

NETTITUDE

excellence as standard

Security Assessment Technical Report

Prepared for: **The Linux Foundation**

System: **Hyperledger Sawtooth**

Type: Security Assessment

Author: Graham Shaw

Date: 13th December 2017

Version: 1.0



Report Contents

Report Contents	2
Distribution List	3
Revision History	3
Engagement Particulars	4
Background	4
Rules of Engagement	4
Scope	4
Testing Windows Observations and Constrains	4
Findings	6
Software Security Assessment	6
Analysis: Software Security Assessment	7
1 High: Private key stored in world-readable file	7
2 Medium: Challenge-Response authentication vulnerable to replay attack	8
3 Medium: Connections to REST API can be made to hang	9
4 Medium: HTML5 Cross-Origin Resource Sharing (CORS)	10
5 Medium: Potential for denial of service attack	11
6 Medium: ZMQ defaults to unencrypted connections	12
7 Low: Log Injection	13
8 Low: Reuse of 'sawtooth' account between subsystems	14

Distribution List

Nettitude	Name	Title
	Graham Shaw	Security Consultant
	Richard Dennis	Security Consultant
	Richard Hicks	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	30 th November 2017	Graham Shaw	Initial Draft
0.2	6 th December 2017	Richard Hicks	Quality Assurance
0.3	6 th December 2017	Kristopher Vasilik	Quality Assurance
1.0	13 th December 2017	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.

Engagement Particulars

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_Linux_Foundation_Hyperledger_December_2017_v1.0.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 performed during the period 30 October to 30 November 2017 (4 days for pentesting, 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 tested software as of commit d1eb66089b2b0b955f0668ea3258f5176c08aa21 (2017-11-01T11:39:54-05:00) for code review, and docker images dates 2017-10-12 for fuzzing.

Scope

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

Component	Description	Source
https://github.com/hyperledger/sawtooth-core	Core repository for Sawtooth	The Linux Foundation

Testing Windows 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 REST and ZMQ 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 performed:

- Against the REST API using AFL. (This required modification of the REST API executable, so that input could be passed to it via the command line.)
- Against the Validator ZMQ API. (This was tested using a custom-written fuzzer, as attempts to integrate it with AFL were unsuccessful in the time allowed.)

Code review was found to be significantly more productive than previously reported in respect of Hyperledger Fabric, due to the quantity and nature of the documentation included within the source code.

Findings

Software Security Assessment

Description	Severity	Ease of Exploitation	Recommendation	Reference
Private key stored in world-readable file	High	Trivial	Restrict access to file containing private key	1
Challenge-Response authentication vulnerable to replay attack	Medium	Complex	Ensure that the nonce used for verification is the one issued by the server	2
Connections to the REST API can be made to hang	Medium	Complex	Improve validation of arguments	3
HTML5 Cross-Origin Resource Sharing (CORS)	Medium	Moderate	Make CORS behaviour configurable, and disabled by default	4
Potential for denial of service attack	Medium	Complex	Allow upper limit on block size to be configured	5
ZMQ defaults to unencrypted connections	Medium	Complex	Allow unencrypted operation only by explicit request, not by default	6
Log Injection	Low	Complex	Escape untrusted strings before logging	7
Reuse of 'sawtooth' account between subsystems	Low	Complex	Create separate account for each subsystem	8

Analysis: Software Security Assessment

1 High: Private key stored in world-readable file

Description of the Issue

A default installation of Sawtooth does not restrict access to the validator private key, located at `"/etc/sawtooth/keys/validator.priv"`.

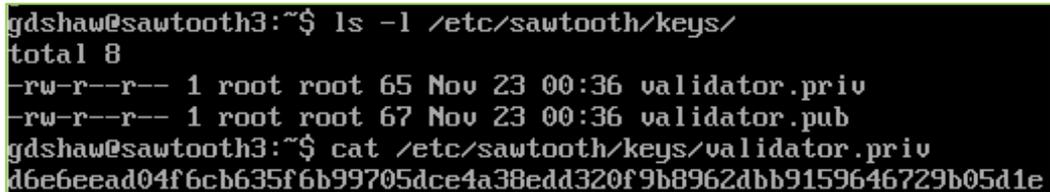
This was found to be an issue with both the current stable Ubuntu packages, and the most recent nightly build of the same, as of 2017-11-23. The installation and configuration procedure was as follows:

```
apt-get install sawtooth
sawadm keygen
```

The following file and directory modes were found:

```
drwxr-xr-x /etc/sawtooth
drwxr-xr-x /etc/sawtooth/keys
-rw-r--r-- /etc/sawtooth/keys/validator.priv
-rw-r--r-- /etc/sawtooth/keys/validator.pub
```

It was then confirmed that the private key could be accessed from an unprivileged account:



```
gdshaw@sawtooth3:~$ ls -l /etc/sawtooth/keys/
total 8
-rw-r--r-- 1 root root 65 Nov 23 00:36 validator.priv
-rw-r--r-- 1 root root 67 Nov 23 00:36 validator.pub
gdshaw@sawtooth3:~$ cat /etc/sawtooth/keys/validator.priv
d6e6eead04f6cb635f6b99705dce4a38edd320f9b8962dbb9159646729b05d1e
```

Figure 1: The validator private key being accessed from an unprivileged account

Running the validator in a separate docker container (as will often be the case) strongly mitigates this issue, however it would be imprudent to assume that configuration.

