Universiteit van Amsterdam
System & Network Engineering

Resilient OpenDNSSEC

research project 2

Student
Aleksandar Kasabov
aleksandar.kasabov@os3.nl

Supervisor
Yuri Schaeffer (NLnetLabs)
yuri.schaeffer@nlnetlabs.nl

August 20, 2012
Final version

Abstract

The operational burden of maintaining encryption keys and signed zone files is the main hindrance to deploying (Domain Name System Security Extensions) DNSSEC. Companies try to tackle this problem by forcing their administrators to follow operational guide books in every step of the daily DNS activities. However, errors are prone to happen in every process where the human factor is involved.

OpenDNSSEC is a turn-key solution for securing DNS zones with DNSSEC. It offers high performance and automatic key management. This project looks at error situations in securing DNS zones with OpenDNSSEC and how those can be avoided. The paper also makes recommendations for increasing the resilience level which OpenDNSSEC can offer against such situations.

Acknowledgements

I would like to thank my supervisor Yuri Schaeffer for his valuable input during the conduction of this project and the subsequent period of writing this report. His help on plotting the graphs in chapter Optimum TTL settings and especially the mathematical functions behind them has been tremendous and I could not thank him enough.

I would also like to show my gratitude to the NLnetLabs team. They are all bright professionals who create very pleasant working environment and share good ideas which have also found place in this report.

Contents

List of Figures

Chapter 1

Introduction

The need for a secure domain name service has been identified by many enterprises[3]. DNSSEC is one of the solutions for secure DNS. It offers authentication and integrity of DNS data and authenticated denial of existence. Its deployment, however, has been proven far from trivial, mainly due to the operational burden related to the management (and rollover) of encryption keys.

Examples include NASA failing to roll over to a new key signing key (KSK) in the beginning of this year. NASA administrators published a new key in the DNSKEY record set but missed to sign it with both the old and the new keys. Thus, resolvers which had cached the old key could not validate the DNSKEY set which resulted in a bogus zone. As a result, all internet service providers in the world who use DNSSEC to authenticate DNS responses, seized access to the zone for all their users. Comcast was one of these providers. Nevertheless, Comcast published an official report[4] explaining that the problem was entirely NASA's fault.

That is why, enterprises strive to automate their DNSSEC operational activities. Otherwise the latter will have to be manually and periodically performed by their employees which increases the risk for failures. One tool which addresses the issue for automating DNSSEC activities is OpenDNSSEC (ODS). It is an open-source software which strives to fully automate the zone signing, key management and auditing processes related to securing a DNS zone. Once configured and started, ODS performs automatic key rollovers so that no manual interaction by a system administrator is required. It has been developed with efficiency, reliability and automation in mind which makes it attractive to companies, maintaining top-level domain zones, such as SIDN (.nl), Nominet (.uk), .SE - The internet infrastructure foundation - (.se) and DK Hostmaster (.dk).

The following section DNSSEC gives an overview of how DNSSEC works. It explains some basic terms which will be used further in this report. Section Research questions formulates the exact research questions which this project targets to answer.

1.1 DNSSEC

Domain Name System Security Extensions (DNSSEC) is a specification from the Internet Engineering Task Force (IETF) which uses asymmetric cryptography in order to secure the DNS protocol[5][6]. DNSSEC provides authentication and integrity of DNS data, including authenticated denial of existence. It does that by attaching a digital signature (a DNS record of RRSIG type) for each DNS resource record set, whose records have all the same label and type. For instance, several A records for the label www.example.com are signed by one RRSIG record. Note that, signatures increase ten-fold the size of DNS packets. Therefore, Extension mechanisms for DNS (EDNS)[7] have been designed to allow for DNS packets with sizes which go beyond the allowed DNS protocol maximum of 512 bytes.

Signatures in a public-key infrastructure (PKI) are verified using a corresponding public key. That is why, an authoritative DNSSEC-enabled DNS server publishes two more additional records:

DNSKEY Contains a public key which is used by a DNSSEC-enabled resolver to validate received signatures.

DS Stands for Delegation Signer and contains a fingerprint (hashed value) of the active key used for encryption at the child zone. The DS record helps a DNSSEC-enabled resolver to verify that public keys have not been forged during a man-in-the-middle-attack.

An inherent feature of DNS is its hierarchical nature. It means that control of a child zone can be delegated to another administrative domain, different than the one maintaining the parent zone. The DS records helps DNSSEC support this DNS feature - it creates a brink in a so-called ”chain of trust” between a child zone and its apex. This way the child zone is in control of how it stores and handles its private keys. However, additional operational burden is imposed compared to plain DNS. The child zone also needs to update the DS record at its parent whenever the encryption key is changed. Several solutions offer secure automation of this process but manual human interaction with the parent is still common practice.

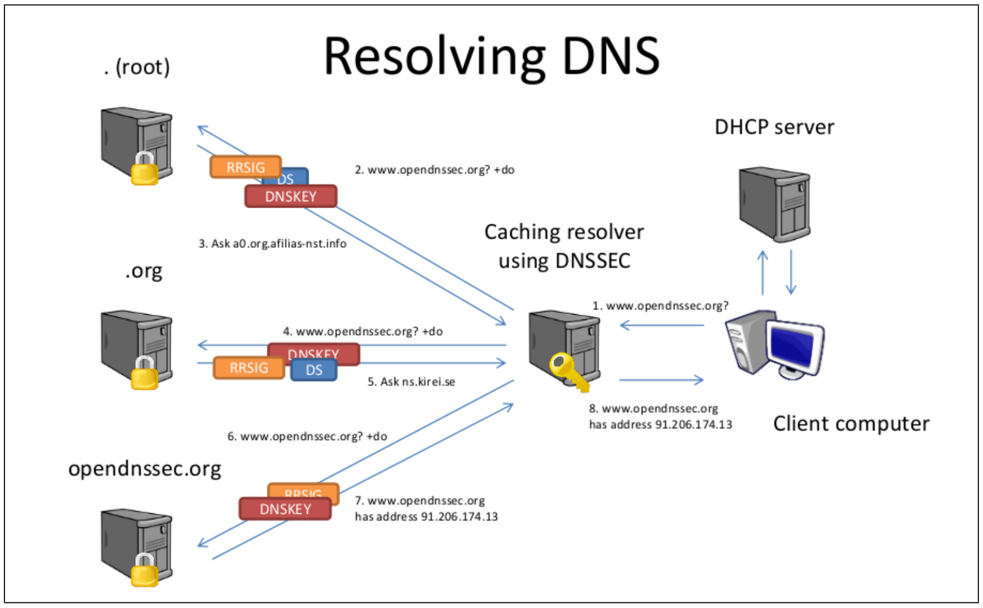

Resolving DNS
. (root)
RRSIG
DS
DNSKEY
2. www.opendnssec.org? +do
3. Ask a0.org.afilias-nst.info
.org
4. www.opendnssec.org? +do
DNSKEY
RRSIG
DS
5. Ask ns.kirei.se
6. www.opendnssec.org? +do
opendnssec.org
RRSIG
DNSKEY
7. www.opendnssec.org has address 91.206.174.13
Caching resolver using DNSSEC
DHCP server
1. www.opendnssec.org?
8. www.opendnssec.org has address 91.206.174.13
Client computer

Figure 1.1: Resolving DNS data with DNSSEC[1]

Figure 1.1 illustrates the ”chain of trust” model. It shows how a caching recursive resolver validates the A record for *www.opendnssec.org* using DNSSEC. The resolver has received a request from a client computer to resolve the address for *www.opendnssec.org* denoted as step 1. It then contacts a root server (step 2) from which it gets DNSKEY, DS and RRSIG records (step 3) and performs the following steps:

1. Validates the DNSKEY against a locally pre-configured *trust anchor*, that is, a DS record which has been manually retrieved and authenticated prior to starting the resolver. Most often this is done by downloading the DS record for the root zone from a verified SSL-enabled web site (e.g. `https://www.kirei.se/en/2010/06/20/root-ksk/` or `https://dnssec.surfnet.nl/?p=371`). Note that a *trust anchor* is used only in step 2 in order to verify the root zone's DNSKEY.

2. Validates the DS record by using the signature in the RRSIG record.

3. Uses the DS record to validate the DNSKEY from the apex zone ".org"

This way the "chain of trust" is traversed until the requested record (www.opendnssec.org/A) is returned from the "opendnssec.org" zone (step 7). The resolver can check each received piece of data during the intermediate steps against its corresponding signature and verify that no forgery has taken place. If verification fails, the zone is called to be 'bogus'.

Apart from providing data authentication, a public-key infrastructure (PKI) can be used for other helpful features, such as DNS-based Authentication of Named Entities (DANE)[8] which, however, is outside this project's scope.

1.2 Research questions

- *What can cause OpenDNSSEC to publish a "bogus" zone?*

This research project has been set up to investigate the resilience level of OpenDNSSEC. It explores cases in which manual user interactions, but also key functions to ODS itself, such as zone signing and key rollovers, can result in publishing bogus zones. During the course of the tests, some bugs have been discovered which will be (or have already been) addressed by ODS developers, but also recommendations have been formed for system administrators who use ODS.

The test cases which were performed are categorised in three groups within chapter Test cases. Section Key rollovers presents how key rollovers were evaluated. Sections Environment changes and Components crash talk about how ODS recovers from changes in its environment or when one of its components crash. Apart form that, various reasons exist for ODS to completely stop working, among which are implementation bugs, human mistakes and natural cataclysms. The aforementioned test cases do not investigate what the risk is for a bogus zone to be published when ODS stops, nor when a zone is expected to go bogus depending on the TTL values of its records.

- *What are the optimum TTL values of keys and signatures which minimise the risk for a bogus zone when OpenDNSSEC has crashed?*

Chapter Optimum TTL settings goes into a more theoretical research. It addresses all those situations during which ODS has completely stopped working for an undefined period of time. This introduces the risk that in the meanwhile encryption keys and/or signatures can expire inside validating resolvers' cache. Even when new keys and/or signatures are published they might not reach validators straight away and validators can try to e.g. validate old signatures with a newly published key. The risk for such cases depends on what 'time-to-live' (TTL) values have been assigned to key and signature records. Hence, the chapter tries to recommend optimum TTL values for key and their corresponding signatures in order to minimise the risk of a zone appearing as 'bogus' for DNSSEC-validating resolvers.

The scope of this project focuses on two particular versions of OpenDNSSEC:

1.4.0a2 This is the latest stable version which has been released prior to the start of this project.

1.5.0a1 This is latest version in development. Hence, it is only available via ODS's svn repository[1] under revision number 6374 and will be released as version 2.0 as soon as it matures to a stable release. This version is also referred as NG (next generation).

The two versions share a lot in their architectural design but differ in their code base. For example, version 2.0 of the Enforcer is rewritten from scratch in order to address better performance benchmarks. The following chapter OpenDNSSEC delves into the details of the two aforementioned versions.

[1] `http://svn.opendnssec.org/branches/OpenDNSSEC-enforcer-ng`

Chapter 2

OpenDNSSEC

Chapter Introduction presented the foundations of DNSSEC and how a zone is validated based on the "chain-of-trust". The chapter also pointed out how DNSSEC relates to this project's research questions. The information in this chapter gives an insight on the inner workings of the OpenDNSSEC suite and the components it's comprised of. This is necessary in order to be able to build an adequate plan for testing its resilience level against unexpected calamities.

The OpenDNSSEC suite has been developed by .SE (The Internet Infrastructure Foundation), Kirei, NLnet Labs, Nominet, SIDN, Sinodun Internet Technologies and SURFnet. It is a turn-key solution for securing DNS zones with DNSSEC requiring minimal-to-no user configuration. OpenDNSSEC takes over the management of encryption keys from system administrators and performs automatic key rollovers. Its other features are high efficiency and resilience against common error cases.

At the time of writing of this report, the latest stable and documented release of ODS is 1.3. However, version 1.4.0a2 is the latest in the ODS svn repositories. Therefore, it was one of the two versions tested during this research project. The main difference between version 1.4 and version 1.3 is the removal of the "auditor" component. Its purpose is to validate produced signed zones and ensure that no bogus zones are published. It has been developed in Ruby and is currently deprecated and removed from version 1.4. The reasons for its deprecation are outside the scope of this project, however, more info can be found on the user mailing list of ODS[1].

[1]`http://comments.gmane.org/gmane.network.dns.opendnssec.user/1086`

Version 1.5.0a1 is the other version used for testing during this research. This version is a total rewrite of any prior versions and will be released as version 2.0 (also known as Next Generation (NG)). It is referenced through out this project as version 2.0. Its performance has considerably been boosted compared to version 1.4 even though the two versions do not have any architectural differences. One notable difference that ODS users should be aware of is that the command line tool *ods-ksmutil* has been removed. Functions which it provides for version 1.4 - management of zones, backups, configuration and key rollovers - have been ported into the *ods-enforcer* tool from version 2.0.

The architectural design of the OpenDNSSEC suite is represented on figure 2.1.

