DNSSEC Mastery

Michael W Lucas

2nd Edition

eSHARAM

DNSSEC Mastery

Second Edition

Michael W Lucas

Author: Michael W Lucas
Copyeditor: Amanda Robinson
Cover art: Eddie Sharam

ISBN (paperback): 978-1-64235-059-3
ISBN (hardcover): 978-1-64235-060-9

Tilted Windmill Press

https://www.tiltedwindmillpress.com

DNSSEC Mastery

Second Edition

Michael W Lucas

Brief Contents

Complete Contents

Acknowledgements

The original *DNSSEC Mastery* was my second independently published book. Back then I still had a day job as a network engineer, Unix admin, and responsible party. Now I'm a full-time author, playing with whatever technology amuses me and spewing mayhem-filled lies on the side. The people I really want to thank?

You folks who buy my books.

I also must thank my technical reviewers: Matthijs Mekking, Jan-Piet Mens, John W. O'Brien, Florian Obser, Mike O'Connor, Neil Roza, Carsten Strotmann, and Grant Taylor. Any errors in this book are my fault, despite the best efforts of these fine people.

My Patronizers (https://patronizeMWL.com) help make this lunatic career possible. These folks send me enough money each month that I must name them in the electronic version of every book I publish: Allan Jude, Florian Obser, Maximilian Elaine Paul, Ray Percival, sungo, and Peter Wemm. A few dedicated readers Patronize me so hard, I list them in the electronic *and* print versions of everything: Kate Ebneter (the fantabulous First Wildebeest), Stefan Johnson, Jeff Marraccini, Eirik Øverby, and Phil Vuchetich. My sincere appreciation to you all.

Chapter 0: Introduction

The Domain Name System (DNS) is the most successful distributed database in the world, simultaneously responsible for the Internet's success and many of its failures. DNS maps hostnames like `mwl.io` to IP addresses, so that we can use the Internet without having to remember numbers like `192.0.2.87` or `2001:db8::bad:c0de:cafe`. It also tells devices where to find their configurations, Certificate Authorities whether or not they can issue X.509 certificates for a domain, SSH clients server host keys, and more. DNS has been in continuous use since 1983 and while it has evolved and expanded, its core design remains unchanged. There's just one minor, itty-bitty problem.

DNS is gullible.

That's not its fault. Think about the Internet in 1983. The fact that the Internet worked, at all, was a technological miracle. Nobody tried to break the network. Even the young students who poked into dusty corners and "hacked" into systems were more interested in learning about the technology and demonstrating their intelligence than doing any damage. DNS was a vast improvement over the centrally managed hosts file that preceded it.

But then the Internet became mainstream, international, and inexpensive. Computer hobbyists and enthusiasts joined, then their families, then every office worker and child and bank and power plant and hospital. We connected ICU ventilators and individual light bulbs. Now that everyone does their banking on the Internet, criminal elements have a large financial interest in breaking it. DNS is an attack vector. Seeding a network with invalid DNS information isn't as easy as it used to be, but when an intruder pulls it off it's wildly effective.

Can an organization guard against these attacks? Sure. But very few organizations have a dedicated DNS administrator. DNS management is usually rolled into another team, such as network administration or server management. You folks have switches to install. Desktops to image. Directories to Active. Tasks that you think of as "your real job," of which DNS management is a tiny inconsequential piece.

For all that, DNS failures have disproportionate impact. A nameserver daemon can run in mere megabytes of memory, but if it dies everything on the network loses its freaking mind. People only care about DNS when it breaks, and they think of it in binary terms. Does it work, or not? Corrupt DNS gets zero attention until someone gets bit.

Corrupt DNS is a serious threat to any organization. If an attacker can make the world think that your organization's web site is on a server he runs, he can intercept the incoming traffic. Users intending to visit your site will go to the intruder's instead. The attacker's site might even appear more legitimate than your own; possessing an X.509 certificate for a site means only that the person who requested the certificate controls the site's DNS. A savvy intruder will silently record the user's data and forward the transaction to your server. You'll realize what happened only when your customers tell you that the day they bought your widgets, someone stole their credit cards.

You don't want to be the client in this situation either. If a critical supplier, a bank, or a major Internet site suffers from corrupt DNS, you can suffer damage from lost credit card numbers to assembly line shutdown. Even theft of web site authentication credentials can cause hours or months of grief.

But what about X.509 certificates on web sites? A certificate verifies the identify of a web site, but it only works once the traffic reaches the correct server. To reach that site, the client must have the correct DNS information. Commercial certificate authorities have repeatedly issued certificates for large organizations to people utterly unrelated to the organization. Non-commercial certificate authorities that issue free certificates use DNS to validate control of the server. An intruder that can get invalid information into your DNS can get a TLS certificate for your site.

Any of these will ruin your day. You do not want to be in the meeting where the accountant asks why an auction site used his company credit card to buy the original set for Scotty's TARDIS from *Space: 1999*. Nor in the meeting where the company president wants to know how the plans for the firm's revolutionary low-lubricant prop shaft went to an unscrupulous competitor rather than the usual prototype printer up the street.

The case for securing DNS distills to: *you don't have time for this crap*.

Domain Name System Security Extensions (DNSSEC) ensure the authenticity and integrity of DNS data.

What's The Problem?

What's wrong with the way DNS currently maps names and addresses? Yes, DNS servers have had a spotty security history, but that's been normal for many protocols and implementations over much of the Internet's history. The DNS protocol itself is subject to abuses. Many smart people have worked very hard to secure DNS, and have done pretty well, but successful protocol security is never an afterthought. DNS' distributed nature exacerbates the problem. Intruders attack DNS at the authoritative servers, the recursive servers, and the clients. To understand the risks, consider the DNS data set.

The basic unit of DNS organization is the *zone*. A zone is a collection of records that share the same suffix. Many people think of a zone as a domain, like **`mwl.io`**. A domain is a zone, yes. Parent domains like **`.com`** and **`.pizza`** are also zones. So are reverse DNS zones like **`2.0.192.in-addr.arpa`**. Zones contain records about the hosts, child zones, and major characteristics of the zone. The zone for **`.com`** contains the delegated nameserver (NS) records about all the zones under **`.com`**, including **`michaelwlucas.com`**. The root zone contains the delegations for its child zones, like **`.com`** and all the other top-level zones. This book frequently uses the more familiar term *domains*, but you should be aware we're talking about all zones.

If an intruder can compromise the authoritative servers for a zone, she can enter any data she likes into the zone. Everyone will treat the information as valid—after all, it comes from the authoritative server. This is the most thorough way to compromise DNS, but it's also one of the most difficult to perform against a security-conscious target. Certain DNSSEC configuration can prevent such attacks.

The recursive servers, such as an organization's client-facing nameservers, get their information by querying authoritative servers. An intruder might corrupt these queries, adding extra information into an otherwise valid query or wholly changing the response. An intruder has any number of ways to taint a recursive server's data, and people keep discovering more. DNSSEC addresses this common class of attack.

Finally, an intruder could interpose themselves between a client and its recursive nameserver. A client that validates DNSSEC is immune to such attacks. A client that does not validate DNSSEC must use another mechanism, such as DNS over HTTPS (DoH) or DNS over TLS (DoT), to secure the connection between it and the recursive nameserver.

DNSSEC and Security

Security is often considered a combination of *confidentiality*, *integrity*, and *availability*. DNS data is not only overwhelmingly public, the public data must remain public to work. DNS availability relies on techniques such as secondary servers located on different networks. Integrity is the big problem of DNS security, and that's where DNSSEC comes in.

DNSSEC adds digital signatures to DNS data. A client can verify that DNSSEC-protected data it received is the data sent by the authoritative server, and reject incorrect data. DNSSEC digital signatures share much in common with digital signatures in other security protocols such as TLS and OpenPGP. If you're familiar with cryptographic hashes and digital signatures, the principles behind DNSSEC will not surprise you.

An intruder could still attack the end client—after all, anyone who controls the desktop controls the user experience—but DNSSEC eliminates one broad path of attack.

DNSSEC does not save you from DNS misconfiguration. It doesn't keep your software from crashing. It doesn't replace packet filters, proxies, antivirus, antimalware, routine server or personal hygiene, or planning. DNSSEC can't protect you from network outages or misconfiguration, like putting all your authoritative servers on the same network segment. If you don't install routine upgrades and security patches, DNSSEC can give you a false sense of security. If you have not upgraded your keys since the first edition of this book, you're in trouble.

DNSSEC has two components. The first is *signing*, or adding DNSSEC to your own zone. The other is *validation*, or verifying the DNSSEC on a zone you don't control. Note that "keep intruders out of my nameserver" does not appear on this list.

Sysadmin Prerequisites

This book is for Domain Name Administrators who want to incorporate DNSSEC into their systems, and for sysadmins and protocol developers who want to reliably distribute information via DNS. I will fling around words like "zone" and "reverse DNS" and expect you to understand. I won't explain basics like A, PTR, NS, MX, and SOA records, or the difference between a caching and authoritative name server. I will detail DNSSEC-specific records.

While the DNSSEC theory, best practices, and diagnostic guidance herein applies to all DNS server software, my reference platform is the Internet Systems Consortium's (ISC) BIND 9. BIND is the most popular authoritative DNS server on the public Internet, run by many of the root nameservers and top-level domains. I've separated the theory from the implementation, however. The simplest way to manage DNSSEC with current BIND requires rndc(8), so configure it first. My examples use BIND 9.16.22 or BIND 9.17.18 when necessary. Many Linux distributions, especially long-term support versions of them, ship with old BIND versions, so terminology and features might differ.

Examples include both IPv4 and IPv6.

Technical Prerequisites

Traditional DNS is fairly forgiving. Very few implementations adhere strictly to the standards. This has made it easy to get DNS up and working. I've seen sites with fifteen-year-old nameservers, where the root zone hints file hadn't been updated since the server was installed, and DNS still worked. I've seen zone files with all sorts of weird shortcuts and abuses, but they parsed and loaded and worked. People have made up random child zones of public domains, and they worked. Many bad practices became institutionalized because they keep working.

This flexibility lets people cram all sorts of stuff into DNS. DNS can be your company phone book. You can exfiltrate data from a company over DNS. You can tunnel interactive SSH over DNS.

DNSSEC is not forgiving. If the data doesn't adhere fairly closely to the standards, it won't validate. If you're uncertain of your data or your DNS skills, deploy on a test domain first. Don't let DNSSEC scare you away, but do respect it. Consider this an opportunity to refine your DNS skills.

I strongly recommend using a test domain for your first DNSSEC deployment.

Server Prerequisites

Configure your DNS server properly before adding DNSSEC. Start by installing the recommended nameserver software and all security patches. Fully patch the operating system and configure the host to deploy updates and patches automatically, if possible. In over a quarter century of system administration, I've rarely had an operating system update make a nameserver refuse to run.[1] While it's possible that an update will break the nameserver, it's far more likely that a failure to update will permit a security breach.

A DNSSEC-validating recursive nameserver makes the same queries as a non-validating one, but adds additional queries to gather verification information. These nameservers use additional memory and processor time to verify signatures. Without DNSSEC, you can serve thousands of clients on 20-year-old hardware. With DNSSEC, those same thousands of clients require 10-year-old hardware.

DNSSEC increases the size of your zone files. Even signed zone files are minuscule compared to modern disks and networks. If your nameserver's zone files already use a substantial amount of disk space, you'll need more.

As with any public server, disable all unnecessary services on the machine. Use netstat(1), sockstat(1), or lsof(1) to see which programs are listening to the network. Disable anything not strictly necessary. Restrict command line access to a short list of management IP addresses, and require public key authentication for SSH. If you don't know how to do that, permit me to suggest my book *SSH Mastery* (Tilted Windmill Press, 2018).

---

[1] The exceptions universally involved SELinux.

Incorrect time, on either the server or the resolver, breaks DNSSEC. Have the host maintain its time from the network via NTP, or invest in a GPS clock. I strongly encourage you to monitor the system clock, especially on virtual machines prone to time skew. Beware making time DNS-dependent, or DNS time-dependent.

Nameservers should run as a dedicated unprivileged user (not as **`nobody`** or **`root`**!) and in a chroot. Most operating systems package nameservers this way by default, but double-check yours.

Different Unix breeds place their BIND configuration files in different directories. I'll refer to the `/etc/namedb` directory, but your Unix might place it anywhere. There's no need to move the BIND directory to `/etc/namedb`.

Verify that all changes remain in place after a reboot. Yes, all this is tedious and annoying. DNS failures are exciting, however. I like boring, and recommend it to everyone in IT.

Network Prerequisites

You'll often hear that traditional DNS used UDP port 53, because the queries and responses could fit inside single UDP packets. TCP was added to DNS in 1987, over a third of a century ago, and RFC 7766 documents the current standard. In the late 1990s, Extended DNS (EDNS) allows DNS queries use large UDP packets, which might get fragmented and reassembled. The client and server should be able to use whatever protocol works for their environment. DNS queries and responses are designed to squirm through almost any network.

Except.

Some firewall administrators believe that DNS only uses UDP and block TCP port 53. Worse, some firewall products have filters intended prevent building tunnels over port 53, inspecting DNS traffic to ensure it really is DNS. Some of these products assume that all legitimate DNS queries are 512 bytes or smaller, and drop anything larger. Others drop all IP fragments, rather than reassembling them. Cable and DSL modems are notorious for this kind of daftness. Some host-based firewalls drop IP fragments by default, but you can change this behavior. If your packet filter has any type of naïve DNS filter, your DNSSEC-aware resolver will give spotty answers.

How do you test your environment to see if you have one of those limits? No method can detect all possible network flaws, but if your DNS server can perform queries on large zones you should be okay. The easiest test is the OARC Reply Size Tester (**`https://www.dns-oarc.net/oarc/services/replysizetest`**). Querying their reply size tester will show the largest response the server can receive.

```
$ dig +short rs.dns-oarc.net TXT
rst.x4050.rs.dns-oarc.net.
rst.x4058.x4050.rs.dns-oarc.net.
rst.x4064.x4058.x4050.rs.dns-oarc.net.
“107.191.49.237 DNS reply size limit is at least 4064”
“107.191.49.237 sent EDNS buffer size 4096”
```

As of 2020, a reply size of 1232 bytes is adequate. This host can receive full sized responses. Other hosts sending the same query might show smaller limits. Here's a request from a different host.

```
$ dig +short rs.dns-oarc.net TXT
rst.x1384.rs.dns-oarc.net.
rst.x1347.x1384.rs.dns-oarc.net.
rst.x1353.x1347.x1384.rs.dns-oarc.net.
“2607:f8b0:4001:c17::117 DNS reply size limit is at least 1384”
“2607:f8b0:4001:c17::117 sent EDNS buffer size 1400”
```

Those of us who have been running DNS since the twentieth century might be worried at this, but 1400 is more than 1232. It's good enough.

The Reply Size Tester page documents other, less common responses.

If OARC stops being so generous and stops providing this service, you'll need to query a large zone and see what happens. Running a command like `dig paypal.com any +notcp` should spew multiple screens of information. (ANY queries default to using TCP, so you must explicitly disable it.) If you can get identical answers over UDP and TCP alike, and can successfully query multiple domains that return large answers, you're probably fine.

If testing shows your nameserver suffers from improper packet filtering, stop testing DNSSEC immediately. DNSSEC will not work until you resolve the network issues. Investigate your network step-by-step until you identify the problem. You might find ISC's document “Testing EDNS Compatibility with dig” useful.

For more information about network issues and DNSSEC, look at the DNS Flag Day web site (https://dnsflagday.net) and the "Fragmentation Avoidance in DNS" Internet-Draft.

Parent Zone and Registrar Prerequisites

Deploying DNSSEC on a zone requires both support from the parent zone and your interface to that zone.

Your parent zone might be something like `.com` or `.org` or whatever. Most but not all of these zones support DNSSEC. It might be a reverse DNS zone like `198.in-addr.arpa`. It could also be a corporate parent zone. If you want to deploy DNSSEC on `detroit.myhugecompany.com` for your local office, the `zone myhugecompany.com` must have DNSSEC.

Your interface with your parent zone must also support DNSSEC. Your public IP addresses are managed via a Routing Information Registry (RIR). If you have a /24 or larger block of IPv4 addresses and control your reverse DNS, you can deploy DNSSEC on your reverse DNS. If you're on a private organizational network, talk to whoever manages the global network. The tricky part comes with domain registrars. Despite DNSSEC being a standard for over a decade, some registrars have not bothered to support it. If your registrar does not support DNSSEC, you cannot deploy DNSSEC while your domain is served by that registrar. Find another registrar, and inform the old one why you're leaving.

Rather than repeatedly spew a tangle about corporate offices and RIRs and domain registrars, I'll use the word *registrar* to mean "your interface to your parent domain."

In addition to the domain name registrar, domain names have a *registry*. The registry is the entity that manages the zone. Your registrar lets you register, say, a `.com` domain name. The registrar hands the request off to the `.com` registry, which manages the zone's records.

Recursive and Authoritative Servers

Many historical DNS attacks relied on a nameserver performing both authoritative and recursive services. The purpose of a recursive server is to accept data *from* the public Internet, while the purpose of an authoritative server is to provide data *to* the public Internet. An intruder who wants to

feed harmful data into a nameserver needs the nameserver to accept data. Authoritative servers that refuse to accept any data from the public eliminate that entire class of attacks.

Configure one set of nameservers to serve your clients. These nameservers should not be authoritative for any public zones. (Recursive nameservers should be authoritative for private address space and other garbage domains like **`.local`**, **`.belkin`**, **`.home`**, **`.invalid`**, **`.localdomain`**, and **`.domain`** to avoid flooding the root servers with useless queries.) Configure a second set of nameservers as authoritative servers, and entirely disable recursion on them.

If you're interested in quickly resolving non-resolvable domains, look at the AS112 Project (**`https://www.as112.net`**).

What's In This Book?

This book is not a comprehensive DNSSEC grimoire. I cover the common cases of securing the information in your forward and reverse zones, as well as a few examples of how DNS can be leveraged once you have DNSSEC. I recommend best practices as per the various Internet standards and documents like NIST's *Secure Domain Name System (DNS) Deployment Guide* (NIST Special Publication 800-81-2). If you want to smuggle SSH over DNS despite DNSSEC, or if you want to study the intimate details of DNSSEC, this is not the book for you. This book will help you upgrade to basic DNSSEC competence, however, and point you in directions to learn more.

Chapter 0 is this Introduction. Every tech book has one, but this one is mine.

Chapter 1, *DNSSEC Cryptography*, explains the cryptography underlying DNSSEC. If you're already familiar with public keys and HMACs, you'll at least want to read the parts on how DNSSEC uses cryptography and the types of keys it uses.

Chapter 2, *DNSSEC Records*, explains the resource records used to implement and troubleshoot DNSSEC. I also cover how the Chain of Trust works, protecting private keys, and interacting with your registrar.

Chapter 3, *Examining DNSSEC*, uses the basic DNS diagnostic tool `dig` to recognize and examine DNSSEC data and investigate DNSSEC issues.

Chapter 4, *Securing Zone Transfers*, covers securely copying zones between DNS servers with transaction signatures.

Chapter 5, *Signing Zones*, discusses how zones get signed.

Chapter 6, *Failure Modes and Troubleshooting*, covers how and why DNSSEC fails. I address useful troubleshooting tools and some of the online resources for debugging your DNSSEC environment.

Chapter 7, *Key Rollover*, covers key lifecycles and how to change your keys. Rolling over keys is a vital DNSSEC maintenance task.

Chapter 8, *BIND Policies and Rolls*, discusses BIND 9's new policy-based key management system and rolling over keys.

Chapter 9, *Securing Clients*, covers providing DNS over TLS (DoT) and DNS over HTTPS (DoH) to your clients.

Chapter 10, *Delegations and Islands of Trust*, covers securing child zones with DNSSEC and implementing DNSSEC on networks using private address space.

Chapter 11, *DNSSEC for Data Distribution*, shows how to provide information other than hostnames and IP addresses via DNS. We'll put SSH host keys and X.509 certificate fingerprints in DNS, confident that DNSSEC prevents information corruption.

Enough talk. Let's see how DNSSEC changes DNS.

Chapter 1: DNSSEC Cryptography

DNSSEC allows clients to cryptographically validate information sent from a server. If my authoritative DNS server claims that `mwl.io` has the IP address `2001:db8::f00d`, clients should get that address upon request. If someone forges DNS response packets or poisons a recursive server's cache, a server that validates DNSSEC queries will recognize the contamination and reject the bogus data. But what does it mean to "cryptographically verify" something?

Most folks think of cryptography as a method to transform text between readable *plaintext* and unreadable *ciphertext*. Only someone who knows the secret key and the encryption method can perform either transformation. This is the stuff of thrillers, spies, and offbeat pseudo-Victorian artists, but while cryptography started as secret messages, it now supports all sorts of tools and techniques. Cryptography is a necessary part of DNSSEC, but it's not the main focus. You don't need to understand how to encrypt and decrypt text, compute digital signatures, or hide secret keys in the sole of your shoe so you can smuggle them through a checkpoint. I discuss these at length in many other books,[2] and other authors can take you deep into the math. You do need to understand how the pieces fit together and how to handle the various components.

Many of the encryption concepts used by DNSSEC are also leveraged by protocols such as HTTPS and TLS. The critical difference between them and DNSSEC is that DNSSEC doesn't do any encryption and signing online. DNS clients do not negotiate encryption parameters or build "secure" channels. There's no equivalent to the encryption handshakes you'll find in SSH or SNMPv3. The primary DNS server signs records every week or so and regenerates those signatures as needed. The DNS clients download the records and validate the signatures.

We'll discuss cryptographic hashes, public key cryptography, digital signatures, DNSSEC keys, and the Chain of Trust.

---

[2] I've written about cryptography and hashes in so many other books that I truly wanted to not cover it yet again. But you need it to understand DNSSEC. Ignore the stains left by my tears.

Cryptographic Hashes

A *hash* or *checksum* is a computation that produces a fixed-length string from any chunk of data, and is used as an integrity verification tool. A method of producing a hash is called an *algorithm*. Each algorithm produces a hash of a fixed length, but different algorithms produce longer or shorter hashes. A SHA256 hash is always 256 bits long, while a SHA512 hash is 512 bits. Hashes can be cryptographic or not.

A *non-cryptographic hash* is useful when you expect corruption rather than tampering. A simple hash to test the integrity of an ebook might be "count the number of words in the book and prepend enough zeroes to make it twelve digits long." If the ebook was damaged in transit to you, this hash would probably catch it. You'll see similar hashes in TCP and throughout manufacturing. The last digit of the ISBN on the printed version of this book is a one-digit hash. These are all valid hashes, but easily forged. Detecting altered data requires a more robust hashing algorithm.

With a *cryptographic hash*, any change in the data changes the hash generated from the data. Take the IP address `2001:db8::f00d`. Its SHA256 hash is 0bc7561a5faae51787e0900e7de577e8f67382883068f3a-2205229665db9b8a4. If an intruder moved the colon, making it `2001::db8:f00d`, a typical overworked and underpaid sysadmin might not even notice. The SHA256 hash of this address is 4fdc0949e0f203bc-11fe1cb3b4de262b63c6898dc775d8322c83ad05ba094ebd, however. Even the most cursory check of the hash shows the difference.

There's an infinite number of possible data sets, and SHA256 only has 256 bytes. Eventually, two data sets will have the same hash. This is called a *hash collision*. Resistance to computing such collisions is what makes a hash cryptographic.

Consider the hash of "how many words are in a book." Creating a document with the same hash is trivial. The non-cryptographic hash is useless for detecting deliberate tampering. If you're using a modern cryptographic hash, finding a data set that has a hash collision with your target data will take decades on a server farm. When you do find a match, it sure won't look like valid DNS data. By the time you find a suitable hash collision, the original data you're targeting has almost certainly changed.

Applications use cryptographic hashes to verify that the contents of a message have not changed. If a server sends a message like a response to a DNS query and includes a hash of the message contents, the recipient can use the hash to verify the original message. The hash must be sent carefully to ensure it has not also been tampered with.

Salted Hashes

One possible way to make a hash more secure is through *salting*. A salt is a value that is added to the original data before the hash is computed. Suppose I compute the hash of **`2001:db8::f00d`** using the salt GiveLucasYourGelato. I would compute the hash based on the string 2001:db8::f00dGiveLucasYourGelato. You use a different salt for every piece of plaintext, and the salt is generally known. When DNSSEC uses a salt, the salt appears in a DNS record. When a salted password appears in Unix's `/etc/master.passwd` file, the salt appears right next to the salted password.

While DNSSEC documentation mentions salts, they've recently been shown to be not useful in this application. Their use in DNSSEC has recently been discouraged. BIND 9.18 will default to not using salts.

Public Key Cryptography

Symmetric encryption algorithms use a secret key to transform plaintext to and from ciphertext. Most algorithms use the same key to encrypt and decrypt text. If you have the key and know the algorithm, you can decipher messages. The confidentiality of messages exchanged via symmetric encryption relies on the secret key remaining secret. This is the traditional encryption algorithm used for thousands of years.

Some algorithms use two different keys for encryption and decryption. These *asymmetric* algorithms work because the keys are very large numbers, and very large numbers behave strangely when multiplied together. Two matching keys are called a *keypair*. Call the keys X and Y. Someone with key X can encrypt messages that can only be decrypted with key Y, and can decrypt messages encrypted with key Y. The reverse is also true. A key owner cannot decrypt messages encrypted with their own key, however.

Having two different keys that only work in concert creates interesting possibilities. The idea behind *public key cryptography* is that you keep one

key secret, calling it the *private key*. The other key, the *public key*, you give away to the world. Spray paint it on a wall. Put it on your web site. Or, since we're working with DNS, list it in a DNS record. Any message encrypted by the private key can be read by anyone in the world. Does this keep the message secret? Nope. But it does prove that the person who encrypted the message has access to the private key. If only one person has the private key, the message came from that person. If someone wants to decrypt messages encrypted with her own private key, she also needs to use the public key.

MAC and HMAC

Using a hash to verify a message is great until an attacker can intercept the message. If I can catch your message in transit, change the message, and change the accompanying hash to match the new message, the hash has failed. A *Message Authentication Code*, or *MAC*, is a hash encrypted with a symmetric key known only to the sender and recipient. Developers could choose any number of encryption methods to create a MAC.

A *Hashed Message Authentication Code*, or *HMAC*, is a specific standard for using defined cryptographic algorithms to create the MAC. Each has a name, like HMAC-SHA256 or HMAC-MD5. An HMAC provides both authenticity and integrity.

