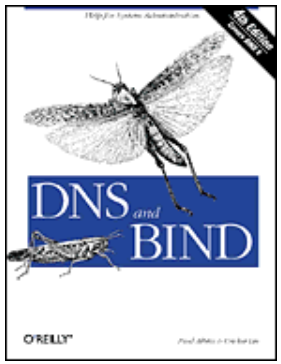


DNS and BIND, 4th Edition

By Paul Albitz & Cricket Liu
4th Edition May 2001
0-596-00158-4, Order Number: 1584
622 pages, $44.95

Chapter 11
Security

In this chapter:
TSIG
Securing Your Name Server
DNS and Internet Firewalls
The DNS Security Extensions

"I hope you've got your hair well fastened on?"
he continued, as they set off.

"Only in the usual way," Alice said, smiling.

"That's hardly enough," he said, anxiously.
"You see the wind is so very strong here.
It's as strong as soup."

"Have you invented a plan for keeping the hair from being blown off?" Alice enquired.

"Not yet," said the Knight. "But I've got a plan for keeping it from falling off."

Why should you care about DNS security? Why go to the trouble of securing a service that mostly maps names to addresses? Let us tell you a story.

In July 1997, during two periods of several days, users around the Internet who typed *www.internic.net* into their web browsers thinking they were going to the InterNIC's web site instead ended up at a web site belonging to the AlterNIC. (The AlterNIC runs an alternate set of root name servers that delegate to additional top-level domains with names like *med* and *porn*.) How'd it happen? Eugene Kashpureff, then affiliated with the AlterNIC, had run a program to "poison" the caches of major name servers around the world, making them believe that *www.internic.net*'s address was actually the address of the AlterNIC web server.

Kashpureff hadn't made any attempt to disguise what he had done; the web site that users reached was plainly the AlterNIC's, not the InterNIC's. But imagine someone poisoning your name server's cache to direct www.amazon.com or www.wellsfargo.com to his own web server, conveniently well outside local law enforcement jurisdiction. Further, imagine your users typing in their credit card numbers and expiration dates. Now you get the idea.

Protecting your users against these kinds of attacks requires DNS security. DNS security comes in several flavors. You can secure transactions--the queries, responses, and other messages your name server sends and receives. You can secure your name server, refusing queries, zone transfer requests, and dynamic updates from unauthorized addresses, for example. You can even secure zone data by digitally signing it.

Since DNS security is one of the most complicated topics in DNS, we'll start you off easy and build up to the hard stuff.

TSIG

BIND 8.2 introduced a new mechanism for securing DNS messages called *transaction signatures*, or TSIG for short. TSIG uses shared secrets and a one-way hash function to authenticate DNS messages, particularly responses and updates.

TSIG, now codified in RFC 2845, is relatively simple to configure, lightweight for resolvers and name servers to use, and flexible enough to secure DNS messages (including zone transfers) and dynamic updates. (Contrast this with the DNS Security Extensions, which we'll discuss at the end of this chapter.)

With TSIG configured, a name server or updater adds a TSIG record to the additional data section of a DNS message. The TSIG record "signs" the DNS message, proving that the message's sender had a cryptographic key shared with the receiver and that the message wasn't modified after it left the sender.[1]

One-Way Hash Functions

TSIG provides authentication and data integrity through the use of a special type of mathematical formula called a *one-way hash function.* A one-way hash function, also known as a cryptographic checksum or message digest, computes a fixed-size hash value based on arbitrarily large input. The magic of a one-way hash function is that each bit of the hash value depends on each and every bit of the input. Change a single bit of the input and the hash value changes dramatically and unpredictably--so unpredictably that it's "computationally infeasible" to reverse the function and find an input that produces a given hash value.

TSIG uses a one-way hash function called MD5. In particular, it uses a variant of MD5 called HMAC-MD5. HMAC-MD5 works in a keyed mode in which the 128-bit hash value depends not only on the input, but also on a key.

The TSIG Record

We won't cover the TSIG record's syntax in detail because you don't need to know it: TSIG is a "meta-record" that never appears in zone data and is never cached by a resolver or name server. A signer adds the TSIG record to a DNS message, and the recipient removes and verifies the record before doing anything further, such as caching the data in the message.

You should know, however, that the TSIG record includes a hash value computed over the entire DNS message as well as some additional fields. (When we say "computed over," we mean that the raw, binary DNS message and the additional fields are fed through the HMAC-MD5 algorithm to produce the hash value.) The hash value is keyed with a secret shared between the signer and the verifier. Verifying the hash value proves both that the DNS message was signed by a holder of the shared secret and that it wasn't modified after it was signed.

The additional fields in the TSIG record include the time the DNS message was signed. This helps combat replay attacks, in which a hacker captures a signed, authorized transaction (say a dynamic update deleting an important resource record) and replays it later. The

recipient of a signed DNS message checks the time signed to make sure it's within the allowable "fudge" (another field in the TSIG record).

Configuring TSIG

Before using TSIG for authentication, we need to configure one or more TSIG keys on either end of the transaction. For example, if we want to use TSIG to secure zone transfers between the master and slave name servers for *movie.edu*, we need to configure both name servers with a common key:

```
key terminator-wormhole.movie.edu. {
        algorithm hmac-md5;
        secret "skrKc4Twy/cIgIykQu7JZA==";
};
```

The argument to the *key* statement in this example, *terminator-wormhole.movie.edu.*, is actually the name of the key, though it looks like a domain name. (It's encoded in the DNS message in the same format as a domain name.) The TSIG RFC suggests you name the key after the two hosts that use it. The RFC also suggests that you use different keys for each pair of hosts.

It's important that the name of the key--not just the binary data the key points to--be identical on both ends of the transaction. If it's not, the recipient tries to verify the TSIG record and finds it doesn't know the key that the TSIG record says was used to compute the hash value. That causes errors like the following:

```
Nov 21 19:43:00 wormhole named-xfer[30326]: SOA TSIG verification from server [192.249.249.1], zone movie.edu: messa
had BADKEY set (17)
```

The algorithm, for now, is always *hmac-md5*. The secret is the base 64 encoding of the binary key. You can create a base 64-encoded key using the *dnssec-keygen* program included in BIND 9 or the *dnskeygen* program included in BIND 8. Here's how you'd create a key using *dnssec-keygen*, the easier of the two to use:

```
# dnssec-keygen -a HMAC-MD5 -b 128 -n HOST terminator-wormhole.movie.edu.
Kterminator-wormhole.movie.edu.+157+28446
```

The *-a* option takes as an argument the name of the algorithm the key will be used with. (That's necessary because *dnssec-keygen* can generate other kinds of keys, as we'll see in the DNSSEC section.) *-b* takes the length of the key as its argument; the RFC recommends using keys 128 bits long. *-n* takes as an argument HOST, the type of key to generate. (DNSSEC uses ZONE keys.) The final argument is

the name of the key.

dnssec-keygen and *dnskeygen* both create files in their working directories that contain the keys generated. *dnssec-keygen* prints the base name of the files to its standard output. In this case, *dnssec-keygen* created the files *Kterminator-wormhole.movie.edu.+157+28446.key* and *Kterminator-wormhole.movie.edu.+157+28446.private*. You can extract the key from either file. The funny numbers (157 and 28446), in case you're wondering, are the key's DNSSEC algorithm number (157 is HMAC-MD5) and the key's fingerprint (28446), a hash value computed over the key to identify it. The fingerprint isn't particularly useful in TSIG, but DNSSEC supports multiple keys per zone, so identifying which key you mean by its fingerprint is important.

Kterminator-wormhole.movie.edu.+157+28446.key contains:

```
terminator-wormhole.movie.edu. IN KEY 512 3 157 skrKc4Twy/cIgIykQu7JZA==
```

and *Kterminator-wormhole.movie.edu.+157+28446.private* contains:

```
Private-key-format: v1.2
Algorithm: 157 (HMAC_MD5)
Key: skrKc4Twy/cIgIykQu7JZA==
```

You can also choose your own key and encode it in base 64 using *mmencode* :

```
% mmencode
foobarbaz
Zm9vYmFyYmF6
```

Since the actual binary key is, as the substatement implies, a secret, we should take care in transferring it to our name servers (e.g., by using *ssh*) and make sure that not just anyone can read it. We can do that by making sure our *named.conf* file isn't world-readable or by using the *include* statement to read the *key* statement from another file, which isn't world-readable:

```
include "/etc/dns.keys.conf";
```

There's one last problem that we see cropping up frequently with TSIG: time synchronization. The timestamp in the TSIG record is useful for preventing replay attacks, but it tripped us up initially because the clocks on our name servers weren't synchronized. That produced error messages like the following:

```
Nov 21 19:56:36 wormhole named-xfer[30420]: SOA TSIG verification from server [192.249.249.1], zone movie.edu:
```

```
BADTIME (-18)
```

We quickly remedied the problem using NTP, the network time protocol.[2]

Using TSIG

Now that we've gone to the trouble of configuring our name servers with TSIG keys, we should probably configure them to use those keys for something. In BIND 8.2 and later name servers, we can secure queries, responses, zone transfers, and dynamic updates with TSIG.

The key to configuring this is the *server* statement's *keys* substatement, which tells a name server to sign queries and zone transfer requests sent to a particular remote name server. This *server* substatement, for example, tells the local name server, *wormhole.movie.edu*, to sign all such requests sent to 192.249.249.1 (*terminator.movie.edu*) with the key *terminator-wormhole.movie.edu*:

```
server 192.249.249.1 {
        keys { terminator-wormhole.movie.edu.; };
};
```

Now, on *terminator.movie.edu*, we can restrict zone transfers to those signed with the *terminator-wormhole.movie.edu* key:

```
zone "movie.edu" {
        type master;
        file "db.movie.edu";
        allow-transfer { key terminator-wormhole.movie.edu.; };
};
```

terminator.movie.edu also signs the zone transfer, which allows *wormhole.movie.edu* to verify it.

You can also restrict dynamic updates with TSIG by using the *allow-update* and *update-policy* substatements, as we showed you in the last chapter.

The *nsupdate* programs shipped with BIND 8.2 and later support sending TSIG-signed dynamic updates. If you have the key files created by *dnssec-keygen* lying around, you can specify either of those as an argument to *nsupdate*'s *-k* option. Here's how you'd do that with BIND 9's version of *nsupdate* :

```
% nsupdate -k Kterminator-wormhole.movie.edu.+157+28446.key
```

or:

```
% nsupdate -k Kterminator-wormhole.movie.edu.+157+28446.private
```

With the BIND 8.2 or later *nsupdate*, the syntax is a little different: *-k* takes a directory and a key name as an argument, separated by a colon:

```
% nsupdate -k /var/named:terminator-wormhole.movie.edu.
```

If you don't have the files around (maybe you're running *nsupdate* from another host), you can still specify the key name and the secret on the command line with the BIND 9 *nsupdate* :

```
% nsupdate -y terminator-wormhole.movie.edu.:skrKc4Twy/cIgIykQu7JZA==
```

The name of the key is the first argument to the *-y* option, followed by a colon and the base 64-encoded secret. You don't need to quote the secret since base 64 values can't contain shell metacharacters, but you can if you like.

Michael Fuhr's Net::DNS Perl module also lets you send TSIG-signed dynamic updates and zone transfer requests. For more information on Net::DNS, see *Chapter 15, Programming with the Resolver and Name Server Library Routines*.

Now that we've got a handy mechanism for securing DNS transactions, let's talk about securing our whole name server.

Securing Your Name Server

BIND 4.9 introduced several important security features that help you protect your name server. BIND 8 and 9 continued the tradition by adding several more. These features are particularly important if your name server is running on the Internet, but they're also useful on purely internal name servers.

We'll start by discussing measures you should take on all name servers for which security is important. Then we'll describe a model in which your name servers are split into two communities, one for serving only resolvers and one for answering other name servers' queries.

BIND Version

One of the most important ways you can enhance the security of your name server is to run a recent version of BIND. All versions of BIND before 8.2.3 are susceptible to at least a few known attacks. Check the ISC's list of vulnerabilities in various BIND versions at http://www.isc.org/products/BIND/bind-security.html for updates.

But don't stop there: new attacks are being thought up all the time, so you'll have to do your best to keep abreast of BIND's vulnerabilities and the latest "safe" version of BIND. One good way to do that is to read the *comp.protocols.dns.bind* newsgroup regularly.

There's another aspect of BIND's version relevant to security: if a hacker can easily find out which version of BIND you're running, he may be able to tailor his attacks to that version of BIND. And, wouldn't you know it, since about BIND 4.9, BIND name servers have replied to a certain query with their version. If you look up TXT records in the CHAOSNET class attached to the domain name *version.bind*, BIND graciously returns something like this:

```
% dig txt chaos version.bind.

; <<>> DiG 9.1.0 <<>> txt chaos version.bind.
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 34772
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;version.bind.                  CH      TXT

;; ANSWER SECTION:
version.bind.           0       CH      TXT     "9.1.0"
```

To address this, BIND Versions 8.2 and later let you tailor your name server's response to the *version.bind* query:

```
options {
        version "None of your business";
};
```

Of course, receiving a response like "None of your business" will tip off the alert hacker to the fact that you're likely running BIND 8.2 or better, but that still leaves a number of possibilities.

Restricting Queries

Before BIND 4.9, administrators had no way to control who could look up names on their name servers. That makes a certain amount of sense; the original idea behind DNS was to make information easily available all over the Internet.

The neighborhood is not such a friendly place anymore, though. In particular, people who run Internet firewalls may have a legitimate need to hide certain parts of their namespace from most of the world while making it available to a limited audience.

The BIND 8 and 9 *allow-query* substatement lets you apply an IP address-based access control list to queries. The access control list can apply to queries for data in a particular zone or to any queries received by the name server. In particular, the access control list specifies which IP addresses are allowed to send queries to the server.

Restricting all queries

The global form of the *allow-query* substatement looks like this:

```
options {
      allow-query { address_match_list; };
};
```

So to restrict our name server to answering queries from the three main Movie U. networks, we'd use:

```
options {
      allow-query { 192.249.249/24; 192.253.253/24; 192.253.254/24; };
};
```

Restricting queries in a particular zone

BIND 8 and 9 also allow you to apply an access control list to a particular zone. In this case, just use *allow-query* as a substatement to the *zone* statement for the zone you want to protect:

```
acl "HP-NET" { 15/8; };

zone "hp.com" {
      type slave;
      file "bak.hp.com";
      masters { 15.255.152.2; };
      allow-query { "HP-NET"; };
```

```
};
```

Any kind of authoritative name server, master or slave, can apply an access control list to the zone. Zone-specific access control lists take precedence over a global ACL for queries in that zone. The zone-specific access control list may even be more permissive than the global ACL. If there's no zone-specific access control list defined, any global ACL will apply.

In BIND 4.9, this functionality is provided by the *secure_zone* record. Not only does it limit queries for individual resource records, it limits zone transfers, too. (In BIND 8 and 9, restricting zone transfers is done separately.) However, BIND 4.9 name servers have no mechanism for restricting who can send your server queries for data in zones your server is *not* authoritative for; the *secure_zone* mechanism works only with authoritative zones.

To use *secure_zone*, include one or more special TXT records in your zone data on the primary master name server. Conveniently, these records are transferred to the zone's slave servers automatically. Of course, only BIND 4.9 slaves will understand them.

The TXT records are special because they're attached to the pseudo-domain name *secure_zone*, and the resource record-specific data has a special format, too:

```
address:mask
```

or:

```
address:H
```

In the first form, *address* is the dotted-octet form of the IP network to which you want to allow access to the data in this zone. The *mask* is the netmask for that network. If you want to allow all of 15/8 access to your zone data, use 15.0.0.0:255.0.0.0. If you want to allow only the range of IP addresses from 15.254.0.0 to 15.255.255.255 access to your zone data, use 15.254.0.0:255.254.0.0.

The second form specifies the address of a particular host you'd like to allow access to your zone data. The *H* is equivalent to the mask 255.255.255.255; in other words, each bit in the 32-bit address is checked. Therefore, 15.255.152.4:H gives the host with the IP address 15.255.152.4 the ability to look up data in the zone.

If we want to restrict queries for information in *movie.edu* to hosts on Movie U.'s networks, but our name servers run BIND 4.9 instead of BIND 8 or 9, we could add the following lines to *db.movie.edu* on the *movie.edu* primary master:

```
secure_zone    IN    TXT    "192.249.249.0:255.255.255.0"
secure_zone    IN    TXT    "192.253.253.0:255.255.255.0"
secure_zone    IN    TXT    "192.253.254.0:255.255.255.0"
secure_zone    IN    TXT    "127.0.0.1:H"
```

Notice that we included the loopback address (127.0.0.1) in our access control list. That's so a resolver running on the same host as a name server can query the local name server.