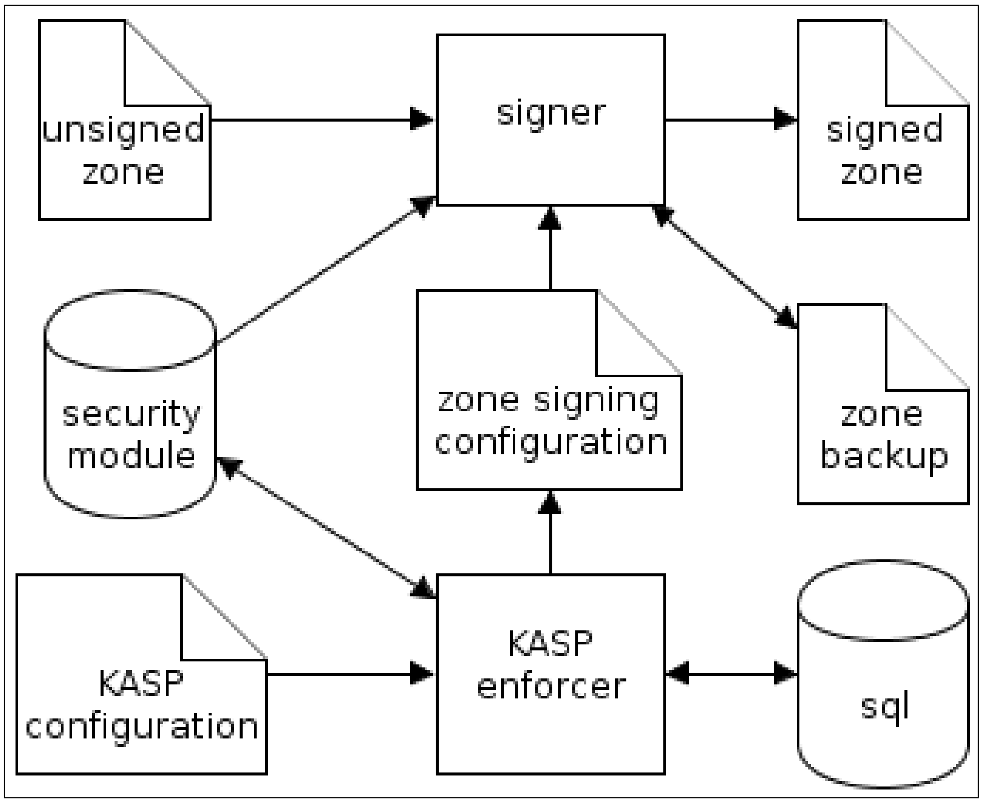


Figure 2.1: OpenDNSSEC architectural design

OpenDNSSEC consists of three components which are developed separately but run in concert with each other:

Enforcer daemon The Enforcer's role is the management of encryption keys. It assigns encryption keys (key-signing and zone-signing keys, or a combined singing key) to unsigned DNS zones in the form of *zone signing configuration files* (residing in directory /var/opendnssec/signconf). Apart from that, it is the Enforcer which takes into account key timing considerations (specified in [9] and extended by [10]) and rolls new encryption keys when old ones are about to expire, based on a predefined key-and-signing policies (KASP's) which are defined in the *KASP configuration* file (located at /etc/opendnssec/kasp.xml). Additionally, the Enforcer makes use of a relational database in order to store configuration from the text files and key state information.

Signer daemon The Signer's responsibility is to consume the *zone signing configuration files* produced by the Enforcer and sign a managed DNS zone. For signing a zone, it uses encryption keys which are referenced from the latter configuration files. Apart from that, the Signer maintains a zone backup file for each of the zones it is responsible for signing.

HSM The Hardware Security Module (HSM) represents a storage for encryption keys. It uses a standard protocol (PKCS#11[11]) to communicate with the Signer which requests particular DNS records to be signed. The HSM is where the actual encryption of DNS records occurs. It never exposes the private part of encryption keys and that makes it highly secure to software exploits.[2]

Key-and-signing policy (KASP) describe how key rollovers and zone signing should be performed. Thus, each policy represents a group of primarily timing configuration parameters, such as:

- the interval at which the Signer runs and checks whether a zone should be resigned
- the period for which DNSSEC signatures are valid for
- the TTL values for encryption keys, how long unused encryption keys are kept before being purged from the system, etc.

[2]A review[12] on several currently available commercial HSM has been provided by Certezza on the commission of .SE (The Internet Infrastructure Foundation).

Be default, two key-and-signing policies are available for use - *default* and *lab*. The default policy targets ordinary users who install ODS and want to have it running without tweaking into configuration settings. It ensures key signing keys are rolled once per year, whereas zone signing keys are rolled every three months. The lab policy is aimed for test users who want to force ODS to roll new keys and resign zones more often (e.g. ZSK are rolled every four hours and zones are resigned every ten minutes). For most of the experiments in this report the lab policy have been used to sign the "example.com" zone.

One interesting feature of the two tested version of ODS is the *timeshift* capability. It is enabled by compiling ODS with the *–enable-timeshift* option and allows users to test ODS' behaviour as if it is running at a different point in time than the current system time. Users can set the desired timestamp to the environment variable ENFORCER_TIMESHIFT and once started ODS will read that one and act based on the time it represents.

Note that for this research a so-called *soft HSM* is used. It is a software version of an HSM which also implements PKCS#11. It is developed by the OpenDNSSEC team and proves very handy for testing or development purposes since it eliminates the need of an actual hardware security module.

Chapter 3

Test cases

This chapter explores potential cases which can cause ODS to publish a bogus zone. Section Key rollovers presents how key rollovers which ODS performs are tested to ensure that signed zones are DNSSEC-valid. The last two sections of this chapter - Environment changes and Components crash - aim to investigate whether ODS properly handles changes in the environment it runs in and when any of its internal components crash. Mitigation advices are given after every described error case. They can be beneficial for both administrators using ODS, but also developers who contribute to its code base. Note that the terms *environment* and *components* have been defined in the context of ODS in this report in chapter OpenDNSSEC.

3.1 Key rollovers

The nature of every cryptographic system requires that keys are replaced after it has reached a certain lifetime. This prevents attackers to deduce an encryption key if they can retain enough encrypted material. In DNSSEC, encryption keys should be replaced (or *rolled*) at regular intervals too. To overcome the operational burden on system administrators managing encryption keys manually, key rollovers should be performed automatically[13] by DNSSEC-enabling software, such as OpenDNSSEC.

Key rollovers in ODS are implemented according to RFC4641[9]. It describes in details the semantics of Key Signing, Zone Singing and Single Signing keys (respectively KSK, ZSK and SSK). Additionally, the RFC presents various key rollover schemes and concerns all timing considerations during key rollovers which should be taken into account by DNSSEC implementors. The aforementioned RFC is in a continuous process of being updated, that is why a *bis* version for it exists which is still a draft. Another internet draft document[10] will also extend the *bis* version with even more key rollover schemes (around 6 more) as soon as it is approved as an RFC.

The RFC4641[9] introduces the notion of key states. This facilitates the modelling of various key rollover schemes because:

- each key is assigned a particular state at any given moment of time
- based on its state, key functions can easily be derived, e.g. whether the key should be published, whether it should be used for signing DNS record sets, whether the latter signatures should be published, etc.

All key states defined in RFC4641 and given succinct names in [10] are illustrated in the figure 3.1.

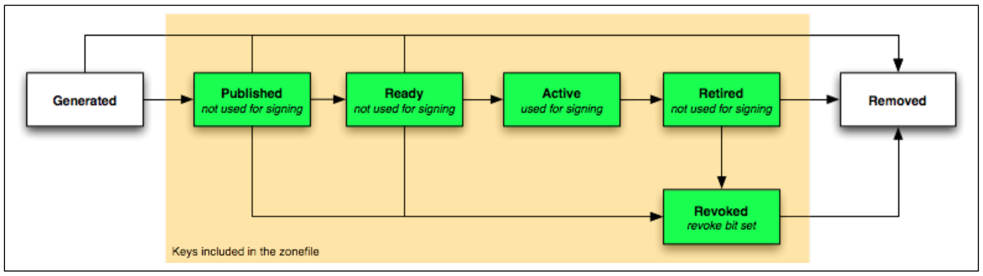


Figure 3.1: OpenDNSSEC key states[2]

According to OpenDNSSEC documentation[2], only keys in the *active* state are used for signing resource records. Thus, if during a key rollover, no active key exists, DNS records cannot be signed. In order to verify the presence of an active key at all times during a key rollover, a test scenario has been created during which key states are monitored as reported by ODS. The actions performed during the test case are as follows:

1. Configure OpenDNSSEC to use the ”lab” policy which rolls a new ZSK each 4 hours
2. Retrieve key state information from OpenDNSSEC each second for more than 4 hours

3. Look at the reported key states on a time line in order to verify an active key exists during each second of the test

In order to use the "lab" policy, one needs to set that in the *zonelist.xml* configuration file for a desired zone (e.g. example.com) as follows:

```
<ZoneList>
        <Zone name="example.com">
                <Policy>lab</Policy>
...
        </Zone>
</ZoneList>
```

Version 1.4 of OpenDNSSEC can report its key states to the user via the *ods-ksmutil key list* command. A bash script writekeystates.sh has been developed in order to automate step 2 of the key rollover test case. Additionally, a python script plotkeystates.py has been programmed to automate the parsing of the collected key state information and plot it on a time line. This way it is easier to visually check whether an active key exists at each point in time during a key rollover.

The generated graph for OpenDNSSEC version 1.4 is depicted on figure 3.1. It clearly shows ODS successfully performs two ZSK key rollovers with accurate timings. It maintains one active (coloured in green) ZSK key at all times, that is each of keys with tag id 60175, 43251 and 63514. The figure also shows a fourth ZSK with tag id 61035 which has just been published. Apart from that, one may notice the key with tag id 22682. It is a KSK which is not rolled over and stays in the ready state at all times since no DS record has been uploaded to the parent. Or more precisely, ODS has not been instructed that a DS record has been uploaded to the parent since the command *ods-ksmutil key ds-seen* has not been invoked. Nevertheless, the latter key is outside the scope of this test scenario.

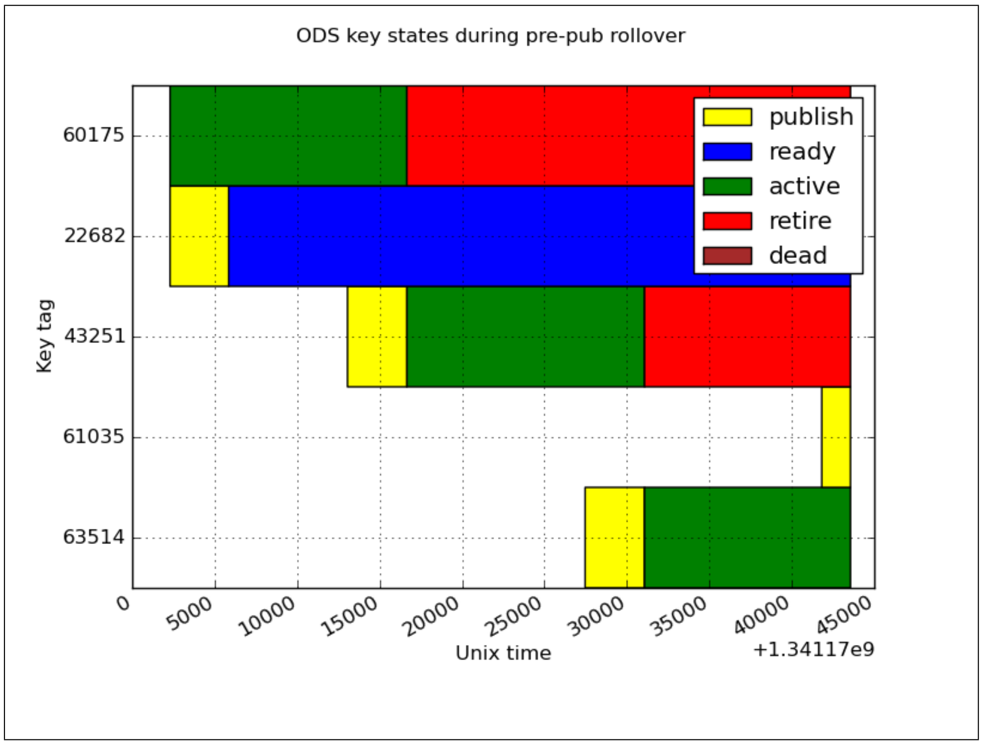


Figure 3.2: Timeline plot of key states during a ZSK rollover with OpenDNSSEC version 1.4

3.2 Environment changes

As described in chapter OpenDNSSEC, the OpenDNSSEC software suite comprises of three processes - Enforcer, Signer and HSM. These processes:

- communicate with each other (through zone configuration files or PKCS#11 interface)
- generate text files (zone configuration files and signed zone files)
- interact with other processes or system environment variables, such as an SQL database and the system date

In the scope of this report, the term ”environment” encompasses all those files or processes which the ODS' processes interact with. Other processes running on the same system where ODS is deployed on, or human mistakes of system administrators, can delete or modify files which ODS uses, i.e. change the environment in which ODS runs. It is essential that ODS handles such changes in its environment and reports to the user when the changes cannot be safely handled. Such behaviour defines the resilience of ODS and prevents error cases which can result in ODS generating bogus zones.

3.2.1 Zone signing configuration file

The Enforcer in OpenDNSSEC creates a zone signing configuration file in compliance with the KASP (Key And Signing Policy) configuration for each zone which ODS is configured to sign. Surprisingly, even though the settings in the KASP configuration are validated when read by the Enforcer, the settings in a zone signing configuration file is not validated by Signer when a zone is about to be signed. Therefore, should the file be edited by hand or by a faulty system process, a bogus zone can be generated.