This issue was found by penetration testing.

Nettitude Recommends

- 1 Change the default file mode for the private key file from 644 to either 600 or 640;
- 2 Consider restricting access to the entire hierarchy rooted at `"/etc/sawtooth"`.

2 Medium: Challenge-Response authentication vulnerable to replay attack

Description of the Issue

There are two methods provided for authenticating to the validator: TRUST and CHALLENGE. The latter provides a higher level of security and is required for some types of operation. It allows the client to prove that it is in possession of the required authentication credentials, without sending them to the server, or anything else which would be of value to an attacker if intercepted.

The reason for doing this is to guard against a 'replay attack', whereby the attacker authenticates himself to the server by replaying the information sent by a previous, legitimate client. To guard against replay attacks, the information required to authenticate must change each time an authentication is performed. The CHALLENGE mechanism tries to achieve this as follows:

- 1 The server chooses a random 64-byte nonce, which it sends to the client;
- 2 The client signs the nonce using its private key, then sends the nonce and signature back to the server;
- 3 The server verifies that the nonce is correctly signed, using the public key for the account in question.

Unfortunately, the server does not keep a copy of the nonce that it sent to the client, and instead trusts the copy that was sent back during step 2. This means that the client is not in fact obliged to use the nonce that was sent to it, and can instead re-use one intercepted during a previous exchange.

Fortunately, this weakness cannot be easily exploited, due to the encryption provided by the ZMQ protocol providing a second layer of defence. However, it would be reasonable to characterise the CHALLENGE mechanism itself as being very seriously broken, since there is no benefit in using a challenge-response protocol if the client can choose what challenge to respond to.

This issue was found by security code review (specifically, understanding the intent of the protocol, identifying requirements which the software would need to satisfy in order to implement the protocol securely, then inspecting the code to determine whether it met those requirements).

Nettitude Recommends

- 1 Record the nonce sent from the server to the client;
- 2 When verifying the signature, use the nonce recorded by the server, not the one sent back by the client;
- 3 If the above recommendations are implemented, there is no particular reason for the client to send a copy of the nonce back to the server, other than backward compatibility perhaps. If it is kept then safe options would be for the server to ignore it, or to compare it with the value sent and terminate the connection if they differ. It should not be used for any other purpose.

(There are alternative options which could be considered if the connections were required to be stateless, however that does not appear to be an issue with the current architecture.)

Further Reading

- Bruce Schneier, Applied Cryptography, 2nd Ed, pp53-54
- RFC 2617, "HTTP Authentication: Basic and Digest Access Authentication", IETF, June 1999

3 Medium: Connections to REST API can be made to hang

Description of the Issue

Two endpoints have been found which can cause connections to the REST API to hang:

- /batch_status
- /receipts

In both cases, the server correctly detects when these are called with no id parameter or with an empty id parameter, and returns immediately with an error message. Similarly, invalid batch IDs are detected and reported immediately provided that they are not empty strings. However, it does not detect empty strings as being invalid when they are part of a list, and doing so causes the connection to hang until it times out after 5 minutes.

One way to reproduce this effect is with a URL of the form:

[http://rest-api:8080/batch_status?id=,](http://rest-api:8080/batch_status?id=)

However, a comma before or after a list of valid IDs will have a similar effect.

This would appear to be caused by the REST API translating the HTTP request into a ZMQ message which the validator considers sufficiently malformed as to not warrant any response at all. The REST API is then left waiting for a response until the default 5-minute timeout is triggered.

It is only the connection which hangs, not the REST API as a whole, therefore the direct impact of this issue on any individual connection is minimal. There are, however, two concerns:

- Any request which can hold onto resources (including, at a minimum, those consumed by an open TCP connection) for an extended period of time has the potential to make a denial of service attack easier to execute. Such attacks can be particularly harmful to blockchain-based systems, due to the potential for influencing the consensus-forming process.
- Although no method for exploiting this has been identified, allowing users of the REST API to send malformed requests into the validator might be considered imprudent.

