2011/0x01-v2

# Authenticated Denial of Existence in the DNS

Miek Gieben, miek.gieben@sidn.nl, SIDN
Matthijs Mekking, matthijs@nlnetlabs.nl, NLnet Labs

January 2012

## Abstract

*Authenticated denial of existence* allows a resolver to validate that a certain domain name does not exist. It is also used to signal that a domain name exists, but does not have the specific RR type you were asking for. This document attempts to answer two simple questions.

> *When returning a negative DNSSEC response, a name server sometimes includes up to two NSEC records. With NSEC3 the maximum amount is three.*
>
> - *Why do you need up to two NSEC records?*
> - *And why does NSEC3 sometimes require an extra record?*

The answer to the questions hinges on the concept of wildcards and the *closest encloser*. With NSEC, the name that is the *closest encloser* is implicitly given in the record that also denies the existence of the domain name. With NSEC3, due to its hashing, this information has to be given explicitly to a resolver. It needs one record to tell the resolver the closest encloser and then another to deny the existence of the domain name. Both NSEC and NSEC3 may need yet another record to deny or assert a wildcard presence. This results in a maximum of two NSEC and three NSEC3 records, respectively.



### Introduction

DNSSEC can be somewhat of a complicated matter, and there are certain areas of the specification that are more difficult to comprehend than others. One such area is "authenticated denial of existence".

Authenticated denial of existence allows a DNSSEC enabled resolver to validate that a certain domain name does not exist. It is also used to signal that a domain name exists, but does not have the specific RR type you were asking for.

The first is referred to as an NXDOMAIN[1] (non-existent domain) and the latter an NODATA[1] response.

In this document we will explain how authenticated denial of existence works. We begin by explaining the current technique in the DNS and work our way up to DNSSEC. We explain the first steps taken in DNSSEC and describe how NXT, NSEC and NSEC3 work. NO, NSEC2 and DNSNR also briefly make their appearance, as they have paved the way for NSEC3.

To complete the picture we also need to explain DNS wildcards as it complicates matters.

Note: In this document domain names in zone file examples will have a trailing dot, in the running text they will not. This text is written for people who have a fair understanding of DNSSEC. This document currently does not explain NSEC3 opt-out and secure delegations. This may be added in a future revision.

The following RFCs are not required reading, but they might help in understanding the problem space.

- RFC 5155[7] - Hashed Authenticated Denial of Existence;
- RFC 4592[8] - The Role of Wildcards in the DNS.

And these provide some general DNSSEC information.

- RFC 4033,RFC 4034, RFC 4035[2, 4, 3] - DNSSEC Spec;
- RFC 4956[5] - DNS Security (DNSSEC) Opt-In. This RFC has the status experimental, but is a good read.

And these three drafts give some background information on the NSEC3 development.

- tools.ietf.org/html/draft-ietf-dnsext-not-existing-rr–01;
- tools.ietf.org/html/draft-laurie-dnsext-nsec2v2–00;
- tools.ietf.org/html/draft-arends-dnsnr–00.

**Denial of Existence**

We start with the basics and take a look at NXDOMAIN handling in the DNS. To make it more visible we are going to use a small DNS zone, with 3 names (`example.org`, `a.example.org` and `d.example.org`) and 3 types (SOA, A and TXT). For brevity the class is not shown (defaults to IN), the NS records are left out and the SOA and RRSIG records are shortened. Resulting in the following unsigned zone file:

```
example.org.        SOA ( ... )
a.example.org.      A 127.0.0.1
                    TXT "a record"
d.example.org.      A 127.0.0.1
                    TXT "d record"
```

**NXDOMAIN**

If a resolver asks for the TXT type belonging to `a.example.org` to the name server serving this zone, it sends the following question: `a.example.org TXT`

The name server looks in its zone data and generates an answer. In this case a positive one: "Yes it exists and this is the data", resulting in this reply:

```
;; status: NOERROR, id: 28203

;; ANSWER SECTION:
a.example.org.      TXT "a record"

;; AUTHORITY SECTION:
example.org.        NS a.example.org.
```

The `status: NOERROR` signals that everything is OK, `id` is an integer used to match questions and answers. In the ANSWER section we find our answer. The AUTHORITY section holds information of the name servers that have information concerning the `example.org` domain.