If you forget the *:H*, you'll see the following *syslog* message:

```
Aug 17 20:58:22 terminator named[2509]: build_secure_netlist
       (movie.edu): addr (127.0.0.1) is not in mask (0xff000000)
```

Also, note that the *secure_zone* records here apply only to the zone they're in--that is, *movie.edu*. If you wanted to prevent unauthorized queries for data in other zones on this server, you'd have to add *secure_zone* records to those zones on their primary master name servers, too.

Preventing Unauthorized Zone Transfers

Arguably even more important than controlling who can query your name server is ensuring that only your real slave name servers can transfer zones from your name server. Users on remote hosts that can query your name server's zone data can only look up records (e.g., addresses) for domain names they already know, one at a time. Users who can start zone transfers from your server can list all of the records in your zones. It's the difference between letting random folks call your company's switchboard and ask for John Q. Cubicle's phone number and sending them a copy of your corporate phone directory.

BIND 8 and 9's *allow-transfer* substatement and 4.9's *xfrnets* directive let administrators apply an access control list to zone transfers. *allow-transfer* restricts transfers of a particular zone when used as a *zone* substatement, and restricts all zone transfers when used as an *options* substatement. It takes an address match list as an argument.

The slave servers for our *movie.edu* zone have the IP addresses 192.249.249.1 and 192.253.253.1 (*wormhole.movie.edu*) and 192.249.249.9 and 192.253.253.9 (*zardoz.movie.edu*). The following *zone* statement:

```
zone "movie.edu" {
      type master;
      file "db.movie.edu";
```

```
      allow-transfer { 192.249.249.1; 192.253.253.1; 192.249.249.9; 192.253.253.9; };
};
```

allows only those slaves to transfer *movie.edu* from the primary master name server. Note that because the default for BIND 8 or 9 is to allow zone transfer requests from any IP address, and because hackers can just as easily transfer the zone from your slaves, you should probably also have a *zone* statement like this on your slaves:

```
zone "movie.edu" {
      type slave;
      masters { 192.249.249.3; };
      file "bak.movie.edu";
      allow-transfer { none; };
};
```

BIND 8 and 9 also let you apply a global access control list to zone transfers. This applies to any zones that don't have their own explicit access control lists defined as *zone* substatements. For example, we might want to limit all zone transfers to our internal IP addresses:

```
options {
      allow-transfer { 192.249.249/24; 192.253.253/24; 192.253.254/24; };
};
```

The BIND 4.9 *xfrnets* directive also applies an access control list to all zone transfers. *xfrnets* takes as its arguments the networks or IP addresses you'd like to allow to transfer zones from your name server. Networks are specified by the dotted-octet form of the network number. For example:

```
xfrnets 15.0.0.0 128.32.0.0
```

allows only hosts on the network 15/8 or the network 128.32/16 to transfer zones from this name server. Unlike the *secure_zone* TXT record, this restriction applies to any zones the server is authoritative for.

If you want to specify just a part of a network, down to a single IP address, you can add a network mask. *network&netmask* is the syntax for including a network mask. Note that spaces aren't allowed between the network and the ampersand or between the ampersand and the netmask.

To pare down the addresses allowed to transfer zones in the previous example to just the IP address 15.255.152.4 and the subnet 128.32.1/24, use the *xfrnets* directive:

```
xfrnets 15.255.152.4&255.255.255.255 128.32.1.0&255.255.255.0
```

Finally, as we mentioned earlier in the chapter, those newfangled BIND 8.2 and later name servers let you restrict zone transfers to slave name servers that include a correct transaction signature with their request. On the master name server, you need to define the key in a key statement and then specify the key in the address match list:

```
key terminator-wormhole. {
        algorithm hmac-md5;
        secret "UNd5xYLjz0FPkoqWRymtgI+paxW927LU/gTrDyulJRI=";
};

zone "movie.edu" {
        type master;
        file "db.movie.edu";
        allow-transfer { key terminator-wormhole.; };
};
```

On the slave's end, you need to configure the slave to sign zone transfer requests with the same key:

```
key terminator-wormhole. {
        algorithm hmac-md5;
        secret "UNd5xYLjz0FPkoqWRymtgI+paxW927LU/gTrDyulJRI=";
};

server 192.249.249.3 {
        keys { terminator-wormhole.; };  // sign all requests to 192.249.249.3
                                         // with this key
};

zone "movie.edu" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.movie.edu";
};
```

For a primary master name server accessible from the Internet, you probably want to limit zone transfers to just your slave name servers. You probably don't need to worry about name servers inside your firewall, unless you're worried about your own employees listing your zone data.

Running BIND with Least Privilege

Running a network server such as BIND as the root user can be dangerous--and BIND normally runs as root. If a hacker finds a vulnerability in the name server through which he can read or write files, he'll have unfettered access to the filesystem. If he can exploit a flaw that allows him to execute commands, he'll execute them as root.

BIND 8.1.2 and later include code that allows you to change the user and group the name server runs as. This allows you to run the name server with what's known as *least privilege* : the minimal set of rights it needs to do its job. That way, if someone breaks into your host through the name server, at least that person won't have root privileges.

BIND 8.1.2 and later also include an option that allows you to *chroot( )* the name server: to change its view of the filesystem so that its root directory is actually a particular directory on your host's filesystem. This effectively traps your name server in this directory, along with any attackers who successfully compromise your name server's security.

The command-line options that implement these features are:

-*u*

Specifies the username or user ID the name server changes to after starting, e.g., *named -u bin*.

-g

Specifies the group or group ID the name server changes to after starting, e.g., *named -g other*. If you specify -*u* without -*g*, the name server uses the user's primary group. BIND 9 name servers always change to the user's primary group, so they don't support -*g*.

-t

Specifies the directory for the name server to *chroot( )* to.

If you opt to use the -*u* and -*g* options, you'll have to decide what user and group to use. Your best bet is to create a new user and group for the name server to run as, such as *named*. Since the name server reads *named.conf* before giving up root privileges, you don't have to change that file's permissions. However, you may have to change the permissions and ownership of your zone data files so that the user the name server runs as can read them. If you use dynamic update, you'll have to make the zone data files for dynamically updated zones

writable by the name server.

If your name server is configured to log to files (instead of to *syslog*), make sure that those files exist and are writable by the name server before starting the server.

The *-t* option takes a little more specialized configuration. In particular, you need to make sure that all the files *named* uses are present in the directory you're restricting the server to. Here's a procedure to follow to set up your *chroot* ed environment, which we'll assume lives under */var/named* :[3]

1. Create the */var/named* directory, if it doesn't exist. Create *dev*, *etc*, *lib*, *usr*, and *var* subdirectories. Within *usr*, create an *sbin* subdirectory. Within *var*, create subdirectories named *named* and *run*:

   ```
   # mkdir /var/named
   # cd /var/named
   # mkdir -p dev etc lib usr/sbin var/named var/run
   ```

2. Copy *named.conf* to */var/named/etc/named.conf*:

   ```
   # cp /etc/named.conf etc
   ```

3. If you're running BIND 8, copy the *named-xfer* binary to the *usr/sbin/* or *etc* subdirectory (depending on whether you found it in */usr/sbin* or */etc*).

   ```
   # cp /usr/sbin/named-xfer usr/sbin
   ```

   Alternately, you can put it wherever you like under */var/named* and use the *named-xfer* substatement to tell *named* where to find it. Just remember to strip */var/named* off of the pathname, since when *named* reads *named.conf*, */var/named* will look like the root of the filesystem. (If you're running BIND 9, skip this step because BIND 9 doesn't use *named-xfer*.)

4. Create *dev/null* in the *chroot* ed environment:

   ```
   # mknod dev/null c 1 3
   ```

5. If you're running BIND 8, copy the standard, shared C library and the loader to the *lib* subdirectory:

```
# cp /lib/libc.so.6 /lib/ld-2.1.3.so lib
# ln -s lib/ld-2.1.3.so lib/ld-linux.so.2
```

The pathnames may vary on your operating system. BIND 9 name servers are self-contained.

6. Edit your startup files to start *syslogd* with an additional option and option argument: *-a /var/named/dev/log*. On many modern versions of Unix, *syslogd* is started from */etc/rc.d/init.d/syslog*. When *syslogd* restarts next, it will create */var/named/dev/log*, and *named* will log to it.

   If your *syslogd* doesn't support the *-a* option, use the *logging* statement described in *Chapter 7, Maintaining BIND*, to log to files in the *chroot* ed directory.

7. If you're running BIND 8 and use the *-u* or *-g* options, create *passwd* and *group* files in the *etc* subdirectory to map the arguments of *-u* and *-g* to their numeric values (or just use numeric values as arguments):

   ```
   # echo "named:x:42:42:named:/:" > etc/passwd
   # echo "named::42" > etc/group
   ```

   Then add the entries to the system's */etc/passwd* and */etc/group* files. If you're running BIND 9, you can *just* add the entries to the system's */etc/passwd* and */etc/group* files, since BIND 9 name servers read the information they need before calling *chroot( )*.

8. Finally, edit your startup files to start *named* with the *-t* option and option argument: *-t /var/named*. Similarly to *syslogd*, many modern versions of Unix start *named* from */etc/rc.d/init.d/named*.

If you're hooked on using *ndc* to control your BIND 8 name server, you can continue to do so as long as you specify the pathname to the Unix domain socket as the argument to *ndc* 's *-c* option:

```
# ndc -c /var/named/var/run/ndc reload
```

rndc will continue to work as before with your BIND 9 name server since it just talks to the server via port 953.

Split-Function Name Servers

Name servers really have two major roles: answering iterative queries from remote name servers and answering recursive queries from

local resolvers. If we separate these roles, dedicating one set of name servers to answering iterative queries and another to answering recursive queries, we can more effectively secure those name servers.

"Delegated" name server configuration

Some of your name servers answer nonrecursive queries from other name servers on the Internet because these name servers appear in NS records delegating your zones to them. We'll call these name servers "delegated" name servers.

There are special measures you can take to secure your delegated name servers. But first, you should make sure that these name servers don't receive any recursive queries (that is, you don't have any resolvers configured to use these servers and no name servers use them as forwarders). Some of the precautions we'll take--like making the server respond nonrecursively even to recursive queries--preclude your resolvers from using these servers. If you do have resolvers using your delegated name servers, consider establishing another class of name servers to serve just your resolvers or using the "Two Name Servers in One" configuration, both described later in this chapter.

Once you know your name server only answers queries from other name servers, you can turn off recursion. This eliminates a major vector of attack: the most common spoofing attacks involve inducing the target name server to query name servers under the hacker's control by sending the target a recursive query for a domain name in a zone served by the hacker's servers. To turn off recursion, use the following statement on a BIND 8 or 9 name server:

```
options {
     recursion no;
};
```

or, on a BIND 4.9 server, use:

```
options no-recursion
```

You should also restrict zone transfers of your zones to known slave servers, as described in the section "Preventing Unauthorized Zone Transfers" earlier in this chapter. Finally, you might also want to turn off glue fetching. Some name servers will automatically try to resolve the domain names of any name servers in NS records; to prevent this from happening and keep your name server from sending any queries of its own, use this on a BIND 8 name server (BIND 9 name servers have glue fetching turned off by default):

```
options {
      fetch-glue no;
};
```

or, on a BIND 4.9 server, use:

```
options no-fetch-glue
```

"Resolving" name server configuration

We'll call a name server that serves one or more resolvers or that is configured as another name server's forwarder a "resolving" name server. Unlike a delegated name server, a resolving name server can't refuse recursive queries. Consequently, we have to configure it a little differently to secure it. Since we know our name server should receive queries only from our own resolvers, we can configure it to deny queries from any but our resolvers' IP addresses.

Only BIND 8 and 9 allow us to restrict which IP addresses can send our name server arbitrary queries. (BIND 4.9 name servers let us restrict which IP addresses can send the server queries in authoritative zones, via the *secure_zone* TXT record, but we're actually more worried about recursive queries in others' zones.) This *allow-query* substatement restricts queries to just our internal network:

```
options {
        allow-query { 192.249.249/24; 192.253.253/24; 192.253.254/24; };
};
```

With this configuration, the only resolvers that can send our name server recursive queries and induce them to query other name servers are our own internal resolvers, which are presumably relatively benevolent.

There's one other option we can use to make our resolving name server a little more secure-- *use-id-pool*:

```
options {
        use-id-pool yes;
};
```

use-id-pool was introduced in BIND 8.2. It tells our name server to take special care to use random message IDs in queries. Normally, the message IDs aren't random enough to prevent brute-force attacks that try to guess the IDs our name server has outstanding in order to spoof a response.

The ID pool code became a standard part of BIND 9, so you don't need to specify it on a BIND 9 name server.

Two Name Servers in One

What if you have only one name server to advertise your zones and serve your resolvers, and you can't afford the additional expense of buying another computer to run a second name server on? There are still a few options open to you. Two are single-server solutions that take advantage of the flexibility of BIND 8 and 9. One of these configurations allows anyone to query the name server for information in zones it's authoritative for, but only our internal resolvers can query the name server for other information. While this doesn't prevent remote resolvers from sending our name server recursive queries, those queries have to be in its authoritative zones so they won't induce our name server to send additional queries.

Here's a *named.conf* file to do that:

```
acl "internal" {
        192.249.249/24; 192.253.253/24; 192.253.254/24;
};

acl "slaves" {
        192.249.249.1; 192.253.253.1; 192.249.249.9; 192.253.253.9;
};

options {
        directory "/var/named";
        allow-query { "internal"; };
        use-id-pool yes;
};

zone "movie.edu" {
        type master;
        file "db.movie.edu";
        allow-query { any; };
        allow-transfer { "slaves"; };
};

zone "249.249.192.in-addr.arpa" {
        type master;
        file "db.192.249.249";
        allow-query { any; };
        allow-transfer { "slaves"; };
};
```

Here, the more permissive zone-specific access control lists apply to queries in the name server's authoritative zones, but the more restrictive global access control list applies to all other queries.

If we were running BIND 8.2.1 or newer, we could simplify this configuration somewhat using the *allow-recursion* substatement:

```
acl "internal" {
        192.249.249/24; 192.253.253/24; 192.253.254/24;
};

acl "slaves" {
        192.249.249.1; 192.253.253.1; 192.249.249.9; 192.253.253.9;
};

options {
        directory "/var/named";
        allow-recursion { "internal"; };
        use-id-pool yes;
};

zone "movie.edu" {
        type master;
        file "db.movie.edu";
        allow-transfer { "slaves"; };
};

zone "249.249.192.in-addr.arpa" {
        type master;
        file "db.192.249.249";
        allow-transfer { "slaves"; };
};
```

We don't need the *allow-query* substatements anymore: although the name server may receive queries from outside our internal network, it'll treat those queries as nonrecursive, regardless of whether they are or not. Consequently, external queries won't induce our name server to send any queries. This configuration also doesn't suffer from a gotcha the previous setup is susceptible to: if your name server is authoritative for a parent zone, it may receive queries from remote name servers resolving domain names in a subdomain of the zone. The *allow-query* solution will refuse those legitimate queries, but the *allow-recursion* solution won't.

Another option is to run two *named* processes on a single host. One is configured as a delegated name server, another as a resolving name

server. Since we have no way of telling remote servers or configuring resolvers to query one of our name servers on a port other than 53, the default DNS port, we have to run these servers on different IP addresses.

Of course, if your host already has more than one network interface, that's no problem. Even if it has only one, the operating system may support IP address aliases. These allow you to attach more than one IP address to a single network interface. One *named* process can listen on each. Finally, if the operating system doesn't support IP aliases, you can still bind one *named* against the network interface's IP address and one against the loopback address. Only the local host will be able to send queries to the instance of *named* listening on the loopback address, but that's fine if the local host's resolver is the only one you need to serve.

First, here's the *named.conf* file for the delegated name server, listening on the network interface's IP address:

```
acl "slaves" {
        192.249.249.1; 192.253.253.1; 192.249.249.9; 192;253.253.9; };
};

options {
        directory "/var/named-delegated";
        recursion no;
        fetch-glue no;
        listen-on { 192.249.249.3; };
        pid-file "/var/run/named.delegated.pid";
};

zone "movie.edu" {
        type master;
        file "db.movie.edu";
        allow-transfer { "slaves"; };
};

zone "249.249.192.in-addr.arpa" {
        type master;
        file "db.192.249.249";
        allow-transfer { "slaves"; };
};

zone "." {
        type hint;
        file "db.cache";
};
```

Next, here's the *named.conf* file for the resolving name server, listening on the loopback address:

```
options {
        directory "/var/named-resolving";
        listen-on { 127.0.0.1; };
        pid-file "/var/run/named.resolving.pid";
        use-id-pool yes;
};

zone "." {
        type hint;
        file "db.cache";
};
```

Note that we didn't need an access control list for the resolving name server, since it's only listening on the loopback address and can't receive queries from other hosts. (If our resolving name server were listening on an IP alias or a second network interface, we could use *allow-query* to prevent others from using our name server.) We turn recursion off on the delegated name server, but we must leave it on on the resolving name server. We also give each name server its own PID file and its own directory so that the servers don't try to use the same default filename for their PID files, debug files, and statistics files.

