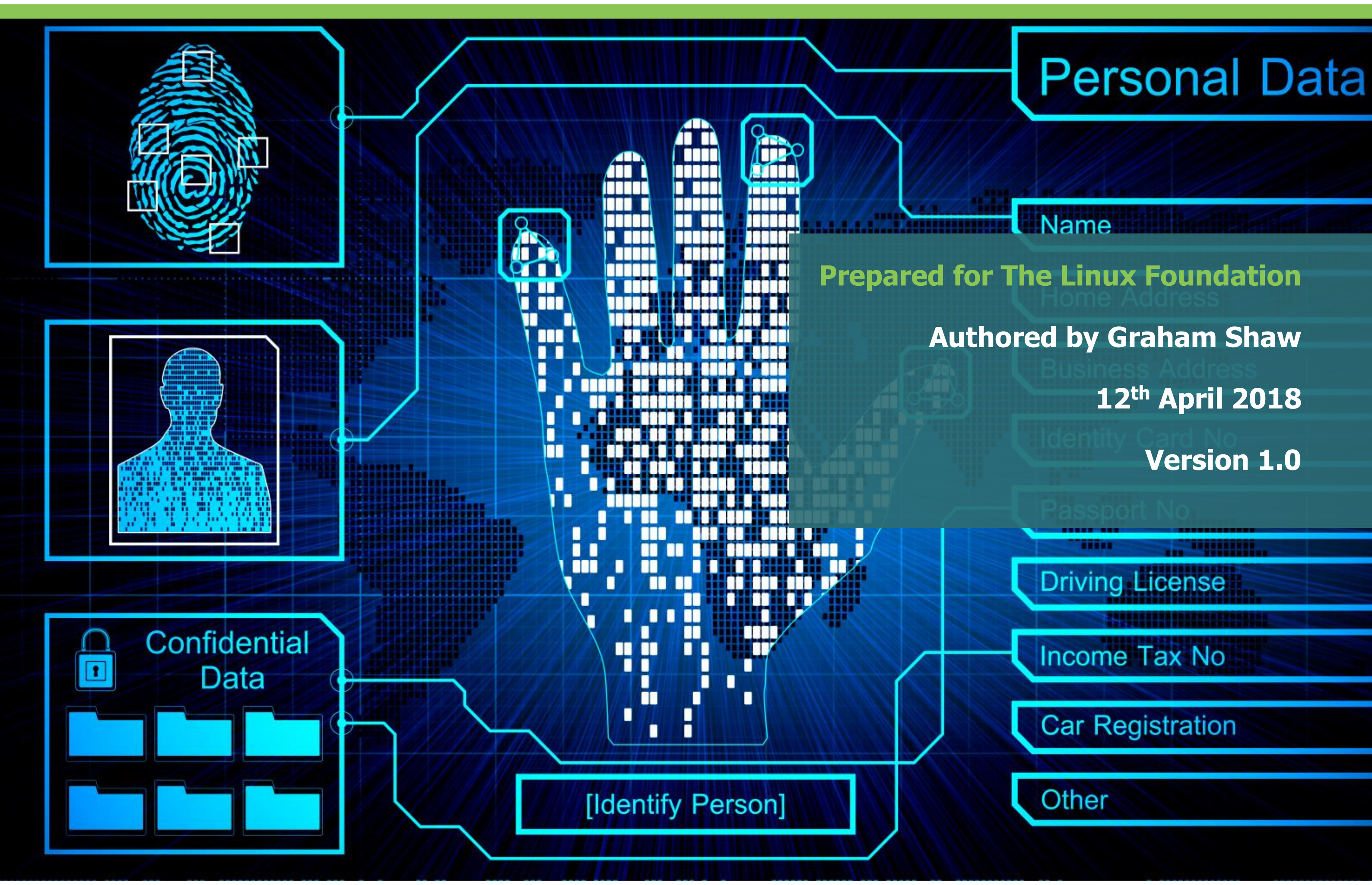
NETTITUDE

INTELLIGENT CYBER SECURITY & RISK MANAGEMENT

Penetration Test Technical Report



Prepared for The Linux Foundation

Authored by Graham Shaw

12th April 2018

Version 1.0

NETTITUDE

INTELLIGENT CYBER SECURITY & RISK MANAGEMENT

Penetration Test Technical Report

Nettitude provides a wealth of knowledge, expertise and experience in regards to Data Security. We provide comprehensive vulnerability assessment, penetration testing and application assessment services. Our team of dedicated security consultants deliver best in class testing capability as well as strong remediation advice and guidance.