When people discuss cryptography, they often say "hash" when they mean HMAC. It's easier to say, and many less experienced sysadmins don't know the difference, but you should know it's a highly specific and defined method of creating and managing a hash.

Digital Signatures

Combining hashes with public key encryption leads to *digital signatures*. The goal of a digital signature is to ensure that a message is authentic—it comes unaltered from the source that signed it. A digital signature has many components and you don't need to know them all, but it includes the message's cryptographic hash encrypted with the sender's private key. Anyone can grab the public key and decrypt the signature, then compute their own hash of the message. If the hashes match, the message is authentic.

If the received message has a different hash than the digital signature, the signature is invalid. The message was altered or the signature was computed

improperly. DNSSEC discards all data with invalid signatures. (Debugging tools can disable validation and examine such data, of course.)

DNSSEC's digital signatures expire. Modern nameservers automatically renew the signatures, without sysadmin intervention. Even so, skewed clocks can cause expired signature errors. If the nameserver's clock is a week slow, the server won't see any need to update signatures for a few days even though the rest of the world sees that they're expired.

You don't need to compute digital signatures yourself. Every worthwhile nameserver includes software to compute and validate signatures. BIND includes several, such as dnssec-keygen(8), dnssec-verify(8), dnssec-signzone(8), and delv(1).

Digital signatures are so central to DNSSEC, a zone protected with DNSSEC is said to be *signed*.

DNSSEC Keypairs

Each zone has at least one keypair. This key is used to sign records in the zone. The parent zone must know the fingerprint of the keypair's public key. You'll need a way to update those keys.

Keys can be managed with either multiple keys or combined keys.

Multiple Keys

A DNSSEC key has multiple conflicting requirements. The key must be changed regularly enough to discourage intruders from computing the private key by brute force, but rarely enough that the necessary interactions with the parent domain are not tedious and troublesome. Additionally, in DNSSEC's early days, key management software ranged from clunky to infuriating. The best way to cope with these conflicting requirements was to use multiple keys (called *split* keys by some vendors), one that lasted a long time and one that was replaced more frequently.

The long-lived key was the *Key Signing Key*, or *KSK*. The KSK is used only to sign the DNS records for the zone's public keys. Each zone's parent domain needs the fingerprint of the KSK's public key. The domain registrar or RIR handles updating the parent zone with the fingerprint, but you must inform the registrar.

The *Zone Signing Key*, or *ZSK*, is used to sign a zone. Every zone has its own ZSK. This is the key that gets the most use.

Essentially, your DNS data has its own tiny Chain of Trust. The majority of DNS data is signed with the ZSK, while the ZSK itself is signed with the KSK.

Combined Key

Modern DNSSEC includes standards for communicating key rollovers to the parent domain. Automation allows combining the KSK and the ZSK into a *Combined Signing Key* (*CSK*), sometimes called a *Single Signing Key* (*SSK*). Using a CSK with automation is simpler than using two keys, and presents a simpler trust model.

What's technologically special about a CSK? Nothing, really. It's called a combined key, but it's a combination of roles and duties.

Which To Use?

Both combined keys and multiple keys work fine with DNSSEC, and automation can handle either. Which should you deploy?

In most environments, a Combined Signing Key is perfectly adequate.

Multiple keys are useful for zones highly valued by intruders and criminals. An attacker might try to reverse-engineer private keys from publicly available information. A savvy attacker can gather all the digital signatures signed with a particular private key, feed them to a server farm, and grind out the corresponding public key. One thing that helps such attackers is having a large number of encrypted samples with known plaintext, which is exactly what digital signatures provide. DNS data is public. Large zones have many entries, and often belong to organizations large enough to interest criminals. A KSK provides only a few samples to work with.

Zones like `.com` most often use separate KSKs and ZSKs. Zones like `mwl.io`, where I host my blog and list the books I've written, not so much. Every zone makes their own choice. The UK's zone for commercial entities, `.co.uk`, uses a CSK.

Key Rollover

An attacker could use your public key to try to reverse-engineer your private key by sheer brute force computation. This can take years. *Key rollover* or key rotation replaces keys on a zone, so that such attackers must start over. You also rotate keys if the private key is compromised. Sysadmins disagree if routine key rollover is necessary. For now, understand that key rotation is part of DNSSEC.

We discuss key rotation in Chapter 7.

The Chain of Trust and Trust Anchors

Any time you look at a public key encryption system, the first question you should ask is "how is trust distributed?" Who decides who to trust? DNS is non-interactive, so that rules out any trust model like OpenPGP's Web of Trust. DNSSEC uses a *trust anchor*, a trusted key for the root zone. Nameservers ship with the trust anchor and have a protocol for updating it.

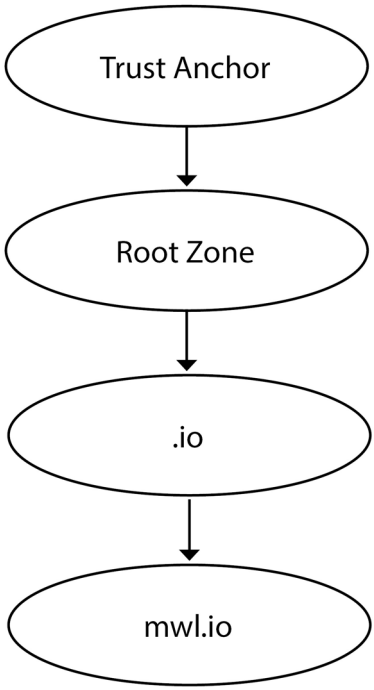

Figure 1-1: Super Simplified DNSSEC Chain of Trust

The trust anchor is used much like the trust anchors for X.509 certificates and , but there's only the one anchor rather than a proliferation of Certificate Authorities. Your recursive nameserver trusts the trust anchor, which is used to sign the root zone. All of the top-level zones that support DNSSEC, like `.com` and `.org` and `.uk`, have their keys signed by the root zone's keys. Child zones of those, like `isc.org` and `mwl.io`, have their keys signed by their parent zone's key. This creates a *chain of trust*.

DNSSEC clients build trust from the bottom up. A client sees a DNSSEC signature on a record, and goes to the parent zone to get the information on the key used to sign that. If it doesn't already trust that key, it gets the next key up the chain. Eventually it finds a trusted key and validates the result—or not.

Suppose your resolver has an empty cache, and sees a signature on **www.mwl.io**. The resolver goes to **mwl.io** and gets the public keys for the keys used to sign that record. The key exists, but your resolver doesn't trust them. It escalates up to **.io** and grabs the key information for **mwl.io**. Your resolver has no reason to trust the **.io** zone yet, so it doesn't trust those. It proceeds to the root zone and pulls the records on the keys for **.io**. Records in the root zone are signed by the key configured as the resolver's trust anchor. Now it can validate the original request by building a chain of trust all the way from the trust anchor down to my web server. As all these records are in DNS, your resolver will cache everything. When the resolver is asked to validate **www.embrogl.io**, it already knows and trusts the public keys for **.io** and everything above it. It will grab the key information from the **embrogl.io** nameservers along with the A and AAAA records and validate everything much more quickly.

You trust the root, so you trust everyone that the root tells you to trust. DNSSEC lets you verify that trust at every link of the chain. If any of these signatures or keys do not match, trust is violated and validation fails. The resolver rejects the information and informs the client that there's no such host. Chapter 3 discusses debugging DNSSEC validation failures, including some pretty diagrams on how everything breaks—er, works. *Works*.

A domain can only deploy validatable DNSSEC if its parent zone is secured with DNSSEC. All generic top-level domains (gTLDs) are signed. Most but not all top-level country codes like **.bb** and **.do** are not signed. An unsigned intermediate zone breaks DNSSEC. Suppose I have a remote office to support my global publishing empire and delegate them a child zone, **fiji.mwl.io**. The folks in Fiji want to sign their zone. If **mwl.io** is unsigned, **fiji.mwl.io** cannot be signed. The Chain of Trust would terminate in the last signed zone. You get unsigned answers.

Child zones do not have to be signed. The **.com** zone is signed, but many companies have not deployed DNSSEC. Instead, the parent domain authoritatively declares that no key exists. As your resolver trusts the parent domain, it accepts that the zone is not signed and provides the unsigned answer to the client.

Algorithms and Life Cycle

DNSSEC uses very few algorithms next to protocols like SSH and TLS. The current standard, RFC 8624, lists 12 algorithms like RSAMD5, RSASHA256, and ED25519.

Number	Name	Signing	Validation
1	RSAMD5	MUST NOT	MUST NOT
3	DSA	MUST NOT	MUST NOT
5	RSASHA1	NOT RECOMMENDED	MUST
6	DSA-NSEC3-SHA1	MUST NOT	MUST NOT
7	RSASHA1-NSEC3-SHA1	NOT RECOMMENDED	MUST
8	RSASHA256	MUST	MUST
10	RSASHA512	NOT RECOMMENDED	MUST
12	ECC-GOST	MUST NOT	MAY
13	ECDSAP256SHA256	MUST	MUST
14	ECDSAP384SHA384	MAY	RECOMMENDED
15	ED25519	RECOMMENDED	RECOMMENDED
16	ED448	MAY	RECOMMENDED

Each algorithm is identified by a number and a name. The number appears in DNS resource records, the name is for human beings. We then get two columns, one saying if the algorithm should be used for signing zones and the other declaring if it should be trusted when verifying zones. In the Signing and Validation columns, the MUST, MUST NOT, MAY, and NOT RECOMMENDED statements are for software implementors.

Consider algorithm number 1, RSAMD5. Software must not permit signing with RSAMD5, or validate zones signed with it. It was once a standard, but the MD5 algorithm is no longer trustworthy. Algorithm 3, DSA, is similar.

Algorithms 5, 7, and 10 are NOT RECOMMENDED for signing, but clients are required to validate zones signed with these algorithms. Algorithms 5 and 7 use SHA1, which is weak and on its way out but still has a large install base. Your DNSSEC software should stop supporting signing zones with SHA1, but you can't quit trusting it on other people's zones quite yet. In related news, other people need to update their software.

Algorithms 8 and 13 are the only ones that current software must be able to sign with, and must be able to validate. RSASHA256 and ECDSAP256SHA256 are the algorithms considered best current

practice. RSASHA256 uses the time-honored RSA public keys, while ECDSAP256SHA256 relies on the fancy new ECDSA elliptic curve algorithm with the P-256 curve. Both compute hashes with SHA-256.

Algorithms 14 through 16 are candidates for future standards. Support for validation is recommended, while support for signing is encouraged. Broader testing will show if these algorithms are viable, useful, and an improvement over the current standards. Not all of these algorithms will become standard deployment options. Consider algorithm 10, RSASHA512. It looked good, but in reality it's no better than RSASHA256.

When you're starting with DNSSEC, use algorithms 8 and 13. People argue over which is better, but either suffices. Leave signing your zones with the Glorious Algorithms of the Future for the future.

Private Key Management

Another critical detail for public key encryption is management of private keys. Any entity that has public keys must secure its private keys. We'll look at how the root zone and various top-level domains manage their keys, and then the options for your organization.

Trust Anchor Private Key Management

An entire standard describes the management of the trust anchor private key. If you read the "DNSSEC Practice Statement for the Root Zone KSK Operator" available from the Internet Assigned Numbers Authority (IANA), you'll see requirements for everything from the physical environment where the private key is stored to the type of Hardware Security Module holding the trust anchor's key. The trust anchor private key is managed by the Internet Corporation for Assigned Names and Numbers (ICANN). The trust anchor is used as a Key Signing Key, and is occasionally used to sign a new Zone Signing Key for the root zone.

The trust anchor's private key is never on the Internet; it's literally air-gapped. Using the trust anchor private key requires multiple activation cards and PIN codes and resembles what computer geeks think the procedure for launching nuclear missiles *should* look like[3]. Once the ceremony is complete

[3] I'd discuss what launching nukes is really like, but it would ruin your day.

ICANN provides the new ZSK to Verisign, who updates and signs the root zone daily. The Internet has hundreds of root nameservers, but none of them have the trust anchor's private key. They have a copy of the root zone that is signed by the trust anchor, but that's it.

Don't worry about the trust anchor's private key. Worry about your own.

Managing Your Private Keys

How tightly will you protect your private keys? A key is a file. An intruder can easily copy a file. Greater key protection requires more ongoing work. You must decide how much labor you're willing to perform in exchange for what level of security. Most software suites, including nameservers, must have their private keys available at startup. That's the purpose of these programs, after all—using the private key to perform useful work. Sysadmins habitually store private keys for web and VPN software on the server, exchanging the risk that an intruder could steal the key for the ability to easily restart the software.

Forged DNS data presents a different scale of threat. If an intruder steals the private key for your web server's X.509 certificate, they can stand up web sites that pretend to be yours. An intruder who steals your DNSSEC private keys can direct people to her rogue web site and publish public keys for all of your public services. She can publish SSH host keys in DNS so that your remote sysadmins believe that her honeypot is your network. She can direct your corporate executives to her VPN concentrator, and make the software believe she's legitimate. If your DNSSEC private keys are stolen, you are *owned*.

Not all of these attacks are useful, yet. Not all SSH clients check for host keys in DNSSEC. Maybe you don't use DNS to distribute VPN information. But do your clients check for information in DNS before silently falling back on other data sources? Are you *sure*? Remember, a vendor's interest is not in providing the safest possible configuration; it's in providing a configuration that generates the fewest support calls. Assume the worst.

There are three common approaches to protecting private keys. Which you should use depends on your threat model.

A *Hardware Security Module* (HSM) is the strongest protection. The HSM is a physical device plugged into your server hardware. Private keys can be stored on the HSM or deleted from it, but not copied from it. Programs must query the HSM to access the private key. There is no "file" for an intruder to copy. HSM hardware can be as small and inexpensive as USB key fobs. If you work for a financial institution, your HSM is probably an expensive rack mount device fitted with explosives rigged to detonate if you tamper with the case.[4]

Many organizations rely on a *stealth primary*. The primary DNS server is tightly secured behind a packet filter or proxy, and probably in private address space. It is the only nameserver with the private keys. The public cannot access the primary server for any service. All nameservers that offer public service are backups to the primary. The secondaries do not need the private keys for the zones that they serve. If a public nameserver is compromised, the private key is safe.

Last, you could do *nothing*. Storing your private key on your nameserver, without any special protection other than filesystem permissions, is certainly an option. It might even be a popular option. But it's the most risky option. If you choose to keep the private keys in files on a public nameserver, anyone who breaks into your nameserver has the keys. On the modern Internet, the question isn't whether or not someone will break into your servers. The questions are: when will it happen, and will you notice?

Which should you use? That depends on your organization. Maybe it's okay if your personal domain keeps the keys on the primary nameserver. An organization large enough to have a server farm can afford an HSM. As a sysadmin with over thirty years of Unix experience, I'm biased towards anything involving explosives.

Speaking of blowing things up, let's proceed to the new record types DNSSEC uses.

---

[4] Explosives do not make hardware tamper-proof, but they do qualify as tamper-evident.

Chapter 2: DNSSEC Records

Folks responsible for managing the Domain Name System are all familiar with the common resource records, like `A`, `PTR`, or `CNAME`. But DNS supports dozens of different record types, with names like `AFSDB` and `HIP` and `OPENPGPKEY`. It should be no surprise that DNSSEC adds eight new record types and leverages aspects of DNS that you might have previously ignored. These resource records contain the information and signatures needed to validate DNSSEC.

Record Types and Entries

You should already be familiar with the beginning of a DNS record. All DNS records start with the same four entries.

```
mwl.io. 3600 IN AAAA 2001:db8::f00d
```

The first field in any record is the domain or owner name. This entry is for the domain **`mwl.io`**.

The second field is the time to live, in seconds. This data can be cached for 3600 seconds, one hour. Many DNSSEC records have short times to live, telling you to not cache keys past their signature's expiration.

On the Internet, the third field is almost always IN (for Internet).[5]

Last, we have the record type. This example is for an `AAAA` record. DNSSEC provides several more record types: `DNSKEY`, `DS`, `RRSIG`, `NSEC`, `NSEC3`, `NSEC3PARAM`, `CDS`, and `CDNSKEY`.

As with all other DNS records, each DNSSEC record has its own format for record data (*rdata*) in the fifth field and beyond. We'll number fields as they appear in the rdata.

[5] If you have CH records, you are accustomed to CHaos.

DNSKEY

The DNSKEY resource records contain a zone's public keys and identify the key algorithm. A domain's DNSKEY records look something like this.

```
mwl.io.  3600  IN  DNSKEY  256  3  13  zEo0…
```

The rdata's first field gives the key *flags*, indicating what the following public key is used for. DNSSEC primarily uses two key flags. 256 represents a Zone Signing Key (ZSK), while 257 is a Key Signing Key (KSK). A Combined Signing Key (CSK) is usually 257, although some software flags it 256. Either validates. You might also see the flag 385, which means that the key has been revoked and should not be trusted.

The rdata's second field is the protocol, which should always be 3. The DNSKEY record is derived from an earlier, now-abandoned resource record type (KEY) that supported multiple types of key in one record. The protocol field is a leftover for backwards compatibility that never got cleaned up. You can use other record types to provide non-DNSSEC public keys, as we'll see in Chapter 12.

The third rdata field is the signing algorithm. DNSSEC uses a narrow selection of key algorithms compared to protocols like SSH or TLS. The two you'll see most often are algorithm 8 (RSASHA256) and 13 (ECDSAP256SHA256), as discussed in Chapter 1.

The last field is the base64 representation of the public key. Clients will use this public key to validate signatures.

The DNSKEY record appears in the zone it supports.

DS

The DS (Delegation Signer) resource record links different levels of the DNS. It ties the root zone (`.`) to, say, **`.com`**, and **`.com`** to **`tiltedwindmillpress.com`**, and **`tiltedwindmillpress.com`** to its child domains. A zone's DS record appears in the parent zone, and is an irrefutable declaration that the child zone is signed. If the parent zone has DS records for a zone, but the child zone is not signed, the child zone will not validate. It disappears from the Internet. Sign your zone before interacting with your zone's parent.

As the maintainer of a zone, you provide your registrar or parent zone maintainer with the DS record or the DNSKEY record used to create the DS record. This is called *publishing*. Verify that the zone is signed with that key and has replicated to all secondaries before publishing it.

The DS record contains a hash of the zone's active KSK or CSK, as well as information about the algorithm and the associated key tag. DS records look like this.

```
mwl.io.  86400  IN  DS  7250  13  2  A30B3F78B6DDE9A4A9A2…
```

The rdata's first field (7250) is the *key tag*, also known as a *key id*. A tag is a five digit number to conveniently identify a particular key. There is no requirement that a tag be unique between zones, algorithms, or even within the same zone. If your authoritative nameserver supports multiple zones, it's possible that two zones will have the same tag. If your zone is signed with multiple algorithms—say, during an algorithm rollover—and you are very unlucky, the zone could have keys of different algorithms with the same tag. A zone could reuse the same tag in subsequent keys. Random chance could simultaneously assign a zone two keys with identical tags, but that's highly unlikely. You don't need to show a tag's leading zeros.

The next field is the algorithm (13). This zone is signed with ECDSAP256SHA256, using SHA-256 and a standard elliptic curve algorithm.

The next field gives the hash algorithm used to identify the public key. Remember, a DS record doesn't contain a complete public key. It contains only a hash of that public key. Algorithm 2, SHA-256, is the standard. Algorithm 1 is the obsolete SHA-1, which clients must be able to validate but shouldn't be deployed. Best practice is for this to always be 2.

The DS record must appear in the zone's parent zone, right next to the zone's nameserver records. To get the DS record for the signed zone **`mwl.io`**, query the `.io` nameservers.

The key that a DS record points at is called a *Secure Entry Point*, or *SEP*. It's where the validating resolver first enters a zone.

RRSIG

The Resource Record Signature, or RRSIG, gives a digital signature of a set of resource records. Resource Records Sets are an oft-overlooked part of DNS, so we'll go over them.

A Resource Record Set, or *RRSet*, is a collection of records of the same domain name, record class (IN), and record type. Resolvers return complete RRSets when answering queries. If you make a query for the address of **`www.mwl.io`**, you'll get back a single A record. That zone has only one A record for that host. It's a complete RRSet. If I request the address of **`www.yahoo.com`**, I get six A records. Those six records are also a complete RRSet.

Why are RRSets important? DNSSEC does not sign DNS records. It signs RRSets. The entry for my web server has a single RRSet. Yahoo's six web servers also have only one RRSet. My domains have two nameservers, and the record listing them is one RRSet. A collection of records is signed, not individual addresses.

An RRSIG record looks like this.

```
mwl.io.  86400  IN  RRSIG  DNSKEY  13  2  7200
  20210711042116 20210611040902  7250  mwl.io.
  F5OZTjzST1IKSJNTWzkZbO
```

The first field of the rdata tells you what kind of DNS resource this signature applies to. This entry is for the DNSKEY record. This could be an A, an SOA, an MX, or any other sort of DNS record that appears in your zone or that DNSSEC adds to your zone.

The second rdata field (13) gives the algorithm used to generate this zone. Algorithm 13 is ECDSAP256SHA256. Chapter 1 discusses key algorithms.

The third rdata field (2) has the number of labels in the resource record set. This helps clients determine if the RRSet includes wildcard DNS entries.

The fourth field (7200) gives the original time-to-live (TTL) for this data, without any subtractions made by intermediate caching nameservers. Validating a signature requires the original TTL.

The next two fields give the UTC date the signature expires (20210711042116) and the date the signature becomes valid (20210611040902). This signature becomes valid on the eleventh of June, 2021, at 4:09:02 UTC. It expires on July 11 2021, at 4:21:16 UTC. Signature

expiration has no relationship to DNS time-to-live (TTL). Signatures should last far longer than the TTL, as generating them is computationally expensive compared to serving a record to a client. A month is very common. These timestamps are why nameserver time synchronization is so important. If the time on the primary and recursive nameservers differ, the recursive server might reject an otherwise valid record.

The seventh rdata field (7250) gives the *key tag* (also known as *key id*) of the key that generated this signature.

The eighth field is the signer's name. This should be the zone including this record. This signature on **`mwl.io`** is included in the zone **`mwl.io`**, exactly as you would expect.

The record ends with several lengthy strings comprising the actual digital signature.

The ZSK is used to sign zone data and generate the RRSIGs. The DNSKEY RRSet containing the ZSK and KSK is signed with the KSK. So are the CDS and CDNSKEY RRSets. If you're using a CSK, it signs everything.

NSEC

The *next secure* resource records, NSEC offers proof of a record's nonexistence. If you try to find a record that does not appear in the zone, you need the DNS server to authoritatively say that the record doesn't exist. Proving **`piratebooks.mwl.io`** doesn't exist is as important as proof it exists, or perhaps even more important.

An NSEC record looks like this.

```
osx.mwl.io. 7200  IN  NSEC  printer.mwl.io. A RRSIG NSEC
```

The first field is the standard name record, but in NSEC it has a twist. It might not be the record you requested. It's an earlier record. The second, third, and fourth fields are all standard.

The fifth field, first entry in the rdata, is the name of the next valid host in the zone.

Why have multiple hostnames in one record? This NSEC record shows that there are no valid hosts between **`osx.mwl.io`** and **`printer.mwl.io`**. The nameserver maintains a sorted list of names, and provides the entries before and after what you requested.

Next we have a list of the record types associated with the host at the beginning of the entry. The host **osx** has A, RRSIG, and NSEC records. It proves that there are no, say, TXT or MX records.

You might hear of something called *aggressive NSEC*. Auditors occasionally want to know if you're using it. It became a standard in 2017 (RFC 8198), and all major nameservers picked it up as soon as possible. Aggressive NSEC protects nameservers against certain corner cases and denial of service attacks. If you are stuck on old nameserver software, though, you might need to verify that it supports aggressive NSEC.

NSEC records are flawed in that they can be used to list all entries in a zone. This NSEC record shows that the hosts **osx.mwl.io** and **printer.mwl.io** are legitimate, but no records exist between these two. Querying the NSEC record for **osx.mwl.io** told me that the next valid host is **printer.mwl.io**. I could query the NSEC record for **printer.mwl.io** and see what the next NSEC record exposes. It's a clunky way to do a zone transfer, but an intruder would find the information invaluable. For some zones, notably IPv4 reverses, this doesn't matter. Everybody knows that 3.2.0.192.in-addr.arpa follows 2.2.0.192.in-addr.arpa, and can perform reconnaissance without zone transfers. Forward zones are unpredictable, however. You can use NSEC3 to improve zone confidentiality.

NSEC3/NSEC3PARAM

NSEC3 resembles NSEC, but it hashes the names of all existing entries. DNS reconnaissance only uncovers that other entries exist, not the names.

```
DRVR6JA3E4VO5UIPOFAO5OEEVV2U4T1K.mwl.io. 3600 IN NSEC3 1 0
  10 03F92714 GJPS66MS4J1N6TIIJ4CL58TS9GQ2KRJ0 A RRSIG
```

The first field is a hash of the previous valid hostname, exactly like an NSEC record.

The second, third, and fourth fields are standard DNS fields.

The fifth field (1) is the algorithm used to generate the hash.

The sixth field (0) are for NSEC3 flags. It is almost always 0. A 1 indicates that some child zones are deliberately unsigned.

The seventh field (10) is the number of times the hostname was passed through the hashing algorithm. Originally considered a security

enhancement, in 2021 repeated hashing was demonstrated to provide no additional benefit.

If you've set a salt, the salt and its length appear next. Salts are no longer considered a useful security enhancement. This record has a salt, 03F92714.

Finally, you get the hashed value of the next entry and the record type bit map. There's little reason to parse these by eye.

As with NSEC, you might hear of *aggressive NSEC3*. Exactly like aggressive NSEC, every major vendor has deployed this by default. You only need worry about non-aggressive NSEC3 if you're stuck using old nameserver software.

If your zone uses NSEC3, it will also have an NSEC3PARAM entry. This provides information authoritative servers need to compute NSEC3 records. NSEC3PARAM entries are not involved in validation or name resolution, and are not intended to be human-readable.

CDS and CDNSKEY

The CDS and CDNSKEY records allow a nameserver to communicate with the domain registrar, without going through any proprietary registrar API. These records look exactly like the DS and DNSKEY records, but represent what the child wants the parent's DS RRSet to look like. If the parent zone has already updated their DS records, these might represent the current key.

A domain registrar should monitor each of its client zones for the appearance of CDS and DNSKEY records. When the record appears, the registrar checks to see if the records are signed by the current key. If they are, the registrar changes the existing DS record to that given in the CDS record. We'll discuss this further in Chapter 7.

