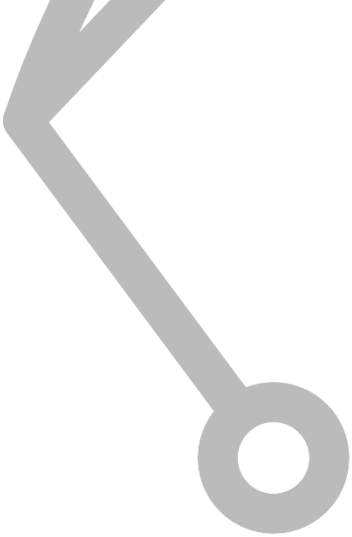
NETITUDE
A member of the Lloyd's Register group



NETITUDE

A member of the Lloyd's Register group

Penetration Testing Technical Report

Prepared for: The Linux Foundation
System: Hyperledger Fabric v1.4 and v2.0
Type: Security Assessment

Author: Graham Shaw
Date: 8 August 2019
Version: 1.0

PCI Security Standards Council ™ CREST CHECK IT Health Check Service

1 Report Contents

2 Distribution List

Nettitude	Name	Title
	Graham Shaw	Security Consultant
	Lilith Toro	Security Consultant
	Jose Lopes	Security Consultant
	Miles Corn	Account Manager

The Linux Foundation	Name	Title
	David Huseby	Security Maven

3 Revision History

Version	Issue Date	Issued by	Comments
0.1	26 July 2019	Graham Shaw	Initial Draft
0.2	28 July 2019	Jose Lopes	Quality Assurance
0.3	29 July 2019	Miles Corn	Quality Assurance
1.0	8 August 2019	Graham Shaw	Final

4 Engagement Particulars

Background

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

```
MGMT_REPORT_Penetration_Test_The_Linux_Foundation_Fabric_2019-07-
26_v1.0.pdf
```

Rules of Engagement

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

- Nettitude's product assurance testing methodology was used.
- Testing of infrastructure owned by The Linux Foundation was not permitted.
- The testing and reporting was permitted and performed during a 40 day period; 3-Jun-19 to 26-Jul-19. Any results held in this report relate to the status of the tested environment on those dates.

Scope

The Linux Foundation tasked Nettitude to perform a security assessment with the following scope:

Component	Description
https://github.com/hyperledger/fabric	Fabric
https://github.com/hyperledger/fabric-ca	Certificate Authority
https://github.com/hyperledger/fabric-chaincode-*	Chaincode support
https://github.com/hyperledger/fabric-sdk-*	Software development kits

The versions tested were v1.4 and v2.0.0-alpha.

Testing Windows Observations and Constraints

The time frame provisioned for the completion of this engagement was adequate.

No constraints were encountered during the engagement.

Findings Summary

During the engagement, a total number of 4 findings were identified. The following table shows the categorization by severity:

0	1	0	3	0
Critical	High	Medium	Low	Info.

In addition to these findings, a number of further observations were recorded which (so far as Nettitude can determine) have no bearing on the security of the product, but are nevertheless issues which The Linux Foundation may wish to address.

5 Findings

1. Security Assessment

Component	Description	Severity	Recommendation	Ref.
Fabric	Content of Private Data Collection potentially guessable by using SHA256 as an oracle	High	Salt private data prior to hashing	6.1
Fabric	PRNG seed length shorter than recommended	Low	Supply recommended seed length of 128 bits	6.2
Fabric	pkcs7UnPadding does not check for an input length of zero	Low	Return gracefully with an error if length is zero	6.3
Fabric	Undefined mode bit allowed in chaincode tarfiles	Low	Disallow the 0100000 bit	6.4

6 Analysis: Security Assessment

6.1 High: Content of Private Data Collections potentially guessable using SHA256 as an oracle

Description of the Issue

Private Data Collections are used to keep transactions private between two or more parties but still take advantage of the unique characteristics of the blockchain. The private data is not itself recorded on the blockchain, however sufficient information is kept to allow interested parties to prove that the transactions in question took place.

In most instances, the data on both the public and private data collections will be highly structured, similar to JSON or YAML, only recording crucial differences over a limited state space. For example, a transaction about a car might be of the form: [car_type: saloon, color: red, price: 10000, brand: BMW].

In addition, a third party might be privy to the underlying structure of this data, for example a third party that uses a different private data channel with the same entity using the same underlying data structure (e.g. a car dealership dealing with 2 different clients).

Ultimately, when Private Data Collections' hashes are written to the public blockchain, a SHA256 hash of the underlying data is created without any additional input or parameters.

