

BIND DNSSEC Guide

Copyright © 2014, 2015, 2016, 2017 Internet Systems Consortium, Inc.

Contents

1	Introduction	1
1.1	Who Should Read this Guide?	1
1.2	Who May Not Want to Read this Guide?	1
1.3	What is DNSSEC?	1
1.4	What does DNSSEC Add to DNS?	2
1.5	How Does DNSSEC Change DNS Lookup?	3
1.5.1	Chain of Trust	4
1.6	Why is DNSSEC Important? (Why Should I Care?)	5
1.7	How does DNSSEC Change My Job as a DNS Administrator?	5
2	Getting Started	6
2.1	Software Requirement	6
2.1.1	BIND Version	6
2.1.2	DNSSEC Support in BIND	6
2.1.3	System Entropy	6
2.2	Hardware Requirement	7
2.2.1	Recursive Server Hardware	7
2.2.2	Authoritative Server Hardware	7
2.3	Network Requirements	7
2.4	Operational Requirements	8
2.4.1	Parent Zone	8
2.4.2	Security Requirements	8
3	Validation	9
3.1	Easy Start Guide for Recursive Servers	9
3.2	How To Test Recursive Server (So You Think You Are Validating)	9
3.2.1	Using Web-based Tools to Verify	10
3.2.2	Using dig to Verify	10
3.2.3	Using delv to Verify	11
3.2.4	Verifying Protection from Bad Domain Names	12
3.2.5	How Do I know I Have a Validation Problem?	13

3.3	Validation Easy Start Explained	14
3.3.1	dnssec-validation	14
3.3.2	How Does DNSSEC Change DNS Lookup (Revisited)?	14
3.3.3	How are Answers Verified?	15
3.4	Trust Anchors	16
3.4.1	How Trust Anchors are Used	17
3.4.2	Trusted Keys and Managed Keys	18
3.5	What's EDNS All About (And Why Should I Care)?	19
3.5.1	EDNS Overview	19
3.5.2	EDNS on DNS Servers	19
3.5.3	Support for Large Packets on Network Equipment	20
3.5.4	Wait... DNS Uses TCP?	20
4	Signing	21
4.1	Easy Start Guide for Signing Authoritative Zones	21
4.1.1	Generate Keys	21
4.1.2	Reconfigure BIND	22
4.1.3	Verification	22
4.1.4	Upload to Parent Zone	23
4.1.5	So... What Now?	23
4.1.6	Your Zone, Before and After DNSSEC	23
4.2	How To Test Authoritative Zones (So You Think You Are Signed)	26
4.2.1	Look for Key Data in Your Zone	26
4.2.2	Look for Signatures in Your Zone	27
4.2.3	Examine the Zone File	27
4.2.4	Check the Parent	28
4.2.5	External Testing Tools	28
4.2.5.1	Verisign DNSSEC Debugger	28
4.2.5.2	DNSViz	29
4.3	Signing Easy Start Explained	31
4.3.1	Generate Keys Explained	31
4.3.2	Reconfigure BIND Explained	32
4.3.2.1	dnssec-enable	32
4.3.2.2	key-directory	32
4.3.2.3	inline-signing	33
4.3.2.4	auto-dnssec	33
4.4	Working with Parent Zone	33
4.4.1	DS Record Format	34
4.4.2	DNSKEY Format	34

4.4.3	Trusted Key Format	34
4.5	Using NSEC3	34
4.6	Maintenance Tasks	35
4.6.1	ZSK Rollover	35
4.6.2	KSK Rollover	35
5	Basic Troubleshooting	37
5.1	Query Path	37
5.2	Visible Symptoms	38
5.3	Logging	38
5.3.1	BIND DNSSEC Debug Logging	39
5.4	Common Problems	40
5.4.1	Security Lameness	40
5.4.2	Incorrect Time	41
5.4.3	Invalid Trust Anchors	42
5.4.4	Unable to Load Keys	42
5.5	Negative Trust Anchors	43
5.6	NSEC3 Troubleshooting	43
5.7	Troubleshooting Example	44
6	Advanced Discussions	49
6.1	Key Generation	49
6.1.1	Can I Use the Same Key Pair for Multiple Zones?	49
6.1.2	Do I Need Separate ZSK and KSK?	49
6.1.3	Which Algorithm?	49
6.1.4	Key Sizes	50
6.2	Proof of Non-Existence (NSEC and NSEC3)	50
6.2.1	NSEC	50
6.2.2	NSEC3	51
6.2.2.1	NSEC3PARAM	52
6.2.2.2	NSEC3 Opt-Out	53
6.2.2.3	NSEC3 Salt	53
6.2.3	NSEC or NSEC3?	53
6.3	Key Storage	53
6.3.1	Public Key Storage	53
6.3.2	Private Key Storage	54
6.3.3	Hardware Security Modules (HSM)	54
6.4	Key Management	54
6.4.1	Key Rollovers	54

6.4.1.1	ZSK Rollover Methods	55
6.4.1.2	KSK Rollover Methods	55
6.4.2	Key Management and Metadata	55
6.4.2.1	Manual Key Management and Signing	56
6.4.3	How Long Do I Wait Before Deleting Old Data?	57
6.4.4	Emergency Key Rollovers	57
6.4.5	DNSKEY Algorithm Rollovers	58
6.5	Other Topics	58
6.5.1	DNSSEC and Dynamic Updates	58
6.5.2	DNSSEC and Inline Signing	59
6.5.3	DNSSEC Look-aside Validation (DLV)	59
6.5.4	DNSSEC on Private Networks	60
6.5.5	Introduction to DANE	60
6.6	Disadvantages of DNSSEC	61
7	Recipes	62
7.1	Inline Signing Recipes	62
7.1.1	Master Server Inline Signing Recipe	62
7.1.2	"Bump in the Wire" Inline Signing Recipe	63
7.2	Rollover Recipes	64
7.2.1	ZSK Rollover Recipe	64
7.2.1.1	One Month Before ZSK Rollover	64
7.2.1.2	Day of ZSK Rollover	66
7.2.1.3	One Month After ZSK Rollover	66
7.2.2	KSK Rollover Recipe	67
7.2.2.1	One Month Before KSK Rollover	67
7.2.2.2	Day of KSK Rollover	72
7.2.2.3	One Month After KSK Rollover	73
7.3	NSEC and NSEC3 Recipes	75
7.3.1	Migrating from NSEC to NSEC3	75
7.3.2	Migrating from NSEC3 to NSEC	76
7.3.3	Changing NSEC3 Salt Recipe	76
7.3.4	NSEC3 Optout Recipe	77
7.4	Reverting to Unsigned Recipe	78
7.5	Self-signed Certificate Recipe	81
8	Commonly Asked Questions	84

List of Figures

1.1	DNSSEC Validation 12 Steps	3
3.1	Signature Generation	15
3.2	Signature Verification	16
3.3	DNSSEC Validation with .gov Trust Anchor	17
4.1	Verisign DNSSEC Debugger	29
4.2	DNSViz	30
5.1	Query Path	46
7.1	Inline Signing Recipe #1	62
7.2	Inline Signing Scenario #2	63
7.3	Upload DS Record Step #1	69
7.4	Upload DS Record Step #2	70
7.5	Upload DS Record Step #3	70
7.6	Upload DS Record Step #4	71
7.7	Upload DS Record Step #5	71
7.8	Upload DS Record Step #6	72
7.9	Remove DS Record Step #1	74
7.10	Remove DS Record Step #2	74
7.11	Remove DS Record Step #3	75
7.12	Revert to Unsigned Step #1	79
7.13	Revert to Unsigned Step #2	79
7.14	Revert to Unsigned Step #3	80
7.15	Revert to Unsigned Step #4	80
7.16	Browser Certificate Warning	82
7.17	DNSSEC TLSA Validator	83

List of Tables

4.1	ZSK KSK Comparison	32
6.1	Key Metadata Comparison	56

Abstract

This is version 1.2 of the DNSSEC deployment guide for BIND 9.

BIND 9 is open source software that implements the Domain Name System (DNS) protocols for the Internet. It is a reference implementation of those protocols, but it is also production-grade software, suitable for use in high-volume and high-reliability applications.

Domain Name System Security Extensions (DNSSEC) extends standard DNS to provide a measure of security; it proves that the data comes from the official source and has not been modified in transit.

ISC BIND 9 supports the full set of DNSSEC standards.

Preface

Organization

This document provides introductory information on how DNSSEC works, how to configure BIND 9 to support some common DNSSEC features, as well as some basic troubleshooting tips. The chapters are organized as such:

Chapter 1 covers the intended audience for this document, assumed background knowledge, and a basic introduction to the topic of DNSSEC.

Chapter 2 covers various requirements that are needed before implementing DNSSEC, such as software versions, hardware capacity, network requirements, and security changes.

Chapter 3 walks through setting up a validating resolver, more information on the validation process, as well as examples of using tools to verify that the resolver is validating answers.

Chapter 4 walks through setting up a basic signed authoritative zone, explains the relationship with the parent zone, and on-going maintenance tasks.

Chapter 5 provides some tips on how to analyze and diagnose DNSSEC-related problems.

Chapter 6 covers several topics, from key generation, key storage, key management, NSEC and NSEC3, to disadvantages of DNSSEC.

Chapter 7 provides several working examples of common solutions, with step-by-step details.

Chapter 8 lists some commonly asked questions and answers about DNSSEC.

Acknowledgement

This document is originally authored by Josh Kuo of *DeepDive Networking*. He can be reached at josh@deepdivenetworking.com

Thanks to the following individuals (in no particular order) who have helped in completing this document: Jeremy C. Reed, Heidi Schempf, Stephen Morris, Jeff Osborn, Vicky Risk, Jim Martin, Evan Hunt, Mark Andrews, Michael McNally, Kelli Blucher, Chuck Aurora, Francis Dupont, Rob Nagy and Ray Bellis.

Special thanks goes to Cricket Liu and Matt Larson for their selflessness in knowledge sharing.

Thanks to all the reviewers and contributors, including: John Allen, Jim Young, Tony Finch, Timothe Litt, and Dr. Jeffry A. Spain.

The sections on key rollover and key timing meta data borrowed heavily from the Internet Engineering Task Force draft titled "*DNSSEC Key Timing Considerations*" by S. Morris, J. Ihren, J. Dickinson, and W. Mekking, subsequently published as *RFC 7583*.

The recipe for TLSA self-signed certificate is based on the work of "*A Step-by-Step guide for implementing DANE with a Proof of Concept*" by Sandoche Balakrishnan, Stephane Bortzmeyer, and Mohsen Souissi (April 15, 2013) .

Icons made by *Freepik* and *SimpleIcon* from <http://www.flaticon.com>, licensed under *Creative Commons BY 3.0* .

Chapter 1

Introduction

1.1 Who Should Read this Guide?

This guide is intended to be an introduction to DNSSEC for the DNS administrator who is comfortable working with the existing BIND and DNS infrastructure. He or she might be curious about DNSSEC, but has not had the time to read up about what DNSSEC is (or is not), whether or not DNSSEC should be a part of her or his environment, and what it means to deploy it in the field.

This guide provides basic information on how to configure DNSSEC using BIND 9. Readers are assumed to have basic working knowledge of the Domain Name System (DNS) and related network infrastructure, such as concepts of TCP/IP. In-depth knowledge of DNS and TCP/IP is not required. The guide assumes no prior knowledge of DNSSEC or related technology such as public key cryptography.

1.2 Who May Not Want to Read this Guide?

If you are already operating a DNSSEC-signed zone, you may not learn much from the first half of the document, and you may want to start with Chapter 6. If you want to learn about details of the protocol extension, such as data fields and flags, or the new record types, this document can help you get started but it won't include all the technical details. If you are experienced in DNSSEC, you may find some of the concepts in this document to be overly simplified for your taste, and some details may be purposely omitted at times for illustration. If you administer a large or complex BIND environment, this guide may not provide enough information for you, as it is intended to provide the most basic, generic working examples. If you are a TLD operator, or administer zones under signed TLDs, this guide can help you get started, but does not provide enough details to serve all of your needs. If your DNS environment uses DNS products other than (or in addition to) BIND, this document may provide some background or overlapping information, but you should check each product's vendor documentation for specifics. Finally, deploying DNSSEC on internal or private networks is not covered in this document, with the exception of a brief discussion in Section 6.5.4.

1.3 What is DNSSEC?

The Domain Name System (DNS) was designed in a day and age when the Internet was a friendly and trusting place. The protocol itself provides little protection against malicious or forged answers. DNS Security Extensions (DNSSEC) addresses this need, by adding digital signatures into DNS data, so each DNS response can be verified for integrity (message did not change during transit) and authenticity (the data came from the true source, not an impostor). In the ideal world when DNSSEC is fully deployed, every single DNS answer can be validated and trusted.

DNSSEC does not provide a secure tunnel; it does not encrypt or hide DNS data. It operates independently of an existing Public Key Infrastructure (PKI). It does not need SSL certificates or shared secrets. It was designed with backwards compatibility in mind, and can be deployed without impacting "old" unsecured domain names.

DNSSEC is deployed on the three major components of the DNS infrastructure:

- *Recursive Server*: People use recursive servers to lookup external domain names such `www.example.com`. Operators of recursive servers need to enable DNSSEC validation. With validation enabled, recursive servers will carry out additional tasks on each DNS response they received to ensure its authenticity.
- *Authoritative Server*: People who publish DNS data on their name servers need to sign that data. This entails creating additional resource records, and publishing them to parent domains where necessary. With DNSSEC enabled, authoritative servers will respond to queries with additional DNS data, such as digital signatures and keys, in addition to the standard answers.
- *Application*: This component lives on every client machine, from web server to smart phones. This includes resolver libraries on different Operating Systems, and applications such as web browsers.

In this guide, we will focus on the first two components, Recursive Server and Authoritative Server, and only lightly touch on the third component. We will look at how DNSSEC works, how to configure a validating resolver, how to sign DNS zone data, and other operational tasks and considerations.

1.4 What does DNSSEC Add to DNS?

Primer on Public Key Cryptography

Public Key Cryptography works on the concept of a pair of keys, one is made available to the world publicly, and one is kept in secrecy privately. Not surprisingly, they are known as public key and private key. If you are not familiar with the concept, think of it as a cleverly designed lock, where one key locks, and one key unlocks. In DNSSEC, we give out the unlocking public key to the rest of the world, while keeping the locking key private. To learn how this is used to secure DNS messages, take a look at Section [3.3.3](#).

DNSSEC introduces six new resource record types:

- RRSIG (digital signature)
- DNSKEY (public key)
- DS (parent-child)
- NSEC (proof of nonexistence)
- NSEC3 (proof of nonexistence)
- NSEC3PARAM (proof of nonexistence)

This guide will not dissect into the anatomy of each resource record type, the details are left for the readers to research and explore. Below is a short introduction on each of the new record types:

- *RRSIG*: With DNSSEC enabled, just about every DNS answer (A, PTR, MX, SOA, DNSKEY, etc.) will come with at least one RRSIG, or resource record signature. These signatures are used by recursive name servers, also known as validating resolvers, to verify the answers received. To learn how digital signatures are generated and used, see Section [3.3.3](#).
- *DNSKEY*: DNSSEC relies on public key cryptography for data authenticity and integrity. There are several keys used in DNSSEC, some private, some public. The public keys are published to the world as part of the zone data, and they are stored in the DNSKEY record type. In general, there are two categories of keys used in DNSSEC, Zone Signing Key (ZSK) is used to protect all zone data, and Key Signing Key (KSK) is used to protect other keys. We will talk about keys in Section [4.3.1](#), and again later in Section [6.1](#).
- *DS*: One of the critical components of DNSSEC is that the parent zone can "vouch" for its child zone. The DS record is verifiable information (generated from one of the child's public keys) that a parent zone publishes about its child as part of the chain of trust. To learn more about the Chain of Trust, see Section [1.5.1](#).

The NSEC, NSEC3, and NSEC3PARAM resource records all deal with a very interesting problem: proving that something really does not exist. We will look at these record types in more detail in Section [6.2](#).

1.5 How Does DNSSEC Change DNS Lookup?

Traditional (insecure) DNS lookup is simple: a recursive name server receives a query from a client to lookup the name `www.isc.org`. The recursive name server tracks down the authoritative name server(s) responsible, sends the query to one of the authoritative name servers, and waits for the authoritative name server to respond with the answer.

With DNSSEC validation enabled, a validating recursive name server (a.k.a. a *validating resolver*) will ask for additional resource records in its query, hoping the remote authoritative name servers will respond with more than just the answer to the query, but some proof to go along with the answer as well. If DNSSEC responses are received, the validating resolver will perform cryptographic computation to verify the authenticity (origin of the data) and integrity (data was not altered during transit) of the answers, and even ask the parent zone as part of the verification. It will repeat this process of get-key, validate, ask-parent, parent, and its parent, and its parent, all the way until the validating resolver reaches a key that it trusts. In the ideal, fully deployed world of DNSSEC, all validating resolvers only need to trust one key: the root key.

The following example shows the DNSSEC validating process of looking up the name `www.isc.org` at a very high level:

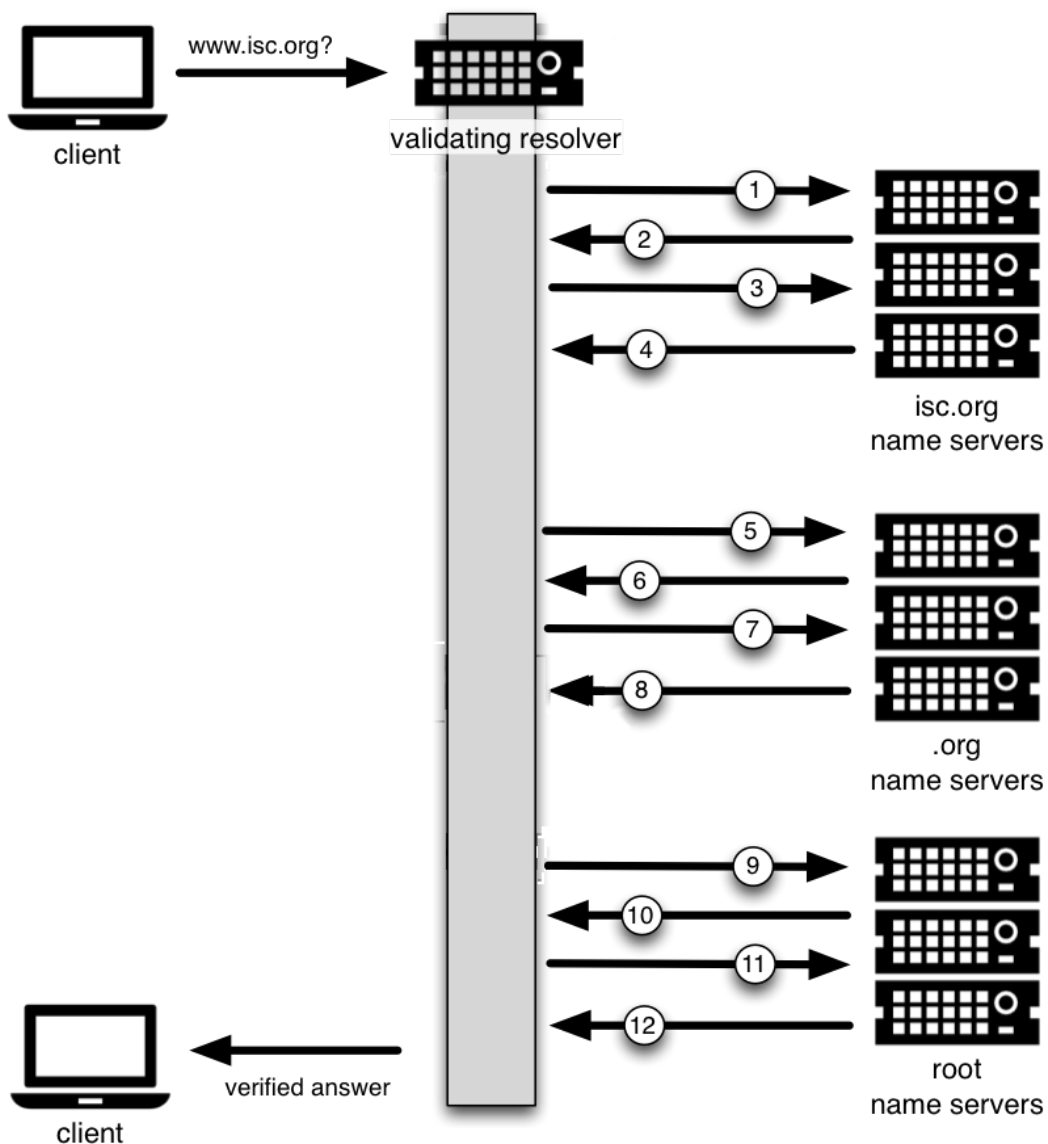


Figure 1.1: DNSSEC Validation 12 Steps

1. Upon receiving a DNS query from a client to resolve `www.isc.org`, the validating resolver follows standard DNS

protocol to track down the name server for `isc.org`, sends it a DNS query to ask for the A record of `www.isc.org`. But since this is a DNSSEC-enabled resolver, the outgoing query has a bit set indicating it wants DNSSEC answers, hoping the name server who receives it speaks DNSSEC and can honor this secure request.

2. `isc.org` name server is DNSSEC-enabled, and responds with the answer (in this case, an A record), and with a digital signature for verification purpose.
3. The validating resolver needs to be able to verify the digital signature, and to do so it requires cryptographic keys. So it asks the `isc.org` name server for those keys.
4. `isc.org` name server responds with the cryptographic keys (and digital signatures of the keys) used to generate the digital signature that was sent in #2. At this point, the validating resolver can use this information to verify the answers received in #2.

Let's take a quick break here and look at what we've got so far... how could we trust this answer? If a clever attacker had taken over the `isc.org` name server(s), or course she would send matching keys and signatures. We need to ask someone else to have confidence that we are really talking to the real `isc.org` name server. This is a critical part of DNSSEC: at some point, the DNS administrators at `isc.org` had uploaded some cryptographic information to its parent, `.org`, maybe it was a secure web submit form, maybe it was through an email exchange, or perhaps it was done in person. No matter the case, at some point, some verifiable information about the child (`isc.org`) was sent to the parent (`.org`), for safekeeping.

5. The validating resolver asks the parent (`.org`) for the verifiable information it keeps on its child, `isc.org`.
6. The verifiable information is sent from the `.org` server. At this point, validating resolver compares this to the answer it received in #4, and the two of them should match, proving the authenticity of `isc.org`.

Let's examine this process. You might be thinking to yourself, well, what if the clever attacker that took over `isc.org` also compromised the `.org` servers? Of course all this information would match! That's why we will turn our attention now to the `.org` servers, interrogate it for its cryptographic keys, and move on one level up to `.org`'s parent, root.

7. The validating resolver asks `.org` authoritative name servers for its cryptographic keys, for the purpose of verifying the answers received in #6.
8. `.org` name server responds with the answer (in this case, keys and signatures). At this point, validating resolver can verify the answers received in #6.
9. The validating resolver asks root (`.org`'s parent) for verifiable information it keeps on its child, `.org`.
10. Root name server sends back the verifiable information it keeps on `.org`. The validating resolver now takes this information and uses it to verify the answers received in #8.

So up to this point, both `isc.org` and `.org` check out. But what about root? What if this attacker is really clever and somehow tricked us into thinking she's the root name server? Of course she would send us all matching information! So we repeat the interrogation process and ask for the keys from the root name server.

11. Validating resolver asks root name server for its cryptographic keys in order to verify the answer(s) received in #10.
12. Root name server sends keys, at this point, validating resolver can verify the answer(s) received in #10.

1.5.1 Chain of Trust

But what about the root server itself? Who do we go to verify root's keys? There's no parent zone for root. In security, you have to trust someone, and in the perfectly protected world of DNSSEC (we'll talk about the current imperfect state later and ways to work around it), each validating resolver would only have to trust one entity, that is the root name server. The validating resolver would already have the root key on file (and we'll talk about later how we got the root key file). So after answers in #12 are received, validating resolver takes the answer received and compare it to the key it already has on file, and these two should match. If they do, it means we can trust the answer from root, thus we can trust `.org`, and thus we can trust `isc.org`. This is known as "chain of trust" in DNSSEC.

We will revisit this 12-step process again later in Section 3.3.2 with more technical details.

1.6 Why is DNSSEC Important? (Why Should I Care?)

You might be thinking to yourself: all this DNSSEC stuff sounds wonderful, but why should I care? Below are some reasons why you may want to consider deploying DNSSEC:

1. *Be a good netizen*: By enabling DNSSEC validation (as described in Chapter 3) on your DNS servers, you're protecting your users or yourself a little more by checking answers returned to you; by signing your zones (as described in Chapter 4), you are making it possible for other people to verify your zone data. As more people adopt DNSSEC, the Internet as a whole becomes more secure for everyone.
2. *Compliance*: You may not even get a say whether or not you want to implement DNSSEC, if your organization is subject to compliance standards that mandate it. For example, the US government set a deadline back in 2008, to have all `.gov` sub domains signed by the December 2009¹. So if you operated a sub domain in `.gov`, you must implement DNSSEC in order to be compliant. One of the widely used compliance standards, PCI DSS for the payment card industry, has been rumored to list DNSSEC as a requirement or recommendation, as part of its on-going efforts to enhance security for online payment transactions. ICANN also requires that all new top-level domains support DNSSEC.
3. *Enhanced Security (C.Y.A.)*: Okay, so the big lofty goal of "let's be good" doesn't appeal to you, and you don't have any compliance standards to worry about. Here is a more practical reason why you should consider DNSSEC: in case of a DNS-based security breach, such as cache poisoning or domain hijacking, after all the financial and brand damage done to your domain name, you might be placed under scrutiny for any preventative measure that could have been put in place. Think of this like having your web site only available via HTTP but not HTTPS.
4. *New Features*: DNSSEC brings not only enhanced security, but with that new level of security, a whole new suite of features. Once DNS can be trusted completely, it becomes possible to publish SSL certificates in DNS, or PGP keys for fully automatic cross-platform email encryption, or SSH fingerprints... People are still coming up with new features, but this all relies on a trust-worthy DNS infrastructure. To take a peek at these next generation DNS features, check out Section 6.5.5.

1.7 How does DNSSEC Change My Job as a DNS Administrator?

With this protocol extension, some of the things you are used to in DNS will change. As the DNS administrator, you will have new maintenance tasks to perform on a regular basis (as described in Section 4.6); when there's a DNS resolution problem, you have new troubleshooting techniques and tools to use (as described in Chapter 5). BIND tries its best to make these things as transparent and seamless as possible. In this guide, we try to use the configuration examples that result in the least amount of work for DNS administrators.

¹ The Office of Management and Budget (OMB) for the US government published a memo in 2008 (www.whitehouse.gov/sites/default/files/omb/memoranda/fy2008/m08-23.pdf), requesting all `.gov` sub-domains to be DNSSEC signed by December 2009. This explains why `.gov` is the most deployed DNSSEC domain currently, with more than 80% sub domains signed.

Chapter 2

Getting Started

2.1 Software Requirement

2.1.1 BIND Version

The configuration examples given in this document requires BIND version 9.9 or newer. To check the version of **named** you have installed, use the `-v` switch as shown below:

```
# named -v
BIND 9.10.1
```

2.1.2 DNSSEC Support in BIND

All versions of BIND 9 since BIND 9.7 can support DNSSEC as currently deployed in the global DNS. The BIND software you are running most likely already supports DNSSEC as shipped. Run the command **named -V** to see what flags it was built with. If it was built with OpenSSL (`--with-openssl`), then it supports DNSSEC. Below is an example screenshot of running **named -V**:

```
$ named -V
BIND 9.10.1 <id:fe66c6b1> built by make with '--prefix=/opt/local'
'--mandir=/opt/local/share/man' '--with-openssl=/opt/local'
'--with-libxml2=/opt/local' '--without-libjson' '--enable-threads'
'--enable-ipv6' 'CC=/usr/bin/clang' 'CFLAGS=-pipe -Os -arch x86_64'
'LDLDFLAGS=-L/opt/local/lib -Wl,-headerpad_max_install_names -arch x86_64'
'CPPFLAGS=-I/opt/local/include'
compiled by CLANG 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)
using OpenSSL version: OpenSSL 1.0.1i 6 Aug 2014
using libxml2 version: 2.9.1
```

If the BIND 9 software you have does not support DNSSEC, it may be necessary to rebuild it from source or upgrade to a newer version.

2.1.3 System Entropy

If you plan on deploying DNSSEC to your authoritative server, you will need to generate cryptographic keys (Section 4.3.1). The amount of time it takes to generate the keys depends on the source of randomness, or entropy, on your systems. On some systems (especially virtual machines) with insufficient entropy, it may take much longer than one cares to wait to generate keys.

There are software packages, such as **haveged** for Linux, that provides additional entropy for your system. Once installed, they will significantly reduce the time needed to generate keys.

The more entropy there is, the better pseudo-random numbers you get, and stronger keys are generated. If you want or need high quality random numbers, take a look at Section 6.3.3 for some of the hardware-based solutions.

2.2 Hardware Requirement

2.2.1 Recursive Server Hardware

Enabling DNSSEC validation on a recursive server makes it a validating resolver. The job of a validating resolver is to fetch additional information that can be used to computationally verify the answer set. Below are the areas that should be considered for possible hardware enhancement for a validating resolver:

1. *CPU*: a validating resolver executes cryptographic functions on many of the answers returned, this usually leads to increased CPU usage, unless your recursive server has built-in hardware to perform cryptographic computations.
2. *System memory*: DNSSEC leads to larger answer sets, and will occupy more memory space.
3. *Network interfaces*: although DNSSEC does increase the amount of DNS traffic overall, it is unlikely that you need to upgrade your network interface card (NIC) on the name server, unless you have some truly out-dated hardware.

One of the factors to consider is the destinations of your current DNS traffic. If your current users spend a lot of time visiting `.gov` web sites, then you should expect a bigger jump in all of the above categories when validation is enabled, because `.gov` is more than 80% signed. This means, more than 80% of the time, your validating resolver will be doing what is described in Section 1.5. However, if your users only care about resources in the `.com` domain, which as of this writing, is 0.5% signed, then your recursive name server is unlikely to experience significant load increase after enabling DNSSEC validation.