Records and the Chain of Trust

Chapter 1 includes a simplified diagram of the Chain of Trust used by DNSSEC. Now that you know about the various record types, you can understand a more realistic view of the Chain of Trust. Here we look at the chain for my host **`www.tiltedwindmillpress.com`**, generated by **`DNSViz.net`**. (We'll look at other DNSViz graphs in Chapter 3.)

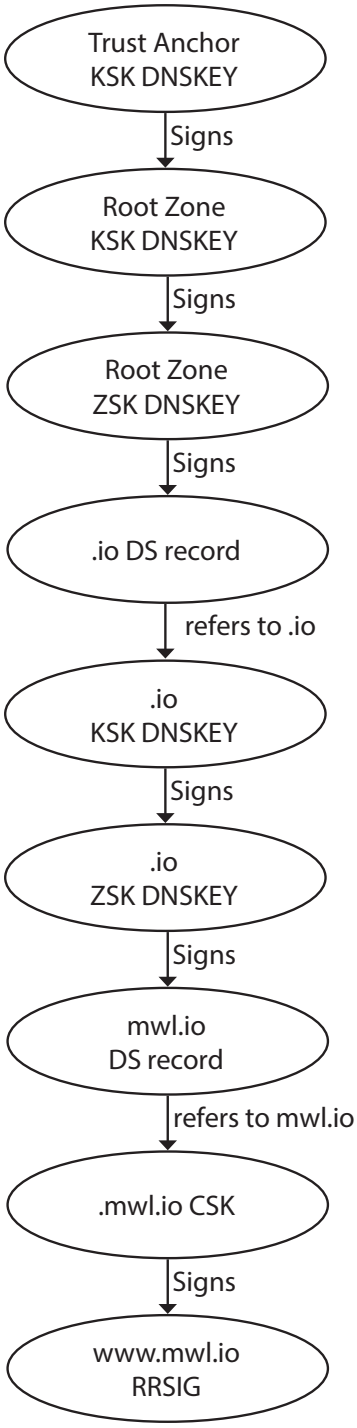

Figure 2-1: DNSSEC Chain of Trust

The start of the chain is the trust anchor configured in your resolver. It's used to validate the KSK for the root zone (.) The root zone's KSK signs the root zone's ZSK. So far, all of this is completely internal. The next link in the chain is the **.com** DS record, provided to the roo zone by the **.com** administrators. The **.com** DS record is in the root zone and is signed by the root zone ZSK. The chain of direct signatures ends there, and the client must proceed to a Secure Entry Point (SEP) for the **.com** domain.

A client working its way down to **www.tiltedwindmillpress.com** would proceed to the authoritative nameservers for **.com**. The DNSKEY record for the **.com** KSK matches the **.com** DS record in the root zone, so it's a SEP. The chain continues.

The **.com** KSK has signed the **.com** ZSK, which in turn signed the **tiltedwindmillpress.com** DS record.

The client would advance to the **tiltedwindmillpress.com** nameservers. As the threat profile for **tiltedwindmillpress.com** is considerably different than that for the root zone, this domain uses a combined signing key rather than separate KSKs and ZSKs. The CSK matches the signed DS record in **.com**, so it is a SEP. The CSK has signed the resource records for **www.tiltedwindmillpress.com**. We have a complete Chain of Trust.

This further illustrates why clients validate each Chain of Trust from the bottom-up. A client that's been running for even a short while has probably already cached the DNSKEY records of major top-level domains like `.org`, `.net`, and `.beer`. The client can grab the records for the specific target host and search upwards, quickly hitting something it already knows is valid.

Publishing DS Records

Before anyone can verify a domain's signature, the domain must make its DS records available in the parent zone. If I want to secure `tiltedwindmillpress.com`, I must send my DS records to `.com`. This isn't terribly different from registering your nameservers: you go to a web form at your registrar, enter the information, and wait for the updates to propagate.

You *must* sign your zone before publishing those DS records. If your parent zone has DS records for your zone, but your domain hasn't signed its records with the matching key, DNSSEC validation will break. The parent zone's DS records are an authoritative declaration that the child zone is signed with that key, so any records that are not signed are clearly bogus and should be rejected. Your zone will disappear from the Internet.

If you ever remove DNSSEC from a zone, start by removing the DS records. Only strip signatures from the zone after those records have had enough time to expire from everyone's cache.

Publish your zone's DS records in the parent zone, via your registrar, ISP, or RIR.

Your domain registrar is the central point for managing public information about your domain, including DS records. Unfortunately, not all registrars support DS records. If your registrar doesn't support DS records, you must switch registrars to implement DNSSEC.

Implementing DNSSEC on your reverse zones requires entering the DS record into the parent zone for your addresses. If you have your own address space, do this at your Regional Internet Registry (RIR). These organizations include ARIN (North America), RIPE (Europe), APNIC (Asia-Pacific), LACNIC (Latin America), and AFRINIC (Africa). The RIRs all support managing DS records from their web interface.

If you don't own your address space, however, you must ask the owner to manage your DS records. If they do not yet support DNSSEC they can't support the Chain of Trust internally. If you have an IPv4 /24 or larger, though, they can enter DS records on your behalf at the RIR. Whether or not they *will* is a separate matter. If your ISP cannot or will not support DNSSEC, encourage them to join the modern millennium.

Now that you have a basic understanding of how DNSSEC works, and the external requirements to fully implement it, let's look at DNSSEC in the real world.

Chapter 3: Examining DNSSEC

You've seen the different types of record and have a basic idea of how the Chain of Trust works, but nothing replaces looking at real world implementations. DNS information is public, so we can study how other domains implement DNSSEC.

Start by configuring your caching resolver to validate DNSSEC, then we'll examine various output.

Clients and DNSSEC

Most desktop systems have a stub resolver–they send DNS queries to a caching recursive nameserver, but don't process the responses or cache them. They do not validate DNSSEC on their own. If the local recursive nameserver claims that a DNS response is signed, that's good enough for them.

If the attacker is on the network between the caching nameserver and the client, it's possible that they could spoof results from that recursive server. The most technically correct solution is to have stub resolvers validate DNSSEC, but that hasn't yet become standard.

Some applications are working around the lack of stub resolver DNSSEC validation by using DNS over TLS (DoT) or DNS over HTTPS (DoH) to securely contact the caching resolvers. We discuss these in Chapter 9.

If you want a Unix workstation to validate DNSSEC, enable the local nameserver and configure it to only accept queries from the local host. Some Linuxes do validate DNSSEC at the desktop.

DNS Troubleshooting Tools

Client tools do not validate DNSSEC, but you'll need a client to query validating resolvers. We'll focus on the most widely used DNS tool, dig(1). It's developed by authors of BIND, and is generally standards compliant. If you're a regular dig user, make sure you don't have any options to hide debugging output in your `.digrc`.

Another popular and reliable tool is drill(1), developed by NLNet Labs—the same folks who brought us Unbound and NSD. The command line options are different than dig, but if you're comfortable with drill there's no reason to switch.

Avoid using either host(1) or nslookup(1) when troubleshooting DNS. The host command is meant for very simple checks, and lacks the detail and discretion of either dig or drill. The nslookup command has been deprecated, abandoned, resurrected, slain, rebuilt, re-implemented, re-abandoned, deployed, consigned to an abandoned factory outside the village of Something Awful, found guilty of one moving violation as well as two hundred seventy-four counts of Malicious Lingering, and integrated into Microsoft Windows. I have no way to tell which of the innumerable versions of nslookup you're using or the quality of that implementation. I can tell you that some nslookups that claim to validate DNSSEC do not. Do not use it.

There's other tools. BIND 9.10 introduced delv(1), a DNSSEC-aware next-generation dig(1). Knot DNS offers kdig(1), and Unbound has unbound-host(1). So long as it's not host(1) or nslookup(1), use whatever tool is most convenient for you.

Resolver Configuration

Most resolvers can enable DNSSEC validation with a simple option, check box, or other simple toggle. Modern versions of BIND perform DNSSEC validation by default, but as that knowledge keeps falling out of my brain I prefer to explicitly declare that support in `named.conf`.

```
dnssec-validation auto;
```

Put this statement in the global options and reload named(8).

Verifying DNSSEC requires using a client like dig(1).

dig(1) Essentials

Dig is the most commonly used tool for examining DNS data, so I'll use it for the detailed examples. Tools like drill present the same information in different format, so if you prefer another tool, follow the concepts rather than the exact format and options. Configure `/etc/resolv.conf` to point at your validating resolvers.

Let's run a basic query and study the results from a DNSSEC perspective.

```
$ dig mwl.io

; <<>> DiG 9.16.16 <<>> mwl.io
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36121
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1
```

We first see the version of dig we're using (9.16.16) and the main subject of the query. Dig's release numbering matches the version of BIND it shipped with. Any additional options used on the command line also appear here. Basic DNS runs over UDP, so DNS uses the id number to match responses to queries.

The *HEADER* line declares what sort of operation dig is performing, and if the DNS server returned an error. This particular request is a QUERY, and the status of NOERROR means that the server has an answer for you. NOERROR doesn't mean that the answer is fresh, authoritative, or not about to expire from cache. It merely indicates that it has an answer.

The *flags* line offers specific details about the query. If you're debugging DNS, let alone DNSSEC, the flags are your new chum. They spell out exactly what happens. The flags you'll see include `qr`, `aa`, `rd`, `ra`, `ad`, and `cd`.

The *qr* flag indicates a Query Response. This message is the answer to a query, and should always be present in dig results.

The *aa* flag gets set for an Authoritative Answer. The flag appears if and only if you query a domain's authoritative nameserver.

The *rd*, or Recursion Desired, flag appears when the client asked the nameserver to find an answer even if the nameserver needed to perform a recursive search.

If the nameserver is willing to perform that recursive search, you'll see the *ra* flag for Recursion Available.

The *ad* flag stands for Authenticated Data. The zone is signed and the recursive nameserver validated the response for you.

The *cd* flag indicates Checking Disabled. The client has requested the nameserver not validate DNSSEC.

Our example has the flags `qr`, `rd`, `ra`, and `ad`. This is a response to a query. We've asked for recursion, and recursion is available. Finally, this is a DNSSEC-validated response.

The QUERY entry shows what we're asking for. We sent one query.

The ANSWER field shows how many responses we got. Here, we receive one answer. If there are multiple records, as is common when you query for "any" record, the number of records appears here. If it's a zero, the type of record you're looking for does not exist.

The ADDITIONAL field indicates the presence of extra stuff.

```
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;mwl.io.                IN      A
```

The next few lines are for Extended DNS (EDNS), or DNS using larger UDP packets. We're on version 0 of extended DNS. Only one *flag* has been defined so far: *do*, for DNSSEC OK. It gets set when the client specifically requests DNSSEC validation, and doesn't appear during normal queries. At the end, we see that the network between client and server can handle up to 4096-byte UDP packets. That's more than sufficient.

If you're running a modern dig with RFC 8914 support, and querying a recursive server with that same support, you will see any extended DNS errors (EDE) beneath the EDNS statement. If present, these errors will tell you exactly why the nameserver rejected the response.

In the Question section, dig repeats the query.

```
;; ANSWER SECTION:
mwl.io.   7200    IN      A       45.63.79.197

;; Query time: 4 msec
;; SERVER: 127.0.0.1#53(127.0.0.1)
;; WHEN: Fri Jul 16 12:05:07 EDT 2021
;; MSG SIZE  rcvd: 65
```

At last, the answer. Dig provides the complete RRSet for your query. The host **`mwl.io`** has a single IP, so you get one answer.[6] We also get debugging information on how the query performed and where it was processed.

If you want to see the DNSSEC information in the zone, add the `+dnssec` flag. I'm also adding `+nocomments` to remove unnecessary detail.

[6] My readers have not purchased so many of my books that I must buy a load balanced server farm to cope with the demand. I sincerely thank every one of you for not condemning me to that fate.

```
$ dig isc.org +dnssec +nocomments
…
                IN  A
isc.org.  60  IN  A       149.20.1.66
isc.org.  60  IN  RRSIG  A 13 2 60 20210713170049
  20210613160056 27566 isc.org. pxHny0SKMI0ih9LWCW/11e0YMBa-
  ryH4cFjYR/f/1Tr/XSh+WqvDkuEhk ljopRVOqGeBrDvRmdYYUM2k+d-
  M+a7Q==
…
```

The presence of the RRSIG record indicates this zone is signed with DNSSEC. Just because the zone was signed does not mean that your recursive server validated the answer, however. Only the presence of the *ad* flag indicates validation.

If you want to see that the DNSSEC records are present, but don't care about the long strings of hashes and keys, add the `+nocrypto` flag.

DNS Errors

In the HEADER section of this example, the `status` field shows NOERROR. Dig went out, found data, and returned it to us. Everything worked. Maybe we got a useful answer. Maybe we didn't. But the system worked. Status errors indicate that the system did not work.

You might see a status of SERVFAIL. The nameserver did not have a cached response to this query and could not fetch one from the authoritative server. Traditionally, this means that the authoritative servers are either down or misconfigured. When a zone has DNSSEC, though, you'll get a SERVFAIL when your nameserver gets an answer but cannot validate it.

If the specific record you're looking for does not exist, and nothing by that name exists, the nameserver will return NXDOMAIN. This is an authoritative declaration that the specific thing you asked is not there. There is no host **`www2.mwl.io`**, and if you ask for it my authoritative server will say "nope, NXDOMAIN."

The NXDOMAIN differs from an authoritative answer with no records of the type you want. Remember the ANSWER field from early in the response? You might get an authoritative answer that says NOERROR, but has an answer of 0. That indicates records with that name exist, but not the sort you requested. Suppose I ask for an MX record for **`www.mwl.io`**.

```
$ dig www.mwl.io mx
…
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 45341
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 1
…
```

There is no error, but my query has zero answers. There are no MX records, but A and AAAA records do exist. Dig is saying "I can find records, but not the kind you're asking for." This doesn't mean that the host **`www.mwl.io`** lacks incoming mail, only that it doesn't have its own MX record. Incoming mail will use the zone's MX record.

In October 2020, RFC 8914 defined a variety of new DNS error messages and an extensible method for adding more. These errors include messages like "DNSSEC Bogus" and "Forged Answer," which will reduce the ~~fun~~ difficulty of diagnosing DNS issues. They will appear over the next few years.

Digging DNSSEC

To view a zone's DNSSEC information, add the `+dnssec` option. Here I examine the DNSSEC records on **`isc.org`** with my validating resolver. As these folks are DNSSEC gurus, I expect their zones to work correctly.

Two options help make the output more legible. Adding `+multi` splits the output across multiple lines. The `+nocrypto` option doesn't hide the DNSSEC records, but does omit the key information that you probably don't want to read anyway.

```
$ dig +dnssec +multi +nocrypto isc.org
…
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 29878
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1
…
```

We've received the `ad` flag, so this is authenticated data. The answer validated with DNSSEC. Look further down for the signature, shown in an RRSIG (Resource Record Signature) record.

```
…
;; ANSWER SECTION:
isc.org.  6794 IN DNSKEY 257 3 13 (
     [key id = 7250]
     ) ; KSK; alg = ECDSAP256SHA256 ; key id = 7250
isc.org.  6794 IN DNSKEY 256 3 13 (
     [key id = 27566]
     ) ; ZSK; alg = ECDSAP256SHA256 ; key id = 27566
isc.org.  6794 IN RRSIG DNSKEY 13 2 7200 (
     20211123004729 20211024000151 7250 isc.org.
     [omitted] )
isc.org.  6794 IN RRSIG DNSKEY 13 2 7200 (
     20211123004729 20211024000151 27566 isc.org.
     [omitted] )
```

If a validating resolver cannot validate the RRSIG record, it returns a SERVFAIL error. To end users, this appears as if the host has vanished from the Internet. Such a zone is called *bogus*. I show assorted failures, and how to diagnose them, in Chapter 6.

A non-validating resolver will show a zone's DNSSEC records if you request them, but it isn't validating that signature. That's what makes a non-validating resolver non-validating. Non-validating resolvers always return any data the authoritative server has for the zone, even if the signature is invalid. Or, if you prefer: non-validating resolvers behave exactly as they always have.

DNSSEC versus Authoritative Servers

Ask ISC's authoritative servers about the zone **`isc.org`** and check the flags.

```
$ dig +dnssec +multi isc.org @ns1.isc.org
…
;; flags: qr aa rd; QUERY: 1, ANSWER: 2, AUTHORITY: 0, ADDITIONAL: 1
…
```

This is an authoritative server for the zone, but its response does not include the `ad` flag. You do get the `aa` flag, which represents authoritative answer. This authoritative answer, straight from the source, is somehow not authenticated. Shouldn't a response directly from the main nameserver be cryptographically secured?

An authoritative answer directly from the authoritative server should not be authenticated data. It is not the authoritative nameserver's job to compute DNSSEC validation for you. That's the validating resolver's job. The authoritative server does its job by providing the signatures for others to validate.

Separating resolvers from authoritative servers is a strongly recommended best current practice. Don't try to troubleshoot the DNSSEC on your zones by testing validation on your authoritative servers, it won't work.

Disabling Validation

Your resolver will return SERVFAIL when the authoritative nameserver is busted, and also when DNSSEC validation fails. When troubleshooting, you must be able to differentiate between the two types of error. The easiest way to figure out which sort of error you have is to disable DNSSEC validation with the `+cd` option.

I recommend seeing how this works in a test environment before you have to troubleshoot it for real. Comcast provides the domain **`dnssec-failed.org`** for exactly this purpose. Run dig on this domain.

```
$ dig dnssec-failed.org
…
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 34316
…
```

A SERVFAIL error with no other results, as expected. Now try the same query, with checking disabled.

```
$ dig dnssec-failed.org +cd
…
dnssec-failed.org.      7197    IN      A       69.252.80.75
```

Getting a response when you disable DNSSEC indicates that the domain's signature is invalid.

Now that you can examine DNSSEC data, let's remedy one of the weak points in authoritative nameservers.

Chapter 4: Securing Zone Transfers

Having your primary server digitally sign zones is great, but if an intruder can corrupt your zone data as it travels between your primary and secondary servers, they can impact the zone's availability. Before deploying DNSSEC on your zones, secure your server's zone transfers. *Transaction Signatures* (TSIG), defined in RFC 2931, can protect the integrity of dynamic updates.

Purists will note that TSIG is not part of DNSSEC. If you're going to the trouble of signing a zone, however, you should certainly protect that data as it goes between authoritative servers. And you should do it before deploying DNSSEC—or, if you've already signed your zones, now would be good.

TSIG works via symmetric encryption. The servers share a key and a key name. Most nameservers allow you to restrict use of that key to particular IP addresses, so an intruder who steals your key still can't use it. While imperfect, it's a significant improvement over purely IP-based access control.

As the name indicates, TSIG signs transactions to preserve integrity. It does not encrypt data. If you want confidentiality between your nameservers, set up a VPN. (While RFC 9103 defines a standard for zone transfers over TLS, it is not yet widely deployed.)

Create a different secret key for every secondary server. If an intruder compromises a secondary server, you need replace only the keys stored on that host. If an intruder breaks into your primary nameserver, you'll still need to replace all the TSIG keys. You'll also need to immediately rotate zone keys, and replace all the keys for other secure protocols like SSH, HTTPS, LDAP, and so on.

Many nameserver programs offer alternatives to TSIG, exactly as they offer replication by means other than zone transfers. Feel free to use them. If you need to interoperate with other nameservers, however, you need to understand TSIG.

TSIG Essentials

Each TSIG key has a secret and a name. Both must be the same on every host that uses the key. Mismatched names, like calling a key "secondary" on the primary and "primary" on the secondary, cause TSIG failures.

TSIG supports several different algorithms, including the strongly discouraged MD5 and SHA1. All keys should use algorithms in the SHA-2 family, such as SHA256 or SHA512. Longer hashes are harder to break, but burn more processor time in routine transactions and are a waste for most sites.[7]

TSIG is one of those protocols that tends to get set up once and forgotten. If you have servers using TSIG, verify that they are not using HMAC-MD5 or HMAC-SHA1. If you find those keys, replace them as soon as possible.

Create the key on one host. Copy the key and the name to the other host. Tell your nameservers about the keys. Test with zone transfers or other queries. That's everything. There's really no reason not to use TSIG.

TSIG requires synchronized time, such as provided by NTP.

TSIG and BIND

Using TSIG requires creating keys for each pair of hosts, telling named about the keys, and assigning the key to a specific server.

Key Creation

To create a key you must know the HMAC algorithm and the key name. I recommend HMAC-SHA256 for all current keys. Create TSIG keys with tsig-keygen(8).

```
# tsig-keygen -a algorithm keyname
```

Here I create an HMAC-256 key named after DNS server `dns2`.

```
# tsig-keygen -a HMAC-SHA256 dns2
key "dns2" {
     algorithm hmac-sha256;
     secret "bYIwx0554io3Pn4UeiFwZsGH8o4iayzrLuyS6b69Bu8=";
};
```

This command produces your key.

[7] Longer TSIG keys don't even give you geek bragging rights the way 4096-bit X.509 certificates do, because only you can see them.

Key Configuration

The output of tsig-keygen(8) is formatted as a *named.conf* entry. Don't change anything about the output; place it verbatim into the configuration file. (You could also use it as an include file.)

Now attach the key to a nameserver with a `server` statement, like so. The secondary server's *named.conf* should reference the same IP address configured as the primary server, while the primary uses the secondary's address. (If the secondary server has multiple IP addresses, use the address that shows up as the source address of connections to the primary server.)

```
server 192.0.2.1 {
     keys { dns2 ; } ;
};
```

Restart named or run `rndc reconfig`. It will use these keys to transfer zones, as you can verify in the debugging log.

dig Versus TSIG

I commonly use dig(1) on a secondary nameserver to verify zone transfers. Once you configure TSIG, the primary nameserver expects all zone transfer requests from the secondary's IP address to be signed. To run a zone transfer at the command line, you must provide dig with the TSIG key.

The easiest way is to create a file containing the key and feed it to `dig` with the `-k` argument. The key file must look like the output of `tsig-keygen`. If you're running BIND, you can copy the key statement straight from *named.conf* into your key file. With other nameservers, you must arrange the key name, HMAC, and secret in a file. Then try your zone transfer as usual, adding the `-k` argument and the filename.

```
$ dig zone @server axfr -k tsig.key
```

The problem with a key file is that contains the secret key. You must protect that file exactly as you would any other secret key. That's often inconvenient and might violate your organization's security policy. You can also provide the HMAC algorithm, name, and secret on the command line with the `-y` flag.

```
$ dig zone @primary axfr -y HMAC-algorithm:keyname:secret
```

It's a longer command line, but less persistent than the file. Unless it's cached in your *.history*. Choose your risk.

Now let's take a look at keys.

Chapter 5: Signing Zones

Now that you understand some basics about DNSSEC, you can enable it. Managing DNSSEC is far easier than it used to be. All broadly deployed nameservers can turn it on with a configuration setting on the nameserver, and most automatically generate and rotate keys for you. There are exceptions—NSD can serve signed zones, but can't generate keys or sign the zones. The vast majority of blog posts about managing DNSSEC are now obsolete.

Perform all DNSSEC configuration on the primary nameserver. Secondary nameservers trust their primary nameservers and will accept the new zone files without any configuration changes. Again, I strongly encourage using a test domain for your first deployment.

The most critical part of managing DNSSEC is the publication of DS records. No matter which nameserver you run, wait to publish the DS records until the zone is signed.

We'll dive into configuring DNSSEC on BIND, but first let's talk about some generic signing concepts.

Key State

You can create keys without publishing them in a zone, or remove keys from a zone, or use those keys to sign the zone. Keys have four formal *states* to identify how they're being used.

A *rumoured* key has been published, but not fully propagated yet. Other nameservers might have older records without this key in cache. You cannot yet rely on a rumoured key.

An *omnipresent* key has been available for long enough that it should be in everyone's cache. Nameservers do not query remote nameservers to determine this, but rather calculate it from the TTL of your zone. If a remote nameserver is playing games with their cache and not flushing obsolete data, the key will still show as omnipresent.[8]

8 I haven't seen this since the 1990s, but I have no doubt that any day now a sysadmin whose cleverness exceeds his knowledge will reinvent it.

An *unretentive* key is on its way out. You've removed it from the zone, but it might still be in remote nameserver caches.

A *hidden* key is not yet published, or has been unpublished for so long that it should have expired from remote nameserver caches.

Signing Zones with BIND

BIND lets you turn on DNSSEC in a zone with a single configuration statement, but don't do it yet unless you want a bunch of new files everywhere. Take a moment to set up an organization first.

Don't let the rush of successfully signing your first zone goad you into immediately signing all your zones. I encourage you to review the next three chapters before a mass deployment of DNSSEC.

Key File Management

Each signed zone starts with two key files and a state file on the primary nameserver. Key files accumulate as you rotate keys. I strongly recommend giving each zone a dedicated key directory named after that zone. The keys are managed by named(8), so the directories must be owned by the user `named` runs as. The directory */etc/namedb/working* is set aside for files and directories created by `named`, so create a *keys* subdirectory and per-domain directories below it. Use the `key-directory` configuration option to set the zone's key directory.

```
zone "mwl.io" {
     type primary;
     key-directory   "/etc/namedb/working/keys/mwl.io";
…
};
```

If you have only a few zones you could create a single key directory and set `key-directory` globally, but per-domain directories are easier to manage and help prevent accidents.

Zone Management

You might manage your zones through dynamic DNS and nsupdate(1), or through static zone files. DNS administrators are often violently attached to one or the other. Both work with DNSSEC, but in different ways.

If you're using dynamic DNS, BIND already maintains your zone file and the related journal files. It can add the DNSSEC records to the zone without any special intervention from you.

Static zone files are trickier. One of the major points of a static zone file is that `named` can't update it. Signing a zone requires inserting records into the zone. BIND evades this limitation by converting your meticulously commented zone file into a dynamic zone internally. When you edit and reload your zone file, `named` creates a dynamic zone file and journal files for it, all without besmirching your precious hand-edited zone files. Older versions of BIND called this *inline signing* and had a configuration option for it, but it's now the default for signing static zones.

One consequence of inline signing is that the serial number served to the world is not exactly the same as what's in your static zone file. When `named` renews signatures on records, it must also increment the zone's serial number. Named manages the serial number on its own, however. When you edit the zone file, you must bump the serial number for `named` to notice the changes.

To create the additional files needed for static zones, the user `named` runs as must have write permissions on the zone file directory. It doesn't need write permissions on the static zone file, however.

DNSSEC Policies

Should a zone use a separate KSK and ZSK, or is a CSK adequate? How often should signatures be renewed? When should a key expire, and what kind of key should replace it? These are all policy decisions. BIND supports creating named key and signing policies (KASP) that contain all of these settings.

