NETTITUDE
excellence as standard

Security Assessment Technical Report

Prepared for: **The Linux Foundation**

System: **Fabric**

Type: Security Assessment

Author: Graham Shaw

Date: 19th September 2017

Version: 1.1

Report Contents

Distribution List

Nettitude	Name	Title
	Graham Shaw	Senior Research Analyst
	Patrick Matthews	Security Consultant
	Jose Lopes	Security Consultant
	Kristopher Vasilik	Key Account Manager

The Linux Foundation	Name	Title
	David Huseby	Security Maven, Hyperledger

Revision History

Version	Issue Date	Issue By	Comments
0.1	1st September 2017	Graham Shaw	Initial Draft
0.2	4th September 2017	Jose Lopes	Quality Assurance
0.3	6th September 2017	Kristopher Vasilik	Quality Assurance
1.0	8th September 2017	Graham Shaw	Final
1.1	19th September 2017	Graham Shaw	Added pentest details

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.

Engagement Particulars

Background

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

```
MANAGEMENT_REPORT_Linux_Foundation_Fabric_August_2017_v1.1.pdf
```

Rules of Engagement

The assessment 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 permitted and performed during the period 14 August to 31 August 2017 (5 days for pentesting, 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 tested software as of commit a73da04290afe396a8106a2fb9ec26fdf20cca21 (2017-08-13T16:55:47+03:00) for code review, and tag x86_64-1.0.1 for the Docker images used for fuzzing.

Scope

Nettitude were task to perform a security assessment with the following scope:

Component	Description	Source
http://gerrit.hyperledger.org/r/fabric	Main Fabric repository	TLF
http://gerrit.hyperledger.org/r/fabric-sdk-java	Java SDK for Fabric	TLF
http://gerrit.hyperledger.org/r/fabric-sdk-node	Node.js SDK for Fabric	TLF
http://gerrit.hyperledger.org/r/fabric-ca	Fabric certificate authority	TLF
http://gerrit.hyperledger.org/r/fabric-sdk-go	Golang SDK for Fabric	TLF
http://gerrit.hyperledger.org/r/fabric-baseimage	Docker base images for Fabric	TLF
http://gerrit.hyperledger.org/r/fabric-sdk-py	Python SDK for Fabric	TLF
http://gerrit.hyperledger.org/r/fabric-chaintool	Fabric chaincode development	TLF

Testing Window Observations and Constrains

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 RPC endpoints which comprise the outward-facing boundary of the attack surface. A list of these can be found in section 5 of this report.
- 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 RPC handlers. The criterion was the extent to which they process untrusted data, regardless of how deep within the system they were located. For example, this methodology resulted in close inspection of the chaincode handling functions, even though the route by which they are invoked is somewhat indirect.)

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 performed at two levels:

- At the presentation layer, using the gRPC command line tool.
- At the application layer, using the Java version of the Fabric SDK.

Modified protocol definition files were used for some of the tests in order to send data that was valid gRPC, but invalid so far as Fabric was concerned. The possibility of fuzzing at the HTTP2 level was briefly investigated, but abandoned once it became clear that this would exercise little if any code that was actually part of Fabric.

The main constraint encountered was that the Fabric codebase does not include detailed documentation regarding the interfaces presented by functions to other parts of the program. It would have been possible to perform the code review quicker and more efficiently, and thereby achieve greater coverage and depth, if such information had been available.

Findings

Software Security Assessment

Repository	Description	Severity	Ease of Exploitation	Recommendation	Reference
fabric	Chaincode sandboxing insufficient to prevent malicious behaviour	**Medium**	**Complex**	Limit resources and capabilities accessible to chaincode	1
Fabric	Comment headers insufficient for checking implementation and usage	**Medium****	**Code Quality Issue**	Specify function interfaces in comment headers	2
fabric	Log injection	**Low**	**Complex**	Escape untrusted strings before logging	3
fabric	Code injection	**Low**	**Latent Issue***	Either document behaviour or validate arguments	4
fabric	Remote imports allowed/encouraged in chaincode	**Low**	**Latent Issue***	Require whitelisting of remote repositories	5

* A latent security issue is one which has not resulted in an exploitable vulnerability, but which could potentially be the cause of one if circumstances were to change.

** This is a code quality issue which could never itself become an exploitable vulnerability, however it is highly detrimental to the verifiability of the system, and it has therefore been given a severity of medium to reflect the level of concern that is warranted.