To use the resolving name server listening on the loopback address, the local host's *resolv.conf* file must include the following:

```
nameserver 127.0.0.1
```

as the first *nameserver* directive.

If you're running BIND 9, you can even consolidate the two name server configurations into one using views:

```
options {
        directory "/var/named";
};

acl "internal" {
        192.249.249/24; 192.253.253/24; 192.253.254/24;
};

view "internal" {
        match-clients { "internal"; };
```

```
        recursion yes;

        zone "movie.edu" {
                type master;
                file "db.movie.edu";
        };

        zone "249.249.192.in-addr.arpa" {
                type master;
                file "db.192.249.249";
        };

        zone "." {
                type hint;
                file "db.cache";
        };
};

view "external" {
        match-clients { any; };
        recursion no;

        zone "movie.edu" {
                type master;
                file "db.movie.edu";
        };

        zone "249.249.192.in-addr.arpa" {
                type master;
                file "db.192.249.249";
        };

        zone "." {
                type hint;
                file "db.cache";
        };
};
```

It’s a fairly simple configuration: two views, internal and external. The internal view, which applies only to our internal network, has recursion on. The external view, which applies to everyone else, has recursion off. The zones *movie.edu* and *249.249.192.in-addr.arpa* are defined identically in both zones. You could do a lot more with it--define different versions of the zones internally and externally, for

example--but we'll hold off on that until the next section.

DNS and Internet Firewalls

The Domain Name System wasn't designed to work with Internet firewalls. It's a testimony to the flexibility of DNS and of its BIND implementation that you can configure DNS to work with, or even through, an Internet firewall.

That said, configuring BIND to work in a firewalled environment, although not difficult, takes a good, complete understanding of DNS and a few of BIND's more obscure features. Describing it also requires a large portion of this chapter, so here's a roadmap.

We'll start by describing the two major families of Internet firewall software--packet filters and application gateways. The capabilities of each family have a bearing on how you'll need to configure BIND to work through the firewall. Next, we'll detail the two most common DNS architectures used with firewalls, forwarders and internal roots, and describe the advantages and disadvantages of each. We'll then introduce a solution using a new feature, forward zones, which combines the best of internal roots and forwarders. Finally, we'll discuss split namespaces and the configuration of the bastion host, the host at the core of your firewall system.

Types of Firewall Software

Before you start configuring BIND to work with your firewall, it's important to understand what your firewall is capable of. Your firewall's capabilities will influence your choice of DNS architecture and determine how you implement it. If you don't know the answers to the questions in this section, track down someone in your organization who does know and ask. Better yet, work with your firewall's administrator when designing your DNS architecture to ensure it will coexist with the firewall.

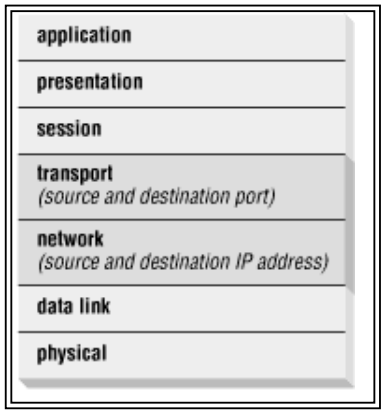
Note that this is far from a complete explanation of Internet firewalls. These few paragraphs describe only the two most common types of Internet firewalls and only in enough detail to show how the differences in their capabilities affect name servers. For a comprehensive treatment of Internet firewalls, see Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman's *Building Internet Firewalls* (O'Reilly).

Packet filters

The first type of firewall we'll cover is the packet-filtering firewall. Packet-filtering firewalls operate largely at the transport and network levels of the TCP/IP stack (layers three and four of the OSI reference model, if you dig that). They decide whether to route a packet based on packet-level criteria like the transport protocol (e.g., whether it's TCP or UDP), the source and destination IP address, and the source

and destination port (see Figure 11-1).

Figure 11-1. Packet filters operate at the network and transport layers of the stack



What's most important to us about packet-filtering firewalls is that you can typically configure them to allow DNS traffic selectively between hosts on the Internet and your internal hosts. That is, you can let an arbitrary set of internal hosts communicate with Internet name servers. Some packet-filtering firewalls can even permit your name servers to query name servers on the Internet, but not vice versa. All router-based Internet firewalls are packet-filtering firewalls. Checkpoint's FireWall-1, Cisco's PIX, and Sun's SunScreen are popular commercial packet-filtering firewalls.

A Gotcha with BIND 8 or 9 and Packet-Filtering Firewalls

BIND 4 name servers always send queries from port 53, the well-known port for DNS servers, to port 53. Resolvers, on the other hand, usually send queries from high-numbered ports (above 1023) to port 53. Though name servers clearly have to send their queries *to* the DNS port on a remote host, there's no reason they have to send the queries *from* the DNS port. And, wouldn't you know it, BIND 8 and 9 name servers don't send queries from port 53 by default. Instead, they send queries from high-numbered ports, the same as resolvers do.

This can cause problems with packet-filtering firewalls that are configured to allow name server-to-name server traffic but not resolver-to-name server traffic, because they typically expect name server-to-name server traffic to originate from port 53 and terminate at port 53.

There are two solutions to this problem:

- Reconfigure the firewall to allow your name server to send and receive queries from ports other than 53 (assuming this doesn't compromise the security of the firewall by allowing packets from Internet hosts to high-numbered ports on internal name servers).

- Configure BIND to revert to its old behavior with the *query-source* substatement.

query-source takes as arguments an address specification and an optional port number. For example, the statement:

```
options { query-source address * port 53; };
```

tells BIND to use port 53 as the source port for queries sent from all local network interfaces. You can use a nonwildcard address specification to limit the addresses that BIND will send queries from. For example, on *wormhole.movie.edu*, the statement:

```
options { query-source address 192.249.249.1 port *; };
```

tells BIND to send all queries from the 192.249.249.1 address (i.e., not from 192.253.253.1) and to use dynamic, high-numbered ports.

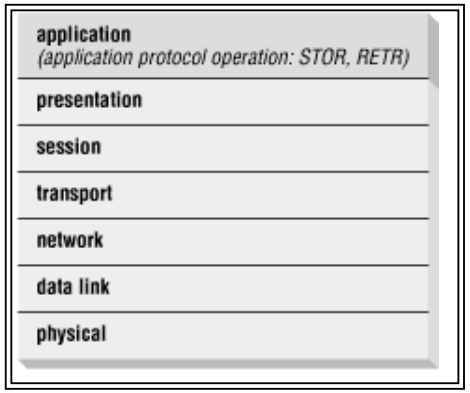
The use of *query-source* with a wildcard address is broken in BIND 9 before 9.1.0, though you can tell an early BIND 9 name server to send all queries from a particular address's port 53.

Application gateways

Application gateways operate at the application protocol level, several layers higher in the OSI reference model than most packet filters (see Figure 11-2). In a sense, they "understand" the application protocol in the same way a server for that particular application would. An FTP application gateway, for example, can make the decision to allow or deny a particular FTP operation, such as a *RETR* (a *get*) or a

STOR (a *put*).

Figure 11-2. Application gateways operate at the application layer of the stack



The bad news, and what's important for our purposes, is that most application gateway-based firewalls handle only TCP-based application protocols. DNS, of course, is largely UDP-based, and we know of no application gateways for DNS. This implies that if you run an application gateway-based firewall, your internal hosts will likely not be able to communicate directly with name servers on the Internet.

The popular Firewall Toolkit from Trusted Information Systems (TIS, now part of Network Associates) is a suite of application gateways for common Internet protocols such as Telnet, FTP, and HTTP. Network Associates' Gauntlet product is also based on application gateways, as is Axent's Eagle Firewall.

Note that these two categories of firewall are really just generalizations. The state of the art in firewalls changes very quickly, and by the time you read this, you may have a firewall that includes an application gateway for DNS. Which family your firewall falls into is important only because it suggests what that firewall is capable of; what's more important is whether your particular firewall will let you permit DNS traffic between arbitrary internal hosts and the Internet.

A Bad Example

The simplest configuration is to allow DNS traffic to pass freely through your firewall (assuming you can configure your firewall to do that). That way, any internal name server can query any name server on the Internet, and any Internet name server can query any of your internal name servers. You don't need any special configuration.

Unfortunately, this is a really bad idea, for a number of reasons:

Version control
: The developers of BIND are constantly finding and fixing security-related bugs in the BIND code. Consequently, it's important to run a recent version of BIND, especially on name servers directly exposed to the Internet. If one or just a few of your name servers communicate directly with name servers on the Internet, upgrading them to a new version is easy. If any of the name servers on your network can communicate directly with name servers on the Internet, upgrading all of them is vastly more difficult.

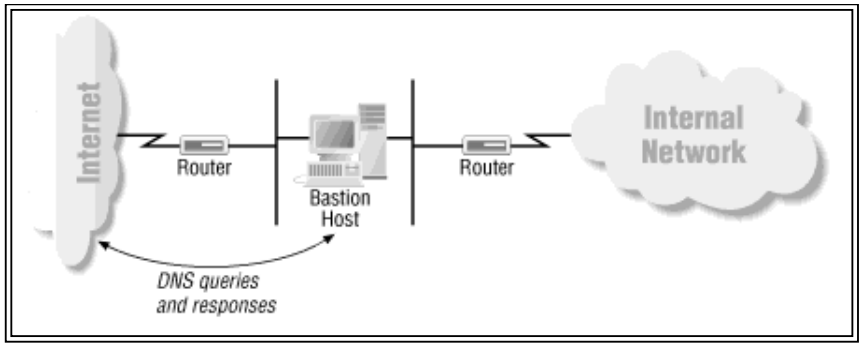
Possible vector for attack
: Even if you're not running a name server on a particular host, a hacker might be able to take advantage of your allowing DNS traffic through your firewall and attack that host. For example, a co-conspirator working on the inside could set up a Telnet daemon listening on the host's DNS port, allowing the hacker to *telnet* right in.

For the rest of this chapter, we'll try to set a good example.

Internet Forwarders

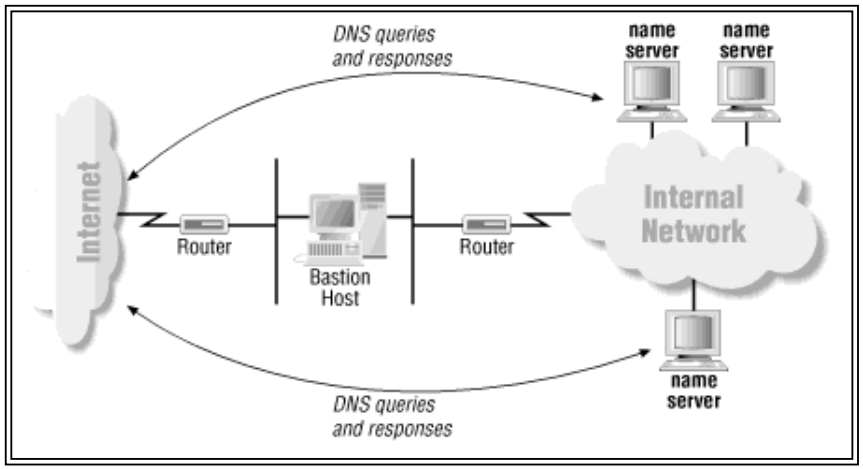
Given the dangers of allowing bidirectional DNS traffic through the firewall unrestricted, most organizations limit the internal hosts that can "talk DNS" to the Internet. In an application gateway firewall, or any firewall without the ability to pass DNS traffic, the only host that can communicate with Internet name servers is the bastion host (see Figure 11-3).

Figure 11-3. A small network, showing the bastion host



In a packet-filtering firewall, the firewall's administrator can configure the firewall to let any set of internal name servers communicate with Internet name servers. Often, this is a small set of hosts that run name servers under the direct control of the network administrator (see Figure 11-4).

Figure 11-4. A small network, showing select internal name servers

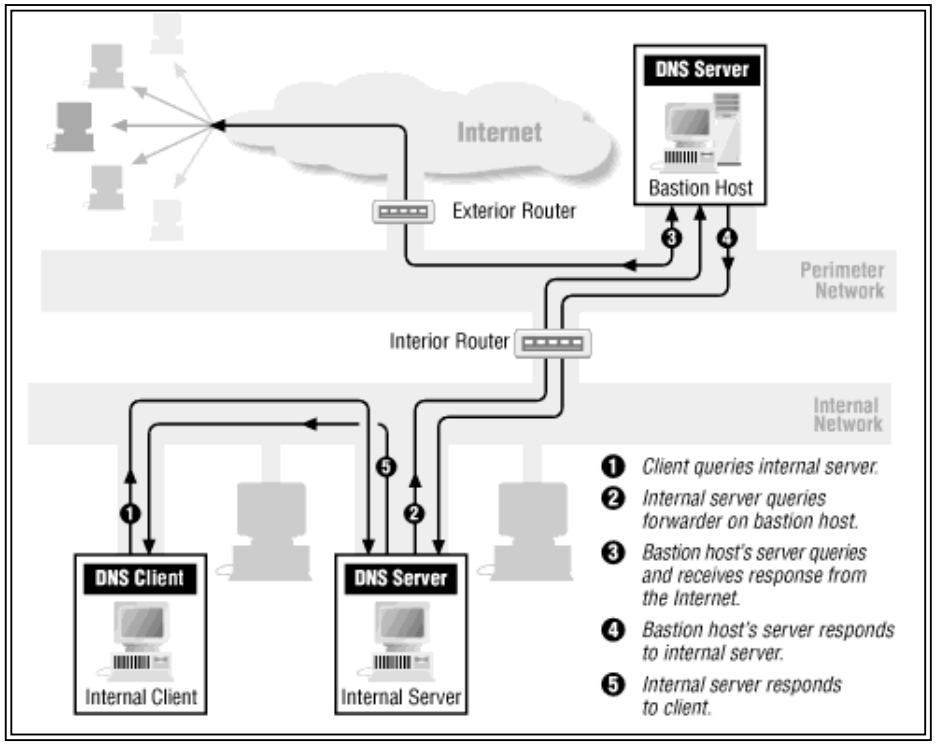


Internal name servers that can directly query name servers on the Internet don't require any special configuration. Their root hints files contain the Internet's root name servers, which enables them to resolve Internet domain names. Internal name servers that *can't* query

name servers on the Internet, however, need to know to forward queries they can't resolve to one of the name servers that can. This is done with the *forwarders* directive or substatement, introduced in *Chapter 10, Advanced Features*.

Figure 11-5 illustrates a common forwarding setup, with internal name servers forwarding queries to a name server running on a bastion host.

Figure 11-5. Using forwarders



At Movie U., we put in a firewall to protect ourselves from the Big Bad Internet several years ago. Ours is a packet-filtering firewall, and we negotiated with our firewall administrator to allow DNS traffic between Internet name servers and two of our name servers, *terminator.movie.edu* and *wormhole.movie.edu*. Here's how we configured the other internal name servers at the university. For our BIND 8 and 9 name servers, we used the following:

```
options {
        forwarders { 192.249.249.1; 192.249.249.3; };
        forward only;
};
```

and for our BIND 4 name servers, we used:

```
forwarders 192.249.249.3 192.249.249.1
options forward-only
```

We vary the order in which the forwarders appear to help spread the load between them, though that's not necessary with BIND 8.2.3 name servers, which choose which forwarder to query according to roundtrip time.

When an internal name server receives a query for a name it can't resolve locally, such as an Internet domain name, it forwards that query to one of our forwarders, which can resolve the name using name servers on the Internet. Simple!

The trouble with forwarding

Unfortunately, it's a little too simple. Forwarding starts to get in the way once you delegate subdomains or build an extensive network. To explain what we mean, take a look at part of the configuration file on *zardoz.movie.edu*:

```
options {
        directory "/var/named";
        forwarders { 192.249.249.1; 192.253.253.3; };
};

zone "movie.edu" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.movie.edu";
};
```

zardoz.movie.edu is a slave for *movie.edu* and uses our two forwarders. What happens when *zardoz.movie.edu* receives a query for a name in *fx.movie.edu* ? As an authoritative *movie.edu* name server, *zardoz.movie.edu* has the NS records that delegate *fx.movie.edu* to its authoritative name servers. But it's also been configured to forward queries it can't resolve locally to *terminator.movie.edu* and *wormhole.movie.edu*. Which will it do?

It turns out that *zardoz.movie.edu* ignores the delegation information and forwards the query to *terminator.movie.edu*. That works since *terminator.movie.edu* receives the recursive query and asks an *fx.movie.edu* name server on *zardoz.movie.edu*'s behalf. But it's not particularly efficient since *zardoz.movie.edu* could easily have sent the query directly.

Now imagine that the scale of the network is much larger: a corporate network that spans continents, with tens of thousands of hosts and hundreds or thousands of name servers. All the internal name servers that don't have direct Internet connectivity--the vast majority of them--use a small set of forwarders. What's wrong with this picture?

Single point of failure
: If the forwarders fail, your name servers lose the ability to resolve both Internet domain names and internal domain names that they don't have cached or stored as authoritative data.