Note the following zone signing configuration file:

```
<Signatures>
        <Resign>PT60S</Resign>
        <Refresh>PT60S</Refresh>
        <Validity>
                <Default>PT30S</Default>
                <Denial>PT3600S</Denial>
        </Validity>
        <Jitter>PT0S</Jitter>
        <InceptionOffset>PT0S</InceptionOffset>
</Signatures>
```

The file contains a *Resign* period of 60 seconds which is greater than the signature *Validity* period of 30 seconds. This introduces a bogus zone every 30 seconds since the signatures during that time interval have expired. The extract of ODS log file proves the zone is signed once every minute:

Relevant lines from /var/log/ods.log

```
Jul  3 08:50:51 debian ods-signerd: [hsm] sign RRset[1] with key
67ca8c91407a6d7d48aa73c20ca216c7 tag 61035
...
```

```
Jul  3 08:51:51 debian ods-signerd: [hsm] sign RRset[1] with key
67ca8c91407a6d7d48aa73c20ca216c7 tag 61035
```

On the other hand, the generated signed zone file shows that the RRSIG record is only valid for half a minute (from 2012-07-03 08:51:51 until 2012-07-03 08:52:21):

Relevant records from /var/opendnssec/signed/example.com

```
ns1.example.com.        86400   IN      A       192.168.0.1
ns1.example.com.        86400   IN      RRSIG   A 8 3 86400
20120703085221 20120703085151 61035 example.com.
REmIPr35wGCszPQfh/leNAThW6IIRj7495Qt+67V2hQL4G9ClOlT5eZwx6vbh
j0I7edFegLRnRMclowPkisNYBP05nKZLoyMfu/qf5KDcQmQJ58DY2L+hNzCGY
mL7Zq1DR/Nk4M9VLkjn2VKaGn4mrjPZ0FX1+o/juxk/EHerMo= ;{id = 61035}
```

The aforementioned error situation is observed with both investigated versions and can be manually mitigated by regenerating the zone signing configuration file. This can be done by issuing the command *ods-ksmutil update conf* which will also update the zone signing configuration in the database. In general, intermediate files such as the zone signing configuration should be validated and the settings they contain - sanitized or at least reported to the user. This suggests that the Enforcer (which validates such timing settings from the KASP file) and the Signer (which reads such timing settings from the zone singing file) should share a common code base for validating configuration parameters.

A more radical solution to this problem is to combine the two processes in one thus eliminating the need of such intermediate files and, thus, their validation in processes which read them. However, changing ODS' architecture is a bold idea and therefore, it is discussed in chapter Future work.

3.2.2 Database

The database is used by Enforcer to store policy and key timing parameters. The latter are imported to the database by the Enforcer or manually via the *ods-ksmutil update kasp* command. Based on the parameter values in the database the Enforcer generates per zone configuration file - the *zone signing configuration file* - which defines when and using which keys the Signer should further sign the zone.

It is essential that information from the database is validated before used for generating *zone signing configuration files*. In a normal DNSSEC infrastructure the database is a remote entity, residing on a separate machine from the one where ODS is deployed on. Thus, if the database gets compromised, it is possible for an attacker to inject bogus information and thus to cause ODS to produce a bogus zone. Therefore, the database becomes the weakest point of security for OpenDNSSEC. Additionally, such an attack will stay totally unnoticed since no validation, nor reporting of invalid database data, is reported to the user.

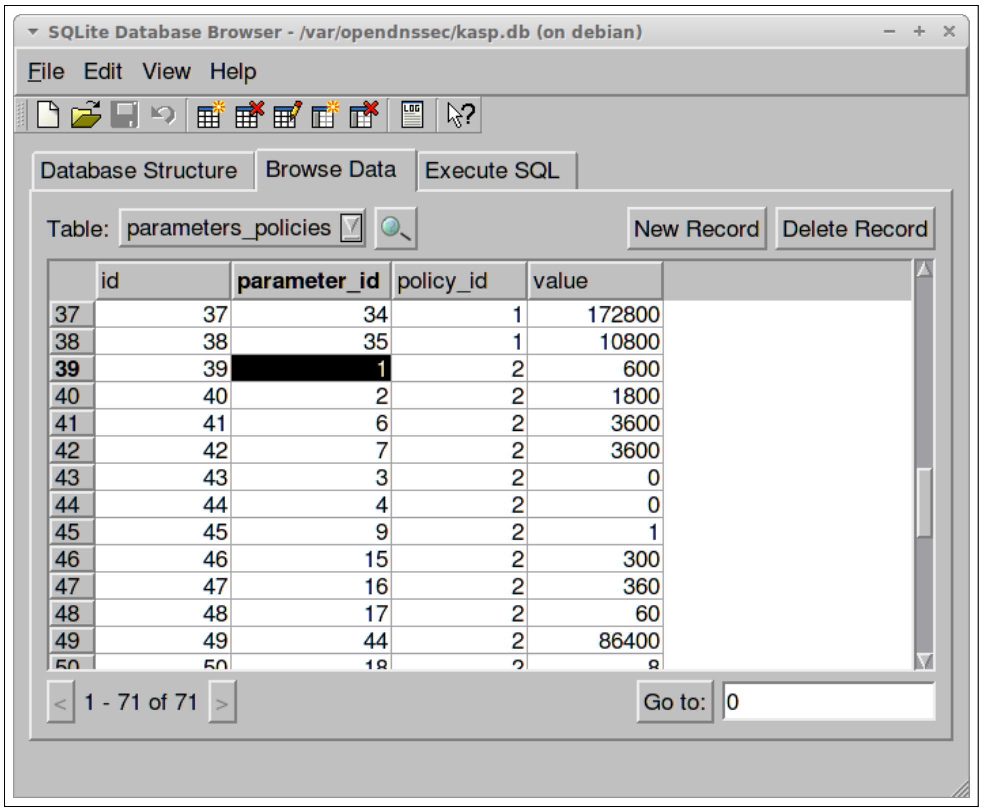
Consider the following case in which a zone is signed according to the "lab" policy. That means that the resign interval is set to 10 minutes and signatures are valid for 1 hour as the following extract from */etc/opendnssec/kasp.xml* demonstrates:

```
<Signatures>
        <Resign>PT10M</Resign>
        ...
        <Validity>
                <Default>PT1H</Default>
        ...
        </Validity>
</Signatures>
```

In the database policy settings are stored in table "parameter_policies". In order to find the value of the "resign" period, one should find the row which contains:

- **parameter_id** equal to 1 which is the id of the resign period parameter (defined in table "parameters")
- **policy_id** equal to 2 which is the policy id for the "lab" policy (defined in table "policies")

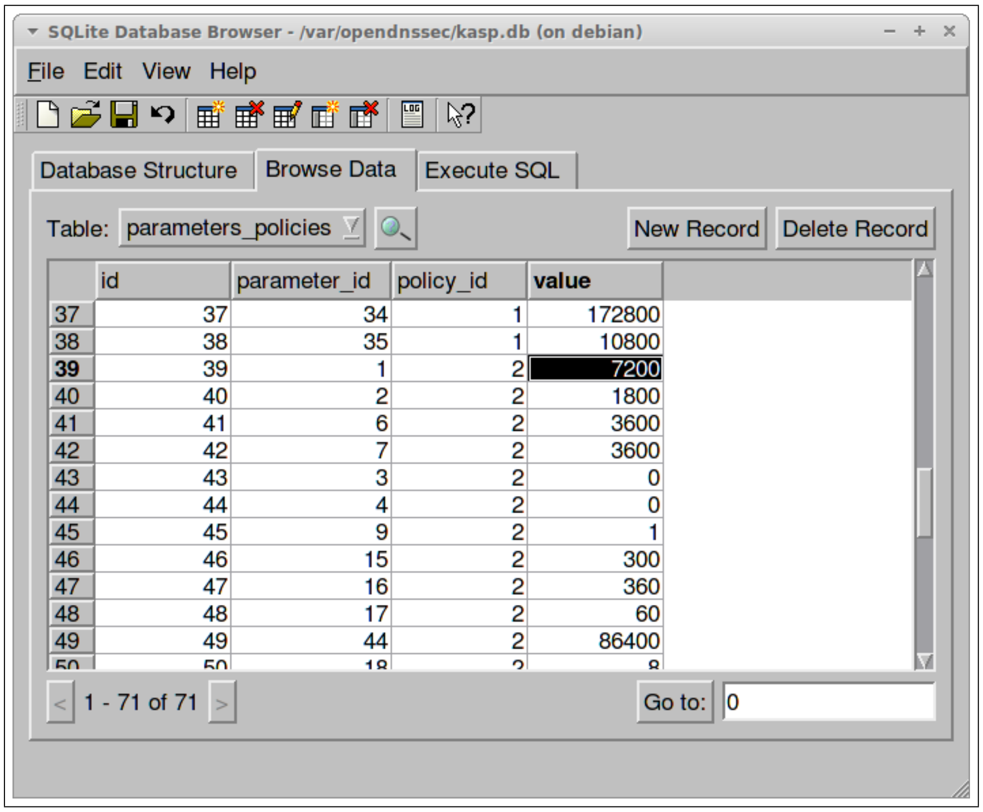
That row should then contain the value of 600 which is the number of seconds in 10 minutes which is the value of the resign interval in the */etc/opendnssec/kasp.xml* file. This is illustrated in figure 3.3 which depicts the use of *sqlitebrowser* - a GUI application for editing SQLite databases.



	id	parameter_id	policy_id	value
37	37	34	1	172800
38	38	35	1	10800
39	39	1	2	600
40	40	2	2	1800
41	41	6	2	3600
42	42	7	2	3600
43	43	3	2	0
44	44	4	2	0
45	45	9	2	1
46	46	15	2	300
47	47	16	2	360
48	48	17	2	60
49	49	44	2	86400
50	50	18	2	8

Figure 3.3: Displaying the resign interval setting for the ”lab” policy in the ODS database using the *sqlitebrowser* GUI application

If an attacker (or a corrupted process or manual user mistake) sets this value to an interval greater than the signatures’ validity period (default for ”lab” policy is 1 hour), he/she will force ODS to sign signatures much slower than the time that they are valid for. Again, the screenshot (figure 3.4) below demonstrates the changed value:



	id	parameter_id	policy_id	value
37	37	34	1	172800
38	38	35	1	10800
39	39	1	2	7200
40	40	2	2	1800
41	41	6	2	3600
42	42	7	2	3600
43	43	3	2	0
44	44	4	2	0
45	45	9	2	1
46	46	15	2	300
47	47	16	2	360
48	48	17	2	60
49	49	44	2	86400
50	50	18	2	8

Figure 3.4: The resign interval has been modified to 7200 seconds for the "lab" policy in the ODS database using the *sqlitebrowser* GUI application

The next time the Enforcer reads the configuration from database it updates the *zone signing configuration file*. Users can manually force this action by calling:

```
root@debian:~/rp2/data$ ods-ksmutil notify
Notifying enforcer of new database...
```

The result is that *zone signing configuration* files are updated with data from the database. A look at the *zone signing configuration* file of "example.com" reveals that the database change has propagated and that the zone resign period has become bigger than the signatures' validity period:

```
<Zone name="example.com">
        <Signatures>
```

```
                <Resign>PT7200S</Resign>
...
                <Validity>
                        <Default>PT3600S</Default>
...
                </Validity>
...
```

As a result, on the next zone sign the Signer uses the new value of the resign interval to adjust its signing schedule, as can be seen in the logs:

```
Jun 18 13:04:00 debian ods-signerd: [signconf] zone example.com
 signconf: RESIGN[PT7200S] REFRESH[PT1800S] VALIDITY[PT3600S]
 DENIAL[PT3600S] JITTER[P] OFFSET[P] NSEC[47] DNSKEYTTL[PT300S]
 SOATTL[PT300S] MINIMUM[PT300S] SERIAL[unixtime]
```

A quick test can prove that after, for example, one and a half hours the Signer will not resign the zone whose signatures are valid for only an hour. In order to test the behaviour of ODS in the future, ODS has been compiled with the *TIMESHIFT* option, as described in chapter OpenDNSSEC. Thus, the environment variable *ENFORCER_TIMESHIFT* is set to "20120618143733" which is exactly one and a half hours after the current system time - "Mon Jun 18 13:07:33 CEST 2012". Once started in the future, the zone does not get resigned because one and a half hours is smaller than the resign interval of two hours (7200 seconds). Further on, the the tool *ldns-verify-zone* is used to prove the validity of the zone during the current time and after one and a half hours by using the *-t* command line argument. The aforementioned actions can be observed in the following terminal snippet:

```
root@debian:/$ date
Mon Jun 18 13:07:33 CEST 2012

root@debian:/$ /root/rp2/ldns-1.6.13/examples/ldns-verify-zone
 example.com
Checking: example.com.
Zone is verified and complete

root@debian:/$ export
 ENFORCER_TIMESHIFT=20120618143733
```

```
root@debian:/$ echo $ENFORCER_TIMESHIFT
20120618143733
root@debian:/$ ods-control start
Starting enforcer...
WARNING: Timeshift mode detected, running once only!
Could not start enforcer
root@debian:/$ /root/rp2/ldns-1.6.13/examples/
 ldns-verify-zone example.com -t 20120618143733
Checking: example.com.
Error: DNSSEC signature has expired for example.com.    NS
Error: DNSSEC signature has expired for example.com.    SOA
Error: DNSSEC signature has expired for example.com.    DNSKEY
Error: DNSSEC signature has expired for example.com.    NSEC
There were errors in the zone
```

Note that, the line saying *Could not start enforcer* is misleading since an instance of the *ods-enforcerd* has actually been started:

```
Jun 18 13:08:44 debian ods-enforcerd: HSM opened successfully.
Jun 18 13:08:44 debian ods-enforcerd: Checking database connection...
Jun 18 13:08:44 debian ods-enforcerd: Database connection ok.
Jun 18 13:08:44 debian ods-enforcerd: Reading config "/etc/opendnssec/con
f.xml"
...
Jun 18 13:08:44 debian ods-enforcerd: Disconnecting from Database...
Jun 18 13:08:44 debian ods-enforcerd: Running once only, exiting...
Jun 18 13:08:44 debian ods-enforcerd: all done! hsm_close result: 0
```

The error message is printed by the *ods-control* application because before completing it verifies that an instance of *ods-enforcerd* is running. However, that is not the case since *ods-enforcerd* is not demonized when *timeshift* mode is engaged. That means that it terminates itself right after performing one full run (just as it reports this to the user: *WARNING: Timeshift mode detected, running once only!*) and right before *ods-control* makes the check whether *ods-enforcerd* is in the process list.