Additionally, to this finding, some inputs containing null characters were observed to cause hangs. Three of these contained nulls within the id argument, and one within the name of the endpoint. See section 6 of this report for the detailed message content.

This issue was found by fuzzing, then further characterised by code review.

Nettitude Recommends

- 1 Confirm that IDs, including those which are part of a list, are syntactically correct before passing them to the validator. Similarly, for other data types which can be validated in a straightforward manner.

Further Reading

- OWASP: Input Validation Cheat Sheet - https://www.owasp.org/index.php/Input_Validation_Cheat_Sheet

4 Medium: HTML5 Cross-Origin Resource Sharing (CORS)

Description of the Issue

CORS is a mechanism for bypassing the “Single Origin Policy”, which normally prevents JavaScript hosted by one domain from accessing resources located within another domain.

In this instance, whenever a pre-flight request containing an ‘Origin’ field is received by the REST API, the value of that field is reflected back as ‘Access-Control-Allow-Origin’. This has the effect of allowing access from any domain.

```
def add_cors_headers(request, headers):
    if 'Origin' in request.headers:
        headers['Access-Control-Allow-Origin'] = request.headers['Origin']
        headers["Access-Control-Allow-Methods"] = "GET, POST"
        headers["Access-Control-Allow-Headers"] = \
            "Origin, X-Requested-With, Content-Type, Accept"
```

Figure 2: CORS response generation in route_handlers.py

For most web services this would be a clear-cut security weakness, however in this instance there are some mitigating factors:

- It is clearly not an oversight (since the code goes out of its way to allow the requests).
- The Sawtooth REST API does not itself appear to make any use of cookies, HTTP authentication, or any other tokens that could be targeted by an XSRF attack.

However, there are two residual concerns:

- https://sawtooth.hyperledger.org/docs/core/releases/latest/sysadmin_guide/rest_auth_proxy.html provides instructions for putting the REST API behind an authenticating proxy. Proxy servers vary somewhat in terms of which headers they preserve, but there appears to be at least a possibility that XSRF could be an issue.
- It would make it easier to mount a denial of service attack using the web browsers of unwitting third parties.

This issue was found by code review.

Nettitude Recommends

- 1 The CORS behaviour should at least be configurable;
- 2 CORS should be non-permissive by default;
- 3 Use a wildcard ('*') in preference to reflecting the origin field, unless there is a good reason not to, as this is less susceptible to XSRF.

Further Reading

- OWASP: HTML5 Security Cheat Sheet, CORS - https://www.owasp.org/index.php/HTML5_Security_Cheat_Sheet#Cross_Origin_Resource_Sharing

5 Medium: Potential for denial of service attack

Description of the Issue

It is possible to flood the blockchain with large quantities of data, due to:

- There being no upper limit on the block size, and
- The challenge required by the confirmation algorithm not being related to the block size.

This means that the same resources are needed irrespective of whether you are adding a kilobyte, megabyte, or terabyte to the blockchain. A malicious node is thus at liberty to flood the blockchain with an arbitrarily large amount of data, without having to make a large number of transactions.

Whether this can actually be done in practice will depend upon the specific transaction processor in question, as these are by their nature more restrictive in what they will accept. For example, the `intkey_python` processor limits the length of the key, and therefore (implicitly) the size of a transaction, but it does not appear to limit the number of transactions.

In addition to its primary role of ensuring fairness in the consensus-forming process, the requirement for Proof of Elapsed Time does have the secondary effect of limited the rate at which new blocks can be added to the chain. However, without a limit on block size, this does not limit the rate at which data can be added.

Once blocks have been confirmed, they must normally be stored indefinitely throughout the network. This would make recovery from a denial of service attack particularly difficult, requiring either permanently increased resource usage at each of the nodes, or a forced fork of the chain (which might be difficult to implement without new software being written).

There is mention in the documentation of a setting:

```
"sawtooth.validator.max_transactions_per_block"
```

however (a) no trace could be found of this being implemented in the source code and (b) whilst a limit on the number of transactions would go some way towards mitigating this concern, it is really a limit on bytes that is wanted.

There is also the concept of an "NofX" on-chain validation rule, but (a) this too is a transaction limit rather than a byte limit and (b) a separate rule is needed for each transaction type.

This issue was found by high-level design review.

Nettitude Recommends

A hard-coded global limit on block size (as, for example, imposed by BitCoin) may be inappropriate for a general-purpose framework such as Sawtooth due to the potentially large variation in block size needed by different applications. However, Nettitude would recommend:

- 1 That there be the option of setting a global block size limit as part of the chain configuration;
- 2 Additionally, that consideration be given to allowing individual limits to be set for each transaction processor (but preferably with the limits enforced centrally, rather than duplicating code).