The default policy uses a single CSK that never expires. This simplifies initial DNSSEC deployments, but you'll probably want something less general over time. We'll dissect KASP in Chapter 8, letting you set your own policies.

Signing Zones

Tell named to sign a zone with the `dnssec-policy` configuration option.

```
zone “mwl.io” {
     type primary;
     key-directory   “/etc/namedb/working/keys/mwl.io”;
     dnssec-policy default;
…
};
```

Run `rndc reload`. BIND will create key files and sign the zone.

If you sign a zone contained in static zone files, `named` creates three new files for DNSSEC-aware versions of the zone. If your zone is in the file *`zonename`*, you'll also see *`zonename.jbk`*, *`zonename.signed`*, and *`zonename.signed.jnl`*. The *`.jbk`* file is a backup journal file, used for the internal conversion to a dynamic zone. The *`.signed`* file is your zone with DNSSEC signatures dynamically added, and the *`.signed.jnl`* file is the journal file for the signed zone.

Verify Signatures

Before proceeding, double-check that your nameserver signed the zone. Signatures are RRSIG records.

```
$ dig www.mwl.io @localhost +dnssec
```

The response should contain an RRSIG record, but it will not have the `ad` flag set. Your parent domain doesn't yet have your DS record. Let's fix that.

Key Files

Go to your key directory for this domain. You'll find three files starting with *`K`*, indicating that they're keys or related to keys. The files use the naming standard:

K*`domain.+algorithm+tag.purpose`*

For example, here are the files for my domain **`mwl.io`**.

```
Kmwl.io.+013+44950.key
Kmwl.io.+013+44950.private
Kmwl.io.+013+44950.state
```

This key uses algorithm 13, or ECDSAP256SHA256 (as per Chapter 1). The key tag is 44950. Every key has a five-digit tag. Tags are not universally

unique, but do help differentiate keys within a zone. If you run DNSSEC long enough and roll over your keys regularly, a domain will eventually get issued a second key with that same ID. The original key with that tag will long since have expired, though.

The file ending in *.private* is the private key. The *.state* file contains details on the zone's status. Only keys created by a policy have *.state* files. Most interesting is the *.key* file, which contains the DNSKEY record for this key and comments about the key.

```
; This is a key-signing key, keyid 44950, for mwl.io.
; Created: 20210824194856 (Tue Aug 24 15:48:56 2021)
; Publish: 20210824194856 (Tue Aug 24 15:48:56 2021)
; Activate: 20210824194856 (Tue Aug 24 15:48:56 2021)
; SyncPublish: 20210825205356 (Wed Aug 25 16:53:56 2021)
mwl.io. 3600 IN DNSKEY 257 3 13 8Xuo…
```

Chapter 7 discusses key timing.

Verifying DNSSEC

Before publishing your DS records, use the `rndc dnssec` command to verify that your zone is truly signed. The `-status` option and a zone name gives you your nameserver's opinion of the zone's DNSSEC health. This check only works on zones configured with a `dnssec-policy` statement.

A successful status check does not mean that the outside world can validate your zone. It only means that your nameserver's local DNSSEC setup is fine, and if you publish its DS records correctly the zone *should* validate.

```
# rndc dnssec -status mwl.io
dnssec-policy: default
current time:  Wed Sep  8 11:54:36 2021

key: 44950 (ECDSAP256SHA256), CSK
     published:      yes - since Tue Aug 24 15:48:56 2021
     key signing:    yes - since Tue Aug 24 15:48:56 2021
     zone signing:   yes - since Tue Aug 24 15:48:56 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             rumoured
     - zone rrsig:     omnipresent
     - key rrsig:      omnipresent
```

This zone uses the default DNSSEC policy. We'll change this in Chapter 8.

You'll see the current key (a CSK) and the dates it was published and when it began to be used. This information is also recorded in the key's state file. Key 44950 has been in use for long enough that it should be available everywhere, but the DS record shows as in the "rumoured" state. You haven't published your DS record yet, so the rumoured state makes sense.

Let's fix that.

Publishing DS Records

Once your zone is signed and you have the RRSIG and DNSKEY records, you can tell the parent zone about your key. Every registrar that supports DNSSEC has a place on their web site for entering key information on your domains. The registrar will ask the actual domain registry to publish a DS record for your zone, including a hash of your key and the hash algorithm. Some registrars want you to provide the DS record, while others want the DNSKEY record.

If you provide the DNSKEY record, the registrar computes the DS record itself. When the registrar needs to change hash algorithms, it can perform that computation without involving the domain owner. A shocking percentage of keys deployed in the 2000s and early 2010s have never been rotated, and their DS records use weak hash algorithms. By requiring the DNSKEY record, these registrars future-proof their records.

Others expect you to send a complete DS record, because they believe you're competent and will rotate your keys yearly. If you're using BIND, extract the DS record from a key file with the dnssec-dsfromkey(1) command. This command takes the key file as an argument and returns a fully formatted DS record.

```
# dnssec-dsfromkey Kmwl.io.+013+44950.key
mwl.io. IN DS 44950 13 2 00A1F9EA9A5A37C2D9094BAD3DFE234A…
```

The default algorithm is SHA-256.

Provide this record to your registrar. Once the DS record propagates everywhere, you will have working DNSSEC.

Chapter 7 discusses timing considerations in publishing DS records, but these don't apply the first time you configure DNSSEC on a zone. No nameserver on the Internet has ever attempted to validate DNSSEC on this domain, so they won't have old RRSIG records cached. If you're adding a new key or removing an old one, however, read Chapter 7 closely.

If you're using BIND, you can update the key state. This isn't useful for zones that never perform key rollovers, but we'll need it in Chapter 8 so let's do it now.

```
$ rndc dnssec -checkds published mwl.io
Marked DS as published since 08-Sep-2021 12:16:46.000
```

The key will still show up as rumoured until enough time has passed for the parent zone to expire from remote nameservers' caches.

Now that you have a signed zone, let's see how DNSSEC can go wrong.

Chapter 6: Failure Modes and Troubleshooting

DNSSEC feels like a high-stakes protocol. If you configure everything correctly, clients will get extra assurance that they're reaching the correct host. If you do something wrong and your DNSSEC doesn't validate, your domain disappears from the Internet. This will make you popular.[9] Only two types of sysadmin errors break DNSSEC: key mismatches and incorrect clocks.

Start debugging DNSSEC by determining if it's really a DNSSEC issue.

DNSSEC and the Network

I want to say that network equipment has caught up to the standards of the 2000s, and there's no way modern gear could interfere with DNSSEC. I really want to say this. I cannot. If your new DNSSEC deployment doesn't work, or someone swapped out a device on your network and everything broke, check for network issues.

DNS resolvers originally used only small packets on UDP port 53. Since 1999 DNS has used large UDP packets as well as TCP, and can switch between them as the query, client, and network conditions demand. Naïve packet filters imposed by many bottom-feeding routers, IoT devices, and network administrators, can interfere with modern DNS. In the last twelve months, in "friendly" devices, I've seen "Allow RFC 2671 DNS" GUI checkboxes that unforgivably default to "off."

Verify your network's ability to pass modern DNS traffic. From your nameserver, run a large dig query. I'll use OARC's reply size tester, as discussed in the Introduction.

[9] Everyone in your organization will want to talk to you. That's the same thing as "popular," right?

```
$ dig +short rs.dns-oarc.net TXT
dig +short rs.dns-oarc.net TXT
rst.x4050.rs.dns-oarc.net.
rst.x4058.x4050.rs.dns-oarc.net.
rst.x4064.x4058.x4050.rs.dns-oarc.net.
“149.28.122.136 sent EDNS buffer size 4096”
“149.28.122.136 DNS reply size limit is at least 4064”
```

This part of the network can pass DNS messages of up to 4096 bytes. It's fine.

```
$ dig +short rs.dns-oarc.net TXT
rst.x1196.rs.dns-oarc.net.
rst.x1206.x1196.rs.dns-oarc.net.
rst.x1212.x1206.x1196.rs.dns-oarc.net.
“2001:19f0:5401:1eb3:5400:1ff:fef9:2ab9 sent EDNS buffer size 1232”
“2001:19f0:5401:1eb3:5400:1ff:fef9:2ab9 DNS reply size limit is at least 1212”
```

This part of the network can pass messages of up to 1232 bytes. This network is *not* fine.

Turn the query around and interrogate your nameserver from the outside world. While a variety of web sites claim to offer such service, many strip DNSSEC from their output. DNS administrators must have access to debugging tools outside their network. Spin up a temporary virtual machine on any cloud service and check. The output won't exactly match what you get running an identical query on the nameserver itself, but the size alone will tell you if the network is blocking traffic.

If you discover a bottleneck, run queries at various points through your network until you identify the offender. If you're the network administrator, you might find a packet sniffer useful.

If the network can pass large DNS messages, your problem is in your DNS.

DNSSEC States

DNSSEC responses have three possible states: secure, insecure, and bogus.

Secure zones have working DNSSEC. Random DNSSEC-aware resolvers can validate them. This is healthy, normal DNSSEC.

Insecure zones do not have working DNSSEC. Old-fashioned DNS data is insecure. So is signed data without a DS record. If you've signed your zone, but haven't published the DS record with your registry, your zone is insecure. The resolver returns an answer, but won't label it as authentic.

Bogus zones are signed, but the signatures are invalid. The resolver rejects the signatures and returns SERVFAIL. Bogus hosts and zones disappear from the Internet.

Two things cause most bogus zones. Your DS record might be incorrect, either by not matching any key in your zone or because you stripped DNSSEC from the zone without removing the DS record. Your signatures might have expired or might not yet be valid, because a clock somewhere went bad.

What about the bogus zones that are not caused by these two problems? Perhaps your key rollover failed, or you deleted your old DS records too soon. You might have done something to your trust anchor. Sysadmins have an unparalleled talent for discovering novel misconfigurations. Or maybe you discovered a totally unique edge case that triggered a bug in the protocol or software. Your network might even be under attack.

Let's look at a few different tools to identify the problem, starting with the prettiest.

Is It DNS or DNSSEC?

You can only have a DNSSEC issue when trying to resolve a signed zone with a validating resolver. If either the target or the resolver doesn't do DNSSEC, it's not a DNSSEC problem. It's a traditional DNS problem. Most of my DNS problems involve typos in a zone file, not the signatures.

Here's one of my domains[10] that has vanished from the outside world.

```
$ dig mwlucas.org @8.8.8.8
…
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 44646
…
;mwlucas.org.  IN  A
```

The critical part here is the SERVFAIL statement. My validating resolver says that the authoritative nameserver cannot give an answer. Perhaps the sysadmin missed a period and broke the zone, triggering a routine DNS failure. Maybe DNSSEC is fouled up beyond all recognition. Add the `+cd` flag to disable validation and see which.

[10] No, I haven't registered every possible variant on my name. I keep this domain to abuse in the same way the world abuses me.

```
$ dig mwlucas.org @8.8.8.8 +cd
…
;; ANSWER SECTION:
mwlucas.org.  3600  IN  A  45.63.79.193
```

This problem disappears when I stop checking DNSSEC. My DNSSEC cannot be validated; it is bogus. (Remember, "bogus" is a technical term.) This is not an error on the protocol's part; the keys are wrong somewhere, or the signature timestamps are not current, so the validating resolver has rejected the zone. That is the purpose of DNSSEC.

If you happen to be running a modern dig with RFC 8914 support, and querying a modern nameserver with that same support, you'll get an extended error message in under OPT PSEUDOSECTION.

```
$ dig fail03.dnssec.works
…
;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 1232
; EDE: 6 (DNSSEC Bogus)
…
```

This is great, when it's available.

Online DNSSEC Debuggers

Several web sites offer online DNSSEC troubleshooting, but the prettiest is Sandia National Laboratories' DNSViz (https://dnsviz.net). Enter a domain name and hit "Analyze." It runs a bunch of queries and produces a diagram of the domain's Chain of Trust. The top level is the current DNS trust anchor. Each level of the domain is a separate box, each link of the chain is a circle, and arrows illustrate the connections between everything. Mouse over anything and you'll get a pop-up box with low-level details.

DNSViz caches its results, so when you're troubleshooting check the page's timestamp. I highly recommend the "Update Now" button.

Secure Zones

Before you can recognize errors, you must be able to recognize normal. If I check the DNSSEC on my e-bookstore at **`www.tiltedwindmillpress.com`**, I get this.

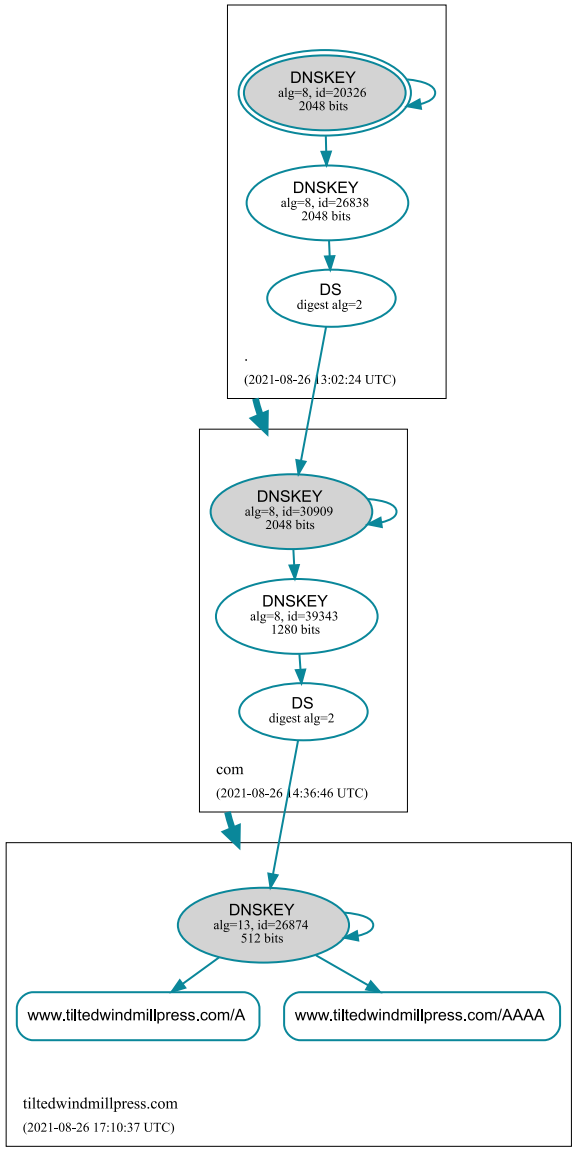

Figure 6-1: normal DNSSEC

The top box represents the root zone. At the very top we have the trust anchor configured in DNSViz's resolver. The trust anchor is a Key Signing Key, key 20326. This KSK signs the root zone's Zone Signing Key, key 26838 in the middle circle. The root zone's ZSK signs the DS record pointing at the **`.com`** zone's KSK.

The middle box is the **`.com`** zone. The top circle, key 30909, is the **`.com`** KSK. The middle box, key 39343, is the ZSK. My domain's DS record is signed by the **`.com`** ZSK.

At the bottom we have a box for **`tiltedwindmillpress.com`**. This zone has only one key, so it's a Combined Signing Key. The zone has an A and an AAAA record for the host **`www.tiltedwindmillpress.com`**, and both are signed. This zone has complete, functional DNSSEC.[11]

[11] Why am I using a host instead of a domain? Because the domain has far more than two records, which makes the image so wide it won't fit legibly on a page.

Insecure Zones

If your zone lacks DNSSEC, you can expect the Chain of Trust to trail down from the root and end in a NSEC or NSEC3 record in the zone's parent domain. Remember, NSEC records are authoritative statements of non-

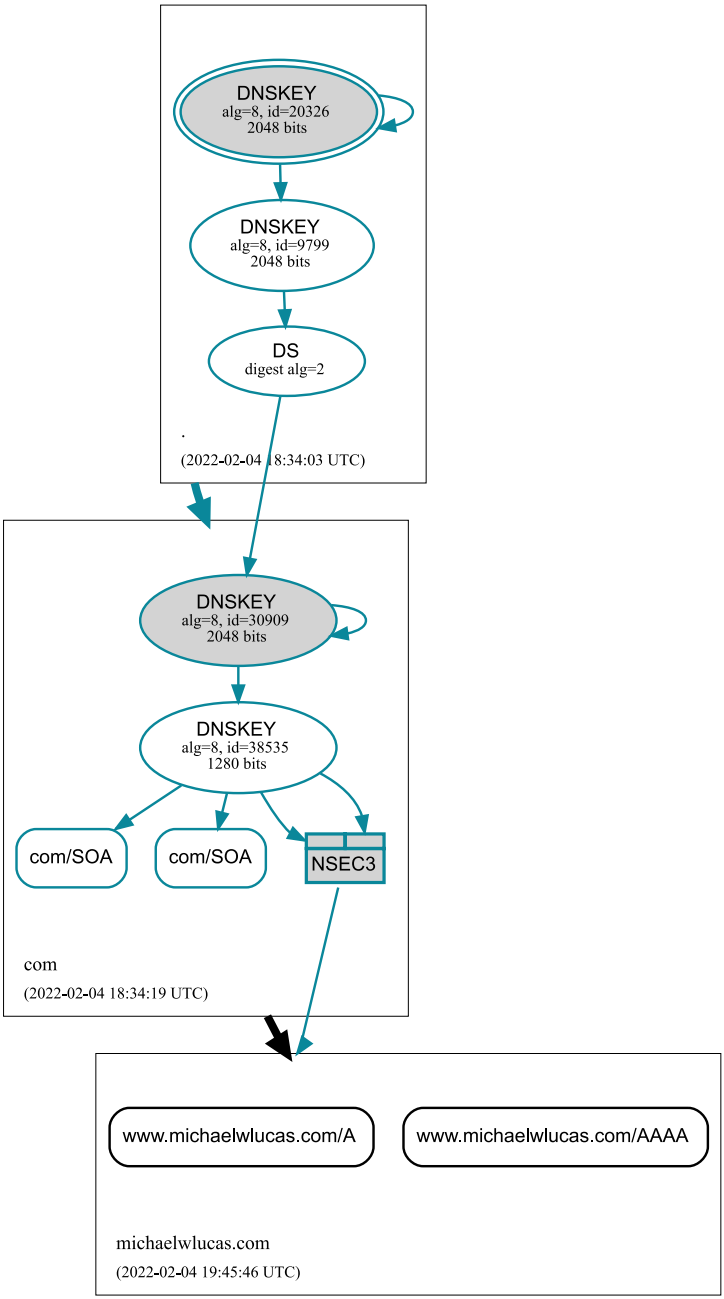

Figure 6-2: Insecure zone

existence. Here's DNSViz on what was my main domain until those hideous virtual keyboards made **mwl.io** a wiser choice.

The root zone is a secure zone. So is **.com**. The **.com** zone has no DS record for this domain, so it has an NSEC3 record declaring the domain has no DNSKEY records. And, indeed, the zone lacks DNSKEYs.

Missing DS Records

If a zone is signed, but the DS records are missing, DNSViz shows that as well. Now that I've realized this old zone lacks signatures, I've signed it.

Again, the root zone and the .com zone are secure. The **.com** zone, in the middle, still has the NSEC3 record declaring that my domain has no keys. My domain is there in the bottom box, waving its hand and declaring it does too have keys. The zone's DNSSEC is unrecognized, though, because the DS records are missing.

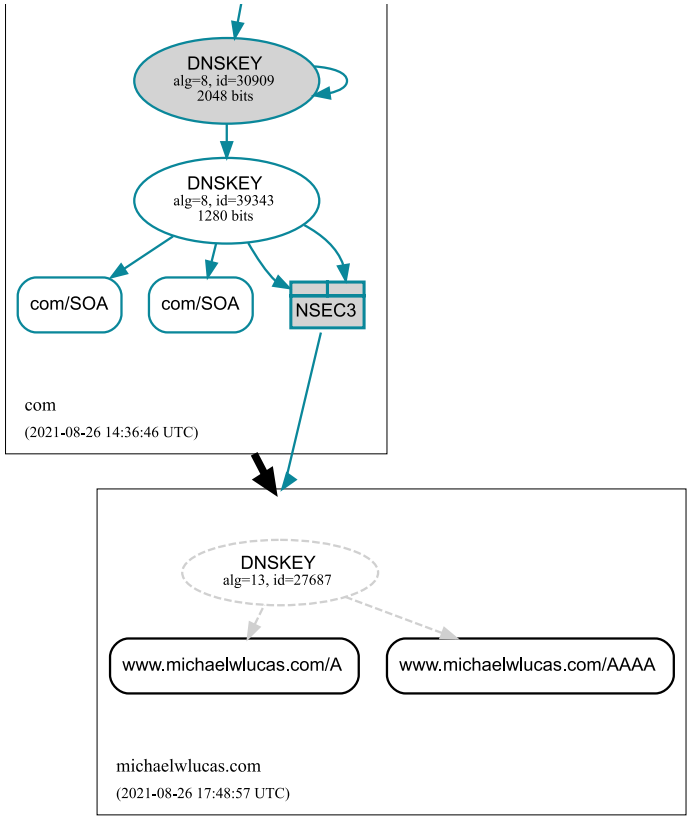

Figure 6-3: Missing DS Records

If you see something like Figure 6-3, activate your DNSSEC by publishing your DS record.

Detritus

Not all zones have pristine Chains of Trust. Here's the chain for my host **www.mwl.io**.

The boxes for my root zone and my zone show valid signatures.

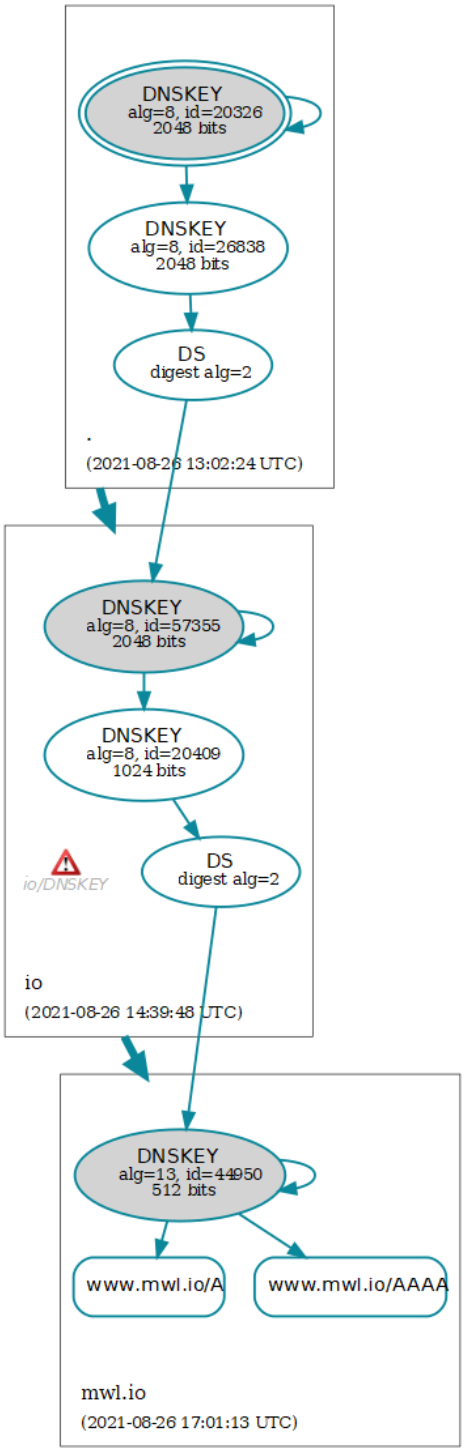

Figure 6-4: Detritus in Parent Domain

Look in the middle square of Figure 6-4, containing the records for the `.io` domain. That triangle with an exclamation point represents a problem. You'll see such detritus occasionally. It's not affecting the Chain of Trust in my domain, however. It is in my parent zone, so it is officially not my problem. So long as at least one valid path exists between the trust anchor and your zone, your DNSSEC is perfectly fine.

Bogus Zones

The secure and insecure instances above all represent correct configuration—perhaps incomplete, but not damaging. The domain **`dnssec-failed.org`** exists to serve as a bad example.

The top two boxes, the root and **`.org`** zones, are fine. The **`.org`** zone contains a DS record for **`dnssec-failed.org`**, pointing to a key with the tag 106. (If you moused over the DS record, you'd see details like the key hash.) The zone's actual DNSKEY record has the tag 29521. It doesn't match the parent domain's DS record. A validating resolver will reject it. Sometimes, the signature on this zone expires as well.

If your zone looks like Figure 6-5, fix your DS record.

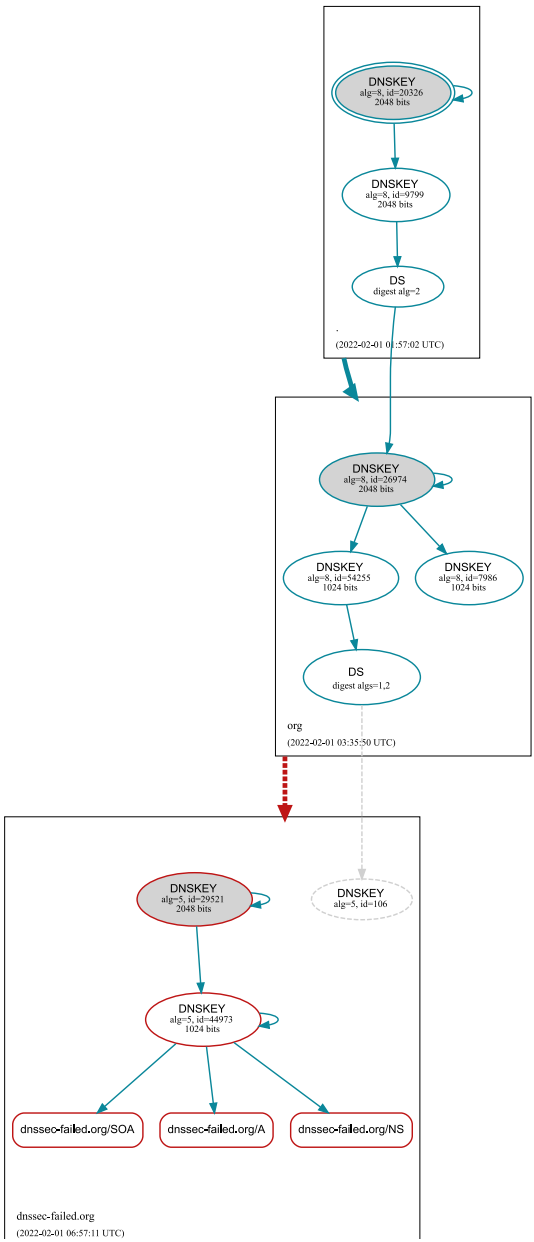

Figure 6.5: Broken DS Records

Zones with expired signatures look very similar, but feature a big red arrow. If you see this, either your nameserver's clock has skewed or the signatures have truly expired. Start by checking your authoritative server's clock.

