Maintainability of the OpenDNSSEC 1.1 KASP Enforcer

OpenFortress*
digital signatures

Abstract:

This document details problems encountered with the KASP Enforcer component in OpenDNSSEC 1.1.x, with the purpose to formulate suggestions for improvement, aiding the long-term maintainability of this component.

Document descriptive information:

Title:	Maintainability of the OpenDNSSEC 1.1 KASP Enforcer
Author:	Dr.ir. Rick van Rein
Version:	1.0
Date:	October 18, 2010
ASN.1 oid:	N/A
Release:	For OpenDNSSEC developers
Legal status:	Informational, OpenFortress assumes no liability

Contents

1 Introduction

OpenDNSSEC has been designed as a DNSSEC implementation that is usable for current DNS operators without the need to study cryptography, or have an in-depth knowledge on DNSSEC. The intention is to empower DNS administrators worldwide to setup DNSSEC for their domains, and to join the leap forward in Internet security that is currently taking place.
The KASP Enforcer is the component that monitors policies, that is, cryptographic and timing requirements. If needed, the KASP Enforcer will instruct a separate signer daemon to start signing zones with other keys, include new key material, and so on.
Being the home of the majority of DNSSEC logic, the KASP Enforcer has had to live up to many requests from end users. Meanwhile, OpenDNSSEC itself is shifting its goals from an initial release for static zones such as TLDs to a dynamic environment suitable for registrars. All this is placing the KASP Enforcer under great stress, to a level that the original concepts underpinning its design are showing signs of wear and tear.
The following section identifies a number of issues in the architecture of the current KASP Enforcer that we ran into while setting up our deployment in a registrar environment. The section after that proposes a number of changes that could help to improve the structure of the KASP Enforcer in OpenDNSSEC versions 1.3 and beyond.
On a final note: we feel that a lot has been achieved in getting OpenDNSSEC off the ground and the current KASP Enforcer effort has played an important part in this. This document should therefore not be seen as criticism but rather as a 'way forward' to extend the scope of OpenDNSSEC such that it becomes usable for a wider audience and can really start to fulfill its push-button promise.

2 Issues in the 1.1.x KASP Enforcer

During the DNSSEC implementation at SURFnet, we ran into a number of issues most of which had their root cause in limitations of the current KASP Enforcer. We spent quite a bit of time examining the current design and source code. This section contains an overview of our findings.
The problems below should not be interpreted as complaints; SURFnet has made a deliberate decision to use OpenDNSSEC 1.1 in a way that was destined for 1.2 versions, so we were bound to run into problems. The reason for writing this is merely to suggest how the structure of the KASP Enforcer could be improved to achieve better maintainability in the long run.

2.1 Key states

While being processed by the Enforcer, keys go through a number of states. The states are a vital part of the functions of the Enforcer, but there is no state diagram detailing what each state represents, what causes each transition, when they occur and on whose initiative. This knowledge is currently spread throughout the Enforcer's code, making it hard for newcomers to modify it reliably.
In our initial patch for 2-phase key backup, we made an error that did not appear until we started doing multiple things at once; as it turned out, one process could leave keys in a temporary state which was then observed by the other process as a key in an unknown state. Although not problematic, this is an indication that the Enforcer has a (lack of) structure that makes it hard to determine whether a change has been made in all the right places.
The database stores key state information in multiple forms; there is an explicit state field, and there are fields that store the timestamp of transitions from one state to the next. This implies a danger of inconsistent data that would not have occurred without this form of data redundancy. Such inconsistencies are made likely because there is no key state diagram. If one piece of code looks at one piece of data and another piece of code looks somewhere else, then inconsistent data can cause unpredictable behaviour of the Enforcer, leading to strange behaviour, possibly including bugs.
This situation contrasts the wish for a maintainable and extensible solution. Like us, others can easily create modifications that endanger the proper operation of OpenDNSSEC.
Documentation is one aspect of solving this problem, especially if it is rigorously done. For instance, facts known about the data of a key in each of its states, the causes of state transitions and on where the initiative for a state change comes from. State diagrams and perhaps a few rules of data consistency (in the form of logic predicates) are useful tools to keep maintainers all looking in the same direction.
Another angle to solving this problem is to aim for nearness of all code that implements related program logic, possibly in a modular or object-oriented design. It would serve the code well if it implemented abstractions that communicated with other abstractions. If code related to the same program logic is bundled in

modules, it is hardly possible to overlook related code if it needs modifications. Encapsulation of objects can help to be sure that there cannot be any code elsewhere that is related.