A malicious attacker can realistically perform brute force attack against a range of hashes gathered from the blockchain if they know the underlying data structure: they just need to create SHA256 hashes of all possible values.

To use the previous example an attacker can first start enumerating all the possible brands of cars that exist, then colors etc., this should be much quicker than a purely brute force attack and ultimately becomes feasible in a real-world scenario.

In some cases, the underlying structure of the data might be too complex to be realistically brute forced (say it contains some sort of pseudo-random value), however the issue remains that the security of the system is a function of the state space of the structure of the data as well as the data itself.

This presents a problem as end users cannot securely use arbitrarily low complexity data in their usage of Private Data Collections.

In addition, there exists a related issue: Transactions with the same data will have the same hash, so even if the attacker cannot retrieve the plaintext it is revealed that the same data was transmitted.

Affected Components

- Fabric (v1.4 and v2.0.0-alpha)

Nettitude Recommends

1. Introduce some secret value only known to the participating parties of a given Private Data Collection, such as salt added to the input. The secret value might be a nonce generated via a shared deterministic algorithm and secret seed. Ideally it would be 256 bits long to preserve the full strength of SHA256, however it is likely that a much shorter value would be sufficient in practice.
2. It might be worthwhile replacing the hashing algorithm (SHA256) with one that is more resistant to brute force attempts (e.g. scrypt)

A workaround which could be implemented by end users would be to incorporate the salt into the private data passed to Fabric, however it would be less error-prone for Fabric to salt the data itself.

6.2 Low: PRNG seed length shorter than recommended

Description of the Issue

The source code for the class amcl.RAND states that it should be seeded with at least 128 bytes of raw entropy:

```
/* Initialize RNG with some real entropy from some external source */
func (R *RAND) Seed(rawlen int,raw []byte) { /* initialise from at least 128 byte string of raw random entropy */
```

Figure 1: Comments serving as documentation for amcl.Seed

However, the function idemix.GetRand seeds it with only 32 bytes:

```
// GetRand returns a new *amcl.RAND with a fresh seed
func GetRand() (*amcl.RAND, error) {
        seedLength := 32
        b := make([]byte, seedLength)
```

Figure 2: Generation of seed for amcl.Rand by idemix.GetRand

This does not necessarily imply that the random number generator is exploitable, because 128 bytes may be a conservative figure and 32 bytes may be sufficient. The available evidence suggests that this is probably the case. Nevertheless, the current usage is unsatisfactory because:

1. idemix.GetRand is relying on amcl.RAND behaving securely outside the envelope promised by the documentation.
2. The decision to recommend 128 bytes may have been intended to provide a margin of safety, which has now been eroded.

The random number generator provided by amcl.RAND is based on the design of Marsaglia & Zaman, which by itself is not cryptographically secure, however the output is hashed using SHA256 prior to use. Assuming that SHA256 is secure, this should be sufficient to preclude any attack based on working backwards from the content of the generated random number stream. That leaves the possibility of an oracle attack, working forwards from a hypothesis about the state of the generator.

Oracle attacks can be prevented by the simple expedient of ensuring that there are too many possible values to search. In this instance, there appear to be two factors which could limit the size of the search space:

- The entropy content of the seed.
- The initial hashing of the seed using SHA256 (which imposes a ceiling of 256 bits on the amount of entropy that can be introduced by the seed).

The generator is capable of utilizing low-quality random data as its seed, however the seed provided appears to be of high quality, meaning that the entropy content can be expected to be approximately equal to the length. From this, it can be concluded that:

- The seed length needed to make an oracle attack impracticable is likely in the range 8 to 16 bytes (32 to 128 bits).
- A seed length of 32 bytes, from a high-quality random source, provides close to the amount of entropy that the random number generator can make use of. However, it does not reach that limit, because of collisions.

There is therefore some minor erosion of the margin of safety, but much less than the difference between 128 bytes and 32 bytes might suggest.

Some of the uses of idemix.GetRand are in test code, however the use in idemixca.go appears not to be.

Affected Components

- Fabric (v1.4 and v2.0.0-alpha)

Nettitude Recommends

Nettitude recommends one of the following, in order of preference:

1. Supply amcl.RAND with a 128-byte seed, as specified in the relevant header comment.
2. Arrange for amcl.RAND to be updated with an amendment to this requirement.
3. Add commentary to idemix.RAND containing an analysis which explains why 32 bytes is considered safe.