If a resolver now asks for `b.example.org TXT` it gets an answer that this name does not exist:

```
;; status: NXDOMAIN, id: 7042

;; AUTHORITY SECTION:
example.org.        SOA ( ... )
```

In this case we do not get an ANSWER section and the status is set to NXDOMAIN. From this the resolver concludes `b.example.org` does not exist.

**NODATA**

It is important to realize, that NXDOMAIN is not the only type of *does-not-exist*. A name may exist, but the type you are asking for may not. This occurrence of non-existence is called an NODATA[1] response. Let us ask our name server for `a.example.org AAAA`, and look at the answer:

```
;; status: NOERROR, id: 7944

;; AUTHORITY SECTION:
example.org.        SOA ( ... )
```

The status is NOERROR meaning that the `a.example.org` name exists. But the reply does *not* contain an ANSWER section. Instead it has an AUTHORITY section which holds the SOA record of `example.org`. The resolver has to

put these pieces of information together and conclude that `a.example.org` exists, but it does not have an `AAAA` record.

### Secure Denial of Existence

The above has to be translated to the security aware world of DNSSEC. But there are a few requirements DNSSEC brings to the table:

1. There is no online signing defined in DNSSEC. Although a name server is free to compute the answer and signature(s) on-the-fly, the protocol is written with a "first sign", "then load" attitude in mind [1].
2. The DNS packet header is not signed. This means that a `status: NXDOMAIN` can not be trusted. In fact the entire header may be forged, including the AD [2] bit, which may give some food for thought;
3. DNS wildcards complicate matters significantly. More about this in later sections.

The first requirement implies that all denial of existence answers need to be pre-computed, but it is impossible to pre-compute (all conceivable) non-existence answers. In the example above, you need a way to tell somebody who is asking for `b.example.org` that it does not exists *without* using the name `b.example.org` in the answer. This has been solved by introducing a record that defines an interval between two existing names. Or to put it another way: it defines the holes (non-existing names) in the zone. This record *can* be signed beforehand and given to the resolver.

> *Given all these troubles why didn't the designers of DNSSEC go for the (easy) route and allowed for online signing? Well, at the time (pre 2000), online signing was not feasible with the current hardware. Keep in mind that the larger servers get between 2000 and 6000 queries per second (qps), with peaks up to 20,000 qps or more. Scaling signature generation to these kind of levels is always a challenge. Another issue was (and is) key management, for online signing to work you need access to the private key(s). This is considered a security risk.*

[1] It is rather asymmetrical, but a lot of the design in DNSSEC stems from fact that you need to accommodate authenticated denial of existence. If the DNS didn't have NXDOMAIN, DNSSEC would be a lot simpler, but a lot less useful!

[2] AD stands for Authenticated Data, see RFC 3655[12].

The road to the current solution (NSEC/NSEC3) was long. It started with the NXT (next) record. The NO (not existing) record was introduced, but never made it to RFC. Later NXT was superseded by NSEC (next secure) record. From there it went through NSEC2/DNSNR to finally reach NSEC3 (next secure, version 3) in RFC 5155.

### NXT

The first attempt to specify authenticated denial of existence was NXT (RFC 2535[6]). Section 5.1 of that RFC introduces the record:

> *The NXT resource record is used to securely indicate that RRs with an owner name in a certain name interval do not exist in a zone and to indicate what RR types are present for an existing name.*

By specifying what you *do* have, you implicitly tell what you *don't* have. NXT is superseded by NSEC. In the next section we explain how NSEC (and thus NXT) works.

### NSEC

In RFC 3755[11] all the DNSSEC types were given new names, SIG was renamed RRSIG, KEY became DNSKEY and NXT was simply renamed to NSEC and a few, minor issues were fixed in the process.

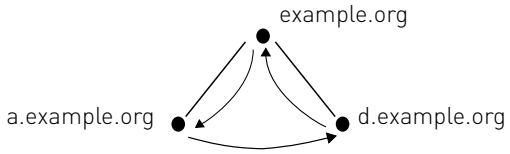
Just as NXT, NSEC is used to describe an interval between names: it indirectly tells a resolver which names *do not* exist in a zone.

