OpenDNSSEC -- a DNS Workflow Description

The following is my (Rick van Rein's) design based on a day of discussion at NLNet Labs regarding DNS aspects and implementation concerns for the DNS-aware parts of the OpenDNSSEC signer. It is a proposed architecture, orthogonal to the KASP which manages keys for DNSSEC.

Please read this together with the diagram that I've drawn. A quick legend for the symbols used is included with the graph.

Structure: Concurrency Control

We need a few clear concepts to guide us through the incredibly versatile possibilities of concurrency. If not, we could easily drown and be left with a scheduling problem that is too broad to be effectively implemented.

Below, the following principles are proposed as guidelines to keep the concurrency model simple yet flexible:

- Atomic changes
- Queues between processes
- Transactional process steps
- Serialisable domain updates
- State-of-the-art DNS queries

These concepts help to avoid race conditions, and the absense of directed cycles (that is, cycles found when following arrows in their pointing direction) makes it possible to be sure that there will be no deadlock either.

Atomic changes: "Diff"

The idea is to transport only changes of DNS information, but to do so in atomic changes. Atomic meaning: no other process can detect an intermediate state, so the whole thing contained in a single communication package on a "Diff" channel is either to-be-done or it is done.

A "Diff" comprises of any combination of removals of a name/type combination of a resource record, followed by new record data (or nothing if there are no new contents). These "Diff" records are conceptually similar to an IXFR, although it is likely that a realistic IXFR will be split into smaller blocks, each of which would separately be treated as an RRset to be signed.

The separation of IXFR into minimalistic “Diff” records (and differencing an AXFR to obtain similarly small “Diff” records) is helpful in enabling other processes to merge in their tasks even if the work from processing an IXFR or AXFR is heavy. This helps to keep OpenDNSSEC responsive to emergency resigning, even when faced with a large zone resigning operation.

Queues between processes

A queue is a conceptually straightforward synchronisation construct. Its principal operation comes down to a first-in, first-out structure. In reality though, a queue could optimise by sending out high-priority events before normal-priority ones. This could be used to advance emergency operations like emergency key rollover.

Queues should adhere to the principle of domain serialisability, described below. Within these constraints, they could optimise to their liking.

Before a process can write into a queue, it should obtain a mutually exclusive lock on that queue. When done writing, the lock is to be released. When reading the next “Diff” record from the queue, a lock is also given on that record; after being done with it, the lock ought to be released.

This is very often a conceptual design stratagy that need not be implemented. For example, if the steps on the left and right side of Q1 are part of the same implementation process, there may not be a need to actually lock anything.

Also, an implementation might do everything before Q2 in one process (probably using `select()` to wait for I/O or alerts) then there is effectively no need to lock the writing side of Q2 at all, let alone compete.

Transactional process steps

Every rectangle in the diagram represents what could conceptually be seen as a simple consumer/producer process. It consumes a “Diff” record (or something else as annotated on the incoming arrow) from the incoming queue, and produces one or more “Diff” records or other on its output.

Every process step, that is the whole process of consuming, chewing on it and producing one or more bits of output, can be seen as a transaction, or an atomic processing step.

Conceptually, it appears as if the consumption of a “Diff” on the input and the production of the corresponding output happens instantaneously, or more accurately, time cannot be split finely enough to look at an intermediate state.

Locks are of course the way to achieve this: First acquire a lock on the input (by consuming an input record) then lock the output and produce whateven comes out, then release the locks at the *same* time. When the read lock on the input queue is lifted, the corresponding element is removed from that queue.

By always obtaining the locks in this order (if we need to implement them at all) it is possible to be certain that no deadlock will ever occur.

Once again, optimising variations would be possible, for example to enable multiple processes to sign in parallel, but it requires a more complicated queue. The queue would enable multiple processes to lock consecutive positions in the queue, and would pass on the information in an order compliant with the conceptual idea of a queue.

Serialisable domain updates

It is probably good to use one constraint to any optimisations that relate to concurrency, and that is the concurrency concept that is also used in databases. This principles states that it should always be possible to assume a particular ordering of transactions after a number of transactions have committed. It could be said that databases enfore a linear model of time, as opposed to branching time.

There are situations where it is safe to change the order of “Diff” records in a queue, possibly to the advantage of emergency updates. This would at least seem possible if they handle two different domains. It may or may not apply to hierarchically ordered domains, such as a parent and its child.