2.2 Database abstraction

The Enforcer uses either SQLite or MySQL as its database. The code contains a home-brewn abstraction to generate queries. The current abstraction appears to serve mainly as a syntactical abstraction, but it does not fully succeed at that; for instance, the developer must clutter his mind with the question whether commas should be infixed between query parts, according to the SQL that is being constructed.

The query-generating layer does not manage to abstract from the type of database used, judging the code variations for the two types of database. An existing database abstraction could improve this and, if it is commonly used access abstraction such as ODBC, JDBC or DBI, it would actually make it simpler for new developers to get on board and work on the Enforcer.

Maintainability and extensibility would profit from replacing the current database abstraction with a generic database abstraction. In addition, as demonstrated in the following example, it could improve the reliability of the code, because most lessons have already been learnt in pre-existing database interface abstractions.

An interesting problem occurred when setting up OpenDNSSEC in the live environment at SURFnet, using master/master replication between two database nodes. In such a setup, MySQL adapts its policy for automatically generated primary keys; instead of numbering them sequentially, one node generates even numbers and the other generates odd numbers. The Enforcer however, assumed that primary keys would be generated consecutively, and has this assumption hard-coded in the software.

When not relying on particular databases but instead on an abstract interface, it is less likely that such assumptions will be made, and hard-coded. Working with less information (due to a more general interface) is usually trivial; in this case, either the Enforcer would insert records with assumed-fixed key values made explicit, or it would ask the database for the last auto-generated primary key value after insertion of a new record. Not being able to oversee all possible databases that could be plugged behind a generic API makes it more likely that the general intentions of the API will be the developing guideline.

2.3 Preparing for change

The Enforcer has been built for the initial purposes of OpenDNSSEC, which were quite modest in the beginning. To recall, it initially supported root zone signing, then moved on to the much larger TLDs and the upcoming 1.2 version is aimed at registrars and the dynamicity of zones being added and removed regularly. It has been difficult for the Enforcer to keep up with the growing dynamicity and other requirements in OpenDNSSEC, in part because the majority of the OpenDNSSEC logic is translated into a requirement for the Enforcer.

In an ideal world, the design of the Enforcer would have been prepared for all this change by actively seeking out possible future requirements. Unfortunately, time constraints in getting an 1.0 out the door have meant that there simply was no room for this/

In the practical world of today, the Enforcer has difficulties coping with policies and zones as they grow into more dynamic versions of themselves. One example is that there has not been early thought about policies with no zones, but the Enforcer is presented with this situation as logical consequence of the desire to add and remove zones dynamically. Shared key shenanigans would not have ended up in the code if the general conception had been that such future developments were going to come.

When making an effort to formalise the structures in the Enforcer, it would be good to try to design as much as possible of the foreseeable future developments into the design, and into assertion statements. Given that, the Enforcer's code can be refactored towards a future-ready code base that should be more straightforward to maintain and extend. Policy changes might have been predicted as a result of leaving zones signed for years and years, while cryptology advances and makes stronger demands on the cryptographic configuration of OpenDNSSEC.

2.4 Efficiency

There are concerns about the efficiency of the Signer and the Enforcer, especially with the roll-out of OpenDNSSEC to large-scale registrars in mind. The Signer has largely been updated to accommodate those parties, but the Enforcer is still a topic of concern.

