and partially resolved *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

We commend TRON's effort to further simplify core components of the code base ( Suggestion 1). We recommend that these efforts continue , including fully addressing issues reported from the SonarQube automated code linter (Suggestion 3), as they could further increase the speed at which new contributors are able to review and comprehend the project. This would also allow for more effective and efficient future security audits.

In addition, the use of PoS and the preferred network actions by the various participants should be closely monitored to ensure the incentives for good behavior and discouragement of selfishness are sufficient. Furthermore, as more PoS systems go into production, they can be points of reference for additional learning.

Finally, TRON has made significant effort to clearly express the intended functionality of the code via the whitepaper and the developer hub, however, additional documentation and more expressive comments would help new contributors become more familiar with the project.

# About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

And for more information about Deflect GmbH who assisted on this audit, please visit https://deflectsecurity.com/#services

# 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. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

## 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.

## Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.