For this to work, we need our `example.org` zone to be sorted in canonical ordering (see RFC 4034, Section 6.1), and then create the NSECs. We add three NSEC records, one for each name, and each one "covers" a certain interval. The last NSEC record points back to the first as required by the RFC. Also see figure 1.

1. The first NSEC covers the interval between `example.org` and `a.example.org`;
2. The second NSEC covers: `a.example.org` to `d.example.org`;
3. The third NSEC points back to `example.org`, and covers `d.example.org` to `example.org` (i.e. the end of the zone).

As we have defined the intervals and put those in resource records, we now have something that can be signed.

Figure 1: The NSEC records of `example.org`. The arrows represent NSEC records, starting from the apex.




This signed zone is loaded into the name server. It looks like this:

```
example.org.        SOA ( ... )
                    DNSKEY ( ... )
                NSEC a.example.org. SOA NSEC DNSKEY RRSIG
                    RRSIG(SOA) ( ... )
                    RRSIG(DNSKEY) ( ... )
                    RRSIG(NSEC) ( ... )
a.example.org.      A 127.0.0.1
                    TXT "a record"
                   NSEC d.example.org. A TXT NSEC RRSIG
                    RRSIG(A) ( ... )
                    RRSIG(TXT) ( ... )
                    RRSIG(NSEC) ( ... )
d.example.org.      A 127.0.0.1
                    TXT "d record"
                    NSEC example.org. A TXT NSEC RRSIG
                    RRSIG(A) ( ... )
                    RRSIG(TXT) ( ... )
                    RRSIG(NSEC) ( ... )
```

If a DNSSEC aware resolver asks for `b.example.org`, it gets back a `status: NXDOMAIN` packet, which by itself is meaningless as the header can be forged. To be able to securely detect that `b` does not exist, there must also be an NSEC record which *covers* the name space where `b` lives:

```
a.example.org.      NSEC d.example.org.
```

does just do that, `b` should come after `a`, but the next owner name is `d.example.org`, so `b` does not exist.

Only by making that calculation, can a resolver conclude that the name `b` does not exist. If the signature of the NSEC record is valid, `b` is proven not to exist. We have: *authenticated denial of existence*.

**NODATA Responses**

NSEC records are also used in NODATA responses. In that case we need to look more closely at the type bit map. The type bit map in an NSEC record tells which types are defined for a name. If we look at the NSEC record of `a.example.org` (see the reply below for an example of the record) we see the following types in the bit map: A, TXT, NSEC and RRSIG. So for the name `a` this indicates we must have an A, TXT, NSEC and RRSIG record in the zone.

With the type bit map of the NSEC record a resolver can establish that a name is there, but the type is not. A resolver asks for `a.example.org AAAA`. This is the reply that comes back:

```
;; status: NOERROR, id: 44638

;; AUTHORITY SECTION:
example.org.        SOA ( ... )
example.org.        RRSIG(SOA) ( ... )
a.example.org.     NSEC d.example.org. A TXT NSEC RRSIG
a.example.org.      RRSIG(NSEC) ( ... )
```

Now the resolver should check the AUTHORITY section and conclude that:

1. `a.example.org` exists (because of the NSEC with that owner name) and;
2. that the type (AAAA) does not as it is *not* listed in the type bit map.

By understanding NSEC records, you have mastered the basics of authenticated denial of existence.

But there were two issues with NSEC (and NXT). The first is that it allows for zone walking. NSEC records point from one name to another, in our example: `example.org`, points to `a.example.org` which points to `d.example.org` which points back to `example.org`. So we can reconstruct the entire `example.org` zone even when zone transfers (AXFR) on the server are denied.

The second issue is that when a large, delegation heavy, zone deploys DNSSEC, every name in the zone gets an NSEC plus RRSIG. This leads to a huge increase in the zone size (when signed). This would in turn mean that operators of large zones (or with a lot of zones) who are deploying DNSSEC, face up front costs. This could hinder DNSSEC adoption.

These two issues eventually lead to NSEC3 which:

- Adds a way to garble the next owner name, thus thwarting zone-walking;
- Makes it possible to skip names for the next owner name. This feature is called opt-out. It means not all names in your zone get an NSEC3 plus ditto signature, making it possible to "grow into" your DNSSEC deployment.