2.2.2 Authoritative Server Hardware

On the authoritative server side, DNSSEC is enabled on a zone-by-zone basis. When a zone is DNSSEC-enabled, it is also known as "signed". Below are the areas that you should consider for possible hardware enhancements for an authoritative server with signed zones:

1. *CPU*: DNSSEC signed zone requires periodic re-signing, which is a cryptographic function that is CPU intensive. If your DNS zone is dynamic or changes frequently, it also adds to higher CPU loads.
2. *System storage*: A signed zone is definitely larger than an unsigned zone. How much larger? See Section 4.1.6 for a comparison example. Roughly speaking, you could expect your zone file to grow at least three times as large, usually more.
3. *System memory*: Larger DNS zone files take up not only more storage space on the file system, but also more space when they are loaded into system memory.
4. *Network interfaces*: While your authoritative name servers will begin sending back larger responses, it is unlikely that you need to upgrade your network interface card (NIC) on the name server, unless you have some truly out-dated hardware.

One of the factors to consider, but you really have no control over, is how many users who query your domain name have DNSSEC enabled. It was estimated in late 2014, that roughly 10% to 15% of the Internet DNS queries were DNSSEC aware, and since then Google DNS has become DNSSEC enabled and is used by a further 15% of global DNS users. This translates to roughly 25% to 30% of the DNS queries for your domain will take advantage the additional security features, which result in the increased system load and possibly network traffic.

2.3 Network Requirements

From a network perspective, DNS and DNSSEC packets are very similar, DNSSEC packets are just bigger, which means DNS is more likely to use TCP. You should test for the following two items, to make sure your network is ready for DNSSEC:

1. *DNS over TCP*: Verify network connectivity over TCP port 53, this may mean updating firewall policies or Access Control List (ACL) on routers. See Section 3.5.4 more details.
2. *Large UDP packets*: Some network equipment such as firewalls may make assumptions about the size of DNS UDP packets and incorrectly reject DNS traffic that appears "too big". You should verify that the responses your nameserver generates are being seen by the rest of the world. See Section 3.5 for more details.

2.4 Operational Requirements

2.4.1 Parent Zone

Before starting your DNSSEC deployment, check with your parent zone administrators to make sure they support DNSSEC. This may or may not be the same entity as your registrar. As you will see later in Section 4.4, a crucial step in DNSSEC deployment is to establish the parent-child trust relationship. If your parent zone does not support DNSSEC yet, contact them to voice your concern.

ICANN maintains a list of registrars who support DNSSEC:

<https://www.icann.org/resources/pages/deployment-2012-02-25-en>

2.4.2 Security Requirements

Some organizations may be subject to stricter security requirements than others. Check to see if your organization requires stronger cryptographic keys be generated and stored, or how often keys need to be rotated. The examples presented in this document are not intended for high value zones. We cover some of these security considerations in Chapter 6.

Chapter 3

Validation

3.1 Easy Start Guide for Recursive Servers

This section provides the minimum amount of information to setup a working DNSSEC-aware recursive server, also known as a validating resolver. A validating resolver performs validation for each remote response received, following the chain of trust to verify the answers it receives are legitimate through the use of public key cryptography and hashing functions.

Once DNSSEC validation is enabled, any DNS response that does not pass the validation checks will result in the domain name not getting resolved (often a SERVFAIL status seen by the client). What this means for the DNS administrator is, if there is a DNSSEC configuration issue (sometimes outside of the administrator's control), a specific name, or sometimes entire domains, may "disappear" from DNS, in that it becomes unreachable through that resolver. What this means for the end user is, name resolution is slow or fails altogether, or some parts of a URL will not load, or web browser will display some error message indicating the page cannot be displayed at all.

For example, if root name servers were misconfigured with the wrong information about `.org`, it could cause all validation for `.org` domains to fail. To the end users, it would appear that no one could get to any `.org` web sites.

You may not need to reconfigure your name server at all, since recent versions of BIND packages and distributions have been shipped with DNSSEC validation enabled by default. Before making any configuration changes, check whether or not you already have DNSSEC validation by following steps described in Section 3.2.

Enabling DNSSEC validation on a BIND 9 recursive name server is easy, you only need one line of configuration in your configuration file:

```
options {
    dnssec-validation auto;
};
```

Restart **named** or use **rndc reconfig**, and your recursive server is now happily validating each DNS response. If this does not work for you, and you have already verified DNSSEC support as described in Section 2.1.2, you most likely have some other network-related configurations that need to be adjusted, take a look at Section 2.3 to make sure your network is ready for DNSSEC.

DNSSEC is enabled by default for BIND, but this line enables automatic trust anchor configuration. To learn more about this configuration, please refer to Section 3.3.

3.2 How To Test Recursive Server (So You Think You Are Validating)

Okay, so now that you have reconfigured your recursive server and restarted it, how do you know that your recursive name server is actually verifying each DNS query? There are several ways to check, and we've listed a few suggestions below, starting with the easiest.

3.2.1 Using Web-based Tools to Verify

For most people, the simplest way to check if the recursive name server is indeed validating DNS queries, is to use one of the many web-based tools.

Configure your client computer to use the newly reconfigured recursive server for DNS resolution, and then you can use any one of these web-based tests to see if it is in fact validating answers DNS responses.

- <http://en.conn.internet.nl/connection/>
- <http://www.dnssec-tools.org/>
- <http://dnssec.vs.uni-due.de/>

3.2.2 Using dig to Verify

The web-based tools often employ JavaScript. If you don't trust the JavaScript magic that the web-based tools rely on, you can take matters into your own hands and use a command line DNS tool to check your validating resolver yourself.

While nslookup is popular, partly because it comes pre-installed on most systems, it is not DNSSEC-aware. The Domain Information Groper (**dig**), on the other hand, fully supports the DNSSEC standard, and comes as a part of BIND. If you do not have dig already installed on your system, install it by downloading it from ISC's web site. ISC provides pre-compiled Windows versions on its web site.

dig is a flexible tool for interrogating DNS name servers. It performs DNS lookups and displays the answers that are returned from the name server(s) that were queried. Most seasoned DNS administrators use **dig** to troubleshoot DNS problems because of its flexibility, ease of use, and clarity of output.

The example below shows using dig to query the name server 192.168.1.7 for the A record for `www.isc.org` if DNSSEC is disabled. The address 192.168.1.7 is only used as an example, you should replace this address with the actual address or host name of your recursive name server. Notice although we specifically used the `+dnssec` command line option, we do not see the DNSSEC OK (`do`) bit in the response, nor do we see any DNSSEC resource records.

```
$ dig @192.168.1.7 www.isc.org. A +dnssec +multiline
; <<>> DiG 9.10.0-P2 <<>> @192.168.1.7 www.isc.org. A +dnssec +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20416
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.isc.org.      IN  A

;; ANSWER SECTION:
www.isc.org.      60  IN  A 149.20.64.69
```

Below shows what the results look like querying the same server (192.168.1.7) after enabling DNSSEC validation. The exact same command is run, and this time notice three key differences:

1. The presence of the Authenticated Data (`ad`) flag in the header
2. The DNSSEC OK (`do`) flag indicating the recursive server is DNSSEC-aware
3. An additional resource record of type RRSIG, with the same name as the A record.

The DNSSEC OK (`do`) flag tells us that the recursive server we are querying (192.168.1.7 in this example) is DNSSEC-aware but not necessarily that it is configured to perform DNSSEC validation. The Authenticated Data (`ad`) flag tells us that the answer received has passed the validation process as described in Section 3.3.3. We can have confidence in the authenticity and integrity of the answer that `www.isc.org` really points to the IP address 149.20.64.69, and it was not a spoofed answer from a clever attacker.

```

$ dig @192.168.1.7 www.isc.org. A +dnssec +multiline

; <<>> DiG 9.10.0-P2 <<>> @192.168.1.7 www.isc.org. A +dnssec +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 32472
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;www.isc.org.      IN A

;; ANSWER SECTION:
www.isc.org.      4 IN A 149.20.64.69
www.isc.org.      4 IN RRSIG A 5 3 60 (
    20141029233238 20140929233238 4521 isc.org.
    DX5BaGVd4KzU2AIH911Kar/UmdmkARyPhJVLr0oyPZaq
    5zoobGqFI4efvzL0mcpncuUg3BSU5Q48WdBu92xinMdb
    E75z1+adgEBOsFgFQR/zqM3myt/8SngWm4+TQ3XFh9eN
    jqExHZZuZ268Ntlxqgf9OmKRRv8X8YigaPShuyU= )

;; Query time: 3 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Fri Oct 03 16:40:04 CST 2014
;; MSG SIZE rcvd: 223

```

3.2.3 Using delv to Verify

If you have BIND 9.10 or later, you can use Domain Entity Lookup & Validation (**delv**) to validate your setup. This program is similar to **dig**, but is specifically tailored for DNSSEC.

Without DNSSEC:

```

$ delv @192.168.1.7 www.isc.org. A
;; no valid RRSIG resolving 'org/DS/IN': 192.168.1.7#53
;; no valid DS resolving 'www.isc.org/A/IN': 192.168.1.7#53
;; resolution failed: no valid DS

```

delv also comes with a handy **+rtrace** (trace resolver fetches) switch that shows a little more information on what was fetched:

```

$ delv @192.168.1.7 www.isc.org. A +rtrace
;; fetch: www.isc.org/A
;; fetch: org/DS
;; no valid RRSIG resolving 'org/DS/IN': 192.168.1.7#53
;; no valid DS resolving 'www.isc.org/A/IN': 192.168.1.7#53
;; resolution failed: no valid DS

```

After enabling DNSSEC validation, re-running the exact same codes show us the following results:

```

$ delv @192.168.1.7 www.isc.org. A +multiline
; fully validated
www.isc.org.      60 IN A 149.20.64.69
www.isc.org.      60 IN RRSIG A 5 3 60 (
    20141029233238 20140929233238 4521 isc.org.
    DX5BaGVd4KzU2AIH911Kar/UmdmkARyPhJVLr0oyPZaq
    5zoobGqFI4efvzL0mcpncuUg3BSU5Q48WdBu92xinMdb
    E75z1+adgEBOsFgFQR/zqM3myt/8SngWm4+TQ3XFh9eN
    jqExHZZuZ268Ntlxqgf9OmKRRv8X8YigaPShuyU= )

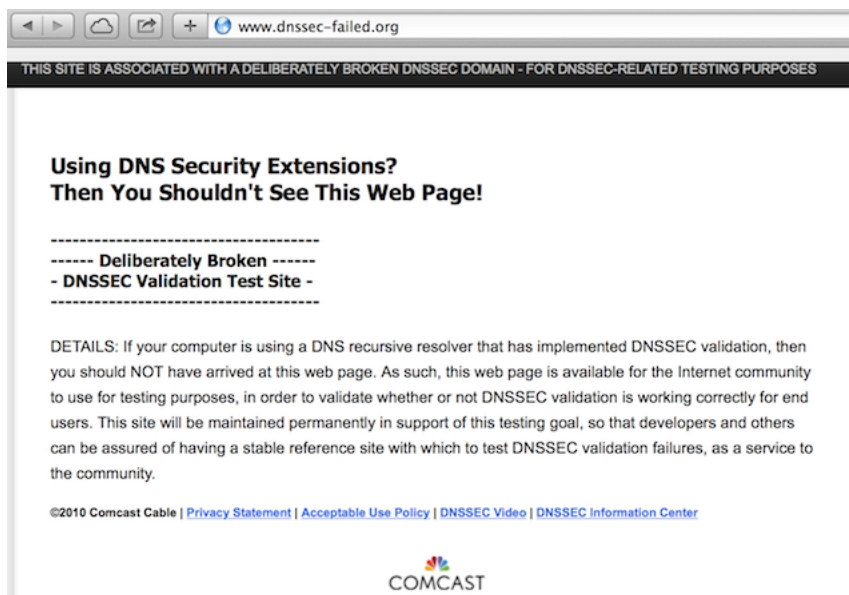
```

And `+rtrace` shows all the glory of what records were fetched to validate this answer:

```
$ delv @192.168.1.7 www.isc.org +rtrace +multiline
;; fetch: www.isc.org/A
;; fetch: isc.org/DNSKEY
;; fetch: isc.org/DS
;; fetch: org/DNSKEY
;; fetch: org/DS
;; fetch: ./DNSKEY
; fully validated
www.isc.org.      19 IN A 149.20.64.69
www.isc.org.      19 IN RRSIG A 5 3 60 (
    20141029233238 20140929233238 4521 isc.org.
    DX5BaGVd4KzU2AIH911Kar/UmdmkARyPhJVLr0oyPZaq
    5zoobGqFI4efvzL0mcpncuUg3BSU5Q48WdBu92xinMdb
    E75zl+adgEBOsFgFQR/zqM3myt/8SngWm4+TQ3XFh9eN
    jqExHZZuZ268Ntlxqgf9OmKRRv8X8YigaPShuyU= )
```

3.2.4 Verifying Protection from Bad Domain Names

It is also important to make sure that DNSSEC is protecting you from domain names that fail to validate; such failures could be caused by attacks on your system, attempting to get it to accept false DNS information. Validation could fail for a number of reasons, maybe the answer doesn't verify because it's a spoofed response; maybe the signature was a replayed network attack that has expired; or maybe the child zone has been compromised along with its keys, and the parent zone's information is telling us that things don't add up. There is a domain name specifically setup to purposely fail DNSSEC validation, `www.dnssec-failed.org`. Prior to enabling DNSSEC validation, you should have no trouble visiting the URL <http://www.dnssec-failed.org/> in your web browser:



And no problem resolving the domain name as shown below using `dig`:

```
$ dig @192.168.1.7 www.dnssec-failed.org. A

; <<>> DiG 9.10.1 <<>> @192.168.1.7 www.dnssec-failed.org. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28878
;; flags: qr rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
```

```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags;; udp: 4096
;; QUESTION SECTION:
;www.dnssec-failed.org.    IN  A

;; ANSWER SECTION:
www.dnssec-failed.org.    7200 IN  A 68.87.109.242
www.dnssec-failed.org.    7200 IN  A 69.252.193.191

;; Query time: 955 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Fri Oct 17 07:42:50 CST 2014
;; MSG SIZE rcvd: 82
```

After DNSSEC validation is enabled, any attempt to loading the URL should result in some kind of "Sorry, this page cannot be displayed" error message from your web browser. And looking up this domain name using **dig** should result in **SERVFAIL**, as shown below:

```
$ dig @192.168.1.7 www.dnssec-failed.org. A

; <<>> DiG 9.10.1 <<>> @192.168.1.7 www.dnssec-failed.org. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 46592
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags;; udp: 4096
;; QUESTION SECTION:
;www.dnssec-failed.org.    IN  A

;; Query time: 2435 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Fri Oct 17 07:44:56 CST 2014
;; MSG SIZE rcvd: 50
```

3.2.5 How Do I know I Have a Validation Problem?

Since all DNSSEC validation failures result in a general **SERVFAIL** message, how do we know that it was related to validation in the first place? Fortunately, there is a flag in **dig**, (**+cd**, checking disabled) which tells the server to disable DNSSEC validation. When you've received a **SERVFAIL** message, re-run the query one more time, and throw in the **+cd** flag. If the query succeeds with **+cd**, but ends in **SERVFAIL** without it, then you know you are dealing with a validation problem.

```
$ dig @192.168.1.7 www.isc.org. A +cd

; <<>> DiG 9.10.1 <<>> @192.168.1.7 www.isc.org. A +cd
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33590
;; flags: qr rd ra cd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags;; udp: 4096
;; QUESTION SECTION:
;www.isc.org.             IN  A

;; ANSWER SECTION:
www.isc.org.             30  IN  A 149.20.64.69
```

For more information on troubleshooting, please see Chapter 5.

3.3 Validation Easy Start Explained

In Section 3.1, we used one line of configuration to turn on DNSSEC validation, the act of chasing down signatures and keys, making sure they are authentic. Now we are going to take a closer look at what it actually does, and some other options.

3.3.1 dnssec-validation

```
options {  
    dnssec-validation auto;  
};
```

This “auto” line enables automatic DNSSEC trust anchor configuration using the `managed-keys` feature. In this case, no manual key configuration is needed. There are three possible choices for the `dnssec-validation` option:

- *yes*: DNSSEC validation is enabled, but a trust anchor must be manually configured. No validation will actually take place until you have manually configured at least one trusted key. This is the default.
- *no*: DNSSEC validation is disabled, and recursive server will behave in the “old fashioned” way of performing insecure DNS lookups.
- *auto*: DNSSEC validation is enabled, and a default trust anchor (included as part of BIND) for the DNS root zone is used.

Let’s discuss the difference between *yes* and *auto*. If you set it to *yes* (the default), the trust anchor will need to be manually defined and maintained using the `trusted-keys` statement in the configuration file; if you set it to *auto* (as shown in the example), then no further action should be required as BIND includes a copy¹ of the root key. When set to *auto*, BIND will automatically keep the keys (also known as trust anchors, which we will look at in Section 3.4) up-to-date without intervention from the DNS administrator.

We recommend *auto* unless you have a good reason for requiring a manual trust anchor. To learn more about trust anchors, please refer to Section 3.4.2.

Note `dnssec-enable` needs to be set to *yes* (default value is *yes*) in order for `dnssec-validation` to be effective.

3.3.2 How Does DNSSEC Change DNS Lookup (Revisited)?

So by now you’ve enabled validation on your recursive name server, and verified that it works. What exactly changed? In Section 1.5 we looked at the very high level, simplified 12-steps of DNSSEC validation process. Let’s revisit that process now and see what your validating resolver is doing in more detail. Again, we are using the example to lookup the A record for the domain name `www.isc.org` (Figure 1.1):

1. Validating resolver queries `isc.org` name servers for the A record of `www.isc.org`. This query has the DNSSEC OK (do) bit set to 1, notifying the remote authoritative server that DNSSEC answers are desired.
2. The zone `isc.org` is signed, and its name servers are DNSSEC-aware, thus it responds with the answer to the A record query plus the RRSIG for the A record.
3. Validating resolver queries for the DNSKEY for `isc.org`.
4. `isc.org` name server responds with the DNSKEY and RRSIG records. The DNSKEY is used to verify the answers received in #2.

¹ BIND technically includes two copies of the root key, one is in `bind.keys.h` and is built into the executable, and one is in `bind.keys` as a `managed-keys` statement. The two copies of the key are identical.

5. Validating resolver queries the parent (.org) for the DS record for isc.org.
6. .org name server responds with the DS and RRSIG records. The DS record is used to verify the answers received in #4.
7. Validating resolver queries for the DNSKEY for .org.
8. .org name server responds with DNSKEY and RRSIG. The DNSKEY is used to verify the answers received in #6.
9. Validating resolver queries the parent (root) for the DS record for .org.
10. Root name server responds with DS and RRSIG records. The DS record is used to verify the answers received in #8.
11. Validating resolver queries for the DNSKEY for root.
12. Root name server responds with DNSKEY and RRSIG. The DNSKEY is used to verify the answers received in #10.

After step #12, the validating resolver takes the DNSKEY received and compares to the key or keys it has configured, to decide whether or not the received key can be trusted. We will talk about these locally configured keys, or trust anchors, in Section 3.4.

As you can see here, with DNSSEC, every response includes not just the answer, but a digital signature (RRSIG) as well. This is so the validating resolver can verify the answer received, and that's what we will look at in the next section, Section 3.3.3.

3.3.3 How are Answers Verified?

Note Keep in mind as you read this section, that although words like *encryption* and *decryption* are used from time to time, DNSSEC does not provide you with privacy. Public key cryptography is used to provide data authenticity (who sent it) and data integrity (it did not change during transit), but any eavesdropper can still see your DNS requests and responses in clear text, even when DNSSEC is enabled.

So how exactly are DNSSEC answers verified? Before we can talk about how they are verified, let's first see how verifiable information is generated. On the authoritative server, each DNS record (or message) is run through a hash function, then this hashed value is encrypted by a private key. This encrypted hash value is the digital signature.

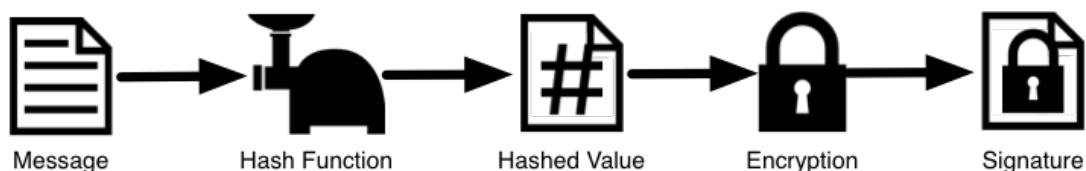


Figure 3.1: Signature Generation

When the validating resolver queries for the resource record, it receives both the plain-text message and the digital signature(s). The validating resolver knows the hash function used (listed in the digital signature record itself), so it can take the plain-text message and run it through the same hash function to produce a hashed value, let's call it hash value X. The validating resolver can also obtain the public key (published as DNSKEY records), decrypt the digital signature, and get back the original hashed value produced by the authoritative server, let's call it hash value Y. If hash values X and Y are identical, and the time is correct (more on what this means below), the answer is verified, meaning we know this answer came from the authoritative server (authenticity), and the content remained intact during transit (integrity).

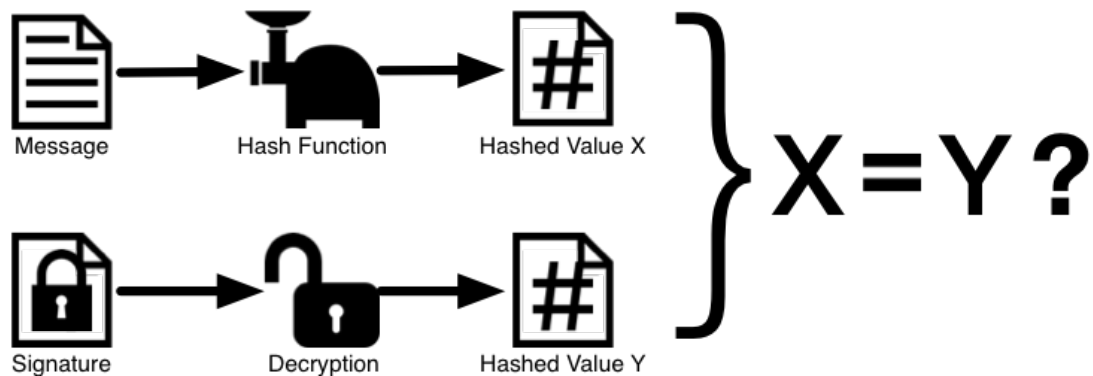


Figure 3.2: Signature Verification

Take the A record `www.isc.org` for example, the plain text is:

```
www.isc.org.      4 IN A   149.20.64.69
```

The digital signature portion is:

```
www.isc.org.      4 IN RRSIG A 5 3 60 (
    20141029233238 20140929233238 4521 isc.org.
    DX5BaGVd4KzU2AIH911Kar/UmdmkARyPhJVLr0oyPZaq
    5zoobGqFI4efvzL0mcpncuUg3BSU5Q48WdBu92xinMdb
    E75z1+adgEBOsFgFQR/zqM3myt/8SngWm4+TQ3XFh9eN
    jqExHZZuZ268Ntlxqgf9OmKRRv8X8YigaPShuyU= )
```

When a validating resolver queries for the A record `www.isc.org`, it receives both the A record and the RRSIG record. It runs the A record through a hash function (in this example, it would be SHA1 as indicated by the number 5, signifying RSA-SHA1) and produces hash value X. The resolver also fetches the appropriate DNSKEY record to decrypt the signature, and the result of the decryption is hash value Y.

But wait! There's more! Just because X equals Y doesn't mean everything is good. We still have to look at the time. Remember we mentioned a little earlier that we need to check if the time is correct? Well, look at the two highlighted timestamps in our example above, the two timestamps are:

- Signature Expiration: 20141029233238
- Signature Inception: 20140929233238

This tells us that this signature was generated UTC September 29th, 2014, at 11:32:38PM (20140929233238), and it is good until UTC October 29th, 2014, 11:32:38PM (20141029233238). And the validating resolver's current system time needs to fall between these two timestamps. Otherwise the validation fails, because it could be an attacker replaying an old captured answer set from the past, or feeding us a crafted one with incorrect future timestamps.

If the answer passes both hash value check and timestamp check, it is validated, and the authenticated data (`ad`) bit is set, and response is sent to the client; if it does not verify, a `SERVFAIL` is returned to the client.

3.4 Trust Anchors

A trust anchor is a key that is placed into a validating resolver so that the validator can verify the results for a given request back to a known or trusted public key (the trust anchor). A validating resolver must have at least one trust anchor installed in order to perform DNSSEC validation.

3.4.1 How Trust Anchors are Used

In the section Section 3.3.2, we walked through the DNSSEC lookup process (12 steps), and at the end of the 12 steps, a critical comparison happens: the key received from the remote server, and the key we have on file are compared to see if we trust it. The key we have on file is called a trust anchor, sometimes also known as a trust key, trust point, or secure entry point.

The 12-step lookup process describes the DNSSEC lookup in the ideal world where every single domain name is signed and properly delegated, each validating resolver only needs to have one trust anchor, and that is the root's public key. But there is no restriction that the validating resolver must only have one trust anchor. In fact, in the early stages of DNSSEC adoption, it was not unusual for a validating resolver to have more than one trust anchor.

For instance, before the root zone was signed (prior to the year 2010), some validating resolvers that wish to validate domain names in the `.gov` zone needed to obtain and install the key for `.gov`. A sample lookup process for `www.fbi.gov` would thus be only 8 steps rather than 12 steps that look like this:

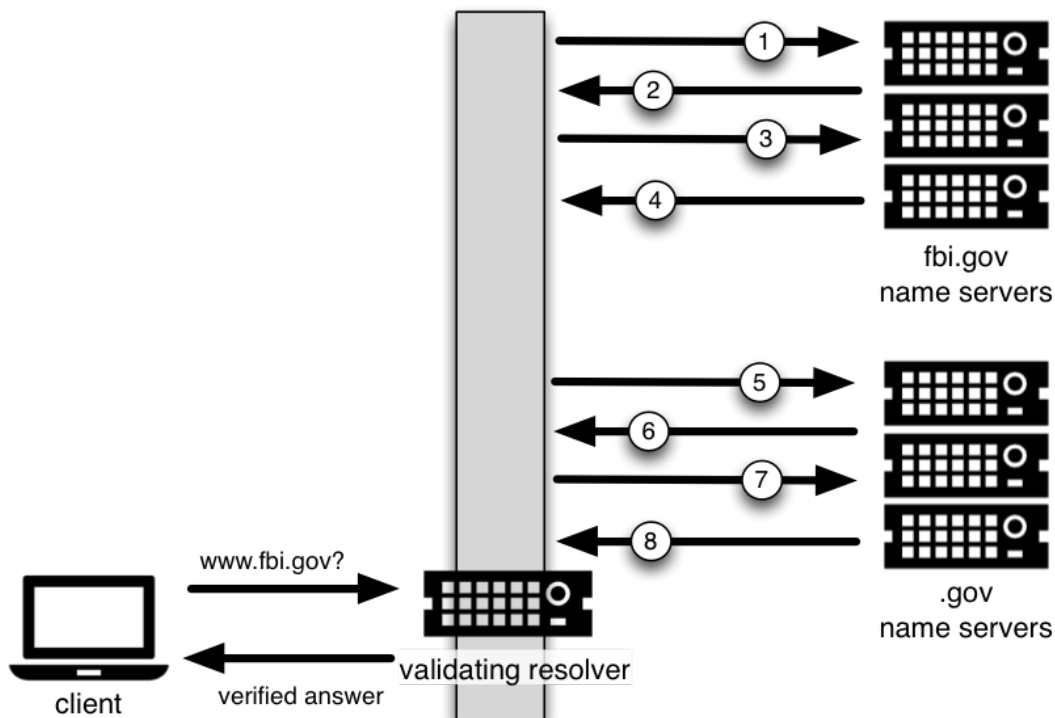


Figure 3.3: DNSSEC Validation with `.gov` Trust Anchor

1. Validating resolver queries `fbi.gov` name server for the A record of `www.fbi.gov`.
2. FBI's name server responds with answer and RRSIG.
3. Validating resolver queries FBI's name server for DNSKEY.
4. FBI's name server responds with DNSKEY and RRSIG.
5. Validating resolver queries `.gov` name server for DS record of `fbi.gov`.
6. `.gov` name server responds with DS record and RRSIG for `fbi.gov`.
7. Validating resolver queries `.gov` name server for DNSKEY.
8. `.gov` name server responds with DNSKEY and RRSIG.

This all looks very similar, except it's shorter than the 12-steps that we saw earlier. Once the validating resolver receives the DNSKEY file in #8, it recognizes that this is the manually configured trusted key (trust anchor), and never goes to the root name servers to ask for the DS record for `.gov`, or ask the root name servers for its DNSKEY.

In fact, whenever the validating resolver receives a DNSKEY, it checks to see if this is a configured trusted key, to decide whether or not it needs to continue chasing down the validation chain.

3.4.2 Trusted Keys and Managed Keys

So, as the resolver is validating, we must have at least one key (trust anchor) configured. How did it get here, and how do we maintain it?