NETTITUDE

REPORT CONTENTS

1 DISTRIBUTION LIST

1.1 Nettitude

Name	Title
Graham Shaw	Security Consultant
Kristopher Vasilik	Account Manager

1.2 The Linux Foundation

Name	Title
David Huseby	Security Maven, Hyperledger

1.3 Revision History

Version	Issue Date	Issue By	Comments
0.1	4th April 2018	Graham Shaw	Initial Draft
0.2	10th April 2018	Jose Lopes	Quality Assurance
0.3	10th April 2018	Kristopher Vasilik	Quality Assurance
1.0	12th April 2018	Graham Shaw	Final

The contents of this report belong to The Linux Foundation. They have been provided by Nettitude based on the work detailed within this report and were accurate at the time of testing. Nettitude presents no guarantee that the details in this report are a true reflection of the tested environment at the present time.

2 ENGAGEMENT PARTICULARS

2.1 Background

This report serves as technical documentation for the recent software security assessment performed for The Linux Foundation by Nettitude. For a high-level assessment of the tested software, please refer to the associated management report:

- MANAGEMENT_REPORT_Security_Assessment_The_Linux_Foundation_March_2018_v1.0.pdf

2.2 Rules of Engagement

The penetration test was performed in line with the following rules of engagement:

- Nettitude’s white box testing methodology was used.
- Social engineering was not permitted.
- The software was installed on equipment under the control of Nettitude for testing. Testing of systems belonging to The Linux Foundation was not permitted.
- The testing and reporting was performed during the period 28 February to 29 March 2018 (5 days for penetration testing, 2.5 days for fuzzing, 6 days for code review, 3 days for reporting).
- Any results held in this report relate to the status of the ‘develop’ branch of the repository during the period of the assessment.

2.3 Scope

Nettitude were tasked with performing a penetration test with the following scope:

Repository	Description
https://github.com/hyperledger/iroha	Core repository for Iroha

2.4 Testing Window Observations / Constraints

The client was offered three options for the required level of thoroughness for this assessment. The level chosen was described as "medium assurance" representing a balance between thoroughness and affordability. For code review this entailed:

- Identifying and excluding from consideration ancillary files such as test harnesses and mock implementations.
- Enumerating the endpoints which comprise the outward-facing boundary of the attack surface.
- Mapping these endpoints to the functions which implement them.
- Tracing the flow of untrusted data through these functions, and (to the extent time permitted) through other functions directly or indirectly invoked by them.

(This does not mean that the review was limited to code that was closely associated with the endpoint handlers. The criterion was the extent to which they process untrusted data, regardless of how deep within the system they were located.)

In addition to the above, semi-automatic scanning of the whole codebase (excluding ancillary files) was performed to look for security issues capable of being found in this way. Examples include injection into format strings and random number generation.

Fuzzing was scoped for the REST API which Iroha had at the time. It was discovered early into the assessment that this had been replaced with a gRPC API which is significantly more difficult to test (mainly due to lack of off-the-shelf tooling). For this reason, and due to other delays in the early stages (see below), Nettitude was unable to make more than a token attempt at fuzzing.

Code review was the most productive assessment method applied, resulting in the discovery of what appears to be a subtle but very serious vulnerability which could not plausibly have been found either by fuzzing or by penetration testing. Nevertheless, productivity would have been improved greatly if more detailed and more accurate documentation had been provided, both within the source code and at an architectural level.

A significant amount of time was lost due to unclear or outdated instructions, including in particular obsolete documentation and code published in what ought to be authoritative locations (including the main Iroha repository itself).

3 FINDINGS

3.1 Software Security Assessment

Description	Severity	Ease of Exploitation	Recommendation	Reference
Blocks can be signed more than once by the same peer	**Critical**	**Moderate**	Store signatures in map not set	4.1
Transactions can be signed more than once by the same signatory	**Medium**	**Moderate**	Store signatures in map not set	4.2
IP addresses can be made permanently unusable using the add peer command	**Low**	**Complex**	Provide method for IP addresses to be reused	4.3
Potential denial of service attack due to memory leak	**Low**	**Trivial**	Use a shared_ptr	4.4

4 ANALYSIS: SOFTWARE SECURITY ASSESSMENT

4.1 Critical: Blocks can be signed more than once by the same peer

Description of the Issue

When Iroha needs to check how many peers a block has been signed by, it does so by:

- Collecting the signatures in a container of type SignatureSetType, which (as its name suggests) has set-like semantics.
- Taking the number of elements in this container as the number of peers.