3.2.3 Signed zone files

Signed zone files are generated by the *Signer* process. They are consumed by another process (normally an authoritative slave name server such as

Bind, NSD, etc.) and the zones they describe are "blindly" published. The word *blindly* implies that the name server does not perform any DNSSEC validation on the singed zone files. This introduces the risk that should these files be manually modified by an incautious administrator or a faulty system process, their modification would remain unnoticed.

Here is a sample zone snippet in which a RRSIG record has its inception date after its expiration date:

```
example.com.    300     IN      RRSIG   SOA 8 2 300 201106131
42916 20120613123008 56184 example.com. M7ZWp6BP1nEhBUcNHfcv8
5GQXK64ISBZ9xdf1w2q05WsIH/YwDi1nOSwWVq2ml3bmyQmqq4BNBiLL+ueh5
h18UPYUsX4pDdkjYjvQUNVXyTxfjXEFrhASQ4xAqY/BJZ/+JxQkEiW5aIG4uQ
W+SnVRgJXqxALp8RSLbObYWFDzP0=
```

This record clearly results in a bogus zone which is what *ldns-verify-zone* reports as well:

```
root@debian:~/$ ldns-verify-zone /var/opendnssec/signed/
example.com

Checking: example.com.
Error: Bogus DNSSEC signature for example.com.  SOA
There were errors in the zone
```

The zone can be resigned on demand by issuing the terminal command *ods-signer sign [zone name]*. However, a couple of solutions exist which can remedy such a risk in the long term and eliminate the possibility of published bogus zones:

DNSSEC should monitor signed zone files DNSSEC validates at regular intervals (or at least at start up) whether signed zones are still valid. It is advisable to use software for DNSSEC validation which is different than *ldns* since *ldns* is used by ODS and thus, a bug in ldns results in a bug in ODS as well. One example of other DNSSEC validators is *validns*[1] which is developed by AFNIC with focus on high performance.

[1]http://www.validns.net/about/

Use of CreDNS CreDNS is a DNSSEC proxy. Its purpose is to reside between a hidden master and a public slave dns servers. It does DNSSEC validation in the notify/transfer-chain and makes sure no bogus zones are offered for transfer to the public slave servers.

The first solution concerns solely ODS whereas the second one involves the modification of a whole DNS infrastructure, especially if use of hidden master and public slave DNS server have not been implemented.

3.2.4 System date

Next to public-key cryptography, time is another key element which should be considered when implementing DNSSEC. Each DNSSEC record signature is assigned an inception and expiration date and it is important that DNSSEC signing software and DNSSEC validating resolvers synchronize the time they work with. Thus, the system date is a part of the environment which ODS heavily interacts with.

During the course of this project, it was discovered that ODS 1.4 does not handle changes to the system date correctly. Such changes can be due to:

- incautious system administrators which manually update the system time
- faulty NTP processes which update the system time automatically
- automatic daylight saving adjustments to the system date

Based on the current system's date, it is possible that ODS starts publishing zones with expired signatures. Even though *all* production servers are properly configured right after their first boot against network time service, changes of system date represent a possible risk to the deployment of DNSSEC with ODS version 1.4.

Consider the following case in which ODS have been initialized with one particular system date value, then stopped and started just after the system date has been adjusted to a date in the past (1 year in the case). It turns out that ODS does not notice the date change. Hence, a zone with signatures which are valid for 1 hour is scheduled to be signed by ODS after 1 year from the current date.

```
root@debian:~/$ date -s 20130613
...
root@debian:~/$ ods-control start
...
root@debian:~/$ ods-control stop
...
root@debian:~/$ ntpdate-debian
...
root@debian:~/$ ods-control start
...
root@debian:~/$ ods-signer queue
It is now Wed Jun 13 14:39:32 2012

I have 1 tasks scheduled.
On Thu Jun 13 00:11:04 2013 I will [sign] zone example.com
```

The cause for this problem is the fact that signatures' meta data, such as inception and expiry dates, are stored in the database once the signatures are generated and their meta data is not compared to the current system date when ODS starts up. The same behaviour is observed when the date changes even while ODS is running. The following terminal snippet shows the Signer who remains oblivious to the system date change:

```
root@debian:~/$ ods-control start
...
root@debian:~/$ date
Wed Jun 13 00:00:00 2013
root@debian:~/$ date -s 20120613
Wed Jun 13 00:00:00 UTC 2012
root@debian:~/$ ods-signer queue
It is now Wed Jun 13 00:00:08 2012

I have 1 tasks scheduled.
On Thu Jun 13 00:01:10 2013 I will [sign] zone example.com
```

It is interesting to note that the aforementioned problems with date changes on the system where ODS is deployed are valid only when the date is adjusted in the past. Detecting date change in the future seems to be no problem and ODS resigns a zone instantly:

```
root@debian:~/$ date -s 20120613
Wed Jun 13 00:00:00 UTC 2012
root@debian:~/$ ods-control start
...
root@debian:~/$ date -s 20130613
Thu Jun 13 00:00:00 UTC 2013
root@debian:~/$ head -2 /var/opendnssec/signed/example.com
example.com.    300     IN      SOA     ns1.example.com.
hostmaster.example.com. 2012062437 10800 15 604800 300
example.com.    300     IN      RRSIG   SOA 8 2 300
20130613000100 20130613000000 12849 example.com.
03NeA13hqlG2r8AZeX9MvpIlPp2Q/ZlxHPhbF1uREtIWRyApqfIwTfl7N1nTa
TZrrtq5GaUXjvPwsqtIGisbumjuNB+V1rD2F0YCw2IQNiSmO1LJfAJBV4GwZj
5OiuW4TGmodSI4dJaDkeenwLv9dFnvz+p9l3AUJjKSuP8m0xM= ;{id = 12849}
root@debian:~/$ ods-signer queue
It is now Thu Jun 13 00:00:13 2013

I have 1 tasks scheduled.
On Thu Jun 13 00:00:20 2013 I will [sign] zone example.com
```

Nevertheless, ODS version 2.0 handles properly the date change - both Enforcer and Signer daemons get notified through a linux HUP signal. When a system date change (in the past or the future) occurs, no matter whether it is prior to ODS' start or whilst its running, the next zone resigning date is scheduled properly:

```
root@nsi:~/# ods-signer queue
It is now Thu Jun  13 00:52:42 2012

I have 1 tasks scheduled.
On Thu Jun  13 02:52:32 2012 I will [sign] zone example.com
```

3.3 Components crash

This section looks into the problems of properly signing a DNS zone with OpenDNSSEC when one its components - Enforcer, Signer and HSM as described in chapter OpenDNSSEC - crash and become unavailable. Even though ODS is in a mature state and is already used by several top-level domain authorities, it is occasionally the case that its components might not be operational due to implementation bugs[2][3] in ODS itself but also external events which cannot be avoided (failing hardware, natural disasters, human mistakes, security breaches, etc.).

3.3.1 Enforcer crash

The Enforcer is the component which manages existing encryption keys and rolls new ones. The product of its work is a *zone signing configuration file* which is consumed by the Signer to further sign a DNS zone. The latter file contains a reference to the key (in terms of a keytag and key id) which is used for encryption and, therefore, it needs to be updated when a new key is introduced. Hence, there exist two cases which influence the impact of a crashing Enforcer:

Enforcer crashes prior to generating a *zone configuration file* If such a situation occurs when no *zone configuration file* exists for an unsigned zone, such zone will simply not be signed by the Signer. Thus, such DNS zone will not be published by ODS at all and will remain hidden to public slave DNS server. This has the worst possible negative effect for securing a zone with ODS.

Enforcer crashes after it has generated a *zone configuration file* Such a situation has no impact for introducing bogus zones. The Signer daemon can continue resigning expired signatures with encryption keys, specified in the *zone configuration file*. Nevertheless, encryption keys will not be rolled to new ones until the Enforcer is restarted. This introduces the slight risk of attackers revealing the private part of such key when they collect enough encrypted material. That is why DNSSEC encryption keys' lifetime is researched separately and key lifetime recommendations are given depending on the particular encryption algorithm and key size used[14].

During the conduction of this project, no Enforcer crashes have been observed. The ODS bugs repository[4] does not contain any reported cases for

[2]Signer bug issue in ODS' jira: `https://issues.opendnssec.org/browse/SUPPORT-29`

[3]Signer bug issue in ODS' jira: `https://issues.opendnssec.org/browse/OPENDNSSEC-269`

[4]http://issues.opendnssec.org

this as well. Hence, a crashing Enforcer has been considered of low probability and any further investigation on this topic has been dropped.

3.3.2 Singer crash

The OpenDNSSEC's Signer is the component which monitors when signatures expire and makes sure they are resigned on time. It communicates with the HSM component in order to have those signatures signed. The keys used for that purpose are referenced from within the *zone signing configuration file* which has been generated by the Enforcer for each zone which is configured in ODS.

Nothing can be done to mitigate a crashing Signer component. Monitoring the process is helpful so that once a crash is detected, the process can be restarted as soon as possible. This can be safely done at any moment in time, however it introduces intricacies if a key roll over has begun while the Signer has been unavailable. This has no direct impact for the generation of a bogus zone and lies on the fact all the Signer needs to sign a zone properly is a reference in the *zone signing configuration file* to a key which exists in the HSM. And this requirement is ensured by the Enforcer who makes sure there is always at least one active key at any moment in time[9] during a key rollover.

However, if the Signer has crashed during a key rollover and has been consequently restarted, some of the steps in publishing a new key might have been omitted, e.g. a new DNSKEY record has not been published on time during a pre-publication rollover, or new DNS record signatures during a double-signature rollover schemes. The result is that the propagation time for publishing new DNS keys (or signatures) has not been met and client resolvers suddenly see new signatures (or keys, respectively). Hence, resolvers will fail to validate the DNS zone if a mismatch exists between:

1. old signatures and new keys (which have not been used to sign the old signatures) or...
2. new signatures and old keys

The latter problem has led this project into more theoretical approach in order to find out how the risk of client resolvers, which fail to validate a zone when the Signer crashes during a key rollover, can be minimized. The findings are discussed in chapter Optimum TTL settings.

3.3.3 HSM crash

The HSM component is not a strictly part of the OpenDNSSEC suite even though ODS requires one to work with. A software implementation of an HSM with PKCS#11 support is released by the ODS team. It can be used by companies which do not want to invest in a hardware security module - HSM's can be expensive depending on the level of performance and features which they offer. For more information, [12] presents an in-depth comparison of several HSM.

The role of an HSM is to store encryption keys and sign data with them without exposing their private part. Both the Enforcer and the Signer components communicate with an HSM store and, that is why, it is interesting to discover how they handle the case when an HSM has been destroyed all encryption keys have been lost. If a crashed HSM has been replaced OpenDNSSEC is expected to create new keys. Once signatures which have been made with old key expire, the newly created key should also be introduced and then used to resign all affected zones. Note that, rolling a new key does not need to be sooner before signatures with the old key are about to expire since those are still valid.