> Describing opt-out is (currently) out of scope for this document. For those interested, opt-out is explained in RFC 4956[5], which is curiously titled "(DNSSEC) Opt-In". Later this is incorporated into RFC 5155.

But before we delve in to NSEC3 lets first take a look at its predecessors, NO, NSEC2 and DNSNR.

#### NO, NSEC2 and DNSNR

The NO record was the first to introduce the idea of hashed owner names. It also fixed other shortcomings of the NXT record. At the time (around 2000) zone walking was not considered important enough to warrant the new record. People were also worried that deployment would be hindered by developing an alternate means of denial of existence. Thus the effort was shelved and NXT remained. When the new DNSSEC specification was written, NSEC saw the light and inherited the two issues from NXT.

Several years after that NSEC2 was introduced as a way to solve the two issues of NSEC. The NSEC2 draft contains the following paragraph:

> *This document proposes an alternate scheme which hides owner names while permitting authenticated denial of existence of non-existent names. The scheme uses two new RR types: NSEC2 and EXIST.*

When an authenticated denial of existence scheme starts to talk about EXIST records it is worth paying extra attention.

NSEC2 solved the zone walking issue, by hashing (with SHA1 and a salt) the "next owner name" in the record, thereby making it useless for zone walking.

But it did not have opt-out. Although promising, the proposal didn't make it because of issues with wildcards and the odd EXISTS resource record.

The DNSNR RR was another attempt that used hashed names to foil zone walking and it also introduced the concept of opting out (called "Authoritative Only Flag") which limited the use of DNSNR in delegation heavy zones. This proposal didn't make it either, but it provided valuable insights into the problem.

#### NSEC3

From the experience gained with NSEC2 and DNSNR, NSEC3 was forged. It incorporates both opt-out and the hashing of names. NSEC3 solves any issues people might have with NSEC, but it introduces some additional complexity.

NSEC3 did not supersede NSEC, they are both defined for DNSSEC. So DNSSEC is blessed with two different means to perform authenticated denial of existence: NSEC and NSEC3. In NSEC3 every name is hashed, including the owner name.

SHA1 is always used for the hashing. To make it even more difficult to retrieve the original names, the hashing can be repeated several times each time taking the previous hash as input. To thwart rainbow table attacks, a custom salt is also added. In the NSEC3 for `example.org` we have hashed the names twice and use the salt `DEAD`. Lets look at typical NSEC3 record:

```
15BG9L6359F5CH23E34DDUA6N1RIHL9H.example.org. (
   NSEC3 1 0 2 DEAD 04SKNAPCA5AL7QOS3KM2L9TL3P5OKQ4C
        SOA RRSIG DNSKEY NSEC3PARAM )
```

On the first line we see the hashed owner name: `15BG9L6359F5CH23E34DDUA6N1RIHL9H.example.org`, this is the hashed name of `example.org`. Note that even though we hashed `example.org`, the zone's name is added to make it look like a domain name again. So un-hashed it sort of looks like: `SHA1(example.org).example.org`.

The next owner name `a.example.org` (line 2) is hashed to: `04SKNAPCA5AL7QOS3KM2L9TL3P5OKQ4C`. Note that `.example.org` is not added to the next owner name, as this name always falls in the current zone.

The "1 0 2 DEAD" section of the NSEC3 states:

- Hash Algorithm = 1 (SHA1, this is the default, no other hash algorithms are defined for use in NSEC3);
- Opt Out = 0 (disabled);
- Hash Iterations = 2;
- Salt = "DEAD".

At the end we see the type bit map, which is identical to NSEC's bit map, that lists the types present at the original owner name. Note that the type NSEC3 is absent from the list in the example above. This is due to the fact that the original owner name (`example.org`) does not have the NSEC3 type. It only exists for the hashed name.

Names like `1.h.example.org` hash to one label in NSEC3, `1.h.example.org` becomes: `117GERCPRCJGG8J04EV1NDRK8D1JT14K.example.org` when used as a owner name. This is an important observation. By hashing the names you loose the depth of a zone - hashing introduces a flat space of names. As opposed to NSEC.