Concentration of load
: The forwarders have an enormous query load placed on them. This is both because of the large number of internal name servers that use them, and because the queries are recursive and require a good deal of work to answer.

Inefficient resolution
: Imagine two internal name servers, authoritative for *west.acmebw.com* and *east.acmebw.com*, respectively, both on the same network segment in Boulder, Colorado. Both are configured to use the company's forwarder in Bethesda, Maryland. For the *west.acmebw.com* name server to resolve a name in *east.acmebw.com*, it sends a query to the forwarder in Bethesda. The forwarder in Bethesda then sends a query back to Boulder to the *east.acmebw.com* name server, the original querier's neighbor. The *east.acmebw.com* name server replies by sending a response back to Bethesda, which the forwarder sends back to Boulder.

  In a traditional configuration with root name servers, the *west.acmebw.com* name server would have learned quickly that an *east.acmebw.com* name server was next door and would favor it (because of its low roundtrip time). Using forwarders "short-circuits" the normally efficient resolution process.

The upshot is that forwarding is fine for small networks and simple namespaces, but probably inadequate for large networks and complex namespaces. We found this out the hard way at Movie U., as our network grew and we were forced to find an alternative.

Using forward zones

We can solve this problem by using the forward zones introduced in BIND 8.2. We change *zardoz.movie.edu*'s configuration to this:

```
options {
        directory "/var/named";
        forwarders { 192.249.249.1; 192.253.253.3; };
};

zone "movie.edu" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.movie.edu";
        forwarders {};
};
```

Now, if *zardoz.movie.edu* receives a query for a domain name ending in *movie.edu* but outside the *movie.edu* zone (e.g., in *fx.movie.edu*), it ignores the forwarders and sends iterative queries.

With this configuration, *zardoz.movie.edu* still sends queries for domain names in our reverse-mapping zones to our forwarders. To relieve that load, we can add a few *zone* statements to *named.conf*:

```
zone "249.249.192.in-addr.arpa" {
        type stub;
        masters { 192.249.249.3; };
        file "stub.192.249.249";
        forwarders {};
};

zone "253.253.192.in-addr.arpa" {
        type stub;
        masters { 192.249.249.3; };
        file "stub.192.253.253";
        forwarders {};
};

zone "254.253.192.in-addr.arpa" {
        type stub;
        masters { 192.253.254.2; };
        file "stub.192.253.254";
        forwarders {};
};
```

```
zone "20.254.192.in-addr.arpa" {
        type stub;
        masters { 192.253.254.2; };
        file "stub.192.254.20";
        forwarders {};
};
```

These new *zone* statements bear some explaining: first of all, they configure Movie U.'s reverse-mapping zones as stubs. That makes our name server track the NS records for those zones by periodically querying the master name servers for those zones. The *forwarders* substatement then turns off forwarding for domain names in the reverse-mapping domains. Now, instead of querying the forwarders for, say, the PTR record for *2.254.253.192.in-addr.arpa*, *zardoz.movie.edu* will query one of the *254.253.192.in-addr.arpa* name servers directly.

We'll need *zone* statements like these on all of our internal name servers, which also implies that we'll need all of our name servers to run some version of BIND 8 after 8.2.[4]

This gives us a fairly robust resolution architecture that minimizes our exposure to the Internet: it uses efficient, robust iterative name resolution to resolve internal domain names, and forwarders only when necessary to resolve Internet domain names. If our forwarders fail or we lose our connection to the Internet, we lose only our ability to resolve Internet domain names.

Internal Roots

If you want to avoid the scalability problems of forwarding, you can set up your own root name servers. These internal roots will serve only the name servers in your organization. They'll know about only the portions of the namespace relevant to your organization.

What good are they? By using an architecture based on root name servers, you gain the scalability of the Internet's namespace (which should be good enough for most companies), plus redundancy, distributed load, and efficient resolution. You can have as many internal roots as the Internet has roots--13 or so--whereas having that many forwarders may be an undue security exposure and a configuration burden. Most of all, the internal roots don't get used frivolously. Name servers need to consult an internal root only when they time out the NS records for your top-level zones. Using forwarders, name servers may have to query a forwarder once *per resolution.*

The moral of our story is that if you have, or intend to have, a large namespace and lots of internal name servers, internal root name servers will scale better than any other solution.

Where to put internal root name servers

Since name servers "lock on" to the closest root name server by favoring the one with the lowest roundtrip time, it pays to pepper your network with internal root name servers. If your organization's network spans the U.S., Europe, and the Pacific Rim, consider locating at least one internal root name server on each continent. If you have three major sites in Europe, give each of them an internal root.

Forward-mapping delegation

Here's how an internal root name server is configured. An internal root delegates directly to any zones you administer. For example, on the *movie.edu* network, the root zone's data file would contain:

```
movie.edu.  86400  IN  NS  terminator.movie.edu.
            86400  IN  NS  wormhole.movie.edu.
            86400  IN  NS  zardoz.movie.edu.
terminator.movie.edu.  86400  IN  A  192.249.249.3
wormhole.movie.edu.    86400  IN  A  192.249.249.1
                       86400  IN  A  192.253.253.1
zardoz.movie.edu.      86400  IN  A  192.249.249.9
                       86400  IN  A  192.253.253.9
```

On the Internet, this information would appear in the *edu* name servers' zone data files. On the *movie.edu* network, of course, there aren't any *edu* name servers, so you delegate directly to *movie.edu* from the root.

Notice that this doesn't contain delegation to *fx.movie.edu* or to any other subdomain of *movie.edu*. The *movie.edu* name servers know which name servers are authoritative for all *movie.edu* subdomains, and all queries for information in those subdomains pass through the *movie.edu* name servers, so there's no need to delegate them here.

in-addr.arpa delegation

We also need to delegate from the internal roots to the *in-addr.arpa* zones that correspond to the networks at the university:

```
249.249.192.in-addr.arpa.  86400  IN  NS  terminator.movie.edu.
                           86400  IN  NS  wormhole.movie.edu.
                           86400  IN  NS  zardoz.movie.edu.
253.253.192.in-addr.arpa.  86400  IN  NS  terminator.movie.edu.
                           86400  IN  NS  wormhole.movie.edu.
```

```
                            86400  IN  NS  zardoz.movie.edu.
254.253.192.in-addr.arpa.   86400  IN  NS  bladerunner.fx.movie.edu.
                            86400  IN  NS  outland.fx.movie.edu.
                            86400  IN  NS  alien.fx.movie.edu.
20.254.192.in-addr.arpa.    86400  IN  NS  bladerunner.fx.movie.edu.
                            86400  IN  NS  outland.fx.movie.edu.
                            86400  IN  NS  alien.fx.movie.edu.
```

Notice that we *did* include delegation for the *254.253.192.in-addr.arpa* and the *20.254.192.in-addr.arpa* zones, even though they correspond to the *fx.movie.edu* zone. We don't need to delegate to *fx.movie.edu* because we'd already delegated to its parent, *movie.edu.* The *movie.edu* name servers delegate to *fx.movie.edu*, so by transitivity the roots delegate to *fx.movie.edu*. Since neither of the other *in-addr.arpa* zones is a parent of *254.253.192.in-addr.arpa* or *20.254.192.in-addr.arpa*, we need to delegate both zones from the root. As we explained earlier, we don't need to add address records for the three Special Effects name servers, *bladerunner.fx.movie.edu*, *outland.fx.movie.edu*, and *alien.fx.movie.edu*, because a remote name server can already find their addresses by following delegation from *movie.edu.*

The db.root file

All that's left is to add an SOA record for the root zone and NS records for this internal root name server and any others:

```
$TTL 1d
.  IN  SOA  rainman.movie.edu.  hostmaster.movie.edu.  (
            1    ; serial
            3h   ; refresh
            1h   ; retry
            1w   ; expire
            1h ) ; negative caching TTL

   IN  NS  rainman.movie.edu.
   IN  NS  awakenings.movie.edu.

rainman.movie.edu.    IN  A  192.249.249.254
awakenings.movie.edu. IN  A  192.253.253.254
```

rainman.movie.edu and *awakenings.movie.edu* are the hosts running the internal root name servers. We shouldn't run an internal root on a bastion host, because if a name server on the Internet accidentally queries it for data it's not authoritative for, the internal root will respond with its list of roots--all internal!

So the whole *db.root* file (by convention, we call the root zone's data file *db.root*) looks like this:

```
$TTL 1d
.  IN  SOA  rainman.movie.edu.  hostmaster.movie.edu.  (
             1    ; serial
             3h   ; refresh
             1h   ; retry
             1w   ; expire
             1h ) ; negative caching TTL

   IN  NS  rainman.movie.edu.
   IN  NS  awakenings.movie.edu.

rainman.movie.edu.    IN  A  192.249.249.254
awakenings.movie.edu. IN  A  192.253.253.254

movie.edu.  IN  NS  terminator.movie.edu.
            IN  NS  wormhole.movie.edu.
            IN  NS  zardoz.movie.edu.

terminator.movie.edu.  IN  A  192.249.249.3
wormhole.movie.edu.    IN  A  192.249.249.1
                       IN  A  192.253.253.1
zardoz.movie.edu.      IN  A  192.249.249.9
                       IN  A  192.253.253.9

249.249.192.in-addr.arpa.  IN  NS  terminator.movie.edu.
                           IN  NS  wormhole.movie.edu.
                           IN  NS  zardoz.movie.edu.
253.253.192.in-addr.arpa.  IN  NS  terminator.movie.edu.
                           IN  NS  wormhole.movie.edu.
                           IN  NS  zardoz.movie.edu.
254.253.192.in-addr.arpa.  IN  NS  bladerunner.fx.movie.edu.
                           IN  NS  outland.fx.movie.edu.
                           IN  NS  alien.fx.movie.edu.
20.254.192.in-addr.arpa.   IN  NS  bladerunner.fx.movie.edu.
                           IN  NS  outland.fx.movie.edu.
                           IN  NS  alien.fx.movie.edu.
```

The *named.conf* file on both the internal root name servers, *rainman.movie.edu* and *awakenings.movie.edu*, contains the lines:

```
zone "." {
        type master;
        file "db.root";
};
```

Or, for a BIND 4 server's *named.boot* file:

```
primary    .    db.root
```

This replaces a *zone* statement of type *hint* or a *cache* directive--a root name server doesn't need a root hints file to tell it where the other roots are; it can find that in *db.root*. Did we really mean that each root name server is a primary master for the root zone? Not unless you're running an ancient version of BIND. All BIND versions after 4.9 let you declare a server as a slave for the root zone, but BIND 4.8.3 and earlier insist that all root name servers load the root zone as primaries.

If you don't have a lot of idle hosts sitting around that you can turn into internal roots, don't despair! Any internal name server (i.e., one that's not running on a bastion host or outside your firewall) can serve double duty as an internal root *and* as an authoritative name server for whatever other zones you need it to load. Remember, a single name server can be authoritative for many, many zones, including the root zone.

Configuring other internal name servers

Once you've set up internal root name servers, configure all your name servers on hosts anywhere on your internal network to use them. Any name server running on a host without direct Internet connectivity (i.e., behind the firewall) should list the internal roots in its root hints file:

```
; Internal root hints file, for Movie U. hosts without direct
; Internet connectivity
;
; Don't use this file on a host with Internet connectivity!
;

.  99999999  IN  NS  rainman.movie.edu.
   99999999  IN  NS  awakenings.movie.edu.

rainman.movie.edu.     99999999  IN  A  192.249.249.254
awakenings.movie.edu.  99999999  IN  A  192.253.253.254
```

Name servers running on hosts using this root hints file will be able to resolve domain names in *movie.edu* and in Movie U.'s *in-addr.arpa* domains, but not outside those domains.

How internal name servers use internal roots

To tie together how this whole scheme works, let's go through an example of name resolution on an internal caching-only name server using these internal root name servers. First, the internal name server receives a query for a domain name in *movie.edu*, say the address of *gump.fx.movie.edu*. If the internal name server doesn't have any "better" information cached, it starts by querying an internal root name server. If it has communicated with the internal roots before, it has a roundtrip time associated with each, telling it which of the internal roots is responding to it most quickly. It then sends a nonrecursive query to that internal root for *gump.fx.movie.edu*'s address. The internal root answers with a referral to the *movie.edu* name servers on *terminator.movie.edu*, *wormhole.movie.edu*, and *zardoz.movie.edu*. The caching-only name server follows up by sending another nonrecursive query to one of the *movie.edu* name servers for *gump.fx.movie.edu*'s address. The *movie.edu* name server responds with a referral to the *fx.movie.edu* name servers. The caching-only name server sends the same nonrecursive query for *gump.fx.movie.edu*'s address to one of the *fx.movie.edu* name servers and finally receives a response.

Contrast this with the way a forwarding setup would work. Let's imagine that instead of using internal root name servers, our caching-only name server were configured to forward queries first to *terminator.movie.edu* and then to *wormhole.movie.edu*. In that case, the caching-only name server would check its cache for the address of *gump.fx.movie.edu* and, not finding it, would forward the query to *terminator.movie.edu*. Then, *terminator.movie.edu* would query an *fx.movie.edu* name server on the caching-only name server's behalf and return the answer. Should the caching-only name server need to look up another name in *fx.movie.edu*, it would still ask the forwarder, even though the forwarder's response to the query for *gump.fx.movie.edu*'s address probably contains the names and addresses of the *fx.movie.edu* name servers.

Mail from internal hosts to the Internet

But wait! That's not all internal roots will do for you. We talked about getting mail to the Internet without changing *sendmail* 's configuration all over the network.

Wildcard records are the key to getting mail to work--specifically, wildcard MX records. Let's say that we want mail to the Internet to be forwarded through *postmanrings2x.movie.edu*, the Movie U. bastion host, which has direct Internet connectivity. Adding the following records to *db.root* will get the job done:

```
*        IN    MX    5 postmanrings2x.movie.edu.
```

```
*.edu.   IN    MX    10 postmanrings2x.movie.edu.
```

We need the **.edu* MX record in addition to the * record because of wildcard production rules, which you can read more about in the "Wildcards" section of *Chapter 16, Miscellaneous*. Basically, since there is explicit data for *movie.edu* in the zone, the first wildcard won't match *movie.edu* or any other subdomains of *edu*. We need another, explicit wildcard record for *edu* to match subdomains of *edu* besides *movie.edu.*

Now mailers on our internal *movie.edu* hosts will send mail addressed to Internet domain names to *postmanrings2x.movie.edu* for forwarding. For example, mail addressed to *nic.ddn.mil* will match the first wildcard MX record:

```
% nslookup -type=mx nic.ddn.mil. --Matches the MX record for *
Server:  rainman.movie.edu
Address:  192.249.249.19

nic.ddn.mil
     preference = 5, mail exchanger = postmanrings2x.movie.edu
postmanrings2x.movie.edu    internet address = 192.249.249.20
```

Mail addressed to *vangogh.cs.berkeley.edu* will match the second MX record:

```
% nslookup -type=mx vangogh.cs.berkeley.edu. --Matches the MX record for *.edu
Server:  rainman.movie.edu
Address:  192.249.249.19

vangogh.cs.berkeley.edu
     preference = 10, mail exchanger = postmanrings2x.movie.edu
postmanrings2x.movie.edu    internet address = 192.249.249.20
```

Once the mail has reached *postmanrings2x.movie.edu*, our bastion host, *postmanrings2x.movie.edu*'s mailer will look up the MX records for these addresses itself. Since *postmanrings2x.movie.edu* will resolve the destination's domain name using the Internet's namespace instead of the internal namespace, it will find the real MX records for the domain name and deliver the mail. No changes to *sendmail* 's configuration are necessary.

Mail to specific Internet domain names

Another nice perk of this internal root scheme is that it gives you the ability to forward mail addressed to certain Internet domain names through particular bastion hosts, if you have more than one. We can choose, for example, to send all mail addressed to recipients in the *uk*

domain to our bastion host in London first and then out onto the Internet. This can be very useful if we want our mail to travel across our own network as far as possible or if we're billed for our usage of some network in the U.K.

Movie U. has a private network connection to our sister university in London near Pinewood Studios. For security reasons, we'd like to send mail addressed to correspondents in the U.K. across our private link and then through the Pinewood host. So we add the following wildcard records to *db.root*:

```
; holygrail.movie.ac.uk is at the other end of our U.K. Internet link
*.uk.    IN    MX    10 holygrail.movie.ac.uk.
holygrail.movie.ac.uk.    IN   A    192.168.76.4
```

Now, mail addressed to users in subdomains of *uk* will be forwarded to the host *holygrail.movie.ac.uk* at our sister university, which presumably has facilities to forward that mail to other points in the U.K.

The trouble with internal roots

Unfortunately, just as forwarding has its problems, internal root architectures have their limitations. Chief among these is the fact that your internal hosts can't see the Internet namespace. On some networks, this isn't an issue because most internal hosts don't have any direct Internet connectivity. The few that do can have their resolvers configured to use a name server on the bastion host. Some of these hosts will probably need to run proxy servers to allow other internal hosts access to services on the Internet.