ODS provides functions for deleting keys in the HSM, and thus the aforementioned error situation can easily be reproduced. The command which purges all encryption keys from the HSM is *ods-hsmutil purge* and it takes the name of a repository to purge the keys from (called "SoftHSM" throughout the project's test set up):

```
root@debian:~/$ ods-hsmutil purge SoftHSM
Purging all keys from repository: SoftHSM
21 keys found.

Are you sure you want to remove ALL keys from repository
 SoftHSM ? (YES/NO) YES

Starting purge...
Key remove successful: f59d17361cb5f154d46e3dab86fe6925
Key remove successful: 939932f5f9b26467e4a643855dd8d377
Key remove successful: 2daa6d688eca473632a3a8907b761c19
Key remove successful: 547885b6d6089f7a2e532a74d41a2df1
Key remove successful: 1425090aba4baf3b15c4ec2bf0fd3b79
Key remove successful: 41b60afd996235ef4d44b118cd65e8a3
Key remove successful: 63900749773e99f5d46d4eaa17c5eb23
Key remove successful: aa81b4c08d24833a1017518997f7f3de
Key remove successful: c5786de6003694e6909b1c02834b7b2c
Key remove successful: 333ac0c1e4f9deb26de2bb08055cfbad
Key remove successful: e71ffadf1957ad33565c61f9ecba5901
Key remove successful: 31b438f2eec5eaf02de005ae9295463c
Key remove successful: 99e5ede665c3ec3b4a504d91898596bc
Key remove successful: 9698063ebb9de9b6bcd5754bf2d76c90
Key remove successful: a3fc87acad6198f55641653db6635a14
Key remove successful: a8f88f6f65e6092a558147173c930076
Key remove successful: d10aee67d9b75d552740cf874804110d
Key remove successful: a4f20ab242c81914f3790fac10073e5e
Key remove successful: 30fe711e018ccd7729808319b2d45e98
Key remove successful: 08f9415c88f9335c624658f6f6ad6acf
Key remove successful: fcdd99fb271ff85c9719f7cb4d27f753
Purge done.
```

After keys have been purges from the HSM, the ODS logs reveal that the Signer has problems resigning the zone and gives up with an error message.

```
Jun  15 17:58:41 debian ods-signerd: [worker[1]] sign zone
 example.com
Jun  15 17:58:41 debian ods-signerd: [zone] zone example.com soa
 serial already up to date
Jun  15 17:58:41 debian ods-signerd: [hsm] libhsm connection ok
Jun  15 17:58:41 debian ods-signerd: [rrset] -RRSIG: example.com.
 300 IN RRSIG DNSKEY 8 2 300 20130614202125 20130614202025 22682
 example.com . a8+gm9UuRgYfZ1co/omISckFBQ6Awo1nshtiLaeqMULKMJtOHp
 /yQ+d8Xns31YQVC4hVT9HVvOHtZwYx13xHjEdIJtYm8ZO5g8Zwh7UHmqYT30GfGj
 80VBFOw0f0+Xcj7cSCkfD2T8xQ5MQ5vA9S7vI25jHqP2+CEf8bJOi4JMLmf3Yjv5
 dJ3B17xSEN9iCI23bXD57IBLBjEi23zcSrQRmlkh1Q25wo2Ywb9j1rslVObU4Saz
 QyrW+2edf5OzwxRpYHBHN5Rm+0iYPweaEksyPrVB1wChD2bNuB//5ab2SjMZGLv7
 b9KFXKtbnjZ46x2F43HP67gkYirXRMrUl0nw== ;{id = 22682}
Jun  15 17:58:41 debian ods-signerd: [hsm] sign RRset[48] with key
 c578c5786de6003694e6909b1c02834b7b2c tag 22682
Jun  15 17:58:41 debian ods-signerd: [hsm] sign init:
 CKR_KEY_HANDLE_INVALID
Jun  15 17:58:41 debian ods-signerd: [hsm] error signing rrset with
 lib hsm
```

```
Jun  15 17:58:41 debian ods-signerd: [rrset] unable to sign
 RRset[48]: lhsm_sign() failed
```

An attempt to roll over to a new key is the logical solution. After the key rollover is done, the current key list is retrieved in order to verify that a new key is indeed in use (key state is *active*). The list of current keys also shows that ODS has detected the purged keys from the HSM:

```
root@debian:~/$ ods-ksmutil key rollover --zone example.com
root@debian:~/$ ods-ksmutil key list --verbose
SQLite database set to: /var/opendnssec/kasp.db
Keys:
Zone:                          Keytype:      State:    Date of next transition (to):
 Algorithm:  CKA_ID:                          Repository:                        Keyt
example.com                    KSK           ready     waiting for ds-seen (active)
 8          a8f88f6f65e6092a558147173c930076  SoftHSM NOT IN repository
example.com                    ZSK           retire    2012-06-30 13:00:07 (dead)
 8          c5786de6003694e6909b1c02834b7b2c  SoftHSM NOT IN repository
example.com                    ZSK           active    2012-09-16 00:00:07 (retire)
 8          2d15019baa11449c9bf46ac7689d914f  SoftHSM                            2260
```

However, such an attempt to recover manually from lost keys by issuing a key rollover fails as well:

```
Jun  15 18:05:01 debian ods-signerd: [hsm] unable to get key: key
 c5786de6003694e6909b1c02834b7b2c not found
Jun  15 18:05:01 debian ods-signerd: [zone] unable to publish dnskeys for
 zone example.com: error creating dnskey
Jun  15 18:05:01 debian ods-signerd: [tools] unable to read zone example.
 com: failed to publish dnskeys (General error)
```

The problem is that even though the Enforcer has rolled a new key, the Signer still thinks that the lost old key is available in the HSM and tries to publish it which results in failure.

The probability of such an error situation where keys get lost from HSM is difficult to estimate. HSM's are built with security and reliability in mind which should eliminate any chance that private keys get corrupted or leak out of the system. Some tamper-resistant security modules (TRSM) implement Tamper-Responsive features[15] - they can self-destruct or deny access to private encryption material if a form of tempering is detected. That means that even a single accidental drop of the TRSM can force to make its contents unavailable. That is a possibility that ODS should gracefully handle by introducing new keys (as soon as the security module is replaced). This has great impact to system administrators since recovering from such situation is not trivial. A radical approach to recover is to reset all ODS metadata in the database and restart the ODS daemons. This is achieved by calling the following three commands:

```
root@debian:~/$ ods-control stop
Stopping enforcer...
Stopping signer engine...
Engine shut down.
root@debian:~/$
root@debian:~/$ ods-ksmutil setup
*WARNING* This will erase all data in the database; are you
 sure? [y/N] y
fixing permissions on file /var/opendnssec/kasp.db
zonelist filename set to /etc/opendnssec/zonelist.xml.
kasp filename set to /etc/opendnssec/kasp.xml.
Repository SoftHSM found
No Maximum Capacity set.
RequireBackup NOT set; please make sure that you know the
 potential problems of using keys which are not recoverable
INFO: The XML in /etc/opendnssec/conf.xml is valid
INFO: The XML in /etc/opendnssec/zonelist.xml is valid
INFO: The XML in /etc/opendnssec/kasp.xml is valid
WARNING: In policy default, Y used in duration field for
 Keys/KSK Lifetime (P1Y) in /etc/opendnssec/kasp.xml - this
 will be interpreted as 365 days
WARNING: In policy lab, Y used in duration field for
 Keys/KSK Lifetime (P1Y) in /etc/opendnssec/kasp.xml - this
 will be interpreted as 365 days
Policy default found
Info: converting P1Y to seconds; M interpreted as 31 days,
 Y interpreted as 365 days
Policy lab found
Info: converting P1Y to seconds; M interpreted as 31 days,
 Y interpreted as 365 days
```

```
Zone example.com found; policy set to lab
Added zone example.com to database
root@debian:~/$
root@debian:~/$ ods-control start
Starting enforcer...
OpenDNSSEC ods-enforcerd started (version 1.4.0a2), pid 8125
Starting signer engine...
DEBUG: timeshift mode enabled, but not set.
OpenDNSSEC signer engine version 1.4.0a2
Engine running.
```

Even though, creating new keys is the logical action to do when old keys have been lost, it does not fully mitigate the risk that some DNSSEC-validating resolvers might fail to validate a zone signed with OpenDNSSEC. The problems stems from the fact that a new key is introduced without performing a key rollover and is further investigated in chapter Optimum TTL settings.

Chapter 4

Optimum TTL settings

Chapter Test cases describes several error scenarios which can cause OpenDNSSEC to produce bogus DNSSEC zone. This can happen as a result of:

- not signing the zone at all
- not signing the zone on time
- or signing the zone with invalid information

Even though mitigations have been presented to handle most of these error cases, two of them - Singer crash and HSM crash - lead to a risky situation in which DNSSEC-validating resolver can fail to validate a DNS zone. This chapter tries to calculate that risk and find out how it can be minimized. Since a crashing component or a crashing HSM is a problem for any DNSSEC software or DNSSEC infrastructure, this chapter leaves the context of OpenDNSSEC and presents a generic solution applicable to any DNSSEC error situation.

The research in this chapter is founded upon the definition of *risk* presented in section Risk - definition and factors and couple of case assumptions documented in the section Assumptions. Section Example case explains the mathematical logic in calculating the risk of publishing a bogus zone when no key rollover is performed through means of an example where a DNS key is assigned a TTL of 4 time units[1] and a signature signed with that key

[1]The particular time unit used is irrelevant - seconds, minutes, hours, etc can be used to produce the same results

is given a TTL of 6 time units. In the end, section Results presents what TTL values minimize the risk of a bogus zone when no key rollover has been done.

4.1 Risk - definition and factors

When a HSM or OpenDNSSEC Signer crashes, the situation will be noticed sooner or later. Once detected, the problem normally gets solved by replacing and/or restarting the faulty component. However, if

- the ODS Signer has crashed and the ODS Enforcer has performed a key rollover in the meanwhile
- or the HSM has been replaced with a new one without being able to retain old keys

then a DNSSEC zone needs to be resigned with a new key and published without performing a key rollover in advance. When the Signer has crashed, it needs to be started again as if no crash has occurred at all. The result is that validating resolvers might see in one moment a DNS zone signed with one key and in the next moment the same DNS zone signed with another key which introduces the risk that they might have:

- cached the old key and see signatures signed with new key
- cached signatures signed with the old key and see the new key

Any of the aforementioned situation results in resolvers failing to validate the DNS zone and declaring it as *bogus*. The key element which defines when each of the two situations occur (or do not occur at all) is the time-to-live(TTL) value assigned to the published key (TTL of DNSKEY record for zone signing keys and the DS record for key signing keys) and the published signatures signed with that key (TTL value of RRSIG records of various DNS data, e.g. RRSIG A, RRSIG MX, etc.). Thus, this chapter investigates how the latter two TTL values influence the risk of resolvers failing to validate a zone when a new key or new signatures are introduced without performing a key rollover.

In the context of information technology, *risk* is formally calculated by:

$$Risk = Likelihood * Impact[16] \tag{4.1}$$

Likelihood stands for the probability of an undesirable event to happen and *impact* represents the scale of the negative effect which such an event would cause to a given organization. In the scope of this project, the value for *impact* is regarded as the time it takes until an undesirable situation is recovered. That is, the time it takes until all DNSSEC resolvers see a valid zone again after it has been signed with a new key and published without performing a key rollover. Hence, that is the point in time when the resource with the higher TTL expires in validators' cache and they all start to use the new key and the new signatures to validate a particular zone.

4.2 Assumptions

In order to be able to calculate the number of validators which declare a zone as *bogus* in the aforementioned situation, a couple of assumptions have been made:

1. the number of client (or end-host) validators is infinite
2. the remaining time in resolvers' cache for each record is uniformly distributed

The first assumption represents a "worst-case" scenario in which validating DNSSEC resolvers are constantly queried by indefinite number of client resolvers. This is the case of an "indefinitely popular" zone. This ensures that DNS records (of DNS data, keys and signatures) in validating resolvers' cache are refreshed as soon as they expire. This assumption helps to exclude the number of validating resolvers which might not see a *bogus* zone in any further calculations since they still have not refreshed their cache and are oblivious to the newly introduced key.

The second assumption simplifies calculations. For example, it ensures that the number of validators which cache one of the two resources (key or signature) is the same as the number of validators which cache the other of the two resources at any given point in time.

4.3 Example case

The previous section of this report talked about several assumptions which are taken when calculating what TTL values should be used in order to minimize the risk of DNSSEC validators to report a zone as bogus when new DNSSEC encryption keys or signatures are introduced without performing a key rollover. This section explains the mathematics behind the calculations through means of an example with two particular TTL values for a key and signatures made with it.

Based on the two assumptions, figure 4.1 has been created using Gnuplot. It illustrates the percentage of of validating resolvers which have cached the two resources in any given point in time when TTL's of 4 and 6 have been used as an example. It makes no difference which of the two resources - a key or a signature - is assigned with which of the two values - 4 or 6 because a zone is bogus when either of the two expire before the other one has expired as well. Hence, for simplicity in this chapter the two records are simply referred as "Resource 1" and "Resource 2" without any further distinction.

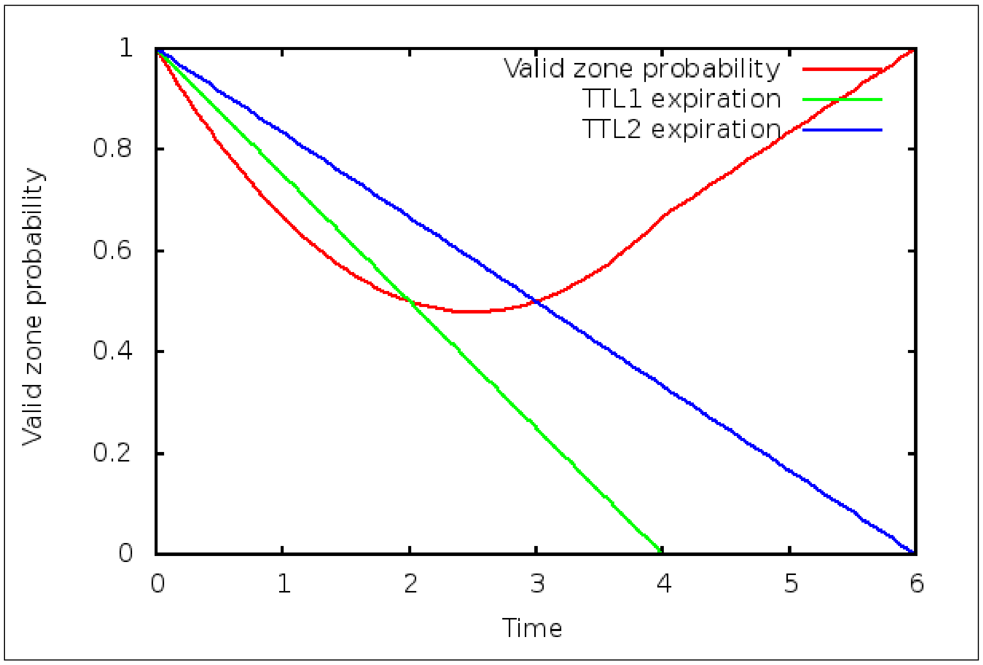


Figure 4.1: Probability of a DNSSEC-resolver seeing a zone as valid at a given time x when TTL1=4, TTL2=6

The x axis represents the advancing time and $x = 0$ is the moment when a new key is introduced and caches of old resources start to expire. The y axis represents the number of resolvers which have cache a particular resource. Thus, based on assumption (1), $y = 1$ when $x = 0$ which means that prior to the moment in time when a new key or signatures are introduced, all resolvers have the old key and the old signatures in their cache. In general, there are four combinations for how two old resources and their corresponding two new versions can be cached inside resolvers' cache. Table 4.1 presents those four combinations together with the mathematical functions for calculating the number of resolvers that fall into each combination.