Analysis: Software Security Assessment

1 Medium: Chaincode sandboxing insufficient to prevent malicious behaviour

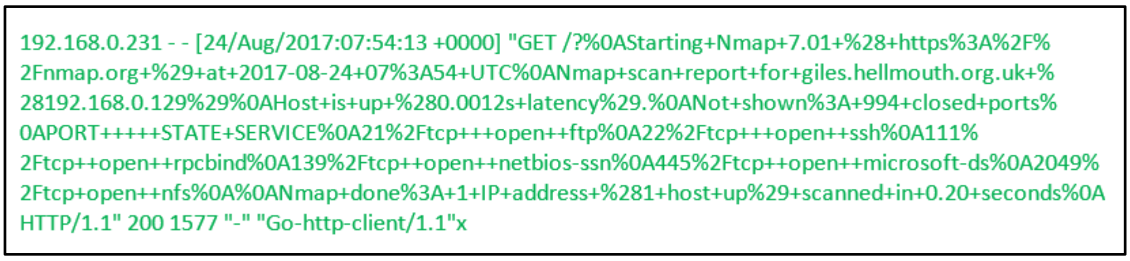
Description of the Issue

According to the documentation, “Chaincode runs in a secured Docker container isolated from the endorsing peer process.” Use of docker does indeed greatly constrain what the chaincode can do, however it was nevertheless found to have sufficient freedom to:

- Install arbitrary software within the container, including security tools such as nmap;
- Perform port scans against public or private networks which are visible to the node;
- Exploit any vulnerable hosts which are discovered;
- Accept commands from, and exfiltrate results to, a remote command and control server;
- Continue executing for a long period of time (perhaps indefinitely).

Taken together, these provide sufficient functionality for a 'Remote Access Trojan' (RAT) to be implemented as chaincode. If installed with an internal company network, such a trojan would provide an excellent foothold which an attacker could used to pivot to other systems.

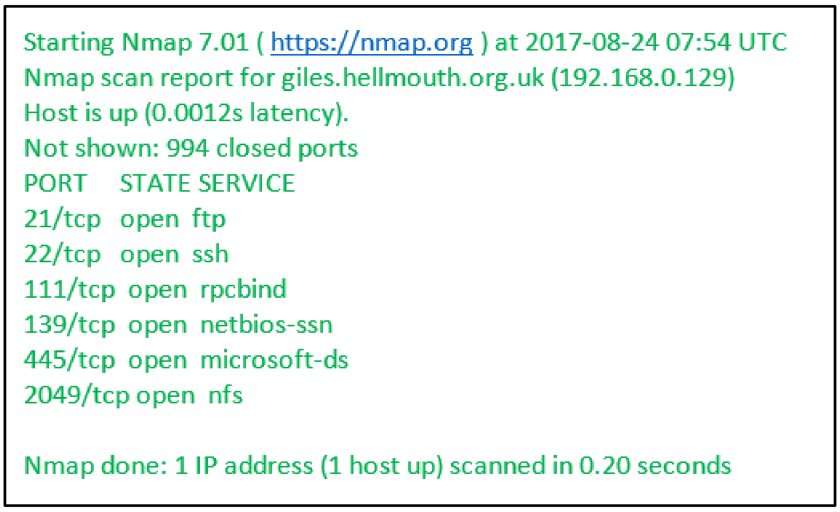
Nettitude was successfully able to demonstrate this concept by writing chaincode to perform an nmap scan of a host attached to a private network, then exfiltrate the result to a remote command and control server. This scan could not have been performed directly from the public Internet, because the host in question did not have a public IP address and was located behind a firewall. The chaincode, however, was in the privileged position of running inside the firewall and connected to the same private network as the host. The report received by the command and control server was as follows:



```
192.168.0.231 - - [24/Aug/2017:07:54:13 +0000] "GET /?%0AStarting+Nmap+7.01+%28+https%3A%2F%
2Fnmap.org+%29+at+2017-08-24+07%3A54+UTC%0ANmap+scan+report+for+giles.hellmouth.org.uk+%
28192.168.0.129%29%0AHost+is+up+%280.0012s+latency%29.%0ANot+shown%3A+994+closed+ports%
0APORT+++++STATE+SERVICE%0A21%2Ftcp+++open++ftp%0A22%2Ftcp+++open++ssh%0A111%
2Ftcp++open++rpcbind%0A139%2Ftcp++open++netbios-ssn%0A445%2Ftcp++open++microsoft-ds%0A2049%
2Ftcp+open++nfs%0A%0ANmap+done%3A+1+IP+address+%281+host+up%29+scanned+in+0.20+seconds%0A
HTTP/1.1" 200 1577 "-" "Go-http-client/1.1"x
```