On other networks, however, the Internet firewall or other software may require that all internal hosts have the ability to resolve names in the Internet's namespace. For these networks, an internal root architecture won't work.

A Split Namespace

Many organizations would like to advertise different zone data to the Internet than they advertise internally. In most cases, much of the internal zone data is irrelevant to the Internet because of the organization's Internet firewall. The firewall may not allow direct access to most internal hosts, and may also translate internal, unregistered IP addresses into a range of IP addresses registered to the organization. Therefore, the organization might need to trim out irrelevant information from the external view of the zone or change internal addresses to their external equivalents.

Unfortunately, BIND doesn't support automatic filtering and translation of zone data. Consequently, many organizations manually create what have become known as "split namespaces." In a split namespace, the real namespace is available only internally, while a pared-down,

translated version of it called the *shadow namespace* is visible to the Internet.

The shadow namespace contains the name-to-address and address-to-name mappings of only those hosts accessible from the Internet through the firewall. The addresses advertised may be the translated equivalents of internal addresses. The shadow namespace may also contain one or more MX records to direct mail from the Internet through the firewall to a mail server.

Since Movie U. has an Internet firewall that greatly limits access from the Internet to the internal network, we elected to create a shadow namespace. For the zone *movie.edu*, the only information we need to give out is about the domain name *movie.edu* (an SOA record and a few NS records), the bastion host (*postmanrings2x.movie.edu*), and our new external name server, *ns.movie.edu*, which also functions as an external web server, *www.movie.edu*. The address of the external interface on the bastion host is 200.1.4.2, and the address of the name/web server is 200.1.4.3. The shadow *movie.edu* zone data file looks like this:

```
$TTL 1d
@    IN    SOA    ns.movie.edu.    hostmaster.movie.edu. (
                            1    ; Serial
                            3h   ; Refresh
                            1h   ; Retry
                            1w   ; Expire
                            1h ) ; Negative caching TTL

    IN    NS    ns.movie.edu.
    IN    NS    ns1.isp.net.        ; our ISP's name server is a movie.edu slave

    IN    A     200.1.4.3
    IN    MX    10 postmanrings2x.movie.edu.
    IN    MX    100 mail.isp.net.

www             IN    CNAME movie.edu.

postmanrings2x  IN    A     200.1.4.2
                IN    MX    10 postmanrings2x.movie.edu.
                IN    MX    100 mail.isp.net.

;postmanrings2x.movie.edu handles mail addressed to ns.movie.edu
ns              IN    A     200.1.4.3
                IN    MX    10 postmanrings2x.movie.edu.
                IN    MX    100 mail.isp.net.

*               IN    MX    10 postmanrings2x.movie.edu.
```

```
                IN    MX    100 mail.isp.net.
```

Note that there's no mention of any of the subdomains of *movie.edu*, including any delegation to the name servers for those subdomains. The information simply isn't necessary since there's nothing in any of the subdomains that you can get to from the Internet, and inbound mail addressed to hosts in the subdomains is caught by the wildcard.

The *db.200.1.4* file, which we need in order to reverse map the two Movie U. IP addresses that hosts on the Internet might see, looks like this:

```
$TTL 1d
@    IN    SOA    ns.movie.edu.    hostmaster.movie.edu. (
                           1    ; Serial
                           3h   ; Refresh
                           1h   ; Retry
                           1w   ; Expire
                           1h ) ; Negative caching TTL

    IN    NS    ns.movie.edu.
    IN    NS    ns.isp.net.

2    IN    PTR    postmanrings2x.movie.edu.
3    IN    PTR    ns1.movie.edu.
```

One precaution we have to take is to make sure that the resolver on our bastion host isn't configured to use the server on *ns.movie.edu*. Since that server can't see the real, internal *movie.edu*, using it would render *postmanrings2x.movie.edu* unable to map internal domain names to addresses or internal addresses to names.

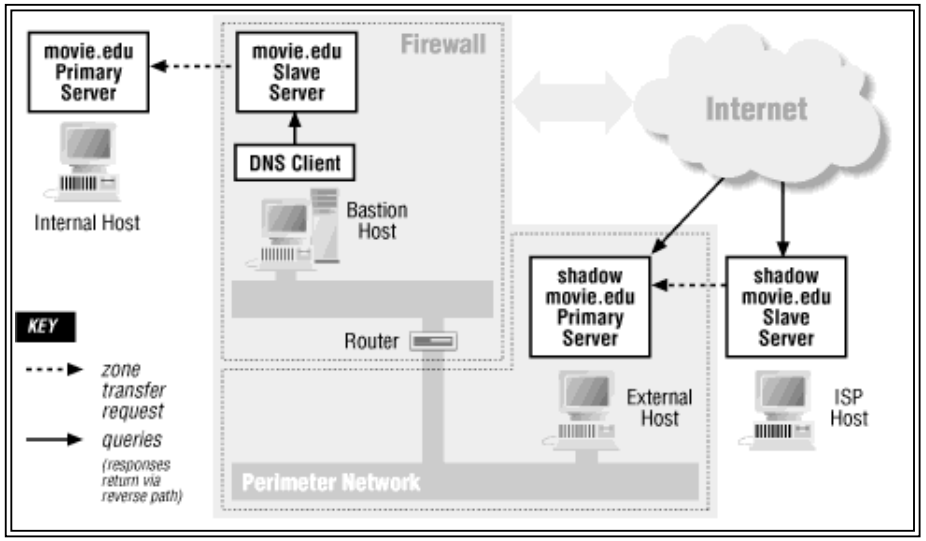
Configuring the bastion host

The bastion host is a special case in a split namespace configuration. It has a foot in each environment: one network interface connects it to the Internet and another connects it to the internal network. Now that we have split our namespace in two, how can our bastion host see both the Internet namespace and our real internal namespace? If we configure it with the Internet's root name servers in its root hints file, it will follow delegation from the Internet's *edu* name servers to an external *movie.edu* name server with shadow zone data. It would be blind to our internal namespace, which it needs to see to log connections, deliver inbound mail, and more. On the other hand, if we configure it with our internal roots, then it won't see the Internet's namespace, which it clearly needs to do in order to function as a bastion host. What to do?

If we have internal name servers that can resolve both internal and Internet domain names--using forward zones per the configuration earlier in this chapter, for example--we can simply configure the bastion host's resolver to query those name servers. But if we use forwarding internally, depending on the type of firewall we're running, we may also need to run a forwarder on the bastion host itself. If the firewall won't pass DNS traffic, we'll need to run at least a caching-only name server, configured with the Internet roots, on the bastion host so that our internal name servers will have somewhere to forward their unresolved queries.

If our internal name servers don't support forward zones, the name server on our bastion host must be configured as a slave for *movie.edu* and any *in-addr.arpa* zones in which it needs to resolve addresses. This way, if it receives a query for a domain name in *movie.edu*, it uses its local authoritative data to resolve the name. (If our internal name servers support forward zones and are configured correctly, the name server on our bastion host will never receive queries for names in *movie.edu*.) If the domain name is in a delegated subdomain of *movie.edu*, it follows NS records in the zone data to query an internal name server for the name. Therefore, it doesn't need to be configured as a slave for any *movie.edu* subdomains, such as *fx.movie.edu*, just the "topmost" zone (see Figure 11-6).

Figure 11-6. A split DNS solution



The *named.conf* file on our bastion host looks like this:

```
options {
```

```
        directory "/var/named";
};

zone "movie.edu" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.movie.edu";
};

zone "249.249.192.in-addr.arpa" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.192.249.249";
};

zone "253.253.192.in-addr.arpa" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.192.253.253";
};

zone "254.253.192.in-addr.arpa" {
        type slave;
        masters { 192.253.254.2; };
        file "bak.192.253.254";
};

zone "20.254.192.in-addr.arpa" {
        type slave;
        masters { 192.253.254.2; };
        file "bak.192.254.20";
};

zone "." {
        type hint;
        file "db.cache";
};
```

An equivalent *named.boot* file would look like this:

```
directory    /var/named
```

```
secondary    movie.edu    192.249.249.3   bak.movie.edu
secondary    249.249.192.in-addr.arpa    192.249.249.3    bak.192.249.249
secondary    253.253.192.in-addr.arpa    192.249.249.3    bak.192.253.253
secondary    254.253.192.in-addr.arpa    192.253.254.2    bak.192.253.254
secondary    20.254.192.in-addr.arpa     192.253.254.2    bak.192.254.20
cache    .    db.cache    ; lists Internet roots
```

Protecting zone data on the bastion host

Unfortunately, loading these zones on the bastion host also exposes them to the possibility of disclosure on the Internet, which we were trying to avoid by splitting the namespace in the first place. But as long as we're running BIND 4.9 or better, we can protect the zone data using the *secure_zone* TXT record or the *allow-query* substatement, both discussed earlier in the chapter. With *allow-query*, we can place a global access list on our zone data. Here's the new *options* statement from our *named.conf* file:

```
options {
        directory "/var/named";
        allow-query { 127/8; 192.249.249/24; 192.253.253/24;
                192.253.254/24; 192.254.20/24; };
};
```

With BIND 4.9's *secure_zone* feature, we can turn off all external access to our zone data by including these TXT records in each zone data file:

```
secure_zone     IN    TXT    "192.249.249.0:255.255.255.0"
                IN    TXT    "192.253.253.0:255.255.255.0"
                IN    TXT    "192.253.254.0:255.255.255.0"
                IN    TXT    "192.254.20.0:255.255.255.0"
                IN    TXT    "127.0.0.1:H"
```

Don't forget to include the loopback address in the list, or the bastion host's resolver may not get answers from its own name server!

The final configuration

Finally, we need to apply the other security precautions we discussed earlier to our bastion host's name server. In particular, we should:

- Restrict zone transfers

- Use the ID pool feature (on BIND 8.2 or newer name servers but not BIND 9)
- (Optionally) Run BIND *chroot* ed and with least privilege

In the end, our *named.conf* file ends up looking like this:

```
acl "internal" {
        127/8; 192.249.249/24; 192.253.253/24;
        192.253.254/24; 192.254.20/24;
};

options {
        directory "/var/named";
        allow-query { "internal"; };
        allow-transfer { none; };
        use-id-pool yes;
};

zone "movie.edu" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.movie.edu";
};

zone "249.249.192.in-addr.arpa" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.192.249.249";
};

zone "253.253.192.in-addr.arpa" {
        type slave;
        masters { 192.249.249.3; };
        file "bak.192.253.253";
};



zone "254.253.192.in-addr.arpa" {
        type slave;
```

```
        masters { 192.253.254.2; };
        file "bak.192.253.254";
};

zone "20.254.192.in-addr.arpa" {
        type slave;
        masters { 192.253.254.2; };
        file "bak.192.254.20";
};

zone "." {
        type hint;
        file "db.cache";
};
```

Using views on the bastion host

If we're running BIND 9 on our bastion host, we can use views to safely present the shadow *movie.edu* to the outside world on the same name server that resolves Internet domain names. That may obviate the need to run an external name server on the same host as our web server, *www.movie.edu*. If not, it'll give us two name servers to advertise the external *movie.edu*.

This configuration is very similar to one shown in the section on :

```
options {
        directory "/var/named";
};

acl "internal" {
        127/8; 192.249.249/24; 192.253.253/24; 192.253.254/24; 192.254.20/24;
};

view "internal" {
        match-clients { "internal"; };
        recursion yes;

        zone "movie.edu" {
                type slave;
                masters { 192.249.249.3; };
                file "bak.movie.edu";
         };
```

```
        zone "249.249.192.in-addr.arpa" {
                type slave;
                masters { 192.249.249.3; };
                file "bak.192.249.249";
        };

        zone "253.253.192.in-addr.arpa" {
                type slave;
                masters { 192.249.249.3; };
                file "bak.192.253.253";
        };

        zone "254.253.192.in-addr.arpa" {
                type slave;
                masters { 192.253.254.2; };
                file "bak.192.253.254";
        };

        zone "20.254.192.in-addr.arpa" {
                type slave;
                masters { 192.253.254.2; };
                file "bak.192.254.20";
        };

        zone "." {
                type hint;
                file "db.cache";
        };
};

view "external" {
        match-clients { any; };
        recursion no;

        acl "ns1.isp.net" { 199.11.28.12; };

        zone "movie.edu" {
                type master;
                file "db.movie.edu.external";
                allow-transfer { "ns1.isp.net"; };
        };
```

```
        zone "4.1.200.in-addr.arpa" {
                type master;
                file "db.200.1.4";
                allow-transfer { "ns1.isp.net"; };
        };

        zone "." {
                type hint;
                file "db.cache";
        };
};
```

Notice that the internal and external views present different versions of *movie.edu*: one loaded from the zone data file *db.movie.edu*, and one loaded from *db.movie.edu.external*. If there were more than a few zones in our external view, we probably would have used a different subdirectory for our external zone data files than we used for the internal zone data files.

The DNS Security Extensions

TSIG, which we described earlier in this chapter, is well suited to securing the communications between two name servers or between an updater and a name server. However, it won't protect you if one of your name servers is compromised: if someone breaks into the host that runs one of your name servers, he may also gain access to its TSIG keys. Moreover, because TSIG uses shared secrets, it isn't practical to configure TSIG among many name servers. You couldn't use TSIG to secure your name servers' communications with arbitrary name servers on the Internet because you can't distribute and manage that many keys.

The most common way to deal with key management problems like these is to use *public key cryptography*. The DNS Security Extensions, described in RFC 2535, use public key cryptography to enable zone administrators to digitally sign their zone data, thereby proving its authenticity.

> **TIP:** Note: we'll describe the DNS Security Extensions, or DNSSEC, in their current form as described by RFC 2535. However, the IETF's DNSEXT working group is still working on DNSSEC and may change aspects of it before it becomes a standard.
>
> Another note: though BIND 8 provided preliminary support of DNSSEC as early as BIND 8.2,[5] DNSSEC wasn't really usable before BIND 9. Consequently, we'll use BIND 9 in our examples. If you want to use DNSSEC, you really shouldn't use

anything older.

Public Key Cryptography and Digital Signatures

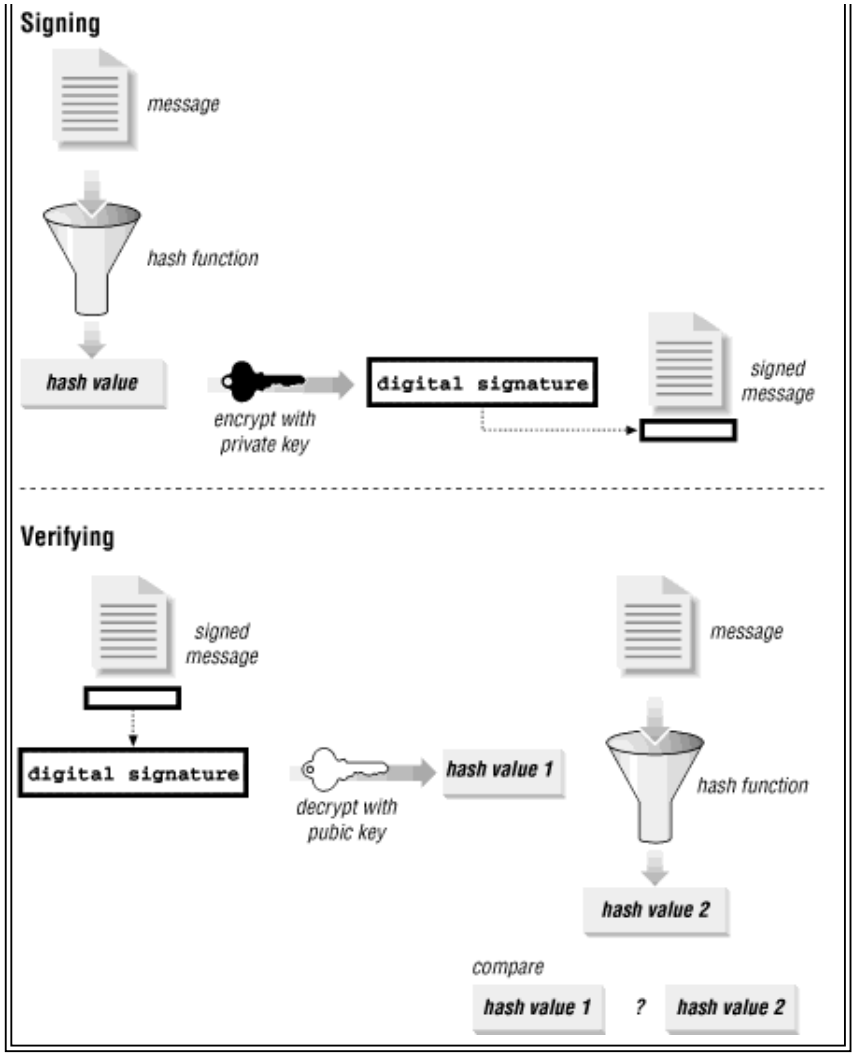
Public key cryptography solves the key distribution problem by using asymmetric cryptographic algorithms. In an asymmetric cryptographic algorithm, one key is used to decrypt data that another has encrypted. These two keys--a *key pair*--are generated at the same time using a mathematical formula. That's the only easy way to find two keys that have this special asymmetry (one decrypts what the other encrypts): it's very difficult to determine one key given the other. (In the most popular asymmetric cryptographic algorithm, RSA, that determination involves factoring very large numbers.)