	Cached	Not cached
Resource 1	$l(x)$	$1 - l(x)$
Resource 2	$b(x)$	$1 - b(x)$

Table 4.1: Number of resolvers which have cached two resources, or not

The green line on figure 4.1 represents the part of validating resolvers which have cached resource 1 with TTL 1. The line starts from point with coordinates $(0, 1)$, ends at point $(TTL1, 0)$ and is plotted using function (4.2):

$$l(x) = 1 - \frac{x}{TTL1} \tag{4.2}$$

The blue line on figure 4.1 represents the part of validating resolvers which have cached resource 2 with TTL 2, respectively. Similarly to (7.2)eq:l

$$b(x) = 1 - \frac{x}{TTL2} \tag{4.3}$$

The red line on the figure 4.1 represents the part of resolvers which have

- cached both old resources (column 1 in table 4.1)
- cached both new resources, thus both old resources have expired and have been replaced with their new versions (column 2 in table 4.1)

Hence, the formula for the red line is the function $f(x)$ (4.4) of the formulae in table 4.1:

$$f(x) = l(x) \times b(x) + (1 - l(x))(1 - b(x)) = 2 \times l(x) \times b(x) + 1 - l(x) - b(x) \tag{4.4}$$

Note that after the point in time when the resource with lower TTL expires all DNSSEC validating resolvers have updated that resource with its new version. From the example on figure 4.1, this is the moment in time when $x = 4$. Thus, it is possible that a zone is rendered as bogus only until the second resource's TTL expires as well in the same validator (when $x = 6$ on figure 4.1). This is the reason why the red curve turns into a straight line after the lower TTL has expired. The part of validators which still cache the resource with the higher TTL value can be calculated by subtracting the part of validators in which the latter resource has expired (using the formula in table 4.2, row 3, column 3) from the total amount of validators (equal to 1). This calculation results in formula (4.3):

$$g(x) = 1 - (1 - \frac{x}{TTL2}) = \frac{x}{TTL2} \tag{4.5}$$

The two formulae $f(x)$ (4.4) and $g(x)$ (4.5) are both valid for calculating the part of all resolvers which might see a valid zone. However, these two forumalae are only applicable in their own domains, that is:

- formula $f(x)$ (4.4) is applicable from the moment when new versions of a DNS key and its corresponding signatures are introduced ($x = 0$) until the moment when the resource with lower TTL expires ($x = 4$)
- forumala $g(x)$ (4.5) is applicable from the moment when the resource with lower TTL expires ($x = 4$) until the moment when the resource with higher TTL expires ($x = 6$)

4.4 Results

Section Example case has explained how probability that resolvers see a valid zone when a zone's public key or signatures made with that key expire, can be calculated for any point in time. This section looks at how to calculate the sum of such probabilities during the total lifetime of the two DNS resources in the resolvers' cache.

Formula (4.4) calculates the probability of resolvers to see a zone as valid at any given point in time x. Actually, the probability that resolvers see a zone as bogus can be deduced by subtracting the probability for a valid zone from 1. That hints to say that the area in figure 4.1 which is enclosed above the red line and the axis at $y = 1$ represents the risk that resolvers will see a bogus zone when its key and signatures are introduced without performing a key rollover. Comparing the area above the red line for all possible TTL values helps to distinguish which combinations of TTL values result in greater or smaller risk for DNSSEC-validators seeing a published *bogus* zone.

The area below the red line is calculated by using the integration of function (4.4) and is denoted with formula $F(x)$ (4.6). Note that function $f(x)$ (4.4) is first expanded with functions $l(x)$ (4.2) and $b(x)$ (4.3) and that L denotes the value of TTL1 and B - the value of TTL2:

$f(x) = 2 \times l(x) \times b(x) + 1 - l(x) - b(x) = 1 - \frac{x}{L} - \frac{x}{B} + \frac{2 \times x^2}{L \times B}$

$$\int f(x)\mathrm{d}x = F(x) = x - \frac{x^2}{2 \times L} - \frac{x^2}{2 \times B} + \frac{2 \times x^3}{3 \times L \times B} \tag{4.6}$$

Mind, however, that function $f(x)$ (4.4) is used to plot the red line up to the point when TTL1 expires. After that point in time, the function $g(x)$ (4.5) is used and the area it encloses with the x axis is found integrating the function $g(x)$:

$$\int_0^\infty g(x)\mathrm{d}x = G(x) = \frac{x^2}{2 \times B} \tag{4.7}$$

In order to find the two areas (before and after the expiration of the resource with lower TTL), the two integrated functions have to applied with the proper boundaries. Hence, the formula for finding the total area is brought down to the following:

$$H(x) = \int_0^L f(x)\mathrm{d}x + \int_L^B g(x)\mathrm{d}x = F(L) - F(0) + G(B) - G(L) \tag{4.8}$$

Using formula $H(x)$ (4.8) one can calculate the possibility of DNSSEC validators seeing a valid zone when any combination of TTL values are used. The following table 4.2 presents the calculations for combinations of TTL values from 1 to 10. It has been generated using a python script which is available in section ttl-risk.py from the Appendix chapter.

B/L	1	2	3	4	5	6	7	8	9	10
1	0.67	1.33	1.89	2.42	2.93	3.44	3.95	4.46	4.96	5.47
2	1.33	1.33	2.06	2.67	3.23	3.78	4.31	4.83	5.35	5.87
3	1.89	2.06	2.0	2.75	3.4	4.0	4.57	5.13	5.67	6.2
4	2.42	2.67	2.75	2.67	3.43	4.11	4.74	5.33	5.91	6.47
5	2.93	3.23	3.4	3.43	3.33	4.11	4.81	5.46	6.07	6.67
6	3.44	3.78	4.0	4.11	4.11	4.0	4.79	5.5	6.17	6.8
7	3.95	4.31	4.57	4.74	4.81	4.79	4.67	5.46	6.19	6.87
8	4.46	4.83	5.13	5.33	5.46	5.5	5.46	5.33	6.13	6.87
9	4.96	5.35	5.67	5.91	6.07	6.17	6.19	6.13	6.0	6.8
10	5.47	5.87	6.2	6.47	6.67	6.8	6.87	6.87	6.8	6.67

Table 4.2: Possibility for DNSSEC validators to see a valid zone when TTL values from 1 to 10 are used

Table 4.2 reveals that when e.g. TTL value of 10 is used for one of the DNS resources, the TTL value of 7 or 8 results in a greater chance (*likelihood*) that resolvers will see a valid zone. The same pattern can be observed for the other combinations of TTL values. It suggests that if administrators use TTL values of ratio $\frac{3}{4}$ for the public DNSSEC key record and signature records produced with that key, they can minimize the undesired risk of having a zone rendered as bogus by validating resolvers when it has been resigned with new a key and published without performing a key rollover.

In order to visualize the contents of table 4.2, the heat map plot 4.4 has been generated with Gnuplot. The code to produce this plot can be found in section heatmap.gpi from the Appendix chapter of this report.

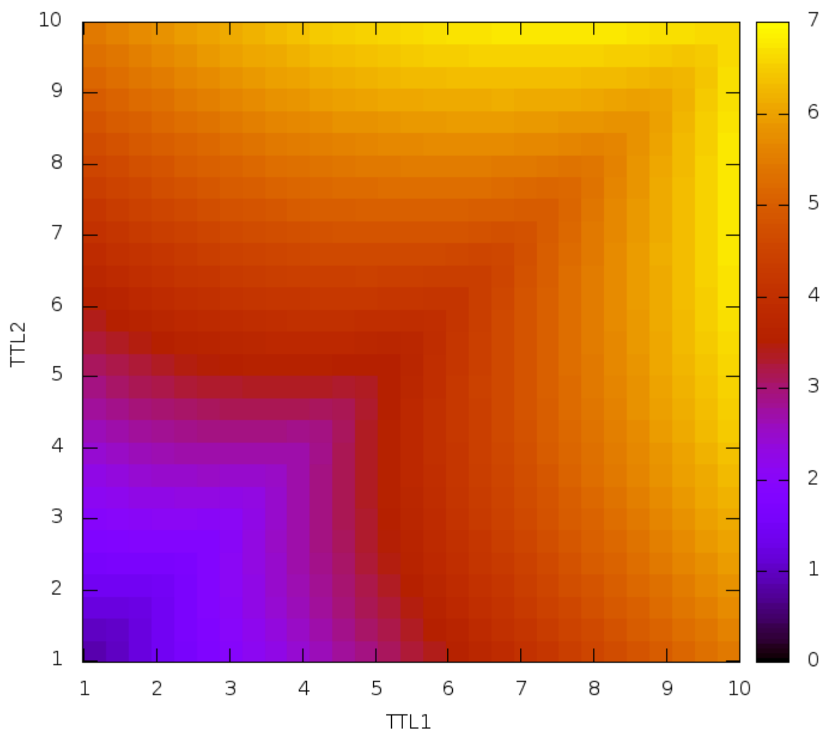


Figure 4.2: Possibility of DNSSEC validators seeing a valid zone when TTL values from 1 to 10 are used

The plot depicts the results from table 4.2 in the form of a heat map and shows once again that the relative number of DNSSEC-validating resolvers which might fail to validate a zone when a new encryption key is introduced without performing a key rollover, is minimal when TTL values with ratio $\frac{3}{4}$ are used for the DNS public key record and the records of signatures generated with that key.

Notice it is not easy to see on figure 4.4 that the $\frac{3}{4}$ ratio used for TTL values indeed contributes to lower risk for resolvers noticing a bogus zone. This can be facilitated by plotting figure 4.3.

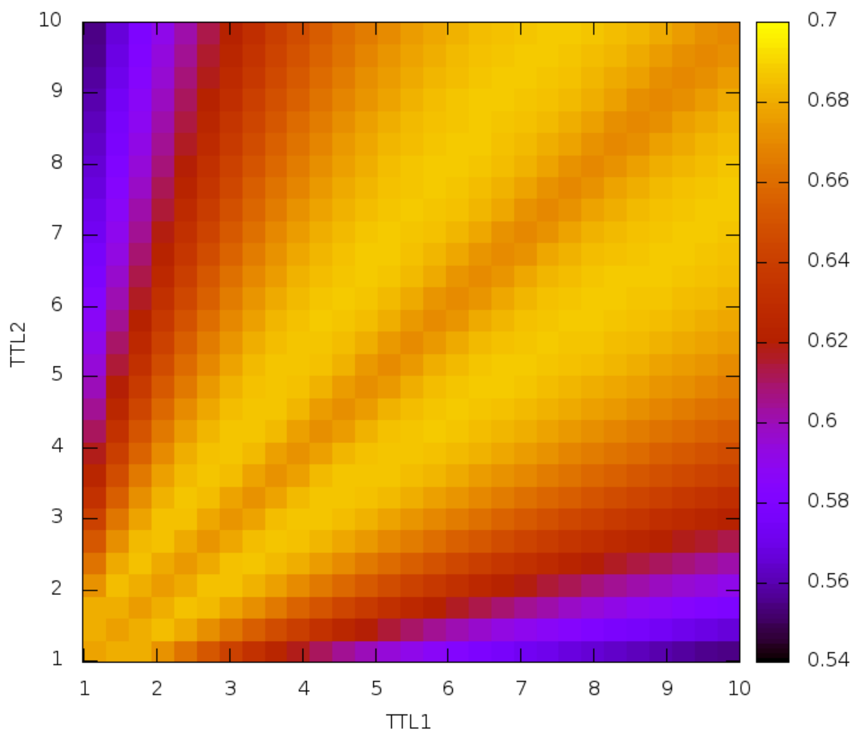


Figure 4.3: Normalized value of the risk that DNSSEC-validators might see a bogus zone when TTL values from 1 to 10 are used

The figure uses formula $N(x)$ (4.9) which is the normalized version of formula $H(x)$ (4.8) over the total area on figure 4.1. The latter is found by multiplying the maximum values of x and y. Since the maximum value of y equals to 1, the total area is effectively the value of the higher TTL value B.

$$N(x) = \frac{H(x)}{B} \tag{4.9}$$

Note that figure 4.3 introduces a strange artefact. It makes it seem that the risk of a bogus zone is the same when pairs of big or small TTL values are used. It looks like there is no difference when e.g. 3 and 1 TTL values are used compared to 10 and 7. Of course this is not true in reality. The lower the TTL values of DNS records are, the faster the records are refreshed, and thus the lower the risk becomes of a zone being noticed as bogus by validating resolvers. The problem of introducing a new encryption key without doing a key rollover can be totally mitigated if DNS caching is switched off, i.e. all records' TTL values are set to 0. This, however, is impossible in practice since it will surely raise the load on authoritative DNS servers.