If you followed the recommendation in Section 3.1, by setting `dnssec-validation` to `auto`, then there is nothing you need to do. BIND already includes a default key (in the file `bind.keys`), that will automatically update itself. It looks something like this:

```
managed-keys {
    . initial-key 257 3 8 "AwEAAagAIK1VZrpC6Ia7gEzahOR+9W29euxhJhVVL0yQbSEW008gcCjF
    FVQUTf6v58fLjwBd0YI0EzrAcQqBGCzh/RStIoO8g0NfnfL2MTJRkxoX
    bfDaUeVPQuYEhg37NZWAJQ9VnMVDxP/VHL496M/QZxkjf5/Efucp2gaD
    X6RS6CXpoY68LsvPVjR0ZSwzz1apAzvN9dlzEheX7ICJBBtuA6G3LQpz
    W5hOA2hzCTMjJPJ8LbqF6dsV6DoBQzgul0sGIcGOY170yQdXfZ57relS
    Qageu+ipAdTTJ25AsRTAoub8ONGcLmqrAmRLKBP1dfwhYB4N7knNnulq
    QxA+Uk1ihz0=";
};
```

You could, of course, decide to manage this key on your own by hand. First, you'll need to make sure that your `dnssec-validation` is set to `yes` rather than `auto`:

```
options {
    dnssec-validation yes;
};
```

Then, download the root key manually from a trustworthy source, such as <https://www.isc.org/bind-keys>. Finally, take the root key you manually downloaded, and put it into a `trusted-keys` statement as shown below:

```
trusted-keys {
    . 257 3 8 "AwEAAagAIK1VZrpC6Ia7gEzahOR+9W29euxhJhVVL0yQbSEW008gcCjF
    FVQUTf6v58fLjwBd0YI0EzrAcQqBGCzh/RStIoO8g0NfnfL2MTJRkxoX
    bfDaUeVPQuYEhg37NZWAJQ9VnMVDxP/VHL496M/QZxkjf5/Efucp2gaD
    X6RS6CXpoY68LsvPVjR0ZSwzz1apAzvN9dlzEheX7ICJBBtuA6G3LQpz
    W5hOA2hzCTMjJPJ8LbqF6dsV6DoBQzgul0sGIcGOY170yQdXfZ57relS
    Qageu+ipAdTTJ25AsRTAoub8ONGcLmqrAmRLKBP1dfwhYB4N7knNnulq
    QxA+Uk1ihz0=";
};
```

Looking back at the example in Section 3.4.1, if you wanted to explicitly trust `.gov` and only validate domain names under `.gov`, your `trusted-keys` statement would look something like this:

```
trusted-keys {
    gov. 257 3 8 "AQO8daaz7B+yshOfL60rytKd9aOSujgponEw3fwBMEC3
    /+e9XzHw2k+VKnBJTZ+QaVtpfUdlq9HKZiv/ck83G15T
    jYKE5jtUZ2kpEDZfVNGv6yx0smtWAXv1nCJS9ohnyOTd
    397eMo jGDHqkEC+uojEscZheEkMxzgCZwDAs+/CSU7mS
    uHtCRZn19x1zUd5Gv7yDQ3mbOUwuy30oSk0z1Q5UUPpo
    ihOugIZHFx6Jk7NLIW2wlqf9qhV4zj7TiBiJY0mCc4z
    HN8/aq2VKDhp2Na7mWzvKyTy+SYQkBQ/08LbPwj9YMc+
    uCzKL6sU/ObHv17EFhD8aPDftTHZvV9L+OZr";
};
```

As the name `trusted-keys` suggests, it is possible to have more than one key configured. You could also leverage Section 6.5.3 as part of your trust-anchor management strategy.

While `trusted-keys` and `managed-keys` appear similar, there is an important difference: `trusted-keys` are always trusted, until they are deleted from `named.conf`; `managed-keys` (specifically, `initial-key`, which is the only supported type currently) are only trusted once: for as long as it takes to load the managed key database and start the key maintenance process.



Warning

Remember, if you choose to manage the keys on your own, whenever the key changes, the configuration needs to be updated manually. Failing to do so will result in breaking nearly all DNS queries for the sub domain of the key. So if you are manually managing `.gov`, all domain names in the `.gov` space may become unresolvable; if you are manually managing the root key, you could break all DNS requests made to your recursive name server.

3.5 What's EDNS All About (And Why Should I Care)?

3.5.1 EDNS Overview

Traditional DNS responses are typically small in size (less than 512 bytes) and fit nicely into a small UDP packet. Extension mechanism for DNS (EDNS, or EDNS(0)) gives us a mechanism to send DNS data in larger packets over UDP. In order to support EDNS, both the DNS server and the network need to be properly prepared to support the larger packet size and multiple fragments.

This is important for DNSSEC, since the **+do** bit that signals DNSSEC-awareness is carried within EDNS, and DNSSEC responses are larger than traditional DNS. If DNS servers and network environment cannot support large UDP packets, it will cause retransmission over TCP, or the larger UDP responses will be discarded. Users will likely experience slow DNS resolution or unable to resolve certain names at all.

Note that EDNS applies whether or not you are validating DNSSEC because BIND has DNSSEC enabled by default.

Please see Section 2.3 for more information on what DNSSEC expects from the network environment.

3.5.2 EDNS on DNS Servers

BIND has been shipped with EDNS enabled by default for over a decade, and the UDP packet size is set to a maximum of 4096 bytes. So as the DNS administrator, there should not be any re-configuration needed. You can use **dig** to verify that your server supports EDNS and the UDP packet size it is allowing as follows:

```
$ dig @192.168.1.7 www.isc.org. A +dnssec +multiline

; <<>> DiG 9.10.0-P2 <<>> @192.168.1.7 www.isc.org. A +dnssec +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 63266
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;www.isc.org.      IN A

;; ANSWER SECTION:
www.isc.org.      23 IN A 149.20.64.69
www.isc.org.      23 IN RRSIG A 5 3 60 (
                20141029233238 20140929233238 4521 isc.org.
```

```
DX5BaGVd4KzU2AIH911Kar/UmdmkARyPhJVLR0oyPZaq
5zoobGqFI4efvzL0mcpncuUg3BSU5Q48WdBu92xinMdb
E75z1+adgEBOsFgFQR/zqM3myt/8SngWm4+TQ3XFh9eN
jqExHZZuZ268Ntlxqgf9OmKRRv8X8YigaPShuyU= )
```

```
;; Query time: 7 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Fri Oct 03 16:31:33 CST 2014
;; MSG SIZE rcvd: 223
```

There is a helpful testing tool available (provided by DNS-OARC) that you can use to verify resolver behavior regarding EDNS support: <https://www.dns-oarc.net/oarc/services/replysizetest/>

So you made sure your name servers have EDNS enabled. That should be the end of the story, right? Unfortunately, EDNS is a hop-by-hop extension to DNS. This means the use of EDNS is negotiated between each pair of hosts in a DNS resolution process, which in turn means if one of your upstream name servers (for instance, your ISP's recursive name server that you forward to) does not support EDNS, you may experience DNS lookup failures or be unable to perform DNSSEC validation.

3.5.3 Support for Large Packets on Network Equipment

Okay, so both your recursive name server and your ISP's name servers support EDNS, we are all good here, right? Not so fast. As these large packets have to traverse through the network, the network infrastructure itself must allow them to pass.

When data is physically transmitted over a network, it has to be broken down into chunks. The size of the data chunk is known as Maximum Transmission Units (MTU), and it can be different from network to network. IP fragmentation occurs when a large data packet needs to be broken down into smaller chunks so that each chunk is smaller than the MTU, and these smaller chunks need to be reassembled back into the large data packet. IP fragmentation is not necessarily a bad thing, it most likely occurs on your network today.

Some network equipment, such as firewalls, may make assumptions about DNS traffic. One of these assumptions may be how large each DNS packet is. When a firewall sees a larger DNS packet than it expects, it either rejects the large packet or drops its fragments because the firewall thinks it's an attack. This configuration probably didn't cause problems in the past since traditional DNS packets are usually pretty small in size. However, with DNSSEC, these configurations need to be updated, since DNSSEC traffic regularly exceeds 1500 bytes (a common MTU value). If the configuration is not updated to support larger DNS packet size, it will often result in the larger packets being rejected, and to the end user it looks like the queries go un-answered. Or in the case of fragmentation, only a part of the answer made it to the validating resolver, and your validating resolver may need to re-ask the question again and again, creating the appearance "DNS/network is really slow" for the end users.

And while you're updating configuration on your network equipment, make sure TCP port 53 is also allowed for DNS traffic.

3.5.4 Wait... DNS Uses TCP?

Yes. DNS uses TCP port 53 as a fallback mechanism, when it cannot use UDP to transmit data. This has always been the case even long before DNSSEC arrived at the scene. Traditional DNS relies on TCP 53 for operations such as zone transfer. The use of DNSSEC, or DNS with IPv6 records such as AAAA, increases the chance that DNS data will be transmitted on TCP.

Due to the increased packet size, DNSSEC may fall back to TCP more often than traditional (insecure) DNS. If your network is blocking or filtering TCP port 53 today, you may already experience instability with DNS resolution before deploying DNSSEC.

Chapter 4

Signing

4.1 Easy Start Guide for Signing Authoritative Zones

This section provides the minimum amount of information to setup a working DNSSEC-enabled authoritative name server. A DNSSEC-enabled zone (or "signed" zone) contains additional resource records that are used to verify the authenticity of its zone information.

To convert a traditional (insecure) DNS zone to a secure one, we need to create various additional records (DNSKEY, RRSIG, NSEC or NSEC3), and upload verifiable information (such as DS record) to the parent zone to complete the chain of trust. For more information about DNSSEC resource records, please see Section 1.4.

Note In this chapter we assume all configuration files, key files, and zone files are stored in `/etc/bind`. And most of the times we show examples of running various commands as the root user. This is arguably not the best setup, but we don't want to distract you from what's important here: learning how to sign a zone. There are many best practices for deploying a more secure BIND installation, with techniques such as jailed process and restricted user privileges, but we are not going to cover any of those in this document. We are trusting you, a responsible DNS administrator, to take the necessary precautions to secure your system.

For our examples below, we will be working with the assumption that there is an existing insecure zone `example.com` that we will be converting to a secure version.

4.1.1 Generate Keys

Everything in DNSSEC centers around keys, and we will begin by generating our own keys. In our example, we are keeping all the keys for `example.com` in its own directory, `/etc/bind/keys/example.com`.

```
# mkdir -p /etc/bind/keys/example.com
# cd /etc/bind/keys/example.com
# dnssec-keygen -a RSASHA256 -b 1024 example.com
Generating key pair...+++++ .....+++++
Kexample.com.+008+17694
# dnssec-keygen -a RSASHA256 -b 2048 -f KSK example.com
Generating key pair.....+++ .....+++
Kexample.com.+008+06817
```

This generated four key files in `/etc/bind/keys/example.com`, and the only one we care about for now is the KSK key, `Kexample.com.+008+06817.key`. Remember this file name: we will need it again shortly. Make sure these files are readable by **named**.

Refer to Section 2.1.3 for information on how you might speed up the key generation process if your random number generator has insufficient entropy.

4.1.2 Reconfigure BIND

Below is a very simple `named.conf`, in our example environment, this file is `/etc/bind/named.conf`. The lines you most likely need to add are in bold.

```
options {
    directory "/etc/bind";
    recursion no;
    minimal-responses yes;
};

zone "example.com" IN {
    type master;
    file "db/example.com.db";
    key-directory "keys/example.com";
    inline-signing yes;
    auto-dnssec maintain;
};
```

When you are done updating the configuration file, tell **named** to reload:

```
# rndc reload
server reload successful
```

4.1.3 Verification

Your zone is now signed. Before moving on to the next step of coordinating with your parent zone, let's make sure everything looks good using **delv**. What we want to do is to simulate what a validating resolver would check, by telling **delv** to use a specific trust anchor.

First of all, we need to make a copy of the key `Kexample.com.+008+06817.key` for editing:

```
# cp /etc/bind/keys/example.com/Kexample.com.+008+06817.key /tmp/example.key
```

The original key file looks like this (actual key shortened for display, and comments omitted):

```
# cat /etc/bind/keys/example.com/Kexample.com.+008+06817.key
...
example.com. IN DNSKEY 257 3 8 AwEAAcWDps...lM3NRn/G/R
```

We want to edit the copy to be the `trusted-keys` format, so that it looks like this:

```
# cat /tmp/example.key
trusted-keys {
    example.com. 257 3 8 "AwEAAcWDps...lM3NRn/G/R";
};
```

Now we can run the **delv** command and point it to using this trusted-key file to validate the answer it receives from the authoritative name server `192.168.1.13`:

```
$ delv @192.168.1.13 -a /tmp/example.key +root=example.com example.com. SOA +multiline
; fully validated
example.com.      600 IN SOA ns1.example.com. admin.example.com. (
    2014112007 ; serial
    1800       ; refresh (30 minutes)
    900        ; retry (15 minutes)
    2419200    ; expire (4 weeks)
    300        ; minimum (5 minutes)
)
example.com.      600 IN RRSIG SOA 8 2 600 (
```



```
20150107091559 20141208081559 17694 example.com.
LwG0rLOm9Q1Lu9bgIz1O+PTCwcCSs1Ev8Eqkqqd3gUJK
qo0F5Vv//axNVJFH2Lz8VLgFypD8xARWj1XQBD/9DI f6
A3ncnrFymKdKze+2ghvTUxpwqctK4RF66mhu93e33+Ir
QJVJgdHJVHudQWICA1AbIBYYzLGkK1p7JAJcgBM= )
```

4.1.4 Upload to Parent Zone

Everything is done on our name server, now we need to generate some information to be uploaded to the parent zone to complete the chain of trust. The formats and the upload methods are actually dictated by your parent zone's administrator, so contact your registrar or parent zone administrator to find out what the actual format should be, and how to deliver or upload the information to your parent zone.

What about your zone between the time you signed it and the time your parent zone accepts the upload? Well, to the rest of the world, your zone still appears to be insecure. That is because when a validating resolver attempts to validate your domain name, it will eventually come across your parent zone, and your parent zone will indicate that you are not yet signed (as far as it knows). The validating resolver will then give up attempting to validate your domain name, and fall back to the insecure DNS. Basically, before you complete this final step with your parent zone, your zone is still insecure.

Note Before uploading to your parent zone, verify that your newly signed zone has propagated to all of your name servers (usually zone transfers). If some of your name servers still have unsigned zone data while the parent tells the world it should be signed, validating resolvers around the world will not resolve your domain name.

Here are some examples of what you may upload to your parent zone, actual keys shortened for display. Note that no matter what formats may be required, the end result will be the parent zone publishing DS record(s) based on the information you upload. Again, contact your parent zone administrator(s) to find out what is the correct format for you.

1. DS Record Format: `example.com.IN DS 6817 8 1 59194A835ACD78D25D538D5F35CA043A8F3F4446`
2. DNSKEY Format: `example.com.172800 IN DNSKEY 257 3 8 (AwEAAcjGaU...zuu55If5) ;key id =06817`
3. Trusted Key Format: `"example.com." 257 3 8 "AwEAAcjGaU...zuu55If5";`

The DS record format may be generated using the `dnssec-dsfromkey` tool which is covered in Section 4.4.1. For more details and examples on how to work with your parent zone, please see Section 4.4

4.1.5 So... What Now?

Congratulations, your zone is signed, your slave servers have received the new zone data, and the parent zone has accepted your upload and published your DS record. Your zone is now officially DNSSEC-enabled. What happens next? Well, there are a few maintenance tasks you need to do on a regular basis, which you can find in Section 4.6. As for updating your zone file, you can continue to update them the same way you have been prior to signing your zone, the normal work flow of editing zone file and using the `rndc` command to reload the zone still works the same, and although you are editing the unsigned version of the zone, BIND will generate the signed version automatically.

Curious as to what all these commands did to your zone file? Read on to Section 4.1.6 and find out. If you are interested in how you can roll this out to your existing master and slave name servers, check out Section 7.1 in Chapter 7.

4.1.6 Your Zone, Before and After DNSSEC

In the previous section Section 4.1, we provided the minimal amount of information to essentially convert a traditional DNS zone into a DNSSEC-enabled zone. This is what the zone looked like before we started:

```
$ dig @192.168.1.13 example.com. AXFR +multiline +onesoa

; <<>> DiG 9.10.1 <<>> @192.168.1.13 example.com. AXFR +multiline +onesoa
; (1 server found)
;; global options: +cmd
example.com.      600 IN SOA ns1.example.com. admin.example.com. (
    2014102100 ; serial
    1800       ; refresh (30 minutes)
    900        ; retry (15 minutes)
    2419200   ; expire (4 weeks)
    300       ; minimum (5 minutes)
)
example.com.      600 IN NS ns1.example.com.
ftp.example.com.  600 IN A 192.168.1.200
ns1.example.com.  600 IN A 192.168.1.1
web.example.com.  600 IN CNAME www.example.com.
www.example.com.  600 IN A 192.168.1.100
```

Below shows the test zone `example.com` after we have reloaded the server configuration. As you can see, the zone grew in size, and the number of records multiplied:

```
# dig @192.168.1.13 example.com. AXFR +multiline +onesoa

; <<>> DiG 9.10.1 <<>> @192.168.1.13 example.com. AXFR +multiline +onesoa
; (1 server found)
;; global options: +cmd
example.com.      600 IN SOA ns1.example.com. admin.example.com. (
    2014102104 ; serial
    1800       ; refresh (30 minutes)
    900        ; retry (15 minutes)
    2419200   ; expire (4 weeks)
    300       ; minimum (5 minutes)
)
example.com.      300 IN RRSIG NSEC 8 2 300 (
    20141126120153 20141027112239 60798 example.com.
    U1je2BEXY7S/u4An8sKEp2Cb8b90+9mNBoFb9nXgkTvC
    cJkT/4OIUIa07EbsIASmlzjDwJEKSO/nPDFg3y6tsR0m
    GT7H7AYAouwBhUlIrDrGQpxqRLJXA1kPkxboX4M2JGx4
    rBcBsN/7lG5CGbjtLW3PMnJRELE5fBJBLZM1KLo= )
example.com.      300 IN NSEC ftp.example.com. NS SOA RRSIG NSEC DNSKEY TYPE65534
example.com.      600 IN RRSIG NS 8 2 600 (
    20141126115318 20141027112239 60798 example.com.
    TGClgAksxHHDoltjJ8ETJ9qdr5MzFjaA05vEllSNWBFe7
    O71Payf8o4MISHNeUoZpu7Ys4zBiTRLqUaLeI0oSenzM
    r2QJkrfUHl1k8D/wCWApeUhm1GxLlKMsEfgd3D5nK52
    6LAVNihx5BKClYxuYZXhhBMYPm7bIlUXFvx13uE= )
example.com.      600 IN RRSIG SOA 8 2 600 (
    20141126122239 20141027112239 60798 example.com.
    OntLprFpg9bKWs5ArGvpZq8uFa9QPEfl6bOmfIwbNnEp
    9PgBS97icRs27JDgIXGM4dWxXiPVX6sZ8jFulSp2mi2v
    4dhtqMfb4S/rfxSmCphIz5PA+bngKSr5fhRI6tTM8fW1
    15V3zfKCPFracE0CgdcWIt0xViEs56Axj942f5w= )
example.com.      0 IN RRSIG TYPE65534 8 2 0 (
    20141126120153 20141027112239 60798 example.com.
    Sv+7vfHXz4zLlSEuWs5Bt5boqSDVCK2GcOgAzVXnW9/0
    zgk3v8VwSXRTFnwdsVl3SK6YsM9S/HNXk32AQs2PjXf4
    2/cmv6VCOGSVGin/L5k4KWjpy0ESV2iY0hIIyXWqUdB
    ZUCu1+2GhHiA5+gKODn/8sg6x04g+lZe6wuhU44= )
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
    20141126122239 20141027112239 45319 example.com.
    F2vrBJqvloP5yNE6X1+b1XJ4gY6DlH1oyJz6pSULmGgQ
```

```

FFztyrCJoV4MI6TeFRXJKLTepewjctzKlaRKJHZRElcm
QaClWr6RdZ2Uc1fFOBgyocrDZtGWdTVg7XCiLJ/ubrDH
7et9rSexmKEwI44T4Vf7w3e+kUQSZMBGfA9Z5Q8WnnBW
IDn4QFqRx0B02bVLV4giJbVp649ukIAPVoHSpJvhrWq8
eSSdP2ojymzr4gAuQZzmZUJnxp7Q3ktuQgIokhLG4FDH
/O9qnhnOkUZFsxbBaDYVn5Tl1lKkmn3LH98JQE0z8a7D
WfgAPAFy0rKHkvyOA8FlimDW090YeJS00A== )
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
20141126122239 20141027112239 60798 example.com.
PoUdallJ00ZF3iFmRoGoKh/iLAJVn52ABVEVku9LK34C
oh4Y29rgK4jd3DwxU7zDLadVs0Fo3JFHp2jdYCEm03BS
XE6d0cijZP5aFt8rO/grO/qbkdv+ScEi1kOhCKcvt3Kg
1mXhiZt4Jx3LeisUpD9mGa+m9ehcPY1AHPKoZMg= )
example.com.      0 IN TYPE65534 \# 5 ( 08B1070001 )
example.com.      0 IN TYPE65534 \# 5 ( 08ED7E0001 )
example.com.      600 IN DNSKEY 256 3 8 (
AwEAAfbc/0ESummlmPVkm025PfHKHNYW62yx0wyLN5LE
4Difn6FzIVSKSGdM0dq+z6vFGxzzjPDz7QZdeC6ttIUA
Bo4tG7dDrsWK+tG5cm4vuylsEVbnnW5i+gFG/02+RYmZ
ZT9AobXB5bvjfXl9SDBgpBluB35WUCAnK9WkRRUS08lf
) ; ZSK; alg = RSASHA256; key id = 60798
example.com.      600 IN DNSKEY 257 3 8 (
AwEAAb4N53kPbdRTAwvJT8OYVeVhQIldwppMy7KBJ+8k
Uggx2PU3yP/qlq4Zj10MMmqRiJhD/S+z9cJLNTZ9tHz1
7aZQjFyGAYuU3DGW16xfMolcIn+c8TpPCzBOFHxk6jvO
VLlz+Wgyi1ES+t29FjYyV5cVNRPMxXLRj1HFd01DzX3N
dmcUoZ+VVJCvaML9+6UpL/6jitNsoU8JHnxT9B2CGKcw
N7VaK419Ida2BqY3/4UVqWzhj03/M5LK6cn1pEQbQMTY
R0TNJURBKdK8bH663h98i23tVX0/85IsCVBL4Dd2boa3
/7HPp7uZN1AjDvcRsOh1mqixwUGmVm1EskDIMy8=
) ; KSK; alg = RSASHA256; key id = 45319
example.com.      600 IN NS ns1.example.com.
ftp.example.com.  600 IN RRSIG A 8 3 600 (
20141126115318 20141027112239 60798 example.com.
PNATas8HPjC6tm7Ldfk7P61lRVSQW085P9lDw8qwITc
TghDdAXHJBkCTxADRWPYQZbGI5fNcxWLkf/HRbKREzQs
eyztZj20BnBnnK0t8SWLyM4+OMSGH17z4ZvBmPO/Wgju
iSm98reH7J87yRlUMQHdPSwAP6q5tZeJbwpSkao= )
ftp.example.com.  300 IN RRSIG NSEC 8 3 300 (
20141126115318 20141027112239 60798 example.com.
ohuvpkqlNR+TYOG79JizEV+SFjVbUKSB3hEfh0JszCIr
bcdPKTS0GgQS62N4ISwfvb735oQzc8pI5R+f+8MMn5c9
/yRnArTpPJOKDnyIiMaMLBU5sEp5wI+OLsLFMkLef4iK
gn+j608Qnoo7dXtWWQPv85FyaT9j3C+EV6rglCk= )
ftp.example.com.  300 IN NSEC ns1.example.com. A RRSIG NSEC
ftp.example.com.  600 IN A 192.168.1.200
ns1.example.com.  600 IN RRSIG A 8 3 600 (
20141126115318 20141027112239 60798 example.com.
zlrnlU9oVnIhwXQey3Zo6ENYmWXtf97RiZng3u6Vke0n
yUC5S3sseZgD8Vc+uJsKVAvaFM/k7NgS+P3pe5gN01Ln
+geKEMZ3pCHpzHG4UgJj4Xw/iuWOVAFURD26GNnPPpDK
RNYPWihA/9FF18vI3V9evNcHGxN+7rOOXF8Qbss= )
ns1.example.com.  300 IN RRSIG NSEC 8 3 300 (
20141126115318 20141027112239 60798 example.com.
chH7c6aW6Qk14d4LB866jva9OcH52pJkeHpdslVmUbJ2
OkJQNhbpOekSjsroxDU4av3/Y49DtG0lqoISvMMqWE5X
5sKmn+CMKfMmi22ui+RsPXyDqjKx+J0si/7kCx7DFJz
jF0a6KcsTWNV2QfZjQ50RyrSBB0kw6aqdUC3oF8= )
ns1.example.com.  300 IN NSEC web.example.com. A RRSIG NSEC
ns1.example.com.  600 IN A 192.168.1.1
web.example.com.  600 IN RRSIG CNAME 8 3 600 (
20141126115318 20141027112239 60798 example.com.

```

```

VgHPC0W1lA1HY56xMF3j8FI7MsxDhLJJBDMSVSQpQkA
z8lkfInYQL4YpAt25WrH7xt4pCmjVl3kUcDGAWWuSiQy
nh2O4CoY/tSqQ7Sr9UnE5SIvpW5yws8iROOHvJYuKAmu
8B3gYw1gp9xCMS4iBOIXTTq1KV2OHwo0cPXVyfI= )
web.example.com. 300 IN RRSIG NSEC 8 3 300 (
20141126115318 20141027112239 60798 example.com.
zh+9fnIQlfpvc0oFRnNIPJoL/S5RQF4HuGwmLxoCXRvP
FCQoZbzNB0IVvaA+0XEslcHq3v4+0j9EeKKQw7xORaGu
TrzcoRGbjukZlpvnVlxYkjCiEKVbkXd5yEMnbgolFkhH
NjFlozSyzyypY0TY0UVRPXWhZ2shASae/ImqFiug= )
web.example.com. 300 IN NSEC www.example.com. CNAME RRSIG NSEC
web.example.com. 600 IN CNAME www.example.com.
www.example.com. 600 IN RRSIG A 8 3 600 (
20141126120153 20141027112239 60798 example.com.
TLkMRs7I3QW8HRS3trNG307Y0/drO4k8uG8Bv8856qPo
33rbEajv49XBBQ+w8jynfKwxMNDfJhpsG0gxo7+0m75Y
HRBZ9Xe3EAMNt2/iu0CL6NHyn99qF9TPNbYnD2Zw077L
GSDOua1KFMxDqFU7rU0+YlKRGbSk1rqAJBqKEH4= )
www.example.com. 300 IN RRSIG NSEC 8 3 300 (
20141126120153 20141027112239 60798 example.com.
GSyFVBgMROqDuoSJoDegX3EkdqA90skBTvi9tkPxoQPo
i8LlAd+wRmcXpYKE3vf/y0WpOOVQXbasyfFk8wGzFsde
fqpFsscbz30YgnkwYcTvGXZF9802ytiC5txli9uxUqjc
lGL+SDl0woYhecwJD63fTDIMszlVR/eptIL9dLc= )
www.example.com. 300 IN NSEC example.com. A RRSIG NSEC
www.example.com. 600 IN A 192.168.1.100

```

But this is a really messy way to tell if your zone is properly setup with DNSSEC. Fortunately, there are tools to help us with that. Read on to Section 4.2 to learn more.

4.2 How To Test Authoritative Zones (So You Think You Are Signed)

So we've generated some keys, run some commands, and uploaded some data to our parent zone. How do we know our zone is signed correctly? Here are a few ways to check.

4.2.1 Look for Key Data in Your Zone

One of the ways to see if your zone is signed, is to check for the presence of DNSKEY record types. In our example, we created two keys, and we expect to see both keys returned when we query for them.

```

$ dig @192.168.1.13 example.com. DNSKEY +multiline +noall +answer

; <<>> DiG 9.10.1 <<>> @192.168.1.13 example.com. DNSKEY +multiline +noall +answer
; (1 server found)
;; global options: +cmd
example.com. 300 IN DNSKEY 256 3 8 (
AwEAAclob7q+ccvDwaTVuMM2ddGIynWyMwaZlhFrU6cC
0qknWoPpkq0gIwTrYf3DJY+eIKPVHxrM+o2AoRIVhubG
jfv1bT5wTYrawZstS84ejCQ+ehA+8DxKyeWUEzW0ZMBe
OhyeG0cuQVK/p6Z1E096JLuODjgbabLspequkw4M+HT7
) ; ZSK; alg = RSASHA256; key id = 57009
example.com. 300 IN DNSKEY 257 3 8 (
AwEAAQ2ctHx8VmryndiOgpchXPdj3NwxMeUvAre6uYI
5KELlFJUghThrz+/CzEc8CXG8wwQ4ZvAey0FGV2nJAFc
ENMxoriCz0oSiQQxryNhACd3RnE2/D7G+Shw1OM6w53E
wUJ/lsgu5UevSxFC+eA3fKeL3TWR44PH4iJQp9QmfW5v
7qG8Sic/HQvBGBdOGfFtHAL0a4jDPBi57imS4YsHcUYD
9bsWmhYWSHJKZ66+JnTiMS0nQM69YwBF43QfDKurs5R6

```

```
qPUDiBlaMCzSxmIaBU6fsI1Mu/yIU8w1ewy26a42rUTU
rPBC30a/zf9VQ8kpUrMZgJ7LEAA4xmR+qwWDh6U=
) ; KSK; alg = RSASHA256; key id = 28267
```

4.2.2 Look for Signatures in Your Zone

Another way to see if your zone data is signed is to check for the presence of signature. With DNSSEC, every record¹ now comes with at least one corresponding signature known as RRSIG.

```
$ dig @192.168.1.13 example.com. SOA +dnssec +multiline

; <<>> DiG 9.10.1 <<>> @192.168.1.13 example.com. SOA +dnssec +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 31466
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 4096
;; QUESTION SECTION:
;example.com.      IN SOA

;; ANSWER SECTION:
example.com.      300 IN SOA ns1.example.com. dnsadmin.example.com. (
    2014102111 ; serial
    10800      ; refresh (3 hours)
    1080       ; retry (18 minutes)
    2419200   ; expire (4 weeks)
    900       ; minimum (15 minutes)
)
example.com.      300 IN RRSIG SOA 8 2 300 (
    20141021122105 20141022112105 57009 example.com.
    NqPGNLkUs40Lg/qQ7Fv+bgYcWVB4s9PsHQOK6p9ZWWk3
    36z2Qz2WjM+Q19S1VBAPux9ji jvcRcjGb6KREuxER9uX
    wdVeiGx9a4X+PaO3qTqdkixuGS2XkK1kBm1CgwhVHTYV
    /nxVPrckU4/mpeUoFVjMnT49JkVJmgck63esPEU= )
```

The serial number was automatically incremented from the old, unsigned version. `named` keeps track of the serial number of the signed version of the zone independently of the unsigned version. If the unsigned zone is updated with a new serial number that's higher than the one in the signed copy, then the signed copy will be increased to match it, but otherwise the two are kept separate.