In fact the domain name used above: `1.h.example.org` creates an empty non-terminal. Empty non-terminals are domain names that exist but have no RR types associated with them.

```
1.h.example.org.    TXT "1.h record"
```

Creates 2 names:

1. `1.h.example.org` that has the type: TXT;
2. `h.example.org` which has no types. This is the empty non-terminal.[3]

**Slaving an NSEC3 Zone**

A secondary server slaving a zone with NSEC3 records need to find out the specifics (hash iterations and salt) to be able to hash incoming query names.

To do this it could scan the zone during the AXFR for NSEC3 records and glance the NSEC3 parameters from them. However, it would need to make sure that there is at least one complete set of NSEC3 records for the zone using the same parameters. Therefore, it would need to inspect all NSEC3 records.

A more graceful solution was designed. This solution was to create a new record, NSEC3PARAM, which must be placed at the apex of the zone. Its sole role is to provide a single, fixed place where a secondary name server can directly see the NSEC3 parameters used. If NSEC3 were designed in the early days of DNS (+/- 1985) this information was probably put in the SOA record.

**Wildcards in the DNS**

In the above sections we haven't revealed the entire story. There is a complication: wildcards. Wildcards have been part of the DNS since the first DNS RFCs. They allow to define *all* names for a certain type in one go. In our `example.org` zone we could for instance add a wildcard record:

```
*.example.org.      TXT "wildcard record"
```

For completeness our (unsigned) zone now looks like this:

```
example.org.        SOA ( ... )
*.example.org.      TXT "wildcard record"
a.example.org.      A 127.0.0.1
                    TXT "a record"
d.example.org.      A 127.0.0.1
                    TXT "d record"
```

[3] An empty non-terminal *will* get an NSEC3 records, but not an NSEC record.

If a resolver asks for `z.example.org TXT`, the name server will respond with an *expanded wildcard*, instead of an NXDOMAIN:

```
;; status: NOERROR, id: 13658

;; ANSWER SECTION:
z.example.org.      TXT "wildcard record"
```

Note however that the resolver can not detect that this answer came from a wildcard. It just sees the answer as-is. How will this answer look with DNSSEC?

```
;; status: NOERROR, id: 51790

;; ANSWER SECTION:
z.example.org.      TXT "wildcard record"
z.example.org.      RRSIG(TXT) ( ... )

;; AUTHORITY SECTION:
*.example.org.      NSEC example.org. TXT RRSIG NSEC
*.example.org.      RRSIG(NSEC) ( ... )
```

The RRSIG of the `z.example.org` TXT record[4] indicates there is a wildcard configured. The RDATA of the signature lists a label count [4, Section 3.1.3.] of two (not visible in the answer above), but the owner name of the signature has three three labels. This mismatch indicates there is a wildcard configured for the name `*.example.org`. Another hint is that the owner name of the NSEC is `*.example.org`. This NSEC proves that the queried name `z.example.org` does not exist, and wildcard name expansion was indeed allowed.

One thing you may notice is that this reply has an NSEC record in it *even* though it is not an NXDOMAIN nor NODATA reply. In this case it is there to tell the resolver this answer was synthesized from a wildcard.

In the reply above we see that `z.example.org` was generated via wildcard expansion. The DNSSEC standard mandates that an NSEC (or NSEC3) is included in such responses. If it didn't, an attacker could poison the cache with false data.

Suppose that the resolver would have asked for `a.example.org TXT`, an attacker could modify the packet in such way that it looks like the response was generated through wildcard expansion, even though there exists a record for `a.example.org TXT`: `a.example.org. TXT "a record"`

[4] An astute reader may notice that it appears as if a `z.example.org` RRSIG(TXT) is created out of thin air. This is not the case. The signature for `z.example.org` does not exist. The signature you are seeing is the one for `*.example.org` which does exist, only the owner name is switched to `z.example.org`. So even with wildcards, no signatures are created on the fly.

The tweaking simply consists of adjusting the ANSWER section to:

```
;; status: NOERROR, id: 31827

;; ANSWER SECTION
a.example.org.      TXT "wildcard record"
a.example.org.      RRSIG(TXT) ( ... )
```

Which would be a perfectly valid answer if we would not require the inclusion of an NSEC or NSEC3 record in the wildcard answer response. The resolver believes that `a.example.org TXT` is a wildcard record, and the real record is obscured. This is bad and defeats all the security DNSSEC can deliver. Because of this, the NSEC or NSEC3 should be present.