The probability for DNSSEC validators marking a zone as bogus when it has been signed with a new key without performing a key rollover can be found by subtracting the value of formula (4.8) (for a given pair of TTL values L and B) from the value of the total area in figure 4.1. Such calculation is expressed in formula $I(L, B)$ (4.10) knowing that the value of the total area is equal to $y \times B$ when $y = 1$ and B is the higher TTL value:

$$I(x) = (1 \times B) - H(x) \tag{4.10}$$

Based on formula (4.1) the actual *risk* can be calculated where the probability of an undesirable event to happen is expressed by formula $I(x)$ (4.10) and the *impact* of such event is equal to the higher TTL value B. Table depicts the calculated risk when TTL values from 1 to 10 are used.

B/L	1	2	3	4	5	6	7	8	9	10
1	0.3	1.3	3.3	6.3	10.3	15.3	21.3	28.3	36.3	45.3
2	1.3	1.3	2.8	5.3	8.8	13.3	18.8	25.3	32.8	41.3
3	3.3	2.8	3.0	5.0	8.0	12.0	17.0	23.0	30.0	38.0
4	6.3	5.3	5.0	5.3	7.8	11.3	15.8	21.3	27.8	35.3
5	10.3	8.8	8.0	7.8	8.3	11.3	15.3	20.3	26.3	33.3
6	15.3	13.3	12.0	11.3	11.3	12.0	15.5	20.0	25.5	32.0
7	21.3	18.8	17.0	15.8	15.3	15.5	16.3	20.3	25.3	31.3
8	28.3	25.3	23.0	21.3	20.3	20.0	20.3	21.3	25.8	31.3
9	36.3	32.8	30.0	27.8	26.3	25.5	25.3	25.8	27.0	32.0
10	45.3	41.3	38.0	35.3	33.3	32.0	31.3	31.3	32.0	33.3

Table 4.3: Risk for DNSSEC validators to see a bogus zone when TTL values from 1 to 10 are used and a zone is resigned with a new key without performing a key rollover

Table 4.3 suggests that if administrators use TTL values of ratio $\frac{3}{4}$ for the public DNSSEC key record and signature records produced with that key, they can minimize the undesired risk of having a zone rendered as bogus by validating resolvers when it has been resigned with new a key and published without performing a key rollover. In order to visualize the contents of table 4.3, the heat map plot 4.4 has been generated with Gnuplot.

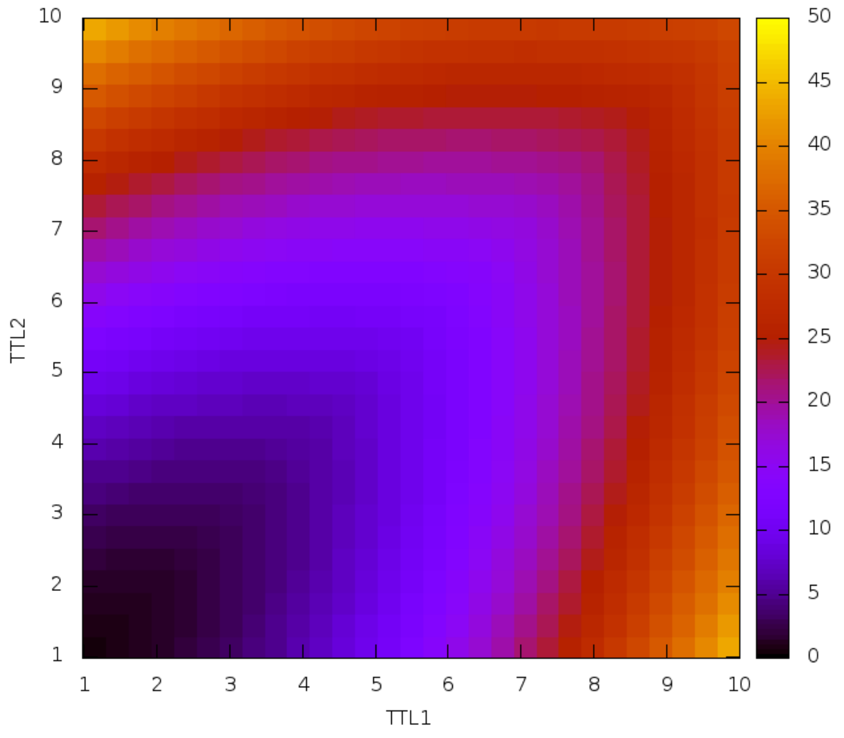


Figure 4.4: Risk of DNSSEC validators seeing a bogus zone when TTL values from 1 to 10 are used

The research in this chapter has looked into the undesired effects when DNSSEC signing components crash (for one or another reason) and DNS zones need to be resigned with a new encryption key without performing a key rollover. Combinations have been investigated when old encryption keys or signatures expire at different points in time inside DNSSEC validators' cache. This might cause validators to mark a popular zone as bogus because if a query for such a zone occurs right after one of the two (a key or a signature) expires, a validator will try to:

- validate an old signature with a new key or...
- use an old key to validate an updated version of a signature

In the both aforementioned cases of mismatching keys and signatures, a DNSSEC validator reports the zone as bogus and stops serving it to any (end-host) resolvers. Such an event can be catastrophic to enterprises because all services which they serve through DNS records become unavailable.

Investigation has shown that the risk for such undesirable situation to happen can be reduced if DNS administrators assign TTL values which have the ratio of $\frac{3}{4}$ to DNS signatures and their corresponding keys. Moreover, such a recommendation is applicable not only for DNSSEC zone signing, but also to any public-key infrastructure where public keys and signatures have a limited timelife and need to be cached at intermediate validators. The function of such validators is vital for providing security to end-host resolvers which, in the case of DNSSEC, still do not often perform signature validation themselves.

Chapter 5

Conclusion and recommendations

This research project has tested the performance of OpenDNSSEC during key rollovers and explored a number of error cases which OpenDNSSEC is supposed to handle gracefully without causing a zone to be seen as bogus by validating DNSSEC resolvers. All described error cases represent examples of:

Changes in the environment which ODS interacts with These can be caused by either faulty system processes or manual human mistakes. The term "environment" in this project's scope includes the contents of configuration files, the database and the system date which ODS interacts with.

Crashing ODS components Whenever one of the ODS core processes fails due to a software bug or a natural disaster

This chapter summarizes the tested error cases and presents points for improvement of the operation of OpenDNSSEC.

5.1 Resilience against ODS environment changes

The components of OpenDNSSEC communicate with each other in several ways. One of them is the use of *zone signing configuration* files which are generated by the ODS Enforcer and read by the ODS Signer. This project presents a case in which the latter configuration files are not validated by the Signer and can lead to ODS generating a bogus zone without even warning the user. Hence, it is advisable that newer versions of ODS sanitize the content of *zone signing configuration files* before using it as an input for the Signer.

Another error case related to changes in the environment of ODS which this project highlights, is when the system date changes relative to the date when a zone has been signed by ODS. In such a case, a zone needs to be resigned straight away if its DNS signatures have been rendered as expired compared to the new value of the system date. The experiment proved that the new version of ODS (2.0) monitors date changes and acts accordingly whereas the older tested version (1.4.0a2) experiences problems and thus, can produce a bogus zone.

A date change is a common event in any production system. The date is normally adjusted on regular intervals against a central NTP service. However, daylight saving time can also trigger system date updates. Such a case has not been tested in this research project but it should be addressed by ODS developers. In general, a recommendation is made for a test case in ODS which identifies any interaction of ODS with the system environment and reports how changes in the environment variables influence the behaviour of ODS.

This research project also concludes that both OpenDNSSEC versions do not monitor the signed zone files which they produce. System administrators are assumed to never edit those files manually. OpenDNSSEC is built upon this assumption and therefore does not monitor for changes in the signed zone files. It is arguable whether such functionality is strictly required in ODS itself. That is why, subsection Signed zone files recommends the use of a DNSSEC proxy, such as CreDNS. System administrators should deploy it between OpenDNSSEC and any public DNS server which serves zones from OpenDNSSEC in order to prevent erroneous zone files getting published to public DNS servers.

5.2 Resilience against crashing components

Subsection HSM crash proves that OpenDNSSEC cannot recover automatically from losses of encryption keys in the HSM. Hence, if an HSM has been destroyed and replaced with a new one, ODS does not introduce new keys and does not initiate a key rollover if signatures produced with any old keys have expired. The way to manually recover from such situation is to clear ODS' key state metadata which resides in the database.

ODS warns the user with an error message whenever it detects that it is working with keys which are not backed up. However, future versions of ODS should also be able to recover gracefully by introducing new encryption keys when unbacked keys are lost. The introduction of such new keys can be performed with a prepublication rollover. However, signatures made with a lost key should be retained in a DNS zone until they have expired in all validators' caches (based on the signatures' TTL value).

5.3 Optimum TTL settings

All performed test scenarios during this project have led into a more theoretical research not limited to OpenDNSSEC only but applicable to DNSSEC in general. Its purpose has been to investigate an error situation which cannot be recovered automatically by DNSSEC-signing software, such as when key components crash (as is the case when ODS Signer crashes). In that case the software is simply restarted which may lead to the introduction of new encryption keys and record signatures without making sure that old keys and signatures have expired from the cache of all DNSSEC-validating resolvers if:

- ODS Signer component has crashed and ODS Enforcer component has rolled a new key in the meantime
- private key material has been lost from the HSM

In order to minimize the impact from one of the two aforementioned situations, system administrators are advised to assign TTL values with $\frac{3}{4}$ ratio to record signatures and their corresponding public keys, or vice versa. In other words, the TTL value of a public key DNS record should be three quarters the TTL value of DNS record signatures generated with that key, or the other way around.

5.4 Framework for visualizing ODS key states

Last but not least, section Key rollovers presents a framework for monitoring the key states which ODS assigns to encryption keys. The framework generates a time line with all used keys and their states and has been used to verify that a key in an *active* state (which is used by ODS for signing a DNS zone) is always available during a ZSK prepublication rollover. The framework can be used by:

- system administrators to monitor the process of key rollovers
- by researchers and ODS developers to verify that other key rollover schemes, supported by ODS, are performed correctly

Chapter 6

Future work

This project has proved several cases in which OpenDNSSEC can be improved in order to handle error situations better. However, it is unknown whether more scenarios exist which allow an incautious system user or a faulty system process to prevent ODS from signing a DNS zone correctly. In this chapter several more cases are advised to be tested. Apart from that, some questions which have not been answered in this report due to either limited scope or time constraints, have been posed for further research.

6.1 Improvements for the ODS key states visualizing framework

Section Key rollovers presents a tool which helps users to monitor assigned key states of encryption keys during a zone signing *prepublication* key rollover with OpenDNSSEC. The tool has been used to verify that at any moment in time during the aforementioned rollover, there exists one key in the *ready* state which can be used for signing a DNS zone. However, other available key rollover schemes have not been tested due to time constraints:

- Key signing key rollovers
  - KSK Double RRset
  - KSK Double DS
  - KSK Double Signature

- Zone signing key rollovers
  - Double Signature
  - Double RRsig
- Combined signing key rollovers
  - DoubleRRset
  - Single Signature
  - Double DS
  - Double Signature
  - PrePublication

Additionally, the key states plotting tool can be significantly optimized. Currently, it polls key states information while the Enforcer is running uninterruptedly. This requires that the tool is ran for at least a period of time which is greater than one key rollover period. Instead, use of the *ENFORCER_TIMESHIFT* environment variable (mentioned in chapter OpenDNSSEC) can be implemented in order to request key states from the Enforcer for future moments in time. This way monitoring of key rollovers will require much less time to complete.

6.2 ODS architecture consideration

Several tested error cases in chapter Test cases have hinted problems related to the architecture of OpenDNSSEC, that is, the functional separation of the Enforcer and Signer components. Subsection Zone signing configuration file recommends that input validation functions should be implemented into both components so that their intercommunication cannot be influenced by external to ODS processes. Subsections Enforcer crash and Singer crash prove that such a functional separation cannot prevent ODS to publish bogus zones. Further and more profound investigation in this direction is required in order to review ODS's architecture design and propose improvements.

6.3 Proof for the optimum TTL settings

Chapter Optimum TTL settings investigated how the risk of resolvers seeing a bogus zone can be minimized when ODS processes crash and need to be restarted. It proved that such a risk is minimal when the assigned TTL values for public key records and DNS record signatures have the ratio of $\frac{3}{4}$. Such a recommendation has been based solely on mathematical functions. However, more profound research is necessary to verify whether such theory really applies in practice and for all TTL values. The latter can be proven by normalizing function $H(x)$ (4.8) from chapter Optimum TTL settings with B, finding its derivative to L or B and solving it when it equals to zero. The expected result is that $L = \frac{3}{4} \times B$ or $B = \frac{3}{4} \times L$.

6.4 Risk analysis

The term *risk* has been used throughout this report in order to denote, in a number of error cases, the negative impact of:

- ODS publishing a bogus zone and...
- such a bogus zone reaching the cache of DNSSEC-validating resolvers

Thus the impact of a particular error case for generating a bogus zone with ODS has been referred to as *risk* whereas, formally, the term risk also relates to the probability of an undesirable event to occur. Hence, a risk analysis needs to be implemented to take into account the probability of each error case. Only then can the error cases be compared with each other and can be measured which one of them contributes to the greatest risk for OpenDNSSEC publishing a bogus zone.