4.2.3 Examine the Zone File

Our original zone file `example.com.db` remains untouched, `named` has generated 3 additional files automatically for us (shown below). The signed DNS data is stored in `example.com.db.signed` and in the associated journal file.

```
# cd /etc/bind/db
# ls
example.com.db  example.com.db.jbk  example.com.db.signed  example.com.db.signed.jnl
```

A quick description of each of the files:

- `.jbk`: transient file used by `named`

¹ Well, almost every record. NS records and glue records for delegations do not have RRSIG records like everyone else. If you do not have any delegations, then yes, every record in your zone will be signed and comes with its own RRSIG.

- `.signed`: signed version of the zone in raw format
- `.signed.jnl`: journal file for the signed version of the zone

These files are stored in raw (binary) format for faster loading. You could reveal the human-readable version by using **named-compilezone** as shown below. In the example below, we are running the command on the raw format zone `example.com.db.signed` to produce a text version of the zone `example.com.text`:

```
# named-compilezone -f raw -F text -o example.com.text example.com example.com.db.signed
zone example.com/IN: loaded serial 2014112008 (DNSSEC signed)
dump zone to example.com.text...done
OK
```

4.2.4 Check the Parent

Although this is not strictly related to whether or not the zone is signed, a critical part of DNSSEC is the trust relationship between the parent and child. Just because we, the child, have all the correctly signed records in our zone doesn't mean it can be fully validated by a validating resolver, unless our parent's data agrees with us. To check if our upload to the parent is successful, ask the parent name server for the DS record of our zone, and we should get back the DS record(s) containing the information we uploaded in Section 4.1.4:

```
$ dig example.com. DS

; <<>> DiG 9.10.1 <<>> example.com. DS
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 49949
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;example.com.      IN  DS

;; ANSWER SECTION:
example.com.  61179 IN  DS  28267 8 1 66D47CE4B4F551BE5EDA43AC5F3109E8C98E2FAE
example.com.  61179 IN  DS  28267 8 2 71 ←
                D9335416B7132519190A95685E18CBF478DCF4CA98867062777938F8FEAB89
```

4.2.5 External Testing Tools

The easiest ways to see if your domain name is fully secured is to use one of these excellent online tools.

1. Verisign Labs DNSSEC Debugger: <http://dnssec-debugger.verisignlabs.com/>
2. DNSViz: <http://dnsviz.net/>

4.2.5.1 Verisign DNSSEC Debugger

URL: <http://dnssec-debugger.verisignlabs.com/>

This tool shows a nice summary of checks performed on your domain name, and you can expand to view more details for each of the items checked to get a detailed report.

.	<ul style="list-style-type: none"> ✔ Found 2 DNSKEY records for . ✔ DS=19036/SHA-1 verifies DNSKEY=19036/SEP ✔ Found 1 RRSIGs over DNSKEY RRset ✔ RRSIG=19036 and DNSKEY=19036/SEP verifies the DNSKEY RRset
com	<ul style="list-style-type: none"> ✔ Found 1 DS records for com in the . zone ✔ Found 1 RRSIGs over DS RRset ✔ RRSIG=22603 and DNSKEY=22603 verifies the DS RRset ✔ Found 2 DNSKEY records for com ✔ DS=30909/SHA-256 verifies DNSKEY=30909/SEP ✔ Found 1 RRSIGs over DNSKEY RRset ✔ RRSIG=30909 and DNSKEY=30909/SEP verifies the DNSKEY RRset
example.com	<ul style="list-style-type: none"> ✔ Found 2 DS records for example.com in the com zone ✔ Found 1 RRSIGs over DS RRset ✔ RRSIG=48758 and DNSKEY=48758 verifies the DS RRset ✔ Found 3 DNSKEY records for example.com ✔ DS=31589/SHA-1 verifies DNSKEY=31589/SEP ✔ Found 1 RRSIGs over DNSKEY RRset ✔ RRSIG=31589 and DNSKEY=31589/SEP verifies the DNSKEY RRset ✔ example.com A RR has value 93.184.216.119 ✔ Found 1 RRSIGs over A RRset ✔ RRSIG=14998 and DNSKEY=14998 verifies the A RRset

Figure 4.1: Verisign DNSSEC Debugger

4.2.5.2 DNSViz

URL: <http://dnsviz.net/>

DNSViz provides a visual analysis of the DNSSEC authentication chain for a domain name and its resolution path in the DNS namespace.

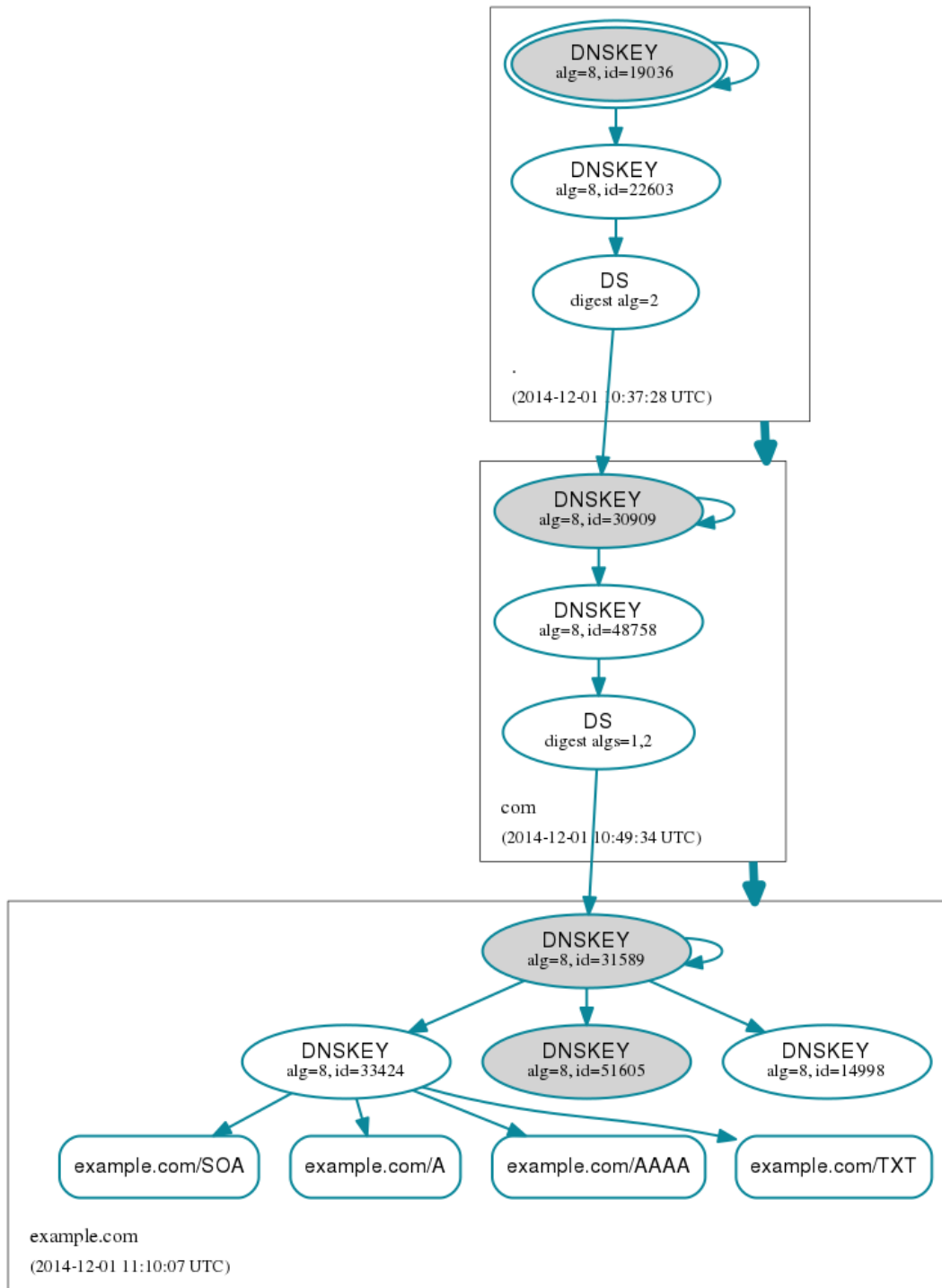


Figure 4.2: DNSViz

4.3 Signing Easy Start Explained

4.3.1 Generate Keys Explained

In Section 4.1.1, we generated two pairs of keys: a pair of Zone Signing Keys (ZSK) and a pair of Key Signing Keys (KSK). To quickly summarize, ZSKs sign the bulk of the zone, but KSKs only sign the DNSKEYs. This makes ZSKs easier to change (since you can do so without updating the parent). We generated keys by running these commands:

```
# cd /etc/bind/keys/example.com
# dnssec-keygen -a RSASHA256 -b 1024 example.com
Generating key pair...+++++ .....+++++
Kexample.com.+008+17694
# dnssec-keygen -a RSASHA256 -b 2048 -f KSK example.com
Generating key pair.....+++ .....+++
Kexample.com.+008+06817
```

With these commands, we generated NSEC3-compatible key pairs (see Section 6.2.2 to learn more about NSEC3). In the end, four key files were created in `/etc/bind/keys/example.com`:

- `Kexample.com.+008+06817.key`
- `Kexample.com.+008+06817.private`
- `Kexample.com.+008+17694.key`
- `Kexample.com.+008+17694.private`

The two files ending in `.private` need to be kept, well, private. These are your private keys, guard them carefully. You should at the very least protect them via file permission settings. Please see Section 6.3 for more information about how to store your keys.

The two files ending in `.key` are your public keys. One is the zone-signing key (ZSK), and one is the key-signing Key (KSK). We can tell which is which by looking at the actual file contents (actual keys shortened for display):

```
# cat Kexample.com.+008+06817.key
; This is a key-signing key, keyid 6817, for example.com.
; Created: 20141120094612 (Thu Nov 20 17:46:12 2014)
; Publish: 20141120094612 (Thu Nov 20 17:46:12 2014)
; Activate: 20141120094612 (Thu Nov 20 17:46:12 2014)
example.com. IN DNSKEY 257 3 8 AwEAAcWDps...lM3NRn/G/R
# cat Kexample.com.+008+17694.key
; This is a zone-signing key, keyid 17694, for example.com.
; Created: 20141120094536 (Thu Nov 20 17:45:36 2014)
; Publish: 20141120094536 (Thu Nov 20 17:45:36 2014)
; Activate: 20141120094536 (Thu Nov 20 17:45:36 2014)
example.com. IN DNSKEY 256 3 8 AwEAAcjGaU...zuu55If5
```

The first line of each file tell us what type of key it is. Also, by looking at the actual DNSKEY record, we could tell them apart: 256 is ZSK, and 257 is KSK.

So, this is a ZSK:

```
# cat Kexample.com.+008+17694.key
...
example.com. IN DNSKEY 256 3 8 AwEAAcjGaU...zuu55If5
```

And this is a KSK:

```
# cat Kexample.com.+008+06817.key
...
example.com. IN DNSKEY 257 3 8 AwEAAcWDps...lM3NRn/G/R
```

The parameters we showed in the example, algorithm of RSASHA256, key length of 1024 and 2048, and the use of NSEC3 are just suggestions, you need to evaluate what values work best for your environment. To learn more about key generation, different algorithm choices, and key sizes, see Section 6.1.

The table below summarizes the usage and frequency of use for each of the keys.

Key	Usage	Frequency of Use
ZSK Private	Used by authoritative server to create RRSIG for zone data	Used somewhat frequently depending on the zone, whenever authoritative zone data changes or re-signing is needed
ZSK Public	Used by recursive server to validate zone data RRset	Used very frequently, whenever recursive server validates a response
KSK Private	Used by authoritative server to create RRSIG for ZSK and KSK Public (DNSKEY)	Very infrequently, whenever ZSK's or KSK's change (every year or every five years in our examples)
KSK Public	Used by recursive server to validate DNSKEY RRset	Used very frequently, whenever recursive server validates a DNSKEY RRset

Table 4.1: ZSK KSK Comparison

4.3.2 Reconfigure BIND Explained

In Section 4.1.2, we highlighted a few lines, let's explain what each one of them does.

```
options {
    directory "/etc/bind";
    recursion no;
    minimal-responses yes;
    dnssec-enable yes;
};

zone "example.com" IN {
    type master;
    file "db/example.com.db";
    key-directory "keys/example.com";
    inline-signing yes;
    auto-dnssec maintain;
};
```

4.3.2.1 dnssec-enable

DNSSEC support is enabled in **named** by default. If the `dnssec-enable` option is turned off, **named** will be unable to serve signed zones.

4.3.2.2 key-directory

```
zone "example.com" IN {
    key-directory "keys/example.com";
};
```

This specifies where **named** should look for public and private DNSSEC key files. The default is **named**'s working directory. In our example, we organized keys based on zone names, and placed all keys for `example.com` under one directory `/etc/bind/keys/example.com`.

4.3.2.3 inline-signing

```
zone "example.com" IN {  
    inline-signing yes;  
};
```

This option is disabled by default. When enabled, BIND converts traditional (insecure) zone data to signed (secure) data automatically and transparently, using keys found in `key-directory`.

This feature alleviates the burden of re-signing zone data put on DNSSEC zone administrators. As the zone administrator, you can continue to manually maintain the unsigned version of the zone just like before, and **named** automatically creates an internal copy of the zone, signs it on the fly, and increments the serial number for the signed zone. The unsigned version of the zone is left intact. With this feature enabled, whenever **named** detects that your zone needs to be signed, either due to a new record being added, removed, or signature expiration, it will automatically re-sign the zone data.

Inline signing can also be used as a strategy to aid DNSSEC deployment in the case where the master zone cannot be easily modified. To learn more about inline signing, please see Section 6.5.2.

4.3.2.4 auto-dnssec

```
zone "example.com" IN {  
    auto-dnssec maintain;  
};
```

With keys, comes the burden of key management. `auto-dnssec` provides varying levels of automatic key management. There are three possible settings:

1. *off*: this is the default, key management is done manually
2. *allow*: permits keys to be updated and the zone fully re-signed whenever the user issues the command **rndc sign [zone-name]**.
3. *maintain*: includes what "allow" has, but also automatically adjusts the zone's DNSSEC keys on schedule, according to the key's timing metadata.

We have opted for the "maintain" mode in our example, which provides the most automated key management. With this option enabled, BIND will periodically check to see if new keys are available, or old keys need to be retired, and automatically add or remove the appropriate DNSKEY records from the zone. The frequency of the check can be controlled via `dnssec-loadkeys-interval`, default is 60 minutes (1 hour).

Note

auto-dnssec is a feature to automate many of the key management tasks, which we discuss in more detail in Section 6.4.2.1, to cover topics such as manual signing and key timing metadata.

4.4 Working with Parent Zone

As we mentioned in Section 4.1.4, the format of the information you upload to your parent zone is dictated by your parent zone administrator, and the three main formats are:

1. DS Record Format
2. DNSKEY Format
3. Trusted Key Format

ccTLDs typically maintain their own lists of registrars and should have a list of which of those support DNSSEC. ICANN maintains a list of accredited registrars who support DNSSEC for gTLDs:

<https://www.icann.org/resources/pages/deployment-2012-02-25-en>

Next, we will take a look at how to get each of the three formats from your existing data.

4.4.1 DS Record Format

Below is an example of generating DS record formats from the KSK we created earlier (`Kexample.com.+008+28267.key`) using two different secure hashing algorithms (SHA-1 and SHA-256, respectively):

```
# cd /etc/bind/keys/example.com
# dnssec-dsfromkey -a SHA-1 Kexample.com.+008+06817.key
example.com. IN DS 6817 8 1 59194A835ACD78D25D538D5F35CA043A8F3F4446
# dnssec-dsfromkey -a SHA-256 Kexample.com.+008+06817.key
example.com. IN DS 6817 8 2 2 ←
    A5F1DF55D5E64CBD7BCFE1EFA6E9586AF335FA56A2473296E975B89AFD31E11
```

Some registrars may ask you to manually specify the types of algorithm and digest used. In the first example, 8 represents the algorithm used, and 1 represents the digest type (SHA-1); in the second example, 8 is the algorithm, and 2 is the digest type (SHA-256). The key tag or key ID is 28267.

Alternatively, you could generate it from the DNSKEY records like this:

```
$ dig @192.168.1.13 example.com. DNSKEY | dnssec-dsfromkey -f - example.com
example.com. IN DS 6817 8 1 59194A835ACD78D25D538D5F35CA043A8F3F4446
example.com. IN DS 6817 8 2 2 ←
    A5F1DF55D5E64CBD7BCFE1EFA6E9586AF335FA56A2473296E975B89AFD31E11
```

4.4.2 DNSKEY Format

Below is an example of a different key ID (16027) using DNSKEY format (actual key shortened for display):

```
example.com. 172800 IN DNSKEY 257 3 8 (AwEAAbGOpf...AX2QHMY8=) ; key id = 16027
```

The key itself is easy to find (it's kind of hard to miss that big long base64 string) in the file. Remember, we want the KSK, so look for the presence of 257.

```
# cd /etc/bind/keys/example.com
# cat Kexample.com.+008+06817.key
; This is a key-signing key, keyid 6817, for example.com.
; Created: 20141120094612 (Thu Nov 20 17:46:12 2014)
; Publish: 20141120094612 (Thu Nov 20 17:46:12 2014)
; Activate: 20141120094612 (Thu Nov 20 17:46:12 2014)
example.com. IN DNSKEY 257 3 8 AwEAAcWDps...lM3NRn/G/R
```

Some registrars may ask you to manually specify the type of algorithm used and the key tag number. In our example above, the chosen algorithm is 8, and the key ID is 6817.

4.4.3 Trusted Key Format

This format is very similar to the DNSKEY format, the differences are mostly cosmetic. Below is an example of `trusted-keys` format (again, actual key shortened for display):

```
"example.com." 257 3 8 "AwEAAbGOpf...9AX2QHMY8=";
```

4.5 Using NSEC3

After reloading the server configuration file, additional DNSSEC resource records are auto-magically generated. By default, BIND will generate NSEC records. If you wish to use NSEC3 instead, please follow the additional steps described in Section 7.3.1. To learn more about the difference between NSEC and NSEC3, please see Section 6.2.

4.6 Maintenance Tasks

Zone data is signed and the parent zone has published your DS records — at this point your zone is officially secure. When other validating resolvers lookup information in your zone, they are able to follow the 12-step process as described in [Section 3.3.2](#) and verify the authenticity and integrity of the answers.

There is not that much left for you to do, as the DNS administrator, at an ongoing basis. Whenever you update your zone, BIND will automatically resign your zone with new RRSIG and NSEC or NSEC3 records, and even increment the serial number for you.

That leaves DNSKEY records. Just like passwords and underwear, keys should be changed periodically. There are arguments for and against rolling keys, which are discussed elsewhere. If you decide to change your keys, we recommend changing your ZSK pair annually, and your KSK pair every five years. This is also known as a key rollover.

Why annually for the ZSK? That's just a convenient length of time that probably coincides with your domain name registration/renewal cycle; And every five years? That's also a generic length of time, one which happens to be the same as the root key's rollover schedule. Some people feel or have the need to do it more frequently, while some argue that there is no need for key rolling. We discuss those considerations in [Section 6.4.1](#). But assuming you do not have special security requirements, nor host a high-valued zone, rotating your ZSK every year and KSK every five years should suffice. We also provide detailed step-by-step examples of each rollover in [Section 7.2](#).

4.6.1 ZSK Rollover

Assuming you are rolling your ZSK every year on January 1st, below is the timeline of what should happen:

1. December 1st, a month before rollover date:
 - Change timer on current ZSK
 - Generate new ZSK
 - Publish new DNSKEY
2. January 1st, day of rollover:
 - New ZSK used to replace RRSIGs for the bulk of the zone
3. February 1st, a month after rollover date:
 - Remove old DNSKEY from zone
 - DNSKEY signatures made with KSK are changed

This may look like a lot of work, but with `inline-signing` and `auto-dnssec`, most of these are automated. The only things that need to be done manually are just the first two items:

- Change timer on current ZSK
- Generate new ZSK

For an example of how to execute a ZSK rollover, please see [Section 7.2.1](#).

4.6.2 KSK Rollover

KSK rollover is very similar to ZSK, with the addition of interacting with the parent zone. In fact, as you can see below, the timeline looks nearly identical to the ZSK rollover, with the addition of interaction with parent zone:

1. December 1st, a month before rollover date:
 - Change timer on current KSK

- Generate new KSK and DS records
 - Upload new DS records to parent zone
2. January 1st, day of rollover:
 - Use new KSK to sign all DNSKEY RRset, this generates new RRSIGs
 - Add new RRSIGs to the zone
 - Remove old RRSIGs from zone
 3. February 1st, a month after rollover date:
 - Remove old KSK DNSKEY from zone
 - Remove old DS records from parent zone

Unfortunately, as of this writing, KSK rollover involves a lot of manual steps. As described above, the only automated tasks are the ones that occur on the day of the rollover (January 1st), everything else needs to be done manually. To see an example of how to perform a KSK rollover, please see [Section 7.2.2](#).

Chapter 5

Basic Troubleshooting

In this chapter, we are going to cover some basic troubleshooting techniques, common DNSSEC symptoms, and their causes and solutions. This is not a comprehensive "how to troubleshoot any DNS or DNSSEC problem" guide, because that in and of itself could easily be an entire book.

5.1 Query Path

The first step to your DNS or DNSSEC troubleshooting should be to determine the query path. This is not a DNSSEC-specific troubleshooting technique. Whenever you are working with a DNS-related issue, it is always a good idea to determine the exact query path to identify the origin of the problem.

End clients, such as laptop computers or mobile phones, are configured to talk to a recursive name server, and the recursive name server may in turn forward on to more recursive name servers, before arriving at the authoritative name server. The giveaway is the presence of the Authoritative Answer (aa) flag: when present, we know we are talking to the authoritative server; when missing, we are talking to the recursive server. The example below shows an answer without the Authoritative Answer flag:

```
$ dig www.example.com. A

; <<>> DiG 9.10.1 <<>> www.example.com. A
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 41006
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.com.    IN    A

;; ANSWER SECTION:
www.example.com.    600 IN    A 192.168.1.100

;; Query time: 21 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Mon Nov 03 19:54:37 CST 2014
;; MSG SIZE rcvd: 60
```

Not only do we not see the aa flag, we see the presence of an ra flag, which represents Recursion Available. This indicates that the server we are talking to (192.168.1.7 in this example) is a recursive name server. And although we were able to get an answer for `www.example.com`, the answer came from somewhere else.

The example below shows when we query the authoritative server directly and see the presence of the aa flag:

```
$ dig @192.168.1.13 www.example.com. A
; <<>> DiG 9.10.1 <<>> @192.168.1.13 www.example.com. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35962
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available
...
```

The presence of the `aa` flag tells us that we are now talking to the authoritative name server for `example.com`, and this is not a cached answer it obtained from some other name server, it served this answer to us right from its own database. In fact, if you look closely, the `ra` flag is not present, which means this name server is not configured to perform recursion (at least not for this client), so it could not have queried another name server to get cached results anyway.

5.2 Visible Symptoms

After you have figured out the query path, the next thing to do is to determine whether or not the problem is actually related to DNSSEC validation. You can use the `+cd` flag in `dig` to disable validation, as described in Section 3.2.5.

When there is indeed a DNSSEC validation problem, the visible symptoms, unfortunately, are very limited. With DNSSEC validation enabled, if a DNS response is not fully validated, it will result in a generic `SERVFAIL` message, as shown below when querying against a recursive name server 192.168.1.7:

```
$ dig @192.168.1.7 www.isc.org. A
; <<>> DiG 9.10.1 <<>> @192.168.1.7 www.isc.org. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 8101
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags; udp: 4096
;; QUESTION SECTION:
;www.isc.org.      IN  A

;; Query time: 973 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Thu Oct 16 20:28:20 CST 2014
;; MSG SIZE rcvd: 40
```

With `delv`, a "resolution failed" message is output instead:

```
$ delv @192.168.1.7 www.isc.org. A +rtrace
;; fetch: www.isc.org/A
;; resolution failed: failure
```

5.3 Logging

DNSSEC validation error messages by default will show up in syslog as a Query-Error. It will have the string "error" at the start of the message. Here is an example of what it may look like:

```
error (insecurity proof failed) resolving './NS/IN': 192.168.1.13#53
```

Usually, this level of error logging should suffice for most. If you would like to get more detailed information about why DNSSEC validation failed, read on to Section 5.3.1 to learn more.

5.3.1 BIND DNSSEC Debug Logging

A word of caution: before you enable debug logging, be aware that this may dramatically increase the load on your name servers.

With that said, sometimes it may become necessary to temporarily enable BIND debug logging to see more details of how DNSSEC is validating (or not). DNSSEC-related messages are not recorded in syslog by default, even if query log is enabled, only DNSSEC errors will show up in syslog. Enabling debug logging is not recommended for production servers, as it increases load on the server.

The example below shows how to enable debug level 3 (to see full DNSSEC validation messages) in BIND 9 and have it sent to syslog:

```
logging {
    channel dnssec_log {
        syslog daemon;
        severity debug 3;
        print-category yes;
    };
    category dnssec { dnssec_log; };
};
```

The example below shows how to log DNSSEC messages to their own file:

```
logging {
    channel dnssec_log {
        file "/var/log/dnssec.log";
        severity debug 3;
    };
    category dnssec { dnssec_log; };
};
```

After restarting BIND, a large number of log messages will appear in syslog. The example below shows the log messages as a result of successfully looking up and validating the domain name `www.isc.org`.

```
validating @0xb8012d88: . NS: starting
validating @0xb8012d88: . NS: attempting positive response validation
validating @0xb805a9b0: . DNSKEY: starting
validating @0xb805a9b0: . DNSKEY: attempting positive response validation
validating @0xb805a9b0: . DNSKEY: verify rdataset (keyid=19036): success
validating @0xb805a9b0: . DNSKEY: signed by trusted key; marking as secure
validator @0xb805a9b0: dns_validator_destroy
validating @0xb8012d88: . NS: in fetch_callback_validator
validating @0xb8012d88: . NS: keyset with trust 8
validating @0xb8012d88: . NS: resuming validate
validating @0xb8012d88: . NS: verify rdataset (keyid=8230): success
validating @0xb8012d88: . NS: marking as secure, noqname proof not needed
validator @0xb8012d88: dns_validator_destroy
validating @0xb8012d88: www.isc.org A: starting
validating @0xb8012d88: www.isc.org A: attempting positive response validation
validating @0xb805a9b0: isc.org DNSKEY: starting
validating @0xb805a9b0: isc.org DNSKEY: attempting positive response validation
validating @0xb827e298: isc.org DS: starting
validating @0xb827e298: isc.org DS: attempting positive response validation
validating @0xb827fd18: org DNSKEY: starting
validating @0xb827fd18: org DNSKEY: attempting positive response validation
validating @0xb8281798: . NS: starting
validating @0xb8281798: . NS: attempting positive response validation
validating @0xb8281798: . NS: keyset with trust 8
validating @0xb8280790: org DS: starting
validating @0xb8280790: org DS: attempting positive response validation
validating @0xb8280790: org DS: keyset with trust 8
validating @0xb8280790: org DS: verify rdataset (keyid=8230): success
```

```

validating @0xb8280790: org DS: marking as secure, noqname proof not needed
validator @0xb8280790: dns_validator_destroy
validating @0xb827fd18: org DNSKEY: in dsfetched
validating @0xb827fd18: org DNSKEY: dsset with trust 8
validating @0xb827fd18: org DNSKEY: verify rdataset (keyid=21366): success
validating @0xb827fd18: org DNSKEY: marking as secure (DS)
validator @0xb827fd18: dns_validator_destroy
validating @0xb827e298: isc.org DS: in fetch_callback_validator
validating @0xb827e298: isc.org DS: keyset with trust 8
validating @0xb827e298: isc.org DS: resuming validate
validating @0xb827e298: isc.org DS: verify rdataset (keyid=33287): success
validating @0xb827e298: isc.org DS: marking as secure, noqname proof not needed
validator @0xb827e298: dns_validator_destroy
validating @0xb805a9b0: isc.org DNSKEY: in dsfetched
validating @0xb805a9b0: isc.org DNSKEY: dsset with trust 8
validating @0xb805a9b0: isc.org DNSKEY: verify rdataset (keyid=12892): success
validating @0xb805a9b0: isc.org DNSKEY: marking as secure (DS)
validator @0xb805a9b0: dns_validator_destroy
validating @0xb8012d88: www.isc.org A: in fetch_callback_validator

```

5.4 Common Problems

5.4.1 Security Lameness

Similar to *Lame Delegation* in traditional DNS, this refers to the symptom when the parent zone holds a set of DS records that point to something that does not exist in the child zone. The resulting symptom is that the entire child zone may "disappear", being marked as bogus by validating resolvers.