The unstated assumption is that the number of distinct signatures is equal to the number of distinct peers. For an unmodified implementation of ed25519 this assumption would be true, because the signing function is deterministic: if you sign a given message with a given keypair multiple times, the signature is always the same.

In older digital signature algorithms this is typically not the case, due to the signature being dependent on a nonce value which is chosen at random during each signing operation. ed25519 also has a nonce, but it is chosen deterministically by hashing some of the input data. This raises the question: can ed25519 be modified to use a random nonce without breaking compatibility?

Nettitude has demonstrated that this is indeed the case. For example, using a keypair of:

- 9d61b19deffd5a60ba844af492ec2cc44449c5697b326919703bac031cae7f60d75a980182b10
  ab7d54bfed3c964073a0ee172f3daa62325af021a68f707511a

and a message of:

- 48656c6c6f20576f726c64

Nettitude was able to generate three distinct signatures:

- 65fa037aeb893265f849d4a1767ca339c92292d9d7a6df1be7ae91e8cc1dc7651807d1340ec3
  bb99451487c466470a9165a97b07e01f68faad71c8fa6444950e
- e4a09a175338fbc8cf31bee80f9833dfd9884b3c586a8413a57f24dda7db77a795f20e493ad70
  a4ef8e3b555dde3917528d4dddb9032422214c8e7e688d61c07
- 04477edfba1b03aae53384d79de35af3f5081bf334c98b02158384467db9e3e422948738d1ff9
  e616cc53c8a1d1148c0e27de13de9bb7dac471b870ba319c806

all of which appear to be valid when checked using a standard implementation of ed25519.

(To be clear, the signing algorithm used above can no longer be described as ed25519, and nor is it necessarily as secure: there is a good reason why the designer chose to use a deterministic nonce. However, it is only the verifiability of the signatures that is at issue here.)

Given this capability, a malicious peer would be able to construct a block containing an illicit transaction, and attach multiple signatures to that block created using its own keypair. There is no practical limit to the number of such signatures that can be generated, so one peer could add enough of them to look like a supermajority. Nettitude believes (but has not yet proved) that the peer would then be able to successfully propagate that block to other peers. Evidence for this includes:

- Analysis of the classes ChainValidatorImpl and SupermajorityCheckerImpl, which use SignatureSetType as described above.
- Successful demonstration to the proof of concept stage for a similar attack against multi-signatory accounts (see finding below), affecting code which is either similar or shared.

This is sufficient for Nettitude to be very confident, at a minimum, that there is a serious weakness in the code which ought to be fixed. What has not been ruled out, in the absence of an end-to-end demonstration, is the possibility of there being some other obstacle which would prevent the block from propagating. However, all of the currently available evidence points towards this being an exploitable vulnerability, and an initial discussion with a member of the Iroha development team reached the same conclusion.

The impact of one peer being able to unilaterally add blocks to the blockchain, without having to go through the normal consensus-forming process, is to comprehensively undermine the security guarantees which Iroha seeks to provide, and to turn the distributed nature of the blockchain network from an asset into a liability (since any peer would be able to execute the attack).

Nettitude Recommends

- Instead of storing the signatures in a set-like collection, use a map-like collection indexed by the public key to which the signature corresponds.

Further Reading

- Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang. High-speed high-security signatures. Journal of Cryptographic Engineering 2 (2012), pp77–89 (https://ed25519.cr.yp.to/ed25519-20110926.pdf)

4.2 Medium: Transactions can be signed more than once by the same signatory

Description of the Issue

Iroha allows multi-signatory accounts to be created, with behaviour analogous to a multi-signatory bank account. Each account has an 'account quorum', which is the number of signatories needed to sign-off a transaction.

When Iroha checks how many signatories a given transaction has been signed by, it does so using essentially the same method as for peers and blocks:

1 Collecting the signatures in a container of type SignatureSetType.

2 Taking the number of elements in this container as the number of peers.

A similar method of attack can be used as well. Nettitude was able to achieve a partial demonstration of this using the following method:

1 The iroha-ed25519 library was modified to use a non-deterministic nonce, then built and installed within the development environment.

2 The Iroha server already reports whether transactions pass validation, however it was further instrumented to report the number of signatures. This was done in the function StatefulValidatorImpl::signaturesSubset, near to the point in the execution path where the number of signatures is normally checked.