If you poke around you'll find other online tools, such as Verisign's DNSSEC Debugger (https://dnssec-debugger.verisignlabs.com/). Pick one you like.

Debugging On Your Nameserver

Web sites are pretty, but you don't always have access to them. Once you've ruled out network problems, normal DNSSEC-specific failures can have two root causes. Either a DS record doesn't match the child domain's DNSKEY, or a clock somewhere is so wrong that the signature is not valid.[12] Debugging them requires no tools beyond dig(1), date(1), and `tail -f`.

Start with your recursive nameserver's log.

[12] Yes, you might have also tickled obscure software bugs. You must demonstrate that your problem is not a normal DNSSEC failure before making that claim, however.

Logging

The easiest way to see why a recursive nameserver refuses to validate a zone is to enable the debugging log. Here I configure a DNSSEC-specific debugging log in BIND. I keep this log enabled on my recursive servers at all times.

```
logging {
     channel dnssec_log {
        file "working/dnssec_log" size 20m;
        print-time yes;
        print-category yes;
        print-severity yes;
        severity debug 1;
        };
     category dnssec { dnssec_log; };
};
```

The first stanza creates a log entity called *dnssec_log*. It defines a file to contain the log and limits the size to 20MB. This log should include the time, category, and all messages of severity 1 (the most critical errors). BIND logs constantly rotate; when they reach the maximum size, BIND deletes the oldest data to make space for the new.

The second stanza tells `named` to send DNSSEC messages to the log called *dnssec_log*. If you spend a lot of time troubleshooting DNS, BIND includes many logging categories.

Reload `named` and run `tail -f` on the newly created log file. In another terminal, make a query you know will fail.

```
$ dig dnssec-failed.org @localhost
```

The log will tell you exactly what's going on.

```
27-Aug-2021 10:53:12.873 dnssec: info: validating
   dnssec-failed.org/DNSKEY: no valid signature found (DS)
```

"No valid signature found" sounds pretty generic, but it generally indicates a DS/DNSKEY mismatch as shown in the last section. Other errors messages are more specific. Here I dig at a host that has an expired signature in a parent domain.

```
$ dig ok.ok.sigexpired.bad-dnssec.wb.sidnlabs.nl
```

The log says exactly where the problem is.

```
27-Aug-2021 11:04:11.929 dnssec: info: validating
  sigexpired.bad-dnssec.wb.sidnlabs.nl/DNSKEY: verify failed
  due to bad signature (keyid=61837): RRSIG has expired
```

The signatures on the **ok** child zones are fine. The signature on the intermediate zone **sigexpired** has expired, and the log says so.

If you need more detail, increase the severity level in *named.conf*. Debug level 2 gives me everything I could hope for, and 3 offers more than I can cope with.

Sample Errors

It's always best to become familiar with errors before you need to debug. Setting up your own authoritative nameserver specifically to provide errors is a fair amount of work, but fortunately a few overly generous folks provide such to the public. While the most certain way to immediately shut down a public technology resource is for me to mention it in a book, I'll discuss a few here and hope some of them survive until this tome reaches you.

The best-known is the straightforward **dnssec-failed.org**, which has a mismatch between the DS record and the key.

Over at **dnslab.org**, entries **bad1** through **bad6** illustrate different failures. They all look the same in dig, but your error log will illustrate reasons.

```
$ dig bad2.dnslab.org
…
;; ->>HEADER<<- opcode: QUERY, status: SERVFAIL, id: 36184
…
```

You can get the specific reason DNSLab thinks each of these is bad by checking the TXT record at **reason.bad*X*.dnslab.org**. Be sure to disable checking so that you can see that error, and compare their explanation to your log.

```
$ dig reason.bad2.dnslab.org txt +cd
…
reason.bad2.dnslab.org. 1894 IN TXT "signatures long expired"
```

If you read German, **`https://dnssec.works/`** offers a few different types of failure. Poke around hosts **`fail01.dnssec.works`** through **`fail05.dnssec.works`**.

SIDN Labs offers an assortment of broken DNSSEC at **`https://workbench.sidnlabs.nl/bad-dnssec.html`**. Their catalog includes zones with broken intermediate zones.

Finally, you can find some broken DNSSEC at **`https://caatestsuite.com/`**. This site is intended for testing Certificate Authority Authorization records, but broken DNSSEC should certainly affect your CAA records.[13]

Pick a convenient site and play with dig. See what different errors look like. When you inevitably break your own DNS or DNSSEC, you'll treasure that experience.

Debugging with Dig

If you don't have access to the log on the recursive nameserver or any other host that can't validate the zone, you'll have to debug with only dig and date. They are more than sufficient to show you which of the two types of failure you're experiencing.

Dissecting RRSIG

Start with the zone's RRSIG records. Add the `+dnssec` flag to your dig command to view them. If the zone is bogus, add the `+cd` flag. You might add the `+multi` flag to split output between multiple lines.

```
$ dig www.mwl.io +dnssec +cd +multi
…
;; flags: qr rd ra cd; QUERY: 1, ANSWER: 2, AUTHORITY: 0,
ADDITIONAL: 1
www.mwl.io.             3421 IN      A       45.63.79.193
www.mwl.io.             3421 IN RRSIG A 13 3 3600 (
     20210908132044 20210825125200 44950 mwl.io.
     t99CfAWljvGaVkyWHnN5KqgGkPeOYN62RdgAVJb1iJAj
     sbh6gasAneE0aYe4VEjPdcsQed3rt29yX4mRYOlK7A== )
…
```

[13] If you don't know about CAA records, may I recommend you pick up a copy of my book *TLS Mastery*? Other TLS books exist, but they're all heavier and longer and contain a disquieting degree of detail.

This gets us started. Let's rip apart this Resource Record signature.

The 3421 after the record name is the time in seconds that the record will remain in cache. If you're using public resolvers, like those provided by Google or Cloudflare, you're actually querying one of a whole farm of servers.

The following IN RRSIG A indicates this is an Internet record, a Resource Record Signature record, and relates to A records. The zone might have multiple matching A records—remember, the signature is on the Resource Record Set, not an individual record. It proves that there are no other matching A records.

Next, the 13 gives the algorithm number. Algorithm 13 is ECDSAP256SHA256, one of the two current standards.

The 3 that follows is the number of records in this resource set.

The original time-to-live for this data comes next. This record can be cached for 3600 seconds, or one hour.

Next we have the date this signature expires, and the date it becomes valid. These dates use a four-digit year, two-digit month, two-digit hour, two-digit minutes, and two-digit seconds. This record expires on 20210908132044, or 8 September 2021 at 1:20 PM and 44 seconds. It is valid after 20210825125200, or 25 August 2021 at exactly 12:52 PM.

The key ID, 44950, follows.

Then we have a couple lines of actual public key. If you need to copy and paste the public key, get rid of the `+multi` option.

Bad Times

A DNSSEC zone signer automatically updates RRSIG records so that they can propagate before the older records expire. Our example RRSIG record becomes valid on 25 August, and expires on 8 September. If it's 9 September and your recursive server has this RRSIG cached, the zone will not validate. Most often, this is caused by a clock problem. Virtual hosts on overloaded hardware are especially susceptible to clock drift.

Unlike protocols like Kerberos, which allow for a small amount of clock drift, validating DNSSEC signature timestamps offers no leeway. If a signature expires at 10:04:07 today, they fail validation at 10:04:08.

Signatures get renewed early enough so that old signatures expire from recursive server caches before they expire of age.[14]

I strongly recommend automating and monitoring the time and date on all your nameservers, authoritative and recursive. You can't help if another organization's authoritative nameservers are serving up signatures that have expired or are not yet valid, but you can make doubly certain that *your* hosts are accurate.

DS and DNSKEY Verification

If the timestamps look good, compare the key tag in the DS and RRSIG records. Zones can have multiple RRSIG and DS records, as we'll see in the next chapter, but so long as there's at least one valid path between the trust anchor and your RRSIG records it's fine.

Once you're experienced with troubleshooting DNSSEC, the easiest way to track down key mismatches is to use dig's `+trace` option. (The drill program also has a nice `-S` option specifically for tracking DNSSEC key errors.) The output is overwhelming to DNSSEC beginners, though, so we'll make more specific queries at first, using the test domain **`dnssec-failed.org`**.

I always start troubleshooting at the bottom of the Chain of Trust. If a domain like **`.com`** or **`.org`** loses its DNSSEC, you will know very quickly. If the type of domain regular organizations register loses its DNSSEC, the error is almost certainly at that domain.

```
$ dig dnssec-failed.org +cd +multi dnskey
…
;; ANSWER SECTION:
dnssec-failed.org.      3441 IN DNSKEY 256 3 5 (
…
     ) ; ZSK; alg = RSASHA1 ; key id = 44973
dnssec-failed.org.      3441 IN DNSKEY 257 3 5 (
…
     ) ; KSK; alg = RSASHA1 ; key id = 29521
```

This zone has two keys, a Key Signing Key and a Zone Signing Key. Remember, the zone's KSK signs the ZSK. The parent domain's DS record should point at a KSK. The KSK is key id (tag) 29521.

[14] In an ideal world.

```
$ dig dnssec-failed.org ds
…
;; ANSWER SECTION:
dnssec-failed.org.  21600  IN  DS  106 5 1 4F219DCE274F82063…
dnssec-failed.org.  21600  IN  DS  106 5 2 AE3424C903767E570…
```

The field immediately following the DS is the key tag. The DS records point at tag 106.

If this was your zone, you should immediately correct your DS records at the domain registry. If it's someone else's, well, you know it's not your problem. Unless it's your parent zone's, or your hosting platform's, or your CDN's, in which case it *is* your problem.

Compiling BIND Zones

With DNS errors, sometimes the only thing you can do is eyeball the zone's contents and try to figure out what's going on. If you're running BIND's DNSSEC automation, even zones backed by static files are effectively dynamic. Dynamic zones are most often backed by binary files, so you can't view them easily. Use named-compilezone(8) to make a text zone file from the binary files.

```
$ named-compilezone -j -f raw -o outputfile zone zonefile.signed
```

Use the `-j` option to include the journal file. The `-f raw` argument tells named-compilezone that it's reading a binary file. Specify an output file with `-o`. An output file of – spills the zone to the terminal. Then give the name of the zone and the signed zone file. The zone file will end in *`.signed`*. Here I compile my zone **`mwl.io`** from the data file *`mwl.io.db.signed`* to a file in */tmp*.

```
$ named-compilezone -j -f raw -o /tmp/mwl.io.zone \
  mwl.io mwl.io.db.signed
```

The file */tmp/mwl.io.zone* now contains the entire zone, including all RRSIG and NSEC records. Trawl through it and figure out what's wrong.

Other People's Problems

People break DNS. If you determine that a major business partner has blown up their DNSSEC and made their zone bogus, you can temporarily disable DNSSEC validation for just their zone. A *negative trust anchor* (NTA), defined in RFC 7646, tells a recursive nameserver that you know the zone is bogus and you are deliberately choosing to disable validation for the zone. NTAs are short-lived, and usually expire in one hour. That gives you enough time to contact the domain's DNS administrators and inform them that they've gone bogus. All major nameservers support NTAs.

With BIND, declare an NTA on BIND with `rndc`.

```
$ rndc nta zone
```

For example, if I blow up my zone and you want to email me to let me know, you could set an NTA on **`mwl.io`**.

```
$ rndc nta mwl.io
Negative trust anchor added: mwl.io/_default, expires
   16-Nov-2021 11:28:06.000
```

BIND checks the NTA'd zone every five minutes to see if the zone validates. If the zone becomes valid again, it automatically expires the NTA.

By default, NTAs expire in one hour. You can use the `nta-lifetime` *`named.conf`* option to set a different expiration time, in seconds. Adjust the polling interval with `nta-recheck`.

To set a different expiration time at the command line, add the `-lifetime` option. If I can see that the other company has already fixed their DNSSEC, but their TTL says that bogus data will be in cache for the next day, I could assign an appropriate lifetime and get on with my life.

```
$ rndc nta -lifetime 24h mwl.io
```

Now that you understand how to debug DNSSEC, let's see how to seriously mess it up by changing keys.

Chapter 7: Key Rollover

The longer a key is used, the more chances intruders have to break it. As computers grow more powerful, key algorithms need updating. And there's always the risk that someone will steal your private keys. All these problems have the same solution: replace your keys. This is called *key rollover*.

Rolling over your keys is the most complicated part of managing DNSSEC. Key rollover is possible only because an RRSet can have multiple signatures. If a nameserver can validate any one of those signatures, the data is secure. Extra signatures do not harm either DNS or DNSSEC.

The trick to managing complications is to make them routine. Minimize dread by doing the tricky part first. I strongly encourage you to perform a rollover *immediately* upon deploying DNSSEC on your test domain. Don't deploy DNSSEC on your production domains until you are so comfortable with the rollover process that it's full-on boring.

Rollover Complications

The last chapter showed how breaking the Chain of Trust makes zones bogus. Bogus zones disappear from the Internet. Thoughtlessly switching out a KSK, ZSK, or CSK breaks the Chain of Trust. Of all the sysadmin tasks I've ever performed, DNSSEC key rollover is the one that most requires stringent adherence to proper process. The main complexity arises from DNS caches. You must time everything so that resolvers don't try to use obsolete keys.

You might think of entirely avoiding this risk by creating the strongest keys possible, signing the zone, and never changing anything. That's possible. Some people have tried that. The longer your keys are in use, the greater the chance someone has brute-forced your private key and can spoof your DNS. The strongest keys from 2011 are laughably breakable in 2021.

Some similar applications never change their keys. An average SSH server generates a keypair when it's installed. That's common practice. But you should consistently update your SSH server to eliminate new security

flaws. Some of those updates add new algorithms and deprecate old ones. Additionally, an SSH server on the Internet gets replaced, upgraded, or penetrated and destroyed every few years. I manage zone files that have been in continuous use since the twentieth century. If I had signed that zone in the 1990s and never changed the keys, any cheap cellphone could break its encryption.

Okay, fine, but do people really break keys? In the real world? The answer is: yes. It hits the news a few times every year. My favorite story is from 2012, when a researcher cracked Google's email keys. The keys had been created several years before, deployed, and forgotten. Fortunately, the forger only used them to forge email from Google's founders telling the operations team to upgrade the keys.

The DNSSEC key rollover process is designed to avoid this class of threat.

Rollover Timing

The single most important goal of a key rollover is to ensure all clients everywhere in the world can validate your zone at all times. A recursive nameserver might still have your keys in cache, but need to make a fresh query to fetch a new record and its signatures. If the signatures don't match the keys, validation fails. Build your rollover plan around zone caching times. The times to consider include the zone's time-to-live, the secondary replication time, and the parent zone propagation time.

Time-To-Live

The key number is the zone's Time To Live, or TTL. This is how long recursive nameservers will cache a zone's data before querying again. You need an authoritative statement on the zone's TTL.

If you query a recursive server to find a zone's TTL, it will tell you how long any cached answer will remain in cache. If the zone has a one-hour TTL, but the recursive server looked up that zone 59 minutes ago, the recursive server will report a TTL of one minute. A rollover plan built on one-minute TTLs will fail. Always query the authoritative server for the TTL.

Here I check the TTL on my zone by querying an authoritative server. I add `+nocomment` to strip away extra debugging output, and `+norecurse`

to be sure I get an authoritative answer. (Disabling recursion should not be necessary, but as a matter of practice I always assume I've messed up somehow.[15]) The TTL appears in the SOA immediately after the zone name and before the class.

```
$ dig mwl.io soa +nocomment +norecurse @mwl.io

; <<>> DiG 9.17.18 <<>> mwl.io soa +nocomment @mwl.io
;; global options: +cmd
;mwl.io.  IN  SOA
mwl.io.  3600  IN  SOA  mail.mwl.io. mwl.mwl.io. 2021111002…
```

The TTL is 3600 seconds, or one hour. You could realistically expect nameservers around the world to query for new data within an hour.

Individual entries in a zone can also have their own TTLs. Most often, these are short-lived entries for dynamic systems. Such short-lived entries are not a concern for rollovers. Entries with extra-long TTLs are a concern, however. There's no way to identify overlong exceptions with dig; if you suspect they exist, you must examine the zone file.

Your zone's DS record is not in your zone, but in the parent zone. Query your parent zone's authoritative nameserver for that record. Here I check the TTL on my zone's DS record.

```
$ dig +nocomment +norecurse mwl.io ds @a0.nic.io +nocrypto
; <<>> DiG 9.17.18 <<>> +nocomment +norecurse mwl.io ds
  @a0.nic.io +nocrypto
;; global options: +cmd
;mwl.io.  IN  DS
mwl.io.  86400  IN  DS  30172 13 2 …
```

You can also have TTLs on individual entries in a zone.

The TTL on this DS record is 86400, or one day. Recursive nameservers will cache our DS record for up to twenty-four before checking for a new one.

Suppose I create a new KSK or CSK for my zone, sign my zone with the new key, immediately publish the new DS record, and remove the old DS record. Everything gets finished, wham-boom, the boss will be happy!

---

[15] Sysadmin Rule #12 states: "If you have to ask who screwed up, it was probably you."

Meanwhile, someone elsewhere on the Internet keeps refreshing my web site. At some point in the last twenty-four hours, their recursive nameserver cached my zone's DS record. Within the last hour, they cached my zone's A and RRSIG records.

The TTL on the DS record is twenty-four times that of the A and RRSIG records, so there's a roughly 4% chance that the DS record expires before the A and RRSIG records. If that happens, their recursive nameserver will fetch the new DS record. The DS record will match the new key. My zone will be secure, and the site will continue to work.

There's a 96% chance that the A and RRSIG records expire before the DS record, however. The recursive nameserver will fetch A and RRSIG records signed with the new key. The new key will not match the old, cached DS record. This recursive nameserver will declare the zone bogus.

A bungled key rollover will not make your zone universally bogus. Clients with an inconvenient mix of cached data will declare your zone bogus, while others will validate your zone just fine. Spotty trouble reports during a key rollover are an indication that you made changes too quickly.

Before scheduling a key rollover, examine the TTL of your zone and any special entries therein, as well as the TTL of your zone's DS record. Build your rollover schedule around the longest TTL. If the default TTL in your zone is one hour and your DS record has a TTL of six hours, but a couple entries in the zone have a TTL of one day, schedule everything based on a one-day TTL.

Replication Times

Modern primary nameservers use the NOTIFY protocol to inform secondary servers that a zone has been updated and they should refresh their copy. The NOTIFY protocol has no mechanism for telling either side if the message fails, though, so the secondary regularly checks for updates.

If you control both the primary and secondary nameservers and you know when you issue new keys, you could monitor the zone transfer and force an update if it fails. (BIND users can do that by giving the zone name to the `rndc retransfer` command.) Reliable NOTIFY saves much effort, though.

How do you verify that NOTIFY works? Update a zone. A few seconds later, compare the serial number on the primary and secondary servers. Identical serial numbers mean that the zone has updated.

If you don't control the secondary nameserver, and if you don't have reliable NOTIFY service, you must account for zone replication time in your DNSSEC rollover plan. Replication times are set in the SOA.

```
$ dig mwl.io +multi soa
…
;; ANSWER SECTION:
mwl.io.  3600 IN SOA mwl.io. root.mwl.io. (
     2021071219 ; serial
     3600       ; refresh (1 hour)
     900        ; retry (15 minutes)
     3600000    ; expire (5 weeks 6 days 16 hours)
     3600       ; minimum (1 hour)
)
```

The three times immediately after the serial number set zone replication timing.

The zone's *refresh* value is how often the secondary server queries the primary for a new copy of the zone. The refresh time here is 3600 seconds, or one hour.

The *retry* value is only used if the secondary server could not reach the primary at the refresh time. It's how often the secondary checks back after a refresh check failed. The retry time on this zone is 900 seconds, or fifteen minutes. If the secondary was refreshing exactly when the primary nameserver was offline, the secondary would try again every fifteen minutes.

The *expire* time shows when the secondary server gives up serving the zone. The expire time here is 3600000 seconds, or just shy of six weeks. If the secondary cannot reach the primary in that time, it declares the zone stale and stops answering queries for it. Hopefully at some point in that month and a half, you'll notice the primary is offline and do something about it.

If you're relying on time-based replication between primary and secondary servers, be pessimistic. Each time you update the zone, assume that the secondary checked for an update a split second before the changes went live on the primary. The secondary will continue to serve old data for the entire refresh time. Recursive nameservers will be caching that old data up until the zone refreshes.

This means your TTL for planning purposes is the zone TTL plus the refresh time. In this example, both of these are one hour. Recursive nameservers will not have fully updated their records on your zone for two hours. In this case, two hours makes no difference; the six-hour TTL on the DS record trumps everything. If you have long refresh times, however, you must account for them.

If your secondary replication is unreliable, you must either make it reliable or not start any countdowns until you've verified the zone replicated. And fix your secondaries.[16]

Parent Zone Updates

If this key rollover involves changing the DS record with your registry, you must consider two times outside your control: the time your registry needs to update your parent zone, and the time the parent zone needs to replicate changes.

If you tell your registrar you have a new DS record for your `.com` domain, how long does it take for that record to appear in `.com`?

Theoretically, parent zone updates happen quickly. When you register a new domain, you want it available immediately. The registrar wants to update zones within minutes. It doesn't always happen, however. Similarly, one would hope that nameservers for zones like `.com` would replicate quickly. You can't assume that they will.

Check the TTL and refresh times on your parent zone. Consider how delays in updates and replication would affect your plans. For my zones, the TTL on the DS record far exceeds the parent zone's refresh time. If your zone is business-critical and intermittent DNS errors would cause meetings, include "verify DS records" and "verify parent zone replication" in your process.

16 If your zone's expire time comes into your key rollover plan, you're doing DNS wrong.

Effective TTL and Recovery

Considering all the timing factors lets you figure out your "effective TTL," or the time you must wait to be sure that all caches everywhere have your new data. Be very generous with your estimate. If you figure you have an eight-hour TTL, call it a full day.

If you mess up your DNS, with or without DNSSEC, this TTL is also the amount of time it will take to recover. You might fix the zone immediately, but any recursive that caught the bad data will cache it until that TTL expires. The TTL is directly proportional to your Mean Time To Recover, and if that's much more than a day you should reconsider your zone times.

Under normal circumstances, I use a one-day effective TTL. If an intruder has captured your private key and you must roll your KSK and ZSK as soon as possible, stick with the effective TTL you've added up from your zone's TTLs. Depending on your threat model and what an intruder might do by spoofing your DNS, you might even immediately delete such a key and publish a new DS with the registrar, taking the availability hit to preserve integrity.

When discussing rollovers, take "TTL" to mean "the time to fully propagate your data, as you've determined based on your zones and your environment."

Rollover Methods

Rollover procedures are built around preferred optimizations. Minimizing the number of interactions with the parent zone lets you speed the operation, at a cost of increasing the size of the zone and the amount of DNS traffic. Minimizing the zone size increases the length of time the process takes and the number of interactions with the parent zone. The standard rollover methods are not *quite* that simple, but that's a major trade-off underlying everything.

To roll over keys, you can either double-sign everything or publish the new keys before using them. The details differ slightly depending on if you're rolling over a ZSK or a KSK. Other standards exist, but they're either for very specific circumstances or have no advantage over these two.

ZSK Rollovers

Zone Signing Key rollovers are completely internal, and do not require interacting with the parent zone. The two standard methods for rolling over ZSKs without interrupting service are called *double signature* and *pre-publishing*.

In the *double signature* method, the zone is simultaneously signed with both the old and the new keys, creating multiple RRSIG records. Recursive clients get all the RRSIG records. The authoritative server maintains those multiple signatures until the TTL has passed, so all clients everywhere will have discarded records signed with only one key and have the new double-signed RRSIG records in cache. At that point, you can remove the old key and the old signatures.

Double signing is unquestionably safe. There is no risk of breaking the Chain of Trust, as you sign everything locally. It almost doubles the size of everything signed by this key, however. Most organizations don't care about that. If you have a huge zone, or if you serve a lot of queries, DNS bandwidth might be a concern.

With the *pre-publishing* method, you make the zone's new DNSKEY available before using them to sign any RRSets. Once the TTL has passed and the record has propagated to all clients, you remove the old signatures and create new ones with the new key.

The pre-publish method is more elegant than double signing. It doesn't double the size of your zone. It replaces signatures as they expire, rather than signing everything twice and leaving heaps of RRSIG records everywhere. It is more complicated, however, and complication leads to outages.

A rollover method called *double-RRSIG* also exists, but it combines the disadvantages of double signatures with the disadvantage of pre-publishing and provides no benefit over either, so we'll ignore it.[17]

Best practice calls for using pre-publishing for ZSK rollovers. ZSKs do not have DS records, so the rollover only changes the DNSKEY and RRSIG records.

---

[17] The fact that the authors of RFC 7583 felt it necessary to include this method says nothing about the method proper, but speaks volumes about network administrators.

KSK Rollovers

The Key Signing Key serves two duties. It's the target for the DS record, and it signs the ZSK. Changing the KSK means changing the DS record, so you must interact with the RIR or domain registrar, which makes them more complicated than ZSK rollovers. The three standard methods for KSK rollovers are double-KSK, double-DS, and double-RRSet.

In the *double-KSK* method, the new public key is added to the DNSKEY RRSet. The RRSet is signed with both the old and new key. You wait a TTL for the old RRSet to expire from everyone's cache, then go to the registrar to add the new DS record and remove the old. Wait another TTL for the new DS record to percolate into the caches, then remove the old key from the RRSet. The advantage to double-KSK is that you need interact with the registrar only once. The disadvantage is that your DNSKEY record is slightly larger during the rollover.

In the *double-DS* method, start by going to the registrar and publishing the new DS record right next to the previous one. Wait a TTL for this change to propagate through all caches. In a single operation, activate the new key and remove the old key.

The *double-RRSet* method does everything at once, but has all the disadvantages of both the other methods. You keep the old key but activate the new key, signing everything with both keys, and immediately publish the DS record. After a TTL passes, you remove the old DS record and the old key.

Which should you use? I prefer the double-KSK method. This allows you to publish the new key and verify that remote nameservers can validate your zone with that key before unpublishing your old DS record and removing the old key. If you're in an emergency, such as recovering from an intruder, the double-RRSet method is the best. All three of these methods work, however.