Below is an example attempting to resolve the A record for a test domain name `www.example.com`. From the user's perspective, as described in Section 3.2.5, only SERVFAIL message is returned. On the validating resolver, we could see the following messages in syslog:

```

named[6703]: error (no valid RRSIG) resolving 'example.com/DNSKEY/IN': 149.20.61.151#53
named[6703]: error (broken trust chain) resolving 'www.example.com/DS/IN': 149.20.61.151#53
named[6703]: error (broken trust chain) resolving 'www.example.com/A/IN': 149.20.61.151#53

```

This gives us a hint that it is a broken trust chain issue. Let's take a look at the DS records that are published by querying one of the public DNS resolvers that supports DNSSEC. We have highlighted in the key tag ID returned, and shortened some keys for display:

```

$ dig @8.8.8.8 example.com. DS

; <<>> DiG 9.10.1 <<>> @8.8.8.8 example.com. DS
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 9640
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;example.com. IN DS

;; ANSWER SECTION:
example.com. 21599 IN DS 53476 8 2 1544D.....7DDA7
example.com. 21599 IN DS 53476 8 1 CD2AF...0B47B

;; Query time: 212 msec

```

```
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Thu Nov 27 17:23:42 CST 2014
;; MSG SIZE rcvd: 133
```

Next, we query for the DNSKEY and RRSIG of example.com, to see if there's anything wrong. Since we are having trouble validating, we flipped on the **+cd** option to disable checking for now to get the results back, even though they do not pass the validation tests. The **+multiline** option tells **dig** to print the type, algorithm type, and key id for DNSKEY records. Again, key tag ID's are highlighted, and some long strings are shortened for display:

```
$ dig @8.8.8.8 example.com. DNSKEY +dnssec +cd +multiline

; <<>> DiG 9.10.1 <<>> @8.8.8.8 example.com. DNSKEY +dnssec +cd +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20329
;; flags: qr rd ra cd; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags: do; udp: 512
;; QUESTION SECTION:
;example.com. IN DNSKEY

;; ANSWER SECTION:
example.com. 299 IN DNSKEY 257 3 8 (
    AwEAAePggU...0VPPEX+DE=
    ) ; KSK; alg = RSASHA256; key id = 48580
example.com. 299 IN DNSKEY 256 3 8 (
    AwEAAbMZp6...NRJnwyC/uX
    ) ; ZSK; alg = RSASHA256; key id = 60426
example.com. 299 IN RRSIG DNSKEY 8 2 300 (
    20141227074820 20141127064820 48580 example.com.
    ph3eBXsBQy...fQTRTlpg== )
example.com. 299 IN RRSIG DNSKEY 8 2 300 (
    20141227074820 20141127064820 60426 example.com.
    VaQ0INia3a...nj3YTPv5A= )

;; Query time: 368 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Fri Nov 28 11:33:00 CST 2014
;; MSG SIZE rcvd: 961
```

Here is our problem: the parent zone is telling the world that example.com is using the key 53476, but the authoritative servers are saying: no no no, I am using keys 48580 and 60426. There might be several causes for this mismatch, one possibility is that a malicious attacker has compromised one side and change the data. The more likely scenario is that the DNS administrator for the child zone did not upload the correct key information to the parent zone.

5.4.2 Incorrect Time

In DNSSEC, every record will come with at least one RRSIG, and RRSIG contains two timestamps indicating when it starts becoming valid, and when it expires. If the validating resolver's current system time does not fall within the RRSIG two timestamps, the following error messages occur in BIND debug log.

First, the example below shows the log messages when the RRSIG has expired. This could mean the validating resolver system time is incorrectly set too far in the future, or the zone administrator has not kept up with RRSIG maintenance.

```
validating @0xb7b839b0: . DNSKEY: verify failed due to bad signature (keyid=19036): RRSIG ←
has expired
```

The logs below show RRSIG validity period has not begun. This could mean validation resolver system is incorrectly set too far in the past, or the zone administrator has incorrectly generated signatures for this domain name.

```
validating @0xb7c1bd88: www.isc.org A: verify failed due to bad signature (keyid=4521): ←
RRSIG validity period has not begun
```

5.4.3 Invalid Trust Anchors

As we have seen in the section Section 3.4, whenever a DNSKEY is received by the validating resolver, it is actually compared to the list of keys the resolver has explicitly trusted to see if further action is needed. If the two keys match, the validating resolver stops performing further verification and returns the answer(s) as validated.

But what if the key file on the validating resolver is misconfigured or missing? Below we show some examples of log messages when things are not working properly.

First of all, if the key you copied is malformed, BIND will not even start up and you will likely find this error message in syslog:

```
named[18235]: /etc/bind/named.conf.options:29: bad base64 encoding
named[18235]: loading configuration: failure
```

If the key is a valid base64 string, but the key algorithm is incorrect, or if the wrong key is installed, the first thing you will notice is that pretty much all of your DNS lookups result in SERVFAIL, even when you are looking up domain names that have not been DNSSEC-enabled. Below shows an example of querying a recursive server 192.168.1.7:

```
$ dig @192.168.1.7 www.example.com. A

; <<>> DiG 9.10.1 <<>> @192.168.1.7 www.example.com. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 8093
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:;; udp: 4096
;; QUESTION SECTION:
;www.example.com.      IN  A
```

delv shows similar result:

```
$ delv @192.168.1.7 www.example.com. +rtrace
;; fetch: www.example.com/A
;; resolution failed: failure
```

The next symptom you will see is in the DNSSEC log messages:

```
validating @0xb8b18a38: . DNSKEY: starting
validating @0xb8b18a38: . DNSKEY: attempting positive response validation
validating @0xb8b18a38: . DNSKEY: unable to find a DNSKEY which verifies the DNSKEY RRset ←
and also matches a trusted key for '.'
validating @0xb8b18a38: . DNSKEY: please check the 'trusted-keys' for '.' in named.conf.
```

5.4.4 Unable to Load Keys

This is a simple yet common issue. If the keys files were present but not readable by **named**, the syslog messages are clear, as shown below:

```
named[32447]: zone example.com/IN (signed): reconfiguring zone keys
named[32447]: dns_dnssec_findmatchingkeys: error reading key file Kexample.com.+008+06817. ←
private: permission denied
named[32447]: dns_dnssec_findmatchingkeys: error reading key file Kexample.com.+008+17694. ←
private: permission denied
named[32447]: zone example.com/IN (signed): next key event: 27-Nov-2014 20:04:36.521
```

However, if no keys are found, the error is not as obvious. Below shows the syslog messages after executing **rndc reload**, with the key files missing from the key directory:

```
named[32516]: received control channel command 'reload'
named[32516]: loading configuration from '/etc/bind/named.conf'
named[32516]: reading built-in trusted keys from file '/etc/bind/bind.keys'
named[32516]: using default UDP/IPv4 port range: [1024, 65535]
named[32516]: using default UDP/IPv6 port range: [1024, 65535]
named[32516]: sizing zone task pool based on 6 zones
named[32516]: the working directory is not writable
named[32516]: reloading configuration succeeded
named[32516]: reloading zones succeeded
named[32516]: all zones loaded
named[32516]: running
named[32516]: zone example.com/IN (signed): reconfiguring zone keys
named[32516]: zone example.com/IN (signed): next key event: 27-Nov-2014 20:07:09.292
```

This happens to look exactly the same as if the keys were present and readable, and **named** loaded the keys and signed the zone. It will even generate the internal (raw) files:

```
# cd /etc/bind/db
# ls
example.com.db  example.com.db.jbk  example.com.db.signed
```

If **named** really loaded the keys and signed the zone, you should see the following files:

```
# cd /etc/bind/db
# ls
example.com.db  example.com.db.jbk  example.com.db.signed  example.com.db.signed.jnl
```

So, unless you see the `*.signed.jnl` file, your zone has not been signed.

5.5 Negative Trust Anchors

BIND 9.11 introduced *Negative Trust Anchors* (NTAs) as a means to *temporarily* disable DNSSEC validation for a zone when you know that the zone's DNSSEC is mis-configured.

NTAs are added using the `rndc` command, e.g:

```
$ rndc nta example.com
Negative trust anchor added: example.com/_default, expires 14-Dec-2016 13:39:09.000
```

The list of currently configured NTAs can also be examined using `rndc`, e.g:

```
$ rndc nta -dump
example.com: expiry 14-Dec-2016 13:39:09.000
```

The default lifetime of an NTA is one hour although, by default, BIND will poll the zone every five minutes to see if the zone now correctly validates, at which point the NTA will automatically expire. Both the default lifetime and the polling interval may be configured via `named.conf`, and the lifetime can be overridden on a per-zone basis using the `-lifetime duration` parameter to `rndc nta`. Both timer values have a permitted maximum value of one week.

5.6 NSEC3 Troubleshooting

BIND includes a tool called **nsec3hash** that runs through the same steps a validating resolver would, to generate the correct hashed name based on NSEC3PARAM parameters. The command takes the following parameters in order: salt, algorithm, iterations, and domain. For example, if the salt is 1234567890ABCDEF, hash algorithm is 1, and iteration is 10, to get the NSEC3-hashed name for `www.example.com` we would execute a command like this:

```
$ nsec3hash 1234567890ABCDEF 1 10 www.example.com
RN7I9ME6E1I6BDKIP91B9TCE4FHJ7LKF (salt=1234567890ABCDEF, hash=1, iterations=10)
```

While it is unlikely you would construct a rainbow table of your own zone data, this tool might be useful to troubleshoot NSEC3 problems.

5.7 Troubleshooting Example

Let's put what we've looked at together into an example and see the steps taken to solve the problem. We start with someone complaining that she is unable to resolve the name `www.example.com`. We use `dig` on her machine to verify the behavior, and we received the following output from `dig`:

```
$ dig www.example.com. A

; <<>> DiG 9.10.1 <<>> www.example.com. A
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 26068
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.com.    IN  A

;; Query time: 784 msec
;; SERVER: 192.168.1.7#53(192.168.1.7)
;; WHEN: Mon Nov 03 20:00:45 CST 2014
;; MSG SIZE rcvd: 44
```

We learned from this output that the recursive name server `192.168.1.7` returned a generic error message when resolving the name `www.example.com`. The next step is to look at the DNS server configuration on `192.168.1.7` to see how it is configured. Below is an excerpt of `named.conf` from `192.168.1.7`:

```
options {
    ...
    forwarders {192.168.1.11;};
    forward only;
    ...
};
```

This tells us that the recursive name server `192.168.1.7` just sends all recursive queries to `192.168.1.11`. Let's query `192.168.1.11`:

```
$ dig @192.168.1.11 www.example.com. A

; <<>> DiG 9.10.1 <<>> @192.168.1.11 www.example.com. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 24171
;; flags: qr rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1
...
```

And we get the same result as when we queries `192.168.1.7`, generic failure message, but we also learned that `192.168.1.11` is not authoritative for `example.com` (no `aa` flag), so it is getting this response from somewhere else. Below is the configuration excerpt from `192.168.1.11`:

```
options {
    ...
    forwarders {};
    forward only;
    ...
};

zone "example.com" IN {
    type forward;
    forwarders { 192.168.1.13; };
    forward only;
};
```

At first glance, it may look like 192.168.1.11 is just performing recursion itself, querying Internet name servers directly; however, further down the configuration file, we see the forward zone definition, which tell us that 192.168.1.11 is doing conditional forwarding just for `example.com`, and it is sending all `example.com` queries to 192.168.1.13.

We then query 192.168.1.13:

```
$ dig @192.168.1.13 www.example.com. A

; <<>> DiG 9.10.1 <<>> @192.168.1.13 www.example.com. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 35962
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.example.com.    IN    A

;; ANSWER SECTION:
www.example.com.    600 IN  A  192.168.1.100

;; Query time: 4 msec
;; SERVER: 192.168.1.13#53(192.168.1.13)
;; WHEN: Mon Nov 03 20:06:26 CST 2014
;; MSG SIZE rcvd: 60
```

Finally! We found the authoritative name server! Now we know our query path looks like this:

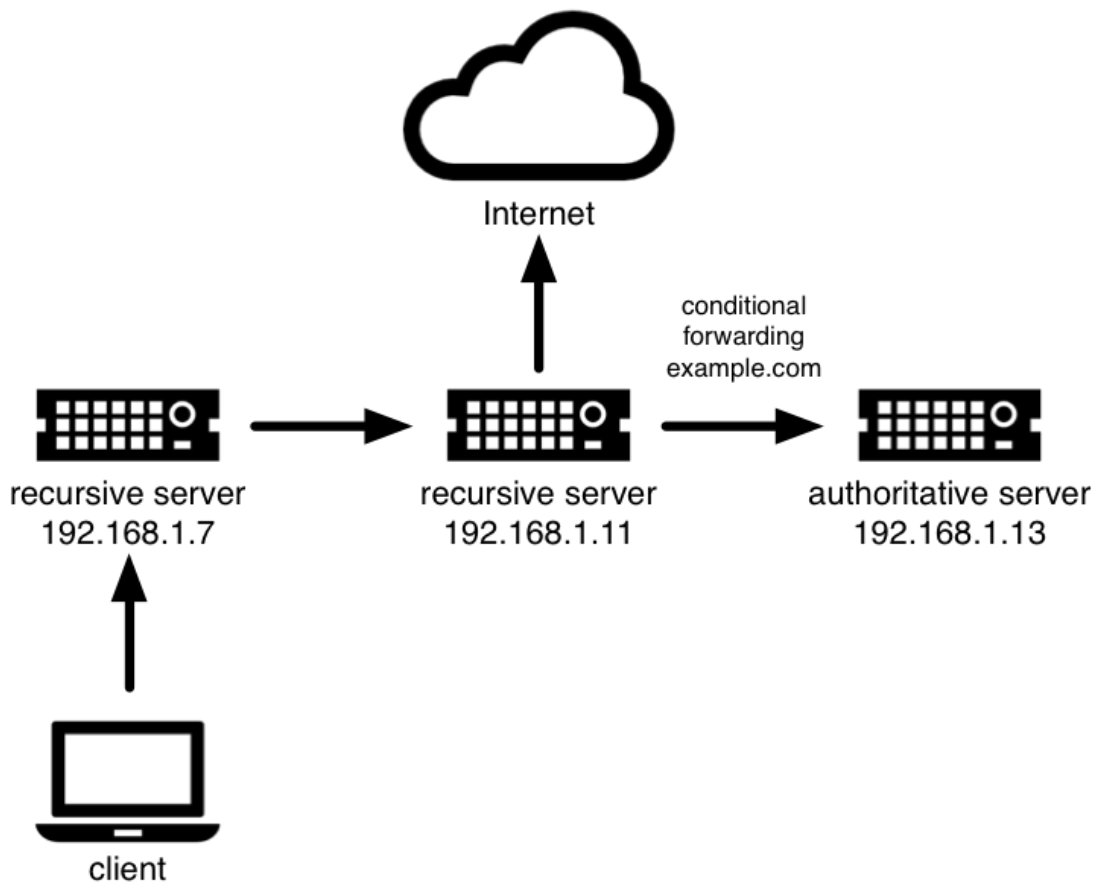


Figure 5.1: Query Path

But 192.168.1.13 has no trouble answering the query for `www.example.com`, so the problem might be between 192.168.1.11 and 192.168.1.13? We know there are no firewalls or network devices between 192.168.1.11 and 192.168.1.13 that could intercept packets. Let's query 192.168.1.11 again, but this time, let's purposely turn off DNSSEC validation by using `+cd` (checking disabled), to see if this error message was caused by DNSSEC validation:

```

$ dig @192.168.1.11 www.example.com. A +cd

; <<>> DiG 9.10.1 <<>> @192.168.1.11 www.example.com. A +cd
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 58332
;; flags: qr rd ra cd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags;; udp: 4096
;; QUESTION SECTION:
;www.example.com. IN A

;; ANSWER SECTION:
www.example.com. 562 IN A 192.168.1.100

;; Query time: 2 msec
;; SERVER: 192.168.1.11#53(192.168.1.11)
;; WHEN: Mon Nov 03 20:01:23 CST 2014

```



```
;; MSG SIZE rcvd: 60
```

Bingo! So the problem is on 192.168.1.11, and specifically, with DNSSEC validation. Now we can focus our attention on the configuration on 192.168.1.11, examine its logs, check its system time, or check its trust anchors, to see what may be the root cause.

Examining log messages from 192.168.1.11, we notice the following two entries:

```
error (no valid KEY) resolving 'example.com/DNSKEY/IN': 192.168.1.13#53
error (broken trust chain) resolving 'www.example.com/A/IN': 192.168.1.13#53
```

So it would appear that on the server 192.168.1.11, there is a broken trust chain. At this point, we can probably conclude the problem is in one of the trusted-keys statements on 192.168.1.11, but let's turn on DNSSEC debug logging (as described in Section 5.3.1), and re-run the **dig** for `www.example.com` one more time to see what log messages get generated:

```
...
validating @0xb4b48968: example.com DNSKEY: attempting positive response validation
validating @0xb4b48968: example.com DNSKEY: unable to find a DNSKEY which verifies the ←
  DNSKEY RRset and also matches a trusted key for 'example.com'
validating @0xb4b48968: example.com DNSKEY: please check the 'trusted-keys' for 'example. ←
  com' in named.conf.
...
```

Okay, so we have a confirmed log message telling us to look at 'trusted-keys'. The `named.conf` on 192.168.1.11 contains the following:

```
trusted-keys {
    example.com. 257 3 8 "AwEAAbluLK0k3dPKnsJNd5tGbO5bgh7WuXzaSDQVwi/qqPdCR65ZDiin
        0GTpL++B1iKYDP4rRL/s/2TMppI1fV638f2SuhNQ9zYIuCo/FuHeJB7/
        DBQ03eJFvN1QHC0we2uUFRxazz8eT9nkI1SUu0fhcs6CA06gGqauDbpU
        mpM7VUX3";
};
```

Let's check the authoritative server (192.168.1.13) for the correct key:

```
$ dig @192.168.1.13 example.com. DNSKEY +multiline

; <<>> DiG 9.10.1 <<>> @192.168.1.13 example.com. DNSKEY +multiline
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 38451
;; flags: qr aa rd; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1
;; WARNING: recursion requested but not available

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;example.com.      IN DNSKEY

;; ANSWER SECTION:
example.com.      600 IN DNSKEY 256 3 8 (
    AwEAAbluLK0k3dPKnsJNd5tGbO5bgh7WuXzaSDQVwi/q
    qPdCR65ZDiin0GTpL++B1iKYDP4rRL/s/2TMppI1fV63
    8f2SuhNQ9zYIuCo/FuHeJB7/DBQ03eJFvN1QHC0we2uU
    FrXazz8eT9nkI1SUu0fhcs6CA06gGqauDbpUmpM7VUX3
    ) ; ZSK; alg = RSASHA256; key id = 4974
example.com.      600 IN DNSKEY 257 3 8 (
    AwEAAb4N53kPbdRTAwvJT8OYVeVhQIldwppMy7KBJ+8k
    Uggx2PU3yP/qlq4Zjl0MMmqRiJhD/S+z9cJLNTZ9tHz1
    7aZQjFyGAYuU3DGW16xfMolcIn+c8TpPCzBOFhxx6jvO
    VLlz+Wgyi1ES+t29FjYYv5cVNRPMxXLRj1HFd01DzX3N
```

```
dmcUoZ+VVJCvaML9+6UpL/6jitNsoU8JHnxT9B2CGKcw
N7VaK419Ida2BqY3/4UVqWzhj03/M5LK6cn1pEQbQMtY
R0TNJURBKdK8bH663h98i23tVX0/85IsCVBL4Dd2boa3
/7HPp7uZN1AjDvcRsOhlmqixwUGmVm1EskDIMy8=
) ; KSK; alg = RSASHA256; key id = 45319
example.com. 600 IN DNSKEY 256 3 8 (
AwEAAfbc/0ESumm1mPVkm025PfHKHNYW62yx0wyLN5LE
4DifN6FzIVSKSGdM0dq+z6vFGxzzjPDz7QZdeC6ttIUA
Bo4tG7dDrsWK+tG5cm4vuylsEVbnnW5i+gFG/02+RYmZ
ZT9AobXB5bvjfX19SDBgpBluB35WUCAnK9WkRRUS08lf
) ; ZSK; alg = RSASHA256; key id = 60798
example.com. 600 IN DNSKEY 257 3 8 (
AwEAAb3lVweaj4dA9dvmcwlkaVpJ4/3ccXbRjgV7jqh1
p0REL8fI0Z42E9SdxdsdTi+2XYcmHDQYEoqwYh70t/4P
4oObZFIUhl+hhKldXQNZGtzT0xF60k527N9cHPddoXzg
AXYBtGL1LMSJcV8s0rw/i+64xNGdRWpFRdo78RhJ5LU3
1SAPUnhi3OvJgsOpBPntrSyX6iA5ZotitxZJNTqP+Jck
lhPWFgFOBgdvWJ369BR1DGy/m8+pctypZq1hy7ZteHet
r55/cLBXY1BEzz3Q8vLUnSOu5An8IF0v2Gt7hOyY3nqu
bU5vjCbogLj1K5ySBAJbHcCPAFrPGSIfmRize+U=
) ; KSK; alg = RSASHA256; key id = 40327

;; Query time: 4 msec
;; SERVER: 192.168.1.13#53(192.168.1.13)
;; WHEN: Mon Nov 03 21:51:28 CST 2014
;; MSG SIZE rcvd: 888
```

Did you spot the mistake? We have the correct key data in our configuration, but the key type was incorrect. In our configuration, the key was configured as a KSK (257), while the authoritative server indicates that it is a ZSK (256).

Chapter 6

Advanced Discussions

6.1 Key Generation

6.1.1 Can I Use the Same Key Pair for Multiple Zones?

Yes and no. Good security practice suggests that you should use unique key pairs for each zone, just like how you should have different passwords for your email account, social media login, and online banking credentials. On a technical level, this is completely feasible, but then multiple zones are at risk when one key pair is compromised. If you have hundreds or thousands (or even hundreds of thousands) of zones to administer, a single key pair for all might be less error-prone to manage. You may choose to use the same approach to password management: use unique passwords for your bank accounts and shopping sites, but use a standard password for your not-very-important logins. So categorize your zones, high valued zones (or zones that have specific key rollover requirements) get their own key pairs, while other more "generic" zones can use a single key pair for easier management.

6.1.2 Do I Need Separate ZSK and KSK?

No, it is not required that you create two separate sets of keys, but you should, for operational ease. The DNSSEC protocol itself does not require two classes of keys, but for operational practicality, having two classes of keys make the life of a typical DNS(SEC) administrator's life easier. One of the advantages of having separate ZSKs and KSKs is a better balance between security and ease of use: the KSK can be stored in a secure and less accessible area, while the ZSK is easily accessible for routine use. For more details and considerations on this topic, please refer to *RFC 6781 Section 3* .

Please refer to Table 4.1 for a comparison of how each of the keys are used.

6.1.3 Which Algorithm?

There are at least three algorithm choices for DNSSEC as of this writing (late 2016):

- RSA
- Digital Signature Algorithm (DSA)
- Elliptic Curve DSA (ECDSA)

While all three are supported by BIND, RSA is the only one that is mandated to be implemented with DNSSEC and, at the time of writing, is the most widely supported algorithm by both name servers and clients. For the time being, RSA/SHA-256 is the algorithm of choice.

However, RSA is a little long in the tooth, and ECDSA is emerging as the next new cryptographic standard. In fact, the US federal government recommended to stop using RSA altogether by September 2015, and migrate to using ECDSA or similar algorithms.

So for now, use RSA, but keep your eyes on emerging new standards or requirements. For details about rolling over DNSKEYs to a new algorithm, see Section 6.4.5.

6.1.4 Key Sizes

The choice of key sizes is a classic issue of finding the balance between performance and security. The larger the key size, the longer it takes for an attacker to crack the key; but larger keys also means more resources are needed both when generating signatures (authoritative servers) and verifying signatures (recursive servers).

Of the two sets of keys, ZSK is used much more frequently. Whenever zone data changes, or when signatures expire, ZSK is used, so performance certainly is of a bigger concern. As for KSK, it is used less frequently, so performance is less of a factor, but its impact is bigger because of its role in signing other keys.

In this guide, the following key length were chosen for each set, with the recommendation that they be rotated more frequently for better security:

- ZSK: RSA 1024 bits, rollover every year
- KSK: RSA 2048 bits, rollover every five years

These should be the minimum key sizes one should choose. At the time of writing (late 2016) the root zone and many TLDs are already using 2048 bit ZSKs.

If you choose to implement larger key sizes, keep in mind that larger key size results in larger DNS responses, and this may mean more load on network resources. Depending on network configuration, end users may even experience resolution failures due to the increased response sizes, as we have discussed in Section 3.5.

6.2 Proof of Non-Existence (NSEC and NSEC3)

How do you prove that something does not exist? This zen-like question is an interesting one, and in this section we will provide an overview of how DNSSEC solves the problem.

Why is it even important to have authenticated denial of existence? Couldn't we just send back a "hey, what you asked for does not exist", and somehow generate a digital signature to go with it, proving it really is from the correct authoritative source? Well, the technical challenge of signing nothing aside, this solution has flaws, one of which is it gives an attacker a way to create the appearance of denial of service by replaying this message on the network.

We are going to use a little story, and tell it three different times to illustrate how proof of nonexistence works. In our story, we run a small company with three employees: Alice, Edward, and Susan. We list their names in a phone directory, and we hired a nameless intern to answer our phone calls.

If we followed the approach of giving back the same answer no matter what was asked, when people called and asked for "Bob", our intern would simply answer: "Sorry, that person doesn't work here, and to prove that I am not lying, here's the signature: 'deaf coffee beef'". Now this is a legitimate answer, but since the signature doesn't change, an attacker could record this message, and when the next person called in asking for Susan, she will hear the exact same message: "Sorry, that person doesn't work here, and to prove that I am not lying, here's the signature: 'deaf coffee beef'". And this answer is verifiable, since the magic signature ("deaf coffee beef") can be validated ¹. Now the attacker has successfully fooled the caller into thinking that Susan doesn't work at our company, and might even be able to convince all callers that no one works at this company (no names exist).

To solve this problem, two different solutions were created, we will look at the first one, NSEC, next.

6.2.1 NSEC

The NSEC record is used to prove that something really does not exist, by providing the name before it, and the name after it. Using our tiny company example, this would be analogous to someone calling for Bob over the phone, and our nameless intern answered the phone with: "I'm sorry, that person doesn't work here. The name before that is Alice, and the name after that

¹ Actually it cannot be verified, read Section 3.3.3 to review why this would not work.

is Edward". Let's say someone called in again for a non-existent person, Oliver, the answer would be: "I'm sorry, that person doesn't work here. The name before that is Edward, and the name after that is Susan". Another caller asked for Todd, and the answer would be: "I'm sorry, that person doesn't work here. The name before that is Susan, and there's no other name after that".

So we end up with four NSEC records:

```
example.com.      300 IN  NSEC  alice.example.com.  A RRSIG NSEC
alice.example.com. 300 IN  NSEC  edward.example.com. A RRSIG NSEC
edward.example.com. 300 IN  NSEC  susan.example.com.  A RRSIG NSEC
susan.example.com. 300 IN  NSEC  example.com.       A RRSIG NSEC
```

What if the attacker tried to use the same replay method described earlier? If someone called for Edward, none of the four answers would fit. If attacker played message #2, "I'm sorry, that person doesn't work here. The name before it is Alice, and the name after it is Edward", it is obviously false, since "Edward" is in the response; same for #3, Edward and Susan. As for #1 and #4, Edward does not fall in range before Alice or after Susan, so the caller can logically deduce that it was an incorrect answer.

In BIND inline signing, your zone data will be automatically sorted on the fly before generating NSEC records, much like how a phone directory is sorted.

The NSEC record allows for a proof of non-existence for record types. If you ask a signed zone for a name that exists but for a record type that doesn't (for that name), the signed NSEC record returned lists all of the record types that *do* exist for the requested domain name.

NSEC records can also be used to show whether a record was generated as the result of a wildcard expansion or not. The details of this are out of scope for this document, but are described well in [RFC 7129](#).

Unfortunately, the NSEC solution has a few drawbacks, one of which is trivial "zone walking". A curious person can keep calling back, and our nameless, gullible intern will keep divulging information about our employees. Imagine if the caller first asked: "Is Bob there?" and received back the names Alice and Edward. The caller can then call back again: "Is Edward A. there?", and will get back Edward and Susan. Repeat the process enough times, the caller will eventually learn every name in our company phone directory. For many of you, this may not be a problem, since the very idea of DNS is similar to a public phone book: if you don't want a name to be known publicly, don't put it in DNS! Consider using DNS views (split DNS) and only display your sensitive names to a selective audience.

The second drawback of NSEC is a actually increased operational overhead: no opt-out mechanism for insecure child zones, this generally is a problem for parent zone operators dealing with a lot of insecure child zones, such as `.com`. To learn more about opt-out, please see Section [6.2.2.2](#).

6.2.2 NSEC3

NSEC3 adds two additional features that NSEC does not have:

1. No easy zone enumeration.
2. Provides a mechanism for child zone to opt out.

Recall, in Section [6.2.1](#), we provided a range of names to prove that something really does not exist. But as it turns out, even disclosing these ranges of names becomes a problem: this made it very easy for the curious minded to look at your entire zone. Not only that, unlike a zone transfer, this "zone walking" is more resource intensive. So how do we disclose something, without actually disclosing it?

The answer is actually quite simple, hashing functions, or one-way hashes. Without going into many details, think of it like a magical meat grinder. A juicy piece of ribeye steak goes in one end, and out comes a predictable shape and size of ground meat (hash) with a somewhat unique pattern. No matter how hard you try, you cannot turn the ground meat back into the juicy ribeye steak, that's what we call a one-way hash.

NSEC3 basically runs the names through a one-way hash, before giving it out, so the recipients can verify the non-existence, without any knowledge of the actual names.

So let's tell our little phone receptionist story for the third time, this time with NSEC3. This time, our intern is not given a list of actual names, he is given a list of "hashed" names. So instead of Alice, Edward, and Susan, the list he is given reads like this (hashes shortened for easier reading):

```
FSK5.... (produced from Edward)
JKMA.... (produced from Susan)
NTQ0.... (produced from Alice)
```

Then, the phone rings, someone's asking for Bob again. Our intern takes the name Bob through a hash function, and the result is L8J2..., so he tells them on the phone: "I'm sorry, that person doesn't work here. The name before that is JKMA..., and the name after that is NTQ0...". There, we proved Bob doesn't exist, without giving away any names! To put that into proper NSEC3 resource records, they would look like this (again, hashes shortened for display):

```
FSK5....example.com. 300 IN NSEC3 1 0 10 1234567890ABCDEF JKMA... A RRSIG
JKMA....example.com. 300 IN NSEC3 1 0 10 1234567890ABCDEF NTQ0... A RRSIG
NTQ0....example.com. 300 IN NSEC3 1 0 10 1234567890ABCDEF FSK5... A RRSIG
```

Hashes and Privacy

Just because we employed one-way hash functions does not mean there's no way for a determined individual to figure out what your zone data is. Someone could still gather all of your NSEC3 records and hashed names, and perform an offline brute-force attack by trying all possible combinations to figure out what the original name is. This would be like if someone really wanted to know how you got the ground meat, he could buy all cuts of meat and ground it up at home using the same model of meat grinder, and compare the output with the meat you gave him. It is expensive and time consuming (especially with real meat), but like everything else in cryptography, if someone has enough resources and time, nothing is truly private forever. If you are concerned about someone performing this type of attack on your zone data, see about adding salt as described in Section 6.2.2.3.