The task of the Enforcer implies managing collections of policies, zones and keys. To do this, it employs a relational database. What the code does is accessing the database on a per-object basis, which is not what relational databases are designed for. The performance of the Enforcer would improve a lot if it approached the database using bulk queries. Even if these are apply to single objects in many cases, then at least the possibility has been created to combine many actions into one, and thereby gain performance gains for large-scale use.

Another point about efficiency is the structure of the database. Even if it is common practice to use numeric keys, this is not the only option for a database. Any data type that implements an equivalence relation[1] works as a key. Practical options in relational databases are strings and enumerated values.

There are several places where this could be helpful; the `state` field in `keypairs` is a candidate, and so are the `parameter_id` values in the table `parameters_policies`.

In terms of code, a query with non-numeric keys could look like

```
update parameters_policies
set value=1152
where parameter='bits'
and value<1152
```

[1] Thus answering questions like *is* `key1` *equal to* `key2`*?*

The advantage of such SQL statements is not only that it is easier to read; it also shares with XML the advantage of data model extensibility. Furthermore, it avoids the need to join with extra tables for views such as `PARAMETER_LIST` and that improves the efficiency of accessing such views.

2.5 Documentation

The documentation of the Enforcer is basically absent (i.e. the code is the documentation). To work with the Enforcer, one needs to interpret code and data structures. Given that the program logic is not concentrated in (logical) modules, this is an uncertain approach to making updates, unless one has seen the entire application (i.e.: you need to have the source 'in your head' comprehensively in order to be able to make changes without risking breaking the Enforcer).
Diagrams and descriptions would be welcome improvements. Given the rather formal responsibilities of the Enforcer, invariant propositions could also be very useful. Without going overboard by proving correctness in terms of pre/postconditions, it should not be difficult to formalise statements like these:

- All zones have at least one ZSK and at least one KSK at any time
- Policies only hold on to keys that are needed to service zones

These are just examples, but it is generally helpful to have statements of this kind to understand the way the Enforcer works. Such explicit assumptions can also help future maintainers to avoid troublesome situations.
Furthermore, when worked out with sufficient precision, these invariants can be transformed into assertion statements that are regularly checked by the Enforcer itself. Such assertions can be useful validation checks just before committing a transaction.
An extra benefit that invariants have is that they keep assumptions firmly rooted in the code, so that future maintenance cannot divert from them without causing an explicit error message that triggers thought on the conceptual design of the Enforcer. It is even possible to create invariants to guard a general concept, even if this has not actually been implemented accordingly.
In short, thorough documentation will not only improve the maintainability and extensibility of the Enforcer, but it can also help to achieve a higher level of reliability.

3 Suggestions

The previous section discussed a number of problems with the KASP Enforcer, and already suggested a few angles of improvement. This section suggests a few constructive changes that could improve the Enforcer in the areas of maintainability, extensibility, reliability, scalability and performance.

3.1 Documentation

Detailed documentation would greatly benefit the Enforcer. A few diagrams that would be helpful are a class diagram and state diagram.
In addition to these diagrams, invariants and state-specific predicates could help to make clear judgements about the intentions of the implementation, and avoid causing conceptual inconsistencies that lead to bugs.
Worked out in assertions, invariants and state-specific predicates have the added benefit of monitoring the consistency of the database and making it clearly recognisable if a code modification oversteps the boundaries and assumptions made in pre-existing code.

3.2 Localised program logic

The current code of the Enforcer consists of many pieces of code that each take on a specific task and work it out in detail. The concrete nature of the code means that program logic is replicated over many pieces of code. This is a problem for the maintenance of the code, because it demands changes to be made in many places, and in the concrete scope of many different applications. The problem causing this appears to be a lack of abstractions at the implementation level.
It would be useful to address each part of the Enforcer's design and to consider its conceptual meaning. This will be part of the documentation process anyway. It may prove necessary to refactor the code to actually reflect the concepts formulated if they are changed, and to keep pieces of related program logic together in one module. One suggestion is to use an object-oriented approach, where code is defined explicitly at a certain level of abstraction (documented and enforced in assertion statements) and held together as a modular unit that can be easily overseen during future changes. Such self-contained code modules are probably best implemented in objects.