In public key cryptography, an individual first generates a key pair. Then one key of the key pair is made public (e.g., published in a directory) while the other is kept private. Someone who wants to communicate securely with that individual can encrypt a message with the individual's public key and then send the encrypted message to the individual. (Or he could even post the message to a newsgroup or on a web site.) If the recipient has kept his private key private, only he can decrypt the message.

Conversely, the individual can encrypt a message with his private key and send it to someone. The recipient can verify that it came from the individual by attempting to decrypt it with the individual's public key. If the message decrypts to something reasonable (i.e., not gibberish) and the sender kept his private key to himself, then the individual must have encrypted it. Successful decryption also proves that the message wasn't modified in transit (e.g., while passing through a mail server), because if it had been, it wouldn't have decrypted correctly. So the recipient has authenticated the message.

Unfortunately, encrypting large amounts of data with asymmetric encryption algorithms tends to be slow--much slower than encryption using symmetric encryption algorithms. But when using public key encryption for authentication (and not for privacy), we don't have to encrypt the whole message. Instead, we run the message through a one-way hash function first. Then we can encrypt just the hash value, which represents the original data. We attach the encrypted hash value, now called a *digital signature*, to the message we want to authenticate. The recipient can still authenticate the message by decrypting the digital signature and running the message through her own copy of the one-way hash function. If the hash values match, the message is authentic. The process of signing and verifying a message is shown in Figure 11-7.

Figure 11-7. Signing and verifying a message



The KEY Record

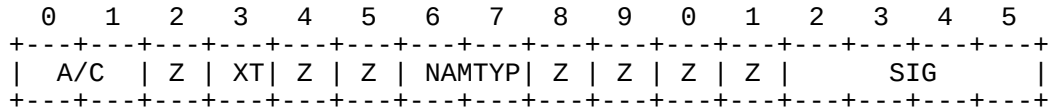
In the DNS Security Extensions, or DNSSEC, each secure zone has a key pair associated with it. The zone's private key is stored somewhere safe, often in a file on the name server's filesystem. The zone's public key is advertised as a new type of record attached to the domain name of the zone, the KEY record.

The KEY record is actually a general-purpose record, as we'll see when we dissect one. You can use the KEY record to store different kinds of cryptographic keys, not just zones' public keys for use with DNSSEC. However, the only use we're going to explore in this book is to store a zone's public key.

A KEY record looks like this:

```
movie.edu. IN KEY 256 3 1 AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX7+uBxB11Bq
L7 LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B
```

The owner is the domain name of the zone that owns this public key. The first field after the type, 256, is the flags field. The flags field is two bytes long and encodes a set of one- and two-bit values:



If the value of the first bit is zero, the key can be used for authentication. Clearly, a key that we can't use for authentication isn't very useful in DNSSEC, so that bit is always zero.

If the value of the second bit is zero, the key can be used for confidentiality. DNSSEC doesn't make your zone data private, but also doesn't prohibit you from using your zone's public key for confidentiality, so this bit is always zero for a zone's public key.

A KEY record in which the first two bits are set to one is called a *null key*. Later on, we'll show you how null keys are used in parent zones.

The third bit is reserved for future use. For now, its value must be zero. The fourth bit is a "flag extension" bit. It's designed to provide future expandability. If it's set, the KEY record must include another two-byte field after the algorithm field (the third field after the type) and before the public key itself (normally the fourth field). The meanings of the bits in that additional two-byte field haven't been defined yet, so for now the fourth bit is always zero. Like the third bit, the fifth and sixth bits are reserved and must be zero.

The seventh and eighth bits encode the type of key:

00
: This is a user's key. A mail user agent might use a user's key to encrypt email addressed to that user. This type of key isn't used in DNSSEC.

01
: This is a zone's public key. All DNSSEC keys are this type of key.

10
: This is a host's key. An IPSEC implementation might use a host's key to encrypt all IP packets sent to that host. DNSSEC doesn't use host keys.

11
: Reserved for future use.

The ninth through twelfth bits are reserved and must be zero. The last four bits are the signatory field, which is now obsolete.

In the KEY record shown earlier, the flags field (the first field in the record after the type) says that this KEY is *movie.edu*'s zone key and can be used for authentication and confidentiality.

The next field in the record, which in the example has the value 3, is called the *protocol octet*. Since you can use KEY records for different purposes, you have to specify which purpose a particular key is intended for. The following values are defined:

0
: Reserved.

1
: This key is used with Transport Layer Security (TLS), as described in RFC 2246.

2

This key is used in connection with email, e.g., an S/MIME key.

3

This key is used with DNSSEC. All DNSSEC keys, obviously, will have a protocol octet of 3.

4

This key is used with IPSEC.

255
This key is used with any protocol that can use a KEY record.

All of the values between 4 and 255 are available for future assignment.

The next (third) field in the KEY record, which here has the value 1, is the algorithm number. DNSSEC can work with a number of public key encryption algorithms, so you need to identify which algorithm a zone uses and which algorithm this key is used with here. The following values are defined:

0

Reserved.

1

RSA/MD5. RFC 2535 recommends, but doesn't require, the use of RSA/MD5. However, RSA is very popular and the patent covering the RSA algorithm recently ran out, so there has been some discussion of making the use of RSA mandatory.

2

Diffie-Hellman. RFC 2535 makes using Diffie-Hellman optional.

3

DSA. RFC 2535 makes *support* (not use) of DSA mandatory. However, as noted earlier, this may change soon.

4

Reserved for an elliptic curve-based public key algorithm.

We'll use RSA keys in our examples because we think RSA keys are likely to become the standard.

The final field in the KEY record is the public key itself, encoded in base 64. DNSSEC supports keys of many lengths, as we'll see shortly when we generate the *movie.edu* public key. The longer the key, the harder it is to find the corresponding private key, but the longer it takes to sign zone data with the private key and verify it with the public key.

Null keys don't have a public key, though they do have a protocol octet and an algorithm number.

The SIG Record

If the KEY record stores a zone's public key, then there must be a new record to store the corresponding private key's signature, right? Sure enough, that's the SIG record. The SIG record stores the private key's digital signature on an *RRset*. An RRset is a group of resource records with the same owner, class, and type; for example, all of *wormhole.movie.edu*'s address records make up an RRset. Likewise, all of *movie.edu*'s MX records are another RRset.

Why sign RRsets rather than individual records? It saves time. There's no way to look up just one of *wormhole.movie.edu*'s address records; a name server will always return them as a group. So why go to the trouble of signing each one individually when you can sign them together?

Here's the SIG record that "covers" *wormhole.movie.edu*'s address records:

```
wormhole.movie.edu.             SIG     A 1 3 86400 20010102235426 (
                                20001203235426 27791 movie.edu.
                                1S/LuuxhSHs2LknPC7K/7v4+PNxESKZnjX6CtgGLZDWf
                                Rmovkw9VpW7htTNJYhzlFck/BO/k17tRj0fbQ6JWaA== )
```

The owner name is *wormhole.movie.edu*, the same as the owner of the records signed. The first field after the type, which holds the value A, is called the *type covered*. That tells us which of *wormhole.movie.edu*'s records were signed; in this case, its address records. There would be a separate SIG record for each type of record *wormhole.movie.edu* might own.

The second field, which has the value 1, is the algorithm number. This is one of the same values used in the KEY record's algorithm number field, so 1 means RSA/MD5. If you generate an RSA key and use it to sign your zone, you'll get RSA/MD5 signatures, naturally. If you sign your zone with multiple types of keys, say an RSA key and a DSA key, you'll end up with two SIG records for each RRset, one with an algorithm number of 1 (RSA/MD5) and one with an algorithm number of 3 (DSA).[6]

The third field is called the *labels field*. It indicates how many labels there are in the owner name of the records signed. *wormhole.movie.edu* obviously has three labels, so the labels field contains 3. When would the labels field ever differ from the number of labels in the SIG's owner? When the SIG record covered a wildcard record of some type. Unfortunately (or maybe fortunately, for our sanity's sake), BIND doesn't support wildcard records in secure zones.

The fourth field is the original TTL on the records in the RRset that was signed. (All the records in an RRset are supposed to have the same TTL.) The TTL needs to be stored here because a name server caching the RRset that this SIG record covers will decrement the TTLs on the cached records. Without the original TTL, it's impossible to feed the original address records through the one-way hash function in their original state to verify the digital signature.

The next two fields are the signature expiration and inception fields, respectively. They're both stored as an unsigned integer number of seconds since the Unix epoch, January 1, 1970, but in the SIG record's text representation, they're presented in the format YYYYMMDDHHMMSS for convenience. (The signature expiration time for the SIG record we showed you earlier is just after 11:54 p.m. on January 2, 2001.) The signature inception time is usually the time you ran the program to sign your zone. You choose the signature expiration time when you run that program, too. After the signature's expiration, the SIG record is no longer valid and can't be used to verify the RRset. Bummer. This means that you have to re-sign your zone data periodically to keep the signatures valid. Fun. Thankfully, re-signing takes much less time than signing it for the first time.

The next (seventh) field in the SIG record, which in this record contains 27791, is the *key tag* field. The key tag is a fingerprint derived from the public key that corresponds to the private key that signed the zone. If the zone has more than one public key (and yours will when you're changing keys), DNSSEC verification software uses the key tag to determine which key to use to verify this signature.

The eighth field, which contains *movie.edu*, is the *signer's name* field. As you'd expect, it's the domain name of the public key that a verifier should use to check the signature. It, together with the key tag, identifies the KEY record to use. In most cases, the signer's name

field is the domain name of the zone the signed records are in. In one case, however--which we'll cover soon--the signer's name is the domain name of the parent zone.

The final field is the *signature field*. This is the digital signature of the zone's private key on the signed records and the SIG record itself, minus this field. Like the key in the KEY record, this signature is encoded in base 64.

The NXT Record

DNSSEC introduces one more new record type: the NXT record. We'll explain what it's for.

What happens if you look up a domain name that doesn't exist in a secure zone? If the zone weren't secure, the name server would simply respond with the "no such domain name" response code. But how do you sign a response code? If you signed the whole response message, it would be difficult to cache.

The NXT record solves the problem of signing negative responses. It "spans" a gap between two consecutive domain names in a zone, telling you which domain name comes next after a given domain name--hence the name of the record.

But doesn't the notion of "consecutive domain names" imply a canonical order to the domain names in a zone? Why, yes, it does.

To order the domain names in a zone, you begin by sorting by the rightmost label in those domain names, then by the next label to the left, and so on. Labels are sorted case-insensitively and lexicographically (by dictionary order), with numbers coming before letters and nonexistent labels before numbers (in other words, *movie.edu* would come before *0.movie.edu*). So the domain names in *movie.edu* would sort to the following:

```
movie.edu
bigt.movie.edu
carrie.movie.edu
cujo.movie.edu
dh.movie.edu
diehard.movie.edu
fx.movie.edu
bladerunner.fx.movie.edu
outland.fx.movie.edu
horror.movie.edu
localhost.movie.edu
```

```
misery.movie.edu
robocop.movie.edu
shining.movie.edu
terminator.movie.edu
wh.movie.edu
wh249.movie.edu
wh253.movie.edu
wormhole.movie.edu
```

Notice that just as *movie.edu* comes before *bigt.movie.edu*, *fx.movie.edu* precedes *bladerunner.fx.movie.edu*.

Once the zone is in canonical order, the NXT records make sense. Here's one NXT record (the first, in fact) from *movie.edu*:

```
movie.edu.                          NXT     bigt.movie.edu. ( NS SOA MX SIG NXT )
```

This record says that the next domain name in the zone after *movie.edu* is *bigt.movie.edu*, which we could see from our sorted list of domain names. It also says that *movie.edu* has NS records, an SOA record, MX records, a SIG record, and a NXT record.

The last NXT record in a zone is special. Since there's really no next domain name after the last one, the last NXT record "wraps around" to the first record in the zone:

```
wormhole.movie.edu.                 NXT     movie.edu. ( A SIG NXT )
```

In other words, to indicate that *wormhole.movie.edu* is the last domain name in the zone, we say that the next domain name is *movie.edu*, the first domain name in the zone.

So how do NXT records provide authenticated negative responses? Well, if you looked up *www.movie.edu* internally, you'd get back the *wormhole.movie.edu* NXT record, telling you that there's no *www.movie.edu* because there are no domain names in the zone after *wormhole.movie.edu*. Similarly, if you tried to look up TXT records for *movie.edu*, you'd get the first NXT record we showed you, which tells you there are no TXT records for *movie.edu*, just NS, SOA, MX, SIG, and NXT records.

A SIG record covering the NXT record accompanies it in the response, authenticating the nonexistence of the domain name or type of data you asked for.

It's important that the NXT records, *in toto*, identify specifically what doesn't exist in the zone. A single catch-all record that simply says "That doesn't exist" could be sniffed off the wire and replayed to claim falsely that existing domain names or records don't actually exist.

For those of you worried about the prospects of adding of these new records to your zone and keeping them up to date manually--uh-oh, now that I've added a host, I've got to adjust my NXT records--take heart: BIND provides a tool to add NXT and SIG records for you automatically.

Some of you may also worry about the information NXT records reveal about your zone. A hacker could, for example, look up the NXT record attached to the domain name of your zone to find the lexicographically next domain name, then repeat the process to learn all the domain names in the zone. That, unfortunately, is an unavoidable side effect of securing your zone. Just repeat this mantra: "My zone data is secure, but public."

The Chain of Trust

There's one more aspect of DNSSEC theory that we should discuss: the chain of trust. (No, this isn't some touchy-feely team-building exercise.) So far, each RRset in our secure zone has a SIG record associated with it. To let others verify those SIG records, our zone advertises its public key to the world in a KEY record. But imagine that someone breaks into our primary master name server. What's to keep him from generating his own key pair? Then he could modify our zone data, re-sign our zone with his newly generated private key, and advertise his newly generated public key in a KEY record.

To combat this problem, our public key is "certified" by a higher authority. This higher authority attests to the fact that the *movie.edu* public key in our KEY record really belongs to the organization that owns and runs the zone, and not to some random yahoo. Before certifying us, this higher authority demanded some sort of proof that we were who we said we were and that we were the duly authorized administrators of *movie.edu*.

This higher authority is our parent zone, *edu*. When we generated our key pair and signed our zone, we also sent our public key to the administrators of *edu*, along with proof of our identity and of our positions as the Two True Administrators of *movie.edu*.[7] They signed our KEY record with the *edu* zone's private key and sent it back to us so that we could add it to our zone. Here's our KEY record and its accompanying SIG record:

```
movie.edu               IN SIG  KEY 1 2 3600 20010104010141 (
                                20001205010141 65398 edu.
                                aE4sCZKgFtp5RuD1sib0+19dc3MF/y9S2Fr8+h66g+Y2
                                1bc31M4y0493cSoyRpapJrd7qfG+Cr7GK+uY+eLCRA== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
```

```
7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
```

Note that the signer's name field of the SIG record is *edu*, not *movie.edu*, showing that our KEY record was signed by our parent zone's private key, not our own.

What if someone were to break into the *edu* zone's primary master name server? The *edu* zone's KEY record is signed by the root zone's private key. And the root zone? Well, the root zone's public key is very widely known and configured on every name server that supports DNSSEC.[8]

That is, the root zone's public key will be configured on every name server once DNSSEC is widely implemented. Right now, neither the root zone nor the *edu* zone is signed, and neither has a key pair. Until DNSSEC is widely implemented, though, it's possible to use DNSSEC piecemeal.

Security roots

Let's say we want to begin using DNSSEC at Movie U. to improve the security of our zone data. We've signed the *movie.edu* zone but can't have *edu* sign our KEY record since they haven't secured their zone yet and don't have a key pair. How can other name servers on the Internet verify our zone data? How can our own name servers verify our zone data, for that matter?