Thus a resolver can detect such a spoofing attempt:

1. If the NSEC(3) is not present, assume the answer is spoofed;
2. If the NSEC(3) is there, check it. If the signature is not correct, assume a spoofed answer.

Another way of putting this is that DNSSEC is there to ensure the name server has followed the steps as outlined in RFC1034[9] Section 4.3.2 for looking up names in the zone. It explicitly lists wildcard lookup as one of these steps (3c), so with DNSSEC this must be communicated to the resolver: hence the NSEC(3) record.

With NSEC the maximum number of NSEC records a resolver can get back is two: one for the denial of existence and another for the wildcard. We say maximum, because sometimes a single NSEC can prove both. With NSEC3 it is three, as to why, we will explain in the next section.

#### Returning Three NSEC3s

With NSEC3 matters are even more complicated. So we have an NSEC3 that denies the existence of the requested name and an NSEC3 that denies wildcard synthesis. What do we miss?

The short answer is that due to the hashing in NSEC3 you loose the depth of your zone: everything is hashed into a flat plain. To make up for this loss of information you need an extra record. The more detailed explanation is quite a bit longer…

To understand NSEC3 we will need two definitions:

**Closest encloser:** Introduced in RFC4592[8], this is the first *existing* name (this may be an empty non-terminal) in the zone that is an ancestor of the name used in the query. Suppose the query name is `x.2.example.org` then `example.org` is the *closest encloser* in our example;

**Next closer name:** Introduced in the NSEC3 RFC, this is the closest encloser with one more label added to the left. So if `example.org` is the closest encloser for the query name `x.2.example.org`, `2.example.org` is the *next closer name*.

An NSEC3 *closest encloser proof* consists of:

1. An NSEC3 RR that **matches** the *closest encloser*. This means the un-hashed owner name of the record *is* the closest encloser. This bit of information tells a resolver: "The name you are asking for does not exist, the closest I have is this".
2. An NSEC3 RR that **covers** the *next closer name*. This means it defines an interval in which the *next closer name* falls. This tells the resolver: "The name in your question falls in this interval, and therefor the name in your question does not exist. In fact, the closest encloser is indeed the closest I have".

Take the following example. We take our zone, but now with the following two records and it is signed with NSEC3. [5]

```
1.h.example.org.    TXT "1.h record"
3.3.example.org.    TXT "3.3 record"
```

The complete unsigned zone now looks like this.

```
example.org.        SOA ( ... )
1.h.example.org.    TXT "1.h record"
3.3.example.org.    TXT "3.3 record"
```

The resolver asks the following: `x.2.example.org TXT`. This leads to an NXDOMAIN response from the server, which contains three NSEC3 records.[6] Also see figure 2, the numbers in the figure correspond with the following NSEC3 records:

```
15BG9L6359F5CH23E34DDUA6N1RIHL9H.example.org. (
 NSEC3 1 0 2 DEAD 1AVVQN74SG75UKFVF25DGCETHGQ638EK SOA
        RRSIG DNSKEY NSEC3PARAM )

75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ.example.org. (
 NSEC3 1 0 2 DEAD 8555T7QEGAU7PJTKSNBCHG4TD2M0JNPJ TXT
```

[5] As said these records create two non-terminals: "h.example.org" and "3.example.org", but that is irrelevant for the theory here.

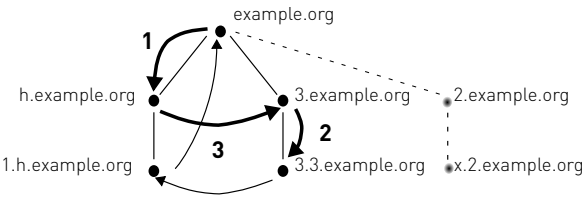
[6] A list of hashed owner names can be found in the section "List of Hashed Owner Names" on page 9.

```
        RRSIG )

1AVVQN74SG75UKFVF25DGCETHGQ638EK.example.org. (
    NSEC3 1 0 2 DEAD 75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ )
```

If we would follow the NSEC approach, the resolver is only interested in one thing. Does the hash of `x.2.example.org` fall in any of the intervals of the NSEC3 records it got?