Figure 1: Raw results of nmap scan as exfiltrated by chaincode

which when decoded yields the following:



```
Starting Nmap 7.01 ( https://nmap.org ) at 2017-08-24 07:54 UTC
Nmap scan report for giles.hellmouth.org.uk (192.168.0.129)
Host is up (0.0012s latency).
Not shown: 994 closed ports
PORT     STATE SERVICE
21/tcp   open  ftp
22/tcp   open  ssh
111/tcp  open  rpcbind
139/tcp  open  netbios-ssn
445/tcp  open  microsoft-ds
2049/tcp open  nfs

Nmap done: 1 IP address (1 host up) scanned in 0.20 seconds
```

Figure 2: Decoded results of nmap scan exfiltrated by chaincode

Nettitude recognises that installation of malicious chaincode would be a non-trivial exercise for most threat actors given the level of access required, however there are some plausible scenarios:

- A threat actor could create a new ledger with associated malicious chaincode, and persuade others to participate.
- A threat actor could infiltrate an organisation responsible for developing and maintaining the chaincode for an existing ledger, then publish an update.

It should be noted that the chaincode need not contain any overtly malicious functionality at the time it is installed on the network: it merely needs to have the capability to download and execute code from a command and control server at some future point in time.

Affected Components

- http://gerrit.hyperledger.org/r/fabric

Nettitude Recommends

Ideally the chaincode would be compiled to some form of custom bytecode with restricted functionality, in accordance with:

- The rule of least power (choosing the least powerful computer language suitable for a given purpose), and
- The principle of least privilege (giving each module of a system access only to the information and resources necessary to carry out its function).

This would, however, require significant development effort, and whilst there is already provision for supporting multiple types of chaincode, it would be necessary to deprecate the existing formats in order to obtain the full security benefit.

Less disruptive mitigations would include:

- Restricting network access provided by the docker container, preferably to just the node which created it;
- Having the chaincode execute as a non-root user;

- Limiting the length of time for which chaincode can run; and
- Ensuring that the chaincode cannot achieve persistence by other means (for example, by spawning a subprocess or running as a cron job).

Whilst it is understood that the system administrator can normally be expected to bear some of the responsibility for ensuring the security of a Docker-based installation, this is a very much less safe assumption for containers created automatically by third-party software without explicit instructions from (and possibly without the knowledge of) the administrator.

Further Reading

- Mitre - http://cwe.mitre.org/data/definitions/265.html

2 Medium: Comment headers insufficient for checking implementation and usage

Description of the Issue

It is desirable that the 'contract' between each function and its callers be documented, as this limits the volume of code which must be read and understood in order to:

- Check that the code which implements the function does what is required of it, and
- Understand the behaviour of code which calls that function.

Although this is clearly not an exploitable vulnerability, it is a security risk if it makes it makes the code review process less effective.

For example:

- Functions such as platforms.generateDockerBuild, golang.flattenEnvPaths, golang.findSource and consumer.processEvents have no interface documentation.
- The comment header for endorser.ProcessProposal does no more than state what is obvious from the function name ('process the proposal') without indicating what this means.
- In pkcs11.importECKey there is a boolean argument keyType, however it is not defined what this means (actually public versus private key), nor specified what values should be passed for each (publicKeyFlag or privateKeyFlag).
- The function endorser.endorseProposal has a byte string arguments named simRes and visibility, however neither the meaning nor encoding of these is specified.
- Much the information provided in the comment headers in fabric-ca/lib/serverinfo.go is wrong. (The handlers are for POST requests, not GET requests, and the path should be /cainfo not /info. The latter should in any event not have been stated as a fact, since there is nothing in the handler function which requires it to be presented via any particular URL path.)
- The function platforms.generateDockerFile would have a potential vulnerability if it were not for the fact that the chaincode name and version strings are constrained in the characters they can contain, and are validated before being passed as arguments. These assumptions are not documented, nor are they obvious from reading the code.