6.2.2.1 NSEC3PARAM

The above NSEC3 examples used four parameters: 1, 0, 10, and 1234567890ABCDEF. The **rndc** tool may be used to set the NSEC3 parameters for a zone; for example:

```
# rndc signing -nsec3param 1 0 10 1234567890abcdef example.com
```

1 represents the algorithm, 0 represents the opt-out flag, 10 represents the number of iterations, and 1234567890abcdef is the salt. Let's look at how each one can be configured:

- *Algorithm*: Not much of a choice here, the only defined value currently is 1 for SHA-1.
- *Opt-out*: Set this to 1 if you want to do NSEC3 opt-out, which we will discuss in Section 6.2.2.2.
- *Iterations*: iterations defines the number of additional times to apply the algorithm when generating an NSEC3 hash. More iterations yields more secure results, but consumes more resources for both authoritative servers and validating resolvers. In this regard, we have similar considerations as we've seen in Section 6.1.4 of security versus resources.
- *Salt*: The salt is a string of data expressed in hexadecimal, or a hyphen ('-') if no salt is to be used. We will learn more about salt in Section 6.2.2.3.

For example, to create an NSEC3 chain using the SHA-1 hash algorithm, no opt-out flag, 10 iterations, and a salt value of "FFFF", use:

```
# rndc signing -nsec3param 1 0 10 FFFF example.com
```

To set the opt-out flag, 15 iterations, and no salt, use:

```
# rndc signing -nsec3param 1 1 15 - example.com
```

6.2.2.2 NSEC3 Opt-Out

One of the advantages of NSEC3 over NSEC is the ability for parent zones to publish less information about its child or delegated zones. Why would you ever want to do that? Well, if a significant number of your delegations are not yet DNSSEC-aware, meaning they are still insecure or unsigned, generating DNSSEC-records for their NS and glue records is not a good use of your precious name server resources.

The resources may not seem like a lot, but imagine in if you are the operator of busy top level domains such as `.com` or `.net`, with millions and millions of insecure delegated domain names, it quickly adds up. As of late 2016, less than 0.5% of all `.com` zones are signed. Basically, without opt-out, if you have 1,000,000 delegations, only 5 of which are secure, you still have to generate NSEC RRset for the other 999,995 delegations; with NSEC3 opt-out, you will have saved yourself 999.995 sets of records.

For most DNS administrators who do not manage a large number of delegations, the decision whether or not to use NSEC3 opt-out is probably not relevant.

To learn more about how to configure NSEC3 opt-out, please see Section [7.3.4](#).

6.2.2.3 NSEC3 Salt

As described in Section [6.2.2](#), while NSEC3 doesn't put your zone data in plain public display, it is still not difficult for an attacker to collect all the hashed names, and perform an offline attack. All that is required is running through all the combinations to construct a database of plaintext names to hashed names, also known as a "rainbow table".

There is one more features NSEC3 gives us to provide additional protection: salt. Basically, salt gives us the ability introduce further randomness into the hashed results. Whenever the salt is changed, any pre-computed rainbow table is rendered useless, and a new rainbow table must be re-computed. If the salt is changed from time to time, it becomes difficult to construct a useful rainbow table, thus difficult to walk the DNS zone data programmatically. How often you want to change your NSEC3 salt is up to you.

To learn more about what steps to take to change NSEC3, please see Section [7.3.3](#).

6.2.3 NSEC or NSEC3?

So which one should you choose? NSEC or NSEC3? There is not really a single right answer here that fits everyone. It all comes down to your needs or requirements.

If you prefer not to make your zone easily enumerable, implementing NSEC3 paired with a periodically changed salt will provide a certain level of privacy protection. However, someone could still randomly guess the names in your zone (such as "ftp" or "www"), as in the traditional insecure DNS.

If you have many many delegations, and have a need for opt-out to save resources, NSEC3 is for you.

Other than that, using NSEC is typically a good choice for most zone administrators, as it relieves the authoritative servers from the additional cryptographic operations that NSEC3 requires, and NSEC is comparatively easier to troubleshoot than NSEC3.

6.3 Key Storage

6.3.1 Public Key Storage

The beauty of a public key cryptography system is that the public key portion can and should be distributed to as many people as possible. As the administrator, you may want to keep the public keys on an easily accessible file system for operational ease, but there is no need to securely store them, since both ZSK and KSK public keys are published in the zone data as DNSKEY resource records.

Additionally, a hash of the KSK public key is also uploaded to the parent zone (see Section [4.4](#) for more details), and is published by the parent zone as DS records.

6.3.2 Private Key Storage

Ideally, private keys should be stored offline, in secure devices such as a smart card. Operationally, however, this creates certain challenges, since we need the private key to create RRSIG resource records, and it would be a hassle to bring the private key out of storage every time the zone file changes or when signatures expire.

A common approach to strike the balance between security and practicality is to have two sets of keys, a ZSK set, and a KSK set. ZSK private key is used to sign zone data, and can be kept online for ease of use; KSK private key is used to sign just the DNSKEY (the ZSK), it is used less frequently, and can be stored in a much more secure and restricted fashion.

For example, a KSK private key stored on a USB flash drive that is kept in a fireproof safe, only brought online once a year to sign a new pair of ZSK, combined with a ZSK private key stored on the network file-system available for routine use, may be a good balance between operational flexibility and security.

And if you need to change your keys, please see Section [6.4.1](#).

6.3.3 Hardware Security Modules (HSM)

A Hardware Security Module (HSM) comes in different shapes and sizes, but as the name indicates, it's a physical device or devices, usually with some or all of the following features:

- Tamper-resistant key storage
- Strong random number generation
- Hardware for faster cryptographic operations

Most organizations do not incorporate HSMs into their security practices due to cost and the added operational complexity.

BIND supports PKCS #11 (Public Key Cryptography Standard #11) for communication with HSMs and other cryptographic support devices. For more information on how to configure BIND to work with HSMs, please refer to the [BIND 9 Administrator Reference Manual](#).

6.4 Key Management

Best practice for DNSSEC key management is to use different keys to sign zone data (ZSK) and DNSKEY data (KSK), as we've discussed in Section [6.1.2](#). Since these keys serve different functions, their timing and methods of rollovers are also different. In Section [4.6](#), we have broadly talked about how to perform a generic ZSK and KSK rollover. In this section, we will discuss two topics in more detail:

1. Different considerations and methods of key rollovers.
2. Key meta data and management

6.4.1 Key Rollovers

Generally speaking, ZSKs should be rolled more frequently than KSKs. In Section [4.6](#), we described at a very high level how to roll the ZSK every year using key pre-publication (described below), and how to roll the KSK every five years using double DS (also described below). Here, we show some other methods of rolling keys. To see examples of key rolling, please refer to Section [7.2](#). For (far) deeper discussions and considerations on the topic of key rolling, check out [RFC 7583](#).

6.4.1.1 ZSK Rollover Methods

Generally speaking, the ZSK is smaller in size (compared to the KSK) for performance, but smaller keys take less time to break, thus the ZSK should be changed, or rolled, more frequently. The ZSK can be rolled in one of the following two ways:

1. *Pre-publication*: Publish the new ZSK into zone data before it is actually used. Wait at least one TTL so the world's recursive servers know about both keys, then stop using the old key and generate new RRSIG using the new key. Wait at least another TTL, so the cached old key data is expunged from world's recursive servers, before removing the old key.

The benefit of the Pre-publication approach is it does not dramatically increase the zone size, but the duration of the rollover is longer. If insufficient time has passed after the new ZSK is published, some resolvers may only have the old ZSK cached when the new RRSIG records are published, and validation may fail. This is the method that was described in Section 4.6.1 and Section 7.2.1

2. *Double Signature*: Publish the new ZSK and new RRSIG, essentially double the size of the zone. Wait at least one TTL before removing the old ZSK and old RRSIG.

The benefit of the Double Signature approach is that it is easier to understand and execute, but suffers from increased zone size (essentially double) during a rollover event.

6.4.1.2 KSK Rollover Methods

Rolling the KSK requires interaction with the parent zone, so operationally this may be more complex than rolling ZSKs. There are three methods of rolling the KSK:

1. *Double-DS*: the new DS record is published. After waiting for this change to propagate into caches, the KSK is changed. After a further interval during which the old DNSKEY RRset expires from caches, the old DS record is removed.

Double-DS is the reverse of Double-KSK: the new DS is published at the parent first, then the KSK at the child is updated, then remove the old DS at the parent. The benefit is that the size of the DNSKEY RRset is kept to a minimum, but interactions with the parent zone is increased to two events. This is the method that is described in Section 4.6.2 and Section 7.2.2.

2. *Double-KSK*: the new KSK is added to the DNSKEY RRset which is then signed with both the old and new key. After waiting for the old RRset to expire from caches, the DS record in the parent zone is changed. After waiting a further interval for this change to be reflected in caches, the old key is removed from the RRset.

Basically, the new KSK is added first at the child zone and being used to sign DNSKEY, then the DS record is changed, followed by the removal of the old KSK. Double-KSK limits the interaction with the parent zone to a minimum, but for the duration of the rollover, the size of the DNSKEY RRset is increased.

3. *Double-RRset*: the new KSK is added to the DNSKEY RRset which is then signed with both the old and new key, and the new DS record added to the parent zone. After waiting a suitable interval for the old DS and DNSKEY RRsets to expire from caches, the old DNSKEY and DS record are removed.

Double-RRset is the fastest way to roll the KSK (shortest rollover time), but has the drawbacks of both of the other methods: a larger DNSKEY RRset and two interactions with the parent.

6.4.2 Key Management and Metadata

In Section 4.3.2.4, we alluded that `auto-dnssec` is doing a lot of automation for us so we don't have to, and we've also alluded to something called the key timing metadata. In fact, if you looked at your key file, it likely already has a section near the top that looks like this:

```
; Publish: 20141120094612 (Thu Nov 20 17:46:12 2014)
; Activate: 20141120094612 (Thu Nov 20 17:46:12 2014)
```

These are only two of the five metadata fields of a key. Below is a complete list of each of the metadata fields, and how it affects your key's behavior:

1. *Publish*: Sets the date on which a key is to be published to the zone. After that date, the key will be included in the zone but will not be used to sign it (yet). This is notifying validating resolvers that we are about to introduce a new key. By default, if not specified during creation time, this is set to the current time, meaning the key will be published as soon as **named** picks it up.
2. *Activate*: Sets the date on which the key is to be activated. After that date, the key will be included in the zone and used to sign it. By default, if not specified during creation time, this is set to the current time, meaning the key will be used to sign data as soon as **named** picks it up.
3. *Revoke*: Sets the date on which the key is to be revoked. After that date, the key will be flagged as revoked. It will be included in the zone and will be used to sign it. This is used to notify validating resolvers that this key is about to be removed or retired from the zone.
4. *Inactive*: Sets the date on which the key is to become inactive. After that date, the key will still be included in the zone, but it will not be used to sign it. This sets the "expiration" or "retire" date for a key.
5. *Delete*: Sets the date on which the key is to be deleted. After that date, the key will no longer be included in the zone, but it continues to exist on the file system or key repository.

You can set these metadata fields on a key pair as you've seen in Section 4.6.1 using commands such as **dnssec-keygen** or **dnssec-settime**.

Metadata	Included in Zone File?	Used to Sign Data?	Purpose
Publish	Yes	No	Introducing a key soon to be active
Activate	Yes	Yes	Activation date for new key
Revoke	Yes	Yes	Notifying a key soon to be retired
Inactive	Yes	No	Inactivate or retire a key
Delete	No	No	Deletion or removal of key from zone

Table 6.1: Key Metadata Comparison

6.4.2.1 Manual Key Management and Signing

This guide is intended as an introductory to DNSSEC deployment using BIND. Since BIND comes with a set of features to automate key loading and signing, that is the recommended configuration for readers of this document. In this section, we will only briefly cover tools and commands to manually load keys and manually sign zone data, but not go into great details.

The directive `auto-dnssec maintain` makes **named** check for new keys and load them automatically on an interval. If you wish to not automate this process, you could opt to change it to `auto-dnssec off`. This makes all key management manual, and to load new keys, you will need to execute the command **rndc loadkeys example.com**.

To manually sign the zone, first, you need to edit the zone file to make sure the proper DNSKEY entries are included in your zone file, then use the command **dnssec-signzone** as such:

```
# cd /etc/bind/keys/example.com/
# dnssec-signzone -A -t -N INCREMENT -o example.com -f /etc/bind/db/example.com.signed.db \
> /etc/bind/db/example.com.db Kexample.com.+008+17694.key Kexample.com.+008+06817.key
Verifying the zone using the following algorithms: RSASHA256.
Zone fully signed:
Algorithm: RSASHA256: KSKs: 1 active, 0 stand-by, 0 revoked
                    ZSKs: 1 active, 0 stand-by, 0 revoked
/etc/bind/db/example.com.signed.db
Signatures generated:          17
Signatures retained:          0
Signatures dropped:           0
```

```
Signatures successfully verified:      0
Signatures unsuccessfully verified:    0
Signing time in seconds:               0.046
Signatures per second:                364.634
Runtime in seconds:                   0.055
```

The `-o` switch explicitly defines the domain name (`example.com` in this case), `-f` switch specifies the output file name. The second line has 3 parameters, they are the unsigned zone name (`/etc/bind/db/example.com.db`), ZSK, and KSK file names. This generated a plain text file `/etc/bind/db/example.com.signed.db`, which you can verify for correctness.

Finally, you'll need to update `named.conf` to load the signed version of the zone, so it looks something like this:

```
zone "example.com" IN {
    type master;
    file "db/example.com.signed.db";
};
```

You will need to re-sign periodically as well as every time the zone data changes.

6.4.3 How Long Do I Wait Before Deleting Old Data?

If you have followed the examples described in Section 4.1 and Section 4.6, old keys and old signatures are automatically removed from the zone.

However, it is still good to discuss why bother waiting a period of time before removing any old data, be it DNSKEY or RRSIG. The goal here is to not "orphan" anyone out there who may be getting a cached answer, and the information to verify that cached answer doesn't exist anymore. For example, if you chose to pre-publish your ZSK, but did not wait long enough before removing the old ZSK, you're running with the risk that there may be users out there receiving the old RRSIG out of cache, but they are unable to verify the cached old RRSIG because the old ZSK has already been removed. To these users, the domain names would fail validation, until the cached RRSIG entries expire, and their validating resolver retrieves the new RRSIG signed by the new ZSK.

In Section 4.6, we recommended using a very generic and easy to remember 30 days as the amount of time to wait, partly because 30 days is also the default validity time for RRSIG. If you decide to manage your own zone signing and record removing, you should wait at least the duration of your old record's TTL before attempting to removing it from the zone. It is probably better to err on the safe side and leave the old data in the zone a little longer.

Note

One of the worst things to happen during a rollover is to "orphan" old keys, by deleting it too soon from the zone. This will result in the world's recursive servers come asking "hey, do you have the key(s) for these older signatures?" and not get an answer to satisfy their needs. The recursive servers would have to fail the validation, and the users may think you zone has been compromised because the keys and signatures to match up. This is why it is a good idea to wait a period of time in-between each phase of the key rollover, to ensure that not only new information has propagated to the world, but also that old information that was previously published to the world have expired from whoever has been caching them. This is why in this document we have chosen a very conservative period of 30 days. If you have unusually long TTL or signature expirations, it may be wise to change the rollover schedule accordingly.

6.4.4 Emergency Key Rollovers

Keys are generally rolled at a regular schedule (that is, if you choose to roll them at all). But sometimes, you may have to rollover keys out-of-schedule due to a security incident. The aim of an emergency rollover is re-sign the zone with a new key as soon as possible, because when a key is suspected of being compromised, the malicious attacker (or anyone who has access to the key) could impersonate you, and trick other validating resolvers into believing that they are receiving authentic, validated answers.

During an emergency rollover, you would follow the same operational procedures as described in Section 7.2, with the added task of reducing the TTL of current active (possibly compromised) DNSKEY RRset, in attempt to phase out the compromised

key faster before the new key takes effect. The time frame should be significantly reduced from the 30-days-apart example, since you probably don't want to wait up to 60 days for the compromised key to be removed from your zone.

Another method is to always carry a spare key with you at all times. You could always have a second key (pre)published (and hopefully this one was not compromised the same time as the first key), so if the active key is compromised, you could save yourself some time to immediately activate the spare key, and all the validating resolvers should already have this spare key cached, thus saving you some time.

With KSK emergency rollover, you would have to also consider factors related to your parent zone, such as how quickly they can remove the old DS record and published the new ones.

As usual, there is a lot more to consider when it comes to emergency key rollovers. For more in-depth considerations, please check out [RFC 7583](#).

6.4.5 DNSKEY Algorithm Rollovers

From time to time new digital signature algorithms with improved security are introduced, and it may be desirable for administrators to roll over DNSKEYs to a new algorithm, e.g. from RSASHA1 (algorithm 5 or 7) to RSASHA256 (algorithm 8). The algorithm rollover must be done with care in a stepwise fashion to avoid breaking DNSSEC validation.

As with other DNSKEY rollovers, when the zone is of type master, an algorithm rollover can be accomplished using dynamic updates or automatic key rollovers. For zones of type slave, only automatic key rollovers are possible, but the **dnssec-settime** utility can be used to control the timing of such.

In any case the first step is to put DNSKEYs using the new algorithm in place. You must generate the `K*` files for the new algorithm and put them in the zone's key directory where **named** can access them. Take care to set appropriate ownership and permissions on the keys. If the `auto-dnssec` zone option is set to `maintain`, **named** will automatically sign the zone with the new keys based on their timing metadata when the `dnssec-loadkeys-interval` elapses or you issue the **rndc loadkeys** command. Otherwise for zones of type master, you can use **nsupdate** to add the new DNSKEYs to the zone. This will cause **named** to use them to sign the zone. For zones of type slave, e.g. on a bump-in-the-wire inline signing server, **nsupdate** cannot be used.

Once the zone has been signed by the new DNSKEYs, you must inform the parent zone and any trust anchor repositories of the new KSKs, e.g. you might place DS records in the parent zone through your DNS registrar's website.

Before starting to remove the old algorithm from a zone, you must allow the maximum TTL on its DS records in the parent zone to expire. This will assure that any subsequent queries will retrieve the new DS records for the new algorithm. After the TTL has expired, you can remove the DS records for the old algorithm from the parent zone and any trust anchor repositories. You must then allow another maximum TTL interval to elapse so that the old DS records disappear from all resolver caches.

The next step is to remove the DNSKEYs using the old algorithm from your zone. Again this can be accomplished using **nsupdate** to delete the old DNSKEYs (master zones only) or by automatic key rollover when `auto-dnssec` is set to `maintain`. You can cause the automatic key rollover to take place immediately by using the **dnssec-settime** utility to set the *Delete* date on all keys to any time in the past. (See **dnssec-settime -D <date/offset>** option.)

After adjusting the timing metadata, the **rndc loadkeys** command will cause **named** to remove the DNSKEYs and RRSIGs for the old algorithm from the zone. Note also that with the **nsupdate** method, removing the DNSKEYs also causes **named** to remove the associated RRSIGs automatically.

Once you have verified that the old DNSKEYs and RRSIGs have been removed from the zone, the final (optional) step is to remove the key files for the old algorithm from the key directory.

6.5 Other Topics

6.5.1 DNSSEC and Dynamic Updates

Dynamic DNS (DDNS) actually is independent of DNSSEC. DDNS provides a mechanism other than editing the zone file or zone database, to edit DNS data. Most clients and DNS servers have the capability to handle dynamic updates, and DDNS can also be integrated as part of your DHCP environment.

When you have both DNSSEC and dynamic updates in your environment, updating zone data works the same way as with traditional (insecure) DNS: you can use **rndc freeze** before editing the zone file, and **rndc thaw** when you have finished editing, or you could use the command **nsupdate** to add, edit, or remove records like this:

```
$ nsupdate
> server 192.168.1.13
> update add xyz.example.com. 300 IN A 1.1.1.1
> send
> quit
```

The examples provided in this guide will make **named** automatically re-sign the zone whenever its content has changed. If you decide to sign your own zone file manually, you will need to remember to execute the **dnssec-signzone** whenever your zone file has been updated.

As far as system resources and performance is concerned, be mindful that when you have a DNSSEC zone that changes frequently, every time the zone changes, your system is executing a series of cryptographic operations to (re)generate signatures and NSEC or NSEC3 records.

6.5.2 DNSSEC and Inline Signing

ISC introduces the "inline-signing" option with the release of BIND 9.9, which allows **named** to sign zones completely transparently. **named** does this by automatically creating an internal version of the zone that is signed on the fly, and only the signed version of the zone is presented to queries. The unsigned version of the zone file is untouched on the file system, but not served.

This feature simplifies DNSSEC deployment, below are two common scenarios of how this feature can be used:

1. *You administer the master server, with zone transfers to other slave servers:* This is what most examples in this guide describes, you have control over the master server, you can follow the instruction in Section 4.1 to generate key pairs and modify **named.conf**. A signed version of the zone is either generated on the fly by inline-signing, and zone transfers will take care of synchronizing the signed zone data to all slave name servers.
2. *You cannot easily modify the master sever configuration, but still would like all slave servers to get DNSSEC zones:* you can setup a "middle box" that all slave name servers transfer data from, and your middle box gets its zone from the master server. You do not need to modify the master name server configuration at all, on the middle box, set it up to have inline signing enabled, whenever your middle box receives zone transfer (unsigned) from the master server, a signed version is generated on the fly, and this is the version that will be transferred out to the other slave name servers.

For more details and configuration examples on Inline Signing, please see Section 7.1.

6.5.3 DNSSEC Look-aside Validation (DLV)

DNSSEC Look-aside Validation (DLV) is an extension to the DNSSEC protocol. It was designed to assist in early DNSSEC adoption by simplifying the configuration of recursive servers and lessen the burden of key management for the administrators. Without DLV, in the absence of a fully signed path from root to a zone, administrators wishing to enable DNSSEC validation would have to configure and maintain multiple trust anchors or managed keys in their configuration.

DLV removes the need for manual key management by identifying a trusted repository through which those keys can be securely retrieved by the validating resolver when it needs them. Basically, someone else (in this case, ISC) is performing the tedious task of trust anchor management, and your validating resolver just needs to trust that that someone else is doing a good job maintaining these trust anchors for you.

To enable DLV on your validating resolver, place this line in your configuration file and reload **named**:

```
dnssec-lookaside auto;
```

**Warning**

DNSSEC deployment has matured to a stage where most top level domains are signed and more and more registrars are DNSSEC-ready. If you are currently relying on DLV service provided by ISC, be aware that the service will not be available forever. ISC stopped accepting new zones into the DLV registry in July 2016 and plans to discontinue the DLV service in early 2017.

6.5.4 DNSSEC on Private Networks

Before we discuss DNSSEC on private networks, let's clarify what we mean by private networks. In this section, private networks really refers to a private or internal DNS view. Most DNS products offer the ability to have different version of DNS answers, depending on the origin of the query. This feature is often called DNS views or split DNS, and is most commonly implemented as an "internal" versus an "external" setup.

For instance, your organization may have a version of `example.com` that is offered to the world, and its names most likely resolves to publicly reachable IP addresses. You may also have an internal version of `example.com` that is only accessible when you are on the company's private networks or via a VPN connection. These private networks typical fall under 10.0.0.0/8, 172.16.0.0/12, or 192.168.0.0/16 for IPv4.

So what if you want to offer DNSSEC for your internal version of `example.com`? This is actually a more involving question, and we can only cover this topic briefly in this document.

Deploying DNSSEC in this context is possible. Because private networks are usually trusted, there may be less need to worry about someone hijacking your DNS traffic. This is commonly known as the "last mile" in DNS delivery. If you wish to deploy DNSSEC on your private networks, here are some scenarios to consider:

1. If your name server is configured as both the validating resolver and the internal authoritative server, the answers returned to your clients will not be validated at all. This is because the answer is coming directly from the authoritative server, thus the Authoritative Answer (aa) bit is set and, by definition, it is not validated. What this means is, to a regular client making the query, the secure authoritative answer looks exactly the same as the insecure authoritative answer.

It is technically possible to achieve validation and authoritative serving within a single instance of BIND by using separate views for each, however the details of doing so are outside the scope of this document. There is a good example of this technique for serving a locally validatable copy of the root zone in Appendix A of [RFC 7706](#).

2. If you have two name server instances running, one acting as the validating resolver, and one acting as the internal authoritative name server, it is possible to actually validate answers, provided that you have installed the trust anchor(s) necessary for your internal zones on the validating resolver. In this setup, the client gets back the Authenticated Data (ad) bit when querying against the validating resolver. If no trust anchors are installed, your validating resolver will go out to root, and attempt to validate internal answers against external authorities (and fail).
3. DNSSEC is designed to protect the communication between the client and the nameserver, however there are few applications or stub resolver libraries that take advantage of this. DNSSEC can help with last mile security in a managed environment, by deploying validating resolvers (such as BIND) on client machines.

In addition, early efforts have concentrated on getting DNSSEC deployed between authoritative servers and recursive servers as that is a prerequisite for working DNSSEC between the recursive server and the application. These efforts also provide a degree of protection for applications that are not DNSSEC-aware.

6.5.5 Introduction to DANE

With your DNS infrastructure now secured with DNSSEC, information can now be stored in DNS and its integrity and authenticity can be proved. One of the new features that takes advantage of this is the DNS-Based Authentication of Named Entities, or DANE. Below is a list of features currently being developed and tested by the DANE community:

- Use DANE as a verification mechanism to verify SSL/TLS certificates received over HTTPS for added security. You can see a live demonstration here: <http://dane.verisignlabs.com/>.

- Store self-signed X.509 certificates, bypass having to pay a third party (such as a Certificate Authority) to sign the certificates.
- Integrate with Mail Transfer Agents (MTA) to provide seamless email encryption.

DANE is an exciting area for DNS administrators. If you would like to learn more about the standards being proposed or new features being discussed, check out the DANE working group: <https://datatracker.ietf.org/wg/dane/charter/>. You can also check out Section 7.5.

6.6 Disadvantages of DNSSEC

DNSSEC, like many things in this world, is not without its own problems. Below are a few challenges and disadvantages that DNSSEC faces.

1. *Increased, well, everything:* With DNSSEC, signed zones are larger, thus taking up more disk space; for DNSSEC-aware servers, the additional cryptographic computation usually results in increased system load; and the network packets are bigger, possibly putting more strains on the network infrastructure.
2. *Different security considerations:* DNSSEC addresses many security concerns, most notably cache poisoning. But at the same time, it may introduce a set of different security considerations, such as amplification attack and zone enumeration through NSEC. These new concerns are still being identified and addressed by the Internet community.
3. *More complexity:* If you have read this far, you probably already concluded this yourself. With additional resource records, keys, signatures, rotations, DNSSEC adds a lot more moving pieces on top of the existing DNS machine. The job of the DNS administrator changes, as DNS becomes the new secure repository of everything from spam avoidance to encryption keys, and the amount of work involved to troubleshoot a DNS-related issue becomes more challenging.
4. *Increased fragility:* The increased complexity means more opportunities for things to go wrong. In the absence of DNSSEC, DNS was essentially "add something to the zone and forget". With DNSSEC, each new component - re-signing, key rollover, interaction with parent zone, key management - adds more scope for error. It is entirely possible that the failure to validate a name is down to errors on the part of one or more zone operators rather than the result of a deliberate attack on the DNS.
5. *New maintenance tasks:* Even if your new secure DNS infrastructure runs without any hiccups or security breaches, it still requires regular attention, from re-signing to key rollovers. While most of these can be automated, some of the tasks, such as KSK rollover, remain manual for the time being.
6. *Not enough people are using it today:* while it's estimated as of late 2016, that roughly 28% of the global Internet DNS traffic is validating², that doesn't mean that many of the DNS zones are actually signed. What this means is, if you signed your company's zone today, only less than 30% of the Internet users are taking advantage of this extra security. It gets worse: with less than 1% of the .com domains signed, if you enabled DNSSEC validation today, it's not likely to buy you or your users a whole lot more protection until these popular domains names decide to sign their zones.

The last point may have more impact than you realize. Consider this: HTTP and HTTPS traffic make up majority of the web. While you may have secured your DNS infrastructure through DNSSEC, if your web hosting is outsourced to a third party that does not yet support DNSSEC in their own domain, or if your web page loads contents and components from insecure domains, the end users may experience validation problems when trying to access your web page. For example, although I may have signed the zone `isc.org`, but my web address `www.isc.org` is actually a CNAME to `96d719dc5612761de516fc.random-cloud-provider.com`. As long as `random-cloud-provider.com` remains an insecure DNS zone, users cannot fully validate everything when they visit my web page and could be redirected elsewhere by a cache poisoning attack.

² based on APNIC statistics at <http://stats.labs.apnic.net/dnssec/XA>

Chapter 7

Recipes

This chapter provides step-by-step examples of some common configurations.

7.1 Inline Signing Recipes

There are two recipes here, the first shows an example of using inline signing on the master server, which is what we have covered in this guide thus far; the second example shows how to setup a "bump in the wire" between the hidden master and the slave servers to seamlessly sign the zone on the fly.

7.1.1 Master Server Inline Signing Recipe

In this recipe, our servers are illustrated as shown in Figure 7.1: we have a master server 192.168.1.1 and three slave servers (192.168.1.2, 192.168.1.3, and 192.168.1.4) that receive zone transfers. In order to get the zone signed, we need to reconfigure the master server, as described in Section 4.1. Once reconfigured, a signed version of the zone is generated on the fly by inline-signing, and zone transfers will take care of synchronizing the signed zone data to all slave name servers, without configuration or software changes on the slave servers.