Other blockchain implementations solve this issue by ensuring that transaction fees reflect resource usage, for example the concept of Ethereum Gas. However, it is difficult to see how this could be implemented in the context of a multi-purpose framework such as Sawtooth.

6 Medium: ZMQ defaults to unencrypted connections

Description of the Issue

Sawtooth relies upon ZMQ to provide security at the transport layer so that it is not possible to hijack an existing connection after it has been authenticated by means of a man-in-the-middle attack. The ZMQ library will provide this security provided that it is supplied with a public-private key pair via the `network_public_key` and `network_private_key` parameters. As noted in the documentation:

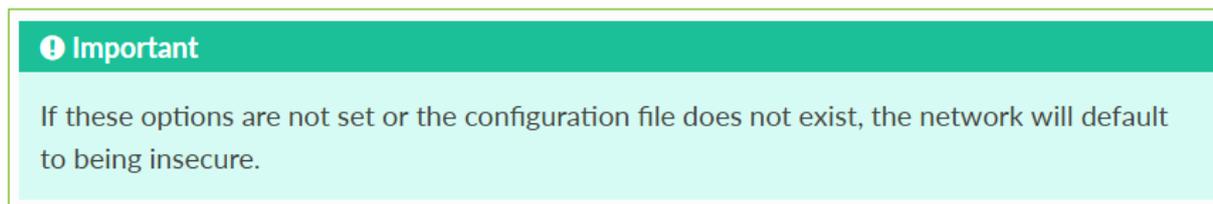


Figure 3: The documented warning regarding behaviour if the ZMQ key pair is missing

A further warning is given in the form of a log message if the keys are not present, so clearly this behaviour is the result of a conscious design decision, and it is possible to conceive of use cases (testing and debugging, for example) where a lack of encryption might be advantageous. Nevertheless, Nettitude would not recommend allowing an insecure configuration like this to arise by default.

This issue was found by code review.

Nettitude Recommends

- 1 Defaulting to a condition where the validator refuses to run if the configuration options, or the configuration file, are missing entirely.
- 2 Requiring the administrator to explicitly request an insecure configuration if that is what is required.

For example, one way to achieve this would be to allow the `'network_public_key'` parameter to take an explicit value of `'none'`. When set that way the validator would allow itself to run with encryption disabled, but if the parameter was absent then it would refuse to run.

Note that if a default/example configuration file is supplied, the `'network_public_key'` parameter should be commented out, absent, or in some other state which prevents the validator from running until the file has been edited by the administrator.

Further Reading

- Sawtooth: Validator Configuration File - https://sawtooth.hyperledger.org/docs/core/nightly/master/sysadmin_guide/configuring_sawtooth_validator_configuration_file.html

7 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:

- Forge 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 `sawtooth_validator.gossip.GossipBroadcastHandler`, which applies `logging.Logger.warning` to the format string:

```
"received %s, not BATCH or BLOCK"
```

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

The placeholder `%s` is replaced with the `gossip_message.content_type`. The value `gossip_message` is the parsed content of a protobuf, and `content_type` is defined in the file `network.proto` as being of type string.

This issue was found by code review.

Nettitude Recommends

- 1 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;
- 2 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>

8 Low: Reuse of 'sawtooth' account between subsystems

Description of the Issue

When the Ubuntu-packaged Sawtooth network services are started by systemd, they all run using the same account named 'sawtooth':

```
sawtooth 1232 1 0 16:11 ? 00:00:00 /usr/bin/python3 /usr/bin/sawtooth-validator
sawtooth 1297 1 0 16:12 ? 00:00:00 /usr/bin/python3 /usr/bin/intkey-tp-python -v -C tcp://localhost:4004
sawtooth 1391 1 8 16:13 ? 00:00:00 /usr/bin/python3 /usr/bin/sawtooth-rest-api --connect tcp://localhost:4004
```

Figure 4: Examples of subsystems running as 'sawtooth'

This is a very considerable improvement upon the traditional practice of running network services as root, however there is an opportunity to go further and use a separate account for each service.

This is warranted because the services have differing security properties in terms of:

- The size of the attack surface that they present;
- The extent to which they are exposed to the public Internet; and
- The action on target that would be possible in the event of a compromise.

On older Debian-based systems this might have been considered profligate due to the limited number of UIDs available for service accounts, however that is not a concern now that UIDs can be dynamically allocated.

This issue was found by penetration testing.

Nettitude Recommends

- 1 Creating a separate user account for each subsystem.
- 2 Where shared access is needed, using the 'sawtooth' group to achieve this.

Further Reading

An early example of a software package which adopted this approach in order to improve its resistance to attack is gmail:

- http://hillside.net/plop/2004/papers/mhafiz1/PLoP2004_mhafiz1_0.pdf