(No particular significance should be attributed to the selection of these particular functions as examples. Depending on the standard aspired to, it would likely be possible to make improvements to most if not all functions in the codebase.)

Some of the burden can be carried by the use of meaningful variable names, but only if those names can be trusted. An example of where this is not the case is validation.ValidateProposalMessage, which in addition to validation is also responsible for demarshalling.

(In this particular instance, rather than change the name it would be preferable to split the function in order to maintain separation of concerns. If it is difficult to name an entity both accurately and concisely then that is often an indication that the required functionality should be decomposed in a different way.)

Affected Components

- All repositories

Nettitude Recommends

Provide comment headers for all functions. Ideally this would provide a complete specification of the interface provided (without constraining the implementation):

- Describe what the function does (and what it does not do, if there is any risk of misunderstanding);
- Provide a specification for each argument and return value, including in particular the meaning of any special values such as true, false and nil; and
- Detail any pre-conditions or assumptions which the function relies upon to operate safely, correctly and securely;
- Detail any post-conditions which the caller is entitled to rely upon.

A good test to apply is whether the function could be:

- Safely re-implemented without reference to the points at which they are called, and
- Safely used without reference to their current implementations.

3 Low: Log Injection

Description of the Issue

Log injection is an attack method which can be used where untrusted data is written verbatim into a system or application log. By doing this an attacker may be able to:

- Fabricate log messages;
- Corrupt the log to prevent it from being automatically processed;
- Exploit terminal emulators or other software used to view the log (uncommon on modern systems).

An example of this can be found in the function lscc.InvalidVersionErr.Error, which applies fmt.Sprintf to the following format string:

```
    "invalid chaincode name '%s'. Names can only consist of alphanumeric, '_' and '-'"
```

The placeholder %s is replaced with the chaincode name which caused the error. Not only could this name contain invalid characters, it is actually known to contain them if this particular function is being executed. This can be demonstrated by executing a command such as:

```
peer chaincode install -n line1$'\n'line2$'\n'line3 -v 1.0 -p mychaincode
```

which will result in a log message being displayed that is split over three lines.

Affected Components

- http://gerrit.hyperledger.org/r/fabric

Nettitude Recommends

- The safest course of action from a security point of view would be to refrain from writing untrusted strings to the logs, however that would be harmful to usability as it would make it more difficult to diagnose errors.
- A more balanced approach would be to escape non-printable characters and limit the length of untrusted strings before they are logged.

Further Reading

- OWASP: Log Injection - https://www.owasp.org/index.php/Log_Injection
- Mitre - http://cwe.mitre.org/data/definitions/116.html

4 Low: Code injection

Description of the Issue

A latent code injection issue was found in the function platforms.generateDockerFile. Strings such as the chaincode name and version are copied verbatim into a docker file, for example in the format string:

```
"LABEL %s.chaincode.id.name = \"%s\" \\"
```

While this remains just a LABEL command there is a limit to how much harm can be done: there may be some adverse impaxct on the software running within the container, but the behaviour of docker itself will not be affected. However, if it were possible to embed newline characters within the string then it would be possible to start a new command on a new line, allowing access to the full range of possible commands.

Fortunately neither chaincode names nor versions strings may contain non-printable characters, therefore this is not currently an exploitable vulnerability. However, this state of affairs is dependent upon those strings being validated elsewhere. This requirement is neither enforced nor documented, either in generateDockerFile itself or in the data structure within which they are passed.

The severity of this issue has been rated as low because the scope for malicious action within a docker file appears to be quite limited. In particular, whilst it is possible to partially specify a bind mount (the purpose of which is to make directories outside the container visible from inside), the external directory is not specified within the docker file, and explicit action is necessary when the container is invoked in order to make that connection. Nettitude cautions, however, that this refers to the version of docker used currently (17.06), and there is a risk that future versions might provide more functionality.

Affected Components

- http://gerrit.hyperledger.org/r/fabric

Nettitude Recommends

The most secure course of action is to adopt a defensive programming style, making no assumptions about the data passed into a function beyond what is enforced by the compiler. In this instance the function would then have a choice between:

- Handling strings containing non-printable characters in a safe manner which does not result in any unexpected behaviour (for example, by escaping any non-printable characters), or
- Validating strings to ensure that they do not contain any non-printable characters, failing with an error if they do.