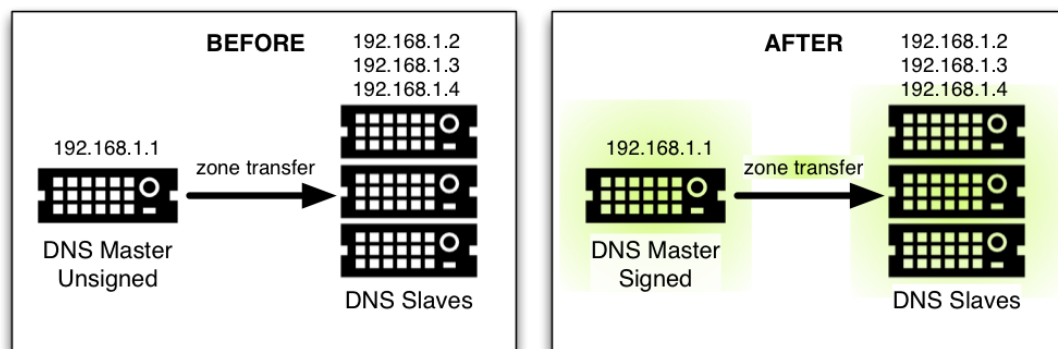


Figure 7.1: Inline Signing Recipe #1

Below is what the `named.conf` looks like on the master server, 192.168.1.1:

```
zone "example.com" IN {
    type master;
    file "db/example.com.db";
    key-directory "keys/example.com";
```



```
inline-signing yes;
auto-dnssec maintain;
allow-transfer { 192.168.1.2; 192.168.1.3; 192.168.1.4; };
};
```

On the slave servers, `named.conf` does not need to be updated, and it looks like this:

```
zone "example.com" IN {
    type slave;
    file "db/example.com.db";
    masters { 192.168.1.1; };
};
```

In fact, the slave servers do not even need to be running BIND, it could be running any other DNS product that has DNSSEC support.

7.1.2 "Bump in the Wire" Inline Signing Recipe

In this recipe, we are taking advantage of the power of inline signing by placing an additional name server 192.168.1.5 between the hidden master (192.168.1.1) and the DNS slaves (192.168.1.2, 192.168.1.3, and 192.168.1.4). The additional name server 192.168.1.5 acts as a "bump in the wire", taking unsigned zone from the hidden master on one end, and sending out signed data on the other end to the slave name servers. The steps described in this recipe may be used as part of the DNSSEC deployment strategy, since it requires minimal changes made to the existing hidden DNS master and DNS slaves.

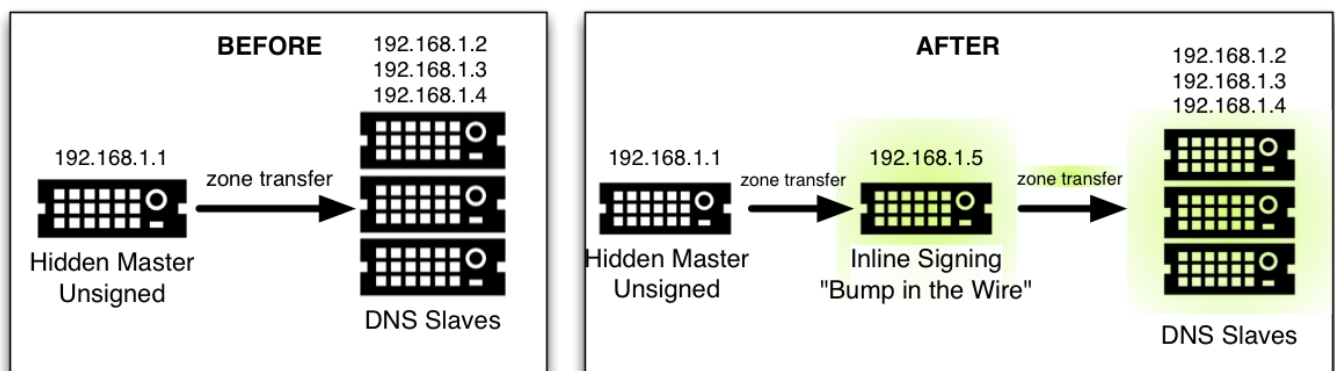


Figure 7.2: Inline Signing Scenario #2

It is important to remember that 192.168.1.1 in this case is a hidden master not exposed to the world, it must not be listed in the NS RRset. Otherwise the world will get conflicting answers, unsigned answers from the hidden master, and signed answers from the other name servers.

The only configuration change needed on the hidden master 192.168.1.1 is to make sure it allows our middle box to perform a zone transfer:

```
zone "example.com" IN {
    ...
    allow-transfer { 192.168.1.5; };
    ...
};
```

On the middle box 192.168.1.5, all the tasks described in Section 4.1 still need to be performed, such as generating key pairs and uploading information to parent zone. This server is configured as slave to the hidden master 192.168.1.1, receiving the unsigned data, and then using keys accessible to this middle box, sign data on the fly, and send out the signed data via zone transfer to the other three DNS slaves. Its `named.conf` looks like this:

```
zone example.com {
    type slave;
    masters { 192.168.1.1; };
    file "db/example.com.db";
    key-directory "keys/example.com";
    inline-signing yes;
    auto-dnssec maintain;
    allow-transfer { 192.168.1.2; 192.168.1.3; 192.168.1.4; };
};
```

Finally, on the three slave servers, configuration should be updated to receive zone transfer from 192.168.1.5 (middle box) instead of 192.168.1.1 (hidden master). If using BIND, the `named.conf` looks like this:

```
zone "example.com" IN {
    type slave;
    file "db/example.com.db";
    masters { 192.168.1.5; }; # this was 192.168.1.1 before!
};
```

7.2 Rollover Recipes

7.2.1 ZSK Rollover Recipe

This recipe covers how to perform a ZSK rollover using what is known as the Pre-Publication method. For other ZSK rolling methods, please see Section 6.4.1.1 in Chapter 6.

Below is the timeline for a ZSK rollover to occur on January 1st, 2017:

1. December 1st, 2016, a month before rollover
 - Generate new ZSK
 - Add DNSKEY for new ZSK to zone
2. January 1st, 2017, day of rollover
 - New ZSK used to replace RRSIGs for the bulk of the zone
3. February 1st, 2017
 - Remove old ZSK DNSKEY RRset from zone
 - DNSKEY signatures made with KSK are changed

The current active ZSK has the ID 17694 in this example. For more information on key management (such as what inactive date is, and why 30 days for example), please see Section 6.4.

7.2.1.1 One Month Before ZSK Rollover

On December 1st, 2016, a month before the planned rollover, you should change the parameters on the current key (17694) to become inactive on January 1st, 2017, and be deleted from the zone on February 1st, 2017, as well as generate a successor key (51623):

```
# cd /etc/bind/keys/example.com/
# dnssec-settime -I 20170101 -D 20170201 Kexample.com.+008+17694
./Kexample.com.+008+17694.key
./Kexample.com.+008+17694.private
# dnssec-keygen -S Kexample.com.+008+17694
Generating key pair..+++++ .....+++++
Kexample.com.+008+51623
```

The first command gets us into the key directory `/etc/bind/keys/example.com/`, where keys for `example.com` are stored.

The second **dnssec-settime** sets an inactive (-I) date of January 1st, 2017, and a deletion (-D) date of February 1st, 2017 for the current ZSK (Kexample.com.+008+17694).

Then the third command **dnssec-keygen** creates a successor key, using the exact same parameters (algorithms, key sizes, etc.) as the current ZSK. The new ZSK created in our example is Kexample.com.+008+51623.

Don't forget to make sure the successor keys are readable by **named**.

You can see in **named**'s logging messages informing you when the next key checking event is scheduled to occur, the frequency of which can be controlled by `dnssec-loadkeys-interval`. The log message looks like this:

```
zone example.com/IN (signed): next key event: 01-Dec-2016 00:13:05.385
```

And you can check the publish date of the key by looking at the key file:

```
# cd /etc/bind/keys/example.com
# cat Kexample.com.+008+51623.key
; This is a zone-signing key, keyid 11623, for example.com.
; Created: 20161130160024 (Mon Dec 1 00:00:24 2016)
; Publish: 20161202000000 (Fri Dec 2 08:00:00 2016)
; Activate: 20170101000000 (Sun Jan 1 08:00:00 2017)
...
```

Since the publish date is set to the morning of December 2nd, the next morning you will notice that your zone has gained a new DNSKEY record, but the new ZSK is not yet being used to generate signatures. Below is the abbreviated output with shortened DNSKEY and RRSIG when querying the authoritative name server, 192.168.1.13:

```
$ dig @192.168.1.13 example.com. DNSKEY +dnssec +multiline
...
;; ANSWER SECTION:
example.com.      600 IN DNSKEY 257 3 8 (
    AwEAAcWDps...lM3NRn/G/R
    ) ; KSK; alg = RSASHA256; key id = 6817
example.com.      600 IN DNSKEY 256 3 8 (
    AwEAAbi6Vo...qBW5+iAqNz
    ) ; ZSK; alg = RSASHA256; key id = 51623
example.com.      600 IN DNSKEY 256 3 8 (
    AwEAAcjGaU...0rzuu55If5
    ) ; ZSK; alg = RSASHA256; key id = 17694
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
    20170101000000 20161201230000 6817 example.com.
    LAiaJM26T7...FU9syh/TQ= )
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
    20170101000000 20161201230000 17694 example.com.
    HK4EBbbOpj...n5V6nvAkI= )
...
```

And for good measures, let's take a look at the SOA record and its signature for this zone. Notice the RRSIG is signed by the current ZSK 17694. This will come in handy later when you want to verify whether or not the new ZSK is in effect:

```
$ dig @192.168.1.13 example.com. SOA +dnssec +multiline
...
;; ANSWER SECTION:
example.com.      600 IN SOA nsl.example.com. admin.example.com. (
    2016120102 ; serial
    1800      ; refresh (30 minutes)
    900      ; retry (15 minutes)
    2419200   ; expire (4 weeks)
```

```

        300          ; minimum (5 minutes)
    )
example.com.      600 IN RRSIG SOA 8 2 600 (
    20161230160109 20161130150109 17694 example.com.
    YUTC8rFULaWbW+nAHzbfGwNqzARHevpryzRIJMvZBYPo
    NAeejNk9saNAoCYKwXGJ0YBc2k+r5fYq1Mg4112JkBF5
    buAsAYLw8vEOIxVpXwlArY+oSp9T1w2wfTZ0vhVIxaYX
    6dkcz4I3wbDx2xmG0yngtA6A8lAcherx2EGy0RM= )

```

These are all the manual tasks you need to perform for a ZSK rollover. If you have followed the configuration examples in this guide of using `inline-signing` and `auto-dnssec`, everything else is automated for you.

7.2.1.2 Day of ZSK Rollover

On the actual day of the rollover, although there is technically nothing for you to do, you should still keep an eye on the zone to make sure new signatures are being generated by the new ZSK (51623 in this example). The easiest way is to query the authoritative name server 192.168.1.13 for the SOA record like you did a month ago:

```

$ dig @192.168.1.13 example.com. SOA +dnssec +multiline

...
;; ANSWER SECTION:
example.com.      600 IN SOA nsl.example.com. admin.example.com. (
    2016112011 ; serial
    1800       ; refresh (30 minutes)
    900        ; retry (15 minutes)
    2419200    ; expire (4 weeks)
    300        ; minimum (5 minutes)
    )
example.com.      600 IN RRSIG SOA 8 2 600 (
    20170131000000 20161231230000 51623 example.com.
    J4RMNpJPOmMidElyBugJp0RLqXoNqfvo/2AT6yAAvx9X
    zZRL1cuhkRcyCSLZ9Z+zZ2y4u2lvQGrNiondaKdQCor7
    uTqH5WCPoqa1OCBjqU7c7v1AM27O9RD11nzPNpVQ7xPs
    y5nkGqf83OXTK26IfnjU1jq1UKSzg6QR7+XpLk0= )
...

```

As you can see, the signature generated by the old ZSK (17694) disappeared, replaced by a new signature generated from the new ZSK (51623).

Life Time of the Signatures

Not all signatures will disappear magically on the same day, depending on when each one is generated. Worst case scenario is that a new signature could have been signed by the old ZSK (17695) moments before it was deactivated, thus the signature could live for almost 30 more days, all the way up to right before February 1st.

This is why it is important that you should keep the old ZSK in the zone for a little bit longer and not delete it right away.

7.2.1.3 One Month After ZSK Rollover

Again, technically there should be nothing you need to do on this day, but it doesn't hurt to verify that the old ZSK (17694) is now completely gone from your zone. `named` will not touch `Kexample.com.+008+17694.private` and `Kexample.com.+008+17694.key` on your file system. Running the same `dig` command for DNSKEY should suffice:

```

$ dig @192.168.1.13 example.com. DNSKEY +multiline +dnssec

...
;; ANSWER SECTION:
example.com.      600 IN DNSKEY 257 3 8 (

```

```

        AwEAAcWDps...lM3NRn/G/R
        ) ; KSK; alg = RSASHA256; key id = 6817
example.com.      600 IN DNSKEY 256 3 8 (
        AwEAAdeCGr...lDnEfX+Xzn
        ) ; ZSK; alg = RSASHA256; key id = 51623
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
        20170203000000 20170102230000 6817 example.com.
        KHY8P0zE21...Y3szrmjAM= )
example.com.      600 IN RRSIG DNSKEY 8 2 600 (
        20170203000000 20170102230000 51623 example.com.
        G2g3crN17h...Oe4gw6gH8= )
...

```

Congratulations, the ZSK rollover is complete! As for the actual key files (the `.key` and `.private` files), they may be deleted at this point, but it's not required.

7.2.2 KSK Rollover Recipe

This recipe describes how to perform KSK rollover using the Double-DS method. For other KSK rolling methods, please see Section 6.4.1.2 in Chapter 6. The registrar used in this recipe is *GoDaddy*. Also for this recipe, we are keeping the number of DS records down to just one per active set using just SHA-1, for the sake of better clarity, although in practice most zone operators choose to upload 2 DS records as we have shown in Section 4.4. For more information on key management (such as what inactive date is, and why 30 days for example), please see Section 6.4.

Below is the timeline for a KSK rollover to occur on January 1st, 2017:

1. December 1st, 2016, a month before rollover
 - Change timer on the current KSK
 - Generate new KSK and DS records
 - Add DNSKEY for the new KSK to zone
 - Upload new DS records to parent zone
2. January 1st, 2017, day of rollover
 - Use the new KSK to sign all DNSKEY RRset, this generates new RRSIGs
 - Add new RRSIGs to the zone
 - Remove RRSIG for the old ZSK from zone
 - Start using the new KSK to sign DNSKEY
3. February 1st, 2017
 - Remove the old KSK DNSKEY from zone
 - Remove old DS records from parent zone

The current active KSK has the ID 24828, and this is the DS record that has already been published by the parent zone:

```

# dnssec-dsfromkey -a SHA-1 Kexample.com.+007+24828.key
example.com. IN DS 24828 7 1 D4A33E8DD550A9567B4C4971A34AD6C4B80A6AD3

```

7.2.2.1 One Month Before KSK Rollover

On December 1st, 2016, a month before the planned rollover, you should change the parameters on the current key to become inactive on January 1st, 2017, and be deleted from the zone on February 1st, 2017, as well as generate a successor key (23550). Finally, you should generate a new DS record based on the new key 23550:

```
# cd /etc/bind/keys/example.com/
# dnssec-settime -I 20170101 -D 20170201 Kexample.com.+007+24828
./Kexample.com.+007+24848.key
./Kexample.com.+007+24848.private
# dnssec-keygen -S Kexample.com.+007+24848
Generating key pair ←
.....++ ←
.....++
Kexample.com.+007+23550
# dnssec-dsfromkey -a SHA-1 Kexample.com.+007+23550.key
example.com. IN DS 23550 7 1 54FCF030AA1C79C0088FDEC1BD1C37DAA2E70DFB
```

The first command gets us into the key directory `/etc/bind/keys/example.com/`, where keys for `example.com` are stored.

The second `dnssec-settime` sets an inactive (-I) date of January 1st, 2017, and a deletion (-D) date of February 1st, 2017 for the current KSK (`Kexample.com.+007+24848`).

Then the third command `dnssec-keygen` creates a successor key, using the exact same parameters (algorithms, key sizes, etc.) as the current KSK. The new key pair created in our example is `Kexample.com.+007+23550`.

The fourth and final command `dnssec-dsfromkey` creates a DS record from the new KSK (23550), using SHA-1 as the digest type. Again, in practice most people generate two DS records for both supported digest types (SHA-1 and SHA-256), but for our example here we are only using one to keep the output small and hopefully clearer.

Don't forget to make sure the successor keys are readable by `named`.

You can see in syslog the messages informing you when the next key checking event is, and it looks like this:

```
zone example.com/IN (signed): next key event: 01-Dec-2016 00:13:05.385
```

And you can check the publish date of the key by looking at the key file:

```
# cd /etc/bind/keys/example.com
# cat Kexample.com.+007+23550.key
; This is a key-signing key, keyid 23550, for example.com.
; Created: 20161130160024 (Thu Dec 1 00:00:24 2016)
; Publish: 20161202000000 (Fri Dec 2 08:00:00 2016)
; Activate: 20170101000000 (Sun Jan 1 08:00:00 2017)
...
```

Since the publish date is set to the morning of December 2nd, the next morning you will notice that your zone has gained a new DNSKEY record based on your new KSK, but no corresponding RRSIG yet. Below is the abbreviated output with shortened DNSKEY and RRSIG when querying the authoritative name server, 192.168.1.13:

```
$ dig @192.168.1.13 example.com. DNSKEY +dnssec +multiline

...
;; ANSWER SECTION:
example.com. 300 IN DNSKEY 256 3 7 (
    AwEAAAdYqAc...TiSlrma6Ef
    ) ; ZSK; alg = NSEC3RSASHA1; key id = 29747
example.com. 300 IN DNSKEY 257 3 7 (
    AwEAAeTJ+w...O+Zy9j0m63
    ) ; KSK; alg = NSEC3RSASHA1; key id = 24828
example.com. 300 IN DNSKEY 257 3 7 (
    AwEAAc1BQN...Wdc0qoH21H
    ) ; KSK; alg = NSEC3RSASHA1; key id = 23550
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
    20161206125617 20161107115617 24828 example.com.
    4yliPVJOrK...aC3iF9vgc= )
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
```

```
20161206125617 20161107115617 29747 example.com.
g/gfmPjr+y...rt/S/xjPo= )
```

...

Any time after you have generated the DS record, you could upload it, you don't have to wait for the DNSKEY to be published in your zone, since this new KSK is not active yet. You could choose to do it immediately after the new DS record has been generated on December 1st, or you could wait until the next day after you have verified that the new DNSKEY record is added to the zone. Below are the screenshots from using GoDaddy's web-based interface to add a new DS record.

1. After logging in, click the green "Launch" button next to the domain name you want to manage.

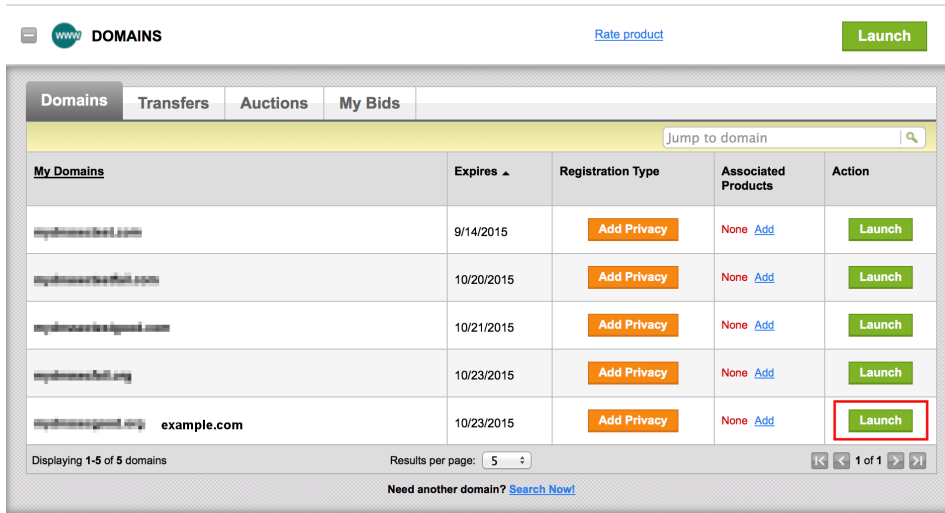


Figure 7.3: Upload DS Record Step #1

2. Scroll down to the "DS Records" section and click Manage.

Settings | DNS Zone File | Contacts

Domain Settings

Auto-Renew ⓘ **Standard:** On
Extended: Off
[Manage](#)

Lock ⓘ On
[Manage](#)

Nameservers ⓘ
Updated 10/24/2014
[Manage](#)

Forwarding ⓘ **Domain:** Off
[Manage](#)
Subdomain: 0 subdomains forwarded
[Manage](#)

Premium DNS ⓘ **Expires:** 10/21/2015
[Manage](#)
Secondary DNS: Off
[Manage](#)
DNSSEC: Unavailable
[Manage](#)
Vanity Nameservers: Off
[Manage](#)

DS Records ⓘ 1 DS record created
[Manage](#)

Figure 7.4: Upload DS Record Step #2

3. A dialog appears, displaying the current key (24828). Click "Add DS Record".

Manage DNSSEC DS Records ✕

MYDOMAIN.COM

Choose how to set up your DS records.

Key tag	Algorithm	Digest Type	Digest	Max Sig Life	Flags	Protocol	Key Data Alg	Public Key
24828	7	1	D4A33E8DD550A9...	1814400	N/A	N/A	N/A	

[Add DS Record](#)

Figure 7.5: Upload DS Record Step #3

4. Enter the Key ID, algorithm, digest type, and the digest, then click "Next".

1 Manage DS Records 2 Review DS Records

Single Bulk

Create DS Record

* Required

Key tag: * ⓘ Algorithm: * ⓘ Digest type: * ⓘ
23550 7 1

Digest: * ⓘ
54FCF030AA1C79C0088FDEC1BD1C37DAA2E70DFB

Max sig life: ⓘ Flags: ⓘ Protocol: ⓘ Key data alg: ⓘ
Select... Select... Select...

Public key: ⓘ

Cancel Back Next

Figure 7.6: Upload DS Record Step #4

5. Address any errors and click "Finish".

1 Manage DS Records 2 Review DS Records

Key tag	Algorithm	Digest Type	Digest	Max Sig Life	Flags	Protocol	Key Data Alg	Public Key	Error
23550	7	1	54FCF030A...	N/A	N/A	N/A	N/A		

Cancel Back Finish

Figure 7.7: Upload DS Record Step #5

6. Both DS records are shown. Click "Save".

Manage DNSSEC DS Records ✕

MYDNSSECGOOD.ORG

Choose how to set up your DS records.

Key tag	Algorithm	Digest Type	Digest	Max Sig Life	Flags	Protocol	Key Data Alg	Public Key
24828	7	1	D4A33E8DD550A9...	1814400	N/A	N/A	N/A	<input checked="" type="checkbox"/>
23550	7	1	54FCF030AA1C79...	N/A	N/A	N/A	N/A	<input checked="" type="checkbox"/>

Figure 7.8: Upload DS Record Step #6

Finally, let's verify that the registrar has published the new DS record. This may take anywhere from a few minutes to a few days, depending on your parent zone. You could verify whether or not your parent zone has published the new DS record by querying for the DS record of your zone. In the example below, the Google public DNS server 8.8.8.8 is used:

```
$ dig @8.8.8.8 example.com. DS
...
;; ANSWER SECTION:
example.com. 21552 IN DS 24828 7 1 D4A33E8DD550A9567B4C4971A34AD6C4B80A6AD3
example.com. 21552 IN DS 23550 7 1 54FCF030AA1C79C0088FDEC1BD1C37DAA2E70DFB
```

You could also query your parent zone's authoritative name servers directly to see if these records have been published. DS records will not show up on your own authoritative zone, so do not query your own name servers for them. In this recipe, the parent zone is `.com`, so querying a few of the `.com` name servers is another appropriate verification.

7.2.2.2 Day of KSK Rollover

If you have followed the examples in this document as described in Section 4.1, there is technically nothing you need to do manually on the actual day of the rollover. However, you should still keep an eye on the zone to make sure new signature(s) are being generated by the new KSK (23550 in this example). The easiest way is to query the authoritative name server 192.168.1.13 for the same DNSKEY and signatures like you did a month ago:

```
$ dig @192.168.1.13 example.com. DNSKEY +dnssec +multiline
...
;; ANSWER SECTION:
example.com. 300 IN DNSKEY 256 3 7 (
    AwEAAAdYqAc...TiSlrma6Ef
    ) ; ZSK; alg = NSEC3RSASHA1; key id = 29747
example.com. 300 IN DNSKEY 257 3 7 (
    AwEAAeTJ+w...O+Zy9j0m63
    ) ; KSK; alg = NSEC3RSASHA1; key id = 24828
example.com. 300 IN DNSKEY 257 3 7 (
    AwEAAc1BQN...Wdc0qoH21H
    ) ; KSK; alg = NSEC3RSASHA1; key id = 23550
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
    20170201074900 20170101064900 23550 mydnssecgood.org.
    S6zTbBTfvU...Ib5eXkbtE= )
```

```
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
    20170105074900 20161206064900 29747 mydnssecgood.org.
    VY5URQA2/d...OVKrl+KX8= )
...

```

As you can see, the signature generated by the old KSK (24828) disappeared, replaced by a new signature generated from the new KSK (23550).

7.2.2.3 One Month After KSK Rollover

While the removal of the old DNSKEY from zone should be automated by **named**, the removal of the DS record is manual. You should make sure the old DNSKEY record is gone from your zone first by querying for the DNSKEY records of the zone, and this time we expect to see one less DNSKEY, namely the key with ID of 24828:

```
$ dig @192.168.1.13 example.com. DNSKEY +dnssec +multiline

...
;; ANSWER SECTION:
example.com. 300 IN DNSKEY 256 3 7 (
    AwEAAAdYqAc...TiSlrma6Ef
    ) ; ZSK; alg = NSEC3RSASHA1; key id = 29747
example.com. 300 IN DNSKEY 257 3 7 (
    AwEAAAc1BQN...Wdc0qoH21H
    ) ; KSK; alg = NSEC3RSASHA1; key id = 23550
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
    20170208000000 20170105230000 23550 mydnssecgood.org.
    Qw9Em3dDok...bNCS7KISw= )
example.com. 300 IN RRSIG DNSKEY 7 2 300 (
    20170208000000 20170105230000 29747 mydnssecgood.org.
    OuelpIlpY9...XfsKupQgc= )
...

```

Now, we can remove the old DS record for key 24828 from our parent zone. Be careful to remove the correct DS record. If we accidentally removed the new DS record(s) of key ID 23550, it could lead to a problem called "security lameness", as discussed in Section 5.4.1, and may cause users unable to resolve any names in our zone.

1. After logging in and launched the domain, scroll down to the "DS Records" section and click Manage.

Domain Settings

Auto-Renew ⓘ **Standard:** On
Extended: Off
[Manage](#)

Lock ⓘ On
[Manage](#)

Nameservers ⓘ [REDACTED]
[REDACTED]
Updated 10/24/2014
[Manage](#)

Forwarding ⓘ **Domain:** Off
[Manage](#)
Subdomain: 0 subdomains forwarded
[Manage](#)

Premium DNS ⓘ **Expires:** 10/21/2015
[Manage](#)
Secondary DNS: Off
[Manage](#)
DNSSEC: Unavailable
[Manage](#)
Vanity Nameservers: Off
[Manage](#)

DS Records ⓘ 2 DS records created
[Manage](#)





Figure 7.9: Remove DS Record Step #1

2. A dialog appears, displaying both keys (24828 and 23550). Use the far right hand X button to remove the key 24828.

Manage DNSSEC DS Records ×

[REDACTED]

Choose how to set up your DS records.

Key tag	Algorithm	Digest Type	Digest	Max Sig Life	Flags	Protocol	Key Data Alg	Public Key	
23550	7	1	54FCF030AA1C79...	1814400	N/A	N/A	N/A		 
24828	7	1	D4A33E8DD550A9...	1814400	N/A	N/A	N/A		 

[Add DS Record](#)

Figure 7.10: Remove DS Record Step #2

3. Key 24828 now appears crossed out, click "Save" to complete the removal.

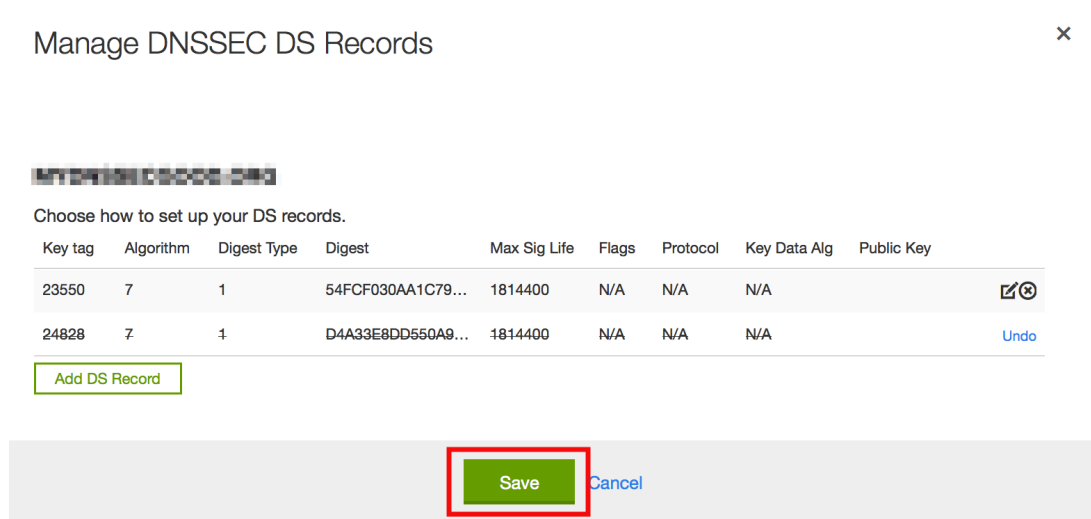


Figure 7.11: Remove DS Record Step #3

Congratulations, the KSK rollover is complete! As for the actual key files (the `.key` and `.private` files), they may be deleted at this point, but it's not required.