BIND 9 name servers provide a mechanism for specifying the public key that corresponds to a particular zone in the *named.conf* file: the *trusted-keys* statement. Here's the *trusted-keys* statement for *movie.edu*:

```
trusted-keys {
        movie.edu. 256 3 1 "AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX7+uBxB11Bq
L7 LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B";
};
```

It's basically the KEY record without the class and type fields and with the key itself quoted. The domain name of the zone may be quoted, but it's not necessary. If *movie.edu* had more than one public key--say a DSA key--we could include it, too:

```
trusted-keys {
        movie.edu. 256 3 1 "AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX7+uBxB11Bq
L7 LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B";
        movie.edu. 256 3 3 "AMnD8GXACuJ5GVnfCJWmRydg2A6JptSm6tjH7QoL81SfBY/kcz1Nbe
Hh z4l9AT1GG2kAZjGLjH07BZHY+joz6iYMPRCDaPOIt9LO+SRfBNZg62P4 aSPT5zVQPahDIMZmTIvv
O7FV6IaTV+cQiKQl6noro8uTk4asCADrAHw0 iVjzjaYpoFF5AsB0cJU18fzDiCNBUb0VqE1mKFuRA/K
```

```
1KyxM2vJ3U7IS to0IgACiCfHkYK5r3qFbMvF1GrjyVwfwCC4NcMsqEXIT8IEI/YYIgFt4 Ennh";
};
```

This *trusted-keys* statement enables a BIND 9 name server to verify any records in the *movie.edu* zone. The name server can also verify any records in child zones like *fx.movie.edu*, assuming their KEY records are signed by *movie.edu*'s private key and it can verify records in *their* child zones (*movie.edu*'s grandchildren), assuming a valid chain of trust back to the *movie.edu* zone's public key. In other words, *movie.edu* becomes a *security root*, below which our name server can verify any secure zone data.

Null keys

A security root lets your name server verify signed records below a certain point in the namespace. *Null keys* do the opposite: they tell your name server that records below a certain point aren't secured. Let's say the administrators of *fx.movie.edu* haven't secured their zone yet. When we sign *movie.edu*, the BIND 9 signer software adds a special null key to the *movie.edu* zone for *fx.movie.edu*:

```
fx.movie.edu.           KEY     49408 3 3 (
                                )
```

Note that there's no base 64 encoding of a public key in the record. If you look *very* closely (or get out your scientific calculator), you'll see that the flags field indicates that this key should not be used for either authentication or confidentiality; that is, it's a null key. DNSSEC-aware name servers interpret this to mean that the *fx.movie.edu* zone isn't secure and that they shouldn't expect signed data from it.

If BIND 9's signer software finds a file containing *fx.movie.edu*'s KEY record when it runs, it omits the null key, implying that *fx.movie.edu* is secure.

How the Records Are Used

Let's go through what a DNSSEC-capable name server does to verify a record in *movie.edu*. In particular, let's see what happens when it looks up the address of *wormhole.movie.edu*. First, of course, the name server sends a query for the address:

```
% dig +dnssec +norec wormhole.movie.edu.

; <<>> DiG 9.1.0 <<>> +dnssec +norec wormhole.movie.edu.
;; global options:  printcmd
;; Got answer:
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15766
;; flags: qr aa ra; QUERY: 1, ANSWER: 3, AUTHORITY: 4, ADDITIONAL: 6

;; OPT PSEUDOSECTION:
; EDNS: version:    0, udp=   4096
;; QUESTION SECTION:
;wormhole.movie.edu.            IN      A

;; ANSWER SECTION:
wormhole.movie.edu.     86400   IN      A       192.249.249.1
wormhole.movie.edu.     86400   IN      A       192.253.253.1
wormhole.movie.edu.     86400   IN      SIG     A 1 3 86400 20010215174848 20010116174848 27791 movie.edu.
cYKQvgVksHjwGedNz72iyIpjXBhtSOeUEQA6V0b6l8asG3mpV6hzrzNf YwTpLoh9FSjSf0kUzmXkW9aYJmd5Bw==

;; AUTHORITY SECTION:
movie.edu.              86400   IN      NS      outland.fx.movie.edu.
movie.edu.              86400   IN      NS      wormhole.movie.edu.
movie.edu.              86400   IN      NS      terminator.movie.edu.
movie.edu.              86400   IN      SIG     NS 1 2 86400 20010215174848 20010116174848 27791 movie.edu.
ZXRnlbJBWJa4XX3YTWgkYnoQjGLFDN+2JwoGpLpxTidwkJ0FT+N3gMSw anSxa22b+X/7v4b99t2WMcxCtUIXvw==

;; ADDITIONAL SECTION:
outland.fx.movie.edu.   86400   IN      A       192.253.254.3
terminator.movie.edu.   86400   IN      A       192.249.249.3
terminator.movie.edu.   86400   IN      SIG     A 1 3 86400 20010215174848 20010116174848 27791 movie.edu.
GSnxseyN4w5sA2Fb9uK9zVNSRJRbbcvr0DaDRwLDO8X2m6ZBbkRssSHJ tZYwoO4ZIFERLKakB//VTDMhYJmNvw==
movie.edu.              86400   IN      KEY     256 3 1
AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX7+uBxB11BqL7 LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B
movie.edu.              86400   IN      SIG     KEY 1 2 86400 20010215174837 20010116174837 65398 edu. LW+nc2gmz618u
LjDtlKSorv9OkJOwC8wj/sa/CpzCJJqceB/55JhsWI t1ADlfQwb4h9hs6oMeN2sU9jHiYQmw==

;; Query time: 3 msec
;; SERVER: 206.168.194.122#53(206.168.194.122)
;; WHEN: Tue Jan 16 10:49:48 2001
;; MSG SIZE  rcvd: 671
```

Notice that we had to specify *+dnssec* on the command line. BIND 9.1.0 and later name servers include DNSSEC records (SIG, NXT, and KEY) in a response only if the querier indicates that it can handle DNSSEC. How do queriers do that? By setting a special flag in a "pseudosection" of the header. It’s called a pseudosection because it’s not actually part of the bits in the header. Instead, it’s a new record type, OPT, that’s carried in the DNS message. OPT records usually indicate the capabilities of the querier.

Also notice that the response includes four SIG records: one covering the records in the answer section, one covering the records in the authority section, one covering *terminator.movie.edu*'s address record in the additional section, and one covering *movie.edu*'s KEY record in the additional section. The additional section would have included a SIG record covering *outland.fx.movie.edu*, address records for *wormhole.movie.edu*, and a SIG record covering those addresses if they'd fit in a single UDP datagram, but there wasn't enough room.

To verify the SIG records, the name server must look at *movie.edu*'s KEY record, included in the additional section. But before using the key, it must verify the SIG on that key. That requires at least one additional query: to one of the *edu* name servers for the *edu* zone's public key--unless, of course, the name server already knows the *movie.edu* public key from a *trusted-keys* statement.

DNSSEC and Performance

It should be evident from this *dig* output that DNSSEC increases the average size of a DNS message, that it requires substantially more computational horsepower from name servers verifying zone data, and that signing a zone increases its size substantially--current estimates are that signing multiplies the size of a zone by a factor of seven. Each of these effects has its own consequences, some of which are less obvious:

- Larger DNS messages means more truncated messages, which means more fallback to the use of TCP. Using TCP, of course, is more resource-intensive than using UDP.
- Verifying zone data takes time and slows the resolution process.
- Larger zones mean larger and harder-to-administer *named* processes.

In fact, DNSSEC's complexity meant that BIND 8's architecture couldn't support DNSSEC. DNSSEC also provided part of the impetus for developing BIND 9 and for designing it to run on multiprocessor hosts. If you're planning on signing your zones, make sure your authoritative name servers have enough memory to load the new, larger zones. If your name servers are resolving more records in secure zones, make sure they have enough processor power to verify all those digital signatures--and remember that BIND 9 can take advantage of any processors you can add to the host it runs on.

Signing a Zone

Okay, now you've got the theoretical background you need to actually sign your zone. We'll show you how we signed *movie.edu*. Remember, we used the BIND 9 tools--they're much easier to use than the BIND 8 tools, and of course BIND 9 supports DNSSEC much

more completely than BIND 8 does.

Generating your key pair

First, we generated a key pair for *movie.edu*:

```
# cd /var/named
# dnssec-keygen -a RSA -b 512 -n ZONE movie.edu.
Kmovie.edu.+001+27791
```

We ran *dnssec-keygen* in our name server's working directory. That's mostly for convenience: the zone data files are in this directory, so we won't need to use full pathnames as arguments. If we want to use dynamic update with DNSSEC, though, we'd need the keys in the name server's working directory.

Recall *dnssec-keygen*'s options from the TSIG section of this chapter (oh, so long ago):

-a

The cryptographic algorithm to use, in this case RSA. We could also have used DSA, but RSA is more efficient.

-b

The length of the keys to generate, in bits. RSA keys can be anywhere from 512 to 2000 bits long. DSA keys can be 512 to 1024 bits long, as long as the length is divisible by 64.

-n

The type of key. DNSSEC keys are always zone keys.

The only non-option argument is the domain name of the zone, *movie.edu*. The *dnssec-keygen* program prints the basename of the files it's written the keys to. The numbers at the end of the basename (001 and 27791), as we explained in the TSIG section, are the key's DNSSEC algorithm number as used in the KEY record (001 is RSA/MD5), and the key's fingerprint, used to distinguish one key from another when multiple keys are associated with the same zone.

The public key is written to the file *basename.key* (*Kmovie.edu.+001+27791.key*). The private key is written to the file *basename.private* (*Kmovie.edu.+001+27791.private).* Remember to protect the private key; anyone who knows the private key can forge signed zone data. *dnssec-keygen* does what it can to help you: it makes the *.private* file readable and writable only by the user who runs it.

Sending your keys to be signed

Next, we sent our KEY record to the administrator of our parent zone to sign. BIND 9 includes a nice little program to package up the key for transmission, *dnssec-makekeyset*:

```
# dnssec-makekeyset -t 172800 Kmovie.edu.+001+27791.key
```

dnssec-makekeyset created a file called *keyset-movie.edu,[9]* which contains the following:

```
$ORIGIN .
$TTL 172800      ; 2 days
movie.edu               IN SIG  KEY 1 2 86400 20010104034839 (
                                20001205034839 27791 movie.edu.
                                M7RDKMyc9w1djDc0mQAXQc1PJdmLRBg3nfaGEUZe9Fbi
                                mjiNVaQK33IWhzI95oD8AS0WqRDy5TusTXt4nx1/dQ== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
```

The *-t* option takes a TTL for the records to submit. This serves as a suggestion to your parent zone's administrator of the TTL (in seconds) you'd like for your record. They may ignore it, of course. The SIG record actually contains a signature covering your zone's KEY record, generated with your zone's private key. That proves you really have the private key that corresponds to the public key in the KEY record--you're not just submitting a KEY record you found on the street.

The signature expiration and inception fields default to "now" and "30 days from now," respectively. These serve as a suggestion to the signer of the signature lifetime you'd like. You can use the *-s* (start) and *-e* (end) options to adjust the signature expiration and inception times. Both options accept either an absolute time as an argument, in the form YYYYMMDDHHMMSS, or an offset. For *-s*, the offset is calculated from the current time. For *-e*, the offset is calculated from the start time.

You can also use the signature expiration and inception fields to bundle up several keysets and submit them all at once to your parent zone's administrator to sign. For example, you could submit keysets valid for January, February, and March to your parent zone's

administrator all at once, and then put them into production one per month.

Then we sent our file off to our parent zone's administrators to sign. Since the message included proof of our identity,[10] they signed it with the *dnssec-signkey* program:

```
# dnssec-signkey keyset-movie.edu Kedu.+001+65398.private
```

and sent the resulting file, *movie.edu.signedkey*, back to us:

```
$ORIGIN .
$TTL 172800       ; 1 hour
movie.edu               IN SIG  KEY 1 2 3600 20010104010141 (
                                20001205010141 65398 edu.
                                aE4sCZKgFtp5RuD1sib0+19dc3MF/y9S2Fr8+h66g+Y2
                                1bc31M4y0493cSoyRpapJrd7qfG+Cr7GK+uY+eLCRA== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
```

If we didn't care about getting our KEY record signed, we could have skipped this step. But then only name servers with a *trusted-keys* entry for *movie.edu* could verify our data.

Signing your zone

Before signing our zone, we had to add the KEY record to our plain-Jane zone data file:

```
# cat "$INCLUDE Kmovie.edu.+001+27791.key" >> db.movie.edu
```

That gave the signer program the information it needed to know which key to use to sign the zone. It automatically finds and includes the contents of *movie.edu.signedkey*.

Then we signed the zone with *dnssec-signzone* :

```
# dnssec-signzone -o movie.edu. db.movie.edu
```

We used the *-o* option to specify the origin in the zone data file, because *dnssec-signzone* doesn't read *named.conf* to determine which

zone the file describes. The only non-option argument is the name of the zone data file.

This produces a new zone data file, *db.movie.edu.signed*, which begins like this:

```
$ORIGIN .
$TTL 86400      ; 1 day
movie.edu               IN SOA  terminator.movie.edu. al.robocop.movie.edu. (
                                2000092603 ; serial
                                10800      ; refresh (3 hours)
                                3600       ; retry (1 hour)
                                604800     ; expire (1 week)
                                3600       ; minimum (1 hour)
                                )
                        SIG     SOA 1 2 86400 20010104041530 (
                                20001205041530 27791 movie.edu.
                                aO1eZKhGSm99GgC9PfLXfHjl3tAWN/Vn33msppmyhN7a
                                RlfvJMTpSoJ9XwQCdjqhz0lcnCnQiL+jZkqU3uUecg== )
                        NS      terminator.movie.edu.
                        NS      wormhole.movie.edu.
                        NS      outland.fx.movie.edu.
                        SIG     NS 1 2 86400 20010104041530 (
                                20001205041530 27791 movie.edu.
                                pkmZJHqFlNmZdNjyupBMMzDDGweGsf9TS1EGci9cwKe5
                                c0o9h/yncInn2e8QSakjxpwKB8aw9D9uiStxJ/sLvQ== )
                        SIG     MX 1 2 86400 20010104041530 (
                                20001205041530 27791 movie.edu.
                                ZcKKeT0XaNlw83eSzRxt74DaLXvQtPYCdgKGOfSiJmYQ
                                WxI5zZUEWA6ku3w48mo9jbVF+/7nF3QcpFTIiwV1ug== )
$TTL 3600       ; 1 hour
                        SIG     NXT 1 2 3600 20010104041530 (
                                20001205041530 27791 movie.edu.
                                upMjK21eD7OQkrHpxSWqkOPcRXbfL8WAgQVK1aGHcTPE
                                X3JtaLtCLuKLd3YFs7T8BuZoN7aJYRVREWSPVedYPw== )
                        NXT     bigt.movie.edu. ( NS SOA MX SIG KEY NXT )
$TTL 172800     ; 2 days
                        SIG     KEY 1 2 172800 20001205040220 (
                                20001205040219 65398 edu.
                                HIReZ98rieIuRI04XsoL+xLRLe8tCQbNKD8USlV35vb4
                                VsLUGCAEgBq7lLsHty7YCskbxhQu8ncysBKnR/muiA== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
```

```
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
$TTL 86400      ; 1 day
                        MX      10 postmanrings2x.movie.edu.
```

Believe it or not, those are just the records attached to the domain name *movie.edu*. The zone data file as a whole nearly quadrupled in length and quintupled in size. Oy!

Finally, we changed the *zone* statement in *named.conf* so that *named* would load the new zone data file:

```
zone "movie.edu" {
        type master;
        file "db.movie.edu.signed";
};
```

Then we reloaded the zone and checked *syslog*.

dnssec-signzone does take some options that we didn't use:

-s , *-e*
: These options specify the signature inception and expiration times to use in SIG records; they have exactly the same syntax as in *dnssec-makekeyset*.

-i
: Specifies as an option argument the cycle period for resigning records (which we'll cover in a minute). This was the *-c* option before BIND 9.1.0.

-f
: Specifies as an option argument the name of the file to write the signed zone to. The default is the name of the zone data file with *.signed* concatenated.

You can also specify, as a second non-option argument, which private key to use to sign the zone. By default, *dnssec-signzone* signs the zone with each of the zone's private keys in the directory. If you specify the name of one or more files that contain the zone's private keys as arguments, it will sign using only those keys.

Remember, you'll need to re-sign the zone each time you change the zone data, though you certainly don't need to generate a new key pair or have your KEY record re-signed each time. You can re-sign the zone by running *dnssec-signzone* on the signed zone data:

```
# dnssec-signzone -o movie.edu -f db.movie.edu.signed.new db.movie.edu.signed
# mv db.movie.edu.signed db.movie.edu.signed.bak
# mv db.movie.edu.signed.new db.movie.edu.signed
# rndc reload movie.edu
```

The program is smart enough to recalculate NXT records, sign new records, and re-sign records whose signature expiration times are approaching. By default, *dnssec-signzone* re-signs records whose signatures expire within 7.5 days (a quarter of the difference between the default signature inception and expiration times). If you specify different inception and expiration times, *dnssec-signzone* adjusts the re-signing cycle time accordingly. Or you can simply specify a cycle time with the *-i* (formerly the *-c*) option.

DNSSEC and Dynamic Update

dnssec-signzone isn't the only way to sign zone data. The BIND 9 name server is capable of signing dynamically updated records on the fly.[11] Color us impressed!

As long as the private key for a secure zone is available in the name server's working directory (in the correctly named *.private* file), a BIND 9 name server signs any records that are added via dynamic update. If any records are added to or deleted from the zone, the name server adjusts (and re-signs) the neighboring NXT records, too.