State-of-the-art DNS queries

An important requirement of several of the processing steps is that they ought to be able to query for the most recent DNS data. The concern for all the various stages of a domain being signed is far from simple.

It should suffice if each processing step can rely on the data that it has output itself -- it does not seem to be necessary to incorporate any updates performed by later processing steps.

This means that a need for DNS information can be sent along the arrows, straight through queues and processing steps until it ends in the zoneDB. That is, if no intermediate “Diff” exists that answers the need for information. Such a “Diff” would have been output by the current process, and is thus of concern. Thanks to the serialisability property, the nearest-by “Diff” is always the most recent one output by the current processing step, so it is to be used. Note that processing steps are atomic, so “Diff” data need only be looked for in queues. Probably the best is to include the read/output side of a queue even if it is locked, but never to include the write/input side of a queue if it is locked -- the latter being subject to change.

For example, if the step “re-list all RR” needs to query for all records in a domain, it would ask Q2, who could either find a “Diff” that answers the query, or pass it on to Q3, Q4 and so on until zoneDB. If the same name occurs in Q2 and zoneDB, the “Diff” in Q2 prevails. If a “Diff” exists in Q3 and Q4 covering the same DNS request, the version in Q3 is delivered back to the inquiring process.

Structure: Timing and scheduling

DNS introduces timing requirements in the form of TTL and SOA parameters. DNSSEC adds even more such constraints.

In essence, the adjoining schema does not take all these timing requirements into account. Instead, the proposal is to look upon it as a generic problem of *realtime scheduling.*

Based on this view, it ought to be possible to infer a set of queries that can be sent to the various blocks of the system to investigate how much work is still in progress, how much is lying ahead (due to timer-based events) and at what time each change should be triggered.

I expect a generic realtime scheduler to solve this problem for us, and that we will only have to supply it with timing details and the proper set of priorities. Low priorities for user-initiated changes (IXFR from the master), medium for emergency rollovers and medium to high for time-scheduled operations.

The realtime scheduler will be linked in with the processes that run, as well as the ordering of queues.

Model: NSEC(3) and ordering

DNS data for signing comprises of resource records as well as *holes* between them. The holes need explicit handling because they will be signed just as well as resource records.

The need for multiple orders

It is not likely that we will want to create NSEC as well as NSEC3 for a zone. It may happen however, that we will want to create a new NSEC3 chain, possibly due to a weakness in the hashing algorithm used.

Note that this is not likely to happen at all, since NSEC3 makes multiple passes with a single hash which is thought to be good. But if it ever happens, it is best to resolve it without domain downtime, especially because such changes would probably apply to many domains at the same time.

Efficiency of NSEC3 chain rollover is an important concern if it is indeed the case that multiple domains will want to do the same rollover at the same time.

Orders and a few useful properties

A *partial order* is a mathematical concept that is defined as a tuple:

```
<S,R>
```

where `S` defines a set of values, or in DNS terms, the resource records; and `R` defines relations between elements of `S`. The relation `R` can have a number of useful properties:

- `R` is *transitively closed* if for any `(a,b)` and `(b,c)` in `R`, their composition `(a,c)` is also part of `R`
- `R` is *complete* if for any pair of different values `a` and `b` in `S`, either `(a,b)` or `(b,a)` is part of `R`

If we use R to model the order in which NSEC or NSEC3 records are ordered, we would assume it was transitively closed, at least conceptually. As soon as R is a complete order over S, we would be able to generate all NSEC or NSEC3 records for the zone with records S.

More practically though, we could also avoid any transitive relations in R; that is, if (a,b) and (b,c) are part of R, we would avoid including (a,c) as well. This can be done if holes like (a,c) when present in R are cut up into (a,b) and (b,c) upon the introduction of b in S. Similarly, when removing an element from S, two holes in R must be composed to one hole.

In this practical way of implementing the NSEC or NSEC3 order, it would be simple to conclude when the cycle is complete, namely when the number of resource records in S is one more than the number of relations in R. This is only true if transitivity is always removed.

Complexity of creating an order

In theory, it would be possible to introduce a new order for a set of resource records by setting up a set of resource records with an empty relation. Then, when resigning the zone, all resource records would be retracted and re-inserted, an opportunity for cutting up the holes.

However.