Bibliography

[1] Rickard Bellgrim. Opendnssec training. https://wiki.opendnssec.org/download/attachments/590430/opendnssec.training.2012.03.pdf?version=1&modificationDate=1330681864000, referenced at https://wiki.opendnssec.org/display/DOCREF/Training+Videos+and+Study+Material, March 2012. Cited on pages 5 and 8.

[2] OpenDNSSEC development team. Opendnssec documentation on key states. https://wiki.opendnssec.org/display/DOCS/Key+States, November 2011. Cited on pages 5 and 16.

[3] US CERT United States Computer Emergency Readiness Team. Multiple dns implementations vulnerable to cache poisoning. http://www.kb.cert.org/vuls/id/800113, July 2008. Cited on page 6.

[4] Comcast. Analysis of dnssec validation failure. http://www.dnssec.comcast.net/DNSSEC_Validation_Failure_NASAGOV_20120118_FINAL.pdf, January 2012. Cited on page 6.

[5] P. Mockapetris. Domain names - concepts and facilities. http://www.ietf.org/rfc/rfc1034.txt, November 1987. Cited on page 7.

[6] P. Mockapetris. Domain names - implementation and specification. http://www.ietf.org/rfc/rfc1035.txt, November 1987. Cited on page 7.

[7] P. Vixie. Extension mechanisms for dns (edns0). http://www.ietf.org/rfc/rfc2671.txt, August 1999. Cited on page 7.

[8] P. Hoffman. The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa. http://www.ietf.org/id/draft-ietf-dane-protocol-23.txt, June 2012. Cited on page 9.

[9] O. Kolkman. Dnssec operational practices. http://tools.ietf.org/html/rfc4641, September 2006. Cited on pages 13, 15, 16, and 31.

[10] W. Mekking. Dnssec key timing considerations follow-up. http://tools.ietf.org/html/draft-mekking-dnsop-dnssec-key-timing-bis-02, July 2011. Cited on pages 13 and 16.

[11] RSA Laboratories. Pkcs#11: Cryptographic token interface standard. http://www.rsa.com/rsalabs/node.asp?id=2133, 1995. Cited on page 13.

[12] Johan Ivarsson. A review of hardware security modules. http://www.opendnssec.org/wp-content/uploads/2011/01/A-Review-of-Hardware-Security-Modules-Fall-2010.pdf, 2010. Cited on pages 13 and 32.

[13] The European Network and Information Security Agency (ENISA). Good practices guide for deploying dnssec. http://www.enisa.europa.eu/activities/Resilience-and-CIIP/networks-and-services-resilience/dnssec/gpgdnssec/at_download/fullReport, January 2010. Cited on page 15.

[14] Inc. SPARTA. Dnssec operations: Setting the parameters. http://www.dnssec-deployment.org/documents/SettingtheParameters.pdf, November 2009. Cited on page 30.

[15] PCI Security Standards Council. Security requirements for hardware security module (hsm). https://www.pcisecuritystandards.org/documents/PCI%20HSM%20Security%20Requirements%20v1.0%20final.pdf, April 2009. Cited on page 35.

[16] Gary Stoneburner. Risk management guide for information technology systems. http://csrc.nist.gov/publications/nistpubs/800-30/sp800-30.pdf, July 2002. Cited on page 38.

Chapter 7

Appendix

This chapter accommodates several scripts which have been developed in the course of this research project.

7.1 writekeystates.sh

This is a bash script which retrieves key state information from OpenDNSSEC version 1.4 until interrupted with SIGTERM signal. The script stores that information to a *.log* file with filename the current date in the current working directory.

```
#!/bin/bash

for ((;;))
do
  ods-ksmutil key list --verbose > `date +%Y%m%d-%H%
     M%S`.log
  sleep 60
done
```

7.2 plotkeystates.py

This is a python script which parses OpenDNSSEC key state information from *.log* files generated with writekeystates.sh. It then uses *mat-*

plotlib.pyplot to generate a time line plot for all collected keys and their states states.

```
#!/usr/bin/python
"""
This script will parse all files in the current
   directory that have a ".log" extension. These
   files should contain the output of "ods-ksmutil
   key list --verbose" command and their filename
   should represent the datetime when the command
   was called. They will be parsed to produce a 2-
   dimentional dictionary object which contains: key
    id -> key state -> date. Such a dictionary
   object is then used to plot the key states on a
   time line graph.
"""

globalKeyStates = dict()
globalStartDate = ""
globalEndDate = ""

#keylistcmd = subprocess.Popen(['ods-ksmutil','key
   ','list','--verbose'], stdout=subprocess.PIPE,
   stderr=subprocess.PIPE)
#keylistcmd.wait()
#lines = keylistcmd.stdout.read().splitlines()

def printKeyStates():
  print("Start date: "+str(globalStartDate))
  print("End date  : "+str(globalEndDate))
  for keyId in globalKeyStates.keys():
    print(keyId+" ("+str(len(globalKeyStates[keyId].
       keys()))+" states)")
    for keyState in globalKeyStates[keyId].keys():
      print(" "+keyState.ljust(7)+": "+str(
         globalKeyStates[keyId][keyState]))

def parseLogFiles():
  import glob
  global globalStartDate, globalEndDate
  filenames = glob.glob('*.log')
  filenames.sort()
  for filename in filenames :
    date = filename.replace(".log","",1)
    date = convertTimestampToUnixtime(date)
    if globalStartDate == "" or globalStartDate >
       date:
      globalStartDate = date
    if globalEndDate == "" or globalEndDate < date:
      globalEndDate = date

    f = open(filename,'r')
    fileContents = f.read()
    f.close()

    parseKeyStates(date,fileContents)

  globalEndDate += 1

def parseKeyStates(date,odsKeyListOutput):
  import re
  global globalKeyStates

  lines = odsKeyListOutput.split("\n")

  #remove last and first 2 lines
  lines.pop(-1)
  lines.pop(-1)
  lines.pop(0)
  lines.pop(0)

  for line in lines:
    fields = re.split("  +",line)
    keyId = fields[8].strip()
    keyState = fields[2].strip()
    if keyId not in globalKeyStates:
      globalKeyStates[keyId] = {}
    if keyState not in globalKeyStates[keyId]:
      globalKeyStates[keyId][keyState] = {}
      globalKeyStates[keyId][keyState] = date
    #elif str(globalKeyStates[keyId][keyState]) <
       date:
    #  globalKeyStates[keyId][keyState] = date
    #else:
    #  print("ERROR: Earlier key state date is
       overwriting date in the future.",keyId,
       keyState,str(globalKeyStates[keyId][keyState
       ]),date)

######### Plotting #######
"""
Make a "broken" horizontal bar plot, ie one with
   gaps
"""

def makeTimeTuples(keyStates):
  times = []
  if "publish" in keyStates:
    times.append( (
      getKeyOrNextKeyStateTime(keyStates, "publish")
         ,
      getKeyOrNextKeyStateTime(keyStates,"ready") -
        getKeyOrNextKeyStateTime(keyStates, "publish
           ")
    ) )
  if "ready" in keyStates:
    times.append( (
      getKeyOrNextKeyStateTime(keyStates, "ready"),
      getKeyOrNextKeyStateTime(keyStates,"active") -
      getKeyOrNextKeyStateTime(keyStates, "ready")
    ) )
  if "active" in keyStates:
    times.append( (
      getKeyOrNextKeyStateTime(keyStates, "active"),
      getKeyOrNextKeyStateTime(keyStates, "retire")
         -
      getKeyOrNextKeyStateTime(keyStates,"active")
    ) )
  if "retire" in keyStates:
    times.append( (
      getKeyOrNextKeyStateTime(keyStates, "retire"),
      getKeyOrNextKeyStateTime(keyStates,"remove") -
      getKeyOrNextKeyStateTime(keyStates, "retire")
    ) )
  return times


def getKeyOrNextKeyStateTime(keyStates, keyState):
  if keyState == "publish":
    if "publish" in keyStates:
      return keyStates["publish"]
    else:
      return getKeyOrNextKeyStateTime(keyStates, "
         ready")
  elif keyState == "ready":
    if "ready" in keyStates:
      return keyStates["ready"]
    else:
      return getKeyOrNextKeyStateTime(keyStates, "
         active")
  elif keyState == "active":
    if "active" in keyStates:
      return keyStates["active"]
    else:
      return getKeyOrNextKeyStateTime(keyStates, "
         retire")
  elif keyState == "retire":
    if "retire" in keyStates:
      return keyStates["retire"]
    else:
      return globalEndDate
  else:
    return globalEndDate


def convertTimestampToUnixtime(timestamp):
  import time
  return time.mktime(time.strptime(timestamp, '%Y%m%
    d-%H%M%S'))
```

```
def getColor(stateName):
#  import re
#  statename = re.sub('[^a-z]','',statename)
  if stateName == "publish":
    return "yellow"
  if stateName == "ready":
    return "blue"
  if stateName == "active":
    return "green"
  elif stateName == "retire":
    return "red"
  elif statename == "dead":
    return "brown"
  else:
    print("ERROR [unknown state]: "+statename)
    return "black"


def makeColors(keyStates):
  colors= list()
  if "publish" in keyStates:
    colors.append( getColor("publish") )
  if "ready" in keyStates:
    colors.append( getColor("ready") )
  if "active" in keyStates:
    colors.append( getColor("active") )
  if "retire" in keyStates:
    colors.append( getColor("retire") )
  return colors


def plotKeys():
  import matplotlib
  matplotlib.use('Agg')
  import matplotlib.pyplot as plt

  print("Generating key states plot...")
  fig = plt.figure()
  fig.suptitle("ODS key states during pre-pub
     rollover")
  ax = fig.add_subplot(111)

  for index, keyId in enumerate(globalKeyStates.keys
     ()):
    times = makeTimeTuples(globalKeyStates[keyId])
    colors = makeColors(globalKeyStates[keyId])
    print("Debug [plot "+keyId+"]: "+str(times)+str(
       colors))
    ax.broken_barh( times , (index*2, 2), facecolor=
       colors  )

  #ax.set_xlim(start-100,end)
  ax.set_xlabel('Unix time')
  #ax.set_ylim(0,5 * len(globalKeyStates))
  ax.set_ylabel("Key tag")
  ax.set_yticks([ i*2+1 for i in range(len(
     globalKeyStates)) ])
  ax.set_yticklabels( [ keyId for keyId in
     globalKeyStates.keys() ] )
  ax.grid(True)

  p1 = plt.Rectangle((0, 0), 1, 1, fc="yellow")
  p2 = plt.Rectangle((0, 0), 1, 1, fc="blue")
  p3 = plt.Rectangle((0, 0), 1, 1, fc="green")
  p4 = plt.Rectangle((0, 0), 1, 1, fc="red")
  p5 = plt.Rectangle((0, 0), 1, 1, fc="brown")
  ax.legend((p1, p2, p3, p4, p5), ('publish','ready'
     ,'active','retire','dead'))

  fig.autofmt_xdate()
  plt.savefig("graph.png")
  plt.close()
  print("Done[graph.png].")


######### main #######
if __name__ == "__main__":
  parseLogFiles()
  printKeyStates()
  plotKeys()
```

7.3 ttl-risk.py

This is Python script which calculates the function $H(x)$ (4.8) for TTL values from 1 to 10 inclusively. This represents the relative number of DNSSEC resolvers which might fail to validate a zone for which a new encryption key has been introduced without performing a key rollover.

```
#!/usr/bin/python

def l(x):
  return 1-x/L
def b(x):
  return 1-x/B
def f(x):
  return 2*l(x)*b(x)+1-(1-l(x))-(1-b(x))
def F(x):
  return (2*(x**3))/(3.0*L*B) - (x**2)/(2.0*B) - (x
     **2)/(2.0*L) + x
def g(x):
  return x/B
def G(x):
  C = max(L,B)
  return x**2/(2.0*C)

cellWidth = 4
maxRows = 11
maxCols = 11

# Print header row
print "B/L".center(cellWidth),"|",
for B in range(1, maxCols):
  print str(B).center(cellWidth),"|",
print

# Print data
for B in range(1, maxRows):
  print str(B).center(cellWidth),"|",
  for L in range(1, maxCols):

    # Integral approximation for goodness
    area = F(L) - F(0) + G(B) - G(L)
    if B<L:
      area = F(B) - F(0) + G(L) - G(B)

    total = max(B,L)
    #area = area / total #normalized (relative)
       goodness
    #area =  total - area #absolute badness

    print str(round(area,2)).center(cellWidth),"|",
    if L==maxCols-1:
      print
```

7.4 heatmap.gpi

This is the Gnuplot code used to generate graph 4.4 in chapter Optimum TTL settings. It plots the relative number of resolvers which might fail to validate a zone for which a new encryption key has been introduced without performing a key rollover.

```
max(x,y) = (x>=y)*x + (x<y)*y
min(x,y) = (x>=y)*y + (x<y)*x

F(x, L, B) = x - (x**2)/(2*B) - (x**2)/(2*L) + (2*x
   **3)/(3*L*B)
G(x, L, B) = x**2/(2*B)

# x=L
# y=B
H(x,y) = ((F(min(x,y), min(x,y), max(x,y))-F(0, min(
   x,y), max(x,y))) + (G(max(x,y), min(x,y), max(x,y
   ))-G(min(x,y), min(x,y), max(x,y))))

heatmap_size = 10
set isosample 30
set hidden3d

set view map
set output "report.heat.png"
set terminal png size 800, 800
set xlabel 'TTL1'
set ylabel 'TTL2'
splot [1:heatmap_size] [1:heatmap_size] [0:100] '++'
    using 1:2:(H($1,$2)):(H($1,$2)) with pm3d
```