3.3 Database

It would be an improvement to replace the current, home-made database abstraction with a widely used database abstraction layer.
Also, the Enforcer would be easier to modify after replacing numeric database keys with enumerated or string values, inasfar as they have a fixed meaning to the Enforcer. The Enforcer is then free to hardcode such strings without endangering extensibility, maintainability or reliability. The strings will make a lot of sense to new/outside developers.

3.4 Origin of active data

The OpenDNSSEC project needs to make a clear statement about the source of active data; when is XML leading and when is it the database; inasfar the signer lacks information that it must now fetch from the XML files, it could be good to consider extending the `.signconf` files with the extra needs, so they can be extracted from the database if that is considered the primary source of information.
It is probably a good idea to treat the database as the active source of information because, unlike the XML files, it can be straightforwardly replicated to other nodes without noticeable delay. Relying on database replication is usually a more robust choice than relying on home-grown file replication procedures.
When the project chooses to rely on the database it is also necesssary to determine if it is desirable to load XML data into the database when the Enforcer is started.

3.5 Key chain

A simplifying suggestion for the Enforcer is to adopt the additional concept of a keychain. In the current design, key management is attached to policies, which seems to be a mismatch that causes difficulties to implement functionality for all but the most static applications of OpenDNSSEC.

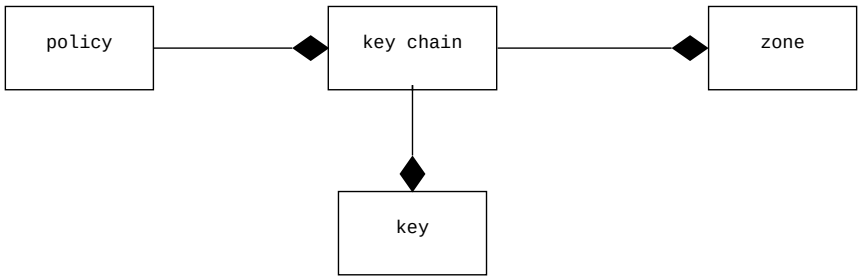

Figure 1: Introducing a missing concept can simplify the Enforcer.

In this alternate design, a policy is a specification of constraints to timing and cryptography; it would be sampled for information at moments that are yet to be defined, but have no other function. Zones would point to the policy that they aim to implement, possibly with a back reference to a previous policy if a zone is allowed to move between policies.
In addition, a zone would point to a keychain. When a policy prescribes shared keys, this would imply that the keychain is shared by zones. If a zone moves between policies, it would change to a new keychain, which may or may not be shared and which may or may not be temporary. It really does not matter much, as long as the keys in use for a zone are taken along and put on the new keychain if not present yet. This means that a key can occur on more than once keychain.
It would also be possible for a key to reside on zero keychains; in this case, it is

pregenerated and is standing by for future assignment to a keychain for a zone under a matching polocy.
The keychain appears to be a missing concept that has complicated many things in the current Enforcer. But like the keys themselves they do not occur in the XML configuration files, so these files need not change to support keychains. The change is entirely an internal affair between the Enforcer and its database.

4 Conclusions

The KASP Enforcer has been under a lot of stress, as a result of ongoing requirements from end users and the maturing process of OpenDNSSEC. The fact that it has held up under this stress deserves great respect. At the same time however, there are concerns about the ability of the OpenDNSSEC team to provide ongoing maintenance on this component as this process continues. The reason for these concerns is a that the KASP Enforcer has evolved into a state that warrants a new look at its structure and design.
This document proposes a number of changes to the KASP Enforcer, based on experiences with patching it. The proposals may be a good place to start discussing what (new) requirements should apply to this component starting OpenDNSSEC 1.3.

Notes:

info@openfortress.nl

http://openfortress.nl

OpenFortress*
digital signatures