CSK Rollovers

A Combined Signing Key fulfills the duties of both the KSK and the CSK. As such, the rollover combines methods used by both. To minimize the size of your zone and parent zone interactions, a CSK uses elements of the ZSK's pre-publishing method and the KSK's double-KSK method. Everything starts with publishing the new DNSKEY.

Changing the CSK requires interacting with the parent zone, so it has the same timing considerations of a KSK rollover.

How Often Should I Rotate?

Knowledgeable people vehemently disagree on how often keys should be rotated.

People with low risk tolerance declare that ZSKs should be rotated every three months and KSKs every year. The duration increases with your risk tolerance, and as the value of your zone decreases. It's squishy. The previous root zone key was used for more than ten years, but the private key was kept offline and protected in a manner most of us wouldn't bother with even if we could afford it.

Some folks say you should never rotate your keys. How often do you rotate your SSH server keys, after all?

This distills to: what is your threat profile?

Is anyone going to attack your personal domain? Perhaps. It depends on the type of trouble you cause and the causes you champion. Will those attackers have access to enough computing power to break your keys in a reasonable time, and the knowledge to so? Perhaps. But the presence of DNSSEC on your zone might dissuade your foes from DNS spoofing, making them break into your web server instead.

I know of more than one big-money financial institution that hasn't changed their RSASHA1 keys for ten years. That's a poor choice.

When your signature algorithms wear out, you *must* rotate your keys. If your signatures use RSASHA1, rotate to a modern algorithm as soon as possible. Eventually even RSASHA256 will fail, but *eventually* might take a long time.

Small sites using a CSK should set their rotation needs based on their threat profile. I'll rotate my personal domain, with my blog and the interminable list of ~~junk~~ books I've written, every few years. I would say than an ecommerce company that stores customer's personal data should rotate keys at least that often. Looking at the practices of `.co.uk`, though: not everybody agrees. Choose your risks, and live with them.

However often you rotate your zone's keys, you must become comfortable with the rotation process. Sysadmins hate doing unfamiliar tasks, especially when they might spark user complaints. Your company should have a test zone that mirrors the production environment's settings, and rotate its keys frequently enough to make the procedure routine. Otherwise, when disaster strikes and you must rotate a production zone's keys *immediately now now now*, you will suffer unnecessarily.

Key Lifecycle

Rollovers mean that keys are created, used, deprecated, and destroyed. RFC 7583 provides formal language to describe DNSSEC key states, but many of these are not widely used. A key has four important dates: publish, activate, inactivate, and delete.

The *publish* date is the date the key can be made available to the world. The day a DNSKEY record appears in the zone is the publish date. When a nameserver sees a new key it publishes it, unless the key has a publish date in the future.

The *activate* date is the date that the key can first be used to sign RRSets. It doesn't mean that the key will be immediately used, mind you. If you're doing pre-publication rollovers and no signatures are due for renewal, the nameserver won't use the key until it needs it. If your key has a limited lifespan, this is the first day of its life.

The *inactivate* date is when the key will no longer be used to create new signatures. Signatures using that key might still be drifting around in client caches, but that's okay. The key's DNSKEY record remains in the zone, and those cached signatures will still validate.

On the *delete* date, the key's DNSKEY is removed from the zone. Signatures using that key will no longer validate.

If you're running BIND, the key file contains any dates that have been set.

```
; This is a key-signing key, keyid 44950, for mwl.io.
; Created: 20210824194856 (Tue Aug 24 15:48:56 2021)
; Publish: 20210824194856 (Tue Aug 24 15:48:56 2021)
; Activate: 20210824194856 (Tue Aug 24 15:48:56 2021)
; Inactive: 20210922184707 (Wed Sep 22 14:47:07 2021)
; Delete: 20211002195207 (Sat Oct  2 15:52:07 2021)
; SyncPublish: 20210825205356 (Wed Aug 25 16:53:56 2021)
```

This key was created, and immediately published and activated, on 24 August 2021. It was deactivated on 22 September 2021, and scheduled to be deleted on 2 October 2021. You might also see a "Revoked" date, if the key was compromised and you must rush a rotation. The SyncPublish date is for CDS and CDNSKEY records, which we'll discuss in "Automation" later this chapter.

Algorithm Rollovers

Cryptographic algorithms don't grow weaker with time, but our ability to attack them increases every day. Eventually, all algorithms become vulnerable and must be replaced with something stronger.

When I first deployed DNSSEC, keys used SHA1. These keys are badly insecure today. If you haven't updated your keys since the first edition of this book, or if someone discovers weaknesses in the modern key algorithm you're using, you must perform an *algorithm rollover*. Algorithm rollovers not only switch keys, they switch the algorithm used by those keys—say, the old key uses algorithm 8 (RSASHA256), but your new keys need algorithm 13 (ECDSAP256SHA256).

Think of an algorithm rollover almost as a new DNSSEC deployment. Enable the new algorithm on your zone and ensure that everything in the zone has signatures using the new algorithm before publishing the DS record. Once the new DS has propagated everywhere, you can remove the old algorithm.

Algorithm rollovers were tricky during DNSSEC's early days. You'll see many articles and posts about problems with software compatibility and key generation and all sorts of headaches. The good news is, most nameservers have coped with those problems. Mostly.

Validators only accept the algorithms given in the DS record for all keys in the child zone. Suppose you successfully deploy a new KSK with algorithm 13. The ZSK must also use algorithm 13. The trick with an algorithm rollover is to roll the KSK *and* the ZSK simultaneously. The nameserver should add the new signatures first, then the key records, and only then should you publish the DS records.

Root Zone Key Rollover

Theoretically, the key currently used to sign the root zone should be valid for many years. Planning to roll the root zone key and replace the trust anchor takes several years. Debacles happen, though, so we'll briefly discuss it.

Like any other KSK or ZSK, the root zone key uses an algorithm and a private key. If that algorithm starts to show its age or the private key is compromised[18], the key must be rolled over and the new trust anchor distributed to validating recursive servers. The process used to roll the previous root zone key is documented in RFC 5011. Most nameservers support trust anchor rollovers as documented in RFC 5011.

New trust anchors are published as DNS records, signed by the old trust anchor. The new trust anchor public keys are published years ahead of time, so that software vendors can include them in updates. If you keep your DNS servers updated, key rollovers are a non-issue.

The last trust anchor rollover was 11 October 2018. Most sites were fine. A few sites got a short sharp lesson on the importance of keeping their DNS servers updated.

Will the next rollover use RFC 5011? Maybe. When will the next scheduled rollover happen? Not yet. But it will either be announced several years in advance, or some master thief compromised the root zone key.

[18] The DNSSEC trust anchor's private key is protected by methods up to and exceeding air gaps, armed guards, and rabid wombats. Don't worry about it.

Rollover Automation

The weak part of key rollovers is informing the registrar of the new DS record. This seems like an operation highly suited to automation, doesn't it?

It is. Sadly, we didn't settle on a standard automation method before first deploying DNSSEC. Many registrars implemented their own unique APIs for communicating with customers. Many nameservers implemented software to communicate with a few particular registrars. The result is a mishmash of standards and methods.

In 2017, standards bodies agreed on a method for authoritative nameservers to communicate with registrars through new record types: the CDS and CDNSKEY records.

The CDS record shows what the zone would like its DS record to look like. The CDNSKEY record gives what the zone would like its DNSKEY records to look like. Both are signed by the current KSK or CSK.

Parent zones should poll their child zones each week to check for updated CDS and CDNSKEY records. If the records are found, validate, and are generally coherent and sensible, the parent zone should update their DS records.

CDS and CDNSKEY records require support at registries, not registrars. Only a handful of registries support CDS and CDNSKEY records yet, however. For now, you're stuck looking for other methods to automate communicate with the registrar.

KSK updates are not frequent. Any automation that runs infrequently is inherently fragile. If you support so many zones that you will use the automation regularly, by all means automate DS records using the registrar's recommended method. If not, update your records manually—and ask your registrar to support CDS records.

Now that you have the theory, let's look at rollovers with BIND policies.

Chapter 8: BIND Policies and Rolls

BIND performs key rollovers according to the *Key and Signing Policy*, or *KASP*. The policy also sets the kinds of keys a zone uses and all the timing. You must understand policies to properly roll keys with BIND, in addition to everything discussed in Chapter 7. The default policy is deliberately conservative, designed to work on any equipment you might have, down to and including a Raspberry Pi. If you want to do anything more complicated, you'll need to write your own policy.

BIND includes other key management tools like dnssec-keymgr(8), dnssec-keygen(8), and dnssec-setdate(8), but as of BIND 9.16 these tools are wholly obsoleted by KASP.

First, let's look at how to represent time in policies.

Duration Time Format

Zone files traditionally express durations in seconds, with occasional use of date(1)-style markers for hours (3h) or minutes (15m). This has worked well for decades, but folks are beginning to migrate to the ISO 8601 duration format. While the older method will remain with us for the rest of my career, you should also understand the ISO format.

All duration definitions start with a capital P. Y represents the number of years, M the number of months, W the number of weeks, and D the number of days. P3Y means "a duration of three years," while P1Y2M3W4D means "a duration of one year, two months, three weeks, and four days."

"Months" is an ambiguous term, and I encourage you to avoid it. If you want a key to expire every three months, use a duration like P90D (ninety days) instead.

A T represents a split between units of a day or longer, and units smaller than a day, rather like a decimal point. Within a day, the units are H for hours, M for minutes, and S for seconds.[19] Exactly like a decimal point, the T is mandatory when expressing times smaller than a day. The duration P5M is five months, while PT5M is five minutes. P1YT5M means "a duration of one year and five minutes."

Which should you use in your policy? Most of the policies I see mix ISO 8601 and date(1) markers. If so inclined, you may now begin crying.

Policy Components

A policy defines your keys and how you sign your zone. It also lets you set information about external entities, like the TTL of the zone's parent, so that BIND can intelligently rotate keys. Policies must be in `named.conf`, perhaps via an include file. A policy looks like any other BIND `named.conf` component.

BIND includes a default policy that includes sensible settings for average zones. You cannot redefine the default policy; instead, create a new policy for your zones. Start your policy by copying the default policy. Here's the BIND 9.16 policy.

```
dnssec-policy "default" {
     keys {
       csk key-directory lifetime unlimited algorithm 13;
     };

     // Key timings
     dnskey-ttl 3600;
     publish-safety 1h;
     retire-safety 1h;
     purge-keys P90D;

     // Signature timings
     signatures-refresh 5d;
     signatures-validity 14d;
     signatures-validity-dnskey 14d;

     // Zone parameters
     max-zone-ttl 86400;
     zone-propagation-delay 300;
```

[19] Yes, ISO used the same marker for months and minutes. If you know the responsible party, you have my blessing to yell at them.

```
        // Parent parameters
        parent-ds-ttl 86400;
        parent-propagation-delay 1h;
};
```

Let's dissect this policy.

```
dnssec-policy "default" {
        keys {
          csk key-directory lifetime unlimited algorithm 13;
        };
        dnskey-ttl 3600;
        publish-safety 1h;
…
};
```

A policy includes key definitions and timing details. The `keys` stanza lets you define if you're using KSKs and ZSKs or standalone CSKs, and which algorithms they will use.

After the key definitions we see timing options. Each stands alone. The defaults are chosen so that new DNSSEC administrators don't have to immediately schedule rolls.

We'll start by discussing the timing options, then proceed to keys.

Timing Options

The policy's timing options let the policy express the TTL and replication factors discussed in Chapter 7. Your secondary nameservers need four hours to replicate a zone from the primary? One of your zones has a two-month TTL? There are options for that. Here are the timing options, as set in the BIND 9.16 default policy.

```
        dnskey-ttl 3600;
        publish-safety 1h;
        retire-safety 1h;
        purge-keys P90D;
```

The `dnskey-ttl` option lets you set the TTL on all DNSKEY records. The default is one hour.

When BIND rotates keys, it calculates the TTL based on the times in zones, exactly like you did in Chapter 7. The `-safety` options let you add extra time as padding, to be absolutely certain remote nameservers have flushed old data from their cache. The `publish-safety` setting is added to the time between publishing a key in a zone and using it to create RRSIGs.

The `retire-safety` option is added to the time a key remains in the zone even though it is no longer active. The default for each is one hour.

After a few years of automated rotation, key files can build up. The `purge-keys` option lets you automatically delete keys that have been published, used, unpublished, and removed from service. The default, P90D, removes key files after 90 days of disuse.

Signature life information appears next.

```
signatures-refresh 5d;
signatures-validity 14d;
signatures-validity-dnskey 14d;
```

When a record is signed, the signature is assigned an expiration date. The various `signatures-` options set the expiration date and dictate when the signatures get renewed. These expiration dates have nothing to do with the zone's TTL. Generating signatures is computationally expensive, and you don't want to do it constantly.

When BIND signs a record set to create an RRSIG, it checks `signatures-validity` to set the expiration date. The default is fourteen days.

Similarly, the `signatures-validity-dnskey` lets you set a separate signature lifetime for signatures on DNSKEY records. You shouldn't need to change the default from 14 days, as the zone's TTL dictates how often clients discard and refresh their data.

The `signatures-refresh` setting tells named how many days before the signature's expiration it should re-sign everything. The default is five days. If your signatures have a maximum lifetime of 14 days, signatures are regenerated when they're 9 days old. Suppose you want to regenerate new signatures every six days? Set `signatures-refresh` to 8 to renew them eight days before the 14 day expiration. Do not reduce the signature refresh to fewer than five days. If your server develops a signing problem on Friday afternoon, would you rather fix it by Saturday morning and trash your weekend, or have until Monday afternoon?

We then have timing details about the zone and replication.

```
max-zone-ttl 86400;
zone-propagation-delay 300;
```

The `max-zone-ttl` setting controls the maximum allowable TTL of data in a zone signed with this policy. The default, 86400 seconds or one day, should be enough for most zones. Key rotation is highly dependent on TTL, so the DNSSEC policy must know the maximum possible value. If your zones include TTLs over one day, you must increase `max-zone-ttl` to match.

If replication between your primary and secondary nameservers is slow, you can use `zone-propagation-delay` to set an appropriate time. If NOTIFY works between your primary and secondary servers, the default of 300 seconds should be perfectly fine.

Lastly, we set timing values for the parent domain.

```
parent-ds-ttl 86400;
parent-propagation-delay 1h;
```

The `parent-ds-ttl` lets you set the parent zone's TTL, so BIND knows how long your DS records will remain in other nameservers' caches. Most zones have a TTL of a few hours, but leaving this at one day is a safe choice.

Most parent zones like **`.com`** and **`.org`** have solid nameserver replication, but `parent-propagation-delay` lets you add a safety margin. The default is one hour.

The keys Stanza

A policy's `keys` stanza declares what sorts of keys the zone will use.

```
keys {
   csk key-directory lifetime unlimited algorithm 13;
};
```

We can list multiple keys here, but the default policy has a single Combined Signing Key. The key is stored in the key directory. Each key has a lifetime, or how long it is good for. In the default policy, keys are good forever (lifetime unlimited). Finally, this key uses algorithm 13: ECDSAP256SHA256.

The only odd thing about this policy is that the key never expires. The default policy is intended for use by DNSSEC beginners, who don't yet understand key rollover and interacting with registries and RIRs. The goal of the default policy is functional DNSSEC that doesn't break anything now or

in the future. It's like training wheels. You'll write your own policy that fits your environment and key rotation needs.

Default Policies vs. Your Policies

Define your policies exactly like the default policy.

```
dnssec-policy "policyname" {
     options here;
};
```

Any options that you don't set in your policy get sucked in from the default policy. BIND's default policy is compiled into named, but you can get a copy from the source code. Here, I've written a policy that entirely matches the default except for one nifty setting that we want to use.

```
dnssec-policy "mycompany" {
     cool-option yes;
};
```

Such a policy might work well initially, but a future major release might change the default policy. (Intrusive updates happen only at major releases, such as 9.18 to 9.20.) If a future BIND release changes the default policy algorithm from 13 to 19, though, when you upgrade the policy engine will immediately generate new keys and begin a rollover. While I'm certain the policy would be written to not make your zones insta-bogus, it would ambush you with extra work. Nobody enjoys ambush labor.

Avoid this risk by always defining your policies completely. To avoid cluttering your `named.conf` with innumerable lengthy policies, consider using include files.

```
include "/etc/namedb/policies/nsec3.conf";
```

When you upgrade your BIND install, check the new default policy for hints on what you should plan to upgrade. If the default algorithm changes start planning a controlled, deliberate algorithm rollover.

I keep a copy of the default policy available, copied from the BIND source tree. Any time I want to create a new policy I copy the default to a new file, give it a new name, and edit it to fit my needs.

NSEC3

DNSSEC provides two ways to securely demonstrate that a record does not exist, NSEC and NSEC3. Chapter 2 discusses both. NSEC is simpler, but it does allow clients to enumerate and examine all the entries in your zone. Offering NSEC3 records requires the authoritative nameservers perform more calculations, but the load is negligible for most zones. The conservative default policy uses NSEC. You need a new policy to deploy NSEC3.

Enable NSEC3 by adding the `nsec3param` option.

```
...
nsec3param iterations 0 salt-length 0;
...
```

NSEC3 has optional features of passing records through the hashing algorithm multiple times, and adding a salt to further obfuscate entries. Recent research has shown that these do not improve security, but do degrade responsiveness and increase system load. Disable them both. If you don't specify these options, BIND 9.16 defaults to using 5 iterations and a salt length of 8. BIND 9.18 will default to using zero iterations and no salt.

You could also change how NSEC3 represents child zones, by enabling NSEC3 opt-out. It's useful only if you have thousands of delegations, though. Look into opt-out if you're one of those unlucky people.

KSK Policies

While the default CSK policy is useful for low-value zones, if you're running an ecommerce site or other high-value site you might want a KSK and a ZSK instead. Configure these in the policy's *keys* stanza, like so.

```
keys {
     key-role lifetime period algorithm algorithm;
};
```

The key-role determines if this key is a CSK (csk), KSK (ksk), or ZSK (zsk).

You'll often see a second argument of `key-directory`. That's an optional statement of where to store the key. A future BIND release might be able to store keys directly on an HSM. For now, all keys get put in the configured key directory.

The `lifetime` sets how long the key will be used for. We saw `lifetime` set to `unlimited` for the default policy, creating a key with no expiration date. Rolling a KSK requires interacting with the zone's registrar or RIR, so I recommend setting such keys to never expire and using your calendar to schedule manual rolls. Specify a key lifetime for a ZSK, and BIND will automatically perform rollovers for you.

Last, the `algorithm` keyword sets the key algorithm. You can use the number, or the algorithm's formal name. Algorithms like RSASHA256 can take key length as well, so add that as an additional argument.

Here's how I might set up my KSK and ZSKs in a policy.

```
dnssec-policy "ksk" {
     keys {
        ksk lifetime unlimited algorithm 13;
        zsk lifetime 90d algorithm 13;
     };
…
```

This policy creates a KSK that never expires, using algorithm 13 or ECDSAP256SHA256. The ZSK also uses ECDSAP256SHA256, but it expires every ninety days.

I copy the remainder of this policy from the default policy, to protect me from unexpected changes during upgrades. Now I can apply this policy to my test zone with a `dnssec-policy ksk` statement in *`named.conf`* and reconfigure my nameserver. What does my nameserver think of this zone's DNSSEC?

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: ksk
current time:  Wed Sep  8 16:16:23 2021

key: 4060 (ECDSAP256SHA256), ZSK
     published:      yes - since Wed Sep  8 16:14:27 2021
     zone signing:   yes - since Wed Sep  8 16:14:27 2021

     Next rollover scheduled on Wed Dec  8 14:09:27 2021
     - goal:           omnipresent
     - dnskey:         rumoured
     - zone rrsig:     rumoured

key: 50062 (ECDSAP256SHA256), KSK
     published:      yes - since Wed Sep  8 16:14:27 2021
     key signing:    yes - since Wed Sep  8 16:14:27 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         rumoured
     - ds:             hidden
     - key rrsig:      rumoured
```

Key 4060 is the new ZSK, while 50062 is the brand new KSK. The repeated *rumoured* states show that I applied this policy so recently that remote nameservers haven't yet had a chance to update their caches. In a few hours, once the TTL has expired, BIND will updated the local statuses to *omnipresent*. Once the DNSKEY and signatures become *omnipresent*, it is safe to publish the DS records. The DS record will become *rumoured*, and the zone will publish new CDS and CDNSKEY records.

Once I have published the DS record in the parent zone and verified that it's actually present in the wild, I can update the key's status with rndc(8).

```
$ rndc dnssec -checkds published mwlucas.org
```

BIND will wait for the TTL to expire, then update the key's status.

Migrating Older Configurations to Policies

If you set up your DNSSEC years ago and haven't touched it since, update your configuration to use policies. All new management tools will expect policies, and older configuration methods will be removed eventually. Before touching anything, make sure you have good backups of the zone and key files. Don't start this migration during a key rollover either.

We'll migrate this inline signing configuration carried forward from BIND 9.9 over to a KASP policy. This is a configuration used in the first edition of this book.

```
zone "mwlucas.org" {
     type primary;
     file "/etc/namedb/primary/mwlucas.org";
     inline-signing yes;
     auto-dnssec maintain;
     key-directory   "/etc/namedb/working/keys/mwlucas.org";
};
```

These older DNSSEC methods required manually generating keys, using whichever algorithm and key length you wanted. The first edition of this book used RSASHA256 keys, or algorithm 8, with a 2048-bit KSK and a 1024-bit ZSK. Unfortunately, `rndc dnssec -status` does not work on zones using the older configurations, so you can't use it to verify your current keys match those original values. Each key created by a policy has a *.state* file amidst the other key files, however, giving the algorithm and length. Double-check using a site like **`https://dnsviz.net`**.

Now define a policy that uses these keys. Everything can use the same settings as the default, but you'll need a custom `keys` stanza. I'm calling this policy *dm1*, after the first edition of this book.

```
dnssec-policy "dm1" {
     keys {
       ksk lifetime unlimited algorithm 8 2048 ;
       zsk lifetime 30d algorithm 8 1024 ;
     };
…
```

Apply the policy to your zone and comment out the `inline-signing` and `auto-dnssec` options.

```
zone "mwlucas.org" {
     type primary;
     file "/etc/namedb/primary/mwlucas.org";
// inline-signing yes;
// auto-dnssec maintain;
     dnssec-policy dm1;
     key-directory   "/etc/namedb/working/keys/mwlucas.org";
};
```

You touched the configuration, so bump the zone's serial number. Reload named, or at least the zone.

```
$ rndc reload mwlucas.org
```

BIND will find and use the existing keys. You are now using a policy. You must entirely reload named for `rndc dnssec` to work on the zone.

Performing Rolls

DNSSEC key rollovers can be managed either automatically within BIND or at the command line via rndc(8). The way key rollovers get handled depends on any changes required in the outside world.

ZSKs are managed entirely within a zone, and do not require interacting with the outside world. BIND rolls over ZSKs without sysadmin intervention, as dictated by policy's setting for the key's lifetime. ZSKs use the pre-publishing rollover method, where the new key's DNSKEY record is published and omnipresent before being used to sign any records.

Changing KSKs and CSKs require interacting with the domain registrar or RIR. Even if you configure a maximum lifetime on your KSKs and CSKs, BIND won't stop using the old key until you inform it that the new key's DS record has been published. You might have add-on software for your registrar that lets you roll over these keys automatically, but most of us must supervise rolls.

As a general rule, only roll over one key in a zone at a time. If you want to roll your KSK but your ZSK is mid-roll, wait until the ZSK roll completes and the old key is no longer in the zone before starting the KSK roll. You can do multiple rolls simultaneously, but you're likely to confuse yourself.

During your first rolls, I highly recommend enabling the DNSSEC debugging log as discussed in Chapter 6. You'll see exactly what actions BIND takes and various countdown timers.[20] Once you're comfortable with rolls, keep the debugging log on hand for when things go weird.

While the last chapter discussed different ways of performing rollovers, policies always use the safest method for that kind of rollover. You cannot change rollover methods via policy.

[20] The debug log reveals that the "unlimited" key lifetime is a lie. It's a mere 84 years or so. I'm sure BIND will get patched before 2100, mind you.

We'll start by performing a simpler ZSK roll, then using what we've learned to perform more complicated KSK and CSK rolls.

Manual ZSK Rolls

Start by checking the status of your zone's DNSSEC keys.

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: ksk
current time:  Mon Sep 20 15:22:07 2021

key: 4060 (ECDSAP256SHA256), ZSK
     published:      yes - since Wed Sep  8 16:14:27 2021
     zone signing:   yes - since Wed Sep  8 16:14:27 2021

     Next rollover scheduled on Fri Oct  8 14:09:27 2021
     - goal:           omnipresent
     - dnskey:         omnipresent
     - zone rrsig:     omnipresent

key: 50062 (ECDSAP256SHA256), KSK
     published:      yes - since Wed Sep  8 16:14:27 2021
     key signing:    yes - since Wed Sep  8 16:14:27 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             omnipresent
     - key rrsig:      omnipresent
```

This zone has two keys. Key 4060 is a ZSK, and key 50062 is the KSK. We must make absolutely certain to roll over only key 4060, and leave key 50062 unchanged. Use `rndc dnssec -rollover` to trigger a roll. The `-key` argument lets us specify a key.

```
# rndc dnssec -rollover -key 4060 mwlucas.org
Key 4060: Rollover scheduled on 20-Sep-2021 15:30:42.000
```

That seems too easy. Check the zone's status.

```
# rndc dnssec -status mwlucas.org
dnssec-policy: ksk
current time:  Mon Sep 20 15:33:36 2021

key: 4060 (ECDSAP256SHA256), ZSK
     published:      yes - since Wed Sep  8 16:14:27 2021
     zone signing:   yes - since Wed Sep  8 16:14:27 2021
```

```
    Key will retire on Mon Sep 20 17:35:42 2021
    - goal:           hidden
    - dnskey:         omnipresent
    - zone rrsig:     omnipresent

key: 50062 (ECDSAP256SHA256), KSK
…
```

The description for key 4060, the ZSK, has changed. Right after the publication data, there's a statement that the key will be retired on a certain date and time. The goal for this key is *hidden*, indicating that we want to remove the key from the zone. The existing records are *omnipresent*, meaning that any caching servers still have these records.

The entry for key 50062, the KSK, is unchanged.