Figure 2: `x.2.example.org` does not exist. The arrows represent the NSEC3 records, the thicker ones are the NSEC3s returned in our answer.




The hash of `x.2.example.org` is `NDTU6DSTE50PR4A1F2QVR1V31G00I2I1`. Checking this hash on the first NSEC3 yields that it does not fall in between the interval: `15BG9L6359F5CH23E34DDUA6N1RIHL9H` and `1AVVQN74SG75UKFVF25DGCETHGQ638EK`. For the second NSEC3 the answer is also negative: the hash sorts outside the interval described by `75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ` and `8555T7QEGAU7PJTKSNBCHG4TD2M0JNPJ`. And the last NSEC3 also isn't of any help. What is a resolver to do? It has been given the maximum amount of NSEC3s and they all seem useless.

A question that you might have at this point is why doesn't the server send an NSEC3 that covers the hash of `x.2.example.org`, so the resolver can validate in one step? While this indeed denies the existence of `x.2.example.org` it is only half the answer. As explained, a denial of existence answer *needs* to say something about whether or not a wildcard should have been expanded. And to communicate *which* wildcard that could have been, you need to tell the resolver what the closest encloser is.

So this is where the closest encloser proof comes into play. And for the proof to work, the resolver needs to know what the *closest encloser* is. There must be an existing ancestor in the zone: a name must exist that is shorter than the query name. The resolvers keeps hashing, increasingly shorter names from the query name until an owner name of an NSEC3 *matches*. This owner name is the *closest encloser*.

When the resolver has found the closest encloser, the next step is to construct the *next closer name*. This is the closest encloser with the last chopped label from query name prepended to it: "<last chopped label>.<closest encloser>". The hash of this name should be covered by the interval set in any of the other NSEC3 records.

Then the resolver needs to check the presence of a wildcard. It creates the wildcard name by prepending the wildcard label to the closest encloser: "*.<closest encloser>", and use the hash of that.

Going back to our example the resolver must first detect the NSEC3 that matches the closest encloser. It does this by chopping up the query name, hashing each instance (with the same number of iterations and hash as the zone it is querying) and comparing that to the answers given. So it has the following hashes to work with:

`x.2.example.org`: `NDTU6DSTE50PR4A1F2QVR1V31G00I2I1`, last chopped label: "<empty>";

`2.example.org`: `7T70DRG4EKC28V93Q7GNBLEOPA7VLP6Q`, last chopped label: "x";

`example.org`: `15BG9L6359F5CH23E34DDUA6N1RIHL9H`, last chopped label: "2";

Of these hashes only one matches the owner name of one of the NSEC3 records: `15BG9L6359F5CH23E34DDUA6N1RIHL9H`. This must be the closest encloser (un-hashed: `example.org`). That's the main purpose of that NSEC3 record: tell the resolver what the closest encloser is.

From that knowledge the resolver constructs the next closer, which in this case is: `2.example.org`; `2` is the last label chopped, when `example.org` is the closest encloser. The hash of *this* name should be covered in any of the other NSEC3s. And it is, `7T70DRG4EKC28V93Q7GNBLEOPA7VLP6Q` falls in the interval set by: `75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ` and `8555T7QEGAU7PJTKSNBCHG4TD2M0JNPJ` (this is our second NSEC3).

So what does the resolver learn from this?

- `example.org` exists;
- `2.example.org` does not exist.

And if `2.example.org` does not exist, `x.2.example.org` also does not exist. But *only* if there was *no* wildcard configured. So this is the last step: check if there is a wildcard configured at the closest encloser.

The resolver hashes `*.example.org` to `22670TRPLHSR72PQQMEDLTG1KDQEOLB7`. Only the last NSEC3 covers this hash. The hash falls in the interval set by `1AVVQN74SG75UKFVF25DGCETHGQ638EK` and `75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ` (this is our third NSEC3). This means there is no wildcard at the closest encloser and `x.2.example.org` definitely does not exist.

When we have validated the signatures, we reached our goal: authenticated denial of existence.