3 Iroha was then built against the modified ed25519 library, and the Python API built against Iroha.

A transaction was signed three times using the same keypair, then submitted to the Iroha server. The server reported that it passed validation with the three signatures intact.

During investigation of this vulnerability, Nettitude found what appears to be a further bug in the signature checking code which would prevent multi-signatory accounts from working as intended: rather than checking that the signatures are a subset of the signatories, it appears to require that the signatories be a subset of the signatures. If that is a correct interpretation then it would make this vulnerability non-exploitable as the code stands currently, but that would likely be a temporary state of affairs.

Assuming that the vulnerability does become exploitable, its impact would be that a single signatory would be able to authorise transactions on an account configured to require multiple signatories. This is moderately serious:

- It has the potential to facilitate embezzlement.
- However, it is only exploitable by users who are already signatories to the account.
- It is only applicable to multi-signatory accounts configured to require more than one signature to authorise a transaction.

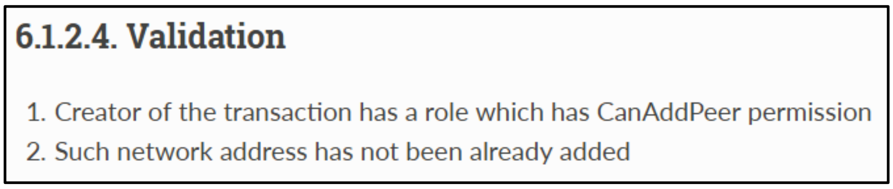
Nettitude Recommends

- Instead of storing the signatures in a set-like collection, use a map-like collection indexed by the public key to which the signature corresponds.

4.3 Low: IP addresses can be made permanently unusable using the add peer command

Description of the Issue

The documentation for the add peer command indicates that a given IP address can only be added once to a given Iroha network (since it is a validation constraint that the address has not already been added).



6.1.2.4. Validation

1. Creator of the transaction has a role which has CanAddPeer permission
2. Such network address has not been already added

Figure 1: Validation constraints for add peer command

A public key is specified at the time when the IP address is added. No evidence is required that the owner of the IP address is in possession of the corresponding private key.

There is not, so far as Nettitude has been able to determine, any mechanism for removing or modifying a peer once it has been added. Therefore, an attacker with the ability to add peers would also have the ability to permanently exclude any number of IP addresses from the network, by associating them with public keys not known to the IP address owners.

Adding peers requires the CanAddPeer permission, which would presumably be issued sparingly. Nevertheless, even though this attack may be difficult to carry out, the irreversibility of the effect makes it a significant cause for concern.

Further issues arising from this limitation are that:

- A peer with reason to fear that its keypair has been compromised is discouraged from switching to a new one, due to the need to switch to a new IP address too.
- A keypair known to have been compromised cannot be locked out.

Nettitude Recommends

Options for resolving this issue include:

- Allowing multiple peers to be registered at the same IP address (although this might increase the risk of a Sybil attack);
- Providing an API call allowing a peer to be removed from the network.
- Providing an API call allowing the public key associated with a peer to be changed.
- Implementing a two-phase registration process, whereby a peer is not fully added until it has begun participating in the network from the specified IP address (although this would not be immune to attack, and does not help with usability concerns such as what happens if a peer loses its private key).

NETTITUDE

4.4 Low: Potential denial of service attack due to memory leak

Description of the Issue

Nettitude has found a memory leak in the Iroha daemon which can be triggered remotely. During a test run during which the supplied example script “tx-example.py” was executed 5000 times, a total of 1920288 bytes in 20003 blocks was leaked. This corresponds to an average of 384 bytes lost per iteration.

Taken to extremes, this behaviour could be used to mount a denial of service attack (although the number of transactions required would be large).

An attempt was made to demonstrate this using the ulimit command to constrain the amount of memory available to the irohad daemon. This was nominally successful, resulting in a segmentation fault, although fragmentation of the heap will also have contributed to that outcome.

The cause in all of the cases observed appears to be failure to delete the object returned by the function shared_model::interface::Transaction::makeOldModel(). Currently this is returned using a built-in pointer. In some instances this is then converted to a shared_ptr by the caller, but not all callers do this (and in any event, this course of action is less safe than using make_shared).

Nettitude Recommends

- Alter makeOldModel to return a shared_ptr, with appropriate changes at points where it is called.

Note

This attack method was originally conceived by Nettitude security researcher Richard Dennis.