OpenDNSSEC Signer Engine v1.0 Testplan

Matthijs Mekking, Jelte Jansen

April 2009

Chapter 1

Introduction

OpenDNSSEC is an open-source turn-key solution for DNSSEC. It secures zone data just before it is published. One of the modules that is used in OpenDNSSEC is the Signer Engine. This document is the testplan for the Signer Engine. It describes the various tests that need to be done and what their expected in- and outputs are. The tests are listed on two different levels, unit testing and component testing.

The Signer Engine takes one or more zones, together with generated signer configuration and keeps zone data continuously signed. The signer configurations may change over time because of key rollover or policy changes. Zone data is updated by the DNS operator. The Signer Engine handles these changes and is responsible for adding the corresponding DNSSEC records. The actual signing occurs at a HSM.

Chapter 2

Test plan

The Signer Engine needs to be tested on two different levels, unit testing and component testing.

2.1 Unit testing

The Signer Engine is partly written in C. It makes use of the LDNS library. The actual engine is written in python. This part schedules the events and handle incoming signals.

The C code can be tested with CuTest. The python code can be tested with PUnit.

TODO: list required unit tests.

2.2 Component testing

The component must be tested on its functionality. These depend on the inputs of the component. For the Signer Engine, these are the zonefiles, the component configuration file, the zonelist configuration file and the signer configuration files per zone. Also, the Signer Engine must be able to handle a signal that leads to direct re-signing. The component tests are based on a single zone. Stress testing deals with multiple zones and large zones.

2.2.1 Input files

The Signer Engine takes several input files. Most of them require the same action to be taken when such a file is missing or contains errors. These files are the component configuration file, the zonelilst configuration file, the zonefiles and the signer configuration files per zone.

1. No such file
IN: This test depends on the absence of an input file.
OUT: The Signer Engine must quit with signal not 0 and a decent error message.

2. File contains syntax errors
IN: This test takes a syntactically incorrect input file.
OUT: The Signer Engine must quit with signal not 0 and a decent error message.

3. File contains semantic errors
IN: This test takes a semantically incorrect input file.
OUT: The Signer Engine must quit with signal not 0 and a decent error message.
OUT: or: The Signer Engine continues with skipping the faulty semantics.

4. Component configuration file contains non-existing locations
IN: This test takes a configuration file with one or more non-existing locations.
OUT: The Signer Engine must quit with signal not 0 and a decent error message.

5. Correct file
IN: This test takes correct input files.
OUT: The Signer Engine must run normally and a ldns-verified-zone must be produced.

2.2.2 Replacing input files

While the Signer Engine runs as a daemon, several input files may be adapted. For instance, the component configuration file may be edited. Furthermore, zones may be added/deleted in the zonelist, signer configuration for a zone may change as a result of a key rollover, or zonefiles may be updated with new DNS data. We could deal with these sitatuations like in the previous section, as if the system was restarted. However, the purpose of the project is to keep the zones continuously signed. Therefore, we would like to prevent fatal exits as much as possible while running.

6. Input file is replaced with one that contains syntax errors
IN: This test takes a running Signer Engine, a correct input file that is replaced by an incorrect input file.
OUT: The Signer Engine must keep running, using the old values. It must produce a warning.

7. Input file is replaced with correct one
IN: This test takes a running Signer Engine, a correct input file that is replaced by another, correct input file.
OUT: The Signer Engine must keep running, updating the values and put a notice in the logs. The new values must be used.

7. Input file is replaced with one that contains semantic errors
IN: This test takes a running Signer Engine, a correct input file that is replaced by another, correct input file, but which has semantic errors.
OUT: The Signer Engine must keep running, updating the values and put a notice in the logs. The new values must be used. If the semantics leads to faulty behaviour, this must be logged.

The reason for not using the old values in test 7, is that it is hard for the Signer Engine to determine semantic errors. For example, a key locator that doese not point to an actual key in the HSM requires the Signer Engine to check the existence of the key while reading the signer configuration.

Note: Some semantics can be caught. For example, a configured NSEC3 algorithm identifier with value 303 makes no sense.

2.2.3 Parameter verification

The Signer Engine must obey the signer configuration that KASP has provided.

8. Resign interval
IN: This test takes a normal setup, with in the signer configuration a Resign value of x.
OUT: The Signer Engine must update the zone output file every x seconds.

9. Refresh interval
IN: This test takes a normal setup, with in the signer configuration a Refresh value of x.
OUT: The Signer Engine must replace only signatures that have exceeded the refresh interval.

10. Default validity
IN: This test takes a normal setup, with in the signer configuration a Default validity of x.
OUT: The Signer Engine must create signatures with a validity of x.

11. Denial validity
IN: This test takes a normal setup, with in the signer configuration a Denial validity of x.
OUT: The Signer Engine must create NSEC(3) with a validity of x.

12. Jitter
IN: This test takes a normal setup, with in the signer configuration a Jitter of x.
OUT: The Signer Engine must create signatures with expiration date within (e-x) and e, e being the original expiration date.

13. Inception offset
IN: This test takes a normal setup, with in the signer configuration a InceptionOffset of x.
OUT: The Signer Engine must create signatures with inception date within i and (i+x), i being the original inception date.

14. Denial
IN: This test takes a normal setup, with in the signer configuration an empty Denial.
OUT: The Signer Engine must create valid NSECs.

15. NSEC3
IN: This test takes a normal setup, with in the signer configuration a NSEC3.
OUT: The Signer Engine must create valid NSEC3s, following the Algorithm, Hash and Salt in the configuration file.

16. OptOut
IN: This test takes a normal setup, with in the signer configuration the OptOut element.
OUT: The Signer Engine must create valid NSEC3s, following the Algorithm, Hash and Salt in the configuration file. The Signer must skip unsigned delegations.

17. KSK
IN: This test takes a normal setup, with in the signer configuration a bunch of Keys, at least one KSK.
OUT: The Signer Engine must create KSK signatures only for the DNSKEY RRset.

18. ZSK
IN: This test takes a normal setup, with in the signer configuration a bunch of Keys, at least one ZSK.
OUT: The Signer Engine must create ZSK signatures for all RRsets.

19. Publish
IN: This test takes a normal setup, with in the signer configuration a bunch of Keys, at least one Published key.
OUT: The DNSKEY RRset must contain the Published key.

20. No Publish
IN: This test takes a normal setup, with in the signer configuration a bunch of Keys, at least one key that has no Publish element.
OUT: The DNSKEY RRset must not contain the Published key.

20. SOA
IN: This test takes a normal setup, with in the signer configuration a SOA configuration.
OUT: The SOA parameters must match the configuration. Upon zone update, the SOA serial must be incremented in such a way, that it satisfies the policy.

2.3 Fuzz testing

The inputs in the signer configuration files may vary a lot. Such files should be generated by a fuzzer and see if the fuzzed signer configuration files satisfy all parameter verifications. Also, these can be used to test no errors occur when encountering unexpected inputs.

2.4 Memory leaking testing

The component must be tested on memory leakage. The fuzz tests can be run again with Valgrind monitoring them.

Also, the utility tools of the Signer Engine can undergo Valgrind checking. If Valgrind only runs above the engine, it will only check if the python code does not leak.

2.4.1 Input signals

The KASP component or the operator may send a UPDATE signal to the Signer Engine, signalling changes in the input files. Changes should be applied immediately.

TODO: list signal tests

Chapter 3

Testing software

tpkg?