This would lead to an order created with the insertion sort algorithm, which is of complexity O(N*N). More efficient algorithms exist, notably heapsort, which are of complexity O(N*log(N)). Quicksort is not as good as heapsort, since it will not always achieve O(N*log(N)).

Since N is the size of the domain for which a new NSEC or NSEC3 cycle would be generated, the choice of algorithm is a big concern for large zones, notably TLDs. This means that it is probably best to create an order in one operation covering all records found in a domain. Later, when records are individually removed or added as a result of "Diff" records, small changes to the NSEC or NSEC3 order can be made by way of cutting-up and joining holes as described above.

Description: Diagram blocks

Following is a description for each block in the diagram.

DNS from master

This is the interaction with the master that feeds OpenDNSSEC. It is most likely based on NOTIFY and IXFR/AXFR interactions.

Check SOA

For incoming IXFR, it is good to know that the order in which data comes in is correct. If not, an AXFR should be requested instead and treated by the "diff" process.

A single IXFR may be split into multiple "Diff" records, each covering an RRset. This means that the "Diff" records are nice and small.

serial/dom

Stores the latest accepted SOA serial number per domain. This may come from either IXFR or AXFR interactions with the master.

Note that SOA serial numbers need not be passed on in the rest of the OpenDNSSEC flow of "Diff" events. The last stages will add a serial number to match the version of the domain.

The reason that incoming and outgoing serial numbers are not coupled is that, between two consecutive incoming numbers, OpenDNSSEC may have to insert a version for its own reasons, for reasons like a key rollover.

diff

This processing step groups an AXFR into the RRsets and then asks what the state-of-the-art DNS data for each is. This consults Q1, Q2, Q3, Q4 and zoneDB, in that order. The first match is reported back to the "diff" process, to learn if it warrants sending a "Diff" record with the change or not.

strip/okay

This processing step could remove record types that could confuse the following processes. SOA, NSEC3 and NSEC3PARAM are all strictly under the control of the logic that follows, and should therefore be removed.

Aside from plain records, it is worth noting that several DNSSEC-related records are welcome to pass through this processing step. These include DS, DLV and DNSKEY -- the latter because a DNSKEY may have to be prepublished before a transition to another hoster (and another OpenDNSSEC machine).

KASP's DNSKEY rollover timer

KASP will trigger a Timeout for a DNSKEY under a zone, and trigger its replacement.

It is a matter of taste if the DNSKEY durability knowledge is located in the KASP, or that the KASP is only the source of Timeouts at given moments. Since a HSM often contains a secure timer, it makes sense to assign the responsibility of timekeeping to the KASP.

Note however that simple USB tokens do not support secure timing, and signed NTP may have to be relied upon. Since NTP is not usually signed, this may involve setting up one's own timing source as part of such a KASP.

key emerg

When a key is compromised, an emergency process is started to replace a DNSKEY under a zone.

The “ZSK emerg” described below also triggers “key emerg”.

DNSKEY RR changes

This creates “Diff” records that introduce and/or remove DNSKEY records. Note that a “Diff” record is an atomic change, so it may be possible to substitute records immediately.

Most likely though, the KASP timer will have to signal different kinds of timeouts, namely those for introduction and those for removal of DNSKEY records.

This process also craetes a “Diff” record for a SOA to indicate a wish to publish a zone’s intermediate state. To get this, it simply requests the state-of-the-art SOA record and creates a “Diff” that both removes and creates that same SOA. Serial numbers are a matter of later concern.

KASP pubkeys

This database contains the public keys in use by the KASP. Since the KASP is likely to create public keys (from its internally generated private keys) it is probably a database that is being queried for the keys in use. In addition to that, public keys (and their complementary private keys) can be removed by the “DNSKEY RR changes” process.

KASP’s NSEC3 rollover timer

Assuming that we will want to replace NSEC3PARAM records, and thus create a new NSEC3 chain, a process similar to DNSKEY rollover can be initiated. The “Diff” submitted will also proceed in two phases; first, a new NSEC3PARAM will be added to the state-of-the-art NSEC3PARAM, and second, the older NSEC3PARAM will be removed.

NSEC3 emerg

This covers the extremely unlikely event where an NSEC3 hashing algorithm would cause an emergency.

NSEC3PARAM generator

This processing step creates new NSEC3PARAM and removes old ones after their useful life. Note that NSEC3 and NSEC chains are created further down the workflow.