Disadvantages of this approach are that:

- Repeatedly validating the same data, and checking for conditions that can never occur in the program as written, can adversely affect run-time performance.
- The extra code needed to perform these checks or handle corner-cases can reduce readability by distracting from the main logic.

Alternatively, the constraint can be documented as part of the interface specification for the function. This can be done by making use of the concept of 'undefined behaviour':

- The interface specification would state that the behaviour of the function is undefined if the strings contain any non-printable characters.

- The onus is then firmly on the caller to ensure that this condition never arises.
- If non-printable characters are passed, the writer of the function can say with a clear conscience that the function is behaving fully in accordance with its published specification.

This approach is used extensively within the ISO C, C++ and POSIX specifications. It has the advantage of imposing no run-time burden. The obvious disadvantage is susceptibility to human error.

Further Reading

- Mitre - http://cwe.mitre.org/data/definitions/74.html
- Mitre - http://cwe.mitre.org/data/definitions/94.html
- Mitre - http://cwe.mitre.org/data/definitions/116.html

5 Low: Remote imports allowed/encouraged in chaincode

Description of the Issue

One of the more notable features of the Go programming languages is the ability to import packages from a remote repository identified by a URL. Fabric encourages use of this feature in chaincode, it being the method used in the documentation to load the shim package and other support code.

Attempts to exploit this feature were unsuccessful. In particular:

- It would not normally be feasible to perform a man-in-the-middle attack between the node and the remote repository, because recent versions of Go default to requiring a secure SSL/TLS connection using a certificate with a verifiable chain of trust.
- It does not appear to be possible to modify the chaincode using this method without changing the chaincode ID, because the downloads are performed at an early stage during the chaincode lifecycle prior to packaging.

Nevertheless, Nettitude would caution that use of this facility has the potential to greatly enlarge the attack surface which a threat actor could target, it potentially being possible to modify the behaviour of chaincode after it has been written by:

- Obtaining credentials with commit rights to a repository hosting a remote import.
- Obtaining an SSL/TLS certificate for such a repository, then performing a man-in-the-middle attack. The former could be done by misappropriation of the private certificate, by fraudulent application to a certificate authority, or with the connivance of a dishonest certificate authority.

Auditing and risk management is complicated by the fact that remote imports are not necessarily limited to those listed in the chaincode itself (it being possible for packages to import other packages).

It can be argued that the risk here is no different in principle to (for example) relying on package managers such as apt/dpkg, or indeed, trusting third-party software generally. Certainly if the only repository used is controlled by the publisher of Fabric itself, then a good case can be made that the risk is acceptable. However, chaincode with a long list of dependencies would be of greater concern, particularly if there were any uncertainty regarding quality of governance.

Affected Components

- http://gerrit.hyperledger.org/r/fabric

Nettitude Recommends

- Require whitelisting of remote repositories. This should apply to both direct and transitive imports. The user should be given appropriate guidance regarding the risks of allowing chaincode to depend on third-party code.
- If this is not already enforced by other means, explicitly check for and refuse to use versions of Go prior to 1.5.

Further Reading

- Go GitHub Issues - https://github.com/golang/go/issues/9637

Available RPC Endpoints

Synopsis

The following RPC endpoints were identified. This list has been provided for informational purposes only.

Endpoints

Service	Path	Description
CA	/cainfo	Get information about certificate authority
CA	/register	Register new user
CA	/enroll	Perform new enrollment
CA	/reenroll	Renew enrollment
CA	/revoke	Revoke enrollment
CA	/tcert	Get transaction certificate batch
Admin	GetStatus	Get peer status
Admin	StartServer	Start server
Admin	GetModuleLogLevel	Get log level for given module
Admin	SetModuleLogLevel	Set log level for given module
Admin	RevertLogLevels	Reset log levels
AtomicBroadcast	Broadcast	Send broadcast
AtomicBroadcast	Deliver	Send broadcast, receive responses
ChaincodeSupport	Register	Communicate with chaincode in container
Endorser	ProcessProposal	Submit proposal to endorser
Events	Chat	Send/receive events
Gossip	GossipStream	Send/receive messages
Gossip	Ping	Probe remote peer’s aliveness

(There is also the pprof service on port 6060, however this is not specific to Fabric or Hyperledger.)