Further Reading

- Eric Bach, "Efficient Prediction of Marsaglia–Zaman Random Number Generators", IEEE Transactions on Information Theory, 44, pp1253-1257, May 1998

6.3 Low: pkcs7UnPadding does not check for an input length of zero

Description of the Issue

The function pkcs7UnPadding in bccsp/sw/aes.go does not check for the case of len(src) == 0, and would panic (as opposed to returning an error) if that was the case. The condition cannot occur in normal use, since padding would always result in there being at least one block. However:

- Due to the location of this function in the call graph, it would be a non-trivial exercise to exclude the possibility that this could be exploited to cause a denial of service using crafted input (and certainly easier to fix the issue than attempt a thorough investigation).
- Irrespective of current usage, future code changes could potentially make this issue exploitable.

Affected Components

- Fabric (v1.4 and v2.0.0-alpha)

Nettitude Recommends

1 Return gracefully with an error if len(src) == 0.

6.4 Low: Undefined mode bit allowed in chaincode tarfiles

Description of the Issue

When tar files containing chaincode are validated, the mode flags for each file are checked against the mask 0100666. The bits 0666 correspond to a mode of -rw-rw-rw and it is fully appropriate to allow them. According to the comments the bit 0100000 is supposedly the ISREG bit, meaning that the file in question is a regular file. If that were the case then it too should also be allowed.

However, the relevant POSIX documentation does not list ISREG as a valid mode bit in a ustar file, nor does it appear to be used by the other common types of tarfile[1].

It is unlikely that setting this bit would facilitate any malicious behavior, and Nettitude has found no reason to believe that it would have any effect at all. However, the fact that this mode bit is currently unassigned means that in principle it could be used for any purpose in the future, with arbitrary effect on the security properties of the files in question.

Affected Components

- Fabric (v1.4 and v2.0.0-alpha)

Nettitude Recommends

1 Disallow the 0100000 bit

Further Reading

- Pax – portable archive interchange, The Open Group Base Specifications Issue 7, 2018 edition

[1] It is valid in cpio files, and this may explain where it came from, however, cpio files are not tarfiles.

7 Further observations

7.1 Introduction

So far as Nettitude has been able to determine the following issues have no bearing on the security of Hyperledger Fabric even if a precautionary approach is taken. However, they represent anomalies which were spotted in the codebase which The Linux Foundation may wish to address.

7.2 bccsp/utils/io.go (DirExists)

Contrary to what the comment (and function name) would lead you to expect, this only checks that there is an object at the specified pathname, not that it is a directory.

7.3 bccsp/sw/aes.go (GetRandomBytes)

The error message states that the length must be larger than zero, but the code allows it to be equal to zero.

7.4 bccsp/sw/aes.go (GetRandomBytes)

The test for n != len is unreachable unless the implementation of rand.Read is non-conformant with the published API, in which case a different (and more alarming) error message would be called for[2]. If you wish to report failure to fill the buffer using this message then the test must precede the one for err != nil.

7.5 bccsp/sw/aeskey.go (SKI)

Using the SHA256 of the secret key as the SKI ought to be safe given that Fabric already relies upon SHA256 for preimage resistance in other parts of the codebase, and the secret key certainly ought not to be guessable. However, it could be argued that having an input that is half the length of the output makes the security case less strong than might otherwise be wished, and that the process can be made more obviously safe by relying on the security guarantees provided by AES itself.

It may be too late to make changes to this without causing backward compatibility problems. However, given free choice, Nettitude's recommendation would be to take

[2] Note that it is only the text of error message which is at issue here, and this observation is not intended to discourage the inclusion of both tests.

a random but fixed block of plaintext, encrypt that using the secret key, then hash the ciphertext to produce the SKI.

7.6 bccsp/sw/conf.go (config)

This class has members named aesBitLength and rsaBitLength, however whereas the latter is measured in bits (as the name suggests), the former is measured in bytes. There does not appear to be anything wrong with the usage given these meanings, however the chosen names introduce significant risk of confusion.

7.7 bccsp/sw/fileks (StoreKey)

The in-memory keystore checks whether a key exists before storing it, whereas the file-based keystore simply overwrites it. Since they are implementations of the same API, it would be reasonable to expect the same semantics.

It should also be noted that the file-based keystore is capable of some very non-intuitive behavior if there were two different key types with the same SKI, or if an attacker were to gain access to the directory where the key files are stored, due mainly to the way in which it handles extensions. However, it appears to behave correctly in all states which should be reachable in normal use.

7.8 bccsp/sw/keyimport.go (KeyImport)

The KeyImport function for aes256ImportKeyOptsKeyImporter checks that the raw data is exactly 32 bytes long, whereas the one for hmacImportKeyOptsKeyImporter only checks that it is non-empty. Even if the latter permits more than one possible key length (not checked this), it could be better validated for consistency.