Coming back to the original question: why do we need (up to) three NSEC3 records? The resolver needs to be explicitly told what the *closest encloser* is and this takes up a full NSEC3 record. Then the next closer name needs to be covered in an NSEC3 record, and finally an NSEC3 must say something about the wildcard. That makes three records.

### List of Hashed Owner Names

The following owner names are used in this document. The origin for these names is `example.org`.

| Original Name | Hashed Name |
|---|---|
| `@` | `15BG9L6359F5CH23E34DDUA6N1RIHL9H` |
| `*` | `22670TRPLHSR72PQQMEDLTG1KDQEOLB7` |
| `1.h` | `117GERCPRCJGG8J04EV1NDRK8D1JT14K` |
| `2` | `7T70DRG4EKC28V93Q7GNBLEOPA7VLP6Q` |
| `3.3` | `8555T7QEGAU7PJTKSNBCHG4TD2M0JNPJ` |
| `3` | `75B9ID679QQOV6LDFHD8OCSHSSSB6JVQ` |
| `a` | `04SKNAPCA5AL7QOS3KM2L9TL3P5OKQ4C` |
| `b` | `IUU8L5LMT76JELTP0BIR3TMG4U3UU8E7` |
| `h` | `1AVVQN74SG75UKFVF25DGCETHGQ638EK` |
| `x.2` | `NDTU6DSTE50PR4A1F2QVR1V31G00I2I1` |

Table 1: Hashed owner names for the `example.org` zone.


### Acknowledgements

This document would not be possible without the help of Ed Lewis Roy Arends, Wouter Wijngaards, Olaf Kolkman, Carsten Strotmann, Jan-Piet Mens, Marco Davids, Esther Makaay and Antoin Verschuren. Also valuable, was the source code of Unbound (`validator/val_nsec3.c`).

For version two extensive feedback was received from Karst Koymans.


### Changes in version two

- Small corrections;
- Mention the NO resource record;
- The NSEC3 example now returns three NSEC3 records;
- Empty non-terminals are explained.

### References


[1] M. Andrews. Negative Caching of DNS Queries (DNS NCACHE). RFC 2308 (Proposed Standard), March 1998. Updated by RFCs 4035, 4033, 4034.

[2] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS Security Introduction and Requirements. RFC 4033 (Proposed Standard), March 2005. Updated by RFC 6014.

[3] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Protocol Modifications for the DNS Security Extensions. RFC 4035 (Proposed Standard), March 2005. Updated by RFCs 4470, 6014.

[4] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. Resource Records for the DNS Security Extensions. RFC 4034 (Proposed Standard), March 2005. Updated by RFCs 4470, 6014.

[5] R. Arends, M. Kosters, and D. Blacka. DNS Security (DNSSEC) Opt-In. RFC 4956 (Experimental), July 2007.

[6] D. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), March 1999. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 2931, 3007, 3008, 3090, 3226, 3445, 3597, 3655, 3658, 3755, 3757, 3845.

[7] B. Laurie, G. Sisson, R. Arends, and D. Blacka. DNS Security (DNSSEC) Hashed Authenticated Denial of Existence. RFC 5155 (Proposed Standard), March 2008.

[8] E. Lewis. The Role of Wildcards in the Domain Name System. RFC 4592 (Proposed Standard), July 2006.

[9] P.V. Mockapetris. Domain names - concepts and facilities. RFC 1034 (Standard), November 1987. Updated by RFCs 1101, 1183, 1348, 1876, 1982, 2065, 2181, 2308, 2535, 4033, 4034, 4035, 4343, 4035, 4592, 5936.

[10] J. Schlyter. DNS Security (DNSSEC) NextSECure (NSEC) RDATA Format. RFC 3845 (Proposed Standard), August 2004. Obsoleted by RFCs 4033, 4034, 4035.

[11] S. Weiler. Legacy Resolver Compatibility for Delegation Signer (DS). RFC 3755 (Proposed Standard), May 2004. Obsoleted by RFCs 4033, 4034, 4035, updated by RFCs 3757, 3845.

[12] B. Wellington and O. Gudmundsson. Redefinition of DNS Authenticated Data (AD) bit. RFC 3655 (Proposed Standard), November 2003. Obsoleted by RFCs 4033, 4034, 4035.