Down at the bottom, though, we have a brand new key entry.

```
…
key: 31047 (ECDSAP256SHA256), ZSK
    published:      yes - since Mon Sep 20 15:30:42 2021
    zone signing:   no  - scheduled Mon Sep 20 17:35:42 2021

    Next rollover scheduled on Sun Oct  3 15:30:42 2021
    - goal:           omnipresent
    - dnskey:         rumoured
    - zone rrsig:     hidden
```

Key 31047 is a newly created ZSK. It was published at the exact same time declared by `rndc dnssec -rollover`, but it is not yet used for zone signing. This fits with the pre-publish rollover method. According to the times set in the policy and the zone's TTL, this key can be used for signing after two hours and five minutes.

Verify the availability of the new key with dig.

```
$ dig mwlucas.org dnskey +nocrypto +nocomments

; <<>> DiG 9.17.18 <<>> mwlucas.org dnskey +nocrypto +nocom-
ments
;; global options: +cmd
;mwlucas.org.                   IN      DNSKEY
mwlucas.org.  3600  IN  DNSKEY  256 3 13 [key id = 50062]
mwlucas.org.  3600  IN  DNSKEY  257 3 13 [key id = 31047]
mwlucas.org.  3600  IN  DNSKEY  256 3 13 [key id = 4060]
…
```

The middle entry here is key 31047. It's really in our DNS. Checking the zone's RRSIG records show that it's not yet in use, though. If you check a tool like DNSViz, you'll see that key 31047 is signed but nothing hangs off of it.

```
$ dig mwlucas.org RRSIG |grep 31047
$
```

The real test is when we return more than two hours and five minutes later to see what the zone looks like.

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: ksk
current time:  Tue Sep 21 13:04:51 2021

key: 4060 (ECDSAP256SHA256), ZSK
     published:     yes - since Wed Sep  8 16:14:27 2021
     zone signing:  no

     Key is retired, will be removed on Thu Sep 23 18:40:42
   2021
     - goal:        hidden
     - dnskey:      omnipresent
     - zone rrsig:  unretentive
```

Key 4060 is the old ZSK. It is published in the zone, but will not be used to generate new signatures. We see the date and time the key will be removed from the zone. The zone *rrsig value* of unretentive means that existing signatures can remain, but when signatures signed with this key expire they will not be replaced.

Our KSK record is unchanged, but then we have the new ZSK.

```
key: 31047 (ECDSAP256SHA256), ZSK
     published:      yes - since Mon Sep 20 15:30:42 2021
     zone signing:   yes - since Mon Sep 20 17:35:42 2021

     Next rollover scheduled on Sun Oct  3 15:30:42 2021
     - goal:        omnipresent
     - dnskey:      omnipresent
     - zone rrsig:  rumoured
```

This key is ready to sign records. As RRSIG records signed with the old key expire, they will be replaced by signatures generated from this new key.

KSK Rollover

Site that are valuable targets for intruders, such as ecommerce companies and anything containing personally identifiable information, should use a separate KSK and ZSK and rotate the KSK every year or so. The command-line process is very similar to the ZSK rollover, but requires interacting with the registrar or RIR at critical moments. BIND uses the double-signature rollover method. You publish the new key's DNSKEY record, wait for it to propagate, then switch the DS record at the parent.

Before starting a KSK rollover, check the parent zone's TTL to see how quickly it will redistribute an updated DS record. Verify that the DNSSEC policy matches or exceeds the TTL, so that BIND can safely change key status.

Here are the zone's current keys.

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: dm1
current time:  Tue Sep 21 15:33:37 2021

key: 15296 (RSASHA256), KSK
     published:      yes - since Fri Sep 17 13:50:18 2021
     key signing:    yes - since Fri Sep 17 13:50:18 2021

     No rollover scheduled
     …

key: 55383 (RSASHA256), ZSK
     published:      yes - since Fri Sep 17 13:50:42 2021
     zone signing:   yes - since Fri Sep 17 13:50:42 2021

     Next rollover scheduled on Sun Oct 17 12:45:42 2021
     …
```

Key 15296 is the KSK, while 55383 is the ZSK. We have only one ZSK, and the next rollover is scheduled for a month from now, so we can roll the KSK with minimal risk.

Start the rollover with `rndc dnssec -rollover`, just as for a ZSK. Use `-key` to specify a key tag.

```
$ rndc dnssec -rollover -key 15296 mwlucas.org
Key 15296: Rollover scheduled on 21-Sep-2021 16:24:29.000
```

BIND acknowledges receipt of its instructions. Let's see what it did.

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: dm1
current time:  Tue Sep 21 16:25:28 2021

key: 15296 (RSASHA256), KSK
     published:      yes - since Fri Sep 17 13:50:18 2021
     key signing:    yes - since Fri Sep 17 13:50:18 2021

     Key will retire on Wed Sep 22 19:29:29 2021
```

Our old KSK is flagged for removal.

```
key: 61372 (RSASHA256), KSK
     published:      yes - since Tue Sep 21 16:24:29 2021
     key signing:    yes - since Tue Sep 21 16:24:29 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         rumoured
     - ds:             hidden
     - key rrsig:      rumoured
```

Key 61372 is our new KSK, created right now. The critical point here is that the *ds* entry is shown as "hidden." The Chain of Trust does not yet know about this new key.

For our first KSK rollover, we're going to be extra careful. Once you understand the process and are comfortable with how the rollover works, we will optimize the procedure to minimize interaction with the parent zone. Go tell your parent zone about the new DS record. Once you've done that, wait until you can verify that the new record is available in the zone. Only once it's available to the public can you inform BIND the key is published.

```
$ rndc dnssec -checkds -key 61372 published mwlucas.org
KSK 61372: Marked DS as published since 21-Sep-2021
16:33:21.000
```

The zone status will not immediately change, but internally BIND starts its countdown to when it can start treating the new key as its primary.

This zone will now have two DS records in its parent zone. You section of the Chain of Trust has parallel links. If you check **`https://DNSViz.net`** you'll see the double links from **`.org`** to my domain.

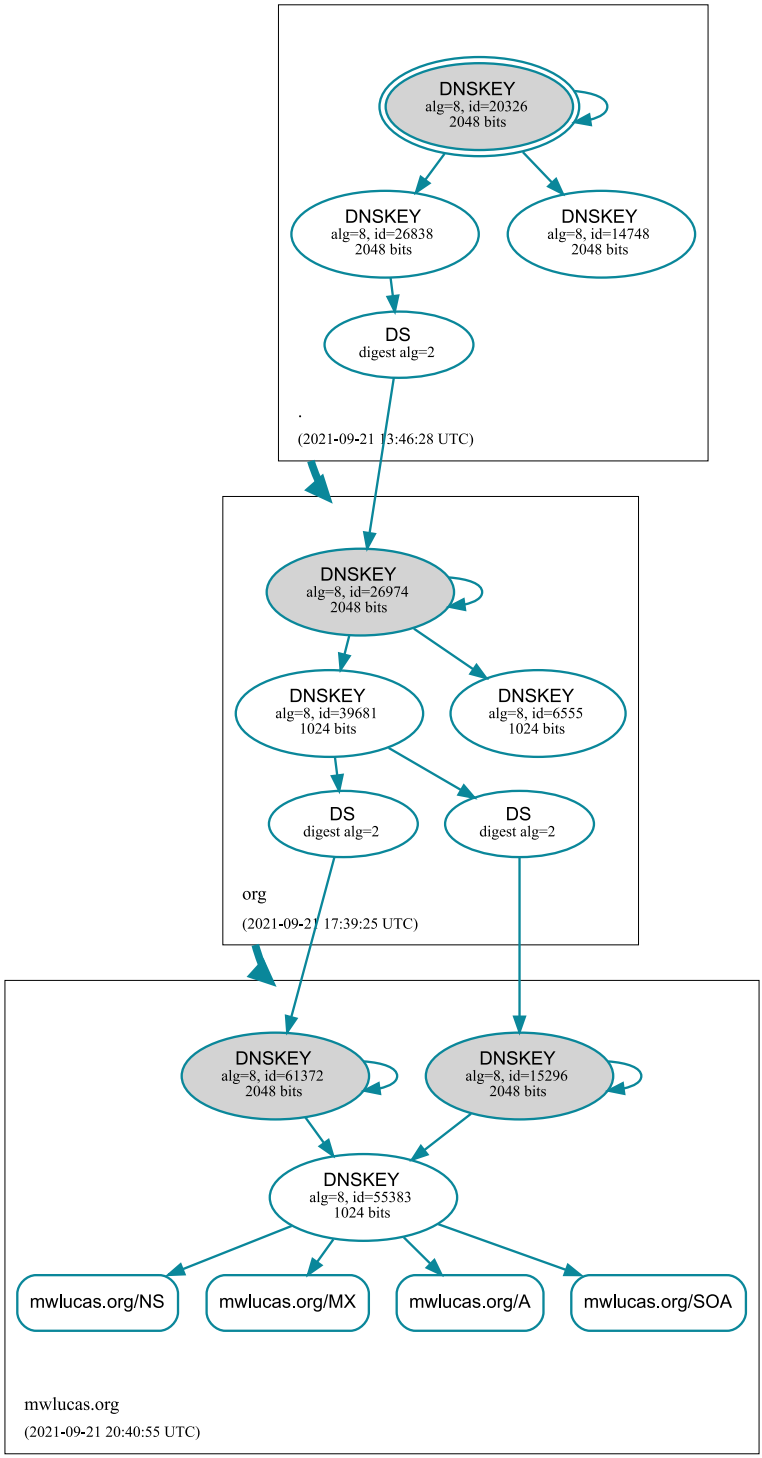

Figure 8-1: Chain of Trust during KSK Rollover with two published DS records

And now; we wait. BIND tracks the TTL as calculated from the various policy settings, so it can tell you when the new key is omnipresent.

```
$ rndc dnssec -status mwlucas.org
dnssec-policy: dm1
current time:  Wed Sep 22 11:36:30 2021

key: 15296 (RSASHA256), KSK
     published:      yes - since Fri Sep 17 13:50:18 2021
     key signing:    yes - since Fri Sep 17 13:50:18 2021

     Key will retire on Wed Sep 22 19:29:29 2021
     - goal:           hidden
     - dnskey:         omnipresent
     - ds:             unretentive
     - key rrsig:      omnipresent
```

This is our old key, 15296, now flagged for retirement on 22 September. Records and signatures with this key are still in broad use, however.

```
key: 61372 (RSASHA256), KSK
     published:      yes - since Tue Sep 21 16:24:29 2021
     key signing:    yes - since Tue Sep 21 16:24:29 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             rumoured
     - key rrsig:      omnipresent
```

The new key, 61372, is available everywhere. BIND does not probe remote DNS servers to see if they can retrieve the new record; the *omnipresent* state is calculated from the times in the policy. If you have any doubts at all, check the policy times again.

The new, omnipresent KSK means we can unpublish the old KSK that we are decommissioning. Go to your registrar and remove the old key's DS record. Once that's complete, tell BIND that you've withdrawn the key.

```
$ rndc dnssec -checkds -key 15296 withdrawn mwlucas.org
KSK 15296: Marked DS as withdrawn since
  22-Sep-2021 11:52:43.000
```

Your zone will include this key and signatures created with it until the retirement date. BIND will continue to use the withdrawn KSK to create signatures for as long as clients might have it cached. Withdrawn ZSKs will not be used to create more signatures. Put a note on your calendar to check the zone after the key is removed.

Minimizing Parent Zone Interactions in KSK Rollovers

Now that you have a grip on how keys flow through their lifecycle, you can optimize the rollover.

The advantage of the Double-signature method is that it minimizes contact with the parent zone. If you wait for the new DNSKEY and its signatures to become omnipresent, you can publish the new DS record and delete the old DS record simultaneously. As you have both DNSKEY records in your zone, clients who have either the old or new DS record will be able to validate your zones.

Once you verify that the new DS record is available in the parent zone, use `rndc` to withdraw the old key.

Rolling CSKs

Rolling a CSK is much like rolling a KSK. You generate new keys, wait for those keys to become omnipresent, then swap out the old key. Here's my main domain, mwl.io, that uses a CSK.

```
$ rndc dnssec -status mwl.io
dnssec-policy: default
current time:  Wed Sep 22 12:37:39 2021

key: 44950 (ECDSAP256SHA256), CSK
     published:      yes - since Tue Aug 24 15:48:56 2021
     key signing:    yes - since Tue Aug 24 15:48:56 2021
     zone signing:   yes - since Tue Aug 24 15:48:56 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             omnipresent
     - zone rrsig:     omnipresent
     - key rrsig:      omnipresent
```

One single key, available everywhere. Let's roll it. Use the `-rollover` option. Specify a key tag with `-key`.

```
# rndc dnssec -rollover -key 44950 mwl.io
Key 44950: Rollover scheduled on 22-Sep-2021 12:42:07.000
```

Check the zone status again to view the new key.

```
$ rndc dnssec -status mwl.io
dnssec-policy: default
current time:  Wed Sep 22 12:43:05 2021

key: 30172 (ECDSAP256SHA256), CSK
     published:      yes - since Wed Sep 22 12:42:07 2021
     key signing:    yes - since Wed Sep 22 12:42:07 2021
     zone signing:   no  - scheduled Wed Sep 22 14:47:07 2021

       No rollover scheduled
     - goal:            omnipresent
     - dnskey:          rumoured
     - ds:              hidden
     - zone rrsig:      hidden
     - key rrsig:       rumoured
     …
```

This is the new key, visible to the world but not yet used to sign keys. The old key is still present, but now has a retirement date.

```
key: 44950 (ECDSAP256SHA256), CSK
…
     Key will retire on Wed Sep 22 14:47:07 2021
```

Once the new key and its signatures are omnipresent, go to your registrar and swap out the old DS record for the new one, exactly as you would for a KSK rollover. Once the new DS record is visible in the parent zone, use `rndc` to withdraw the old key.

Changing A Zone's Policy

There's lots of reasons to switch between policies. Maybe RSASHA256 get cracked. Perhaps you started with a CSK, but regulators want you to use two keys. Maybe the boss read the wrong whitepaper. Whatever the reason, you can switch policies without too much trouble.

Policy changes don't throw away all the old stuff immediately. Rather, all new records are created as per the new policy. If you change key algorithms, BIND will create new key but retain the old ones so you can have an orderly rollover.

In an emergency you can use nsupdate to remove the old records and reload the zone, but that's ugly and error-prone. As with many things DNS, I recommend patience.

If you want more detail on how BIND policies behave, the policy engine is based on Schaeffer, Overeinder, and Mekking's paper "Flexible and Robust Key Rollover in DNSSEC." Videos based on the paper are also available.

We'll demonstrate by performing the most difficult policy change, an algorithm migration. My zone **immortalclay.com** uses a RSASHA1 KSK and ZSK, straight from the first edition of this book, as discussed in "Migrating Older Configurations to Policies" earlier this chapter. Let's migrate it to modern ECDSA-based keys policy, as in "KSK Policies."

Before changing anything, the zone looks like this.

```
$ rndc dnssec -status immortalclay.com
dnssec-policy: dm1
current time:  Wed Nov 10 16:26:02 2021

key: 38754 (RSASHA256), ZSK
     published:      yes - since Sat Oct 23 09:08:50 2021
     zone signing:   yes - since Sat Oct 23 11:13:50 2021

     Next rollover scheduled on Mon Nov 22 08:08:50 2021
     - goal:           omnipresent
     - dnskey:         omnipresent
     - zone rrsig:     omnipresent

key: 9318 (RSASHA256), ZSK
     published:      no
     zone signing:   no

     Key has been removed from the zone
     - goal:           hidden
     - dnskey:         hidden
     - zone rrsig:     hidden

key: 12107 (RSASHA256), KSK
     published:      yes - since Thu Sep 23 11:13:50 2021
     key signing:    yes - since Thu Sep 23 11:13:50 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             rumoured
     - key rrsig:      omnipresent
```

This zone uses the *dm1* policy.

Key 38754 is our current ZSK. It's omnipresent and happy.

Key 9318 was the old, dead ZSK. It's no longer in use, and no longer in the zone, but BIND still has the key file. The policy won't delete it until 90 days after it's no longer used. For clarity, I will delete it from further output.

Key 12107 is our current, omnipresent KSK.

A check of public DNS servers shows that this reflects what the world sees.

```
$ dig immortalclay.com +dnssec +nocrypto dnskey \
  +nocomment @8.8.8.8
; <<>> DiG 9.16.22 <<>> immortalclay.com +dnssec +nocrypto
dnskey +nocomment @8.8.8.8
;; global options: +cmd
;immortalclay.com.       IN  DNSKEY
immortalclay.com.  3588  IN  DNSKEY  257 3 8 [key id = 12107]
immortalclay.com.  3588  IN  DNSKEY  256 3 8 [key id = 38754]
immortalclay.com.  3588  IN  RRSIG   DNSKEY 8 2 3600 …
```

Go into `named.conf`, change the policy, and reload the nameserver. Then check the zone's status.

```
$ rndc dnssec -status immortalclay.com
dnssec-policy: ksk
current time:  Wed Nov 10 16:40:55 2021
```

The policy has changed! And with the change, the zone has sprouted two additional keys.

```
key: 29193 (ECDSAP256SHA256), ZSK
     published:      yes - since Wed Nov 10 16:40:27 2021
     zone signing:   yes - since Wed Nov 10 16:40:27 2021

     Next rollover scheduled on Tue Feb  8 14:35:27 2022
     - goal:           omnipresent
     - dnskey:         rumoured
     - zone rrsig:     rumoured
```

This is our brand new ZSK. It's currently used for signing. Clients will not have this key cached yet, so we can't quite rely on it. Wait for the key to become omnipresent before relying on it for validation.

```
key: 61011 (ECDSAP256SHA256), KSK
     published:      yes - since Wed Nov 10 16:40:27 2021
     key signing:    yes - since Wed Nov 10 16:40:27 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         rumoured
     - ds:             hidden
     - key rrsig:      rumoured
```

Our brand new KSK is in the zone, but is also not in any client's cache.

```
key: 38754 (RSASHA256), ZSK

     published:      yes - since Sat Oct 23 09:08:50 2021
     zone signing:   yes - since Sat Oct 23 11:13:50 2021

     Rollover is due since Wed Nov 10 16:40:27 2021
     - goal:           hidden
     - dnskey:         omnipresent
     - zone rrsig:     omnipresent
```

Our RSASHA256 ZSK is still present, but it's changed. Rollover is due. The policy knows this key needs to go away.

```
key: 12107 (RSASHA256), KSK
     published:      yes - since Thu Sep 23 11:13:50 2021
     key signing:    yes - since Thu Sep 23 11:13:50 2021

     Key will retire on Thu Nov 11 18:40:27 2021
     - goal:           hidden
     - dnskey:         omnipresent
     - ds:             rumoured
     - key rrsig:      omnipresent
```

Similarly, the RSASHA256 KSK is still present in the zone. The policy wants it to go away as well.

An algorithm migration is nothing more than a KSK rollover. You can either wait for the new key to become omnipresent and swap out the DS records, or publish the new DS immediately and return later to remove the old DS. This time, I'm publishing the new DS record immediately, and will return later to remove the old DS.

Once you've verified that the new DS is available in the parent zone, inform BIND that the key has been published.

```
$ rndc dnssec -checkds -key 61011 published immortalclay.com
KSK 61011: Marked DS as published since
   10-Nov-2021 16:55:44.000
```

As with any other KSK rollover, wait for the new keys to become omnipresent before removing the old DS record. DS records in `.com` currently have a TTL of 86400, so I can remove the old DS record then. As a final check, verify that BIND shows the new key as omnipresent before deleting the old DS.

```
$ rndc dnssec -status immortalclay.com
…
key: 61011 (ECDSAP256SHA256), KSK
     published:      yes - since Wed Nov 10 16:40:27 2021
     key signing:    yes - since Wed Nov 10 16:40:27 2021

     No rollover scheduled
     - goal:           omnipresent
     - dnskey:         omnipresent
     - ds:             omnipresent
     - key rrsig:      omnipresent
```

This key and everything related to it is omnipresent. Every recursive server that has looked up this zone should have the new key in cache. I can withdraw the old RSASHA256 keys.

```
$ rndc dnssec -checkds -key 12107 withdrawn immortalclay.com
KSK 12107: Marked DS as withdrawn since
     12-Nov-2021 13:49:04.000
```

BIND will count down the time clients might have this cached, as configured in the policy, and remove it from the zone.

Following these examples, you can roll keys any number of ways. That still leaves us with the problem of securing traffic between the client and the recursive server, though. We get into that next.

Chapter 9: Securing Clients

Authoritative nameservers use DNSSEC so that recursive servers can verify the integrity of zone data. User systems querying those recursive servers, however, perform integrity checks designed to detect accidental corruption rather than malice. This is fine for many corporate and home networks with local recursive servers, but what if you're on an untrusted network? What if your ISP or government intercepts and alters DNS traffic? What if you run a local network that's a valuable target, and want to reduce the chances of an intruder breaking DNS at the very last step? The traditional solution has been to run a validating resolver on the client system, but that gives network administrators dozens of hundreds of resolvers to troubleshoot rather than a couple central ones. Many people use VPNs, but that only changes who you trust. We have two emerging standards for transporting client DNS securely: DNS over TLS and DNS over HTTPS.

DNS over TLS, or *DoT*, uses standard TLS over TCP port 853. Where traditional DNS could manage simple queries in a single packet each way, with DoT those simple queries require several packets. Most users will not notice the difference, however.

DNS over HTTPS, or *DoH*, performs its queries using standard HTTP queries over TLS on port 443. Nameservers are identified by an URI, like `https://dns.google.com/dns-query`. To an outside observer or a network manager, DoH is difficult to differentiate from regular HTTPS traffic. Users might consider this an advantage—after all, who wants their ISP spying on them? Unfortunately, all it does is change who you trust.

Both DoH and DoT fundamentally change client behavior.

DoH/DoT and Client Resolvers

Traditionally, the operating system manages DNS lookups. An application like a web browser asks the operating system to look up DNS information. The operating system queries the recursive server and returns an answer to the application. Often, the operating system maintains a cache of answers.

With DoH and DoT, the application bypasses the operating system and queries recursive servers directly. If the application's DoH/DoT servers are not reachable or don't return an answer, they might fall back to the operating system's offerings. Your operating system's resolver becomes the system default, while applications can have their own preferred resolvers.

When an application uses a different nameserver than the operating system, troubleshooting becomes much more difficult. While the inability to ping a web site has always been a dubious diagnostic tool, it's even more irrelevant when applications perform their own DNS queries. This change frustrates and even infuriates many sysadmins and network operators[21] who are responsible for maintaining these applications.

Operating systems might or might not support direct DoH or DoT queries as I write this, but I expect that it will be an increasingly common feature.

Personal or Public DoH/DoT?

DNS is a core service needed to make the Internet work. Companies deployed their own DNS servers next to their mail and file servers. As with many other core services, though, Internet firms realized that they could financially exploit DNS query data. Firms like Cloudflare and Google have launched public DNS servers, providing this service for free in exchange for being allowed to record every web site clients visit and commercially exploit that data. This transformed a boring but necessary service into a point of contention for network managers and privacy advocates.

The question boils down to: who gets access to your data, and who can change your data?

With DoH and DoT, your queries are private as they cross the wire. The recursive server operator can log all your queries, however. A political activist working against an authoritarian government must decide if they

[21] Including me.

want their queries logged on the government-run recursive server, or by Google or Cloudflare.

Similarly, even with DNSSEC, the recursive server operator could change the answers you get. If the recursive server says NXDOMAIN, the client gets NXDOMAIN. Maybe that response isn't signed, but the client doesn't validate. Who is going to tamper the least with your queries?

If you're a network manager responsible for organizational security and data confidentiality, you should absolutely block DoH and DoT from leaving your network. Force clients to use the organization's recursive nameservers, proxy servers, and so on. While DNS is not a tunneling protocol, a modestly skilled sysadmin can tunnel SSH over DNS. Once you have SSH, you have everything.

The question of trust underlies the entirety of DNS. Integrity is a driving force behind DNSSEC. Your decisions on trust are your responsibility. Who do you trust the most—or, perhaps, who do you mistrust the least?

DoH/DoT and BIND

BIND supports integrated DoH and DoT as of 9.17, and that support should be backported into BIND 9.16. It will definitely be in BIND 9.18. If you're stuck on older versions of BIND, you'll find many tutorials and recipes on using third party software to provide front end proxies that allow clients to access your nameserver via DoH and DoT. We'll discuss BIND's native support, as provided in version 9.17. While the syntax appears to have stabilized, it's still relatively new so additional features might appear.

Configuring TLS in BIND

Both DoH and DoT use Transport Layer Security (TLS), and thus require an X.509 certificate. The application must recognize the CA that signed the certificate. If you want to use a self-signed certificate, you'll need to configure the certificate in your application or trust anchor bundle exactly like you would any other self-signed certificate. DNS is a non-interactive protocol, and provides no opportunity to click "ignore this warning and continue."

An X.509 certificate has two components, a certificate file and a key file. I've placed the files *tls.cert* and *tls.key* in the *namedb* directory. Before I can use them, I must define them in *named.conf*.

```
tls mysite {
     key-file "/etc/namedb/tls.key";
     cert-file "/etc/namedb/tls.cert";
};
```

The first line defines a certificate for TLS called *mysite*. Later in the configuration, I can refer to this combination of files as "mysite." I define the locations of the key file and the certificate file. If you have an intermediate certificate, use the full chain file rather than your certificate file. (All applications must *always* provide all X.509 intermediate certificates!) You could stash intermediate certificates in a separate file and give that file with the `ca-file` option, but a chain file is simplest.

If the key file uses a passphrase, you must provide the passphrase each time you restart named. For most organizations, protecting your DoH or DoT with a passphrase inflicts more pain than it prevents.

TLS certificates can use several encryption algorithms. The most popular are RSA and ECDSA. I recommend choosing the algorithm you're comfortable decrypting with a packet sniffer. Applications provide minimal support for debugging DoH and DoT. (The code is far more complex than for traditional DNS.) Sometimes, the only way to see what's going wrong is to break out tcpdump or Wireshark.

Configuring DNS over TLS

Configure DoT with a `listen-on` or `listen-on-v6` statement in the options section of *named.conf*. Here I tell named to listen for DNS over TLS on port 853 on any IP address on the host, using the certificate defined in *mysite*.

```
listen-on tls mysite { any ; } ;
```

You could add a `port` keyword to specify an alternate TCP port, exactly like a standard `listen-on` statement. Here we listen for DoT traffic on port 999.

```
listen-on 999 tls mysite { any ; } ;
```

Once you set a DoT listener and restart named, test it using dig's `+tls` option. If you're running DoT on an alternate port, set that port with `-p`.