8 Appendix

A. Severity Rating Matrix

The severity rating is determined by the likelihood and impact of a vulnerability on a system and, where possible, in the context in which that vulnerability is exposed, e.g. remote attack vs. internal attack.

The table below is used to calculate the overall severity rating of an issue based on these criteria.

This is not an assessment of risk as it does not include a valuation of the data or system, but it does provide the ability to prioritize the vulnerabilities identified within the target system or application and to integrate into their own risk management systems.

Likelihood \ Impact	Negligible	Minimal	Moderate	Major	Catastrophic
Rare	LOW	LOW	LOW	MEDIUM	HIGH
Unlikely	LOW	LOW	MEDIUM	HIGH	CRITICAL
Moderate	LOW	MEDIUM	MEDIUM	HIGH	CRITICAL
Likely	MEDIUM	MEDIUM	HIGH	CRITICAL	CRITICAL
Very Likely	MEDIUM	HIGH	HIGH	CRITICAL	CRITICAL

Likelihood

The likelihood rating of a vulnerability encompasses both the likelihood of the vulnerability being identified and attacked as well as the likelihood of that attack being successful. This is evaluated by taking into consideration the following elements:

Exploitability

- Difficulty and technical knowledge or skill required to identify/exploit the issue
- Time or resources required to mount a successful attack
- Availability of exploit code and automated attack tools

Reproducibility

- Ease of reproducing a successful attack

- Additional requirements for the attack to be successful, for example:
  - Victim user must be logged in
  - Some level of interaction by the victim user is required

Discoverability

- Number of instances of the vulnerability identified in the system
- Level of authentication required to access affected components
- Accessibility of the system (internet-facing or internal)
- Degree of specific Insider knowledge required

Frequency

- How often the issue is likely to occur over a period of time
- History of the issue in the industry
- Existence of self-propagating malware targeting the issue

These factors will be employed to formulate a final likelihood rating for a given issue.

Impact

The impact rating assesses the significance of exposure to a particular vulnerability. This is evaluated by considering the impacts to the affected system and the underlying business. The factors under consideration are outlined in the following table.

Impact	Negligible	Minimal	Moderate	Major	Catastrophic
Confidentiality	Disclosure of public information	Minor disclosure of commercial-in-confidence information	Major disclosure of commercial-in-confidence information	Minor disclosure of highly-confidential information	Major disclosure of highly confidential information
Integrity	Unauthorized modification of public data	Small-scale unauthorized modification of private data	Large-scale unauthorized modification of private data	Small-scale unauthorized modification of trusted data	Large-scale unauthorized modification of trusted data
Availability	Minor increase in processing load	Minor outage in a business system	Outage or unavailability of a business system	Extended unavailability or outage of a business system	Unavailability or outage of a business-critical system
Brand or Reputation	Complaints from small number of customers	Complaints from small number of customers across a broader customer base	Complaints from a large number of customers and localized media coverage	Short term adverse large scale media coverage	Extended adverse large scale media coverage
Regulatory and Legal	Warnings for minor breaches	Formal caution for regulatory breaches or threat of legal proceedings	Targeted audit / investigation by regulator or minor legal proceedings brought against the organization	Fines imposed and negative media coverage or major legal proceedings brought against the organization	Service line closed down

B. Penetration Testing Methodology

Nettitude has a series of approaches for conducting Penetration Tests.

Black Box Testing

In a Black Box test, the client does not provide Nettitude with any information about their infrastructure. For internal tests the customer may provide no more than a network point for the tester to connect in to. For external tests, this may simply be a URL or even just the company name that is in scope for assessment.

Nettitude is tasked with testing the environment as if they were an attacker with no information about the infrastructure or application logic that they are testing. Black Box tests tend to take longer to commission than White Box tests and may identify less exposures and vulnerabilities than those of White Box tests.

White Box Testing

In a White Box test, clients provide Nettitude with information about the applications and infrastructure prior to the commencement of the testing engagement. Usernames and Passwords are provided to Nettitude's testing team as part of the engagement, and the client may provide Nettitude's consultants with access to source code. In this type of testing engagement, Nettitude works closely with the client to perform the assessment. These types of tests tend to gain deeper understanding of the application and infrastructure logic, and may generate highly comprehensive test results.

Grey Box Testing

A Grey Box test is a blend of Black Box testing techniques and White Box testing techniques. In Grey Box testing, clients provide Nettitude with snippets of information to help with the testing procedures. This results in a highly focused test.