Let's show you this in action. First, we'll look up a domain name that doesn't yet exist in *movie.edu*:

```
% dig +dnssec perfectstorm.movie.edu.

; <<>> DiG 9.1.0 <<>> +dnssec perfectstorm.movie.edu.
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 4705
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 4, ADDITIONAL: 3

;; OPT PSEUDOSECTION:
; EDNS: version:    0, udp=   4096
;; QUESTION SECTION:
;perfectstorm.movie.edu.               IN      A
```

```
;; AUTHORITY SECTION:
movie.edu.              3600    IN      SOA     terminator.movie.edu. al.robocop.movie.edu. 2001011600 10800 3600 60
movie.edu.              3600    IN      SIG     SOA 1 2 86400 20010215174848 20010116174848 27791 movie.edu.
Ea0+xyEsj0Hy4JP115r0D0UFVpWfxqf0NQA8hpKwlLCsxJ3rA+sJBg2Q ZiCTEwfAcwGRfbNsRYu/CcuV/VJTDA==
misery.movie.edu.       86400   IN      NXT     robocop.movie.edu. A SIG NXT
misery.movie.edu.       86400   IN      SIG     NXT 1 3 86400 20010215174848 20010116174848 27791 movie.edu.
ZVfV9KbPb8hKZdZirlpv+WnUxv72di8lUgZiot/JaWDsZPfNoYqSnKPW ND4H92guwj7oR6CgrhsgLJ9dMDYSpg==
```

(We trimmed the output a little.) Notice *misery.movie.edu*'s NXT record, indicating that the domain name doesn't exist. Now we'll use *nsupdate* to add an address record for *perfectstorm.movie.edu*:

```
% nsupdate
> update add perfectstorm.movie.edu. 3600 IN A 192.249.249.91
>
```

Now, let's look up *perfectstorm.movie.edu* again:

```
% dig +dnssec perfectstorm.movie.edu.

; <<>> DiG 9.1.0 <<>> +dnssec perfectstorm.movie.edu.
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 11973
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 2, AUTHORITY: 4, ADDITIONAL: 9

;; OPT PSEUDOSECTION:
; EDNS: version:    0, udp=   4096
;; QUESTION SECTION:
;perfectstorm.movie.edu.                IN      A

;; ANSWER SECTION:
perfectstorm.movie.edu. 3600    IN      A       192.249.249.91
perfectstorm.movie.edu. 3600    IN      SIG     A 1 3 3600 20010215195456 20010116185456 27791 movie.edu. C/
JXdCLUdugxN91v0DZuUDTusi2XNNttb4bdB2nBujLxjwwPAf/D5MJz //cDtuZ3X+uYzhkN8MDROqOwUQuQSA==

;; AUTHORITY SECTION:
movie.edu.              86400   IN      NS      terminator.movie.edu.
movie.edu.              86400   IN      NS      outland.fx.movie.edu.
movie.edu.              86400   IN      NS      wormhole.movie.edu.
movie.edu.              86400   IN      SIG     NS 1 2 86400 20010215195301 20010116195301 27791 movie.edu.
```

```
1ZR592izM1AjMusJ26e4lvQ0V9lFiFvQh6hCluBxSv7FwNqF7TcJFImc W52XhXbHUEtiFOzDqYMHOzPV7j23nA==
```

(Again, we trimmed the output a little.) Now not only was an address record generated, but there is also a SIG record generated from *movie.edu*'s private key. The signature expiration is set to 30 days from the update by default, but you can change it with the *sig-validity-interval* substatement, which takes a number of days as an argument:[12]

```
options {
        sig-validity-interval 7;  // We want SIG on updated records to last a week
};
```

The signature inception is always set to one hour before the update to allow for verifiers with clocks that may be slightly skewed from ours.

If we look up *perfectstorm2.movie.edu* (though how there'd be a sequel to *that* movie I don't know), we find the following:

```
% dig +dnssec perfectstorm2.movie.edu.

; <<>> DiG 9.1.0 <<>> +dnssec perfectstorm2.movie.edu.
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 11232
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 0, AUTHORITY: 4, ADDITIONAL: 3

;; OPT PSEUDOSECTION:
; EDNS: version:    0, udp=   4096
;; QUESTION SECTION:
;perfectstorm2.movie.edu.       IN      A

;; AUTHORITY SECTION:
movie.edu.              3600    IN      SOA     terminator.movie.edu. al.robocop.movie.edu. 2001011601 10800 3600 60
movie.edu.              3600    IN      SIG     SOA 1 2 86400 20010215195456 20010116185456 27791 movie.edu.
c1RwtgBX2SO8Q7Hz7vJD0aJfNfA6lsrqH4txHJI/slRpx/UFYbnz3Gje N0JspZEihdLw0ZYMEiN6hnwRAzB4ag==
perfectstorm.movie.edu. 3600    IN      NXT     robocop.movie.edu. A SIG NXT
perfectstorm.movie.edu. 3600    IN      SIG     NXT 1 3 3600 20010215195456 20010116185456 27791 movie.edu.
qsB9l5AmSrB+qKmv+cKa+htCw84zwaakTmPC2yl+shzSEparrKwIMSR6 x5N69w8cze/AW+gyFIwQZZkfZInJZA==
```

Notice the NXT record: it was added automatically when we added *perfectstorm.movie.edu*'s address record because *perfectstorm.movie.edu* was a new domain name in the zone. Sweet!

As impressive as this is, you should be careful when allowing dynamic updates to secure zones. You should make sure that you use strong authentication (e.g., TSIG) to authenticate the updates, or you'll give a hacker an easy backdoor to use to modify your "secure" zone. And you should ensure you have enough horsepower for the task: normally, dynamic updates don't take much to process. But dynamic updates to a secure zone require NXT recalculation and, more significantly, asymmetric encryption (to calculate new SIG records), so you should expect your name server to take longer and need more resources to process them.

Changing Keys

Though we said you don't need to generate a new key each time you sign your zone, there are occasions when you'll need to create a new key, either because you've "used up" your private key or, worse, because your private key has been cracked.

After a certain amount of use, it becomes dangerous to continue signing records with your private key. The larger the pool of available data that's been encrypted using your private key, the easier it is for a hacker to determine your private key using cryptanalysis. While there's no simple rule to tell you when your private key's time is up, here are some guidelines:

- The larger your zone is, the more data your private key encrypts when it signs it. If your zone is very large, change keys more frequently.
- The longer your key, the harder it is to crack. Long keys don't need to be changed as often as short keys.
- The more frequently you update and sign your zone data, the more data encrypted by your private key is available. If you update your zone data frequently--and particularly if you dynamically update your secure zone--change keys frequently.
- The more valuable it would be for a hacker to spoof your zone data, the more time and money he will spend trying to crack your private key. If the integrity of your zone data is particularly crucial, change keys frequently.

Since we update *movie.edu* only about once a day and it's not particularly large, we change our key pair every six months. We're only a university, after all. If we were more concerned about our zone data, we would use longer keys or change keys more frequently.

Unfortunately, rolling over to a new key isn't as easy as just generating a new key and replacing the old one with it. If you did that, you'd leave name servers that had cached your zone's data with no way to retrieve your zone's KEY record and verify that data. So rolling over to a new key is a multistep process:

1. Generate a new key pair.
2. Make a new keyset that includes both your new KEY record and your old KEY record, and send it to your parent zone's administrator.
3. Make a keyset that includes just your new KEY record, and send that to your parent zone's administrator, too.
4. If you use *trusted-keys*, add an entry to the statement for your new KEY record. And tell others who use *trusted-keys*, too.
5. Incorporate the signed keyset from your parent zone's administrator that includes both KEY records.
6. Sign your zone data with the new private key, but leave the old KEY record in the zone.
7. After all records signed with the old private key have expired, remove the old KEY record from the zone.
8. Incorporate the signed keyset from your parent zone's administrator that includes just the new KEY record.
9. Sign your zone data with the new private key.

Let's go through the process. First, we generate a new key pair:

```
# dnssec-keygen -a RSA -b 512 -n ZONE movie.edu.
Kmovie.edu.+001+47703
```

Next, we make a new keyset that contains both KEY records and send it to our parent zone's administrator:

```
# dnssec-makekeyset -t 172800 Kmovie.edu.+001+27791.key Kmovie.edu.+001+47703.key
# mail -s "Sign my keys, please" hostmaster@nsiregistry.net < keyset-movie.edu
# mv keyset-movie.edu keyset-movie.edu.2key
```

Then we make a keyset that contains just the new KEY record and send it to our parent zone's administrator, too:

```
% dnssec-makekeyset -t 172800 Kmovie.edu.+001+47703.key
% mail -s "Sign my keys, please" hostmaster@nsiregistry.net < keyset-movie.edu
```

(Okay, it would take a lot more than just those messages to get anyone to sign our keys.)

The first keyset includes both KEY records and SIG records covering both:

```
$ORIGIN .
$TTL 172800     ; 2 days
movie.edu               IN SIG  KEY 1 2 172800 20010104060917 (
                                20001205060917 27791 movie.edu.
                                RyNYoZ/k0tHqnFhUiVs2yjJWPFNeP8BKZ/Jaw+7xO9Jl
                                ZwJN2ZYQjVNVGLk30rJlxQRjCCdaaYQSq8u81up3xw== )
                        IN SIG  KEY 1 2 172800 20010104060917 (
                                20001205060917 47703 movie.edu.
                                1JGNBQydq6U+qKfq1wxfu1nsu283Zf7mNDDmuBtuuB7o
                                lwaeBL96tzBKpMUAcDYXsM8zxiStF+wTY+I5wfgevA== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
                        KEY     256 3 1 (
                                AQPjAfGtDkx6PSgYFs6G2vY1bJEldAMudngn6ZpGJnyg
                                k+LeU5PYiYd48wFLimihjyzlTWMb+C+egtQGpDVQulez )
```

Note that one of the SIG records was generated by the key with key tag 27791 (the old private key) while the other was generated by the key with tag 47703 (the new private key). This proves we have both of the corresponding private keys.

Once we get a response back from our parent zone's administrator, we save it to */var/named* as *movie.edu.signedkey*, the filename we $INCLUDEd into *db.movie.edu.signed*. Here's what *movie.edu.signedkey* looks like:

```
$ORIGIN .
$TTL 172800     ; 2 days
movie.edu               IN SIG  KEY 1 2 172800 20010104060917 (
                                20001205060917 65398 edu.
                                qzvmuTVv9yGZf963ZuN2jxk8brEX/VP3sI5pOM/g2mU/
                                EPa57fyhHDNo7ny8Q2Su5vXnAIoxaaKAR8VmognQ7A== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
                        KEY     256 3 1 (
                                AQPjAfGtDkx6PSgYFs6G2vY1bJEldAMudngn6ZpGJnyg
                                k+LeU5PYiYd48wFLimihjyzlTWMb+C+egtQGpDVQulez )
```

edu's SIG record covers both KEY records, so we can use either or both to sign our zone data.

Then we sign our zone data using *only* the new private key:

```
# dnssec-signzone -o movie.edu. db.movie.edu Kmovie.edu.+001+47703.private
```

dnssec-signzone doesn't like re-signing zones that were signed with another key, so we started over from an unsigned version of the *movie.edu* zone. Here's a snippet from the signed zone data file, *db.movie.edu.signed*:

```
$ORIGIN .
$TTL 86400      ; 1 day
movie.edu               IN SOA  terminator.movie.edu. al.robocop.movie.edu. (
                                2000092603 ; serial
                                10800      ; refresh (3 hours)
                                3600       ; retry (1 hour)
                                604800     ; expire (1 week)
                                3600       ; minimum (1 hour)
                                )
                        SIG     SOA 1 2 86400 20010104062430 (
                                20001205062430 47703 movie.edu.
                                LIsndGD5q2VPWb+Ha0ffFP54UE6RYPweqtTp1xhgw4B9
                                Pyb/7z54J8q8LC0NmzQ6SthnfecBQhDBpc72HfNeJQ== )
                        NS      terminator.movie.edu.
                        NS      wormhole.movie.edu.
                        NS      outland.fx.movie.edu.
                        SIG     NS 1 2 86400 20010104062430 (
                                20001205062430 47703 movie.edu.
                                Ktq2mYMzTrBfGjdSb2F7ghyh2nXaLc0iTPV4k8I64jl0
                                nJt/hsBZPpeyM2u+Zymvp3mJMWg66E4tirj0AvlGXw== )
                        SIG     MX 1 2 86400 20010104062430 (
                                20001205062430 47703 movie.edu.
                                20001205062430 47703 movie.edu.
                                l/XnJ+JWhmAZLp6YF27YQQl0yT7iZ0qGDXPw860P6U1H
                                NmgDkUKoHfD6CdYwpKz15NyxRKilVmx2ne3oB0TUEQ== )
$TTL 3600       ; 1 hour
                        SIG     NXT 1 2 3600 20010104062430 (
                                20001205062430 47703 movie.edu.
                                2sxN3rQXn/JklugmyGV+on1Io6tV1wEYP6m4oD1xHCP1
                                +NHPR+uT2IknW8SvGc3Kaj16kb2Ej+i3RvleWSI4Tg== )
                        NXT     bigt.movie.edu. ( NS SOA MX SIG KEY NXT )
```

```
$TTL 172800     ; 2 days
                        SIG     KEY 1 2 172800 20010104060917 (
                                20001205060917 65398 edu.
                                qzvmuTVv9yGZf963ZuN2jxk8brEX/VP3sI5pOM/g2mU/
                                EPa57fyhHDNo7ny8Q2Su5vXnAIoxaaKAR8VmognQ7A== )
                        KEY     256 3 1 (
                                AQPdWbrGbVv1eDhNgRhpJMPonJfA3reyEo82ekwRnjbX
                                7+uBxB11BqL7LAB7/C+eb0vCtI53FwMhkkNkTmA6bI8B )
                        KEY     256 3 1 (
                                AQPjAfGtDkx6PSgYFs6G2vY1bJEldAMudngn6ZpGJnyg
                                k+LeU5PYiYd48wFLimihjyzlTWMb+C+egtQGpDVQulez )
$TTL 86400      ; 1 day
                        MX      10 postmanrings2x.movie.edu.
```

Although the zone includes two KEY records and *edu*'s SIG record, which covers both, the other records in the zone were signed only by the new private key, with key tag 47703.

We need the second signed keyset when we delete the old KEY record: at that point, the SIG record *edu* sent us that covers both KEYs is no good. If we use the keyset that includes just the new KEY record, we'll be fine.

We're guessing that after reading that, you'll probably decide to use the longest keys available just to avoid ever needing to roll your keys over.

What Was That All About?

We realize that DNSSEC is a bit, er, daunting. (We nearly fainted the first time we saw it.) But it's designed to do something very important: make DNS spoofing much, much harder. And as people do more and more business over the Internet, knowing you're really getting where you thought you were going becomes crucial.

That said, we realize that DNSSEC and the other security measures we've described in this chapter aren't for all of you. (Certainly they're not *all* for all of you.) You should balance your need for security against the cost of implementing it, in terms of the burden it places both on your infrastructure and on your productivity.

---

1. Cryptography wonks may argue that TSIG "signatures" aren't really signatures in a cryptographic sense because they don't provide

nonrepudiation. Since either holder of the shared key can create a signed message, the recipient of a signed message can't claim that only the sender could have sent it (the recipient could have forged it himself).

2. See the Time Synchronization Server web site at http://www.eecis.udel.edu/~ntp for information on NTP.

3. This procedure is based on Red Hat Linux 6.2, so if you use a different operating system, your mileage may vary.

4. As we mentioned in the last chapter, BIND 9 doesn't support forward zones until BIND 9.1.0.

5. In particular, BIND 8 can't follow a chain of trust. It can verify SIG records only in zones it has *trusted-keys* statements for.

6. You might sign your zone with two different algorithms' keys so that people whose software supported only DSA could verify your data while people who preferred RSA could use RSA.

7. In fact, there aren't any top-level zones signing their child zone's KEY records yet, though some European registries will likely begin signing KEY records soon.

8. This reminds us of the tale of the man who asks the priest what holds the Earth up. The priest tells him that the Earth rests on the back of a turtle, which holds it up. The man then asks what the turtle rests on. "On the back of an elephant," replies the priest. "But what," the man asks, "does the elephant rest on?" The frustrated priest snaps back, "It's elephants all the way down!"

9. In versions of *dnssec-makekeyset* shipped with BIND 9.0.1 and earlier, the name of the file would have been *movie.edu.keyset*. However, putting the domain name of the zone first caused problems: the name of the root zone's keyset file was *.keyset*, which is a hidden file on a Unix filesystem.

10. Since top-level zones haven't started signing zones yet, there's still some question as to how they'll require us to authenticate ourselves. The use of cryptographically signed email messages is a possibility.

11. Yet another DNSSEC capability BIND 8 doesn't have.

12. Before BIND 9.1.0, *sig-validity-interval* interpreted its argument as seconds, not days.

Back to: DNS and BIND, 4th Edition