This process also craetes a “Diff” record for a SOA to indicate a wish to publish a zone’s intermediate state. To get this, it simply requests the state-of-the-art SOA record and creates a “Diff” that both removes and creates that same SOA. Serial numbers are a matter of later concern.

AND

This block causes an event after an “NSEC3 emerg” process has led to a corresponding “Diff”.

KASP’s zone resign timer

This timer fires when it is time to re-sign a zone. As with key rollover, it is still to be decided if the KASP knows when a zone must be signed freshly, or if the KASP only handles abstract timing requests as required by the process steps in this workflow.

ZSK emerg

In case of an emergency related to the zone signing key, the zone must be freshly signed. Note that “key emerg” is also triggered and must be finished before “ZSK emerg” is useful.

re-list all RR in small, atomic RRsets

This processing step walks through the state-of-the-art DNS records and for each RRset it finds it constructs a “Diff” which retracts it and, in the same atomic “Diff”, reconstructs it. This is intended to trigger the signing processes downstream.

This process also craetes a “Diff” record for a SOA to indicate a wish to publish a zone’s intermediate state. This may be prudent to do after partial updates, so that other processes can intervene and the domain updates are split in separate IXFR records. To get this, it requests the state-of-the-art SOA record and creates a “Diff” that both removes and creates that same SOA. Serial numbers are a matter of later concern.

name*order storage

This database stores the `(S,R)` relations of DNS names and the corresponding order or orders that are currently in use. If multiple NSEC/NSEC3 chains are being built or used, multiple name*order relations are stored for a single zone.

add NSEC(3)

This processing step responds in a special way to NSEC3PARAM changes, and in a general way to all others.

If an NSEC3PARAM is removed, the corresponding “name*order” is removed from storage, and by way of newly generated “Diff” records, from the signed zone data.

If a new NSEC3PARAM is supplied, a new order is created in the storage for “name*order”. As part of the same transaction, all NSEC3 records are supplied as “Diff” records on the process’s output.

In general, when an RRset is changed by a “Diff” on the input (and that includes NSEC3PARAM) the “name*order” storage and the NSEC and/or NSEC3 records are created to match:

- If a record name is removed, the holes surrounding it are recombined.
- If a record name is inserted, the hole in which it falls is cut in two.
- If a new record type is added to or removed from a name that exists beforehand and afterwards, the set of RRtypes in any NSEC3 record is altered and a new NSEC3 record created.

Note: Wildcards should also be created in this step.

KASP's privkey ops

This is an API to the part of the KASP that accesses private key operations. This is especially used to sign RRsets.

It remains to be determined where knowledge about the private keys to use is stored; this may be part of the KASP or part of the surrounding DNS workflow.

RRset signer

This process step consumes a "Diff" record and signs all the changes contained in it. This usually includes NSEC3 records as well as plain records.

This step also creates a new SOA serial number for the newly signed zone. It does this whenever it comes accross a SOA record, which it sees as an indication that the zone should be published.

The simplicity of the RRset signer mainly stems from the preparation by the previously added NSEC(3) records and by the separation of "Diff" messages into atomic RRset changes.

The relative indpendency of the RRsets to be signed means that an optimisation for parallel signing could be setup. Note that the best way of doing this is probably to delegate such concurrency to the KASP's privkey ops, that is, the signer engine.

serial/dom

This is a database similar to its same-named compadre in the input processing stage. It stores the SOA serial values added to the output records.

zoneDB

This database stores the completely signed records for a zone. This database supports a number of operations, all of which are transactional:

- Collecting a new IXFR as a sequence of RRsets
- Inserting an additional "Diff" for a zone as an IXFR
- Retrieving a zone's IXFR and AXFR data
- Replacing a zone's IXFR and AXFR data with less IXFR and updated AXFR data

IXFR to AXFR smasher

This process avoids an evergrowing “zoneDB”. It can employ a number of principles, possibly in combination, to achieve this effect:

- When there is more IXFR data than AXFR data for a zone, any number of the oldest IXFRs can be combined with the AXFR
- When an IXFR is past its SOA-indicated lifetime, it can be combined with the AXFR
- When all slaves have retrieved the oldest IXFR, it can be combined with the AXFR to save storage

zone publish

This I/O operation sends NOTIFY and responds to AXFR and IXFR requests from a zone’s slaves.

Scenarios

This section shows a number of scenarios or use cases rolling through the workflow. It ends by detailing the interactions between the DNS workflow module and the KASP module.

TODO