7.3 NSEC and NSEC3 Recipes

7.3.1 Migrating from NSEC to NSEC3

This recipe describes how to go from using NSEC to NSEC3, as described in both Section 4.5 and Section 6.2. This recipe assumes that the zones are already signed, and `named` is configured according to the steps described in Section 4.1.



Warning If your zone is signed with RSASHA1 (algorithm 5) you cannot migrate to NSEC3 without also performing an *algorithm rollover* to RSASHA1-NSEC3-SHA1 (algorithm 7) as described in Section 6.4.5. This ensures that older validating resolvers that don't understand NSEC3 will fallback to treating the zone as unsecured (rather than "bogus") as described in Section 2 of RFC 5155.

This command below enables NSEC3 for the zone `example.com`, using a pseudo-random string `1234567890abcdef` for its salt:

```
# rndc signing -nsec3param 1 0 10 1234567890abcdef example.com
```

You'll know it worked if you see the following log messages:

```
Oct 21 13:47:21 received control channel command 'signing -nsec3param 1 0 10 1234567890 ↵
  abcdef example.com'
Oct 21 13:47:21 zone example.com/IN (signed): zone_addnsec3chain(1,CREATE,10,1234567890 ↵
  ABCDEF)
```

You can also verify that this worked by querying for a name you know that does not exist, and check for the presence of the NSEC3 record, such as this:

```
$ dig @192.168.1.13 thereisnowaythisexists.example.com. A +dnssec +multiline
```

```
...
TOM10UQBL336NFAQB3P6MOO53LSVG8UI.example.com. 300 IN NSEC3 1 0 10 1234567890ABCDEF (
```

```
TQ9QBEGA6CROHEOC8KIH1A2C06IVQ5ER
NS SOA RRSIG DNSKEY NSEC3PARAM )
...
```

Our example used four parameters: 1, 0, 10, and 1234567890ABCDEF, in the order they appeared. 1 represents the algorithm, 0 represents the opt-out flag, 10 represents the number of iterations, and 1234567890abcdef is the salt. To learn more about each of these parameters, please see Section 6.2.2.1.

For example, to create an NSEC3 chain using the SHA-1 hash algorithm, no opt-out flag, 10 iterations, and a salt value of "FFFF", use:

```
# rndc signing -nsec3param 1 0 10 FFFF example.com
```

To set the opt-out flag, 15 iterations, and no salt, use:

```
# rndc signing -nsec3param 1 1 15 - example.com
```

7.3.2 Migrating from NSEC3 to NSEC

This recipe describes how to migrate from NSEC3 to NSEC.

Migrating from NSEC3 back to NSEC is easy, just use the **rndc** command like this:

```
$ rndc signing -nsec3param none example.com
```

You know that it worked if you see these messages in log:

```
named[14093]: received control channel command 'signing -nsec3param none example.com'
named[14093]: zone example.com/IN: zone_addnsec3chain(1,REMOVE,10,1234567890ABCDEF)
```

Of course, you can query for a name that you know that does not exist, and you should no longer see any traces of NSEC3 records.

```
$ dig @192.168.1.13 reieiergiuhewhiouwe.example.com. A +dnssec +multiline
...
example.com.      300 IN NSEC aaa.example.com. NS SOA RRSIG NSEC DNSKEY
...
ns1.example.com.  300 IN NSEC web.example.com. A RRSIG NSEC
...
```

7.3.3 Changing NSEC3 Salt Recipe

In Section 6.2.2.3, we've discussed the reasons why you may want to change your salt once in a while for better privacy. In this recipe, we will look at what command to execute to actually change the salt, and how to verify that it has been changed.

To change your NSEC3 salt to "fedcba0987654321", you may run the **rndc signing** command like this:

```
# rndc signing -nsec3param 1 1 10 fedcba0987654321 example.com
```

You should see the following messages in log, assuming your old salt was "1234567890abcdef":

```
named[15848]: zone example.com/IN: zone_addnsec3chain(1,REMOVE,10,1234567890ABCDEF)
named[15848]: zone example.com/IN: zone_addnsec3chain(1,CREATE|OPTOUT,10,FEDCBA0987654321)
```

You can of course, try to query the name server (192.168.1.13 in our example) for a name that does not exist, and check the NSEC3 record returned:

```
$ dig @192.168.1.13 thereisnowaythisexists.example.com. A +dnssec +multiline
...
TOM10UQBL336NFAQB3P6MO053LSVG8UI.example.com. 300 IN NSEC3 1 0 10 FEDCBA0987654321 (
    TQ9QBEGA6CROHEOC8KIH1A2C06IVQ5ER
    NS SOA RRSIG DNSKEY NSEC3PARAM )
...
```

Pseudo-Random Salt

You can use a pseudo-random source to create the salt for you. Here is an example on Linux to create a 16-character hex string:

```
# rndc signing -nsec3param 1 0 10 $(head -c 300 /dev/random | shasum | cut -b 1-16) <-
example.com
```

BIND 9.10 and newer provides the keyword “auto” which may be used in place of the salt field for **named** to generate a random salt.

7.3.4 NSEC3 Optout Recipe

This recipe discusses how to enable and disable NSEC3 opt-out, and show the results of each action. As discussed in Section 6.2.2.2, NSEC3 opt-out is a feature that can help conserve resources on parent zones that have many delegations that have yet been signed.

Before starting, for this recipe we will assume the zone `example.com` has the following 4 entries (for this example, it is not relevant what record types these entries are):

- `ns1.example.com`
- `ftp.example.com`
- `www.example.com`
- `web.example.com`

And the zone `example.com` has 5 delegations to 5 sub domains, only one of which is signed and has a valid DS RRset:

- `aaa.example.com`, not signed
- `bbb.example.com`, signed
- `ccc.example.com`, not signed
- `ddd.example.com`, not signed
- `eee.example.com`, not signed

Before enabling NSEC3 opt-out, the zone `example.com` contains ten NSEC3 records, below is the list with plain text name before the actual NSEC3 record:

- *aaa.example.com*: 9NE0VJGTRTMJOS171EC3EDL6I6GT4P1Q.example.com.
 - *bbb.example.com*: AESO0NT3N44OOSDQS3PSL0HACHUE1O0U.example.com.
 - *ccc.example.com*: SF3J3VR29LDDO3ONT1PM6HAPHV372F37.example.com.
 - *ddd.example.com*: TQ9QBEGA6CROHEOC8KIH1A2C06IVQ5ER.example.com.
-

- *eee.example.com*: L16L08NEH48IFQIEIPS1HNRMQ523MJ8G.example.com.
- *ftp.example.com*: JKMAVHL8V7EMCL8JHIEN8KBOAB0MGUK2.example.com.
- *ns1.example.com*: FSK5TK9964BNE7BPHN0QMMD68IUDKT8I.example.com.
- *web.example.com*: D65CIIG0GTRKQ26Q774DVMRCNHQO6F81.example.com.
- *www.example.com*: NTQ0CQEJHM0S17POMCUSLG5IOQQEDTBJ.example.com.
- *example.com*: TOM10UQBL336NFAQB3P6MOO53LSVG8UI.example.com.

We can enable NSEC3 opt-out with this command, changing the opt-out bit (the second parameter of the 4) from 0 to 1 (see Section 6.2.2.1 to review what each parameter is):

```
# rndc signing -nsec3param 1 1 10 1234567890abcdef example.com
```

After NSEC3 opt-out is enabled, the number of NSEC3 records is reduced. Notice that the unsigned delegations *aaa*, *ccc*, *ddd*, and *eee* now don't have corresponding NSEC3 records.

- *bbb.example.com*: AESO0NT3N44OOSDQS3PSL0HACHUE1O0U.example.com.
- *ftp.example.com*: JKMAVHL8V7EMCL8JHIEN8KBOAB0MGUK2.example.com.
- *ns1.example.com*: FSK5TK9964BNE7BPHN0QMMD68IUDKT8I.example.com.
- *web.example.com*: D65CIIG0GTRKQ26Q774DVMRCNHQO6F81.example.com.
- *www.example.com*: NTQ0CQEJHM0S17POMCUSLG5IOQQEDTBJ.example.com.
- *example.com*: TOM10UQBL336NFAQB3P6MOO53LSVG8UI.example.com.

To undo NSEC3 opt-out, run the same **rndc** command with the opt-out bit set to 0:

```
# rndc signing -nsec3param 1 0 10 1234567890abcdef example.com
```

nsec3hash

NSEC3 hashes the plain text domain name, and we can compute our own hashes using the tool **nsec3hash**. For example, to compute the hashed name for "www.example.com" using the parameters we listed above, we would execute the command like this:

```
# nsec3hash 1234567890ABCDEF 1 10 www.example.com.
NTQ0CQEJHM0S17POMCUSLG5IOQQEDTBJ (salt=1234567890ABCDEF, hash=1, iterations=10)
```

7.4 Reverting to Unsigned Recipe

This recipe describes how to revert from signed zone (DNSSEC) back to unsigned (DNS).

Whether or not the world thinks your zone is signed really comes down to the DS records hosted by your parent zone. If there are no DS records, the world thinks your zone is not signed. So reverting to unsigned is as easy as removing all DS records from the parent zone.

Below is an example of removing using *GoDaddy* web-based interface to remove all DS records.

1. After logging in, click the green "Launch" button next to the domain name you want to manage.

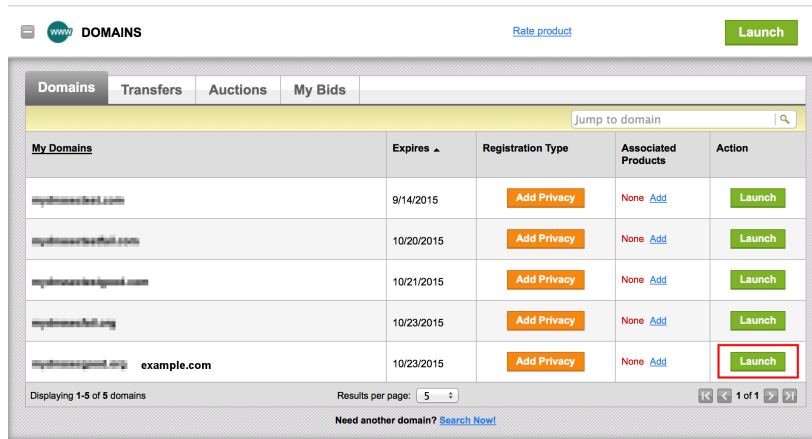


Figure 7.12: Revert to Unsigned Step #1

2. Scroll down to the "DS Records" section and click Manage.

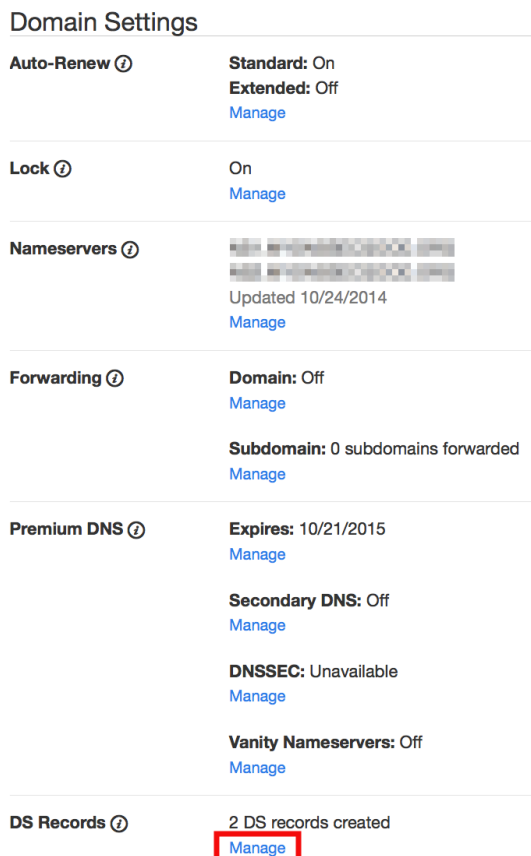


Figure 7.13: Revert to Unsigned Step #2

3. A dialog appears, displaying all current keys. Use the far right hand X button to remove each key.

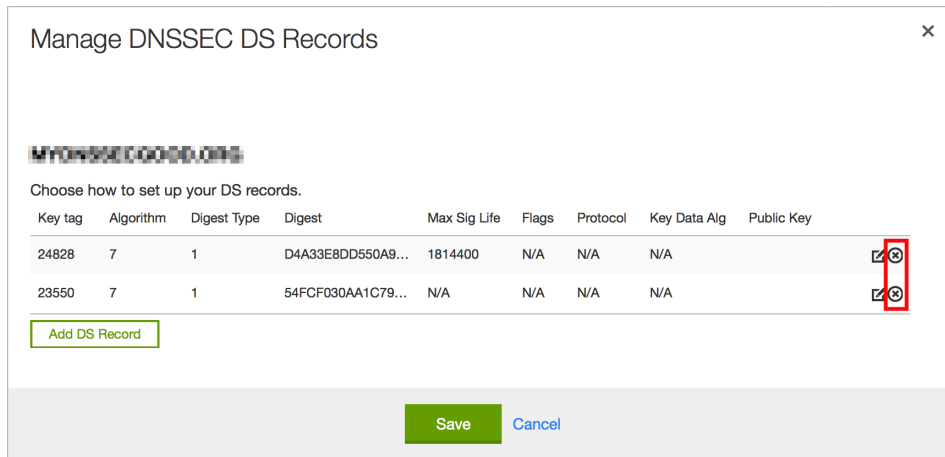


Figure 7.14: Revert to Unsigned Step #3

4. Click Save

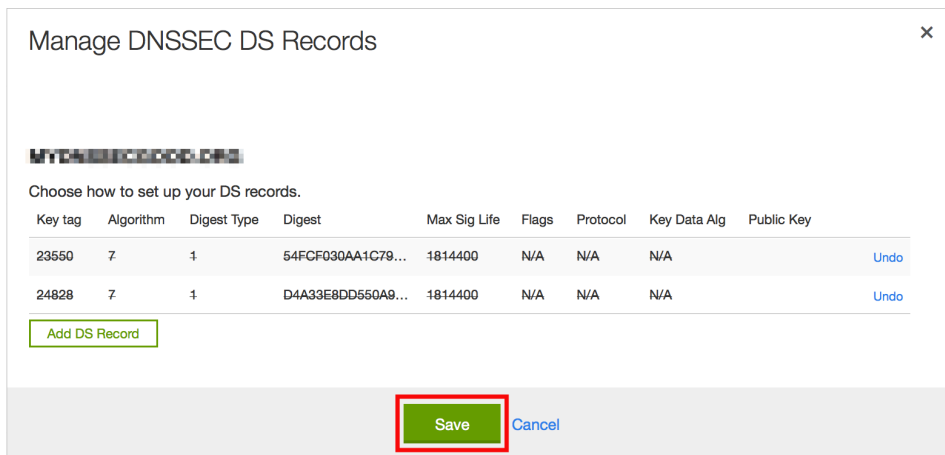


Figure 7.15: Revert to Unsigned Step #4

To be on the safe side, you should wait a while before actually deleting all signed data from your zone, just in case some validating resolvers out there have cached information. After you are certain that all cached information have expired (usually this means TTL has passed), you may reconfigure your zone. This is the `named.conf` when it is signed, with DNSSEC-related configurations in bold:

```
zone "example.com" IN {
    type master;
    file "db/example.com.db";
    key-directory "keys/example.com";
    inline-signing yes;
    auto-dnssec maintain;
    allow-transfer { any; };
};
```

Remove the 3 lines so your `named.conf` looks like this, then use `rndc reload` to reload the zone:

```
zone "example.com" IN {
    type master;
    file "db/example.com.db";
    allow-transfer { any; };
};
```

```
};
```

Your zone is now reverted back to the traditional, insecure DNS format.

7.5 Self-signed Certificate Recipe

This recipe describes how to configure DNS(SEC) to include a TLSA record that acts as a different channel to provide verification for a self-signed x509 (SSL) certificate.

Note

TLSA is still in an early stage of deployment. One of the road blocks that exists as of late 2016 is the lack of built-in application support in web browsers and mail applications. As these applications add support for TLSA, steps described in this recipe will become more relevant and practical. Today (as of late 2016), unfortunately, most applications lack native support and will likely result in some kind of error message or warning even if you have deployed TLSA correctly.

The [Wikipedia page for DANE](#) contains a list of current applications, libraries and plugins that provide TLSA support.

For this recipe, we are assuming that you already have a working web server configured with a self-signed x509 certificate. Although the steps described below works for self-signed certificates, it can also be used for "real" certificates that were signed by a Certificate Authority (usually a service you pay for). This is one of several possible uses of DNS-Based Authentication of Named Entities, or DANE (we briefly talked about DANE in Section 6.5.5).

First, let's take a look at the certificate used by you web server:

```
# cat server.crt
-----BEGIN CERTIFICATE-----
MIICVTCCAb4CCQChk/gPoAqkWjANBgkqhkiG9w0BAQsFADBvMQswCQYDVQQGEwJV
UzELMAkGA1UECAwCQ0ExFjAUBgNVBACMDVNhbiBGcmFuY2IzY28xITAfBgNVBAOM
GEludGVybV0IFdpZGdpdHMgUHR5IEExOZDEYMBYGA1UEAwPd3d3LmV4YW1wbGUu
Y29tMB4XDTE0MTIwODA2MDMxNFoXDTI0MTIwNTA2MDMxNFowbzELMAkGA1UEBhMC
VVMxZCZAJBgNVBAGMAkNBMRyYwFAyDVQQHDA1TYW4gRnJhbmNpc2NmMSEwHwYDVQQK
DBhJbnRlcm5ldCBXaWRnaXRzIFB0eSBMdGQxGDAWBgNVBAMMD3d3dy5leGFtcGxl
LmNvbTCBnzANBgkqhkiG9w0BAQEFAAOBjQAwGyKCGYEAzovVMAexPZHg8MaL2xfE
IfwPKxCdcCzF2eEv13euIk2esQ0r3GE+xEVqf/lggCC00H0q6TXs+6XFYvc4+O/m
LEh6DFnfn8Kz3T0d6mG218YuhhkLLrwugvvaAcHqMoVeZRPqFLh1faUsoGxb+CPs
3B8xYFisUqNjP6Tr26MhSVECAwEAATANBgkqhkiG9w0BAQsFAAOBgQC7lQbdSkaM
x8B6RIs+PMOZ14R1A1UcPNUPPAK0vK23/ie1SfxSHIw6PlOba+ZQZusrDRYooR3B
viM+cUnhD5UvhU4bn3ZP0cp+WNsimycf/gdfkAe47EmloVNZP6abUgqMPStongIB
7uonP6j74A/BTF5kdUsaDPoDfvGnCjZMsQ==
-----END CERTIFICATE-----
```

Next, use **openssl** to generate a SHA-256 fingerprint of this certificate, this is what you will list in DNS as a TLSA record. Also, you need to remove all the colons, hence the added **sed** at the end to filter out all ":" characters:

```
# openssl x509 -noout -fingerprint -sha256 < server.crt | tr -d :
SHA256 Fingerprint=294874DA378148CDD1B9C57D2E891E8C294D2958F0BCA7400A0D6D6F50C4A3BB
```

Now you can insert the TLSA record by editing the zone file the old fashioned way, or if your DNS server is allowing dynamic updates, you could use **nsupdate** like this to inject the TLSA record:

```
# nsupdate
> server localhost
> update add _443._tcp.www.example.com. 3600 IN TLSA 3 0 1 294874 ↵
    DA378148CDD1B9C57D2E891E8C294D2958F0BCA7400A0D6D6F50C4A3BB
> send
> quit
```

Let's talk briefly about the record you just added. The name is a specifically formed "_443._tcp.www.example.com", which specifies the usage of TCP port 443, for the name "www.example.com". It is followed by three parameters, each representing usage, selector, and matching type. For this recipe, we will not dissect into all the possible combinations of these parameters. The examples listed here are 3, 0, and 1, which represent:

- Usage: 3 = self-signed certificate
- Selector 0 = full certificate is included
- Matching Type 1 = SHA-256

If you are interested in learning (a lot) more about the TLSA record type, check out *"A Step-by-Step guide for implementing DANE with a Proof of Concept"* by *Sandoche Balakrishnan, Stephane Bortzmeyer, and Mohsen Souissi (April 15, 2013)*

Assuming you have successfully added the new TLSA record and generated the appropriate signature(s), now you can query for it:

```
$ dig _443._tcp.www.example.com. TLSA

...
;; ANSWER SECTION:
_443._tcp.www.example.com. 3600 IN TLSA 3 0 1 294874 ←
    DA378148CDD1B9C57D2E891E8C294D2958F0BCA7400A0D6D6F 50C4A3BB
...
```

Great! But that's still only half of the equation. We still need to make your web browser utilize this new information. For this recipe, we are showing you results of using Firefox with a plugin called DNSSEC TLSA Validator from <https://www.dnssec-validator.cz>.

Once the plugin is installed, activated, and Firefox restarted, when you visit the URL <https://www.example.com>, your browser will prompt you for a warning, because this is a self-signed certificate:

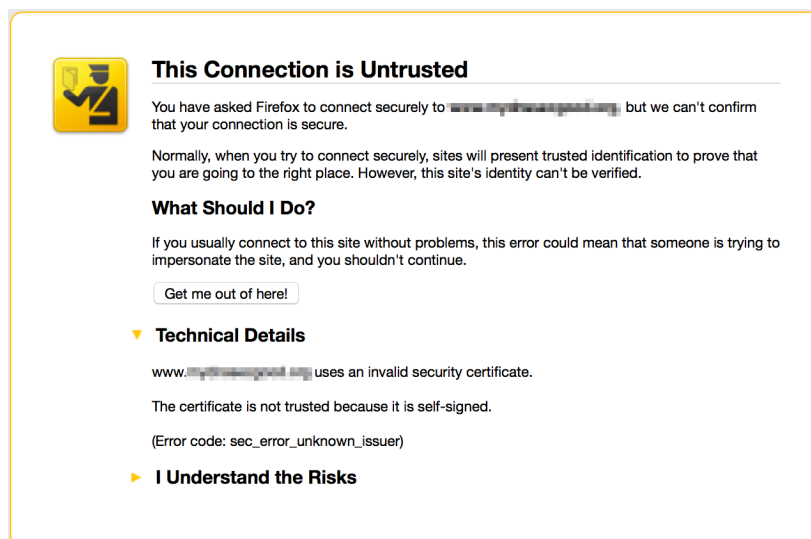


Figure 7.16: Browser Certificate Warning

Although the certificate is not trusted by the browser itself (if you want to you'll have to install a custom CA root or make the browser trust the certificate individually), the plugin shows that it was able to verify the information it received via HTTPS (port 443), and that it matches the information it received via TLSA lookup over DNS (port 53).

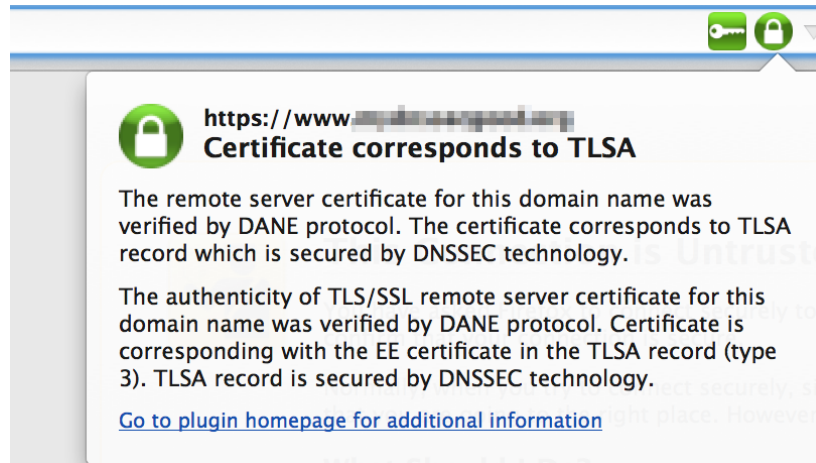


Figure 7.17: DNSSEC TLSA Validator

Chapter 8

Commonly Asked Questions

No questions are too stupid to ask, below is a collection of such questions and answers.

Q: *Do I need IPv6 to have DNSSEC?*

A: No. DNSSEC can be deployed independent of IPv6.

Q: *Does DNSSEC encrypt my DNS traffic, so others cannot eavesdrop on my DNS queries?*

A: No. Although cryptographic keys and digital signatures are used in DNSSEC, they only provide authenticity and integrity, not privacy. Someone who sniffs network traffic can still see all the DNS queries and answers in plain text, DNSSEC just makes it very difficult for the eavesdropper to alter or spoof the DNS responses.

Q: *Does DNSSEC protect the communication between my laptop and my name server?*

A: Unfortunately, currently, no. DNSSEC is designed to protect the communication between the end clients (laptop) and the name servers, however, there are few applications or stub resolver libraries as of late 2016 that take advantage of this capability. This communication between the recursive server to the clients are commonly called the "last mile", while enabling DNSSEC today does little to enhance the security for the last mile, we hope that will change in the near future as more and more applications become DNSSEC-aware.

Q: *Does DNSSEC secure zone transfers?*

A: No. You should consider using TSIG to secure zone transfers among your name servers.

Q: *Is DNSSEC going to protect me from malicious web sites?*

A: The answer for now is, unfortunately for early stages of DNSSEC deployment, no. DNSSEC is designed so you can have confidence that when you received the DNS response for `www.isc.org` over port 53, you know it really came from the ISC name servers, and the answers are authentic. But that does not mean the web server you visit over port 80 or port 443 is necessarily safe. Further more, 99% of the domain names (as of this writing) have not signed their zones yet, so DNSSEC cannot even validate their answers. The answer for sometime in the future is, as more and more zones are signed and more and more recursive servers are validating, DNSSEC will make it much more difficult for attackers to spoof DNS responses or perform cache poisoning. It still does not protect users from visiting a malicious web site that the attacker owns and operates, or prevent users from mis-typing a domain name, it just becomes unlikely that the attacker can hijack other domain names.

Q: *If I enable DNSSEC validation, will it break DNS lookup for majority of the domain names, since most domains names don't have DNSSEC yet?*

A: No, DNSSEC is backwards compatible to "standard" DNS. As of this writing, although 99.5% of the .com domains have yet to be signed, a DNSSEC-enabled validating resolver can still lookup all of these domain names following the "old fashioned way". There are four (4) categories of responses ([RFC 4035 Sec 4.3](#)):

1. *Secure*: Domains that have DNSSEC deployed correctly
2. *Insecure*: Domains that have yet to deploy DNSSEC
3. *Bogus*: Domains that deployed DNSSEC but did it incorrectly

4. *Indeterminate*: Unable to determine whether or not to use DNSSEC

A validating resolver will still resolve #1 and #2, only #3 and #4 will result in a SERVFAIL. You may already be using DNSSEC validation without realizing it, since some ISP's have begun enabling DNSSEC validation on their recursive name servers. Google public DNS (8.8.8.8) also has enabled DNSSEC validation.

Q: *Do I need to have special client software to use DNSSEC?*

A: The short answer is no, DNSSEC only changes the communication behavior among DNS servers, not DNS server (validating resolver) and client (stub resolver). With DNSSEC validation enabled on your recursive server, if a domain name doesn't pass the checks, an error message (typically SERVFAIL) is returned to the clients, and to most client software today, it looks as if the DNS query has failed, or the domain name does not exist. The longer answer is although you don't have to, you may want to. There are more and more client softwares that take advantage of the new DNSSEC features and give user better feedback about the domain name they are visiting. CZ.NIC Labs has created a plugin for several popular web browsers, and Mozilla has created a new web browser Bloodhound that performs DNSSEC validation. As DNSSEC deployment becomes more common place, we are sure to see more and more software libraries and applications be updated to support its features.

Q: *Since DNSSEC uses public key cryptography, do I need Public Key Infrastructure (PKI) in order to use DNSSEC?*

A: No.

Q: *Do I need to purchase SSL certificates from a Certificate Authority (CA) to use DNSSEC?*

A: No. With DNSSEC, you generate and publish your own keys, and sign your own data as well. There is no need to pay someone else to do it for you.

Q: *My parent zone does not support DNSSEC, can I still sign my zone?*

A: Technically, yes, you can sign your zone, but you wouldn't be getting the full benefit of DNSSEC, as other validating resolvers would not be able to validate your zone data. Without the DS record(s) in your parent zone, other validating resolvers will treat your zone as an insecure (traditional) zone, thus no actual verification is carried out. The end result is, to the rest of the world, your zone still appears to be insecure, and it will continue to be insecure until your parent zone can host DS record(s) for you, effectively telling the rest of the world that your zone is signed. An interim solution is to take advantage of DLV (DNSSEC Look-aside Validation), by submitting your key to a DLV registry such as <https://dlv.isc.org/>, you can still get the benefits of DNSSEC even if your parent zone is not yet supporting it.

Q: *Is DNSSEC the same thing as TSIG that I have between my master and slave servers?*

A: No. TSIG is typically used between master and slave name servers to secure zone transfers, DNSSEC secures DNS lookup by validating answers. Even if you enabled DNSSEC, zone transfers are still not validated, and if you wish to secure the communication between your master and slave name servers, you should consider setting up TSIG or similar secure channels.

Q: *How are keys copied from master to slave server(s)?*

A: DNSSEC uses public cryptography, which results in two types of keys: public and private. The public keys are part of the zone data, stored as DNSKEY record types. Thus the public keys are synchronized from master to slave server(s) as part of the zone transfer. The private keys do not, and should not be stored anywhere else but the master server in a secured fashion. See Section 6.3 for more information on key storage options and considerations.