```
$ dig mwl.io @ns3 +short +tls
45.63.79.193
```

DoT works!

Configuring DNS over HTTPS

Much like DoT, configure DNS over HTTPS with a `listen-on` or `listen-on-v6` keyword in the `options` section of *`named.conf`*. Here I tell named to listen for DNS over HTTPS on port 443 on any address it can attach to, using the certificate defined in *mysite*.

```
listen-on tls mysite http default { any ; } ;
```

It looks exactly like the DoT configuration, with the addition of the keywords `http default`. These keywords define an HTTP endpoint, a path on the web server where named should accept DNS queries. The default endpoint is */dns-query*. If my **`host ns3.mwl.io`** provided DoH queries with this configuration, I would configure my DoH clients to query **`https://ns3.mwl.io/dns-query`**. You could define other endpoints with the `http` keyword.

Test your new DoH server with dig before pointing a web browser at it. Use the `+https` option to invoke DoH.

```
$ dig mwl.io @dnssec-test +short +https
```

If compelled to run a web server and a DoH server on the same server, I might have BIND listen on an alternate port and make the web server proxy queries to it. Multiplexing applications on a single port gets complicated, and complications lead to outages. Separate your web servers from your DoH servers.

Now let's look at serving as a middle link in the Chain of Trust, or even a trust anchor.

Chapter 10: Delegations and Islands of Trust

Large zones often delegate child zones to other networks. Large corporations delegate zones to offices or even nations, Internet Service Providers delegate address space to customers, and even tiny organizations like mine might have subdomains for different offices. If you're the lucky one managing DNS at the head office or the ISP, you must be able to delegate zones to your clients. If you want these child zones protected with DNSSEC, you'll need to delegate the zones properly and enter the proper DS records.

Delegations

Most zones are child zones of another zone. They are often called *subdomains*, though that's not always strictly accurate. My zone **`mwl.io`** is a child zone of **`.io`**. Another way to say this is that **`.io`** has delegated **`mwl.io`** to me. When my writing empire goes global, I might need to set up my own delegations like **`us.mwl.io`** and **`uk.mwl.io`**, assign them their own nameservers, and eventually have hostnames like **`comfychair.manchester.uk.mwl.io`**.[22]

Zones all need NS records in their parent domains, declaring the zone's nameservers. It's okay to have the same nameservers serve the child and parent zones, but your name needs separate nameservers for them. If you want to have separate records beneath **`us.mwl.io`** and **`uk.mwl.io`**, they need separate zone files and separate NS records in the parent zone.
One of the most common uses for child zones is delegating reverse DNS for address space. A global firm might use 10/8 for internal addresses, and assign various /16 or /24 blocks to different divisions or locations. (They might also use RFC 2317 to delegate on non-octet boundaries, but demonstrating that here would only complicate the examples.) Such an organization has an internal authority server responsible for maintaining NS records for each delegation.

[22] Internet of Things or no, I will never permit my comfy chair to have an IP address.

Suppose our global firm has delegated 10.0/16 to the United States and 10.1/16 to the UK. You run the headquarters nameserver responsible for delegating zones to other divisions. Your authoritative nameserver for 10/8 needs three separate zone files, one for each zone. Here's the start of a zone file for the 10/8 zone, containing the NS records for each child domain.

```
$TTL 3600
$ORIGIN 10.in-addr.arpa.
@ SOA mwl.io. mwl.mwl.io. 2021093002 3600 900 3600000 3600
     NS  ns1.mwl.io.
     NS  ns2.mwl.io.
;USA
     0  NS  ns1.mwl.io.
     0  NS  ns2.mwl.io.
;UK
     1  NS  ns3.mwl.io.
     1  NS  ns4.mwl.io.
```

The 10.0/16 zone on your authoritative server contains the NS records for /24 zones , along with any glue records recursive clients require to find those servers. Similarly, the 10.1/16 zone needs its NS and glue records.

The delegated server that's authoritative for 10.1/16 doesn't need any records for each zone's desktops, file servers, or the other usual DNS entries. It only needs NS and glue records to guide recursive clients to the authoritative servers for zones like 10.1.0/24, exactly like the example above.

Only once you reach the bottom of the delegation chain do you need to enter actual host-specific records.

Once your clients can successfully and routinely perform reverse DNS checks on your internal addresses, and you've verified that every level of the delegation is properly delegated with its own NS records, you can proceed. Double-check your work with a series of queries. Again, you must query your own nameservers, not public ones.

```
$ dig -x 10 any
```

This should get you a list of the authoritative nameservers for 10/8. Then pick a child zone you know should be delegated, like 10.0/16.

```
$ dig -x 10.0 any
```

You should get a list of the authoritative nameservers for this zone. Pick a child zone of that zone that you know should be delegated, such as 10.0.1/24.

```
$ dig -x 10.0.1 any
```

You should get a list of authoritative nameservers for this child zone. Finally, grab the reverse DNS on a host you know should exist.

```
$ dig -x 10.0.1.1 +short
gateway.nyc.us.mwl.io.
```

Everything is properly delegated, and your client can recurse through the delegation chain to resolve individual hosts. You may proceed.

Adding DNSSEC to Delegations

Before trying to delegate DNSSEC to your delegated zones, make sure that your primary zone has valid signatures and that unrelated resolvers can validate them. If I want to secure child zones of **`mwl.io`**, I must first secure **`mwl.io`** itself. The Chain of Trust doesn't work if there's a broken link in the middle. If you're delegating private address space or a truly private domain that's not available to the outside world, double-check that your recursive servers can perform routine DNSSEC validation.

When you add DNSSEC to a zone you're responsible for, you create keys for your zone and submit the DS records to your registrar or RIR. The registrar adds the DS records to the parent zone. When you run the parent zone, you're responsible for adding DS records for child zones. DS records are public information, so don't resort to extraordinary means to ensure confidentiality. While registrars have pretty web sites for submitting DS records, your organization probably doesn't. How you transmit and verify those records depends on your organization and security profile, but you'll end up with a zone file that looks like this.

```
$TTL 3600
$ORIGIN 10.in-addr.arpa.
@ SOA mwl.io. mwl.mwl.io. 2021093002 3600 900 3600000 3600
     NS  ns1.mwl.io.
     NS  ns2.mwl.io.
;USA
     0  NS  ns1.mwl.io.
  0   NS  ns2.mwl.io.
  0   DS  6696 13 2 F838747750FC505D90C20F2E9317E1A834ACB479…

;UK
     1  NS  ns3.mwl.io.
     1  NS  ns4.mwl.io.
     1  DS  408 13 2 6D25E5CAAC3FB76D645204DEAAB5E6D789FB83F…
```

Reload the zone, and these DS records will attach the child zone to the Chain of Trust.

If you're using BIND, securing a zone you've delegated is exactly like securing a zone delegated to you, and is discussed in Chapter 5. Apply a policy to the zone to generate keys. Run `dnssec-dsfromkey` on the CSK or KSK to get the DS record. Send that record to the parent zone administrator. You're done.

If you have problems, fall back to the troubleshooting steps in Chapter 6. The most common errors include bad clocks and DS records that don't match the child zone's KSK or CSK—the exact same errors you're likely to make when dealing with your zone's parent domain.

Islands of Trust

Sometimes organizations want completely private zones. The most common case is when they're using RFC 6761 addresses like 10/8, 172.16/12, and 192.168/16, and must provide their own reverse DNS. You can secure these private addresses with DNSSEC, at least for your internal clients that use your own recursive nameservers.

Many organizations also want a private domain name for use inside their firewall. Certain protocols use **`.local`**, but that's reserved for mDNS. Domains like **`.test`** and **`.example`** are available for internal use, but using them in production looks ramshackle. Some large organizations create their own top-level domain, but ICANN is constantly licensing new top-level domains and eventually someone's DNS is going to flop down right on top

of you. You have two choices. The domain **home.arpa** is specifically assigned for use in private networks, such as your house. If you are a big company and don't want to look like a home network, register a public domain name and give it minimal or non-existent public DNS. If you insist on making something up, though, you can still secure it with DNSSEC by creating a new trust anchor.

A trust anchor is the public key of a KSK. If you tell your recursive nameservers to trust that key, you'll have your own Chain of Trust that doesn't rely on the outside world. This is often called an *island of trust*.

Before starting, remember that querying a zone's authoritative servers never returns authenticated data. If you want DNSSEC validation on, say, **10.in-addr.arpa**, you can't query the authoritative server. Instead, you must configure a recursive server and configure it to query the zone's authoritative server. If your nameservers are both authoritative and recursive, you cannot deploy an island of trust without very tricky configurations. Tricky configurations are fragile, and virtual machines are cheap.

Sign your private zones exactly as you would any public zone, and enter the DS records in the parent zones.

Preparing the Trust Anchor

The KSK for the uppermost private zone is your private trust anchor. You must deploy it on your recursive, validating nameservers. If you're deploying DNSSEC on 10/8, and you have /16 and /24 delegations through the network, 10/8's KSK is the trust anchor for the entire zone. Go into the key directory for that zone and find the KSK or CSK file. The first line will identify the key's role, while the last line is the key.

```
; This is a key-signing key, keyid 22456, for
; 10.in-addr.arpa.
…
10.in-addr.arpa. 3600 IN DNSKEY 257 3 13 NflrczcSQAK891ZfcL…
```

Verify this with dig.

```
$ dig zonename dnskey @nameserver +nocomments +nostats
```

This spits out all the DNSKEY records for the zone, and you should see your key among the results.

Key 22456 is our KSK. Pull that last line into a separate file.

```
$ tail -1 K10.in-addr.arpa.+013+22456.key \
  > /tmp/trust-anchor.txt
```

We'll edit the key so that it's suitable for a BIND trust anchor. Other nameservers use a similar but not necessarily identical format.

```
10.in-addr.arpa.  3600  IN  DNSKEY  257 3 13 NflrczcSQ…
```

Remove the TTL (3600), the IN, and the DNSKEY from the line. Replace them with the string `static-key`. Put double quotes around the address and the key, and put a semicolon at the end.

```
"10.in-addr.arpa." static-key 257 3 13 "NflrczcSQ…";
```

Save the file. This is your Island of Trust's private trust anchor. Now you can configure your recursive server to trust this key for the zone 10/8, and to use it for queries.

BIND and New Trust Anchors

BIND automatically maintains the public root zone trust anchor. For an island of trust, you must configure the private trust anchor. Add a `trust-anchors` stanza to your *`named.conf`*,

```
trust-anchors {
     zone static-key 257 3 algorithm key ;
};
```

The zone statement restricts BIND to using this trust anchor for only this zone. The `static-key` keyword tells BIND that it should not check this key for updates, the way it does the global trust anchor. The `257 3` indicates this is a key signing key used for DNSSEC. Give the algorithm number next, and then the key itself, in quotes.

Our key for 10/8 looks like this.

```
trust-anchors {
     "10.in-addr.arpa." static-key 257 3 13 "NflrczcSQtJ3noM-
  vrRclohTXmJVdxsFxHIp2qCGO…";
};
```

Now tell BIND to refer queries for this domain to your internal authoritative servers rather than the public root servers. Forwarders neither validate nor cache, so don't forward queries. Use a *stub zone*. Stub zones direct their queries to specific nameservers, but the results are cached locally.

```
zone "10.in-addr.arpa." {
     type stub;
     primaries { 192.0.2.100; 203.0.113.100; };
};
```

Reload the nameserver. This recursive nameserver now uses your trust anchor on this zone.

Testing the Island of Trust

To verify your Island of Trust works, query your recursive nameserver.

```
$ dig -x 10.0.1.1 @localhost +ad
…
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0,
   ADDITIONAL: 1
…
;; ANSWER SECTION:
1.1.0.10.in-addr.arpa.  3010  IN  PTR gateway.nyc.us.mwl.io.
…
```

This query returns a PTR record, and the AD flag is set. This is authenticated data. Your DNSSEC works. You have a cozy little Island of Trust.

Now that you have a secure distributed database intended for hosts and addresses, let's round out our explorations by cramming other information into it.

Chapter 11: DNSSEC for Data Distribution

One of the huge problems of computing security infrastructure is the distribution of authentic information. The purpose of a Certificate Authority is to assess identities. OpenPGP relies on a Web of Trust, where individuals verify other individuals. Protocols such as Secure Shell (SSH) require users to manually verify each server's public keys. Organizations deploying VPNs must securely distribute configurations and mutual authentication details to the users. DNSSEC offers a solution to this entire category of problems. An organization using DNSSEC has a cryptographically verified distribution method for non-confidential information, including public keys.

Some vendors already use DNS for their applications and appliances. More than one VPN client looks for an IPSECKEY record for the VPN concentrator, and will not connect without it. Without DNSSEC, bad actors can alter that record and redirect clients any way they want. Vendors of such devices provide specific information for their products, so I'm going to look at two more generally useful, standards-based cases for DNSSEC-based data distribution: distributing SSH host keys, and verifying self-signed X.509 certificates.[23]

SSH Host Key Fingerprints

The SSH protocol uses public key cryptography to verify that the server you're connecting to is truly the server you think it is, and to conceal the data exchanged between client and server. Unfortunately, correct use of SSH requires that the first time a user logs into an SSH server, she must examine the offered public key and compare it to an out-of-band copy of the server's public key. Even those that do perform such checks are most often perfunctory rather than meticulous. Worse, most SSH users learn to ignore warnings about public key problems. I've seen multiple new sysadmins become so numb to messages about unknown keys that they ignore warnings about mismatched keys. Teaching users to ignore security issues is bad.

---

[23] "Why no mention of HTTPSSVC?" Well, where's my RFC?

You can distribute SSH host keys via DNS by using *SSH Fingerprint* (SSHFP) records, defined in RFC 4255. As host keys are critical to identifying servers, SSHFP records must be immune from tampering. Do not deploy SSHFP records until you have fully functional DNSSEC.

Using SSHFP records requires creating the records, inserting the records into your zone, and configuring the client to use them.

Creating SSHFP Records

The ssh-keygen(8) program has a flag to read existing SSH public key files and create formatted SSHFP records. When you use the `-r` flag and specify a hostname, it scans `/etc/ssh` for public key files and spits out the records.

```
$ ssh-keygen -r mail
mail IN SSHFP 1 1 357ea20d767893fc050b599984bbb8d29bf5090b
mail IN SSHFP 1 2 da611f28ed47843630f03896ce26314b9a443d880f…
mail IN SSHFP 3 1 a2525ff9a485be1c00d4258346a23c781ce0abd0
mail IN SSHFP 3 2 7d7f6ec211a216dadb6f3bb1b809a9e58248f15ee1…
mail IN SSHFP 4 1 13ede0ad31203f03fa2e9ef02b2924f3f997e4dd
mail IN SSHFP 4 2 d938ece8beb04be200ce9123788c71b12be3b8671e…
```

This command displays the SSHFP records for all the host keys on the local machine, and formats records for a host called **`mail`**. This record will be specific to the particular zone, and this particular hostname. If you or your users SSH to this host using one of several names—that is, if **`mail`** also serves as **`www`**—you'll need SSHFP for each variant name.

Insert the SSHFP record into the zone exactly like any other resource record.

Configuring the Client

The two most popular SSH clients are PuTTY and OpenSSH. PuTTY does not support SSHFP records. OpenSSH does, and is included with most Unixes, Mac OS, and even Windows. OpenSSH uses the `VerifyHostKeyDNS` option to tell ssh(1) to check for SSHFP records on hosts. It's set to `yes` by default, but some vendors turn it off when they bundle OpenSSH.

When I SSH to this host for the first time,

```
$ ssh -p 822 mail
Last login: Mon Oct  4 13:45:17 2021 from headdesk.mwl.io
FreeBSD 12.2-RELEASE-p7 GENERIC
```

I am not prompted to compare host keys. If I added the `-v` option when connecting, I would see `ssh` find the host key in DNSSEC, perform the comparison, and accept the host's public key. When OpenSSH finds an SSHFP record, it does not update *`known_hosts`*. This eliminates the well-known problem of obsolete keys being used to spoof a server.

Search order is critical. The entries for **`mail.mwl.io`** and **`mail.mwlucas.org`** point to the same machine. Suppose my client searches **`mwlucas.org`** first, but I put the SSHFP records in **`mwl.io`**. My client will find the entry for **`mail.mwlucas.org`** first, so it won't see the SSHFP records. You'll get prompted to accept the host key. You can avoid this by carefully managing the domain search order, using OpenSSH's `CanonicalDomains` directive, or *`ssh_config`* alias entries.

Distributing SSHFP records via DNSSEC-protected DNS removes feeble users like you and I from authenticating hosts. Implement it if you can. If you want to learn more about SSH, permit me to recommend my book *SSH Mastery* (Tilted Windmill Press, 2018).

X.509 Certificate Verification: TLSA

Transport Layer Security (TLS), previously known as Secure Sockets Layer (SSL), is the industry-standard way to identify entities on the Internet and encrypt the data they exchange. TLS uses X.509 certificates to attest to the identity of servers and users, permitting entities to communicate securely without having to do any SSH-like key verification.

X.509 certificates are issued by Certificate Authorities (CAs). Applications like browsers and mail servers use lists of valid CAs, and automatically recognize certificates issued by them. While ACME-powered free CAs have eliminated the expense of routine certificates, Certificate Authorities are still vulnerable to fraud and trickery.

By using DNS-Based Authentication of Named Entities (DANE), your organization can provide its own independent declaration of the valid X.509 certification associated with a TCP/IP port in a TLSA record. TLSA is not an acronym, it's the type of DNS record.[24] When a client application that

[24] Presumably, the IETF is leaving space for TLSB, TLSC, and so on once the world exposes all the abuses and flaws they didn't imagine.

supports DANE connects to a TLS-protected port, it checks for a DNSSEC-secured TLSA record for that host and port. The client compares the TLSA record to the certificate. If everything matches, the client accepts the certificate. RFC 6698 documents DANE and TLSA.

Applications even accept self-signed certificates authenticated by TLSA. This is a slightly different trust model than that expected by traditional X.509. DANE assumes that the organization that that controls the DNS for a site also controls the server. The CA model assumes that the CA does a credible job of verifying certificate requests. Both have weaknesses. DANE can be applied to CA-issued certificates, and can be used to require certificates be signed by a particular trusted CA, so you can combine the two models to achieve the best and worst of both. (Which is the best, and which is the worst, is up to you.)

While TLSA was originally viewed as a way to bypass Certificate Authorities and authenticate self-signed X.509 certificates, Let's Encrypt's free CA certificates effectively ended that need. If you research DANE you'll see references to browser plug-ins that check TLSA records, but those plugins are unmaintained and obsolete. Today's most common consumer of TLSA records are SMTP servers, which traditionally accept self-signed certificates.

TLSA Record Format

The key to DANE is the TLSA record, which provides either a hash of the certificate's public key or the actual public key. The TLSA record looks like this.

```
_port._protocol.hostname TLSA (usage selector match key)
```

The *port* is the TCP/IP port this record represents. A host can provide different X.509 certificates on different ports, so your email team can maintain their certificate without access to the web site certificate.

The *protocol* is the TCP/IP protocol. TLSA supports TCP, UDP, and SCTP. Give the protocol in lower case.

The *hostname* is the host using the certificate.

The *usage* field describes constraints on the type of certificate that can be used on this host and port. I'll cover those in "CA Usage Polices" later in this chapter.

The *selector* shows how to compare the TLSA record to the site certificate. If set to 0, the client analyzes the complete certificate. If set to 1, it checks the certificate's `SubjectPublicKeyInfo` field.

The *match* field declares what sort of data the record contains. A 0 indicates a full public key, a 1 is a SHA-256 hash, and a 2 is a SHA-512 hash. The full public key is not usually necessary. Not all clients support SHA-512, so SHA-256 is the best choice. While I'll refer to this field as containing a hash, remember that you can put a full public key here if you wish.

CA Usage Policies

The usage field lets you define how this TLSA record interacts with a Certificate Authority.

A 0 means that the record contains the public key of a CA signing certificate, or a hash thereof. This means that a specific official CA must have used a particular signing certificate to sign the X.509 certificate used on this port. While it seems to be an obvious choice for many uses, it requires that client and server agree on which CAs are valid. Anyone who has studied trust anchor bundles provided by the major vendors will tell you that this is a poor assumption. On the other hand, you don't have to worry about rolling your TLSA record properly when you renew the certificate with the same CA.

A 1 means that the record contains the public key of a CA-signed certificate. The service this record represents must have an official CA-signed certificate, and this record represents that information. When the sysadmin renews the certificate, this record must be updated. Like 0, this usage is considered fragile and is not recommended.

A 2 means that the record contains the public key of a private certificate authority. If you run your own internal CA and use it to sign certificates exposed to the public Internet, usage 2 might be for you.

A 3 declares that the record contains the public key of a certificate. Any kind of certificate, signed by anyone. This is the most robust usage.

Creating TLSA Records

I recommend using NLnet Lab's ldns-dane(1) for creating and verifying TLSA records. It's part of their ldns toolkit. If you want a perilous journey into exotic OpenSSL commands instead, though, you're welcome to that adventure.

Start by deciding which CA use policy, selector, and match you want to set in your record. I recommend a CA use policy of 3 for most sites, declaring that the record contains the public key of a specific certificate. You must update the TLSA record whenever you update the certificate, but that's not onerous with ldns-dane. The most common selector is 0, telling the client to analyze the whole certificate. Finally, the match declares what sort of hash you're providing. A 1 is sufficient. You'll set these TLSA fields to 3 0 1.

The ldns-dane program can create new TLSA records with the `create` option. You provide the host and port the TLS service is on, and then the chosen policy, selector and match.

```
$ ldns-dane create host port 3 1 0
```

I want to create a TLSA record for my host mwl.io. I have a web server on port 443.

```
$ ldns-dane create mwl.io 443 3 1 0
_443._tcp.mwl.io.  3600  IN  TLSA 3 0 1 fe3eb7ae334921291da3…
```

That's my TLSA record. I copy it into my zone, and it's ready. I'll need to add records for **`www.mwl.io`** as well, even though it's the same host. The mail exchanger is a different host, so **`mail.mwl.io`** needs a different record for its SMTP.

If you're maintaining your X.509 certificates with ACME, you can add ldns-dane calls into your renewal process and automate TLSA record maintenance.

Verifying TLSA Records

Running dig on your zone shows that the record exists, but how can you double-check that clients can use this string of gobbledygook to verify your X.509 certificate?

The ldns-dane program can verify existing records, with the (wait for it) `verify` option. Give the hostname and port.

```
$ ldns-dane verify mwl.io 443
45.63.79.193 dane-validated successfully
2001:19f0:5c01:fcf:5400:1ff:fe4b:e2df dane-validated successfully
```

Run this command for each hostname and port you want to verify.

If you want off-network confirmation, several folks have helpfully provided public TLSA validators. If you're using TLSA for HTTPS, the validator at **`https://check.sidnlabs.nl/dane/`** is fast and convenient. The validator at **`https://www.huque.com/bin/danecheck`** is more flexible. The best way to make a web site disappear is for me to mention it in a book, but you won't have any trouble finding other validators.

Rolling TLSA Records

You get a new certificate, but other people's nameservers have the old certificate's TLSA record cached. How do you properly roll the TLSA record to support the new certificate?

Once you have the new certificate, don't immediately deploy it. Create the new TLSA record and add it to the zone, so you have two TLSA records. Wait for the new TLSA record to become omnipresent. Once that happens, deploy the new certificate and remove the old record from the zone.

With these examples, your DNS can provide any information with the guaranteed integrity of DNSSEC.

Afterword

After a lifetime of abuse, DNS finally has an integrity verification infrastructure. People have tested and honed DNSSEC for decades, and it's now fairly straightforward. While DNSSEC might seem complex, when you first encountered DNS proper it probably confused you. Deploying DNSSEC will enhance your trust in your own network resources even as it improves your trust in other people's DNS. Your users can rest assured that the goofy video they're watching is the goofy video their mom sent.

If you're interested in learning more about DNSSEC, check out resources like the Secure Domain Name System (DNS) Deployment Guide and the papers at https://www.dnssec.net.

I've lost count of the number of folks who told me that the first edition of this book helped them not only deploy DNSSEC, but *understand* it. Hearing that delights me. If this book helps you, please tell your peers and leave a review at your favorite online bookstore.

Sponsors

You might have heard about this thing called "advances" that writers are supposed to get. Like "justice" and "dinner," they're not exactly mythical—but if we want them, we have to make them ourselves. The following fabulous people wanted this book to exist, and sent me money to support me as I wrote.

Thank you all, even if you were a notable smart aleck and sent me several small sponsorships in amounts corresponding to the DNSSEC RFCs. I'll find a way to give you the credit you deserve… perhaps by upgrading you from a several-times-over ebook sponsor to a print sponsor.

Print Sponsors

Brad Ackerman, Bob Beck, Xavier Belanger, Patrick Bucher, Chris Dunbar, Bob Eager, Trond Endrestøl, Joachim Ernst, Trix Farrar, David Hansen, Philip Jocks, Bob Beck, Bob Beck, Bernd Kohler, Rogier Krieger, Andreas Maier, Craig Maloney, Doug McIntyre, Jan-Piet Mens, Niall Navin, John O'Brien, Lex Onderwater, Dan Parriott, Phi Network Systems, Matthew Silvey, Brad Sliger, Carsten Strotmann, tanamar corporation, Andreas Taudte, and Bob Beck.

Index

About the Author

https://mwl.io

Never miss a new release! Sign up for Lucas' mailing list at his web site.

More Tech Books from Michael W Lucas

Absolute BSD —Absolute OpenBSD (1st and 2nd edition)
Cisco Routers for the Desperate (1st and 2nd edition) — PGP and GPG
Absolute FreeBSD (2nd and 3rd edition) — Network Flow Analysis

the IT Mastery Series

SSH Mastery (1st and 2nd edition) — DNSSEC Mastery (1st and 2nd edition)
Sudo Mastery (1st and 2nd edition) — FreeBSD Mastery: Storage Essentials
Networking for Systems Administrators — Tarsnap Mastery
FreeBSD Mastery: ZFS — FreeBSD Mastery: Specialty Filesystems
FreeBSD Mastery: Advanced ZFS — PAM Mastery
Relayd and Httpd Mastery — Ed Mastery — FreeBSD Mastery: Jails
SNMP Mastery — TLS Mastery

The Networknomicon
(only for readers who are over 18 and, preferably, already dead)

Other Nonfiction

Domesticate Your Badgers
Cash Flow For Creators
Only Footnotes

Books and Novels (as Michael Warren Lucas)

Immortal Clay — Kipuka Blues
Butterfly Stomp Waltz — Terrapin Sky Tango
Forever Falls — Hydrogen Sleets — Drinking Heavy Water
Aidan Redding Against the Universes
git commit murder — git sync murder
Prohibition Orcs (coming 2022)

See your local bookstore for more!